GBASE

GBase 8s SQL 参考手册

www.gbase.cn

GBase 8s SQL 参考手册，南大通用数据技术股份有限公司

通讯方式

南大通用数据技术股份有限公司

天津市高新区华苑产业园区工华道 2 号天百中心 3 层(300384)

电话：400-013-9696　　邮箱：info@gbase.cn

目　录

1 SQL

SQL 是用于访问和处理数据库的标准计算机语言。SQL 提供了各种任务的语句，包括：

- 查询数据。
- 在表中插入、更新和删除行。
- 创建、替换、更改和删除对象。
- 控制对数据库及其对象的访问。
- 保证数据库的一致性和完整性。

SQL 语言由用于处理数据库和数据库对象的命令和函数组成。该语言还会强制实施有关数据类型、表达式和文本使用的规则。因此本文包含 SQL 语法参考，以及有关数据类型、表达式、函数和操作符等信息。

1.1 SQL 发展简史

SQL 发展简史如下：

1986 年，ANSI X3.135-1986，ISO/IEC 9075:1986，SQL-86

1989 年，ANSI X3.135-1989，ISO/IEC 9075:1989，SQL-89

1992 年，ANSI X3.135-1992，ISO/IEC 9075:1992，SQL-92（SQL2）

1999 年，ISO/IEC 9075:1999，SQL:1999（SQL3）

2003 年，ISO/IEC 9075:2003，SQL:2003（SQL4）

2011 年，ISO/IEC 9075:200N，SQL:2011（SQL5）

1.2 GBase 8s 支持的 SQL 标准

GBase 8s 默认支持 SQL2、SQL3 和 SQL4 的主要特性。

2 SQL 语法格式说明

表 2-1 SQL 语法格式说明

格式	意义
[ ]	表示用“[ ]”括起来的部分是可选的。
...	表示前面的元素可重复出现。
[x | y | ...]	表示从两个或多个选项中选取一个或者不选。
{ x | y | ... }	表示从两个或多个选项中选取一个。
[x | y | ...] [...]	表示可选多个参数或者不选，如果选择多个参数，则参数之间用空格分隔。
[x | y | ...] [,...]	表示可选多个参数或者不选，如果选择多个参数，则参数之间用逗号分隔。
{ x | y | ... } [...]	表示可选多个参数，至少选一个，如果选择多个参数，则参数之间以空格 分隔。
{ x | y | ... } [,...]	表示可选多个参数，至少选一个，如果选择多个参数，则参数之间用逗号 分隔。

3 SQL 语言结构和语法

3.1 关键字

SQL 里有保留字和非保留字之分。根据标准，保留字决不能用做其他标识符。非保留字只是在特定的环境里有特殊的含义，而在其他环境里是可以用做标识符的。

标识符的命名需要遵守如下规范：

- 标识符需要为字母、下划线、数字（0-9）或美元符号（$）。
- 标识符必须以字母（a-z）或下划线（_）开头。

说明

- 此命名规范为建议项，非强制项。
- 特殊情况下可以使用双引号规避特殊字符报错。

表 3-1 关键字说明

字母顺序	关键字	GBase 8s	SQL:1999	SQL-92
A	ABORT	非保留	——	——
	ABS	——	非保留	——
	ABSOLUTE	非保留	保留	保留
	ACCESS	非保留	——	——
	ACCOUNT	非保留	——	——
	ACTION	非保留	保留	保留
	ADA	——	非保留	非保留
	ADD	非保留	保留	保留
	ADMIN	非保留	保留	——

	AFTER	非保留	保留	——
	AGGREGATE	非保留	保留	——
	ALGORITHM	非保留	——	——
	ALIAS	——	保留	——
	ALL	保留	保留	保留
	ALLOCATE	——	保留	保留
	ALSO	非保留	——	——
	ALTER	非保留	保留	保留
	ALWAYS	非保留	——	——
	ANALYSE	保留	——	——
	ANALYZE	保留	——	——
	AND	保留	保留	保留
	ANY	保留	保留	保留
	APP	非保留	——	——
	APPEND	非保留	——	——
	ARCHIVE	非保留	——	——
	ARE	——	保留	保留
	ARRAY	保留	保留	——
	AS	保留	保留	保留
	ASC	保留	保留	保留

	ASENSITIVE	——	非保留	——
	ASSERTION	非保留	保留	保留
	ASSIGNMENT	非保留	非保留	——
	ASYMMETRIC	保留	非保留	——
	AT	非保留	保留	保留
	ATOMIC	——	非保留	——
	ATTRIBUTE	非保留	——	——
	AUDIT	非保留	——	——
	AUTHID	保留	——	——
	AUTHORIZATION	保留（可为函数或类型）	保留	保留
	AUTOEXTEND	非保留	——	——
	AUTOMAPPED	非保留	——	——
	AVG	——	非保留	保留
B	BACKWARD	非保留	——	——
	BARRIER	非保留	——	——
	BEFORE	非保留	保留	——
	BEGIN	非保留	保留	保留
	BEGIN_NON_ANOYBLOCK	非保留	——	——

	BETWEEN	非保留（不能是函数或类型）	非保留	保留
	BIGINT	非保留（不能是函数或类型）	——	——
	BINARY	保留（可为函数或类型）	保留	——
	BINARY_DOUBLE	非保留（不能是函数或类型）	——	——
	BINARY_INTEGER	非保留（不能是函数或类型）	——	——
	BIT	非保留（不能是函数或类型）	保留	保留
	BITVAR	——	非保留	——
	BIT_LENGTH	——	非保留	保留
	BLANKS	非保留	——	——
	BLOB	非保留	保留	——
	BLOCKCHAIN	非保留	——	——
	BODY	非保留	——	——
	BOOLEAN	非保留（不	保留	——

		能是函数或类型)		
	BOTH	保留	保留	保留
	BUCKETCNT	非保留（不能是函数或类型)	——	——
	BUCKETS	保留	——	——
	BREADTH	——	保留	——
	BY	非保留	保留	保留
	BYTEAWITHOUTORDER	非保留（不能是函数或类型)	——	——
	BYTEAWITHOUTORDERW ITHEQUAL	非保留（不能是函数或类型)	——	——
C	C	——	非保留	非保留
	CACHE	非保留	——	——
	CALL	非保留	保留	——
	CALLED	非保留	非保留	——
	CANCELABLE	非保留	——	——
	CARDINALITY	——	非保留	——
	CASCADE	非保留	保留	保留
	CASCADED	非保留	保留	保留

	CASE	保留	保留	保留
	CAST	保留	保留	保留
	CATALOG	非保留	保留	保留
	CATALOG_NAME	——	非保留	非保留
	CHAIN	非保留	非保留	——
	CHAR	非保留（不能是函数或类型）	保留	保留
	CHARACTER	非保留（不能是函数或类型）	保留	保留
	CHARACTERISTICS	非保留	——	——
	CHARACTERSET	非保留	——	——
	CHARACTER_LENGTH	——	非保留	保留
	CHARACTER_SET_CATALOG	——	非保留	非保留
	CHARACTER_SET_NAME	——	非保留	非保留
	CHARACTER_SET_SCHEMA	——	非保留	非保留
	CHAR_LENGTH	——	非保留	保留
	CHECK	保留	保留	保留
	CHECKED	——	非保留	——

	CHECKPOINT	非保留	——	——
	CLASS	非保留	保留	——
	CLEAN	非保留	——	——
	CLASS_ORIGIN	——	非保留	非保留
	CLIENT	非保留	——	——
	CLIENT_MASTER_KEY	非保留	——	——
	CLIENT_MASTER_KEYS	非保留	——	——
	CLOB	非保留	保留	——
	CLOSE	非保留	保留	保留
	CLUSTER	非保留	——	——
	COALESCE	非保留（不能是函数或类型）	非保留	保留
	COBOL	——	非保留	非保留
	COLLATE	保留	保留	保留
	COLLATION	保留（可为函数或类型）	保留	保留
	COLLATION_CATALOG	——	非保留	非保留
	COLLATION_NAME	——	非保留	非保留
	COLLATION_SCHEMA	——	非保留	非保留

	COLUMN	保留	保留	保留
	COLUMN_ENCRYPTION_KEY	非保留	——	——
	COLUMN_ENCRYPTION_KEYS	非保留	——	——
	COLUMN_NAME	——	非保留	非保留
	COMPACT	保留（可为函数或类型）	——	——
	COMPATIBLE_ILLEGAL_C HARS	非保留	——	——
	command_FUNCTION	——	非保留	非保留
	COMPLETE	非保留	——	——
	command_FUNCTION_CODE	——	非保留	——
	COMMENT	非保留	——	——
	COMMENTS	非保留	——	——
	COMMIT	非保留	保留	保留
	COMMITTED	非保留	非保留	非保留
	COMPRESS	非保留	——	——
	COMPLETION	——	保留	——
	CONCURRENTLY	保留（可为函数或类	——	——

		型)		
	CONDITION	非保留	——	——
	CONDITION_NUMBER	——	非保留	非保留
	CONFIGURATION	非保留	——	——
	CONNECT	非保留	保留	保留
	CONNECTION	非保留	保留	保留
	CONNECTION_NAME	——	非保留	非保留
	CONSTANT	非保留	——	——
	CONSTRAINT	保留	保留	保留
	CONSTRAINTS	非保留	保留	保留
	CONSTRAINT_CATALOG	——	非保留	非保留
	CONSTRAINT_NAME	——	非保留	非保留
	CONSTRAINT_SCHEMA	——	非保留	非保留
	CONSTRUCTOR	——	保留	——
	CONTAINS	——	非保留	——
	CONTENT	非保留	——	——
	CONTINUE	非保留	保留	保留
	CONTVIEW	非保留	——	——
	CONVERSION	非保留	——	——

	CONVERT	——	非保留	保留
	COORDINATOR	非保留	——	——
	COORDINATORS	非保留	——	——
	COPY	非保留	——	——
	CORRESPONDING	——	保留	保留
	COST	非保留	——	——
	COUNT	——	非保留	保留
	CREATE	保留	保留	保留
	CROSS	保留（可为函数或类型）	保留	保留
	CSN	保留（可为函数或类型）	——	——
	CSV	非保留	——	——
	CUBE	非保留	保留	——
	CURRENT	非保留	保留	保留
	CURRENT_CATALOG	保留	——	——
	CURRENT_DATE	保留	保留	保留
	CURRENT_PATH	——	保留	——
	CURRENT_ROLE	保留	保留	——
	CURRENT_SCHEMA	保留（可为	——	——

		函数或类型)		
	CURRENT_TIME	保留	保留	保留
	CURRENT_TIMESTAMP	保留	保留	保留
	CURRENT_USER	保留	保留	保留
	CURSOR	非保留	保留	保留
	CURSOR_NAME	——	非保留	非保留
	CYCLE	非保留	保留	——
D	DATA	非保留	保留	非保留
	DATABASE	非保留	——	——
	DATAFILE	非保留	——	——
	DATANODE	非保留	——	——
	DATANODES	非保留	——	——
	DATE_FORMAT	非保留	——	——
	DATATYPE_CL	非保留	——	——
	DATE	非保留（不能是函数或类型)	保留	保留
	DELTAMERGE	保留（可为函数或类型)	——	——
	DATETIME_INTERVAL_CO DE	——	非保留	非保留

	DATETIME_INTERVAL_PR ECISION	——	非保留	非保留
	DAY	非保留	保留	保留
	DBCOMPATIBILITY	非保留	——	——
	DEALLOCATE	非保留	保留	保留
	DEC	非保留（不能是函数或类型）	保留	保留
	DECIMAL	非保留（不能是函数或类型）	保留	保留
	DECLARE	非保留	保留	保留
	DECODE	非保留（不能是函数或类型）	——	——
	DEFAULT	保留	保留	保留
	DEFAULTS	非保留	——	——
	DEFERRABLE	保留	保留	保留
	DEFERRED	非保留	保留	保留
	DEFINED	——	非保留	——
	DEFINER	非保留	非保留	——
	DELETE	非保留	保留	保留
	DELIMITER	非保留	——	——

	DELIMITERS	非保留	——	——
	DELTA	非保留	——	——
	DEPTH	——	保留	——
	DEREF	——	保留	——
	DESC	保留	保留	保留
	DESCRIBE	——	保留	保留
	DESCRIPTOR	——	保留	保留
	DESTROY	——	保留	——
	DESTRUCTOR	——	保留	——
	DETERMINISTIC	非保留	保留	——
	DIAGNOSTICS	——	保留	保留
	DICTIONARY	非保留	保留	——
	DIRECT	非保留	——	——
	DIRECTORY	非保留	——	——
	DISABLE	非保留	——	——
	DISCARD	非保留	——	——
	DISCONNECT	非保留	保留	保留
	DISPATCH	——	非保留	——
	DISTINCT	保留	保留	保留
	DISTRIBUTE	非保留	——	——

	DISTRIBUTION	非保留	——	——
	DO	保留	——	——
	DOCUMENT	非保留	——	——
	DOMAIN	非保留	保留	保留
	DOUBLE	非保留	保留	保留
	DROP	非保留	保留	保留
	DUPLICATE	非保留	——	——
	DYNAMIC	——	保留	——
	DYNAMIC_FUNCTION	——	非保留	非保留
	DYNAMIC_FUNCTION_C ODE	——	非保留	——
E	EACH	非保留	保留	——
	ELSE	保留	保留	保留
	ELASTIC	非保留	——	——
	ENABLE	非保留	——	——
	ENCLOSED	非保留	——	——
	ENCODING	非保留	——	——
	ENCRYPTED	非保留	——	——
	ENCRYPTED_VALUE	非保留	——	——
	ENCRYPTION	非保留	——	——
	ENCRYPTION_TYPE	非保留	——	——

	END	保留	保留	保留
	END－－EXEC	－－	保留	保留
	ENFORCED	非保留	－－	－－
	ENUM	非保留	－－	－－
	EOL	非保留	－－	－－
	ERRORS	非保留	－－	－－
	EQUALS	－－	保留	－－
	ESCAPE	非保留	保留	保留
	ESCAPING	非保留	－－	－－
	EVERY	非保留	保留	－－
	EXCEPT	保留	保留	保留
	EXCEPTION	－－	保留	保留
	EXCHANGE	非保留	－－	－－
	EXCLUDE	非保留	－－	－－
	EXCLUDED	保留	－－	－－
	EXCLUDING	非保留	－－	－－
	EXCLUSIVE	非保留	－－	－－
	EXEC	－－	保留	保留
	EXECUTE	非保留	保留	保留
	EXISTING	－－	非保留	－－

	EXISTS	非保留（不能是函数或类型）	非保留	保留
	EXPIRED_P	非保留	——	——
	EXPLAIN	非保留	——	——
	EXTENSION	非保留	——	——
	EXTERNAL	非保留	保留	保留
	EXTRACT	非保留（不能是函数或类型）	非保留	保留
F	FALSE	保留	保留	保留
	FAMILY	非保留	——	——
	FAST	非保留	——	——
	FEATURES	非保留	——	——
	FETCH	保留	保留	保留
	FENCED	保留	——	——
	FIELDS	非保留	——	——
	FILEHEADER	非保留	——	——
	FILLER	非保留	——	——
	FILTER	非保留	保留	保留
	FINAL	——	非保留	——
	FIRST	非保留	保留	保留

	FIXED	非保留	保留	保留
	FILL_MISSING_FIELDS	非保留	——	——
	FLOAT	非保留（不能是函数或类型）	保留	保留
	FOLLOWING	非保留	——	——
	FOR	保留	保留	保留
	FORCE	非保留	——	——
	FOREIGN	保留	保留	保留
	FORMATTER	非保留	——	——
	FORTRAN	——	非保留	非保留
	FORWARD	非保留	——	——
	FOUND	——	保留	保留
	FREE	——	保留	——
	FREEZE	保留（可为函数或类型）	——	——
	FROM	保留	保留	保留
	FULL	保留（可为函数或类型）	保留	保留
	FUNCTION	非保留	保留	——
	FUNCTIONS	非保留	——	——

G	G	——	非保留	——
	GENERAL	——	保留	——
	GENERATED	非保留	非保留	——
	GET	——	保留	保留
	GLOBAL	非保留	保留	保留
	GO	——	保留	保留
	GOTO	——	保留	保留
	GRANT	保留	保留	保留
	GRANTED	非保留	非保留	——
	GREATEST	非保留（不能是函数或类型）	——	——
	GROUP	保留	保留	保留
	GROUPING	非保留（不能是函数或类型）	保留	——
	GROUPPARENT	保留	——	——
H	HANDLER	非保留	——	——
	HAVING	保留	保留	保留
	HDFSDIRECTORY	保留（可为函数或类型）	——	——
	HEADER	非保留	——	——

	HIERARCHY	——	非保留	——
	HOLD	非保留	非保留	——
	HOST	——	保留	——
	HOUR	非保留	保留	保留
I	IDENTIFIED	非保留	——	——
	IDENTITY	非保留	保留	保留
	IF	非保留	——	——
	IGNORE	——	保留	——
	IGNORE_EXTRA_DATA	非保留	——	——
	ILIKE	保留（可为函数或类型）	——	——
	IMMEDIATE	非保留	保留	保留
	IMMUTABLE	非保留	——	——
	IMPLEMENTATION	——	非保留	——
	IMPLICIT	非保留	——	——
	IN	保留	保留	保留
	INTERNAL	非保留	——	——
	INCLUDE	非保留	——	——
	INCLUDING	非保留	——	——
	INCREMENT	非保留	——	——

	INCREMENTAL	非保留	——	——
	INDEX	非保留	——	——
	INDEXES	非保留	——	——
	INDICATOR	——	保留	保留
	INFILE	非保留	——	——
	INFIX	——	非保留	——
	INHERIT	非保留	——	——
	INHERITS	非保留	——	——
	INITIAL	非保留	——	——
	INITIALIZE	——	保留	——
	INITIALLY	保留	保留	保留
	INITRANS	非保留	——	——
	INLINE	非保留	——	——
	INNER	保留（可为函数或类型）	保留	保留
	INOUT	非保留（不能是函数或类型）	保留	——
	INPUT	非保留	保留	保留
	INSENSITIVE	非保留	非保留	保留
	INSERT	非保留	保留	保留

	INSTANCE	——	非保留	——
	INSTANTIABLE	——	非保留	——
	INSTEAD	非保留	——	——
	INT	非保留（不能是函数或类型）	保留	保留
	INTEGER	非保留（不能是函数或类型）	保留	保留
	INTERSECT	保留	保留	保留
	INTERVAL	非保留（不能是函数或类型）	保留	保留
	INTO	保留	保留	保留
	INVOKER	非保留	非保留	——
	IP	非保留	——	——
	IS	保留	保留	保留
	ISNULL	非保留	——	——
	ISOLATION	非保留	保留	保留
	ITERATE	——	保留	——
J	JOIN	保留（可为函数或类型）	保留	保留

K	K	——	非保留	——
	KEY	非保留	保留	保留
	KEY_PATH	非保留	——	——
	KEY_MEMBER	——	非保留	——
	KEY_STORE	非保留	——	——
	KEY_TYPE	——	非保留	——
	KILL	非保留	——	——
L	LABEL	非保留	——	——
	LANGUAGE	非保留	保留	保留
	LARGE	非保留	保留	——
	LAST	非保留	保留	保留
	LATERAL	——	保留	——
	LC_COLLATE	非保留	——	——
	LC_CTYPE	非保留	——	——
	LEADING	保留	保留	保留
	LEAKPROOF	非保留	——	——
	LEAST	非保留（不能是函数或类型）	——	——
	LEFT	保留（可为函数或类型）	保留	保留

	LENGTH	——	非保留	非保留
	LESS	保留	保留	——
	LEVEL	非保留	保留	保留
	LIKE	保留（可为函数或类型）	保留	保留
	LIMIT	保留	保留	——
	LIST	非保留	——	——
	LISTEN	非保留	——	——
	LOAD	非保留	——	——
	LOCAL	非保留	保留	保留
	LOCALTIME	保留	保留	——
	LOCALTIMESTAMP	保留	保留	——
	LOCATION	非保留	——	——
	LOCATOR	——	保留	——
	LOCK	非保留	——	——
	LOG	非保留	——	——
	LOGGING	非保留	——	——
	LOGIN_ANY	非保留	——	——
	LOGIN_FAILURE	非保留	——	——
	LOGIN_SUCCESS	非保留	——	——

	LOGOUT	非保留	——	——
	LOOP	非保留	——	——
	LOWER	——	非保留	保留
M	MAP	——	保留	——
	MAPPING	非保留	——	——
	MASKING	非保留	——	——
	MASTER	非保留	——	——
	MATCH	非保留	保留	保留
	MATCHED	非保留	——	——
	MATERIALIZED	非保留	——	——
	MAX	——	非保留	保留
	MAXEXTENTS	非保留	——	——
	MAXSIZE	非保留	——	——
	MAXTRANS	非保留	——	——
	MAXVALUE	保留	——	——
	MERGE	非保留	——	——
	MESSAGE_LENGTH	——	非保留	非保留
	MESSAGE_OCTET_LEN GT H	——	非保留	非保留
	MESSAGE_TEXT	——	非保留	非保留
	METHOD	——	非保留	——

	MIN	——	非保留	保留
	MINEXTENTS	非保留	——	——
	MINUS	保留	——	——
	MINUTE	非保留	保留	保留
	MINVALUE	非保留	——	——
	MOD	——	非保留	——
	MODE	非保留	——	——
	MODEL	非保留	——	——
	MODIFIES	——	保留	——
	MODIFY	保留	保留	——
	MODULE	——	保留	保留
	MONTH	非保留	保留	保留
	MORE	——	非保留	非保留
	MOVE	非保留	——	——
	MOVEMENT	非保留	——	——
	MUMPS	——	非保留	非保留
N	NAME	非保留	非保留	非保留
	NAMES	非保留	保留	保留
	NATIONAL	非保留（不能是函数或类型）	保留	保留

	NATURAL	保留（可为函数或类型）	保留	保留
	NCHAR	非保留（不能是函数或类型）	保留	保留
	NCLOB	——	保留	——
	NEW	——	保留	——
	NEXT	非保留	保留	保留
	NO	非保留	保留	保留
	NOCOMPRESS	非保留	——	——
	NOCYCLE	非保留	——	——
	NODE	非保留	——	——
	NOLOGGING	非保留	——	——
	NOMAXVALUE	非保留	——	——
	NOMINVALUE	非保留	——	——
	NONE	非保留（不能是函数或类型）	保留	——
	NOT	保留	保留	保留
	NOTHING	非保留	——	——
	NOTIFY	非保留	——	——
	NOTNULL	保留（可为	——	——

		函数或类型)		
	NOWAIT	非保留	——	——
	NULL	保留	保留	保留
	NULLABLE	——	非保留	非保留
	NULLCOLS	非保留	——	——
	NULLIF	非保留（不能是函数或类型)	非保留	保留
	NULLS	非保留	——	——
	NUMBER	非保留（不能是函数或类型)	非保留	非保留
	NUMERIC	非保留（不能是函数或类型)	保留	保留
	NUMSTR	非保留	——	——
	NVARCHAR	非保留（不能是函数或类型)	——	——
	NVARCHAR2	非保留（不能是函数或类型)	——	——
	NVL	非保留（不能是函数或	——	——

		类型)		
O	OBJECT	非保留	保留	——
	OCTET_LENGTH	——	非保留	保留
	OF	非保留	保留	保留
	OFF	非保留	保留	——
	OFFSET	保留	——	——
	OIDS	非保留	——	——
	OLD	——	保留	——
	ON	保留	保留	保留
	ONLY	保留	保留	保留
	OPEN	——	保留	保留
	OPERATION	——	保留	——
	OPERATOR	非保留	——	——
	OPTIMIZATION	非保留	——	——
	OPTION	非保留	保留	保留
	OPTIONALLY	非保留	——	——
	OPTIONS	非保留	非保留	——
	OR	保留	保留	保留
	ORDER	保留	保留	保留
	ORDINALITY	——	保留	——
	OUT	非保留（不	保留	——

		能是函数或类型)		
	OUTER	保留（可为函数或类型)	保留	保留
	OUTPUT	——	保留	保留
	OVER	非保留	——	——
	OVERLAPS	保留（可为函数或类型)	非保留	保留
	OVERLAY	非保留（不能是函数或类型)	非保留	——
	OVERRIDING	——	非保留	——
	OWNED	非保留	——	——
	OWNER	非保留	——	——
P	PACKAGE	非保留	——	——
	PACKAGES	非保留	——	——
	PAD	——	保留	保留
	PARAMETER	——	保留	——
	PARAMETERS	——	保留	——
	PARAMETER_MODE	——	非保留	——
	PARAMETER_NAME	——	非保留	——

	PARAMETER_ORDINAL_P OSITION	——	非保留	——
	PARAMETER_SPECIFIC_CA TALOG	——	非保留	——
	PARAMETER_SPECIFIC_N AME	——	非保留	——
	PARAMETER_SPECIFIC_SC HEMA	——	非保留	——
	PARSER	非保留	——	——
	PARTIAL	非保留	保留	保留
	PARTITION	非保留	——	——
	PARTITIONS	非保留	——	——
	PASCAL	——	非保留	非保留
	PASSING	非保留	——	——
	PASSWORD	非保留	——	——
	PATH	——	保留	——
	PCTFREE	非保留	——	——
	PER	非保留	——	——
	PERM	非保留	——	——
	PERCENT	非保留	——	——
	PERFORMANCE	保留	——	——
	PLACING	保留	——	——

	PLAN	非保留	——	——
	PLANS	非保留	——	——
	PLI	——	非保留	非保留
	POOL	非保留	——	——
	POLICY	非保留	——	——
	POSITION	非保留（不能是函数或类型）	非保留	保留
	POSTFIX	——	保留	——
	PRECEDING	非保留	——	——
	PRECISION	非保留（不能是函数或类型）	保留	保留
	PREDICT	非保留	——	——
	PREFERRED	非保留	——	——
	PREFIX	非保留	保留	——
	PREORDER	——	保留	——
	PREPARE	非保留	保留	保留
	PREPARED	非保留	——	——
	PRESERVE	非保留	保留	保留
	PRIMARY	保留	保留	保留
	PRIOR	非保留	保留	保留

	PRIORER	保留	——	——
	PRIVATE	非保留	——	——
	PRIVILEGE	非保留	——	——
	PRIVILEGES	非保留	保留	保留
	PROCEDURAL	非保留	——	——
	PROCEDURE	保留	保留	保留
	PROFILE	非保留	——	——
	PUBLIC	——	保留	保留
	PUBLICATION	非保留	——	——
	PUBLISH	非保留	——	——
	PURGE	非保留	——	——
Q	QUERY	非保留	——	——
	QUOTE	非保留	——	——
R	RANDOMIZED	非保留	——	——
	RANGE	非保留	——	——
	RATIO	非保留	——	——
	RAW	非保留	——	——
	READ	非保留	保留	保留
	READS	——	保留	——
	REAL	非保留（不能是函数或	保留	保留

		类型)		
	REASSIGN	非保留	——	——
	REBUILD	非保留	——	——
	RECHECK	非保留	——	——
	RECURSIVE	非保留	保留	——
	RECYCLEBIN	保留（可为函数或类型)	——	——
	REDISANYVALUE	非保留	——	——
	REF	非保留	保留	——
	REFERENCES	保留	保留	保留
	REFERENCING	——	保留	——
	REFRESH	非保留	——	——
	REINDEX	非保留	——	——
	REJECT	保留	——	——
	RELATIVE	非保留	保留	保留
	RELEASE	非保留	——	——
	RELOPTIONS	非保留	——	——
	REMOTE	非保留	——	——
	REMOVE	非保留	——	——
	RENAME	非保留	——	——

	REPEATABLE	非保留	非保留	非保留
	REPLACE	非保留	——	——
	REPLICA	非保留	——	——
	RESET	非保留	——	——
	RESIZE	非保留	——	——
	RESOURCE	非保留	——	——
	RESTART	非保留	——	——
	RESTRICT	非保留	保留	保留
	RESULT	——	保留	——
	RETURN	非保留	保留	——
	RETURNED_LENGTH	——	非保留	非保留
	RETURNED_OCTET_LENG TH	——	非保留	非保留
	RETURNED_SQLSTATE	——	非保留	非保留
	RETURNING	保留	——	——
	RETURNS	非保留	保留	——
	REUSE	非保留	——	——
	REVOKE	非保留	保留	保留
	RIGHT	保留（可为函数或类型）	保留	保留

	ROLE	非保留	保留	——
	ROLES	非保留	——	——
	ROLLBACK	非保留	保留	保留
	ROLLUP	非保留	保留	——
	ROTATION	非保留	——	——
	ROUTINE	——	保留	——
	ROUTINE_CATALOG	——	非保留	——
	ROUTINE_NAME	——	非保留	——
	ROUTINE_SCHEMA	——	非保留	——
	ROW	非保留（不能是函数或类型）	保留	——
	ROWS	非保留	保留	保留
	ROWTYPE	非保留	——	——
	ROW_COUNT	——	非保留	非保留
	RULE	非保留	——	——
	ROWNUM	保留	保留	——
S	SAMPLE	非保留	——	——
	SAVEPOINT	非保留	保留	——
	SCALE	——	非保留	非保留
	SCHEMA	非保留	保留	保留

	SCHEMA_NAME	——	非保留	非保留
	SCOPE	——	保留	——
	SCROLL	非保留	保留	保留
	SEARCH	非保留	保留	——
	SECOND	非保留	保留	保留
	SECTION	——	保留	保留
	SECURITY	非保留	非保留	——
	SELECT	保留	保留	保留
	SELF	——	非保留	——
	SENSITIVE	——	非保留	——
	SEQUENCE	非保留	保留	——
	SEQUENCES	非保留	——	——
	SERIALIZABLE	非保留	非保留	非保留
	SERVER	非保留	——	——
	SERVER_NAME	——	非保留	非保留
	SESSION	非保留	保留	保留
	SESSION_USER	保留	保留	保留
	SET	非保留	保留	保留
	SETOF	非保留（不能是函数或类型）	——	——

	SETS	非保留	保留	——
	SHARE	非保留	——	——
	SHIPPABLE	非保留	——	——
	SHOW	非保留	——	——
	SHUTDOWN	非保留	——	——
	SIBLINGS	非保留	——	——
	SIMILAR	保留（可为函数或类型）	非保留	——
	SIMPLE	非保留	非保留	——
	SIZE	非保留	保留	保留
	SKIP	非保留	——	——
	SLICE	非保留	——	——
	SMALLDATETIME_FORMAT	非保留	——	——
	SMALLDATETIME	非保留（不能是函数或类型）	——	——
	SMALLINT	非保留（不能是函数或类型）	保留	保留
	SNAPSHOT	非保留	——	——
	SOME	保留	保留	保留

	SOURCE	非保留	非保留	——
	SPACE	非保留	保留	保留
	SPECIFIC	——	保留	——
	SPECIFICTYPE	——	保留	——
	SPECIFIC_NAME	——	非保留	——
	SPILL	非保留	——	——
	SPLIT	非保留	——	——
	SQL	——	保留	保留
	SQLCODE	——	——	保留
	SQLERROR	——	——	保留
	SQLEXCEPTION	——	保留	——
	SQLSTATE	——	保留	保留
	SQLWARNING	——	保留	——
	STABLE	非保留	——	——
	STANDALONE	非保留	——	——
	START	非保留	保留	——
	STATE	——	保留	——
	STATEMENT	非保留	保留	——
	STATEMENT_ID	非保留	——	——
	STATIC	——	保留	——

	STATISTICS	非保留	——	——
	STDIN	非保留	——	——
	STDOUT	非保留	——	——
	STORAGE	非保留	——	——
	STORE	非保留	——	——
	STORED	非保留	——	——
	STRATIFY	非保留	——	——
	STREAM	非保留	——	——
	STRICT	非保留	——	——
	STRIP	非保留	——	——
	STRUCTURE	——	保留	——
	STYLE	——	非保留	——
	SUBCLASS_ORIGIN	——	非保留	非保留
	SUBLIST	——	非保留	——
	SUBPARTITION	非保留	——	——
	SUBSCRIPTION	非保留	——	——
	SUBSTRING	非保留（不能是函数或类型）	非保留	保留
	SUM	——	非保留	保留
	SYMMETRIC	保留	非保留	——

	SYNONYM	非保留	——	——
	SYS_REFCURSOR	非保留	——	——
	SYSDATE	保留	——	——
	SYSID	非保留	——	——
	SYSTEM	非保留	非保留	——
	SYSTEM_USER	——	保留	保留
T	TABLE	保留	保留	保留
	TABLES	非保留	——	——
	TABLESAMPLE	保留（可为函数或类型）	——	——
	TABLESPACE	非保留	——	——
	TABLE_NAME	——	非保留	非保留
	TARGET	非保留	——	——
	TIME_FORMAT	非保留	——	——
	TIMESTAMP_FORMAT	非保留	——	——
	TEMP	非保留	——	——
	TEMPLATE	非保留	——	——
	TEMPORARY	非保留	保留	保留
	TERMINATE	——	保留	——
	TERMINATED	非保留	——	——

	TEXT	非保留	——	——
	THAN	非保留	保留	——
	THEN	保留	保留	保留
	TIME	非保留（不能是函数或类型）	保留	保留
	TIMECAPSULE	保留（可为函数或类型）	——	——
	TIMESTAMP	非保留（不能是函数或类型）	保留	保留
	TIMESTAMPDIFF	非保留（不能是函数或类型）	——	——
	TIMEZONE_HOUR	——	保留	保留
	TIMEZONE_MINUTE	——	保留	保留
	TINYINT	非保留（不能是函数或类型）	——	——
	TO	保留	保留	保留
	TRAILING	保留	保留	保留
	TRANSACTION	非保留	保留	保留
	TRANSACTIONS_comm I TTED	——	非保留	——

	TRANSACTIONS_ROLLED_BACK	——	非保留	——
	TRANSACTION_ACTIVE	——	非保留	——
	TRANSFORM	非保留	非保留	——
	TRANSFORMS	——	非保留	——
	TRANSLATE	——	非保留	保留
	TRANSLATION	——	保留	保留
	TREAT	非保留（不能是函数或类型）	保留	——
	TRIGGER	非保留	保留	——
	TRIGGER_CATALOG	——	非保留	——
	TRIGGER_NAME	——	非保留	——
	TRIGGER_SCHEMA	——	非保留	——
	TRIM	非保留（不能是函数或类型）	非保留	保留
	TRUE	保留	保留	保留
	TRUNCATE	非保留	——	——
	TRUSTED	非保留	——	——
	TSFIELD	非保留	——	——

	TSTAG	非保留	——	——
	TSTIME	非保留	——	——
	TYPE	非保留	非保留	非保留
	TYPES	非保留	——	——
U	UESCAPE	——	——	——
	UNBOUNDED	非保留	——	——
	UNCOMMITTED	非保留	非保留	非保留
	UNDER	——	保留	——
	UNENCRYPTED	非保留	——	——
	UNION	保留	保留	保留
	UNIQUE	保留	保留	保留
	UNKNOWN	非保留	保留	保留
	UNLIMITED	非保留	——	——
	UNLISTEN	非保留	——	——
	UNLOCK	非保留	——	——
	UNLOGGED	非保留	——	——
	UNNAMED	——	非保留	非保留
	UNNEST	——	保留	——
	UNTIL	非保留	——	——
	UNUSABLE	非保留	——	——

	UPDATE	非保留	保留	保留
	UPPER	——	非保留	保留
	USAGE	——	保留	保留
	USEEOF	非保留	——	——
	USER	保留	保留	保留
	USER_DEFINED_TYPE_CA TALOG	——	非保留	——
	USER_DEFINED_TYPE_NA ME	——	非保留	——
	USER_DEFINED_TYPE_SC HEMA	——	非保留	——
	USING	保留	保留	保留
V	VACUUM	非保留	——	——
	VALID	非保留	——	——
	VALIDATE	非保留	——	——
	VALIDATION	非保留	——	——
	VALIDATOR	非保留	——	——
	VALUE	非保留	保留	保留
	VALUES	非保留（不能是函数或类型）	保留	保留
	VARCHAR	非保留（不能是函数或	保留	保留

		类型)		
	VARCHAR2	非保留（不能是函数或类型)	——	——
	VARIABLE	——	保留	——
	VARIABLES	非保留	——	——
	VARIADIC	保留	——	——
	VARYING	非保留	保留	保留
	VCGROUP	非保留	——	——
	VERBOSE	保留（可为函数或类型)	——	——
	VERSION	非保留	——	——
	VERIFY	保留	——	——
	VIEW	非保留	保留	保留
	VOLATILE	非保留	——	——
W	WAIT	非保留	——	——
	WEAK	非保留	——	——
	WHEN	保留	保留	保留
	WHENEVER	——	保留	保留
	WHERE	保留	保留	保留
	WHITESPACE	非保留	——	——

	WINDOW	保留	——	——
	WITH	保留	保留	保留
	WITHIN	非保留	——	——
	WITHOUT	非保留	保留	——
	WORK	非保留	保留	保留
	WORKLOAD	非保留	——	——
	WRAPPER	非保留	——	——
	WRITE	非保留	保留	保留
X	XML	非保留	——	——
	XMLATTRIBUTES	非保留（不能是函数或类型）	——	——
	XMLCONCAT	非保留（不能是函数或类型）	——	——
	XMLELEMENT	非保留（不能是函数或类型）	——	——
	XMLEXISTS	非保留（不能是函数或类型）	——	——
	XMLFOREST	非保留（不能是函数或类型）	——	——

	XMLPARSE	非保留（不能是函数或类型）	——	——
	XMLPI	非保留（不能是函数或类型）	——	——
	XMLROOT	非保留（不能是函数或类型）	——	——
	XMLSERIALIZE	非保留（不能是函数或类型）	——	——
Y	YEAR	非保留	保留	保留
	YES	非保留	——	——
Z	ZONE	非保留	保留	保留

3.2 常量与宏

GBase 8s 支持的常量和宏。

表 3-2 常量与宏

参数	描述	示例
CURRENT_CATALOG	当前数据库	postgres=# SELECT CURRENT_CATALOG; current_database ---------- postgres (1 row)
CURRENT_ROLE	当前用户	postgres=# SELECT CURRENT_ROLE; current_user ---------- gbase

		(1 row)
CURRENT_SCHEMA	当前数据库模式	postgres=# SELECT CURRENT_SCHEMA; current_schema ---------------- public (1 row)
CURRENT_USER	当前用户	postgres=# SELECT CURRENT_USER; current_user -------------- gbase (1 row)
LOCALTIMESTAMP	当前会话时间（无时区）	postgres=# SELECT LOCALTIMESTAMP; timestamp ---------------------------- 2022-05-10 15:37:30.968538 (1 row)
NULL	空值	——
SESSION_USER	当前系统用户	postgres=# SELECT SESSION_USER; session_user --------------- gbase (1 row)
SYSDATE	当前系统日期	postgres=# SELECT SYSDATE; sysdate --------------------- 2022-05-10 15:48:53 (1 row)
USER	当前用户。为CURRENT_USER 的别名。	postgres=# SELECT USER; current_user --------------- gbase (1 row)

3.3 表达式

3.3.1 简单表达式

3.3.1.1 逻辑表达式

逻辑表达式的操作符和运算规则，请参见逻辑操作符。

3.3.1.2 比较表达式

常用的比较操作符，请参见比较操作符。

除比较操作符外，还可以使用以下句式结构：

- BETWEEN 操作符

  a BETWEEN x AND y 等效于 a >= x AND a <= y。

  a NOT BETWEEN x AND y 等效于 a < x OR a > y。

- 检查一个值是不是 null，可使用：

```
expression IS NULL
expression IS NOT NULL
```

或者与之等价的句式结构，但不是标准的：

```
expression ISNULL
expression NOTNULL
```

须知

不要写 expression=NULL 或 expression<>(!=)NULL，因为 NULL 代表一个未知的值，不能通过该表达式判断两个未知值是否相等。

- is distinct from/is not distinct from

  - is distinct from

  A 和 B 的数据类型、值不完全相同时为 true。

  A 和 B 的数据类型、值完全相同时为 false。

  将空值视为相同。

  - is not distinct from

  A 和 B 的数据类型、值不完全相同时为 false。

  A 和 B 的数据类型、值完全相同时为 true。

  将空值视为相同。

3.3.1.3 伪列

- ROWNUM

ROWNUM 是一个伪列，它返回一个数字，表示从查询中获取结果的行编号。第一行的 ROWNUM 为 1，第二行的为 2，依此类推。

ROWNUM 的返回类型为 numeric。ROWNUM 可以用于限制查询返回的总行数，例如下面语句限制查询从 Students 表中返回最多 10 条记录。

```
postgres=# select * from Students where rownum <= 10;
```

示例

```
 postgres=# SELECT 2 BETWEEN 1 AND 3 AS RESULT;
 result
----------
 t
(1 row)
postgres=# SELECT 2 >= 1 AND 2 <= 3 AS RESULT;
 result
----------
 t
(1 row)
postgres=# SELECT 2 NOT BETWEEN 1 AND 3 AS RESULT;
 result
----------
 f
(1 row)
postgres=# SELECT 2 < 1 OR 2 > 3 AS RESULT;
 result
----------
 f
(1 row)
postgres=# SELECT 2+2 IS NULL AS RESULT;
 result
----------
 f
(1 row)
postgres=# SELECT 2+2 IS NOT NULL AS RESULT;
 result
----------
 t
```

```
(1 row)
postgres=# SELECT 2+2 ISNULL AS RESULT;
 result
----------
 f
(1 row)
postgres=# SELECT 2+2 NOTNULL AS RESULT;
 result
----------
 t
(1 row)
postgres=# SELECT 2+2 IS DISTINCT FROM NULL AS RESULT;
 result
----------
 t
(1 row)
postgres=# SELECT 2+2 IS NOT DISTINCT FROM NULL AS RESULT;
 result
----------
 f
(1 row)
```

3.3.2 条件表达式

在执行 SQL 语句时，可通过条件表达式筛选出符合条件的数据。

条件表达式主要有以下几种：

- CASE

CASE 表达式是条件表达式，类似于其他编程语言中的 CASE 语句。

CASE 表达式的语法图请参考下图。

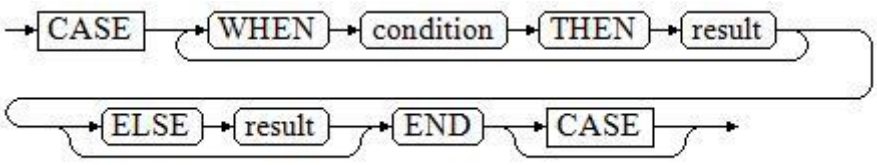

图 3-1 case::=

CASE 子句可以用于合法的表达式中。condition 是一个返回 BOOLEAN 数据类型的表达式：

- 如果结果为真，CASE 表达式的结果就是符合该条件所对应的 result。
- 如果结果为假，则以相同方式处理随后的 WHEN 或 ELSE 子句。
- 如果各 WHEN condition 都不为真，表达式的结果就是在 ELSE 子句执行的 result。如果省略了 ELSE 子句且没有匹配的条件，结果为 NULL。

示例：

```
postgres=# CREATE TABLE public.case_when_t1(CW_COL1 INT);
CREATE TABLE
postgres=# INSERT INTO public.case_when_t1 VALUES (1), (2), (3);
INSERT 0 3
postgres=# SELECT * FROM public.case_when_t1;
cw_col1
---------
       1
       2
       3
(3 rows)
postgres=# SELECT CW_COL1, CASE WHEN CW_COL1=1 THEN 'one' WHEN CW_COL1=2 THEN
'two' ELSE 'other' END FROM public.case_when_t1 ORDER BY 1;
cw_col1 | case
---------+-------
       1 | one
       2 | two
       3 | other
(3 rows)
postgres=# DROP TABLE public.case_when_t1;
DROP TABLE
```

- DECODE

DECODE 的语法图请参见下图。

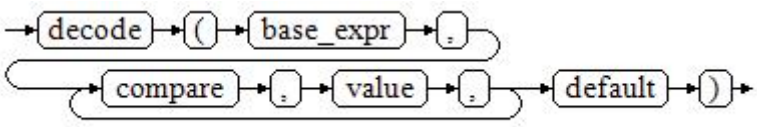

图 3-2 decode::=

将表达式 base_expr 与后面的每个 compare(n) 进行比较，如果匹配返回相应的 value(n)。如果没有发生匹配，则返回 default。

示例请参见条件表达式函数。

```
postgres=# SELECT DECODE('A','A',1,'B',2,0);
 case
------
    1
(1 row)
```

- COALESCE

COALESCE 的语法图请参见下图。

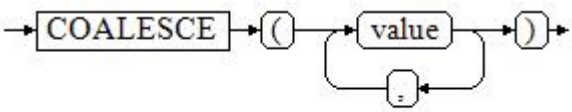

图 3-3 coalesce::=

COALESCE 返回它的第一个非 NULL 的参数值。如果参数都为 NULL，则返回 NULL。它常用于在显示数据时用缺省值替换 NULL。和 CASE 表达式一样，COALESCE 只计算用来判断结果的参数，即在第一个非空参数右边的参数不会被计算。

示例

```
postgres=# CREATE TABLE public.c_tabl(description varchar(10),
short_description varchar(10), last_value varchar(10)) ;
CREATE TABLE
postgres=# INSERT INTO public.c_tabl VALUES('abc', 'efg', '123');
INSERT 0 1
postgres=# INSERT INTO public.c_tabl VALUES(NULL, 'efg', '123');
INSERT 0 1
postgres=# INSERT INTO public.c_tabl VALUES(NULL, NULL, '123');
INSERT 0 1
postgres=# SELECT description, short_description, last_value,
COALESCE(description, short_description, last_value) FROM public.c_tabl ORDER
BY 1, 2, 3, 4;
 description | short_description | last_value | coalesce
-------------+-------------------+------------+----------
 abc         | efg               | 123        | abc
             | efg               | 123        | efg
             |                   | 123        | 123
(3 rows)
postgres=# DROP TABLE public.c_tabl;
```

```
DROP TABLE
```

如果 description 不为 NULL，则返回 description 的值，否则计算下一个参数 short_description；如果 short_description 不为 NULL，则返回 short_description 的值，否则计算下一个参数 last_value；如果 last_value 不为 NULL，则返回 last_value 的值，否则返回(none)。

```
postgres=# SELECT COALESCE(NULL,'Hello World');
  coalesce
--------------
 Hello World
(1 row)
```

- NULLIF

NULLIF 的语法图请参见下图。

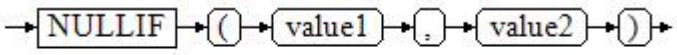

图 3-4 nullif::=

只有当 value1 和 value2 相等时，NULLIF 才返回 NULL。否则它返回 value1。

示例

```
postgres=# CREATE TABLE public.null_if_t1 ( NI_VALUE1 VARCHAR(10),
NI_VALUE2 VARCHAR(10));
CREATE TABLE
postgres=# INSERT INTO public.null_if_t1 VALUES('abc', 'abc');
INSERT 0 1
postgres=# INSERT INTO public.null_if_t1 VALUES('abc', 'efg');
INSERT 0 1
postgres=# SELECT NI_VALUE1, NI_VALUE2, NULLIF(NI_VALUE1, NI_VALUE2) FROM
public.null_if_t1 ORDER BY 1, 2, 3;
ni_value1 | ni_value2 | nullif
----------+----------+-------
 abc      | abc      |
 abc      | efg      | abc
(2 rows)
postgres=# DROP TABLE public.null_if_t1;
DROP TABLE
```

如果 value1 等于 value2 则返回 NULL，否则返回 value1。

```
postgres=# SELECT NULLIF('Hello','Hello World');
```

```
 nullif
--------
 Hello
(1 row)
```

- GREATEST（最大值），LEAST（最小值）

  GREATEST 的语法图请参见下图。

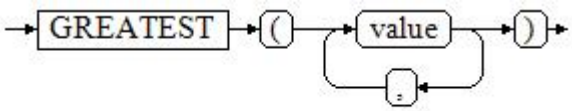

图 3-5 greatest::=

从一个任意数字表达式的列表里选取最大的数值。

示例：

```
postgres=# SELECT greatest(9000,155555,2.01);
 greatest
----------
   155555
(1 row)
```

LEAST 的语法图请参见下图。

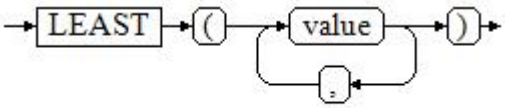

图 3-6 least::=

从一个任意数字表达式的列表里选取最小的数值。

以上的数字表达式必须都可以转换成一个普通的数据类型，该数据类型将是结果类型。

列表中的 NULL 值将被忽略。只有所有表达式的结果都是 NULL 的时候，结果才是 NULL。

```
postgres=# SELECT least(9000,2);
 least
-------
     2
(1 row)
```

示例请参见条件表达式函数。

- NVL

NVL 的语法图请参见下图。

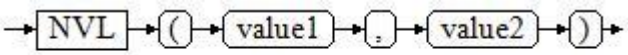

图 3-7 nvl::=

如果 value1 为 NULL 则返回 value2，如果 value1 非 NULL，则返回 value1。

示例：

```
postgres=# SELECT nvl(null,1);
nvl
-----
 1
(1 row)
postgres=# SELECT nvl ('Hello World' ,1);
      nvl
---------------
 Hello World
(1 row)
```

3.3.3 子查询表达式

子查询表达式主要有以下几种：

- EXISTS/NOT EXISTS

EXISTS/NOT EXISTS 的语法图请参见下图。

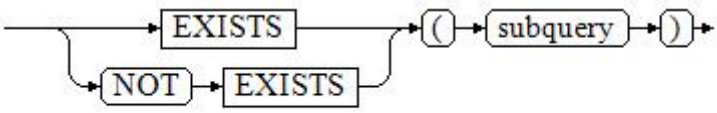

图 3-8 EXISTS/NOT EXISTS::=

EXISTS 的参数是一个任意的 SELECT 语句，或者说子查询。系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，则 EXISTS 结果就为“真”；如果子查询没有返回任何行，EXISTS 的结果是“假”。

这个子查询通常只是运行到能判断它是否可以生成至少一行为止，而不是等到全部结束。

示例：

```
 postgres=# SELECT sr_reason_sk, sr_customer_sk FROM public.store_returns WHERE
EXISTS (SELECT d_dom FROM public.date_dim WHERE d_dom =
store_returns.sr_reason_sk and sr_customer_sk <10);
sr_reason_sk | sr_customer_sk
--------------+----------------
           13 |              2
           22 |              5
           17 |              7
           25 |              7
            3 |              7
           31 |              5
            7 |              7
           14 |              6
           20 |              4
            5 |              6
           10 |              3
            1 |              5
           15 |              2
            4 |              1
           26 |              3
(15 rows)
```

- IN/NOT IN

IN/NOT IN 的语法请参见下图。

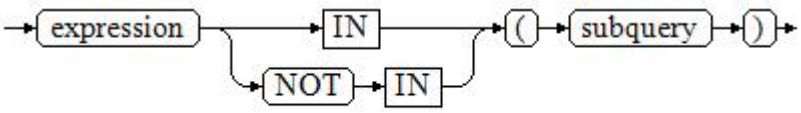

图 3-9 IN/NOT IN::=

右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式对子查询结果的每一行进行一次计算和比较。如果找到任何相等的子查询行，则 IN 结果为“真”。如果没有找到任何相等行，则结果为“假”（包括子查询没有返回任何行的情况）。

表达式或子查询行里的 NULL 遵照 SQL 处理布尔值和 NULL 组合时的规则。如果两个行对应的字段都相等且非空，则这两行相等；如果任意对应字段不等且非空，则这两行不等；否则结果是未知（NULL）。如果每一行的结果都是不等或 NULL ，并且至少有一个 NULL ，

则 IN 的结果是 NULL 。

示例:

```
postgres=# SELECT sr_reason_sk, sr_customer_sk  FROM public.store_returns WHERE
sr_customer_sk IN (SELECT d_dom FROM public.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
--------------+----------------
          10 |              3
          26 |              3
          22 |              5
          31 |              5
           1 |              5
          32 |              5
          32 |              5
           4 |              1
          15 |              2
          13 |              2
          33 |              4
          20 |              4
          33 |              8
           5 |              6
          14 |              6
          17 |              7
           3 |              7
          25 |              7
           7 |              7
(19 rows)
```

- ANY/SOME

ANY/SOME 的语法图请参见下图。

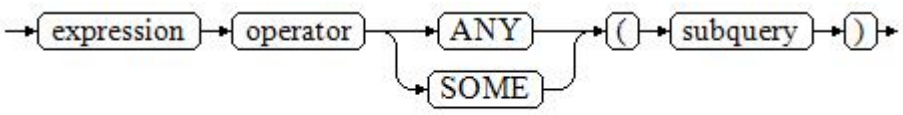

图 3-10 any/some::=

右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式使用 operator 对子查询结果的每一行进行一次计算和比较，其结果必须是布尔值。如果至少获得一个真值，则 ANY 结果为“真”。如果全部获得假值，则结果是“假”（包括子查询没有返回任何行的情况）。SOME 是 ANY 的同义词。IN 与 ANY 可以等效替换 。

示例：

```
postgres=# SELECT sr_reason_sk, sr_customer_sk  FROM public.store_returns WHERE
sr_customer_sk < ANY (SELECT d_dom FROM public.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
--------------+----------------
           26 |              3
           17 |              7
           32 |              5
           32 |              5
           13 |              2
           31 |              5
           25 |              7
            5 |              6
            7 |              7
           10 |              3
            1 |              5
           14 |              6
            4 |              1
            3 |              7
           22 |              5
           33 |              4
           20 |              4
           33 |              8
           15 |              2
(19 rows)
```

- ALL

ALL 的语法请参见下图。

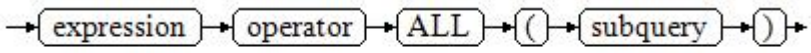

图 3-11 all::=

右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式使用 operator 对子查询结果的每一行进行一次计算和比较，其结果必须是布尔值。如果全部获得真值，ALL 结果为“真”（包括子查询没有返回任何行的情况）。如果至少获得一个假值，则结果是“假”。

示例：

```
postgres=# SELECT sr_reason_sk, sr_customer_sk  FROM public.store_returns WHERE
sr_customer_sk < all(SELECT d_dom FROM public.date_dim WHERE d_dom < 10);
 sr_reason_sk | sr_customer_sk
--------------+----------------
(0 rows)
```

3.3.4 数组表达式

3.3.4.1 IN

expression IN (value [, …])

右侧括号中的是一个表达式列表。左侧表达式的结果与表达式列表的内容进行比较。如果列表中的内容符合左侧表达式的结果，则 IN 的结果为 true。如果没有相符的结果，则 IN 的结果为 false。

示例如下：

```
postgres=# SELECT 8000+500 IN (10000, 9000) AS RESULT;
  result
----------
 f
(1 row)
```

如果表达式结果为 null，或者表达式列表不符合表达式的条件且右侧表达式列表返回结果至少一处为空，则 IN 的返回结果为 null，而不是 false。这样的处理方式和 SQL 返回空值的布尔组合规则是一致的。

3.3.4.2 NOT IN

expression NOT IN (value [, ...])

右侧括号中的是一个表达式列表。左侧表达式的结果与表达式列表的内容进行比较。如果在列表中的内容没有符合左侧表达式结果的内容，则 NOT IN 的结果为 true。如果有符合的内容，则 NOT IN 的结果为 false。

示例如下：

```
postgres=# SELECT 8000+500 NOT IN (10000, 9000) AS RESULT;
result
----------
t
(1 row)
```

如果查询语句返回结果为空，或者表达式列表不符合表达式的条件且右侧表达式列表返

回结果至少一处为空，则 NOT IN 的返回结果为 null，而不是 false。这样的处理方式和 SQL 返回空值的布尔组合规则是一致的。

说明

- 在所有情况下 X NOT IN Y 等价于 NOT(X IN Y)。

3.3.4.3 ANY/SOME (array)

expression operator ANY (array expression) expression operator SOME (array expression)

```
postgres=# SELECT 8000+500 < SOME (array[10000,9000]) AS RESULT;
result
----------
t
(1 row)
postgres=# SELECT 8000+500 < ANY (array[10000,9000]) AS RESULT;
result
----------
t
(1 row)
```

右侧括号中的是一个数组表达式，它必须产生一个数组值。左侧表达式的结果使用操作符对数组表达式的每一行结果都进行计算和比较，比较结果必须是布尔值。

如果对比结果至少获取一个真值，则 ANY 的结果为 true。

如果对比结果没有真值，则 ANY 的结果为 false。

如果结果没有真值，并且数组表达式生成至少一个值为 null，则 ANY 的值为 NULL，而不是 false。这样的处理方式和 SQL 返回空值的布尔组合规则是一致的。

说明

- SOME 是 ANY 的同义词。

3.3.4.4 ALL (array)

expression operator ALL (array expression)

右侧括号中的是一个数组表达式，它必须产生一个数组值。左侧表达式的结果使用操作符对数组表达式的每一行结果都进行计算和比较，比较结果必须是布尔值。

如果所有的比较结果都为真值(包括数组不含任何元素的情况)，则 ALL 的结果为 true。

如果存在一个或多个比较结果为假值，则 ALL 的结果为 false。

如果数组表达式产生一个 NULL 数组，则 ALL 的结果为 NULL。如果左边表达式的值为 NULL ，则 ALL 的结果通常也为 NULL(某些不严格的比较操作符可能得到不同的结果)。另外，如果右边的数组表达式中包含 null 元素并且比较结果没有假值，则 ALL 的结果将是 NULL(某些不严格的比较操作符可能得到不同的结果)， 而不是真。这样的处理方式和 SQL 返回空值的布尔组合规则是一致的。

```
postgres=# SELECT 8000+500 < ALL (array[10000,9000]) AS RESULT;
result
----------
t
(1 row)
```

3.3.5 行表达式

语法如下：

```
row_constructor  operator row_constructor
```

两边都是一个行构造器，两行值必须具有相同数目的字段，每一行都进行比较，行比较允许使用=，<>，<，<=，>=等操作符，或其中一个相似的语义符。

=<>和别的操作符使用略有不同。如果两行值的所有字段都是非空并且相等，则认为两行是相等的；如果两行值的任意字段为非空并且不相等，则认为两行是不相等的；否则比较结果是未知的（null）。

对于<，<=，>，>=的情况下，行中元素从左到右依次比较，直到遇到一对不相等的元素或者一对为空的元素。如果这对元素中存在至少一个 null 值，则比较结果是未知的

（null），否则这对元素的比较结果为最终的结果。

示例：

```
postgres=# SELECT ROW(1,2,NULL) < ROW(1,3,0) AS RESULT;
result
----------
t
(1 row)
```

3.4 DDL 语法一览表

DDL（Data Definition Language 数据定义语言），用于定义或修改数据库中的对象。如：表、索引、视图等。

说明

- GBase 8s 不支持数据库主节点不完整时进行 DDL 操作。例如：GBase 8s 中有 1 个数据库主节点故障时执行新建数据库、表等操作都会失败。

3.4.1 定义客户端加密主密钥

客户端加密主密钥主要用于密态数据库特性，用来加密列加密密钥(cek)。客户端加密主密钥定义主要包括创建客户端加密主密钥以及删除客户端加密主密钥。所涉及的 SQL 语句，请参考下表。

表 3-3 客户端加密主密钥定义相关 SQL

功能	相关 SQL
创建客户端加密主密钥	CREATE CLIENT MASTER KEY
删除客户端加密主密钥	DROP CLIENT MASTER KEY

3.4.2 定义列加密密钥

列加密密钥主要用于密态数据库特性中，用来加密数据。列加密密钥定义主要包括创建列加密密钥以及删除列加密密钥。所涉及的 SQL 语句，请参考下表。

表 3-4 列加密密钥定义相关 SQL

功能	相关 SQL
创建列加密密钥	CREATE COLUMN ENCRYPTION KEY
删列加密密钥	DROP COLUMN ENCRYPTION KEY

3.4.3 定义数据库

数据库是组织、存储和管理数据的仓库，而数据库定义主要包括：创建数据库、修改数据库属性，以及删除数据库。所涉及的 SQL 语句，请参考下表。

表 3-5 数据库定义相关 SQL

功能	相关 SQL
创建数据库	CREATE DATABASE
修改数据库属性	ALTER DATABASE
删除数据库	DROP DATABASE

3.4.4 定义模式

模式是一组数据库对象的集合，主要用于控制对数据库对象的访问。所涉及的 SQL 语句，请参考下表。

表 3-6 模式定义相关 SQL

功能	相关 SQL
创建模式	CREATE SCHEMA
修改模式属性	ALTER SCHEMA
删除模式	DROP SCHEMA

3.4.5 定义表空间

表空间用于管理数据对象，与磁盘上的一个目录对应。所涉及的 SQL 语句，请参考下表。

表 3-7 表空间定义相关 SQL

功能	相关 SQL
创建表空间	CREATE TABLESPACE
修改表空间属性	ALTER TABLESPACE
删除表空间	DROP TABLESPACE

3.4.6 定义表

表是数据库中的一种特殊数据结构，用于存储数据对象以及对象之间的关系。所涉及的 SQL 语句，请参考下表。

表 3-8 表定义相关 SQL

功能	相关 SQL
创建表	CREATE TABLE
修改表属性	ALTER TABLE
删除表	DROP TABLE

3.4.7 定义分区表

分区表是一种逻辑表，数据是由普通表存储的，主要用于提升查询性能。所涉及的 SQL 语句，请参考下表。

表 3-9 分区表定义相关 SQL

功能	相关 SQL
创建分区表	CREATE TABLE PARTITION
创建分区	ALTER TABLE PARTITION
修改分区表属性	
删除分区	
删除分区表	DROP TABLE

3.4.8 定义索引

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。所涉及的 SQL 语句，请参考下表。

表 3-10 索引定义相关 SQL

功能	相关 SQL
创建索引	CREATE INDEX
修改索引属性	ALTER INDEX
删除索引	DROP INDEX
重建索引	REINDEX

3.4.9 定义存储过程

存储过程是一组为了完成特定功能的 SQL 语句集，经编译后存储在数据库中，用户通过指定存储过程的名称并给出参数（如果该存储过程带有参数）来执行它。所涉及的 SQL 语句，请参考下表。

表 3-11 存储过程定义相关 SQL

功能	相关 SQL
创建存储过程	CREATE PROCEDURE
删除存储过程	DROP PROCEDURE

3.4.10定义函数

在 GBase 8s 中，它和存储过程类似，也是一组 SQL 语句集，使用上没有差别。所涉及的 SQL 语句，请参考下表。

表 3-12 函数定义相关 SQL

功能	相关 SQL
创建函数	CREATE FUNCTION
修改函数属性	ALTER FUNCTION

删除函数	DROP FUNCTION

3.4.11 定义包

包是模块化的思想，由包头（package specification）和包体(package body)组成， 用来分类管理存储过程和函数，类似于 Java、C++等语言中的类。

表 3-13 包定义相关 SQL

功能	相关 SQL
创建包	CREATE PACKAGE
删除包	DROP PACKAGE
修改包属性	ALTER PACKAGE

3.4.12 定义视图

视图是从一个或几个基本表中导出的虚表，可用于控制用户对数据访问，请参考下表。

表 3-14 视图定义相关 SQL

功能	相关 SQL
创建视图	CREATE VIEW
删除视图	DROP VIEW

3.4.13 定义游标

为了处理 SQL 语句，存储过程进程分配一段内存区域来保存上下文联系。游标是指向上下文区域的句柄或指针。借助游标，存储过程可以控制上下文区域的变化，请参考下表。

表 3-15 游标定义相关 SQL

功能	相关 SQL

创建游标	CURSOR
移动游标	MOVE
从游标中提取数据	FETCH
关闭游标	CLOSE

3.4.14定义聚合函数

表 3-16 聚合函数定义相关 SQL

功能	相关 SQL
创建一个新的聚合函数	CREATE AGGREGATE
修改聚合函数	ALTER AGGREGATE
删除聚合函数	DROP AGGREGATE

3.4.15定义数据类型转换

表 3-17 数据类型定义相关 SQL

功能	相关 SQL
创建一个新的用户自定义数据类型转换	CREATE CAST
删除用户自定义数据类型转换	DROP CAST

3.4.16定义插件扩展

表 3-18 插件扩展定义相关 SQL

功能	相关 SQL
创建一个新的插件扩展	CREATE EXTENSION

修改插件扩展	ALTER EXTENSION
删除插件扩展	DROP EXTENSION

3.4.17定义操作符

表 3-19 操作符定义相关 SQL

功能	相关 SQL
创建一个新的操作符	CREATE OPERATOR
修改操作符	ALTER OPERATOR
删除操作符	DROP OPERATOR

3.4.18定义过程语言

表 3-20 过程语言定义相关 SQL

功能	相关 SQL
创建一个新的过程语言	CREATE LANGUAGE
修改过程语言	ALTER LANGUAGE
删除过程语言	DROP LANGUAGE

3.4.19定义数据类型

表 3-21 数据类型定义相关 SQL

功能	相关 SQL
创建一个新的数据类型	CREATE TYPE
修改数据类型	ALTER TYPE

删除数据类型	DROP TYPE

3.5 DML 语法一览表

DML（Data Manipulation Language 数据操作语言），用于对数据库表中的数据进行操作。如：插入、更新、查询、删除。

3.5.1 插入数据

插入数据是往数据库表中添加一条或多条记录，请参考 INSERT。

3.5.2 修改数据

修改数据是修改数据库表中的一条或多条记录，请参考 UPDATE。

3.5.3 查询数据

数据库查询语句 SELECT 是用于在数据库中检索适合条件的信息，请参考 SELECT。

3.5.4 删除数据

GBase 8s 提供了两种删除表数据的语句：删除表中指定条件的数据，请参考 DELETE；或删除表的所有数据，请参考 TRUNCATE。

TRUNCATE 快速地从表中删除所有行，它和在每个表上进行无条件的 DELETE 有同样的效果，不过因为它不做表扫描，因而快得多。在大表上最有用。

3.5.5 拷贝数据

GBase 8s 提供了在表和文件之间拷贝数据的语句，请参考 COPY。

3.5.6 锁定表

GBase 8s 提供了多种锁模式用于控制对表中数据的并发访问，请参考 LOCK。

3.5.7 调用函数

GBase 8s 提供了三个用于调用函数的语句，它们在语法结构上没有差别，请参考 CALL。

3.5.8 操作会话

用户与数据库之间建立的连接称为会话，请参考下表。

表 3-22 会话相关 SQL

功能	相关 SQL
修改会话	ALTER SESSION
结束会话	ALTER SYSTEM KILL SESSION

3.6 DCL 语法一览表

DCL（Data Control Language 数据控制语言），是用来创建用户角色、设置或更改数据库用户或角色权限的语句。

3.6.1 定义角色

角色是用来管理权限的，从数据库安全的角度考虑，可以把所有的管理和操作权限划分到不同的角色上。所涉及的 SQL 语句，请参考下表。

表 3-23 角色定义相关 SQL

功能	相关 SQL
创建角色	CREATE ROLE
修改角色属性	ALTER ROLE
删除角色	DROP ROLE

3.6.2 定义用户

用户是用来登录数据库的，通过对用户赋予不同的权限，可以方便地管理用户对数据库的访问及操作。所涉及的 SQL 语句，请参考下表。

表 3-24 用户定义相关 SQL

功能	相关 SQL
创建用户	CREATE USER
修改用户属性	ALTER USER
删除用户	DROP USER

3.6.3 授权

GBase 8s 提供了针对数据对象和角色授权的语句，请参考 GRANT。

3.6.4 收回权限

GBase 8s 提供了收回权限的语句，请参考 REVOKE。

3.6.5 设置默认权限

GBase 8s 允许设置应用于将来创建的对象的权限，请参考 ALTER DEFAULT PRIVILEGES。

3.6.6 关闭当前节点

GBase 8s 支持使用 shutdown 命令关闭当前数据库节点，请参考 SHUTDOWN。

3.7 子查询

子查询或称为内部查询，嵌套查询，指的是在数据库查询的 WHERE 子句中嵌入查询语句，相当于临时表。一个 SELECT 语句的查询结果能够作为另一个语句的输入值。

子查询可以与 SELECT，INSERT，UPDATE 和 DELETE 语句一起使用。

以下是子查询必须遵守的几个规则：

- 子查询必须用括号括起来。
- 子查询在 SELECT 子句中只能有一个列，除非在主查询中有多列，与子查询的所选列进行比较。
- ORDER BY 不能用在子查询中，虽然主查询可以使用 ORDER BY。可以在子查询中使

用 GROUP BY，功能与 ORDER BY 相同。

- 子查询返回多于一行，只能与多值运算符一起使用，如 IN 运算符。
- BETWEEN 运算符不能与子查询一起使用，但是，BETWEEN 可在子查询内部使用。

3.7.1 SELECT 语句中的子查询使用

SELECT 语句在子查询返回的数据中进行查询。基本语法如下：

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
      (SELECT column_name [, column_name ]
      FROM table1 [, table2 ]
      [WHERE]);
```

示例：

创建表 customer，数据内容如下。

```
postgres=# SELECT * FROM customer_t1;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          3869 | hello         | Grace        |             |   1000
          3869 |               | Grace        |             |
          3869 | hello         |              |             |
          6985 | maps          | Joes         |             |   2200
          9976 | world         | James        |             |   5000
          4421 | Admin         | Local        |             |   3000
(6 rows)
```

在 SELECT 中使用子查询，语句如下。

```
postgres=# SELECT * FROM customer_t1 WHERE c_customer_sk IN (SELECT c_customer_sk
FROM customer_t1 WHERE amount > 2500) ;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          9976 | world         | James        |             |   5000
          4421 | Admin         | Local        |             |   3000
(2 rows)
```

3.7.2 INSERT 语句中的子查询使用

子查询也可以与 INSERT 语句一起使用。INSERT 语句使用子查询返回的数据插入到另

一个表中。基本语法如下：

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
SELECT [ *|column1 [, column2 ] ]
FROM table1 [, table2 ]
[ WHERE VALUE OPERATOR ]
```

示例：

创建表 customer_bak，表结构与 customer_t1 一致。

```
postgres=# CREATE TABLE customer_bak
(
    c_customer_sk             integer,
    c_customer_id             char(5),
    c_first_name              char(6),
    c_last_name               char(8),
    Amount                    integer
);
CREATE TABLE
```

将表 customer_t1 中的数据插入 customer_bak。

```
postgres=# INSERT INTO customer_bak SELECT * FROM customer_t1  WHERE
c_customer_sk  IN (SELECT c_customer_sk  FROM customer_t1) ;
INSERT 0 6
```

插入数据后的 customer_bak 的表如下：

```
postgres=# SELECT * FROM customer_bak;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          3869 | hello         | Grace        |             |   1000
          3869 |               | Grace        |             |
          3869 | hello         |              |             |
          6985 | maps          | Joes         |             |   2200
          9976 | world         | James        |             |   5000
          4421 | Admin         | Local        |             |   3000
(6 rows)
```

3.7.3 UPDATE 语句中的子查询使用

通过 UPDATE 语句使用子查询时，表中多个列被更新。基本语法如下：

```
UPDATE table
SET column_name = new_value
```

```
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME
   [ WHERE ])
```

示例：

把表 customer_t1 中所有 c_customer_sk 大于 4000 的客户的 amount 更新为原来的 0.50 倍：

```
postgres=# UPDATE customer_t1 SET amount = amount * 0.50 WHERE c_customer_sk IN
(SELECT c_customer_sk FROM customer_bak WHERE c_customer_sk > 5000 );
UPDATE 2
```

更新影响 2 行，更新后表 customer_t1 数据如下：

```
postgres=# SELECT *  FROM  customer_t1;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          3869 | hello         | Grace        |             |   1000
          3869 |               | Grace        |             |
          3869 | hello         |              |             |
          4421 | Admin         | Local        |             |   3000
          6985 | maps          | Joes         |             |   1100
          9976 | world         | James        |             |   2500
(6 rows)
```

3.7.4 DELETE 语句中的子查询使用

基本语法如下：

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME
   [ WHERE ])
```

示例：

删除表 customer_t1 中所有 c_customer_sk 大于 4000 的客户：

```
postgres=# DELETE FROM customer_t1 WHERE c_customer_sk IN (SELECT c_customer_sk
FROM customer_bak WHERE  c_customer_sk > 5000 );
DELETE 2
```

删除影响 2 行，删除后的表 customer_t1 数据如下：

```
postgres=# SELECT *  FROM  customer_t1;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          3869 | hello         | Grace        |             |   1000
          3869 |               | Grace        |             |
          3869 | hello         |              |             |
          4421 | Admin         | Local        |             |   3000
(4 rows)
```

3.8 SQL 语法

3.8.1 ABORT

功能描述

回滚当前事务并且撤销所有当前事务中所做的更改。

作用等同于 ROLLBACK，早期 SQL 有用 ABORT，现在推荐使用 ROLLBACK。

注意事项

在事务外部执行 ABORT 语句不会影响事务的执行，但是会抛出一个 NOTICE 信息。

语法格式

```
ABORT [ WORK | TRANSACTION ] ;
```

参数说明

- WORK | TRANSACTION

  可选关键字，除了增加可读性没有其他任何作用。

示例

```
--创建表 customer_demographics_t1。
postgres=# CREATE TABLE customer_demographics_t1
(
    CD_DEMO_SK                INTEGER               NOT NULL,
    CD_GENDER                 CHAR(1)                       ,
    CD_MARITAL_STATUS         CHAR(1)                       ,
    CD_EDUCATION_STATUS       CHAR(20)                      ,
    CD_PURCHASE_ESTIMATE      INTEGER                       ,
    CD_CREDIT_RATING          CHAR(10)                      ,
    CD_DEP_COUNT              INTEGER                       ,
    CD_DEP_EMPLOYED_COUNT     INTEGER                       ,
    CD_DEP_COLLEGE_COUNT      INTEGER
```

```
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
;

--插入记录。
postgres=# INSERT INTO customer_demographics_t1 VALUES(1920801,'M', 'U',
'DOCTOR DEGREE', 200, 'GOOD', 1, 0,0);

--开启事务。
postgres=# START TRANSACTION;

--更新字段值。
postgres=# UPDATE customer_demographics_t1 SET cd_education_status='Unknown';

--终止事务，上面所执行的更新会被撤销掉。
postgres=# ABORT;

--查询数据。
postgres=# SELECT * FROM customer_demographics_t1 WHERE cd_demo_sk = 1920801;
cd_demo_sk | cd_gender | cd_marital_status | cd_education_status   |
cd_purchase_estimate | cd_credit_rating | cd_dep_count | cd_dep_employed_count
| cd_dep_college_count
-----------+-----------+-------------------+----------------------+---------
------------+------------------+-------------+----------------------+-----
-----------------
    1920801 | M         | U                 | DOCTOR DEGREE        |
200 | GOOD             |            1 |                     0 |
0
(1 row)

--删除表。
postgres=# DROP TABLE customer_demographics_t1;
```

相关命令

SET TRANSACTION，COMMIT| END，ROLLBACK

3.8.2 ALTER AGGREGATE

功能描述

修改一个聚合函数的定义。

注意事项

要使用 ALTER AGGREGATE，你必须是该聚合函数的所有者。要改变一个聚合函数的模式，你必须在新模式上有 CREATE 权限。要改变所有者，你必须是新所有角色的一个直接或间接成员，并且该角色必须在聚合函数的模式上有 CREATE 权限。（这些限制强制了修改该所有者不会做任何通过删除和重建聚合函数不能做的事情。不过，具有 SYSADMIN 权限用户可以用任何方法任意更改聚合函数的所属关系）。

语法格式

```
ALTER AGGREGATE name ( argtype [ , ... ] ) RENAME TO new_name
ALTER AGGREGATE name ( argtype [ , ... ] ) OWNER TO new_owner
ALTER AGGREGATE name ( argtype [ , ... ] ) SET SCHEMA new_schema
```

参数说明

- name

  现有的聚合函数的名称（可以有模式修饰）。

- argtype

  聚合函数操作的输入数据类型。要引用一个零参数聚合函数，可以写入*代替输入数据类型列表。

- new_name

  聚合函数的新名字。

- new_owner

  聚合函数的新所有者。

- new_schema

  聚合函数的新模式。

示例

把一个接受 integer 类型参数的聚合函数 myavg 重命名为 my_average ：

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

把一个接受 integer 类型参数的聚合函数 myavg 的所有者改为 joe ：

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

把一个接受 integer 类型参数的聚合函数 myavg 移动到模式 myschema 里：

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

兼容性

SQL 标准里没有 ALTER AGGREGATE 语句。

功能描述

修改统一审计策略。

3.8.3 ALTER AUDIT POLICY

注意事项

只有 poladmin、sysadmin 或初始用户才能进行此操作。

需要打开 enable_security_policy 开关统一审计策略才可以生效，开关打开方式请参考《GBase 8s V8.8.5 5.0.0_数据库参考手册》中“安全配置”章节。

语法格式

```
ALTER AUDIT POLICY [ IF EXISTS ] policy_name { ADD | REMOVE }
{ [ privilege_audit_clause ] [ access_audit_clause ] };
ALTER AUDIT POLICY [ IF EXISTS ] policy_name MODIFY ( filter_group_clause );
ALTER AUDIT POLICY [ IF EXISTS ] policy_name DROP FILTER;
ALTER AUDIT POLICY [ IF EXISTS ] policy_name COMMENTS policy_comments;
ALTER AUDIT POLICY [ IF EXISTS ] policy_name { ENABLE | DISABLE };
```

- privilege_audit_clause：

  PRIVILEGES { DDL | ALL }

- access_audit_clause：

  ACCESS { DML | ALL }

- filter_group_clause：

  FILTER ON { (FILTER_TYPE (filter_value [, ...])) [, ...] }

参数说明

- policy_name

  审计策略名称，需要唯一，不可重复。

  取值范围：字符串，要符合标识符的命名规范。

- DDL

指的是针对数据库执行如下操作时进行审计，目前支持：CREATE、ALTER、DROP、ANALYZE、COMMENT、GRANT、REVOKE、SET、SHOW、LOGIN_ANY、LOGIN_FAILURE、LOGIN_SUCCESS、LOGOUT。

- ALL

  指的是上述 DDL 支持的所有对数据库的操作。

- DML

  指的是针对数据库执行如下操作时进行审计，目前支持：SELECT、COPY、DEALLOCATE、DELETE、EXECUTE、INSERT、PREPARE、REINDEX、TRUNCATE、UPDATE。

- FILTER_TYPE

  指定审计策略的过滤信息，过滤类型包括：IP、ROLES、APP。

- filter_value

  指具体过滤信息内容。

- policy_comments

  用于记录策略相关的描述信息。

- ENABLE|DISABLE

  可以打开或关闭统一审计策略。若不指定 ENABLE|DISABLE，语句默认为 ENABLE。

示例

请参考 CREATE AUDIT POLICY 的示例。

相关命令

CREATE AUDIT POLICY，DROP AUDIT POLICY。

3.8.4 ALTER DATABASE

功能描述

修改数据库的属性，包括它的名称、所有者、连接数限制、对象隔离属性等。

注意事项

只有数据库的所有者或者被授予了数据库 ALTER 权限的用户才能执行 ALTER

DATABASE 命令，系统管理员默认拥有此权限。针对所要修改属性的不同，还有以下权限约束：

修改数据库名称，必须拥有 CREATEDB 权限。

修改数据库所有者，当前用户必须是该 database 的所有者或者系统管理员，必须拥有 CREATEDB 权限，且该用户是新所有者角色的成员。

修改数据库默认表空间，必须拥有新表空间的 CREATE 权限。这个语句会从物理上将一个数据库原来缺省表空间上的表和索引移至新的表空间。注意不在缺省表空间的表和索引不受此影响。

不能重命名当前使用的数据库，如果需要重新命名，须连接至其他数据库上。

语法格式

- 修改数据库的最大连接数。

```
ALTER DATABASE database_name [ [ WITH ] CONNECTION LIMIT connlimit ];
```

- 修改数据库名称。

```
ALTER DATABASE database_name RENAME TO new_name;
```

- 修改数据库所属者。

```
ALTER DATABASE database_name OWNER TO new_owner;
```

- 修改数据库默认表空间。

```
ALTER DATABASE database_name SET TABLESPACE new_tablespace;
```

- 修改数据库指定会话参数值。

```
ALTER DATABASE database_name SET configuration_parameter { { TO | = } { value |
DEFAULT } | FROM CURRENT };
```

- 数据库配置参数重置。

```
ALTER DATABASE database_name RESET { configuration_parameter | ALL };
```

- 修改数据库对象隔离属性。

```
ALTER DATABASE database_name [ WITH ] { ENABLE | DISABLE } PRIVATE OBJECT;
```

说明

修改数据库的对象隔离属性时须连接至该数据库，否则无法更改。

新创建的数据库，对象隔离属性默认是关闭的。当开启数据库对象隔离属性后，普通用户只能查看有权访问的对象（表、函数、视图、字段等）。对象隔离特性对管理员用户不生效，当开启对象隔离特性后，管理员也可以查看到全量的数据库对象。

参数说明

- database_name

  需要修改属性的数据库名称。

  取值范围：字符串，要符合标识符的命名规范。

- connlimit

  数据库可以接收的最大并发连接数（管理员用户连接除外）。

  取值范围：整数，建议填写 1~50 的整数。-1（缺省）表示没有限制。

- new_name

  数据库的新名称。

  取值范围：字符串，要符合标识符的命名规范。

- new_owner

  数据库的新所有者。

  取值范围：字符串，有效的用户名。

- new_tablespace

  数据库新的默认表空间，该表空间为数据库中已经存在的表空间。默认的表空间为 pg_default。

  取值范围：字符串，有效的表空间名。

- configuration_parameter

  把指定的数据库会话参数值设置为给定的值。。

  取值范围：

  - DEFAULT
  - OFF
  - RESET

如果 value 是 DEFAULT 或者 RESET，则在新的会话中使用系统的缺省设置。OFF 关闭设置。

- FROM CURRENT

  根据当前会话连接的数据库设置该参数的值。

- RESET configuration_parameter

  重置指定的数据库会话参数值。

- RESET ALL

  重置全部的数据库会话参数值。修改的数据库会话参数值，将在下一次会话中生效。

  说明

  修改数据库默认表空间，会将旧表空间中的所有表和索引转移到新表空间中，该操作不会影响其他非默认表空间中的表和索引。

示例

请参考 CREATE DATABASE 的示例。

相关命令

CREATE DATABASE，DROP DATABASE

3.8.5 ALTER DATA SOURCE

功能描述

修改 Data Source 对象的属性和内容。

属性有：名称和属主；内容有：类型、版本和连接选项。

注意选项

只有初始用户/系统管理员/属主才拥有修改 Data Source 的权限。

修改属主时，新的属主用户必须是初始用户或系统管理员。

当在 OPTIONS 中出现 password 选项时，需要保证 GBase 8s 每个节点的 $GAUSSHOME/bin 目录下存在 datasource.key.cipher 和 datasource.key.rand 文件，如果不存在这两个文件，请使用 gs_guc 工具生成并使用 gs_ssh 工具发布到每个节点的 $GAUSSHOME/bin 目录下。

语法格式

```
ALTER DATA SOURCE src_name
    [TYPE  'type_str']
    [VERSION  {'version_str' | NULL}]
    [OPTIONS ( {[ ADD | SET | DROP ] optname ['optvalue']} [, ...] )];
ALTER DATA SOURCE src_name RENAME TO src_new_name;
ALTER DATA SOURCE src_name OWNER TO new_owner;
```

参数说明

- src_name

  待修改的 Data Source 的名称。

  取值范围：字符串，需要符合标识符的命名规范。

- TYPE

  将 Data Source 原来的 TYPE 修改为指定值。

  取值范围：空串或非空字符串。

- VERSION

  将 Data Source 原来的 VERSION 修改为指定值。

  取值范围：空串或非空字符串或 NULL。

- OPTIONS

  修改 OPTIONS 中的字段：增加（ADD）、修改（SET）、删除（DROP），且字段名称 optname 需唯一，具体要求如下：

  增加字段：ADD 可以省略，待增加字段不能已经存在了；

  修改字段：SET 不可省略，待修改字段必须存在；

  删除字段：DROP 不可省略，待删除字段必须存在，且不能指定 optvalue；

- src_new_name

  新的 Data Source 名称。

  取值范围：字符串，需符合标识符命名规范。

- new_user

对象的新属主。

取值范围：字符串，有效的用户名。

示例

```
--创建一个空 Data Source 对象。
postgres=# CREATE DATA SOURCE ds_test1;

--修改名称。
postgres=# ALTER DATA SOURCE ds_test1 RENAME TO ds_test;

--修改属主。
postgres=# CREATE USER user_test1 IDENTIFIED BY 'Gs@123456';
postgres=# ALTER USER user_test1 WITH SYSADMIN;
postgres=# ALTER DATA SOURCE ds_test OWNER TO user_test1;

--修改 TYPE 和 VERSION。
postgres=# ALTER DATA SOURCE ds_test TYPE 'MPPDB_TYPE' VERSION 'XXX';

--添加字段。
postgres=# ALTER DATA SOURCE ds_test OPTIONS (add dsn 'gaussdb', username
'test_user');

--修改字段。
postgres=# ALTER DATA SOURCE ds_test OPTIONS (set dsn 'unknown');

--删除字段。
postgres=# ALTER DATA SOURCE ds_test OPTIONS (drop username);

--删除 Data Source 和 user 对象。
postgres=# DROP DATA SOURCE ds_test;
postgres=# DROP USER user_test1;
```

相关命令

CREATE DATA SOURCE，DROP DATA SOURCE

3.8.6 ALTER DEFAULT PRIVILEGES

功能描述

设置应用于将来创建的对象的权限（这不会影响分配到已有对象中的权限）。

注意事项

目前只支持表（包括视图）、序列、函数、类型、密态数据库客户端主密钥和列加密密钥的权限更改。

语法格式

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke;
```

其中 abbreviated_grant_or_revoke 子句用于指定对哪些对象进行授权或回收权限。

```
grant_on_tables_clause
  | grant_on_sequences_clause
  | grant_on_functions_clause
  | grant_on_types_clause
  | grant_on_client_master_keys_clause
  | grant_on_column_encryption_keys_clause
  | revoke_on_tables_clause
  | revoke_on_sequences_clause
  | revoke_on_functions_clause
  | revoke_on_types_clause
  | revoke_on_client_master_keys_clause
  | revoke_on_column_encryption_keys_clause
```

其中 grant_on_tables_clause 子句用于对表授权。

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | ALTER | DROP
| COMMENT | INDEX | VACUUM }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

其中 grant_on_sequences_clause 子句用于对序列授权。

```
GRANT { { SELECT | UPDATE | USAGE | ALTER | DROP | COMMENT }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

其中 grant_on_functions_clause 子句用于对函数授权。

```
GRANT { { EXECUTE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
```

```
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

其中 grant_on_types_clause 子句用于对类型授权。

```
GRANT { { USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

其中 grant_on_client_master_keys_clause 子句用于对客户端主密钥授权。

```
GRANT { { USAGE  | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON CLIENT_MASTER_KEYS
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

其中 grant_on_column_encryption_keys_clause 子句用于对列加密密钥授权。

```
GRANT { { USAGE | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON COLUMN_ENCRYPTION_KEYS
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

其中 revoke_on_tables_clause 子句用于回收表对象的权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | ALTER | DROP
| COMMENT | INDEX | VACUUM }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

其中 revoke_on_sequences_clause 子句用于回收序列的权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE | USAGE | ALTER | DROP | COMMENT }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

其中 revoke_on_functions_clause 子句用于回收函数的权限。

```
REVOKE [ GRANT OPTION FOR ]
    { {EXECUTE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
```

```
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

其中 revoke_on_types_clause 子句用于回收类型的权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

其中 revoke_on_client_master_keys_clause 子句用于回收客户端主密钥的权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON CLIENT_MASTER_KEYS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

其中 revoke_on_column_encryption_keys_clause 子句用于回收列加密密钥的权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON COLUMN_ENCRYPTION_KEYS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

参数说明

- target_role

  已有角色的名称。如果省略 FOR ROLE/USER，则缺省值为当前角色/用户。

  取值范围：已有角色的名称。

- schema_name

  现有模式的名称。

  target_role 必须有 schema_name 的 CREATE 权限。

  取值范围：现有模式的名称。

- role_name

  被授予或者取消权限角色的名称。

  取值范围：已存在的角色名称。

须知

如果想删除一个被赋予了默认权限的角色，有必要恢复改变的缺省权限或者使用 DROP OWNED BY 来为角色脱离缺省的权限记录。

示例

```
--将创建在模式 tpcds 里的所有表（和视图）的 SELECT 权限授予每一个用户。
postgres=# ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds GRANT SELECT ON TABLES TO
PUBLIC;

--创建用户普通用户 jack。
postgres=# CREATE USER jack PASSWORD 'xxxxxxxxx';

--将 tpcds 下的所有表的插入权限授予用户 jack。
postgres=# ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds GRANT INSERT ON TABLES TO jack;

--撤销上述权限。
postgres=# ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds REVOKE SELECT ON TABLES FROM
PUBLIC;
postgres=# ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds REVOKE INSERT ON TABLES FROM
jack;

--删除用户 jack。
postgres=# DROP USER jack;
```

相关命令

GRANT，REVOKE

3.8.7 ALTER DIRECTORY

功能描述

对 directory 属性进行修改。

注意事项

目前只支持修改 directory 属主。

当 enable_access_server_directory=off 时，只允许初始用户修改 directory 属主；当 enable_access_server_directory=on 时，具有 SYSADMIN 权限的用户和 directory 对象的属主可以修改 directory，且要求该用户是新属主的成员。

语法格式

```
ALTER DIRECTORY directory_name
    OWNER TO new_owner;
```

参数描述

- directory_name

需要修改的目录名称，范围为已经存在的目录名称。

示例

```
--创建目录。
postgres=# CREATE OR REPLACE DIRECTORY  dir  as '/tmp/';

--修改目录的 owner。
postgres=# ALTER DIRECTORY dir OWNER TO system;

--删除目录。
postgres=# DROP DIRECTORY dir;
```

相关命令

CREATE DIRECTORY，DROP DIRECTORY

3.8.8 ALTER EXTENSION

功能描述

修改插件扩展。

注意事项

ALTER EXTENSION 修改一个已安装的扩展的定义。这里有几种方式：

- UPDATE

这种方式更新这个扩展到一个新的版本。这个扩展必须满足一个适用的更新脚本（或者一系列脚本）这样就能修改当前安装版本到一个要求的版本。

- SET SCHEMA

这种方式移动扩展对象到另一个模式。这个扩展必须 relocatable 才能使命令成功。

- ADD member_object

这种方式添加一个已存在对象到扩展。这主要在扩展更新脚本上有用。这个对象接着会被视为扩展的成员。该对象只能通过删除扩展来级联删除。

- DROP member_object

这个方式从扩展上移除一个成员对象。这主要在扩展更新脚本上有用。这个对象没有被删除，只是从扩展里移除。

必须在已拥有的扩展上来使用 ALTER EXTENSION。ADD/DROP 则是要求添加/删除对象的所有权。

语法格式

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object

where member_object is:

  FOREIGN TABLE object_name |
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
  [ PROCEDURAL ] LANGUAGE object_name |
  SCHEMA object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TYPE object_name |
  VIEW object_name
```

参数说明

- name

  已安装扩展的名称。

- new_version

  扩展的新版本。可以通过被标识符和字面字符重写。如果不指定的扩展的新版本，ALTER EXTENSION UPDATE 会更新到扩展的控制文件中显示的默认版本。

- new_schema

  扩展的新模式。

- object_name

  function_name

从扩展里被被添加或移除的对象的名称。包含表、函数、文本搜索对象、类型和能被模式合格的视图的名称。

- argmode

这个函数参数的模型：IN、OUT、INOUT 或者 VARIADIC。如果省略的话，默认值为 IN。ALTER EXTENSION 不关心 OUT 参数，因为确认函数的一致性只需要输入参数，因此列出 IN、INOUT 和 VARIADIC 参数就足够了。

- argname

函数参数的名称。ALTER EXTENSION 不关心参数名称，确认函数的一致性只需要参数数据类型。

- argtype

函数参数的数据类型（可以有模式修饰）。

示例

```
--更新 hstore 扩展到版本 2.0:
ALTER EXTENSION hstore UPDATE TO '2.0';
--更新 hstore 扩展的模式为 utils:
ALTER EXTENSION hstore SET SCHEMA utils;
--添加一个已存在的函数给 hstore 扩展:
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

3.8.9 ALTER EVENT TRIGGER

功能描述

修改事件触发器。

注意事项

只有系统管理员或者超级用户才有权限对事件触发器进行修改。

语法格式

```
ALTER EVENT TRIGGER name DISABLE
ALTER EVENT TRIGGER name ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER } ALTER
EVENT TRIGGER name RENAME TO new_name
```

参数说明

- name

要修改的事件触发器名称。

取值范围：已存在的事件触发器。

- new name

修改后的新名称。

取值范围：符合标识符命名规范的字符串，最大长度不超过 63 个字符，且不能与所在表 上其他事件触发器同名。

示例

请参见 CREATE EVENT TRIGGER 的示例。

3.8.10 ALTER FOREIGN DATA WRAPPER

功能描述

修改外部数据包装器的定义。

语法格式

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [,...] ) ]
```

参数说明

- name

要修改的外部数据包装器名。

- HANDLER handler_function

为外部数据包装器指定一个新的处理器函数。

- NO HANDLER

指定外部数据包装器不再具有处理器函数。

须知：

不能访问使用没有处理器的外部数据包装器的外部表。

- VALIDATOR validator_function

为外部数据包装器指定一个新的验证器函数。

须知：

在修改验证器函数后，外部数据包装器，服务器和用户映射的选项可能会失效。在使用外部数据包装器之前，用户应确保这些选项是正确的。

- NO VALIDATOR

  指定外部数据包装器不再具有验证器函数。

- OPTIONS ([ADD | SET | DROP] option ['value'] [,…])

  外部数据包装器的修改选项。添加，设置和删除指定要执行的操作。如果未明确指定操作，则假定添加。选项名称不许是唯一的；如果有的话，使用外部数据包装器的验证器函数验证名称和值。

示例

```
--创建外部包装器 dbi
postgres=# CREATE FOREIGN DATA WRAPPER dbi OPTIONS (debug 'true');
--修改外部包装器 dbi，添加选项 foo，删除选项 debug
postgres=# ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP debug);
--修改外部数据包装器 dbi 的验证器为 myvalidator
postgres=# ALTER FOREIGN DATA WRAPPER dbi VALIDATOR file_fdw_validator;
```

3.8.11 ALTER FOREIGN TABLE

功能描述

对外表进行修改。

语法格式

```
ALTER FOREIGN TABLE [ IF EXISTS ]  table_name
    OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ]);
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    OWNER TO new_owner;
```

参数说明

- table_name

  需要修改的外表名称。

  取值范围：已存在的外表名。

- option

  改变外表或者外表字段的选项。ADD、SET 和 DROP 指定执行的操作。如果没有显式

设置，那么默认为 ADD。选项的名字不允许重复（尽管表选项和表字段选项可以有相同的名字）。选项的名称和值也会通过外部数据封装器的类库进行校验。

- oracle_fdw 支持的 options 包括：
  - table

oracle server 侧的表名。需要同 oracle 系统表中记录的表名完全一致，通常是由大写字符组成。

  - schema

表所对应的 schema（或 owner）。需要与 oracle 系统表中记录的表名完全一致，通常是由大写字符组成。

- mysql_fdw 支持的 options 包括：
  - dbname

MySQL 的 database 名称。

  - table_name

MySQL 侧的表名。

- postgres_fdw 支持的 options 包括：
  - schema_name

远端 server 的 schema 名称。如果不指定的话，将使用外表自身的 schema 名称作为远端的 schema 名称。

  - table_name

远端 server 的表名。如果不指定的话，将使用外表自身的表名作为远端的表名。

  - column_name

远端 server 的表的列名。如果不指定的话，将使用外表自身的列名作为远端的的表的列名。

- file_fdw 支持的 options 包括：
  - filename

指定要读取的文件，必需的参数，且必须是一个绝对路径名。

  - format

远端 server 的文件格式，支持 text/csv/binary/fixed 四种格式，和 COPY 语句的 FORMAT 选项相同。

- header

指定的文件是否有标题行，与 COPY 语句的 HEADER 选项相同。

- delimiter

指定文件的分隔符，与 COPY 的 DELIMITER 选项相同。

- quote

指定文件的引用字符，与 COPY 的 QUOTE 选项相同。

- escape

指定文件的转义字符，与 COPY 的 ESCAPE 选项相同。

- null

指定文件的 null 字符串，与 COPY 的 NULL 选项相同。

- encoding

指定文件的编码，与 COPY 的 ENCODING 选项相同。

- force_not_null

这是一个布尔选项。如果为真，则声明字段的值不应该匹配空字符串（也就是，文件级别 null 选项）。与 COPY 的 FORCE_NOT_NULL 选项里的字段相同。

说明：file_fdw 更多使用请参见《GBase 8s V8.8.5_5.0.0_数据库管理指南》中“file_fdw”章节。

- value

option 的新值。

相关命令

CREATE FOREIGN TABLE，DROP FOREIGN TABLE

3.8.12 ALTER FUNCTION

功能描述

修改自定义函数的属性。

注意事项

只有函数的所有者或者被授予了函数 ALTER 权限的用户才能执行 ALTER FUNCTION 命令，系统管理员默认拥有该权限。针对所要修改属性的不同，还有以下权限约束：

如果函数中涉及对临时表相关的操作，则无法使用 ALTER FUNCTION。

修改函数的所有者或修改函数的模式，当前用户必须是该函数的所有者或者系统管理员，且该用户是新所有者角色的成员。

只有系统管理员和初始化用户可以将 function 的 schema 修改成 public。

语法格式

修改自定义函数的附加参数。

```
ALTER FUNCTION function_name ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    action [ ... ] [ RESTRICT ];

where action can be:
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
|    {IMMUTABLE | STABLE | VOLATILE}
|    {NOT FENCED | FENCED}
|    [ NOT ] LEAKPROOF
|    {[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER}
|    AUTHID { DEFINER | CURRENT_USER }
|    COST execution_cost
|    ROWS result_rows
|    SET configuration_parameter {{ TO | = } { value | DEFAULT }| FROM CURRENT}
|    RESET {configuration_parameter| ALL}
```

其中附加参数 action 子句语法为。

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
 | {IMMUTABLE | STABLE | VOLATILE}
 | {NOT FENCED | FENCED}
 | [ NOT ] LEAKPROOF
 | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
 | AUTHID { DEFINER | CURRENT_USER }
 | COST execution_cost
 | ROWS result_rows
 | SET configuration_parameter { { TO | = } { value | DEFAULT }| FROM CURRENT}
 | RESET {configuration_parameter | ALL}
```

修改自定义函数的名称。

```
ALTER FUNCTION funname ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    RENAME TO new_name;
```

修改自定义函数的所属者。

```
ALTER FUNCTION funname ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    OWNER TO new_owner;
```

修改自定义函数的模式。

```
ALTER FUNCTION funname ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    SET SCHEMA new_schema;
```

参数说明

- function_name

  要修改的函数名称。

  取值范围：已存在的函数名。

- argmode

  标识该参数是输入、输出参数。

  取值范围：IN/OUT/INOUT/VARIADIC。

- argname

  参数名称。

  取值范围：字符串，符合标识符命名规范。

- argtype

  函数参数的类型。

- CALLED ON NULL INPUT

  表明该函数的某些参数是 NULL 的时候可以按照正常的方式调用。缺省时与指定此参数的作用相同。

- RETURNS NULL ON NULL INPUT | STRICT

  STRICT 用于指定如果函数的某个参数是 NULL，此函数总是返回 NULL。如果声明了这个参数，则如果存在 NULL 参数时不会执行该函数；而只是自动假设一个 NULL 结果。

  RETURNS NULL ON NULL INPUT 和 STRICT 的功能相同。

- IMMUTABLE

表示该函数在给出同样的参数值时总是返回同样的结果。

- STABLE

表示该函数不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同 SQL 语句之间变化。

- VOLATILE

表示该函数值可以在一次表扫描内改变，不会做任何优化。

- LEAKPROOF

表示该函数没有副作用，指出参数只包括返回值。LEAKPROOF 只能由系统管理员设置。

- EXTERNAL

（可选）目的是和 SQL 兼容，这个特性适合于所有函数，而不仅是外部函数。

- SECURITY INVOKER | AUTHID CURRENT_USER

表明该函数将以调用它的用户的权限执行。缺省时与指定此参数的作用相同。

SECURITY INVOKER 和 AUTHID CURRENT_USER 的功能相同。

- SECURITY DEFINER | AUTHID DEFINER

声明该函数将以创建它的用户的权限执行。

AUTHID DEFINER 和 SECURITY DEFINER 的功能相同。

- COST execution_cost

用来估计函数的执行成本。

execution_cost 以 cpu_operator_cost 为单位。

取值范围：正数

- ROWS result_rows

估计函数返回的行数。用于函数返回的是一个集合。

取值范围：正数，默认值是 1000 行。

- configuration_parameter | value

把指定的数据库会话参数值设置为给定的值。如果 value 是 DEFAULT 或者 RESET，则在新的会话中使用系统的缺省设置。OFF 关闭设置。

取值范围：字符串

DEFAULT

OFF

- RESET

  指定默认值。

- from current

  取当前会话中的值设置为 configuration_parameter 的值。

- new_name

  函数的新名称。要修改函数的所属模式，必须拥有新模式的 CREATE 权限。

  取值范围：字符串，符合标识符命名规范。

- new_owner

  函数的新所有者。要修改函数的所有者，新所有者必须拥有该函数所属模式的 CREATE 权限。

  取值范围：已存在的用户角色。

- new_schema

  函数的新模式。

  取值范围：已存在的模式。

示例

请参见 CREATE FUNCTION 的示例。

相关命令

CREATE FUNCTION，DROP FUNCTION

3.8.13 ALTER GLOBAL CONFIGURATION

功能描述

新增、修改系统表 gs_global_config，增加 key-value 值。

注意事项

仅支持数据库初始用户运行此命令。

不支持创建修改关键字为 weak_password。

语法格式

```
ALTER GLOBAL CONFIGURATION with(paraname=value, paraname=value...);
```

参数说明

参数名称和参数值都是 text 类型。

3.8.14 ALTER GROUP

功能描述

修改一个用户组的属性。

注意事项

ALTER GROUP 是 ALTER ROLE 的别名，非 SQL 标准语法，不推荐使用，建议用户直接使用 ALTER ROLE 替代。

语法格式

向用户组中添加用户。

```
ALTER GROUP group_name ADD USER user_name [, ... ];
```

从用户组中删除用户。

```
ALTER GROUP group_name DROP USER user_name [, ... ];
```

修改用户组的名称。

```
ALTER GROUP group_name RENAME TO new_name;
```

参数说明

请参考 ALTER ROLE 的参数说明。

示例

向用户组中添加用户。

```
postgres=# ALTER GROUP super_users ADD USER lche, jim;
```

从用户组中删除用户。

```
postgres=# ALTER GROUP super_users DROP USER jim;
```

修改用户组的名称。

```
postgres=# ALTER GROUP super_users RENAME TO normal_users;
```

相关命令

ALTER GROUP，DROP GROUP，ALTER ROLE

3.8.15 ALTER INDEX

功能描述

ALTER INDEX 用于修改现有索引的定义。

它有几种子形式：

- IF EXISTS

  如果指定的索引不存在，则发出一个 notice 而不是 error。

- RENAME TO

  只改变索引的名称。对存储的数据没有影响。

- SET TABLESPACE

  这个选项会改变索引的表空间为指定表空间，并且把索引相关的数据文件移动到新的表空间里。

- SET ({ STORAGE_PARAMETER = value } [, ···])

  改变索引的一个或多个索引方法特定的存储参数。 需要注意的是索引内容不会被这个命令立即修改，根据参数的不同，可能需要使用 REINDEX 重建索引来获得期望的效果。

- RESET ({ storage_parameter } [, ···])

  重置索引的一个或多个索引方法特定的存储参数为缺省值。与 SET 一样，可能需要使用 REINDEX 来完全更新索引。

- [MODIFY PARTITION index_partition_name] UNUSABLE

  用于设置表或者索引分区上的索引不可用。

- REBUILD [PARTITION index_partition_name]

  用于重建表或者索引分区上的索引。

- RENAME PARTITION

  用于重命名索引分区。

- MOVE PARTITION

用于修改索引分区的所属表空间。

注意事项

只有索引的所有者或者拥有索引所在表的 INDEX 权限的用户有权限执行此命令，系统管理员默认拥有此权限。

语法格式

重命名表索引的名称。

```
ALTER INDEX [ IF EXISTS ] index_name
    RENAME TO new_name;
```

修改表索引的所属空间。

```
ALTER INDEX [ IF EXISTS ] index_name
    SET TABLESPACE tablespace_name;
```

修改表索引的存储参数。

```
ALTER INDEX [ IF EXISTS ] index_name
    SET ( {storage_parameter = value} [, ... ] );
```

重置表索引的存储参数。

```
ALTER INDEX [ IF EXISTS ] index_name
    RESET ( storage_parameter [, ... ] ) ;
```

设置表索引或索引分区不可用。

```
ALTER INDEX [ IF EXISTS ] index_name
    [ MODIFY PARTITION index_partition_name ] UNUSABLE;
```

说明：列存表不支持该语法。

重建表索引或索引分区。

```
ALTER INDEX index_name
    REBUILD [ PARTITION index_partition_name ];
```

重命名索引分区。

```
ALTER INDEX [ IF EXISTS ] index_name
    RENAME PARTITION index_partition_name TO new_index_partition_name;
```

修改索引分区的所属表空间。

```
ALTER INDEX [ IF EXISTS ] index_name
    MOVE PARTITION index_partition_name TABLESPACE new_tablespace;
```

参数说明

- index_name

  要修改的索引名。

- new_name

  新的索引名。

  取值范围：字符串，且符合标识符命名规范。

- tablespace_name

  表空间的名称。

  取值范围：已存在的表空间。

- storage_parameter

  索引方法特定的参数名。

- value

  索引方法特定的存储参数的新值。根据参数的不同，这可能是一个数字或单词。

- new_index_partition_name

  新索引分区名。

- index_partition_name

  索引分区名。

- new_tablespace

  新表空间。

示例

请参见 CREATE INDEX 的示例。

相关命令

CREATE INDEX，DROP INDEX，REINDEX

3.8.16 ALTER LANGUAGE

功能描述

修改一个过程语言的定义。暂不支持修改过程语言。

语法格式

```
ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO new_name ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO new_owner
```

参数说明

- name

  语言的名字。

- new_name

  语言的新名字。

- new_owner

  语言的新的所有者。

兼容性

SQL 标准里没有 ALTER LANGUAGE 语句。

3.8.17 ALTER LARGE OBJECT

功能描述

ALTER LARGE OBJECT 用于更改一个 large object 的定义。它的唯一的功能是分配一个新的所有者。

注意事项

使用 ALTER LARGE OBJECT 必须是系统管理员或者是其所有者。

语法格式

```
ALTER LARGE OBJECT large_object_oid
    OWNER TO new_owner;
```

参数说明

- large_object_oid

要被变 large object 的 OID 。

取值范围：已存在的大对象名。

- OWNER TO new_owner

  large object 新的所有者。

  取值范围：已存在的用户名/角色名。

示例

无。

3.8.18 ALTER MASKING POLICY

功能描述

修改脱敏策略。

注意事项

只有 poladmin、sysadmin 或初始用户才能执行此操作。

需要打开 enable_security_policy 开关脱敏策略才可以生效。

语法格式

- 修改策略描述：

```
ALTER MASKING POLICY policy_name COMMENTS policy_comments;
```

- 修改脱敏方式：

```
ALTER MASKING POLICY policy_name [ADD | REMOVE | MODIFY] masking_actions[, ...]*;
```

其中 masking_action:

```
    masking_function ON LABEL(label_name[, ...]*)
where masking_function can be:
{ maskall | randgbaseasking | creditcardmasking | basicemailmasking |
fullemailmasking | shufflemaskexpmasking }
```

- 修改脱敏策略生效场景：

```
ALTER MASKING POLICY policy_name MODIFY ( filter_group_clause );
where filter_group_clause can be:
FILTER ON { ( FILTER_TYPE ( filter_value [, ... ] ) ) [, ... ] }
```

- 移除脱敏策略生效场景，使策略对所用场景生效：

```
ALTER MASKING POLICY policy_name DROP FILTER;
```

- 修改脱敏策略开启/关闭：

```
ALTER MASKING POLICY policy_name [ENABLE | DISABLE];
```

参数说明

- policy_name

  脱敏策略名称，需要唯一，不可重复。

  取值范围：字符串，要符合标识符的命名规范。

- policy_comments

  需要为脱敏策略添加或修改的描述信息。

- masking_function

  指的是预置的八种脱敏方式或者用户自定义的函数，支持模式。

  maskall 不是预置函数，硬编码在代码中，不支持\df 展示。

  预置时脱敏方式如下：

  maskall | randgbaseasking | creditcardmasking | basicemailmasking | fullemailmasking | shufflemasking | alldigitsmasking | regexpmasking

- label_name

  资源标签名称。

- FILTER_TYPE

  指定脱敏策略的过滤信息，过滤类型包括：IP、ROLES、APP。

- filter_value

  指具体过滤信息内容，例如具体的 IP、具体的 APP 名称、具体的用户名。

- ENABLE|DISABLE

  可以打开或关闭脱敏策略。若不指定 ENABLE|DISABLE，语句默认为 ENABLE。

示例

```
--创建 dev_mask 和 bob_mask 用户。
postgres=# CREATE USER dev_mask PASSWORD 'dev@1234';
postgres=# CREATE USER bob_mask PASSWORD 'bob@1234';
```

```
--创建一个表 tb_for_masking
postgres=# CREATE TABLE tb_for_masking(col1 text, col2 text, col3 text);

--创建资源标签标记敏感列 col1
postgres=# CREATE RESOURCE LABEL mask_lb1 ADD COLUMN(tb_for_masking.col1);

--创建资源标签标记敏感列 col2
postgres=# CREATE RESOURCE LABEL mask_lb2 ADD COLUMN(tb_for_masking.col2);

--对访问敏感列 col1 的操作创建脱敏策略
postgres=# CREATE MASKING POLICY maskpol1 maskall ON LABEL(mask_lb1);

--为脱敏策略 maskpol1 添加描述
postgres=# ALTER MASKING POLICY maskpol1 COMMENTS 'masking policy for
tb_for_masking.col1';

--修改脱敏策略 maskpol1, 新增一项脱敏方式
postgres=# ALTER MASKING POLICY maskpol1 ADD randgbaseasking ON LABEL(mask_lb2);

--修改脱敏策略 maskpol1, 移除一项脱敏方式
postgres=# ALTER MASKING POLICY maskpol1 REMOVE randgbaseasking ON
LABEL(mask_lb2);

--修改脱敏策略 maskpol1, 修改一项脱敏方式
postgres=# ALTER MASKING POLICY maskpol1 MODIFY randgbaseasking ON
LABEL(mask_lb1);

--修改脱敏策略maskpol1使之仅对用户dev_mask和bob_mask,客户端工具为psql和gsql,
IP 地址为'10.20.30.40', '127.0.0.0/24'场景生效。
postgres=# ALTER MASKING POLICY maskpol1 MODIFY (FILTER ON ROLES(dev_mask,
bob_mask), APP(psql, gsql), IP('10.20.30.40', '127.0.0.0/24'));

--修改脱敏策略 maskpol1, 使之对所有用户场景生效
postgres=# ALTER MASKING POLICY maskpol1 DROP FILTER;

--禁用脱敏策略 maskpol1
postgres=# ALTER MASKING POLICY maskpol1 DISABLE;
```

相关命令

CREATE MASKING POLICY,DROP MASKING POLICY。

3.8.19 ALTER MATERIALIZED VIEW

功能描述

更改一个现有物化视图的多个辅助属性。

可用于 ALTER MATERIALIZED VIEW 的语句形式和动作是 ALTER TABLE 的一个子集，并且在用于物化视图时具有相同的含义。详见 ALTER TABLE。

注意事项

只有物化视图的所有者有权限执行 ALTER TMATERIALIZED VIEW 命令，系统管理员默认拥有此权限。

不支持更改物化视图结构。

语法格式

修改物化视图的所属用户。

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] mv_name
    OWNER TO new_owner;
```

修改物化视图的列。

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] mv_name
    RENAME [ COLUMN ] column_name TO new_column_name;
```

重命名物化视图。

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] mv_name
    RENAME TO new_name;
```

参数说明

- mv_name

  一个现有物化视图的名称，可以用模式修饰。

  取值范围：字符串，符合标识符命名规范。

- column_name

  一个新的或者现有的列的名称。

  取值范围：字符串，符合标识符命名规范。

- new_column_name

  一个现有列的新名称。

- new_owner

  该物化视图的新拥有者的用户名。

- new_name

  该物化视图的新名称。

示例

```
--把物化视图 foo 重命名为 bar。
postgres=# ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

相关命令

CREATE MATERIALIZED VIEW，CREATE INCREMENTAL MATERIALIZED VIEW，DROP MATERIALIZED VIEW，REFRESH INCREMENTAL MATERIALIZED VIEW，REFRESH MATERIALIZED VIEW

3.8.20 ALTER OPERATOR

功能描述

修改一个操作符的定义。

注意事项

ALTER OPERATOR 改变一个操作符的定义。目前唯一能用的功能是改变操作符的所有者。

要使用 ALTER OPERATOR，你必须是该操作符的所有者。要修改所有者，你还必须是新的所有角色的直接或间接成员，并且该成员必须在此操作符的模式上有 CREATE 权限。（这些限制强制了修改该所有者不会做任何通过删除和重建操作符不能做的事情。不过，具有 AYSADMIN 权限用户可以以任何方式修改任意操作符的所有权。）

语法格式

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } ) OWNER TO
new_owner
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } ) SET SCHEMA
new_schema
```

参数说明

- name

  一个现有操作符的名字。

- left_type

  操作符的左操作数的数据类型；如果没有左操作数，那么写 NONE。

- right_type

  操作符的右操作数的数据类型；如果没有右操作数，那么写 NONE。

- new_owner

  操作符的新所有者。

- new_schema

  操作符的新模式名。

示例

改变一个用于 text 的用户定义操作符 a @@ b：

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

兼容性

SQL 标准里没有 ALTER OPERATOR 语句。

3.8.21 ALTER PUBLICATION

功能描述

更改发布 PUBLICATION 的属性。

注意事项

发布的属主和系统管理员才能执行 ALTER PUBLICATION。新所有者角色的直接或间接成员才可以改变所有者。新的所有者必须在当前数据库上拥有 CREATE 权限。此外，FOR ALL TABLES 发布的新所有者必须是系统管理员。但是，系统管理员可以在避开这些限制的情况下更改发布的所有权。

语法格式

用指定的表替换当前发布的表。

```
ALTER PUBLICATION name SET TABLE table_name [, ...]
```

从发布中添加一个或多个表。

```
ALTER PUBLICATION name ADD TABLE table_name [, ...]
```

从发布中删除一个或多个表。

```
ALTER PUBLICATION name DROP TABLE table_name [, ...]
```

改变在 CREATE PUBLICATION 中指定的所有发布属性，未提及的属性保留其之前的设置。

```
ALTER PUBLICATION name SET ( publication_parameter [= value] [, ... ] )
```

更改发布的所有者。

```
ALTER PUBLICATION name OWNER TO new_owner
```

更改发布的名称。

```
ALTER PUBLICATION name RENAME TO new_name
```

参数说明

- name

  待修改的发布的名称。

- table_name

  现有表的名称。

- SET (publication_parameter [= value] [, …])。

  该子句修改最初由 CREATE PUBLICATION 设置的发布参数。

- new_owner

  发布的新所有者的用户名。

- new_name

  发布的新名称。

示例

详情请参见示例。

相关命令

CREATE PUBLICATION，DROP PUBLICATION

3.8.22 ALTER PACKAGE

功能描述

修改 PACKAGE 的属性。

注意事项

目前仅支持 ALTER PACKAGE OWNER 功能，系统管理员默认拥有该权限，有以下权限约束：

- 当前用户必须是该 PACKAGE 的所有者或者系统管理员，且该用户是新所有者角色的成员。

语法格式

修改 PACKAGE 的所属者。

```
ALTER PACKAGE package_name OWNER TO new_owner;
```

参数说明

- package_name

  要修改的 PACKAGE 名称。

  取值范围：已存在的 PACKAGE 名，仅支持修改单个 PACKAGE。

- new_owner

  PACKAGE 的新所有者。要修改函数的所有者，新所有者必须拥有该 PACKAGE 所属模式的 CREATE 权限。

  取值范围：已存在的用户角色。

示例

请参见 CREATE PACKAGE 中示例。

相关命令

CREATE PACKAGE，DROP PACKAGE

3.8.23 ALTER PROCEDURE

功能描述

修改自定义存储过程的属性。

注意事项

只有存储过程的所有者或者被授予了存储过程 ALTER 权限的用户才能执行 ALTER PROCEDURE 命令，系统管理员默认拥有该权限。针对所要修改属性的不同，还有以下权限约束：

如果存储过程中涉及对临时表相关的操作，则无法使用 ALTER PROCEDURE。

修改存储过程的所有者或修改存储过程的模式，当前用户必须是该存储过程的所有者或者系统管理员，且该用户是新所有者角色的成员。

只有系统管理员和初始化用户可以将 procedure 的 schema 修改成 public。

语法格式

- 修改自定义存储过程的附加参数。

```
ALTER PROCEDURE procedure_name ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    action [ ... ] [ RESTRICT ];
```

其中附加参数 action 子句语法为。

```
{CALLED ON NULL INPUT  | STRICT}
 | {IMMUTABLE | STABLE | VOLATILE}
 | {SHIPPABLE | NOT SHIPPABLE}
 | {NOT FENCED | FENCED}
 | [ NOT ] LEAKPROOF
 | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
 | AUTHID { DEFINER | CURRENT_USER }
 | COST execution_cost
 | ROWS result_rows
 | SET configuration_parameter { { TO | = } { value | DEFAULT }| FROM CURRENT}
 | RESET {configuration_parameter | ALL}
```

- 修改自定义存储过程的名称。

```
ALTER PROCEDURE proname ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    RENAME TO new_name;
```

- 修改自定义存储过程的所属者。

```
ALTER PROCEDURE proname ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    OWNER TO new_owner;
```

- 修改自定义存储过程的模式。

```
ALTER PROCEDURE proname ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    SET SCHEMA new_schema;
```

参数说明

- procedure_name

  要修改的存储过程名称。

取值范围：已存在的存储过程名。

- argmode

  标识该参数是输入、输出参数。

  取值范围：IN/OUT/INOUT/VARIADIC。

- argname

  参数名称。

  取值范围：字符串，符合标识符命名规范。

- argtype

  存储过程参数的类型。

- CALLED ON NULL INPUT

  表明该存储过程的某些参数是 NULL 的时候可以按照正常的方式调用。缺省时与指定此参数的作用相同。

- IMMUTABLE

  表示该存储过程在给出同样的参数值时总是返回同样的结果。

- STABLE

  表示该存储过程不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同 SQL 语句之间变化。

- VOLATILE

  表示该存储过程值可以在一次表扫描内改变，不会做任何优化。

- NOT FENCED | FENCED

  FENCED 或 NOT FENCED:这个子句指定例程是否被认为可以在数据库管理器操作环境的进程或地址空间中“安全地”运行。

- LEAKPROOF

  表示该存储过程没有副作用，指出参数只包括返回值。LEAKPROOF 只能由系统管理员设置。

- EXTERNAL

  （可选）目的是和 SQL 兼容，这个特性适合于所有函数，而不仅是外部函数。

- SECURITY INVOKER AUTHID CURRENT_USER

  表明该存储过程将以调用它的用户的权限执行。缺省时与指定此参数的作用相同。

  SECURITY INVOKER 和 AUTHID CURRENT_USER 的功能相同。

- SECURITY DEFINER AUTHID DEFINER

  声明该存储过程将以创建它的用户的权限执行。

  AUTHID DEFINER 和 SECURITY DEFINER 的功能相同。

- COST execution_cost

  用来估计存储过程的执行成本。

  execution_cost 以 cpu_operator_cost 为单位。

  取值范围：正数。

- ROWS result_rows

  估计存储过程返回的行数。用于存储过程返回的是一个集合。

  取值范围：正数，默认值是 1000 行。

- configuration_parameter value

  把指定的数据库会话参数值设置为给定的值。如果 value 是 DEFAULT 或者 RESET，则在新的会话中使用系统的缺省设置。OFF 关闭设置。

  取值范围：字符串。

  - DEFAULT
  - OFF
  - RESET

  指定默认值。

- from current

  取当前会话中的值设置为 configuration_parameter 的值。

- new_name

  存储过程的新名称。要修改存储过程的所属模式，必须拥有新模式的 CREATE 权限。

  取值范围：字符串，符合标识符命名规范。

- new_owner

  存储过程的新所有者。要修改存储过程的所有者，新所有者必须拥有该存储过程所属模式的 CREATE 权限。

  取值范围：已存在的用户角色。

- new_schema

  存储过程的新模式。

  取值范围：已存在的模式。

示例

请参见 CREATE FUNCTION 的示例。

相关命令

CREATE PROCEDURE，DROP PROCEDURE

3.8.24 ALTER RESOURCE LABEL

功能描述

修改资源标签。

注意事项

只有 poladmin、 sysadmin 或初始用户才能执行此操作。

语法格式

```
ALTER RESOURCE LABEL label_name (ADD|REMOVE)
  label_item_list[, ...]*;
label_item_list:
resource_type(resource_path[, ...]*)
resource_type:
TABLE | COLUMN | SCHEMA | VIEW | FUNCTION
```

参数说明

- label_name

资源标签名称。

取值范围：字符串，要符合标识符的命名规范。

- resource_type

  指的是要标记的数据库资源类型。

- resource_path

  指的是描述具体的数据库资源的路径。

示例

```
--创建基本表 table_for_label。
postgres=# CREATE TABLE table_for_label(col1 int, col2 text);

--创建资源标签 table_label。
postgres=# CREATE RESOURCE LABEL table_label ADD COLUMN(table_for_label.col1);

--将 col2 添加至资源标签 table_label 中
postgres=# ALTER RESOURCE LABEL table_label ADD COLUMN(table_for_label.col2)

--将资源标签 table_label 中的一项移除
postgres=# ALTER RESOURCE LABEL table_label REMOVE COLUMN(table_for_label.col1);
```

相关命令

CREATE RESOURCE LABEL,，DROP RESOURCE LABEL。

3.8.25 ALTER RESOURCE POOL

功能描述

修改一个资源池，指定其他控制组。

注意事项

只要用户对当前数据库有 ALTER 权限，就可以修改资源池。

语法格式

```
ALTER RESOURCE POOL pool_name
    WITH ({MEM_PERCENT=pct | CONTROL_GROUP="group_name" |
ACTIVE_STATEMENTS=stmt | MAX_DOP = dop | MEMORY_LIMIT='memory_size' |
io_limits=io_limits | io_priority='priority' |
nodegroup='nodegroup_name' }[, ... ]);
```

参数说明

- pool_name

  资源池名称。

  资源池名称为已创建的资源池。

  取值范围：字符串，要符合标识符的命名规范。

- group_name

  控制组名称。

  说明：

  设置控制组名称时，语法可以使用双引号，也可以使用单引号。

  group_name 对大小写敏感。

  不指定 group_name 时，默认指定的字符串为 “Medium”，代表指定 DefaultClass 控制组的“Medium” Timeshare 控制组。

  若数据库管理员指定自定义 Class 组下的 Workload 控制组，如 control_group 的字符串为：“class1:workload1”；代表此资源池指定到 class1 控制组下的 workload1 控制组。也可同时指定 Workload 控制组的层次，如 control_group 的字符串为：“class1:workload1:1”。

  若数据库用户指定 Timeshare 控制组代表的字符串，即“Rush”、“High”、“Medium”或“Low”其中一种，如 control_group 的字符串为“High”；代表资源池指定到 DefaultClass 控制组下的“High” Timeshare 控制组。

  取值范围：已创建的控制组。

- stmt

  资源池语句执行的最大并发数量。

  取值范围：数值型，-1~2147483647。

- dop

  资源池最大并发度，语句执行时能够创建的最多线程数量。

  取值范围：数值型，1~2147483647。

- memory_size

  资源池最大使用内存。

取值范围：字符串，内容范围 1KB~2047GB。

- mem_percent

  资源池可用内存占全部内存或者组用户内存使用的比例。

  在多租户场景下，组用户和业务用户的 mem_percent 范围为 1-100 的整数，默认为 20。

  在普通场景下，普通用户的 mem_percent 范围为 0-100 的整数，默认值为 0。

  说明：mem_percent 和 memory_limit 同时指定时，只有 mem_percent 起作用。

- io_limits

  资源池每秒可触发 IO 次数上限。

  对于行存，以万次为单位计数，而列存则以正常次数计数。

- io_priority

  IO 利用率高达 90%时，重消耗 IO 作业进行 IO 资源管控时关联的优先级等级。

  包括三档可选：Low、Medium 和 High。不控制时可设置为 None，默认为 None。

说明：io_limits 和 io_priority 的设置都仅对复杂作业有效。包括批量导入（INSERT INTO SELECT、COPY FROM、CREATE TABLE AS 等），单 DN 数据量大约超过 500MB 的复杂查询和 VACUUM FULL 等操作。

示例

本示例假定用户已成功创建自定义的 class1 控制组及其下属的 Low、wg1、wg2 三个 Workload 控制组。

```
--创建一个资源池。
postgres=# CREATE RESOURCE POOL pool1;

--更新一个资源池，其控制组指定为"DefaultClass"组下属的"High" Timeshare Workload 控制组。
postgres=# ALTER RESOURCE POOL pool1 WITH (CONTROL_GROUP="High");

--更新一个资源池，其控制组指定为"class1"组下属的"Low" Timeshare Workload 控制组。
postgres=# ALTER RESOURCE POOL pool1 WITH (CONTROL_GROUP="class1:Low");

--更新一个资源池，其控制组指定为"class1"组下属的"wg1" Workload 控制组。
postgres=# ALTER RESOURCE POOL pool1 WITH (CONTROL_GROUP="class1:wg1");
```

```
--更新一个资源池，其控制组指定为"class1"组下属的"wg2" Workload 控制组。
postgres=# ALTER RESOURCE POOL pool1 WITH (CONTROL_GROUP="class1:wg2:3");
--删除资源池 pool1。
postgres=# DROP RESOURCE POOL pool1;
```

相关命令

CREATE RESOURCE POOL，DROP RESOURCE POOL

3.8.26 ALTER ROLE

功能描述

修改角色属性。

注意事项

无。

语法格式

- 修改角色的权限。

```
ALTER ROLE role_name [ [ WITH ] option [ ... ] ];
```

其中权限项子句 option 为：

```
  {CREATEDB | NOCREATEDB}
      | {CREATEROLE | NOCREATEROLE}
      | {INHERIT | NOINHERIT}
      | {AUDITADMIN | NOAUDITADMIN}
      | {SYSADMIN | NOSYSADMIN}
      | {MONADMIN | NOMONADMIN}
      | {OPRADMIN | NOOPRADMIN}
      | {POLADMIN | NOPOLADMIN}
      | {USEFT | NOUSEFT}
      | {LOGIN | NOLOGIN}
      | {REPLICATION | NOREPLICATION}
      | {INDEPENDENT | NOINDEPENDENT}
      | {VCADMIN | NOVCADMIN}
      | {PERSISTENCE | NOPERSISTENCE}
      | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD { 'password' [ EXPIRED ] | DISABLE |
EXPIRED }
    | [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY { 'password' [ REPLACE
'old_password' | EXPIRED ] | DISABLE }
```

```
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
    | NODE GROUP logic_cluster_name
    | ACCOUNT { LOCK | UNLOCK }
    | PGUSER
```

- 修改角色的名称。

```
ALTER ROLE role_name
    RENAME TO new_name;
```

- 设置角色的配置参数。

```
ALTER ROLE role_name [ IN DATABASE database_name ]
  SET configuration_parameter {{ TO | = } { value | DEFAULT } | FROM CURRENT};
```

- 重置角色的配置参数。

```
ALTER ROLE role_name
    [ IN DATABASE database_name ] RESET {configuration_parameter|ALL};
```

参数说明

- role_name

  现有角色名。

  取值范围：已存在的用户名。

- IN DATABASE database_name

  表示修改角色在指定数据库上的参数。

- SET configuration_parameter

  设置角色的参数。ALTER ROLE 中修改的会话参数只针对指定的角色，且在下一次该角色启动的会话中有效。

  取值范围：

  configuration_parameter 和 value 的取值请参见 SET。

DEFAULT：表示清除 configuration_parameter 参数的值，configuration_parameter 参数的值将继承本角色新产生的 SESSION 的默认值。

FROM CURRENT：取当前会话中的值设置为 configuration_parameter 参数的值。

- RESET configuration_parameter/ALL

清除 configuration_parameter 参数的值。与 SET configuration_parameter TO DEFAULT 的效果相同。

取值范围：ALL 表示清除所有参数的值。

- PGUSER

当前版本不允许修改角色的 PGUSER 属性。

- PASSWORD/IDENTIFIED BY 'password'

重置或修改用户密码。除了初始用户外其他管理员或普通用户修改自己的密码需要输入正确的旧密码。只有初始用户、系统管理员（sysadmin）或拥有创建用户（CREATEROLE）权限的用户才可以重置普通用户密码，无需输入旧密码。初始用户可以重置系统管理员的密码，系统管理员不允许重置其他系统管理员的密码。

- EXPIRED

设置密码失效。只有初始用户、系统管理员（sysadmin）或拥有创建用户（CREATEROLE）权限的用户才可以设置用户密码失效，其中系统管理员也可以设置自己或其他系统管理员密码失效。不允许设置初始用户密码失效。

密码失效的用户可以登录数据库但不能执行查询操作，只有修改密码或由管理员重置密码后才可以恢复正常查询操作。

其他参数请参见 CREATE ROLE 的参数说明。

示例

请参见 CREATE ROLE 的示例。

相关命令

CREATE ROLE，DROP ROLE，SET

3.8.27 ALTER ROW LEVEL SECURITY POLICY

功能描述

对已存在的行访问控制策略（包括行访问控制策略的名称、行访问控制指定的用户、行访问控制的策略表达式）进行修改。

注意事项

表的所有者或管理员用户才能进行此操作。

语法格式

```
ALTER [ ROW LEVEL SECURITY ] POLICY [ IF EXISTS ] policy_name ON table_name RENAME
TO new_policy_name;

ALTER [ ROW LEVEL SECURITY ] POLICY policy_name ON table_name
    [ TO { role_name | PUBLIC } [, ...] ]
    [ USING ( using_expression ) ];
```

参数说明

- policy_name

  行访问控制策略名称。

- table_name

  行访问控制策略的表名。

- new_policy_name

  新的行访问控制策略名称。

- role_name

  行访问控制策略应用的数据库用户，可以指定多个用户，PUBLIC 表示应用到所有用户。

- using_expression

  行访问控制的表达式，返回值为 boolean 类型。

示例

```
--创建数据表 all_data
postgres=# CREATE TABLE all_data(id int, role varchar(100), data varchar(100));

--创建行访问控制策略，当前用户只能查看用户自身的数据
postgres=# CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role
= CURRENT_USER);
postgres=# \d+ all_data
```

```
                              Table "public.all_data"
 Column |          Type          | Modifiers | Storage  | Stats target |
Description
--------+------------------------+-----------+----------+--------------+-----
--------
 id     | integer                |           | plain    |              |
 role   | character varying(100) |           | extended |              |
 data   | character varying(100) |           | extended |              |
Row Level Security Policies:
    POLICY "all_data_rls"
      USING (((role)::name = "current_user"()))
Has OIDs: no
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no

--修改行访问控制 all_data_rls 的名称
postgres=# ALTER ROW LEVEL SECURITY POLICY all_data_rls ON all_data RENAME TO
all_data_new_rls;

--修改行访问控制策略影响的用户
postgres=# ALTER ROW LEVEL SECURITY POLICY all_data_new_rls ON all_data TO alice,
bob;
postgres=# \d+ all_data
                              Table "public.all_data"
 Column |          Type          | Modifiers | Storage  | Stats target |
Description
--------+------------------------+-----------+----------+--------------+-----
--------
 id     | integer                |           | plain    |              |
 role   | character varying(100) |           | extended |              |
 data   | character varying(100) |           | extended |              |
Row Level Security Policies:
    POLICY "all_data_new_rls"
      TO alice,bob
      USING (((role)::name = "current_user"()))
Has OIDs: no
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, enable_rowsecurity=true

--修改行访问控制策略表达式
postgres=# ALTER ROW LEVEL SECURITY POLICY all_data_new_rls ON all_data USING
(id > 100 AND role = current_user);
```

```
postgres=# \d+ all_data
                                Table "public.all_data"
 Column |          Type          | Modifiers | Storage  | Stats target |
Description
--------+------------------------+-----------+----------+--------------+-----
--------
 id     | integer                |           | plain    |              |
 role   | character varying(100) |           | extended |              |
 data   | character varying(100) |           | extended |              |
Row Level Security Policies:
    POLICY "all_data_new_rls"
      TO alice,bob
      USING (((id > 100) AND ((role)::name = "current_user"())))
Has OIDs: no
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, enable_rowsecurity=true
```

相关命令

CREATE ROW LEVEL SECURITY POLICY，DROP ROW LEVEL SECURITY POLICY

3.8.28 ALTER SCHEMA

功能描述

修改模式属性。

注意事项

只有模式的所有者或者被授予了模式 ALTER 权限的用户有权限执行 ALTER SCHEMA 命令，系统管理员默认拥有此权限。但要修改模式的所有者，当前用户必须是该模式的所有者或者系统管理员，且该用户是新所有者角色的成员。

对于系统模式 pg_catalog，只允许初始用户修改模式的所有者。

语法格式

修改模式的防篡改属性。

```
ALTER SCHEMA schema_name { WITH | WITHOUT } BLOCKCHAIN
```

修改模式的名称。

```
ALTER SCHEMA schema_name
    RENAME TO new_name;
```

修改模式的所有者。

```
ALTER SCHEMA schema_name
    OWNER TO new_owner;
```

参数说明

- schema_name

  现有模式的名称。

  取值范围：已存在的模式名。

- RENAME TO new_name

  修改模式的名称。非系统管理员要改变模式的名称，则该用户必须在此数据库上有 CREATE 权限。

  new_name：模式的新名称。

  取值范围：字符串，要符合标识符命名规范。

- OWNER TO new_owner

  修改模式的所有者。非系统管理员要改变模式的所有者，该用户还必须是新的所有角色的直接或间接成员， 并且该成员必须在此数据库上有 CREATE 权限。

  new_owner：模式的新所有者。

  取值范围：已存在的用户名/角色名。

- { WITH | WITHOUT } BLOCKCHAIN

  修改模式的防篡改属性。具有防篡改属性模式下的普通行存表均为防篡改历史表，不包括外表、临时表、系统表。当该模式下不包含任何表时才可修改防篡改属性。另外，不支持临时表模式。toast 表模式、dbe_perf 模式、blockchain 模式修改防篡改属性。

示例

```
--创建模式 ds。
postgres=# CREATE SCHEMA ds;

--将当前模式 ds 更名为 ds_new。
postgres=# ALTER SCHEMA ds RENAME TO ds_new;

--创建用户 jack。
```

```
postgres=# CREATE USER jack PASSWORD 'xxxxxxxxx';

--将 DS_NEW 的所有者修改为 jack。
postgres=# ALTER SCHEMA ds_new OWNER TO jack;

--删除用户 jack 和模式 ds_new。
postgres=# DROP SCHEMA ds_new;
postgres=# DROP USER jack;
```

相关命令

CREATE SCHEMA，DROP SCHEMA

3.8.29 ALTER SEQUENCE

功能描述

修改一个现有的序列的参数。

注意事项

只有序列的所有者或者被授予了序列 ALTER 权限的用户才能执行 ALTER SEQUENCE 命令，系统管理员默认拥有该权限。但要修改序列的所有者，当前用户必须是该序列的所有者或者系统管理员，且该用户是新所有者角色的成员。

当前版本仅支持修改拥有者、归属列和最大值。若要修改其他参数，可以删除重建，并用 Setval 函数恢复当前值。

ALTER SEQUENCE MAXVALUE 不支持在事务、函数和存储过程中使用。

修改序列的最大值后，会清空该序列在所有会话的 cache。

如果 Sequence 被创建时使用了 LARGE 标识，则 ALTER 时也需要使用 LARGE 标识。

ALTER SEQUENCE 会阻塞 nextval、setval、currval 和 lastval 的调用。

语法格式

- 修改序列归属列

```
ALTER [ LARGE ] SEQUENCE [ IF EXISTS ] name
    [ MAXVALUE maxvalue | NO MAXVALUE | NOMAXVALUE ]
    [ OWNED BY { table_name.column_name | NONE } ];

NOTICE: '[ LARGE ]' is only avaliable in CENTRALIZED mode!
```

- 修改序列的拥有者

```
ALTER [ LARGE ] SEQUENCE [ IF EXISTS ] name OWNER TO new_owner;
```

参数说明

- name

  将要修改的序列名称。

- IF EXISTS

  当序列不存在时使用该选项不会出现错误消息，仅有一个通知。

- CACHE

  为了快速访问，而在内存中预先存储序列号的个数。如果没有指定，将保持旧的缓冲值。

- OWNED BY

  将序列和一个表的指定字段进行关联。这样，在删除那个字段或其所在表的时候会自动删除已关联的序列。

  如果序列已经和表有关联后，使用这个选项后新的关联关系会覆盖旧的关联。

  关联的表和序列的所有者必须是同一个用户，并且在同一个模式中。

  使用 OWNED BY NONE 将删除任何已经存在的关联。

- new_owner

  序列新所有者的用户名。用户要修改序列的所有者，必须是新角色的直接或者间接成员，并且那个角色必须有序列所在模式上的 CREATE 权限。

示例

```
--创建一个名为 serial 的递增序列，从 101 开始。
postgres=# CREATE SEQUENCE serial START 101;

--创建一个表,定义默认值。
postgres=# CREATE TABLE T1(C1 bigint default nextval('serial'));

--将序列 serial 的归属列变为 T1.C1。
postgres=# ALTER SEQUENCE serial OWNED BY T1.C1;

--删除序列和表。
postgres=# DROP SEQUENCE serial cascade;
postgres=# DROP TABLE T1;
```

相关命令

CREATE SEQUENCE，DROP SEQUENCE

3.8.30 ALTER SERVER

功能描述

增加、修改和删除一个现有 server 的参数。已有 server 可以从 pg_foreign_server 系统表中查询。

注意事项

只有 SERVER 的所有者或者被授予了 SERVER 的 ALTER 权限的用户才可以执行 ALTER SERVER 命令，系统管理员默认拥有该权限。但要修改 SERVER 的所有者，当前用户必须是该 SERVER 的所有者或者系统管理员，且该用户是新所有者角色的成员。

语法格式

修改外部服务的参数。

```
ALTER SERVER server_name [ VERSION 'new_version' ]
  [ OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ] ) ];
```

在 OPTIONS 选项里，ADD、SET 和 DROP 指定要执行的操作，未指定时默认为 ADD 操作。option 和 value 为对应操作的参数。

修改外部服务的所有者。

```
ALTER SERVER server_name
    OWNER TO new_owner;
```

修改外部服务的名称。

```
ALTER SERVER server_name
   RENAME TO new_name;
```

参数说明

- server_name

  所修改的 server 的名称。

- new_version

  修改后 server 的新版本名称。

- OPTIONS

更改该服务器的选项。ADD、SET 和 DROP 指定要执行的动作。如果没有显式地指定操作， 将会假定为 ADD。选项名称必须唯一，名称和值也会使用该服务器的外部数据包装器库进行验证。

- oracle_fdw 支持的 options 包括：

dbserver

远端 oracle 数据库的连接字符串。

isolation_level （默认值为 serializable）

oracle 数据库的事务隔离级别。

取值范围：serializable、 read_committed 、 read_only

- mysql_fdw 支持的 options 包括：

host （默认值为 127.0.0.1）

MySQL Server/MariaDB 的地址。

port （默认值为 3306）

MySQL Server/MariaDB 侦听的端口号。

postgres_fdw 支持的 options 同 libpq 支持的连接参数一致，可参考链接字符。需要注意的是，以下几个 options 不支持修改：

user 和 password

用户名和密码将在创建 user mapping 时指定。

client_encoding

将自动获取本地 server 的编码方式并设置该值。

application_name

总是设置成 postgres_fdw。

除了 libpq 支持的连接参数外，还额外提供 3 个 options：

use_remote_estimate

控制 postgres_fdw 是否发出 EXPLAIN 命令以获取运行消耗估算。默认值为 false。

fdw_startup_cost

执行一个外表扫描时的启动耗时估算。这个值通常包含建立连接、远端对请求的分析和

生成计划的耗时。默认值为 100。

fdw_typle_cost

在远端服务器上对每一个元组进行扫描时的额外消耗。这个值通常表示数据在 server 间传输的额外消耗。默认值为 0.01。

- new_name

修改后 server 的新名称。

相关命令

CREATE SERVER ，CREATE SERVER

3.8.31 ALTER SESSION

功能描述

ALTER SESSION 命令用于定义或修改那些对当前会话有影响的条件或参数。修改后的会话参数会一直保持，直到断开当前会话。

注意事项

如果执行 SET TRANSACTION 之前没有执行 START TRANSACTION，则事务立即结束，命令无法显示效果。

可以用 START TRANSACTION 里面声明所需要的 transaction_mode(s)的方法来避免使用 SET TRANSACTION。

语法格式

设置会话的事务参数。

```
ALTER SESSION SET [ SESSION CHARACTERISTICS AS ] TRANSACTION
    { ISOLATION LEVEL  {  READ COMMITTED  | READ UNCOMMITTED  }  |  { READ ONLY
| READ WRITE  }  } [, ...] ;
```

设置会话的其他运行时参数。

```
ALTER SESSION SET
    {{config_parameter { { TO  | =  }  { value | DEFAULT }
      | FROM CURRENT }} | CURRENT_SCHEMA [ TO | = ] { schema | DEFAULT }
      | TIME ZONE time_zone
      | SCHEMA schema
      | NAMES encoding_name
      | ROLE role_name PASSWORD 'password'
```

```
    | SESSION AUTHORIZATION { role_name PASSWORD 'password' | DEFAULT }
    | XML OPTION { DOCUMENT | CONTENT }
  } ;
```

参数说明

修改会话涉及到的参数说明请参见 SET 语法中的参数说明。

示例

```
-- 创建模式 ds。
postgres=# CREATE SCHEMA ds;

--设置模式搜索路径。
postgres=# SET SEARCH_PATH TO ds, public;

--设置日期时间风格为传统的 POSTGRES 风格（日在月前）。
postgres=# SET DATESTYLE TO postgres, dmy;

--设置当前会话的字符编码为 UTF8。
postgres=# ALTER SESSION SET NAMES 'UTF8';

--设置时区为加州伯克利。
postgres=# SET TIME ZONE 'PST8PDT';

--设置时区为意大利。
postgres=# SET TIME ZONE 'Europe/Rome';

--设置当前模式。
postgres=# ALTER SESSION SET CURRENT_SCHEMA TO tpcds;

--设置 XML OPTION 为 DOCUMENT。
postgres=# ALTER SESSION SET XML OPTION DOCUMENT;

--创建角色 joe，并设置会话的角色为 joe。
postgres=# CREATE ROLE joe WITH PASSWORD 'xxxxxxxxx';
postgres=# ALTER SESSION SET SESSION AUTHORIZATION joe PASSWORD 'xxxxxxxxx';

--切换到默认用户。
gbase=> ALTER SESSION SET SESSION AUTHORIZATION default;

--删除 ds 模式。
postgres=# DROP SCHEMA ds;
```

```
--删除 joe。
postgres=# DROP ROLE joe;
```

相关命令

SET

3.8.32 ALTER SUBSCRIPTION

功能描述

ALTER SUBSCRIPTION 可以修改在 CREATE SUBSCRIPTION 中指定的订阅属性。

注意事项

订阅的所有者才能执行 ALTER SUBSCRIPTION，并且新的所有者必须是系统管理员。

语法格式

更新订阅的连接信息。

```
ALTER SUBSCRIPTION name CONNECTION 'conninfo'
```

更新订阅的发布端的发布名称。

```
ALTER SUBSCRIPTION name SET PUBLICATION publication_name [, ...]
```

激活订阅。

```
ALTER SUBSCRIPTION name ENABLE
```

更新 CREATE SUBSCRIPTION 中定义的属性。

```
ALTER SUBSCRIPTION name SET ( subscription_parameter [= value] [, ... ] )
```

更新订阅的属主。

```
ALTER SUBSCRIPTION name OWNER TO new_owner
```

修改订阅的名称。

```
ALTER SUBSCRIPTION name RENAME TO new_name
```

参数说明

- name

  要修改属性的订阅的名称。

- CONNECTION 'conninfo'

该子句修改最初由 CREATE SUBSCRIPTION 设置的连接属性。

- ENABLE (boolean)

  指定订阅是否应该主动复制，或者是否应该只是设置，但尚未启动。默认值是 true。

- SET (subscription_parameter [= value] [, …])

  该子句修改原先由 CREATE SUBSCRIPTION 设置的参数。允许的选项是 slot_name 和 synchronous_commit。

  如果创建订阅时设置 enabled 为 false，则 slot_name 将被强制设置为 NONE，即空值，即使用户指定了 slot_name 的值，复制槽也不存在。

  将 enabled 参数的值由 false 改为 true，即启用订阅时，将会连接发布端创建复制槽，此时如果用户未指定 slot_name 参数的值，则会使用默认值，即对应的订阅的名称。

  当 enabled 为 true，即订阅处于正常使用状态，不能修改 slot_name 为空，但可以修改复制槽的名称为其他非空合法名称。

- new_owner

  订阅的新所有者的用户名。

- new_name

  订阅的新名称。

示例

请参见示例。

相关命令

CREATE SUBSCRIPTION，DROP SUBSCRIPTION

3.8.33 ALTER SYNONYM

功能描述

修改 SYNONYM 对象的属性。

注意事项

目前仅支持修改 SYNONYM 对象的属主。

只有系统管理员有权限修改 SYNONYM 对象的属主信息。

新属主必须具有 SYNONYM 对象所在模式的 CREATE 权限。

语法格式

```
ALTER SYNONYM synonym_name
    OWNER TO new_owner;
```

参数说明

- synonym

  待修改的同义词名字，可以带模式名。

  取值范围：字符串，需要符合标识符的命名规范。

- new_owner

  同义词对象的新所有者。

  取值范围：字符串，有效的用户名。

示例

```
--创建同义词 t1。
postgres=# CREATE OR REPLACE SYNONYM t1 FOR ot.t1;

--创建新用户 u1。
postgres=# CREATE USER u1 PASSWORD 'user@111';

--修改同义词 t1 的 owner 为 u1。
postgres=# ALTER SYNONYM t1 OWNER TO u1;

--删除同义词 t1。
postgres=# DROP SYNONYM t1;

--删除用户 u1。
postgres=# DROP USER u1;
```

相关命令

CREATE SYNONYM，DROP SYNONYM

3.8.34 ALTER SYSTEM KILL SESSION

功能描述

ALTER SYSTEM KILL SESSION 命令用于结束一个会话。

注意事项

无。

语法格式

```
ALTER SYSTEM KILL SESSION 'session_sid, serial' [ IMMEDIATE ];
```

参数说明

- session_sid, serial

  会话的 SID 和 SERIAL（获取方法请参考示例）。

- IMMEDIATE

  表明会话将在命令执行后立即结束。

示例

```
--查询会话信息。
postgres=#
SELECT sa.sessionid AS sid,0::integer AS serial#,ad.rolname AS username FROM
pg_stat_get_activity(NULL) AS sa
LEFT JOIN pg_authid ad ON(sa.usesysid = ad.oid)WHERE sa.application_name <>
'JobScheduler';
       sid       | serial# | username
-----------------+---------+----------
 140131075880720 |       0 | gbase
 140131025549072 |       0 | gbase
 140131073779472 |       0 | gbase
 140131071678224 |       0 | gbase
 140131125774096 |       0 |
 140131127875344 |       0 |
 140131113629456 |       0 |
 140131094742800 |       0 |
(8 rows)

--结束 SID 为 140131075880720 的会话。
postgres=# ALTER SYSTEM KILL SESSION '140131075880720,0' IMMEDIATE;
```

3.8.35 ALTER SYSTEM SET

功能描述

ALTER SYSTEM SET 命令用于设置 POSTMASTER、SIGHUP、BACKEND 级别的 GUC

参数。此命令会将参数写入配置文件，不同级别生效方式有所不同。

注意事项

此命令仅限初始用户和拥有 sysadmin 权限的用户才可使用。

不同级别 GUC 参数生效时间如下：

- POSTMASTER 级别的 GUC 参数需要重启后才生效。
- BACKEND 级别的 GUC 参数需要会话重新连接后才生效。
- SIGHUP 级别的 GUC 参数立即生效（需要等待线程重新加载参数，实际略微有延迟）。

通过配置 audit_set_parameter 参数，可以配置此操作是否被审计。

操作可被备机同步。

同 gs_guc 一致，并不关注数据库是主或备节点、是否只读。

不可在事务中执行，因为此操作无法被回滚。

部分参数只能由初始用户修改，具体如下：

audit_copy_exec, audit_data_format, audit_database_process, audit_directory, audit_dml_state,

audit_dml_state_select, audit_enabled, audit_file_remain_threshold, audit_file_remain_time,

audit_function_exec, audit_grant_revoke, audit_login_logout, audit_resource_policy,

audit_rotation_interval, audit_rotation_size, audit_set_parameter, audit_space_limit,

audit_system_object, audit_user_locked, audit_user_violation,

asp_log_directory, config_file, data_directory, enable_access_server_directory,

enable_copy_server_files, external_pid_file, hba_file, ident_file, log_directory, perf_directory,

query_log_directory, ssl_ca_file, ssl_cert_file, ssl_crl_file, ssl_key_file, stats_temp_directory,

unix_socket_directory, unix_socket_group, unix_socket_permissions,

krb_caseins_users, krb_server_keyfile, krb_srvname, allow_system_table_mods, enableSeparationOfDuty,

modify_initial_password, password_encryption_type, password_policy

语法格式

```
ALTER SYSTEM SET { GUC_name } TO { GUC_value };
```

参数说明

- GUC_name

  GUC 参数名。

- GUC_value

  GUC 参数值。

示例

```
--设置 SIGHUP 级别参数 audit_enabled。
postgres=# alter system set audit_enabled to off;
ALTER SYSTEM SET
postgres=# show audit_enabled;
 audit_enabled
---------------
 off
(1 row)

--设置 POSTMASTER 级别参数 enable_thread_pool，将在重启之后生效。
postgres=# alter system set enable_thread_pool to on;
NOTICE:  please restart the database for the POSTMASTER level parameter to take
effect.
ALTER SYSTEM SET
```

3.8.36 ALTER TABLE

功能描述

修改表，包括修改表的定义、重命名表、重命名表中指定的列、重命名表的约束、设置表的所属模式、添加/更新多个列、打开/关闭行访问控制开关。

注意事项

表的所有者被授予了表 ALTER 权限的用户或被授予 ALTER ANY TABLE 的用户有权限执行 ALTER TABLE 命令，系统管理员默认拥有此权限。但要修改表的所有者或者修改表的模式，当前用户必须是该表的所有者或者系统管理员，且该用户是新所有者角色的成员。

不能修改分区表的 tablespace，但可以修改分区的 tablespace。

不支持修改存储参数 ORIENTATION。

SET SCHEMA 操作不支持修改为系统内部模式，当前仅支持用户模式之间的修改。

列存表只支持 PARTIAL CLUSTER KEY、UNIQUE、PRIMARY KEY 表级约束，不支持外键等表级约束。

列存表只支持添加字段 ADD COLUMN、修改字段的数据类型 ALTER TYPE、设置单个字段的收集目标 SET STATISTICS、支持更改表名称、支持更改表空间、支持删除字段 DROP COLUMN。对于添加的字段和修改的字段类型要求是列存支持的数据类型。ALTER TYPE 的 USING 选项只支持常量表达式和涉及本字段的表达式，暂不支持涉及其他字段的表达式。

列存表支持的字段约束包括 NULL、NOT NULL、DEFAULT 常量值、UNIQUE 和 PRIMARY KEY；对字段约束的修改当前只支持对 DEFAULT 值的修改（SET DEFAULT）和删除（DROP DEFAULT），暂不支持对非空约束 NULL/NOT NULL 的修改。

不支持增加自增列，或者增加 DEFAULT 值中包含 nextval()表达式的列。

不支持对外表、临时表开启行访问控制开关。

通过约束名删除 PRIMARY KEY 约束时，不会删除 NOT NULL 约束，如果有需要，请手动删除 NOT NULL 约束。

使用 JDBC 时，支持通过 PrepareStatement 对 DEFAUTL 值进行参数化设置。

语法格式

- 修改表的定义。

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }
    action [, ... ];
```

其中具体表操作 action 可以是以下子句之一：

```
column_clause
    | ADD table_constraint [ NOT VALID ]
    | ADD table_constraint_using_index
    | VALIDATE CONSTRAINT constraint_name
    | DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    | CLUSTER ON index_name
    | SET WITHOUT CLUSTER
    | SET ( {storage_parameter = value} [, ... ] )
    | RESET ( storage_parameter [, ... ] )
    | OWNER TO new_owner
```

```
| SET TABLESPACE new_tablespace
| SET {COMPRESS|NOCOMPRESS}
| TO { GROUP groupname | NODE ( nodename [, ... ] ) }
| ADD NODE ( nodename [, ... ] )
| DELETE NODE ( nodename [, ... ] )
| UPDATE SLICE LIKE table_name
| DISABLE TRIGGER [ trigger_name | ALL | USER ]
| ENABLE TRIGGER [ trigger_name | ALL | USER ]
| ENABLE REPLICA TRIGGER trigger_name
| ENABLE ALWAYS TRIGGER trigger_name
| ENABLE ROW LEVEL SECURITY
| DISABLE ROW LEVEL SECURITY
| FORCE ROW LEVEL SECURITY
| NO FORCE ROW LEVEL SECURITY
| ENCRYPTION KEY ROTATION
```

说明

ADD table_constraint [NOT VALID] > 给表增加一个新的约束。

ADD table_constraint_using_index > 根据已有唯一索引为表增加主键约束或唯一约束。

VALIDATE CONSTRAINT constraint_name > 验证一个使用 NOT VALID 选项创建的检查类约束，通过扫描全表来保证所有记录都符合约束条件。如果约束已标记为有效时，什么操作也不会发生。

DROP CONSTRAINT [IF EXISTS] constraint_name [RESTRICT | CASCADE] > 删除一个表上的约束。

CLUSTER ON index_name > 为将来的 CLUSTER（聚簇）操作选择默认索引。实际上并没有重新盘簇化处理该表。

SET WITHOUT CLUSTER > 从表中删除最新使用的 CLUSTER 索引。这样会影响将来那些没有声明索引的 CLUSTER（聚簇）操作。

SET ({storage_parameter = value} [, ···]) > 修改表的一个或多个存储参数。

RESET (storage_parameter [, ···]) > 重置表的一个或多个存储参数。与 SET 一样，根据参数的不同可能需要重写表才能获得想要的效果。

OWNER TO new_owner > 将表、序列、视图的属主改变成指定的用户。

SET TABLESPACE new_tablespace > 这种形式将表空间修改为指定的表空间并将相关的数据文件移动到新的表空间。但是表上的所有索引都不会被移动，索引可以通过 ALTER

INDEX 语法的 SET TABLESPACE 选项来修改索引的表空间。

SET {COMPRESS|NOCOMPRESS} > 修改表的压缩特性。表压缩特性的改变只会影响后续批量插入的数据的存储方式，对已有数据的存储毫无影响。也就是说，表压缩特性的修改会导致该表中同时存在着已压缩和未压缩的数据。行存表不支持压缩。

TO { GROUP groupname | NODE (nodename [, …]) } > 此语法仅在扩展模式(GUC 参数 support_extended_features 为 on 时) 下可用。该模式谨慎打开，主要供内部扩容工具使用，一般用户不应使用该模式。

ADD NODE (nodename [, …]) > 此语法主要供内部扩容工具使用，一般用户不建议使用。

DELETE NODE (nodename [, …]) > 此语法主要供内部缩容工具使用，一般用户不建议使用。

DISABLE TRIGGER [trigger_name | ALL | USER] > 禁用 trigger_name 所表示的单个触发器，或禁用所有触发器，或仅禁用用户触发器（此选项不包括内部生成的约束触发器，例如，可延迟唯一性和排除约束的约束触发器）。 > 应谨慎使用此功能，因为如果不执行触发器，则无法保证原先期望的约束的完整性。

| ENABLE TRIGGER [trigger_name | ALL | USER] > 启用 trigger_name 所表示的单个触发器，或启用所有触发器，或仅启用用户触发器。

| ENABLE REPLICA TRIGGER trigger_name > 触发器触发机制受配置变量 session_replication_role 的影响，当复制角色为“origin”（默认值）或“local”时，将触发简单启用的触发器。 > 配置为 ENABLE REPLICA 的触发器仅在会话处于“replica”模式时触发。

| ENABLE ALWAYS TRIGGER trigger_name > 无论当前复制模式如何，配置为 ENABLE ALWAYS 的触发器都将触发。

| DISABLE/ENABLE [REPLICA | ALWAYS] RULE > 配置属于表的重写规则,已禁用的规则对系统来说仍然是可见的，只是在查询重写期间不被应用。语义为关闭/启动规则。由于关系到视图的实现，ON SELECT 规则不可禁用。 配置为 ENABLE REPLICA 的规则将会仅在会话为”replica” 模式时启动，而配置为 ENABLE ALWAYS 的触发器将总是会启动，不考虑当前复制模式。规则触发机制也受配置变量 session_replication_role 的影响，类似于上述触发器。

| DISABLE/ENABLE ROW LEVEL SECURITY > 开启或关闭表的行访问控制开关。 >

当开启行访问控制开关时，如果未在该数据表定义相关行访问控制策略，数据表的行级访问将不受影响；如果关闭表的行访问控制开关，即使定义了行访问控制策略，数据表的行访问也不受影响。详细信息参见 CREATE ROW LEVEL SECURITY POLICY 章节。

| NO FORCE/FORCE ROW LEVEL SECURITY > 强制开启或关闭表的行访问控制开关。 > 默认情况，表所有者不受行访问控制特性影响，但当强制开启表的行访问控制开关时，表的所有者（不包含系统管理员用户）会受影响。系统管理员可以绕过所有的行访问控制策略，不受影响。

其中列相关的操作 column_clause 可以是以下子句之一：

```
ADD [ COLUMN ] column_name data_type [ compress_mode ] [ COLLATE collation ]
[ column_constraint [ ... ] ]
| MODIFY column_name data_type
| MODIFY column_name [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ]
| MODIFY column_name [ CONSTRAINT constraint_name ] NULL
| DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
| ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
[ USING expression ]
| ALTER [ COLUMN ] column_name { SET DEFAULT expression | DROP DEFAULT }
| ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
| ALTER [ COLUMN ] column_name SET STATISTICS [PERCENT] integer
| ADD STATISTICS (( column_1_name, column_2_name [, ...] ))
| DELETE STATISTICS (( column_1_name, column_2_name [, ...] ))
| ALTER [ COLUMN ] column_name SET ( {attribute_option = value} [, ... ] )
| ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
| ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
```

ADD [COLUMN] column_name data_type [compress_mode] [COLLATE collation] [column_constraint […]] 向表中增加一个新的字段。用 ADD COLUMN 增加一个字段，所有表中现有行都初始化为该字段的缺省值（如果没有声明 DEFAULT 子句，值为 NULL）。

ADD ({ column_name data_type [compress_mode] } [, …]) 向表中增加多列。

MODIFY ({ column_name data_type | column_name [CONSTRAINT constraint_name] NOT NULL [ENABLE] | column_name [CONSTRAINT constraint_name] NULL } [, …]) 修改表已存在字段的数据类型。

DROP [COLUMN] [IF EXISTS] column_name [RESTRICT | CASCADE] 从表中删除一个字段，和这个字段相关的索引和表约束也会被自动删除。如果任何表之外的对象依赖于

这个字段，必须声明 CASCADE ，比如视图。 DROP COLUMN 命令并不是物理上把字段删除，而只是简单地把它标记为对 SQL 操作不可见。随后对该表的插入和更新将在该字段存储一个 NULL。因此，删除一个字段是很快的，但是它不会立即释放表在磁盘上的空间，因为被删除了的字段占据的空间还没有回收。这些空间将在执行 VACUUM 时而得到回收。

ALTER [COLUMN] column_name [SET DATA] TYPE data_type [COLLATE collation] [USING expression] 改变表字段的数据类型。该字段涉及的索引和简单的表约束将被自动地转换为使用新的字段类型，方法是重新分析最初提供的表达式。 ALTER TYPE 要求重写整个表的特性有时候是一个优点，因为重写的过程消除了表中没用的空间。比如，要想立刻回收被一个已经删除的字段占据的空间，最快的方法是：

```
ALTER TABLE table ALTER COLUMN anycol TYPE anytype;
```

这里的 anycol 是任何在表中还存在的字段，而 anytype 是和该字段的原类型一样的类型。这样的结果是在表上没有任何可见的语意的变化，但是这个命令强制重写，这样就删除了不再使用的数据。

ALTER [COLUMN] column_name { SET DEFAULT expression | DROP DEFAULT } 为一个字段设置或者删除缺省值。请注意缺省值只应用于随后的 INSERT 命令，它们不会修改表中已经存在的行。也可以为视图创建缺省，这个时候它们是在视图的 ON INSERT 规则应用之前插入到 INSERT 句中的。

ALTER [COLUMN] column_name { SET | DROP } NOT NULL 修改一个字段是否允许 NULL 值或者拒绝 NULL 值。如果表在字段中包含非 NULL，则只能使用 SET NOT NULL。

ALTER [COLUMN] column_name SET STATISTICS [PERCENT] integer 为随后的 ANALYZE 操作设置针对每个字段的统计收集目标。目标的范围可以在 0 到 10000 之内设置。设置为-1 时表示重新恢复到使用系统缺省的统计目标。

{ADD | DELETE} STATISTICS ((column_1_name, column_2_name [, ...]))用于添加和删除多列统计信息声明（不实际进行多列统计信息收集），以便在后续进行全表或全库 analyze 时进行多列统计信息收集。每组多列统计信息最多支持 32 列。不支持添加/删除多列统计信息声明的表：系统表、外表。

ALTER [COLUMN] column_name SET ({attribute_option = value} [, ...])设置/重置属性选项。目前，属性选项只定义了 n_distinct 和 n_distinct_inherited。n_distinct 影响表本身的统计值，而 n_distinct_inherited 影响表及其继承子表的统计。目前，只支持 SET/RESET n_distinct 参数，禁止 SET/RESET n_distinct_inherited 参数。

ALTER [COLUMN] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED

| MAIN }为一个字段设置存储模式。这个设置控制这个字段是内联保存还是保存在一个附属的表里，以及数据是否要压缩。仅支持对行存表的设置；对列存表没有意义，执行时报错。SET STORAGE 本身并不改变表上的任何东西，只是设置将来的表操作时，建议使用的策略。

其中列约束 column_constraint 为：

```
[ CONSTRAINT constraint_name ]
    { NOT NULL |
      NULL |
      CHECK ( expression ) |
      DEFAULT default_expr |
      GENERATED ALWAYS AS ( generation_expr ) STORED |
      UNIQUE index_parameters |
      PRIMARY KEY index_parameters |
      ENCRYPTED WITH ( COLUMN_ENCRYPTION_KEY = column_encryption_key,
ENCRYPTION_TYPE = encryption
_type_value ) |
      REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
SIMPLE ]
        [ ON DELETE action ] [ ON UPDATE action ] }
    [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

其中列的压缩可选项 compress_mode 为：

```
[ DELTA | PREFIX | DICTIONARY | NUMSTR | NOCOMPRESS ]
```

其中根据已有唯一索引为表增加主键约束或唯一约束 table_constraint_using_index 为：

```
[ CONSTRAINT constraint_name ]
    { UNIQUE | PRIMARY KEY } USING INDEX index_name
    [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

其中表约束 table_constraint 为：

```
[ CONSTRAINT constraint_name ]
    { CHECK ( expression ) |
      UNIQUE ( column_name [, ... ] ) index_parameters |
      PRIMARY KEY ( column_name [, ... ] ) index_parameters |
      PARTIAL CLUSTER KEY ( column_name [, ... ] ) |
      FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
[, ... ] ) ]
        [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON
UPDATE action ] }
    [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

其中索引参数 index_parameters 为：

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
    [ USING INDEX TABLESPACE tablespace_name ]
```

- 重命名表。对名称的修改不会影响所存储的数据。

```
ALTER TABLE [ IF EXISTS ] table_name
    RENAME TO new_table_name;
```

- 重命名表中指定的列。

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name )}
RENAME [ COLUMN ] column_name TO new_column_name;
```

- 重命名表的约束。

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }
    RENAME CONSTRAINT constraint_name TO new_constraint_name;
```

- 设置表的所属模式。

```
ALTER TABLE [ IF EXISTS ] table_name
    SET SCHEMA new_schema;
```

说明

这种形式把表移动到另外一个模式。相关的索引、约束都跟着移动。目前序列不支持改变 schema。若该表拥有序列，需要将序列删除，重建，或者取消拥有关系，才能将表 schema 更改成功。

要修改一个表的模式，用户必须在新模式上拥有 CREATE 权限。要把该表添加为一个父表的新子表，用户必须同时又是父表的所有者。要修改所有者，用户还必须是新的所有角色的直接或间接成员，并且该成员必须在此表的模式上有 CREATE 权限。这些限制规定了该用户不能做出了重建和删除表之外的事情。不过，系统管理员可以以任何方式修改任意表的所有权限。

除了 RENAME 和 SET SCHEMA 之外所有动作都可以捆绑在一个经过多次修改的列表中并行使用。比如，可以在一个命令里增加几个字段或修改几个字段的类型。对于大表，此种操作带来的效率提升更明显，原因在于只需要对该大表做一次处理。

增加一个 CHECK 或 NOT NULL 约束将会扫描该表，以保证现有的行符合约束要求。

用一个非空缺省值增加一个字段或者改变一个字段的现有类型会重写整个表。对于大表来说，这个操作可能会花很长时间，并且它还临时需要两倍的磁盘空间。

- 添加多个列

```
ALTER TABLE [ IF EXISTS ] table_name
    ADD ( { column_name data_type [ compress_mode ] [ COLLATE collation ]
[ column_constraint [ ... ] ]} [, ...] );
```

- 更新多个列

```
ALTER TABLE [ IF EXISTS ] table_name
MODIFY ( { column_name data_type | column_name [ CONSTRAINT constraint_name ] NOT
NULL [ ENABLE ] | column_name [ CONSTRAINT constraint_name ] NULL } [, ...] );
```

参数说明

- IF EXISTS

  如果不存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表不存在。

- table_name [*] | ONLY table_name | ONLY (table_name)

  table_name 是需要修改的表名。

  若声明了 ONLY 选项，则只有那个表被更改。若未声明 ONLY，该表及其所有子表都将会被更改。另外，可以在表名称后面显示地增加*选项来指定包括子表，即表示所有后代表都被扫描，这是默认行为。

- constraint_name

  要删除的现有约束的名称。

- index_name

  索引名称。

- storage_parameter

  表的存储参数的名称。

  创建索引新增一个选项：

- parallel_workers（int 类型）

  取值范围：[0,32]，0 表示关闭并发。

  表示创建索引时起的 bgworker 线程数量，例如 2 就表示将会起 2 个 bgworker 线程并发创建索引。

  如果未设置，启动 bgworker 线程数量与表大小相关，一般不超过 4 个线程。

- hasuids（bool 类型）

  默认值：off

  参数开启：更新表元组时，为元组分配表级唯一标识 id。

- new_owner

  表新拥有者的名称。

- new_tablespace

  表所属新的表空间名称。

- column_name、column_1_name、 column_2_name

  现存的或新字段的名称。

- data_type

  新字段的类型，或者现存字段的新类型。

- compress_mode

  表字段的压缩可选项。该子句指定该字段优先使用的压缩算法。行存表不支持压缩。

- collation

  字段排序规则名称。可选字段 COLLATE 指定了新字段的排序规则，如果省略，排序规则为新字段的默认类型。排序规则可以使用“select * from pg_collation;”命令从 pg_collation 系统表中查询，默认的排序规则为查询结果中以 default 开始的行。

- USING expression

  USING 子句声明如何从旧的字段值里计算新的字段值；如果省略，缺省从旧类型向新类型的赋值转换。如果从旧数据类型到新类型没有隐含或者赋值的转换，则必须提供一个 USING 子句。

  说明：ALTER TYPE 的 USING 选项实际上可以声明涉及该行旧值的任何表达式，即它可以引用除了正在被转换的字段之外其他的字段。这样，就可以用 ALTER TYPE 语法做非常普遍性的转换。因为这个灵活性，USING 表达式并没有作用于该字段的缺省值（如果有的话），结果可能不是缺省表达式要求的常量表达式。这就意味着如果从旧类型到新类型没有隐含或者赋值转换的话，即使存在 USING 子句，ALTER TYPE 也可能无法把缺省值转换成新的类型。在这种情况下，应该用 DROP DEFAULT 先删除缺省，执行 ALTER TYPE，然后使用 SET DEFAULT 增加一个合适的新缺省值。类似的考虑也适用于涉及该字段的索引

和约束。

- NOT NULL | NULL

  设置列是否允许空值。

- integer

  带符号的整数常值。当使用 PERCENT 时表示按照表数据的百分比收集统计信息，integer 的取值范围为 0-100。

- attribute_option

  属性选项。

- PLAIN | EXTERNAL | EXTENDED | MAIN

  字段存储模式。

  PLAIN 必需用于定长的数值（比如 integer）并且是内联的、不压缩的。

  MAIN 用于内联、可压缩的数据。

  EXTERNAL 用于外部保存、不压缩的数据。使用 EXTERNAL 将令在 text 和 bytea 字段上的子字符串操作更快，但付出的代价是增加了存储空间。

  EXTENDED 用于外部的压缩数据，EXTENDED 是大多数支持非 PLAIN 存储的数据的缺省。

- CHECK (expression)

  每次将要插入的新行或者将要被更新的行必须使表达式结果为真才能成功，否则会抛出一个异常并且不会修改数据库。

  声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以引用多个字段。

  目前，CHECK 表达式不能包含子查询也不能引用除当前行字段之外的变量。

- DEFAULT default_expr

  给字段指定缺省值。

  缺省表达式的数据类型必须和字段类型匹配。

  缺省表达式将被用于任何未声明该字段数值的插入操作。如果没有指定缺省值则缺省值为 NULL 。

- UNIQUE index_parameters | UNIQUE (column_name [, …]) index_parameters

  UNIQUE 约束表示表里的一个或多个字段的组合必须在全表范围内唯一。

- PRIMARY KEY index_parameters | PRIMARY KEY (column_name [, …]) index_parameters

  主键约束表明表中的一个或者一些字段只能包含唯一（不重复）的非 NULL 值。

- REFERENCES reftable [(refcolum)] [MATCH matchtype] [ON DELETE action] [ON UPDATE action] (column constraint) | FOREIGN KEY (column_name [, …]) REFERENCES reftable [(refcolumn [, …])] [MATCH matchtype] [ON DELETE action] [ON UPDATE action] (table constraint)

  ⚠ 注意

  外键约束要求新表中一列或多列构成的组应该只包含、匹配被参考表中被参考字段值。若省略 refcolum，则将使用 reftable 的主键。被参考列应该是被参考表中的唯一字段或主键。外键约束不能被定义在临时表和永久表之间。

  参考字段与被参考字段之间存在三种类型匹配，分别是：

  MATCH FULL：不允许一个多字段外键的字段为 NULL，除非全部外键字段都是 NULL。

  MATCH SIMPLE（缺省）：允许任意外键字段为 NULL。

  MATCH PARTIAL：目前暂不支持。

  另外，当被参考表中的数据发生改变时，某些操作也会在新表对应字段的数据上执行。ON DELETE 子句声明当被参考表中的被参考行被删除时要执行的操作。ON UPDATE 子句声明当被参考表中的被参考字段数据更新时要执行的操作。对于 ON DELETE 子句、ON UPDATE 子句的可能动作：

  NO ACTION（缺省）：删除或更新时，创建一个表明违反外键约束的错误。若约束可推迟，且若仍存在任何引用行，那这个错误将会在检查约束的时候产生。

  RESTRICT：删除或更新时，创建一个表明违反外键约束的错误。与 NO ACTION 相同，只是动作不可推迟。

  CASCADE：删除新表中任何引用了被删除行的行，或更新新表中引用行的字段值为被参考字段的新值。

  SET NULL：设置引用字段为 NULL。

SET DEFAULT：设置引用字段为它们的缺省值。

DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE

设置该约束是否可推迟。

DEFERRABLE：可以推迟到事务结尾使用 SET CONSTRAINTS 命令检查。

NOT DEFERRABLE：在每条命令之后马上检查。

INITIALLY IMMEDIATE：那么每条语句之后就立即检查它。

INITIALLY DEFERRED：只有在事务结尾才检查它。

说明：Ustore 表不支持新增 DEFERRABLE 以及 INITIALLY DEFERRED 约束。

- PARTIAL CLUSTER KEY

  局部聚簇存储，列存表导入数据时按照指定的列(单列或多列)，进行局部排序。

- WITH ({storage_parameter = value} [, •••])

  为表或索引指定一个可选的存储参数。

- tablespace_name

  索引所在表空间的名称。

- COMPRESS|NOCOMPRESS

  NOCOMPRESS：如果指定关键字 NOCOMPRESS 则不会修改表的现有压缩特性。

  COMPRESS：如果指定 COMPRESS 关键字，则对该表进行批量插入元组时触发该特性。行存表不支持压缩。

- new_table_name

  修改后新的表名称。

- new_column_name

  表中指定列修改后新的列名称。

- new_constraint_name

  修改后表约束的新名称。

- new_schema

修改后新的模式名称。

- CASCADE

  级联删除依赖于被依赖字段或者约束的对象（比如引用该字段的视图）。

- RESTRICT

  如果字段或者约束还有任何依赖的对象，则拒绝删除该字段。这是缺省行为。

- schema_name

  表所在的模式名称。

示例

请参考 CREATE TABLE 的示例。

相关命令

CREATE TABLE，DROP TABLE

3.8.37 ALTER TABLE PARTITION

功能描述

修改表分区，包括增删分区、切割分区、合成分区以及修改分区属性等。

注意事项

- 添加分区的表空间不能是 PG_GLOBAL。
- 添加分区的名称不能与该分区表已有分区的名称相同。
- 添加分区的分区键值要和分区表的分区键的类型一致。
- 若添加 RANGE 分区，添加分区键值要大于分区表中最后一个范围分区的上边界。
- 若添加 LIST 分区，添加分区键值不能与现有分区键值重复。
- 不支持添加 HASH 分区。
- 如果目标分区表中已有分区数达到了最大值 1048575，则不能继续添加分区。
- 当分区表只有一个分区时，不能删除该分区。
- 选择分区使用 PARTITION FOR()，括号里指定值个数应该与定义分区时使用的列个数相同，并且一一对应。
- Value 分区表不支持相应的 Alter Partition 操作。

- 列存分区表不支持切割分区。
- 间隔分区表不支持添加分区。
- 哈希分区表不支持切割分区，不支持合成分区，不支持添加和删除分区。
- 列表分区表不支持切割分区，不支持合成分区。
- 只有分区表的所有者或者被授予了分区表 ALTER 权限的用户有权限执行 ALTER TABLE PARTITION 命令，系统管理员默认拥有此权限。

语法格式

- 修改表分区主语法。

```
ALTER TABLE [ IF EXISTS ] { table_name  [*] | ONLY table_name | ONLY
( table_name  )}
    action [, ... ];
```

action 统指如下分区维护子语法。当存在多个分区维护子句时，保证了分区的连续性，无论这些子句的排序如何，GBase 8s 总会先执行 DROP PARTITION 再执行 ADD PARTITION 操作，最后顺序执行其它分区维护操作。

```
    move_clause  |
    exchange_clause  |
    row_clause  |
    merge_clause  |
    modify_clause  |
    split_clause  |
    add_clause  |
    drop_clause  |
    truncate_clause
```

move_clause 子语法用于移动分区到新的表空间。

```
MOVE PARTITION { partion_name | FOR ( partition_value [, ...] ) } TABLESPACE
tablespacename
```

exchange_clause 子语法用于把普通表的数据迁移到指定的分区。

```
EXCHANGE PARTITION { ( partition_name ) | FOR ( partition_value [, ...] ) }
    WITH TABLE {[ ONLY ] ordinary_table_name | ordinary_table_name * | ONLY
( ordinary_table_name
)}
    [ { WITH | WITHOUT } VALIDATION ] [ VERBOSE ]
```

进行交换的普通表和分区必须满足如下条件：

- 普通表和分区的列数目相同，对应列的信息严格一致，包括：列名、列的数据类型、列约束、列的 Collation 信息、列的存储参数、列的压缩信息等。
- 普通表和分区的表压缩信息严格一致。
- 普通表和分区的索引个数相同，且对应索引的信息严格一致。
- 普通表和分区的表约束个数相同，且对应表约束的信息严格一致。
- 普通表不可以是临时表，分区表只能是范围分区表，列表分区表，哈希分区表。
- 普通表和分区表上不可以有动态数据脱敏，行访问控制约束。
- 列表分区表，哈希分区表不能是列存储。
- List/Hash/Range 类型分区表支持 exchange_clause。

须知

- 完成交换后，普通表和分区的数据被置换，同时普通表和分区的表空间信息被置换。此时，普通表和分区的统计信息变得不可靠，需要对普通表和分区重新执行 analyze。
- 由于非分区键不能建立本地唯一索引，只能建立全局唯一索引，所以如果普通表含有唯一索引时，会导致不能交换数据。

row_clause 子语法用于设置分区表的行迁移开关。

```
{ ENABLE | DISABLE } ROW MOVEMENT
```

merge_clause 子语法用于把多个分区合并成一个分区。

```
MERGE PARTITIONS { partition_name } [, ...] INTO PARTITION partition_name
 [ TABLESPACE tablespacename ]
```

modify_clause 子语法用于设置分区索引是否可用。

```
MODIFY PARTITION partition_name { UNUSABLE LOCAL INDEXES | REBUILD UNUSABLE LOCAL
INDEXES }
```

split_clause 子语法用于把一个分区切割成多个分区。

```
SPLIT PARTITION { partition_name | FOR ( partition_value [, ...] ) }
{ split_point_clause | no_split_point_clause }
```

指定切割点 split_point_clause 的语法为。

```
  AT ( partition_value ) INTO ( PARTITION partition_name [ TABLESPACE
tablespacename ] , PARTITION partition_name [ TABLESPACE tablespacename ] )
```

须知

- 列存分区表不支持切割分区。
- 切割点的大小要位于正在被切割的分区的分区键范围内，指定切割点的方式只能把一个分区切割成两个新分区。

不指定切割点 no_split_point_clause 的语法为：

```
INTO { ( partition_less_than_item [, ...] ) | ( partition_start_end_item [, ...] ) }
```

须知

- 不指定切割点的方式，partition_less_than_item 指定的第一个新分区的分区键要大于正在被切割的分区的前一个分区（如果存在的话）的分区键，partition_less_than_item 指定的最后一个分区的分区键要等于正在被切割的分区的分区键大小。
- 不指定切割点的方式，partition_start_end_item 指定的第一个新分区的起始点（如果存在的话）必须等于正在被切割的分区的前一个分区（如果存在的话）的分区键，partition_start_end_item 指定的最后一个分区的终止点（如果存在的话）必须等于正在被切割的分区的分区键。
- partition_less_than_item 支持的分区键个数最多为 4，而 partition_start_end_item 仅支持 1 个分区键，其支持的数据类型参见 [PARTITION BY RANGE(parti...](CREATE-TABLE-PARTITION.html#zh-cn_topic_0283136653_zh-cn_topic_0237122119_zh-cn_topic_0059777586_l00efc30fe63048ffa2ef68s5b18bb455)。
- 在同一语句中 partition_less_than_item 和 partition_start_end_item 两者不可同时使用；不同 split 语句之间没有限制。

add_clause 子语法用于为指定的分区表添加一个或多个分区。

```
ADD PARTITION ( partition_col1_name = partition_col1_value [,
partition_col2_name = partition_col2
_value ] [, ...] )
    [ LOCATION 'location1' ]
    [ PARTITION (partition_colA_name = partition_colA_value [,
partition_colB_name = partition_col
B_value ] [, ...] ) ]
    [ LOCATION 'location2' ]
ADD {partition_less_than_item | partition_start_end_item}
```

分区项 partition_less_than_item 的语法为

```
PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE }
[, ...] ) [ TABLES
PACE tablespacename ]
```

分区项 partition_start_end_item 的语法为

```
PARTITION partition_name {
        {START(partition_value) END (partition_value) EVERY (interval_value)} |
        {START(partition_value) END ({partition_value | MAXVALUE})} |
        {START(partition_value)} |
        {END({partition_value | MAXVALUE})}
} [TABLESPACE tablespace_name]
```

drop_clause 子语法用于删除分区表中的指定分区。

```
DROP PARTITION  { partition_name  | FOR (  partition_value [, ...] )  }
```

须知

- 间隔/哈希分区表不支持添加分区。

truncate_clause 子语法用于清空分区表中的指定分区。

```
TRUNCATE PARTITION  { partition_name | FOR (  partition_value [, ...] )  }
[ UPDATE GLOBAL INDEX ]
NOTICE: 'truncate_clause' is only avaliable in CENTRALIZED mode!
```

修改表分区名称的语法：

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name )}
RENAME PARTITION { partion_name | FOR ( partition_value [, ...] ) } TO
partition_new_name;
```

参数说明

- table_name

  分区表名。

  取值范围：已存在的分区表名。

- partition_name

  分区名。

  取值范围：已存在的分区名。

- tablespacename

  指定分区要移动到哪个表空间。

取值范围：已存在的表空间名。

- partition_value

  分区键值。

  通过 PARTITION FOR (partition_value [, …])子句指定的这一组值，可以唯一确定一个分区。

  取值范围：需要进行重命名的分区的分区键的取值范围。

- UNUSABLE LOCAL INDEXES

  设置该分区上的所有索引不可用。

- REBUILD UNUSABLE LOCAL INDEXES

  重建该分区上的所有索引。

- ENABLE/DISABLE ROW MOVEMET

  行迁移开关。

  如果进行 UPDATE 操作时，更新了元组在分区键上的值，造成了该元组所在分区发生变化，就会根据该开关给出报错信息，或者进行元组在分区间的转移。

  取值范围：

  ENABLE：打开行迁移开关。

  DISABLE：关闭行迁移开关。

  默认是打开状态。

- ordinary_table_name

  进行迁移的普通表的名称。

  取值范围：已存在的普通表名。

- { WITH | WITHOUT } VALIDATION

  在进行数据迁移时，是否检查普通表中的数据满足指定分区的分区键范围。

  取值范围：

  WITH：对于普通表中的数据要检查是否满足分区的分区键范围，如果有数据不满足，则报错。

  WITHOUT：对于普通表中的数据不检查是否满足分区的分区键范围。

默认是 WITH 状态。

由于检查比较耗时，特别是当数据量很大的情况下更甚。所以在保证当前普通表中的数据满足分区的分区键范围时，可以加上 WITHOUT 来指明不进行检查。

- VERBOSE

在 VALIDATION 是 WITH 状态时，如果检查出普通表有不满足要交换分区的分区键范围的数据，那么把这些数据插入到正确的分区，如果路由不到任何分区，再报错。

须知

只有在 VALIDATION 是 WITH 状态时，才可以指定 VERBOSE。

- partition_new_name

分区的新名称。

取值范围：字符串，要符合标识符的命名规范。

示例

请参考 CREATE TABLE PARTITION 的示例。

相关命令

CREATE TABLE PARTITION，DROP TABLE

3.8.38 ALTER TABLE SUBPARTITION

功能描述

修改二级分区表分区，包括增删分区、清空分区、切割分区等。

注意事项

- 目前二级分区表只支持增删分区、清空分区、切割分区。
- 添加分区的表空间不能是 PG_GLOBAL。
- 添加分区的名称不能与该分区表已有一级分区和二级分区的名称相同。
- 添加分区的分区键值要和分区表的分区键的类型一致。
- 若添加 RANGE 分区，添加分区键值要大于分区表中最后一个范围分区的上边界。若需要在有 MAXVALUE 分区的表上新增分区，建议使用 SPLIT 语法。
- 若添加 LIST 分区，添加分区键值不能与现有分区键值重复。若需要在有 DEFAULT 分区的表上新增分区，建议使用 SPLIT 语法。

- 不支持添加 HASH 分区。只有一种情况例外，二级分区表的二级分区方式为 HASH 且一级分区方式不是 HASH，此时支持新增一级分区并创建对应的二级分区。
- 如果目标分区表中已有分区数达到了最大值 1048575，则不能继续添加分区。
- 当分区表只有一个一级分区或二级分区时，不能删除该分区。
- 不支持删除 HASH 分区。
- 选择分区使用 PARTITION FOR()，括号里指定值个数应该与定义分区时使用的列个数相同，并且一一对应。
- 切割分区只能对二级分区（叶子节点）进行切割，被切割分区只能是 Range、List 分区策略，不支持切割 hash 分区策略。List 分区策略只能是 default 分区才能被切割。
- 只有分区表的所有者或者被授予了分区表 ALTER 权限的用户有权限执行 ALTER TABLE PARTITION 命令，系统管理员默认拥有此权限。
- 如果 alter 语句不带有 UPDATE GLOBAL INDEX，那么原有的 GLOBAL 索引将失效，查询时将使用其他索引进行查询；如果 alter 语句带有 UPDATEGLOBAL INDEX，原有的 GLOBAL 索引仍然有效，并且索引功能正确。

语法格式

- 修改表分区主语法。

```
ALTER TABLE [ IF EXISTS ] { table_name  [*] | ONLY table_name | ONLY
( table_name  )}
  action [, ... ];
```

其中 action 统指如下分区维护子语法。

```
    add_clause    |
    drop_clause   |
    split_clause  |
    truncate_clause
```

add_clause 子语法用于为指定的分区表添加一个或多个分区。语法可以作用在一级分区上。

```
ADD {partition_less_than_item | partition_list_item }
[ ( subpartition_definition_list ) ]
```

也可以作用在二级分区上。

```
 MODIFY PARTITION partition_name ADD subpartition_definition
```

其中，分区项 partition_less_than_item 为 RANGE 分区定义语法，具体语法如下。

```
 PARTITION partition_name VALUES LESS THAN ( partition_value | MAXVALUE )
[ TABLESPACE tablespacename ]
```

分区项 partition_list_item 为 LIST 分区定义语法，具体语法如下。

```
 PARTITION partition_name VALUES ( partition_value [, ...] | DEFAULT )
[ TABLESPACE tablespacename ]
```

subpartition_definition_list 为 1 到多个二级分区 subpartition_definition 对象，subpartition_definition 具体语法如下。

```
 SUBPARTITION subpartition_name [ VALUES LESS THAN ( partition_value |
MAXVALUE ) | VALUES ( partition_value [, ...] | DEFAULT )]  [ TABLESPACE
tablespace ]
```

须知：若一级分区为 HASH 分区，不支持以 ADD 形式新增一级分区；若二级分区为 HASH 分区，不支持以 MODIFY 形式新增二级分区。

drop_clause 子语法用于删除分区表中的指定分区。语法可以作用在一级分区上。

```
DROP PARTITION  { partition_name | FOR (  partition_value )  } [ UPDATE GLOBAL
INDEX ]
```

也可以作用在二级分区上。

```
 DROP SUBPARTITION  { subpartition_name | FOR (  partition_value,
subpartition_value )  } [ UPDATE GLOBAL INDEX ]
```

须知：

若一级分区为 HASH 分区，不支持删除一级分区；若二级分区为 HASH 分区，不支持删除二级分区。

不支持删除唯一子分区。

split_clause 子语法用于把一个分区切割成多个分区。

```
SPLIT SUBPARTITION { subpartition_name} { split_point_clause  } [ UPDATE GLOBAL
INDEX ]
```

指定切割点 split_point_clause 的语法为：

```
 AT ( subpartition_value ) INTO ( SUBPARTITION subpartition_name [ TABLESPACE
tablespacename ], SUBPARTITION subpartition_name [ TABLESPACE tablespacename ] )
```

须知：

切割点的大小要位于正在被切割的分区的分区键范围内。

只能把一个分区切割成两个新分区。

指定 Range 分区策略切割点 split_point_clause 的语法为：

```
AT ( subpartition_value ) INTO ( SUBPARTITION subpartition_name [ TABLESPACE tablespacename ], SUBPARTITION subpartition_name [ TABLESPACE tablespacename ] )
```

指定 List 分区策略切割点 split_point_clause 的语法为：

```
VALUES ( subpartition_value ) INTO ( SUBPARTITION subpartition_name [ TABLESPACE tablespacename ], SUBPARTITION subpartition_name [ TABLESPACE tablespacename ] )
```

须知：

切割点的大小要位于正在被切割的分区的分区键范围内。

只能把一个分区切割成两个新分区。

Range 分区策略切割点是把当前分区以此切割点分割为两个分区（小于此分割点为一个分区，大于此分割点为另一个分区），所以 Range 分区策略切割点只能为一个。List 分区策略切割点可以为多个，但不超过 64 个，即把这些切割点从当前分区的边界值提取出来作为一个新分区，当前分区剩余边界值作为另一个新分区。

truncate_clause 子语法用于清空分区表中的指定分区。

```
TRUNCATE SUBPARTITION  { subpartition_name } [ UPDATE GLOBAL INDEX ]
```

参数说明

- table_name

  分区表名。

  取值范围：已存在的分区表名。

- subpartition_name

  二级分区名。

  取值范围：已存在的二级分区名。

- tablespacename

  指定分区要移动到哪个表空间。

  取值范围：已存在的表空间名。

示例

请参考 CREATE TABLE SUBPARTITION 的示例。

3.8.39 ALTER TABLESPACE

功能描述

修改表空间的属性。

注意事项

只有表空间的所有者或者被授予了表空间 ALTER 权限的用户有权限执行 ALTER TABLESPACE 命令，系统管理员默认拥有此权限。但要修改表空间的所有者，当前用户必须是该表空间的所有者或系统管理员，且该用户是新所有者角色的成员。

要修改表空间的所有者 A 为 B，则 A 必须是 B 的直接或者间接成员。

说明：如果 new_owner 与 old_owner 一致，此处不再校验当前执行操作的用户是否具有修改权限，而直接显示 ALTER 成功。

语法格式

重命名表空间的语法。

```
ALTER TABLESPACE tablespace_name
    RENAME TO new_tablespace_name;
```

设置表空间所有者的语法。

```
ALTER TABLESPACE tablespace_name
    OWNER TO new_owner;
```

设置表空间属性的语法。

```
ALTER TABLESPACE tablespace_name
    SET ( {tablespace_option = value} [, ... ] );
```

重置表空间属性的语法。

```
ALTER TABLESPACE tablespace_name
    RESET ( { tablespace_option } [, ...] );
```

设置表空间限额的语法。

```
ALTER TABLESPACE tablespace_name
    RESIZE MAXSIZE { UNLIMITED | 'space_size'};
```

参数说明

- tablespace_name

要修改的表空间。

取值范围：已存在的表空间名。

- new_tablespace_name

  表空间的新名称。

  新名称不能以“PG_”开头。

  取值范围：字符串，符合标识符命名规范。

- new_owner

  表空间的新所有者。

  取值范围：已存在的用户名。

- tablespace_option

  设置或者重置表空间的参数。

  取值范围：

  seq_page_cost：设置优化器计算一次顺序获取磁盘页面的开销。缺省为 1.0。

  random_page_cost：设置优化器计算一次非顺序获取磁盘页面的开销。缺省为 4.0。

  说明

  - random_page_cost 是相对于 seq_page_cost 的取值，等于或者小于 seq_page_cost 时毫无意义。

  默认值为 4.0 的前提条件是，优化器采用索引来扫描表数据，并且表数据在 cache 中命中率可以 90%左右。

  如果表数据空间要比物理内存小，那么减小该值到一个适当水平；相反地，如果表数据在 cache 中命中率要低于 90%，那么适当增大该值。

  如果采用了类似于 SSD 的随机访问代价较小的存储器，可以适当减小该值，以反映真正的随机扫描代价。

  value 的取值范围：正的浮点类型。

- RESIZE MAXSIZE

  重新设置表空间限额的数值。

  取值范围：

UNLIMITED，该表空间不设置限额。

由 space_size 来确定，其格式参考 CREATE TABLESPACE。

说明

- 若调整后的限额值比当前表空间实际使用的值要小，调整操作可以执行成功，后续用户需要将该表空间的使用值降低到新限额值之下，才能继续往该表空间中写入数据。

修改参数 MAXSIZE 时也可使用：

```
ALTER TABLESPACE tablespace_name RESIZE MAXSIZE
{'UNLIMITED' | 'space_size'};
```

示例

请参考 CREATE TABLESPACE 的示例。

相关命令

CREATE TABLESPACE，DROP TABLESPACE

3.8.40 ALTER TEXT SEARCH CONFIGURATION

功能描述

更改文本搜索配置的定义。用户可以将映射从字串类型调整为字典，或者改变配置的名称或者所有者，或者修改搜索配置的配置参数。

ADD MAPPING FOR 选项为文本搜索配置增加字串类型映射；如果 ADD MAPPING FOR 后面任何一个字串类型的映射已经存在于此文本搜索配置中，那么系统将会报错。

ALTER MAPPING FOR 选项会首先清除已有的字串类型映射，然后添加指定的字串类型映射。

ALTER MAPPING REPLACE … WITH … 与 ALTER MAPPING FOR … REPLACE … WITH … 选项会直接使用 new_dictionary 替换 old_dictionary。需要注意的是，只有 pg_ts_config_map 系统表中存在 maptokentype 与 old_dictionary 对应关系的元组时，才能更新成功，否则不会成功，也不会有任何提示信息返回。

DROP MAPPING FOR 选项会删除当前文本搜索配置中指定的字串类型映射。 如果没有指定 IF EXISTS 选项，当 DROP MAPPING FOR 选项指定的字串类型映射在文本搜索配置中不存在时，数据库会报错。

注意事项

当一个搜索配置已经被引用(如被用来创建索引),则不允许用户修改此文本搜索配置。

要使用 ALTER TEXT SEARCH CONFIGURATION，用户必须是配置的所有者。

语法格式

增加文本搜索配置字串类型映射语法

```
ALTER TEXT SEARCH CONFIGURATION name
ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ];
```

修改文本搜索配置字典语法

```
ALTER TEXT SEARCH CONFIGURATION name
ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary;
```

更改文本搜索配置字典语法

```
ALTER TEXT SEARCH CONFIGURATION name
ALTER MAPPING REPLACE old_dictionary WITH new_dictionary;
```

删除文本搜索配置字串类型映射语法

```
ALTER TEXT SEARCH CONFIGURATION name
DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ];
```

重命名文本搜索配置所有者语法

```
ALTER TEXT SEARCH CONFIGURATION name OWNER TO new_owner;
```

重命名文本搜索配置名称语法

```
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name;
```

重命名文本搜索配置命名空间语法

```
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema;
```

修改文本搜索配置属性语法

```
ALTER TEXT SEARCH CONFIGURATION name SET ( {configuration_option = value}
[, ...] )
```

重置文本搜索配置属性语法

```
ALTER TEXT SEARCH CONFIGURATION name RESET ( {configuration_option} [, ...] );
```

参数说明

- name

已有文本搜索配置的名称（可以有模式修饰）。

- token_type

  与配置的语法解析器关联的字串类型的名称。详细信息参见解析器。

- dictionary_name

  文本搜索字典名称。 如果有多个字典，则它们会按指定的顺序搜索。

- old_dictionary

  映身中拟被替换的文本搜索字典名称。

- new_dictionary

  替换 old_dictionary 的文本搜索字典的名称。

- new_owner

  文本搜索配置的新所有者。

- new_name

  文本搜索配置的新名称。

- new_schema

  文本搜索配置的新模式名。

- configuration_option

  文本搜索配置项。详细信息参见 CREATE TEXT SEARCH CONFIGURATION。

- value

  文本搜索配置项的值。

示例

```
--创建文本搜索配置。
postgres=# CREATE TEXT SEARCH CONFIGURATION english_1 (parser=default);
CREATE TEXT SEARCH CONFIGURATION

--增加文本搜索配置字串类型映射语法。
postgres=# ALTER TEXT SEARCH CONFIGURATION english_1 ADD MAPPING FOR word WITH
simple,english_stem;
ALTER TEXT SEARCH CONFIGURATION
```

```
--增加文本搜索配置字串类型映射语法。
postgres=# ALTER TEXT SEARCH CONFIGURATION english_1 ADD MAPPING FOR email WITH
english_stem, french_stem;
ALTER TEXT SEARCH CONFIGURATION

--查询文本搜索配置相关信息。
postgres=# SELECT b.cfgname,a.maptokentype,a.mapseqno,a.mapdict,c.dictname
FROM pg_ts_config_map a,pg_ts_config b, pg_ts_dict c WHERE a.mapcfg=b.oid AND
a.mapdict=c.oid AND b.cfgname='english_1' ORDER BY 1,2,3,4,5;
  cfgname   | maptokentype | mapseqno | mapdict |   dictname
-----------+--------------+----------+---------+--------------
 english_1 |            2 |        1 |    3765 | simple
 english_1 |            2 |        2 |   12960 | english_stem
 english_1 |            4 |        1 |   12960 | english_stem
 english_1 |            4 |        2 |   12964 | french_stem
(4 rows)

--增加文本搜索配置字串类型映射语法。
postgres=# ALTER TEXT SEARCH CONFIGURATION english_1 ALTER MAPPING REPLACE
french_stem with german_stem;
ALTER TEXT SEARCH CONFIGURATION

--查询文本搜索配置相关信息。
postgres=# SELECT b.cfgname,a.maptokentype,a.mapseqno,a.mapdict,c.dictname
FROM pg_ts_config_map a,pg_ts_config b, pg_ts_dict c WHERE a.mapcfg=b.oid AND
a.mapdict=c.oid AND b.cfgname='english_1' ORDER BY 1,2,3,4,5;
  cfgname   | maptokentype | mapseqno | mapdict |   dictname
-----------+--------------+----------+---------+--------------
 english_1 |            2 |        1 |    3765 | simple
 english_1 |            2 |        2 |   12960 | english_stem
 english_1 |            4 |        1 |   12960 | english_stem
 english_1 |            4 |        2 |   12966 | german_stem
(4 rows)
```

示例

请参见 CREATE TEXT SEARCH CONFIGURATION 的示例。

相关命令

CREATE TEXT SEARCH CONFIGURATION , DROP TEXT SEARCH CONFIGURATION

3.8.41 ALTER TEXT SEARCH DICTIONARY

功能描述

修改全文检索词典的相关定义，包括参数、名称、所有者以及模式等。

注意事项

预定义词典不支持 ALTER 操作。

只有词典的所有者可以执行 ALTER 操作，系统管理员默认拥有此权限。

创建或修改词典之后，任何对于 filepath 路径下用户自定义的词典定义文件的修改，将不会影响到数据库中的词典。如果需要在数据库中使用这些修改，需使用 ALTER TEXT SEARCH DICTIONARY 语句更新对应词典的定义文件。

语法格式

修改词典定义。

```
ALTER TEXT SEARCH DICTIONARY name (
    option [ = value ] [, ... ]
);
```

重命名词典。

```
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name;
```

设置词典的所属模式。

```
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema;
```

修改词典的所属者。

```
ALTER TEXT SEARCH DICTIONARY name OWNER TO new_owner;
```

参数说明

- name

  已存在的词典名（可指定模式名，否则默认在当前模式下）。

  取值范围：已存在的词典名。

- option

  要修改的参数名。与 template 对应，不同的词典类型具有不同的参数列表，且与指定顺序无关。详细参数说明请见 option。

  说明

- 不支持修改词典的 TEMPLATE 参数值。
- 不支持仅修改 FILEPATH 参数而不修改对应的词典定义文件参数。
- 词典定义文件的文件名仅支持小写字母、数据、下划线混合。

- value

要修改的参数值。如果省略等号（=）和 value，则表示删除该 option 的先前设置，使用默认值。

取值范围：对应 option 定义。

- new_name

词典的新名称。

取值范围：符合标识符命名规范的字符串，且最大长度不超过 63 个字符。

- new_owner

词典新的所有者。

取值范围：已存在的用户。

- new_schema

词典的新模式。

取值范围：已存在的模式。

示例

```
--更改 Snowball 类型字典的停用词定义，其他参数保持不变。
postgres=# ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian, FilePath
= 'file:///home/dicts' );

--更改 Snowball 类型字典的 Language 参数，并删除停用词定义。
postgres=# ALTER TEXT SEARCH DICTIONARY my_dict ( Language = dutch,  StopWords );

--更新词典定义，不实际更改任何内容。
postgres=# ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

相关命令

CREATE TEXT SEARCH DICTIONARY，DROP TEXT SEARCH DICTIONARY

3.8.42 ALTER TRIGGER

功能描述

修改触发器名称。

注意事项

只有触发器所在表的所有者可以执行 ALTER TRIGGER 操作，系统管理员默认拥有此权限。

语法格式

```
ALTER TRIGGER trigger_name ON table_name RENAME TO new_name;
```

参数说明

- trigger_name

  要修改的触发器名称。

  取值范围：已存在的触发器。

- table_name

  要修改的触发器所在的表名称。

  取值范围：已存在的含触发器的表。

- new_name

  修改后的新名称。

  取值范围：符合标识符命名规范的字符串，最大长度不超过 63 个字符，且不能与所在表上其他触发器同名。

示例

请参见 CREATE TRIGGER 的示例。

相关命令

CREATE TRIGGER，DROP TRIGGER，ALTER TABLE

3.8.43 ALTER TYPE

功能描述

修改一个类型的定义。

注意事项

只有类型的所有者或者被授予了类型 ALTER 权限的用户可以执行 ALTER TYPE 命令，系统管理员默认拥有此权限。但要修改类型的所有者或者修改类型的模式，当前用户必须是该类型的所有者或者系统管理员，且该用户是新所有者角色的成员。

语法格式

修改类型。

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE
| RESTRICT ]
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER }
neighbor_enum_value ]
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value

where action is one of:
    ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE |
RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE
collation ] [ CASCADE | RESTRICT ]
```

给复合类型增加新的属性。

```
ALTER TYPE name ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ]
[ CASCADE | RESTRICT ]
```

从复合类型删除一个属性。

```
ALTER TYPE name DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
```

改变一种复合类型中某个属性的类型。

```
ALTER TYPE name ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type
[ COLLATE collation ] [ CASCADE | RESTRICT ]
```

改变类型的所有者。

```
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

改变类型的名称或是一个复合类型中的一个属性的名称。

```
ALTER TYPE name RENAME TO new_name
```

```
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE | RESTRICT ]
```

将类型移至一个新的模式中。

```
ALTER TYPE name SET SCHEMA new_schema
```

为枚举类型增加一个新值。

```
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER } neighbor_enum_value ]
```

重命名枚举类型的一个标签值。

```
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value
```

参数说明

- name

  一个需要修改的现有的类型的名称(可以有模式修饰) 。

- new_name

  该类型的新名称。

- new_owner

  新所有者的用户名 。

- new_schema

  该类型的新模式 。

- attribute_name

  拟增加、更改或删除的属性的名称。

- new_attribute_name

  拟改名的属性的新名称。

- data_type

  拟新增属性的数据类型或是拟更改的属性的新类型名。

- new_enum_value

  枚举类型新增加的标签值，是一个非空的长度不超过 63 个字节的字符串。

- neighbor_enum_value

一个已有枚举标签值，新值应该被增加在紧接着该枚举值之前或者之后的位置上。

- existing_enum_value

  现有的要重命名的枚举值，是一个非空的长度不超过 63 个字节的字符串

- CASCADE

  自动级联更新需更新类型以及相关联的记录和继承它们的子表。

- RESTRICT

  如果需联动更新类型是已更新类型的关联记录，则拒绝更新。这是缺省选项。

  须知

  - ADD ATTRIBUTE、DROP ATTRIBUTE 和 ALTER ATTRIBUTE 选项可以组合成一个列表同时处理。例如，在一条命令中同时增加几个属性或是更改几个属性的类型是可以实现的。
  - 要修改一个类型的模式，必须在新模式上拥有 CREATE 权限。
  - 要修改所有者，必须是新的所有角色的直接或间接成员，并且该成员必须在此类型的模式上有 CREATE 权限。（这些限制强制了修改所有者不会做任何通过删除和重建类型不能做的事情。不过，系统管理员可以以任何方式修改任意类型的所有权。）要增加一个属性或是修改一个属性的类型，也必须有该类型的 USAGE 权限。

示例

请参考 CREATE TYPE 的示例。

相关命令

CREATE TYPE，DROP TYPE

3.8.44 ALTER USER

功能描述

修改数据库用户的属性。

注意事项

ALTER USER 中修改的会话参数只针对指定的用户，且在下一次会话中有效。

语法格式

修改用户的权限等信息。

```
ALTER USER user_name [ [ WITH ] option [ ... ] ];
```

其中 option 子句为：

```
{CREATEDB | NOCREATEDB}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {AUDITADMIN | NOAUDITADMIN}
    | {SYSADMIN | NOSYSADMIN}
    | {MONADMIN | NOMONADMIN}
    | {OPRADMIN | NOOPRADMIN}
    | {POLADMIN | NOPOLADMIN}
    | {USEFT | NOUSEFT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | {PERSISTENCE | NOPERSISTENCE}
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD { 'password' [ EXPIRED ] | DISABLE |
EXPIRED }
    | [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY { 'password' [ REPLACE
'old_password' | EXPIRED ]
| DISABLE }
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
    | NODE GROUP logic_cluster_name
    | ACCOUNT { LOCK | UNLOCK }
    | PGUSER
```

修改用户名。

```
ALTER USER user_name
    RENAME TO new_name;
```

修改与用户关联的指定会话参数值。

```
ALTER USER user_name [ IN DATABASE database_name ]
    SET configuration_parameter {{ TO | = } { value | DEFAULT }|FROM CURRENT};
```

重置与用户关联的指定会话参数值。

```
ALTER USER user_name
    [ IN DATABASE database_name ] RESET {configuration_parameter|ALL};
```

参数说明

- user_name

  现有用户名。

  取值范围：已存在的用户名。

- new_password

  新密码。

  密码规则如下：

  - 不能与当前密码相同。
  - 密码默认不少于 8 个字符。
  - 不能与用户名及用户名倒序相同。
  - 至少包含大写字母（A-Z）、小写字母（a-z）、数字（0-9）、非字母数字字符（限定为~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。

  取值范围：字符串。

- old_password

  旧密码。

- ACCOUNT LOCK | ACCOUNT UNLOCK

  - ACCOUNT LOCK：锁定帐户，禁止登录数据库。
  - ACCOUNT UNLOCK：解锁帐户，允许登录数据库。

- PGUSER

  当前版本不允许修改用户的 PGUSER 属性。

  其他参数请参见 CREATE ROLE 和 ALTER ROLE 的参数说明。

示例

请参考 CREATE USER 的示例。

相关命令

CREATE ROLE，CREATE USER，DROP USER

3.8.45 ALTER USER MAPPING

功能描述

更改一个用户映射的定义。

注意事项

当在 OPTIONS 中出现 password 选项时，需要保证 GBase 8s 数据库集群每个节点的 $GAUSSHOME/bin 目录下存在 usermapping.key.cipher 和 usermapping.key.rand 文件，如果不存在这两个文件，请使用 gs_guc 工具生成，并发布到每个节点的$GAUSSHOME/bin 目录下。

语法格式

```
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

在 OPTIONS 选项里，ADD、SET 和 DROP 指定要执行的操作，未指定时默认为 ADD 操作。option 和 value 为对应操作的参数及参数值。

参数说明

- user_name

  该映射的用户名。

  CURRENT_USER 和 USER 匹配当前用户的名称。PUBLIC 被用来匹配系统中所有当前以及未来的用户名。

- server_name

  该用户映射的服务器名。

- OPTIONS

  为该用户映射更改选项。新选项会覆盖任何之前指定的选项。ADD、SET 和 DROP 指定要被执行的动作。如果没有显式地指定操作，将假定为 ADD。选项名称必须为唯一，该服务器的外部数据包装器也会验证选项。

  - oracle_fdw 支持的 options 包括：

- user

oracle server 的用户名。

- password

oracle 用户对应的密码。

- mysql_fdw 支持的 options 包括：

  - username

MySQL Server/MariaDB 的用户名。

  - password

MySQL Server/MariaDB 用户对应的密码。

- postgres_fdw 支持的 options 包括：

  - user

远端数据库用户的用户名。

  - password

远端数据库用户对应的密码。

相关命令

CREATE USER MAPPING，DROP USER MAPPING

3.8.46 ALTER VIEW

功能描述

ALTER VIEW 更改视图的各种辅助属性。（如果用户是更改视图的查询定义，要使用 CREATE OR REPLACE VIEW。）

注意事项

只有视图的所有者或者被授予了视图 ALTER 权限的用户才可以执行 ALTER VIEW 命令，系统管理员默认拥有该权限。针对所要修改属性的不同，对其还有以下权限约束：

修改视图的模式，当前用户必须是视图的所有者或者系统管理员，且要有新模式的 CREATE 权限。

修改视图的所有者，当前用户必须是视图的所有者或者系统管理员，且该用户必须是新所有者角色的成员，并且此角色必须有视图所在模式的 CREATE 权限。

语法格式

设置视图列的默认值。

```
ALTER VIEW [ IF EXISTS ] view_name
    ALTER [ COLUMN ] column_name SET DEFAULT expression;
```

取消列视图列的默认值。

```
ALTER VIEW [ IF EXISTS ] view_name
    ALTER [ COLUMN ] column_name DROP DEFAULT;
```

修改视图的所有者。

```
ALTER VIEW [ IF EXISTS ] view_name
    OWNER TO new_owner;
```

重命名视图。

```
ALTER VIEW [ IF EXISTS ] view_name
    RENAME TO new_name;
```

设置视图的所属模式。

```
ALTER VIEW [ IF EXISTS ] view_name
    SET SCHEMA new_schema;
```

设置视图的选项。

```
ALTER VIEW [ IF EXISTS ] view_name
    SET ( { view_option_name [ = view_option_value ] } [, ... ] );
```

重置视图的选项。

```
ALTER VIEW [ IF EXISTS ] view_name
    RESET ( view_option_name [, ... ] );
```

参数说明

- IF EXISTS

  使用这个选项，如果视图不存在时不会产生错误，仅有会有一个提示信息。

- view_name

  视图名称，可以用模式修饰。

  取值范围：字符串，符合标识符命名规范。

- column_name

  可选的名称列表，视图的字段名。如果没有给出，字段名取自查询中的字段名。

  取值范围：字符串，符合标识符命名规范。

- SET/DROP DEFAULT

  设置或删除一个列的缺省值，该参数暂无实际意义。

- new_owner

  视图新所有者的用户名称。

- new_name

  视图的新名称。

- new_schema

  视图的新模式。

- view_option_name [= view_option_value]

  该子句为视图指定一个可选的参数。

  目前 view_option_name 支持的参数仅有 security_barrier，当 VIEW 试图提供行级安全时，应使用该参数。

  取值范围：Boolean 类型，TRUE、FALSE。

示例

```
--创建一个由 c_customer_sk 小于 150 的内容组成的视图。
postgres=# CREATE VIEW tpcds.customer_details_view_v1 AS
    SELECT * FROM tpcds.customer
    WHERE c_customer_sk < 150;

--修改视图名称。
postgres=# ALTER VIEW tpcds.customer_details_view_v1 RENAME TO
customer_details_view_v2;

--修改视图所属 schema。
postgres=# ALTER VIEW tpcds.customer_details_view_v2 SET schema public;

--删除视图。
postgres=# DROP VIEW public.customer_details_view_v2;
```

相关命令

CREATE VIEW，DROP VIEW

3.8.47 ANALYZE | ANALYSE

功能描述

用于收集与数据库中普通表内容相关的统计信息，统计结果存储在系统表 PG_STATISTIC 下。执行计划生成器会使用这些统计数据，以确定最有效的执行计划。

如果没有指定参数，ANALYZE 会分析当前数据库中的每个表和分区表。同时也可以通过指定 table_name、column 和 partition_name 参数把分析限定在特定的表、列或分区表中。

ANALYZE|ANALYSE VERIFY 用于检测数据库中普通表（行存表、列存表）的数据文件是否损坏。

注意事项

ANALYZE 非临时表不能在一个匿名块、事务块、函数或存储过程内被执行。支持存储过程中 ANALYZE 临时表，不支持统计信息回滚操作。

ANALYZE VERIFY 操作处理的大多为异常场景检测需要使用 RELEASE 版本。ANALYZE VERIFY 场景不触发远程读，因此远程读参数不生效。对于关键系统表出现错误被系统检测出页面损坏时，将直接报错不再继续检测。

如果没有指定参数，ANALYZE 处理当前数据库里用户拥有相应权限的每个表。如果参数中指定了表，ANALYZE 只处理指定的表。

要对一个表进行 ANALYZE 操作，通常用户必须是表的所有者或者被授予了指定表 VACUUM 权限的用户，默认系统管理员有该权限。数据库的所有者允许对数据库中除了共享目录以外的所有表进行 ANALYZE 操作（该限制意味着只有系统管理员才能真正对一个数据库进行 ANALYZE 操作）。ANALYZE 命令会跳过那些用户没有权限的表。

语法格式

收集表的统计信息。

```
{ ANALYZE | ANALYSE } [ VERBOSE ]
    [ table_name [ ( column_name [, ...] ) ] ];
```

收集分区表的统计信息。

```
{ ANALYZE | ANALYSE } [ VERBOSE ]
    [ table_name [ ( column_name [, ...] ) ] ]
```

```
    PARTITION ( patrition_name ) ;
```

说明

- 普通分区表目前支持针对某个分区的统计信息的语法，但功能上不支持针对某个分区的统计信息收集。

收集外部表的统计信息。

```
{ ANALYZE | ANALYSE } [ VERBOSE ]
    { foreign_table_name | FOREIGN TABLES };
```

收集多列统计信息。

```
{ANALYZE | ANALYSE} [ VERBOSE ]
    table_name (( column_1_name, column_2_name [, ...] ));
```

说明

- 收集多列统计信息时，请设置 GUC 参数 default_statistics_target 为负数，以使用百分比采样方式。
- 每组多列统计信息最多支持 32 列。
- 不支持收集多列统计信息的表：系统表。

检测当前库的数据文件。

```
{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE};
```

说明

- Fast 模式校验时，需要对校验的表有并发的 DML 操作，会导致校验过程中有误报的问题，因为当前 Fast 模式是直接从磁盘上读取，并发有其他线程修改文件时，会导致获取的数据不准确，建议离线操作。
- 支持对全库进行操作，由于涉及的表较多，建议以重定向保存结果 gsql -d database -p port -f “verify.sql” > verify_warning.txt 2>&1。
- 对外提示 NOTICE 只核对外可见的表，内部表的检测会包含在它所依赖的外部表，不对外显示和呈现。
- 此命令的处理可容错 ERROR 级别的处理。由于 debug 版本的 Assert 可能会导致 core 无法继续执行命令，建议在 release 模式下操作。
- 对于全库操作时，当关键系统表出现损坏则直接报错，不再继续执行。

检测表和索引的数据文件

```
{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE} table_name|index_name [CASCADE];
```

说明

- 支持对普通表的操作和索引表的操作，但不支持对索引表 index 使用 CASCADE 操作。原因是由于 CASCADE 模式用于处理主表的所有索引表，当单独对索引表进行检测时，无需使用 CASCADE 模式。
- 对于主表的检测会同步检测主表的内部表，例如 toast 表、cudesc 表等。
- 当提示索引表损坏时，建议使用 reindex 命令进行重建索引操作。

检测表分区的数据文件

```
{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE} table_name PARTITION
{(patrition_name)}[CASCADE];
```

说明

- 支持对表的单独分区进行检测操作，但不支持对索引表 index 使用 CASCADE 操作。

参数说明

- VERBOSE

  启用显示进度信息。

  说明：如果指定了 VERBOSE，ANALYZE 发出进度信息，表明目前正在处理的表。各种有关表的统计信息也会打印出来。

- table_name

  需要分析的特定表的表名（可能会带模式名），如果省略，将对数据库中的所有表（非外部表）进行分析。

  对于 ANALYZE 收集统计信息，目前仅支持行存表、列存表。

  取值范围：已有的表名。

- column_name，column_1_name，column_2_name

  需要分析特定列的列名，默认为所有列。

  取值范围：已有的列名。

- partition_name

  如果 table 为分区表，在关键字 PARTITION 后面指定分区名 partition_name 表示分析该

分区表的统计信息。目前语法上支持分区表做 ANALYZE，但功能实现上暂不支持对指定分区统计信息的分析。

取值范围：表的某一个分区名。

- index_name

  需要分析的特定索引表的表名（可能会带模式名）。

  取值范围：已有的表名。

- FAST|COMPLETE

  对于行存表，FAST 模式下主要对于行存表的 CRC 和 page header 进行校验，如果校验失败则会告警; 而 COMPLETE 模式下，则主要对行存表的指针、tuple 进行解析校验。 对于列存表，FAST 模式下主要对于列存表的 CRC 和 magic 进行校验，如果校验失败则会告警; 而 COMPLETE 模式下，则主要对列存表的 CU 进行解析校验。

- CASCADE

  CASCADE 模式下会对当前表的所有索引进行检测处理。

示例

```
— 创建表。
postgres=# CREATE TABLE customer_info
(
WR_RETURNED_DATE_SK       INTEGER                       ,
WR_RETURNED_TIME_SK       INTEGER                       ,
WR_ITEM_SK                INTEGER               NOT NULL,
WR_REFUNDED_CUSTOMER_SK   INTEGER
)
;
— 创建分区表。
postgres=# CREATE TABLE customer_par
(
WR_RETURNED_DATE_SK       INTEGER                       ,
WR_RETURNED_TIME_SK       INTEGER                       ,
WR_ITEM_SK                INTEGER               NOT NULL,
WR_REFUNDED_CUSTOMER_SK   INTEGER
)
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
PARTITION P1 VALUES LESS THAN(2452275),
```

```
PARTITION P2 VALUES LESS THAN(2452640),
PARTITION P3 VALUES LESS THAN(2453000),
PARTITION P4 VALUES LESS THAN(MAXVALUE)
)
ENABLE ROW MOVEMENT;
— 使用 ANALYZE 语句更新统计信息。
postgres=# ANALYZE customer_info;
postgres=# ANALYZE customer_par;
— 使用 ANALYZE VERBOSE 语句更新统计信息，并输出表的相关信息。
postgres=# ANALYZE VERBOSE customer_info;
INFO:  analyzing "cstore.pg_delta_3394584009"(cn_5002 pid=53078)
INFO:  analyzing "public.customer_info"(cn_5002 pid=53078)
INFO:  analyzing "public.customer_info" inheritance tree(cn_5002 pid=53078)
ANALYZE
```

说明

- 若环境若有故障，需查看数据库主节点的 log。

```
— 删除表。
postgres=# DROP TABLE customer_info;
postgres=# DROP TABLE customer_par;
```

3.8.48 BEGIN

功能描述

BEGIN 可以用于开始一个匿名块，也可以用于开始一个事务。本节描述用 BEGIN 开始匿名块的语法，以 BEGIN 开始事务的语法见 START TRANSACTION。

匿名块是能够动态地创建和执行过程代码的结构，而不需要以持久化的方式将代码作为数据库对象储存在数据库中。

注意事项

无。

语法格式

开启匿名块

```
[DECLARE [declare_statements]]
BEGIN
execution_statements
END;
/
```

开启事务

```
BEGIN [ WORK | TRANSACTION ]
  [
    {
       ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE |
REPEATABLE READ }
       | { READ WRITE | READ ONLY }
      } [, ...]
  ];
```

参数说明

- declare_statements

  声明变量，包括变量名和变量类型，如“sales_cnt int”。

- execution_statements

  匿名块中要执行的语句。

  取值范围：DML 操作(数据操纵操作：select、insert、delete、update)或系统表中已注册的函数名称。

示例

无。

相关命令

START TRANSACTION

3.8.49 CALL

功能描述

使用 CALL 命令可以调用已定义的函数和存储过程。

注意事项

函数或存储过程的所有者、被授予了函数或存储过程 EXECUTE 权限的用户或被授予 EXECUTE ANY FUNCTION 权限的用户有权调用函数或存储过程，系统管理员默认拥有此权限。

语法格式

```
CALL [ schema. ] func_name ( param_expr );
```

参数说明

- schema

  函数或存储过程所在的模式名称。

- param_expr

参数列表可以用符号":="或者"=>"将参数名和参数值隔开，这种方法的好处是参数可以以任意顺序排列。若参数列表中仅出现参数值，则参数值的排列顺序必须和函数或存储过程定义时的相同。

取值范围：已存在的函数参数名称或存储过程参数名称。

说明

- 参数可以包含入参（参数名和类型之间指定"IN"关键字）和出参（参数名和类型之间指定"OUT"关键字），使用 CALL 命令调用函数或存储过程时，对于非重载的函数，参数列表必须包含出参，出参可以传入一个变量或者任一常量，详见示例。对于重载的 package 函数，参数列表里可以忽略出参，忽略出参时可能会导致函数找不到。包含出参时，出参只能是常量。

示例

```
--创建一个函数 func_add_sql，计算两个整数的和，并返回结果。
postgres=# CREATE FUNCTION func_add_sql(num1 integer, num2 integer) RETURN
integer
AS
BEGIN
RETURN num1 + num2;
END;
/

--按参数值传递。
postgres=# CALL func_add_sql(1, 3);

--使用命名标记法传参。
postgres=# CALL func_add_sql(num1 => 1,num2 => 3);
postgres=# CALL func_add_sql(num2 := 2, num1 := 3);

--删除函数。
postgres=# DROP FUNCTION func_add_sql;

--创建带出参的函数。
```

```
postgres=# CREATE FUNCTION func_increment_sql(num1 IN integer, num2 IN integer,
res OUT integer)
RETURN integer
AS
BEGIN
res := num1 + num2;
END;
/

--出参传入常量。
postgres=# CALL func_increment_sql(1,2,1);

--删除函数。
postgres=# DROP FUNCTION func_increment_sql;
```

3.8.50 CHECKPOINT

功能描述

检查点（CHECKPOINT）是一个事务日志中的点，所有数据文件都在该点被更新以反映日志中的信息，所有数据文件都将被刷新到磁盘。

设置事务日志检查点。预写式日志（WAL）缺省时在事务日志中每隔一段时间放置一个检查点。可以使用 gs_guc 命令设置相关运行时参数（checkpoint_segments、checkpoint_timeout 和 incremental_checkpoint_timeout）来调整这个原子化检查点的间隔。

注意事项

只有系统管理员和运维管理员可以调用 CHECKPOINT。

CHECKPOINT 强制立即进行检查，而不是等到下一次调度时的检查点。

语法格式

CHECKPOINT;

参数说明

无

示例

```
--设置检查点。
postgres=# CHECKPOINT;
```

3.8.51 CLEAN CONNECTION

功能描述

用来清理数据库连接。允许在节点上清理指定数据库的指定用户的相关连接。

注意事项

- GBase 8s 下不支持指定节点，仅支持 TO ALL。
- 在非 force 模式下，该功能不清理连接；在 fore 模式下，可以清理正在使用的正常连接。

语法格式

```
CLEAN CONNECTION
      TO { COORDINATOR ( nodename [, ... ] ) | NODE ( nodename [, ... ] )| ALL
[ CHECK ] [ FORCE ] }
      [ FOR DATABASE dbname ]
      [ TO USER username ];
```

参数说明

- CHECK

仅在节点列表为 TO ALL 时可以指定。如果指定该参数，会在清理连接之前检查数据库是否被其他会话连接访问。此参数主要用于 DROP DATABASE 之前的连接访问检查，如果发现有其他会话连接，则将报错并停止删除数据库。

- FORCE

仅在节点列表为 TO ALL 时可以指定，如果指定该参数，所有和指定 dbname 和 username 相关的线程都会收到 SIGTERM 信号，然后被强制关闭。

- ALL

删除指定节点上的连接。有三种场景：

删除指定 CN 上的连接，GBase 8s 不支持。

删除指定 DN 上的连接，GBase 8s 不支持。

删除所有节点上的连接(TO ALL)，GBase 8s 仅支持该场景。

- dbname

删除指定数据库上的连接。如果不指定，则删除所有数据库的连接。

取值范围：已存在数据库名。

- username

  删除指定用户上的连接。如果不指定，则删除所有用户的连接。

  取值范围：已存在的用户。

示例

```
--创建 jack 用户。
CREATE USER jack PASSWORD 'Bigdata123@';

--删除用户 jack 在数据库 template1 上的所有连接。
CLEAN CONNECTION TO ALL FOR DATABASE template1 TO USER jack;

--删除用户 jack 的所有连接。
CLEAN CONNECTION TO ALL TO USER jack;

--删除在数据库 gaussdb 上的所有连接。
CLEAN CONNECTION TO ALL FORCE FOR DATABASE gaussdb;

--删除用户 jack。
DROP USER jack;
```

3.8.52 CLOSE

功能描述

CLOSE 释放和一个游标关联的所有资源。

注意事项

不允许对一个已关闭的游标再做任何操作。

一个不再使用的游标应该尽早关闭。

当创建游标的事务用 COMMIT 或 ROLLBACK 终止之后，每个不可保持的已打开游标都隐含关闭。

当创建游标的事务通过 ROLLBACK 退出之后，每个可以保持的游标都将隐含关闭。

当创建游标的事务成功提交，可保持的游标将保持打开，直到执行一个明确的 CLOSE 或者客户端断开。

GBase 8s 没有明确打开游标的 OPEN 语句，因为一个游标在使用 CURSOR 命令定义的

时候就打开了。可以通过查询系统视图 pg_cursors 看到所有可用的游标。

语法格式

```
CLOSE { cursor_name | ALL } ;
```

参数说明

- cursor_name

  一个待关闭的游标名称。

- ALL

  关闭所有已打开的游标。

示例

请参考 FETCH 的示例。

相关命令

FETCH，MOVE

3.8.53 CLUSTER

功能描述

根据一个索引对表进行聚簇排序。

CLUSTER 指定 GBase 8s 通过索引名指定的索引聚簇由表名指定的表。表名上必须已经定义该索引。

当对一个表聚集后，该表将基于索引信息进行物理存储。聚集是一次性操作：当表被更新之后，更改的内容不会被聚集。也就是说，系统不会试图按照索引顺序对新的存储内容及更新记录进行重新聚集。

在对一个表聚簇之后，GBase 8s 会记录在哪个索引上建立了聚集。CLUSTER table_name 的聚集形式在之前的同一个索引的表上重新聚集。用户也可以用 ALTER TABLE 的 CLUSTER 或 SET WITHOUT CLUSTER 形式来设置索引来用于后续的聚集操作或清除任何之前的设置。

不含参数的 CLUSTER 会将当前用户所拥有的数据库中的先前做过聚簇的所有表重新处理，或者系统管理员调用的这些表。

在对一个表进行聚簇的时候，会在其上请求一个 ACCESS EXCLUSIVE 锁。这样就避免了在 CLUSTER 完成之前对此表执行其它的操作(包括读写)。

注意事项

只有行存 B-tree 索引支持 CLUSTER 操作。

如果用户只是随机访问表中的行，那么表中数据的实际存储顺序是无关紧要的。但是，如果对某些数据的访问多于其它数据，而且有一个索引将这些数据分组，那么将使用 CLUSTER 中会有所帮助。如果从一个表中请求一定索引范围的值，或者是一个索引值对应多行，CLUSTER 也会有助于应用，因为如果索引标识出第一匹配行所在的存储页，所有其它行也可能已经在同一个存储页里了，这样便节省了磁盘访问的时间，加速了查询。

在聚簇过程中，系统先创建一个按照索引顺序建立的表的临时拷贝。同时也建立表上的每个索引的临时拷贝。因此，需要磁盘上有足够的剩余空间， 至少是表大小和索引大小的和。

因为 CLUSTER 记忆聚集信息，可以在第一次的时候手工对表进行聚簇，然后设置一个类似 VACUUM 的时间，这样就可以周期地自动对表进行聚簇操作。

因为优化器记录着有关表的排序的统计，所以建议在新近聚簇的表上运行 ANALYZE。否则，优化器可能会选择很差劲的查询规划。

CLUSTER 不允许在事务中执行。

如果没有打开 xc_maintenance_mode 参数，那么 CLUSTER 操作将跳过所有系统表。

语法格式

对一个表进行聚簇排序。

```
CLUSTER [ VERBOSE ] table_name [ USING index_name ];
```

对一个分区进行聚簇排序。

```
CLUSTER [ VERBOSE ] table_name PARTITION ( partition_name ) [ USING index_name ];
```

对已做过聚簇的表重新进行聚簇。

```
CLUSTER [ VERBOSE ];
```

参数说明

- VERBOSE

  启用显示进度信息。

- table_name

  表名称。

取值范围：已存在的表名称。

- index_name

  索引名称。

  取值范围：已存在的索引名称。

- partition_name

  分区名称。

  取值范围：已存在的分区名称。

示例

```
-- 创建一个分区表。
postgres=# CREATE TABLE tpcds.inventory_p1
(
    INV_DATE_SK             INTEGER             NOT NULL,
    INV_ITEM_SK             INTEGER             NOT NULL,
    INV_WAREHOUSE_SK        INTEGER             NOT NULL,
    INV_QUANTITY_ON_HAND    INTEGER
)
PARTITION BY RANGE(INV_DATE_SK)
(
        PARTITION P1 VALUES LESS THAN(2451179),
        PARTITION P2 VALUES LESS THAN(2451544),
        PARTITION P3 VALUES LESS THAN(2451910),
        PARTITION P4 VALUES LESS THAN(2452275),
        PARTITION P5 VALUES LESS THAN(2452640),
        PARTITION P6 VALUES LESS THAN(2453005),
        PARTITION P7 VALUES LESS THAN(MAXVALUE)
);

-- 创建索引 ds_inventory_p1_index1。
postgres=# CREATE INDEX ds_inventory_p1_index1 ON tpcds.inventory_p1
(INV_ITEM_SK) LOCAL;

-- 对表 tpcds.inventory_p1 进行聚集。
postgres=# CLUSTER tpcds.inventory_p1 USING ds_inventory_p1_index1;

-- 对分区 p3 进行聚集。
postgres=# CLUSTER tpcds.inventory_p1 PARTITION (p3) USING
ds_inventory_p1_index1;
```

```
-- 对数据库中可以进行聚集的表进聚集。
postgres=# CLUSTER;

--删除索引。
postgres=# DROP INDEX tpcds.ds_inventory_p1_index1;

--删除分区表。
postgres=# DROP TABLE tpcds.inventory_p1;
```

优化建议

建议在新近聚簇的表上运行 ANALYZE。否则，优化器可能会选择很差劲的查询规划。

不允许在事务中执行 CLUSTER。

3.8.54 COMMENT

功能描述

定义或修改一个对象的注释。

注意事项

每个对象只存储一条注释，因此要修改一个注释，对同一个对象发出一条新的 COMMENT 命令即可。要删除注释，在文本字符串的位置写上 NULL 即可。当删除对象时，注释自动被删除掉。

目前注释浏览没有安全机制：任何连接到某数据库上的用户都可以看到所有该数据库对象的注释。共享对象（比如数据库、角色、表空间）的注释是全局存储的，连接到任何数据库的任何用户都可以看到它们。因此，不要在注释里存放与安全有关的敏感信息。

对大多数对象，只有对象的所有者或者被授予了对象 COMMENT 权限的用户可以设置注释，系统管理员默认拥有该权限。

角色没有所有者，所以 COMMENT ON ROLE 命令仅可以由系统管理员对系统管理员角色执行，有 CREATEROLE 权限的角色也可以为非系统管理员角色设置注释。系统管理员可以对所有对象进行注释。

语法格式

```
COMMENT ON
{
  AGGREGATE agg_name (agg_type [, ...] ) |
  CAST (source_type AS target_type) |
```

```
  COLLATION object_name |
  COLUMN { table_name.column_name | view_name.column_name } |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name ( [ {[ argmode ] [ argname ] argtype} [, ...] ] ) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SERVER object_name |
  TABLE object_name |
  TABLESPACE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TYPE object_name |
  VIEW object_name
}
   IS 'text';
```

参数说明

- agg_name

  聚集函数的名称。

- agg_type

  聚集函数参数的类型。

- source_type

  类型转换的源数据类型。

- target_type

  类型转换的目标数据类型。

- object_name

  对象名。

- table_name.column_name 、view_name.column_name

  定义/修改注释的列名称。前缀可加表名称或者视图名称。

- constraint_name

  定义/修改注释的表约束的名称。

- table_name

  表的名称。

- function_name

  定义/修改注释的函数名称。

- argname,argmode,argtype

  函数参数的模式、名称、类型。

- large_object_oid

  定义/修改注释的大对象的 OID 值。

- operator_name

  操作符名称。

- left_type,right_type

  操作参数的数据类型（可以用模式修饰）。当前置或者后置操作符不存在时，可以增加 NONE 选项。

- trigger_name

  触发器名称。

- text

  注释。

示例

```
postgres=# CREATE TABLE tpcds.customer_demographics_t2
(
    CD_DEMO_SK                INTEGER               NOT NULL,
    CD_GENDER                 CHAR(1)                       ,
    CD_MARITAL_STATUS         CHAR(1)                       ,
    CD_EDUCATION_STATUS       CHAR(20)                      ,
    CD_PURCHASE_ESTIMATE      INTEGER                       ,
    CD_CREDIT_RATING          CHAR(10)                      ,
    CD_DEP_COUNT              INTEGER                       ,
    CD_DEP_EMPLOYED_COUNT     INTEGER                       ,
    CD_DEP_COLLEGE_COUNT      INTEGER
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
;

-- 为 tpcds.customer_demographics_t2.cd_demo_sk 列加注释。
postgres=# COMMENT ON COLUMN tpcds.customer_demographics_t2.cd_demo_sk IS
'Primary key of customer demographics table.';

--创建一个由 c_customer_sk 小于 150 的内容组成的视图。
postgres=# CREATE VIEW tpcds.customer_details_view_v2 AS
    SELECT *
    FROM tpcds.customer
    WHERE c_customer_sk < 150;

-- 为 tpcds.customer_details_view_v2 视图加注释。
postgres=# COMMENT ON VIEW tpcds.customer_details_view_v2 IS 'View of customer
detail';

-- 删除 view。
postgres=# DROP VIEW tpcds.customer_details_view_v2;

-- 删除 tpcds.customer_demographics_t2。
postgres=# DROP TABLE tpcds.customer_demographics_t2;
```

3.8.55COMMIT | END

功能描述

通过 COMMIT 或者 END 可完成提交事务的功能，即提交事务的所有操作。

注意事项

执行 COMMIT 这个命令的时候，命令执行者必须是该事务的创建者或系统管理员，且创建和提交操作只能在同一个会话中。

语法格式

```
{ COMMIT | END } [ WORK | TRANSACTION ] ;
```

参数说明

- COMMIT | END

  提交当前事务，让所有当前事务的更改为其他事务可见。

- WORK | TRANSACTION

  可选关键字，除了增加可读性没有其他任何作用。

示例

```
--创建表。
postgres=# CREATE TABLE tpcds.customer_demographics_t2
(
    CD_DEMO_SK                INTEGER               NOT NULL,
    CD_GENDER                 CHAR(1)                       ,
    CD_MARITAL_STATUS         CHAR(1)                       ,
    CD_EDUCATION_STATUS       CHAR(20)                      ,
    CD_PURCHASE_ESTIMATE      INTEGER                       ,
    CD_CREDIT_RATING          CHAR(10)                      ,
    CD_DEP_COUNT              INTEGER                       ,
    CD_DEP_EMPLOYED_COUNT     INTEGER                       ,
    CD_DEP_COLLEGE_COUNT      INTEGER
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
;

--开启事务。
postgres=# START TRANSACTION;

--插入数据。
postgres=# INSERT INTO tpcds.customer_demographics_t2 VALUES(1,'M','U','DOCTOR
DEGREE', 1200, 'GOOD', 1, 0, 0);
postgres=# INSERT INTO tpcds.customer_demographics_t2 VALUES(2,'F','U','MASTER
DEGREE', 300, 'BAD', 1, 0, 0);

--提交事务，让所有更改永久化。
```

```
postgres=# COMMIT;

--查询数据。
postgres=# SELECT * FROM tpcds.customer_demographics_t2;

--删除表 tpcds.customer_demographics_t2。
postgres=# DROP TABLE tpcds.customer_demographics_t2;
```

相关命令

ROLLBACK

3.8.56 COMMIT PREPARED

功能描述

提交一个早先为两阶段提交准备好的事务。

注意事项

该功能仅在维护模式（GUC 参数 xc_maintenance_mode 为 on 时）下可用。该模式谨慎打开，一般供维护人员排查问题使用，一般用户不应使用该模式。

命令执行者必须是该事务的创建者或系统管理员，且创建和提交操作只能在同一个会话中。

事务功能由数据库自动维护，不应显式使用事务功能。

语法格式

```
COMMIT PREPARED transaction_id ;
```

参数说明

- transaction_id

  待提交事务的标识符。它不能和任何当前预备事务已经使用了的标识符同名。

示例

```
--提交标识符为的 trans_test 的事务。
postgres=# COMMIT PREPARED 'trans_test';
```

相关命令

PREPARE TRANSACTION，ROLLBACK PREPARED。

3.8.57 COPY

功能描述

通过 COPY 命令实现在表和文件之间拷贝数据。

COPY FROM 从一个文件拷贝数据到一个表，COPY TO 把一个表的数据拷贝到一个文件。

注意事项

当参数 enable_copy_server_files 关闭时，只允许初始用户执行 COPY FROM FILENAME 或 COPY TO FILENAME 命令，当参数 enable_copy_server_files 打开，允许具有 SYSADMIN 权限的用户或继承了内置角色 gs_role_copy_files 权限的用户执行，但默认禁止对数据库配置文件、密钥文件、证书文件和审计日志执行 COPY FROM FILENAME 或 COPY TO FILENAME，以防止用户越权查看或修改敏感文件。

COPY 只能用于表，不能用于视图。

COPY TO 需要读取的表的 select 权限，copy from 需要插入的表的 insert 权限。

如果声明了一个字段列表，COPY 将只在文件和表之间拷贝已声明字段的数据。如果表中有任何不在字段列表里的字段，COPY FROM 将为那些字段插入缺省值。

如果声明了数据源文件，服务器必须可以访问该文件；如果指定了 STDIN，数据将在客户前端和服务器之间流动，输入时，表的列与列之间使用 TAB 键分隔，在新的一行中以反斜杠和句点（\.）表示输入结束。

如果数据文件的任意行包含比预期多或者少的字段，COPY FROM 将抛出一个错误。

数据的结束可以用一个只包含反斜杠和句点（\.）的行表示。如果从文件中读取数据，数据结束的标记是不必要的；如果在客户端应用之间拷贝数据，必须要有结束标记。

COPY FROM 中\N 为空字符串，如果要输入实际数据值\N ，使用\\N。

COPY FROM 不支持在导入过程中对数据做预处理（比如说表达式运算、填充指定默认值等）。如果需要在导入过程中对数据做预处理，用户需先把数据导入到临时表中，然后执行 SQL 语句通过运算插入到表中，但此方法会导致 I/O 膨胀，降低导入性能。

COPY FROM 在遇到数据格式错误时会回滚事务，但没有足够的错误信息，不方便用户从大量的原始数据中定位错误数据。

COPY FROM/TO 适合低并发，本地小数据量导入导出。

目标表存在 trigger，支持 COPY 操作。

语法格式

从一个文件拷贝数据到一个表。

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ USING ] DELIMITERS 'delimiters' ]
    [ WITHOUT ESCAPING ]
    [ LOG ERRORS ]
    [ LOG ERRORS DATA ]
    [ REJECT LIMIT 'limit' ]
    [ [ WITH ] ( option [, ...] ) ]
    | copy_option
    | [ FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) ]
    | [ TRANSFORM ( { column_name [ data_type ] [ AS transform_expr ] } [, ...] ) ];
```

说明

- 语法中的 FIXED FORMATTER ({ column_name(offset, length) } [, …])以及 [(option [, …]) | copy_option […]] 可以任意排列组合。

把一个表的数据拷贝到一个文件。

```
COPY table_name [ ( column_name [, ...] ) ]
    TO { 'filename' | STDOUT }
    [ [ USING ] DELIMITERS 'delimiters' ]
    [ WITHOUT ESCAPING ]
    [ [ WITH ] ( option [, ...] ) ]
    | copy_option
    | [ FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) ];
COPY query
    TO { 'filename' | STDOUT }
    [ WITHOUT ESCAPING ]
    [ [ WITH ] ( option [, ...] ) ]
    | copy_option
    | [ FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) ];
```

说明

- COPY TO 语法形式约束如下：(query)与[USING] DELIMITER 不兼容，即若 COPY TO 的数据来自于一个 query 的查询结果，那么 COPY TO 语法不能再指定[USING] DELIMITERS 语法子句。

- 对于 FIXED FORMATTTER 语法后面跟随的 copy_option 是以空格进行分隔的。
- copy_option 是指 COPY 原生的参数形式，而 option 是兼容外表导入的参数形式。
- 语法中的 FIXED FORMATTER ({ column_name(offset, length) } [, …])以及 [(option [, …]) | copy_option […]] 可以任意排列组合。

其中可选参数 option 子句语法为：

```
FORMAT 'format_name'
| OIDS [ boolean ]
| DELIMITER 'delimiter_character'
| NULL 'null_string'
| HEADER [ boolean ]
| FILEHEADER 'header_file_string'
| FREEZE [ boolean ]
| QUOTE 'quote_character'
| ESCAPE 'escape_character'
| EOL 'newline_character'
| NOESCAPING [ boolean ]
| FORCE_QUOTE { ( column_name [, ...] ) | * }
| FORCE_NOT_NULL ( column_name [, ...] )
| ENCODING 'encoding_name'
| IGNORE_EXTRA_DATA [ boolean ]
| FILL_MISSING_FIELDS [ boolean ]
| COMPATIBLE_ILLEGAL_CHARS [ boolean ]
| DATE_FORMAT 'date_format_string'
| TIME_FORMAT 'time_format_string'
| TIMESTAMP_FORMAT 'timestamp_format_string'
| SMALLDATETIME_FORMAT 'smalldatetime_format_string'
```

其中可选参数 copy_option 子句语法为：

```
OIDS
| NULL 'null_string'
| HEADER
| FILEHEADER 'header_file_string'
| FREEZE
| FORCE NOT NULL column_name [, ...]
| FORCE QUOTE { column_name [, ...] | * }
| BINARY
| CSV
| QUOTE [ AS ] 'quote_character'
| ESCAPE [ AS ] 'escape_character'
```

```
| EOL 'newline_character'
| ENCODING 'encoding_name'
| IGNORE_EXTRA_DATA
| FILL_MISSING_FIELDS
| COMPATIBLE_ILLEGAL_CHARS
| DATE_FORMAT 'date_format_string'
| TIME_FORMAT 'time_format_string'
| TIMESTAMP_FORMAT 'timestamp_format_string'
| SMALLDATETIME_FORMAT 'smalldatetime_format_string'
```

参数说明

- query

  其结果将被拷贝。

  取值范围：一个必须用圆括弧包围的 SELECT 或 VALUES 命令。

- table_name

  表的名称（可以有模式修饰）。

  取值范围：已存在的表名。

- column_name

  可选的待拷贝字段列表。

  取值范围：如果没有声明字段列表，将使用所有字段。

- STDIN

  声明输入是来自标准输入。

- STDOUT

  声明输出打印到标准输出。

- FIXED

  打开字段固定长度模式。在字段固定长度模式下，不能声明 DELIMITER、NULL、CSV 选项。指定 FIXED 类型后，不能再通过 option 或 copy_option 指定 BINARY、CSV、TEXT 等类型。

  说明

  定长格式定义如下：

- 每条记录的每个字段长度相同。
- 长度不足的字段以空格填充，数字类型字段左对齐，字符字段右对齐。
- 字段和字段之间没有分隔符。

- [USING] DELIMITER 'delimiters'

  在文件中分隔各个字段的字符串，分隔符最大长度不超过 10 个字节。

  取值范围：不允许包含\.abcdefghijklmnopqrstuvwxyz0123456789 中的任何一个字符。

  缺省值：在文本模式下，缺省是水平制表符，在 CSV 模式下是一个逗号。

- WITHOUT ESCAPING

  在 TEXT 格式中，不对'\'和后面的字符进行转义。

  取值范围：仅支持 TEXT 格式。

- LOG ERRORS

  若指定，则开启对于 COPY FROM 语句中数据类型错误的容错机制。

  取值范围：仅支持导入（即 COPY FROM）时指定。

  说明

  此容错选项的使用限制如下：

  - 此容错机制仅捕捉 COPY FROM 过程中数据库主节点上数据解析过程中相关的数据类型错误（DATA_EXCEPTION）。
  - COPY 已有的容错选项（如 IGNORE_EXTRA_DATA）开启时，对应类型的错误会按照已有的方式处理而不会报出异常，因此错误表也不会有相应数据。

- LOG ERRORS DATA

  LOG ERRORS DATA 和 LOG ERRORS 的区别：

  LOG ERRORS DATA 会填充容错表的 rawrecord 字段。

  只有 supper 权限的用户才能使用 LOG ERRORS DATA 参数选项。

  注意

  - 使用“LOG ERRORS DATA”时，若错误内容过于复杂可能存在写入容错表失败的风险，导致任务失败。

- REJECT LIMIT 'imit'

与 LOG ERROR 选项共同使用，对 COPY FROM 的容错机制设置数值上限，一旦此 COPY FROM 语句错误数据超过选项指定条数，则会按照原有机制报错。

取值范围：正整数（1-INTMAX），'unlimited'（无最大值限制）

缺省值：若未指定 LOG ERRORS，则会报错；若指定 LOG ERRORS，则默认为 0。

说明

- 如上述 LOG ERRORS 中描述的容错机制，REJECT LIMIT 的计数也是按照执行 COPY FROM 的数据库主节点上遇到的解析错误数量计算，而不是数据库节点的错误数量。

- FORMATTER

在固定长度模式中，定义每一个字段在数据文件中的位置。按照 column(offset,length) 格式定义每一列在数据文件中的位置。

取值范围：

- offset 取值不能小于 0，以字节为单位。
- length 取值不能小于 0，以字节为单位。

所有列的总长度和不能大于 1GB。

文件中没有出现的列默认以空值代替。

- OPTION { option_name ' value ' }

用于指定兼容外表的各类参数。

- FORMAT

数据源文件的格式。

取值范围：CSV、TEXT、FIXED、BINARY。

- CSV 格式的文件，可以有效处理数据列中的换行符，但对一些特殊字符处理有欠缺。
- TEXT 格式的文件，可以有效处理一些特殊字符，但无法正确处理数据列中的换行符。
- FIXED 格式的文件，适用于每条数据的数据列都比较固定的数据，长度不足的列会添加空格补齐，过长的列则会自动截断。

- BINARY 形式的选项会使得所有的数据被存储/读作二进制格式而不是文本。 这比 TEXT 和 CSV 格式的要快一些，但是一个 BINARY 格式文件可移植性比较差。

缺省值：TEXT

- DELIMITER

指定数据文件行数据的字段分隔符。

说明

- 分隔符不能是\r 和\n。
- 分隔符不能和 null 参数相同，CSV 格式数据的分隔符不能和 quote 参数相同。
- TEXT 格式数据的分隔符不能包含：小写字母、数字和特殊字符.\。
- 数据文件中单行数据长度需<1GB，如果分隔符较长且数据列较多的情况下，会影响导出有效数据的长度。
- 分隔符推荐使用多字符和不可见字符。多字符例如'$^&'；不可见字符例如 0x07、0x08、0x1b 等。

取值范围：支持多字符分隔符，但分隔符不能超过 10 个字节。

缺省值：

- TEXT 格式的默认分隔符是水平制表符（tab）。
- CSV 格式的默认分隔符为“,”。
- FIXED 格式没有分隔符。

- NULL

用来指定数据文件中空值的表示。

取值范围：

- null 值不能是\r 和\n，最大为 100 个字符。
- null 值不能和分隔符、quote 参数相同。

缺省值：

- CSV 格式下默认值是一个没有引号的空字符串。
- 在 TEXT 格式下默认值是\N。

- HEADER

指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。header 只能用于 CSV、FIXED 格式的文件中。

在导入数据时，如果 header 选项为 on，则数据文本第一行会被识别为标题行，会忽略此行。如果 header 为 off，而数据文件中第一行会被识别为数据。

在导出数据时，如果 header 选项为 on，则需要指定 fileheader。如果 header 为 off，则导出数据文件不包含标题行。

取值范围：true/on、false/off。

缺省值：false

- QUOTE

  CSV 格式文件下的引号字符。

  缺省值：双引号

  说明

  - quote 参数不能和分隔符、null 参数相同。
  - quote 参数只能是单字节的字符。
  - 推荐不可见字符作为 quote，例如 0x07、0x08、0x1b 等。

- ESCAPE

  CSV 格式下，用来指定逃逸字符，逃逸字符只能指定为单字节字符。

  缺省值：双引号。当与 quote 值相同时，会被替换为'\0'。

- EOL 'newline_character'

  指定导入导出数据文件换行符样式。

  取值范围：支持多字符换行符，但换行符不能超过 10 个字节。常见的换行符，如\r、\n、\r\n（设成 0x0D、0x0A、0x0D0A 效果是相同的），其他字符或字符串，如$、#。

  说明

  - EOL 参数只能用于 TEXT 格式的导入导出，不支持 CSV 格式和 FIXED 格式导入。为了兼容原有 EOL 参数，仍然支持导出 CSV 格式和 FIXED 格式时指定 EOL 参数为 0x0D 或 0x0D0A。
  - EOL 参数不能和分隔符、null 参数相同。

- EOL 参数不能包含：.abcdefghijklmnopqrstuvwxyz0123456789。

- FORCE_QUOTE { (column_name [, …]) | * }

在 CSV COPY TO 模式下，强制在每个声明的字段周围对所有非 NULL 值都使用引号包围。NULL 输出不会被引号包围。

取值范围：已存在的字段。

- FORCE_NOT_NULL (column_name [, …])

在 CSV COPY FROM 模式下，指定的字段输入不能为空。

取值范围：已存在的字段。

- ENCODING

指定数据文件的编码格式名称，缺省为当前数据库编码格式。

- IGNORE_EXTRA_DATA

若数据源文件比外表定义列数多，是否会忽略对多出的列。该参数只在数据导入过程中使用。

取值范围：true/on、false/off。

- 参数为 true/on，若数据源文件比外表定义列数多，则忽略行尾多出来的列。
- 参数为 false/off，若数据源文件比外表定义列数多，会显示如下错误信息。

```
extra data after last expected column
```

缺省值：false。

须知

- 如果行尾换行符丢失，使两行变成一行时，设置此参数为 true 将导致后一行数据被忽略掉。

- COMPATIBLE_ILLEGAL_CHARS

导入非法字符容错参数。此语法仅对 COPY FROM 导入有效。

取值范围：true/on、false/off。

- 参数为 true/on，则导入时遇到非法字符进行容错处理，非法字符转换后入库，不报错，不中断导入。
- 参数为 false/off，导入时遇到非法字符进行报错，中断导入。

缺省值：false/off

说明

导入非法字符容错规则如下：

- 对于'\\0'，容错后转换为空格；
- 对于其他非法字符，容错后转换为问号；
- 若 compatible_illegal_chars 为 true/on 标识导入时对于非法字符进行容错处理，则若 NULL、DELIMITER、QUOTE、ESCAPE 设置为空格或问号则会通过如“illegal chars conversion may confuse COPY escape 0x20”等报错信息提示用户修改可能引起混淆的参数以避免导入错误。

- FILL_MISSING_FIELDS

当数据加载时，若数据源文件中一行的最后一个字段缺失的处理方式。

取值范围：true/on、false/off。

缺省值：false/off

- DATE_FORMAT

导入对于 DATE 类型指定格式。此参数不支持 BINARY 格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

取值范围：合法 DATE 格式。可参考时间和日期处理函数和操作符。

说明

- 对于 DATE 类型内建为 TIMESTAMP 类型的数据库，在导入的时候，若需指定格式，可以参考下面的 timestamp_format 参数。

- TIME_FORMAT

导入对于 TIME 类型指定格式。此参数不支持 BINARY 格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

取值范围：合法 TIME 格式，不支持时区。可参考时间和日期处理函数和操作符。

- TIMESTAMP_FORMAT

导入对于 TIMESTAMP 类型指定格式。此参数不支持 BINARY 格式，会报“cannot specify

bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

取值范围：合法 TIMESTAMP 格式，不支持时区。可参考时间和日期处理函数和操作符。

- SMALLDATETIME_FORMAT

导入对于 SMALLDATETIME 类型指定格式。此参数不支持 BINARY 格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

取值范围：合法 SMALLDATETIME 格式。可参考[时间和日期处理函数和操作符](时间和日期处理函数和操作符.html)。

- COPY_OPTION { option_name ' value ' }

用于指定 COPY 原生的各类参数。

- NULL null_string

用来指定数据文件中空值的表示。

须知

➢ 在使用 COPY FROM 的时候，任何匹配这个字符串的字符串将被存储为 NULL 值，所以应该确保指定的字符串和 COPY TO 相同。

取值范围：

■ null 值不能是\r 和\n，最大为 100 个字符。

■ null 值不能和分隔符、quote 参数相同。

缺省值：

■ 在 TEXT 格式下默认值是\N。

■ CSV 格式下默认值是一个没有引号的空字符串。

- HEADER

指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。header 只能用于 CSV、FIXED 格式的文件中。

在导入数据时，如果 header 选项为 on，则数据文本第一行会被识别为标题行，会忽略此行。如果 header 为 off，而数据文件中第一行会被识别为数据。

在导出数据时，如果 header 选项为 on，则需要指定 fileheader。如果 header 为 off，则导出数据文件不包含标题行。

- FILEHEADER

  导出数据时用于定义标题行的文件，一般用来描述每一列的数据信息。

  须知

  - 仅在 header 为 on 或 true 的情况下有效。
  - fileheader 指定的是绝对路径。
  - 该文件只能包含一行标题信息，并以换行符结尾，多余的行将被丢弃（标题信息不能包含换行符）。
  - 该文件包括换行符在内长度不超过 1M。

- FREEZE

  将 COPY 加载的数据行设置为已经被 frozen，就像这些数据行执行过 VACUUM FREEZE。

  这是一个初始数据加载的性能选项。仅当以下三个条件同时满足时，数据行会被 frozen：

  - 在同一事务中 create 或 truncate 这张表之后执行 COPY。
  - 当前事务中没有打开的游标。
  - 当前事务中没有原有的快照。

  说明

  - COPY 完成后，所有其他会话将会立刻看到这些数据。但是这违反了 MVCC 可见性的一般原则，用户应当了解这样会导致潜在的风险。

- FORCE NOT NULL column_name [, …]

  在 CSV COPY FROM 模式下，指定的字段不为空。若输入为空，则将视为长度为 0 的字符串。

  取值范围：已存在的字段。

- FORCE QUOTE { column_name [, …] | * }

  在 CSV COPY TO 模式下，强制在每个声明的字段周围对所有非 NULL 值都使用引号包围。NULL 输出不会被引号包围。

  取值范围：已存在的字段。

- BINARY

使用二进制格式存储和读取，而不是以文本的方式。在二进制模式下，不能声明 DELIMITER、NULL、CSV 选项。指定 BINARY 类型后，不能再通过 option 或 copy_option 指定 CSV、FIXED、TEXT 等类型。

- CSV

打开逗号分隔变量（CSV）模式。指定 CSV 类型后，不能再通过 option 或 copy_option 指定 BINARY、FIXED、TEXT 等类型。

- QUOTE [AS] 'quote_character'

CSV 格式文件下的引号字符。

缺省值：双引号。

说明

➢ quote 参数不能和分隔符、null 参数相同。

➢ quote 参数只能是单字节的字符。

推荐不可见字符作为 quote，例如 0x07、0x08、0x1b 等。

- ESCAPE \[AS\] 'escape_character'

CSV 格式下，用来指定逃逸字符，逃逸字符只能指定为单字节字符。

默认值为双引号。当与 quote 值相同时，会被替换为'\\0'。

- EOL 'newline_character'

指定导入导出数据文件换行符样式。

取值范围：支持多字符换行符，但换行符不能超过 10 个字节。常见的换行符，如\\r、\\n、\\r\\n（设成 0x0D、0x0A、0x0D0A 效果是相同的），其他字符或字符串，如$、\#。

说明

➢ EOL 参数只能用于 TEXT 格式的导入导出，不支持 CSV 格式和 FIXED 格式。为了兼容原有 EOL 参数，仍然支持导出 CSV 格式和 FIXED 格式时指定 EOL 参数为 0x0D 或 0x0D0A。

➢ EOL 参数不能和分隔符、null 参数相同。

➢ EOL 参数不能包含：.abcdefghijklmnopqrstuvwxyz0123456789。

- ENCODING 'encoding_name'

  指定文件编码格式名称。

  取值范围：有效的编码格式。

  缺省值：当前编码格式。

- IGNORE_EXTRA_DATA

  指定当数据源文件比外表定义列数多时，忽略行尾多出来的列。该参数只在数据导入过程中使用。

  若不使用该参数，在数据源文件比外表定义列数多，会显示如下错误信息。

```
extra data after last expected column
```

- COMPATIBLE_ILLEGAL_CHARS

  指定导入时对非法字符进行容错处理，非法字符转换后入库。不报错，不中断导入。此参数不支持 BINARY 格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

  若不使用该参数，导入时遇到非法字符进行报错，中断导入。

  说明

  - 导入非法字符容错规则如下：
  - 对于'\0'，容错后转换为空格；
  - 对于其他非法字符，容错后转换为问号；
  - 若 compatible_illegal_chars 为 true/on 标识，导入时对于非法字符进行容错处理，则若 NULL、DELIMITER、QUOTE、ESCAPE 设置为空格或问号则会通过如“illegal chars conversion may confuse COPY escape 0x20”等报错信息提示用户修改可能引起混淆的参数以避免导入错误。

- FILL_MISSING_FIELDS

  当数据加载时，若数据源文件中一行的最后一个字段缺失的处理方式。

  取值范围：true/on、false/off。

  缺省值：false/off。

  须知

- 目前 COPY 指定此 Option 实际不会生效，即不会有相应的容错处理效果(不生效)。需要额外注意的是，打开此选项会导致解析器在数据库主节点数据解析阶段（即 COPY 错误表容错的涵盖范围）忽略此数据问题，而到数据库节点重新报错，从而使得 COPY 错误表（打开 LOG ERRORS REJECT LIMIT）在此选项打开的情况下无法成功捕获这类少列的数据异常。因此请不要指定此选项。

- DATE_FORMAT 'date_format_string'

导入对于 DATE 类型指定格式。此参数不支持 BINARY 格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

取值范围：合法 DATE 格式。可参考[时间和日期处理函数和操作符](时间和日期处理函数和操作符.html)

说明

- 对于 DATE 类型内建为 TIMESTAMP 类型的数据库，在导入的时候，若需指定格式，可以参考下面的 timestamp_format 参数。

- TIME_FORMAT 'time_format_string'

导入对于 TIME 类型指定格式。此参数不支持 BINARY 格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

取值范围：合法 TIME 格式，不支持时区。可参考时间和日期处理函数和操作符。

- TIMESTAMP_FORMAT 'timestamp_format_string'

导入对于 TIMESTAMP 类型指定格式。此参数不支持 BINARY 格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

取值范围：合法 TIMESTAMP 格式，不支持时区。可参考时间和日期处理函数和操作符。

- SMALLDATETIME_FORMAT 'smalldatetime_format_string'

导入对于 SMALLDATETIME 类型指定格式。此参数不支持 BINARY 格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对 COPY FROM 导入有效。

取值范围：合法 SMALLDATETIME 格式。可参考时间和日期处理函数和操作符。

- TRANSFORM ({ column_name [data_type] [AS transform_expr] } [, …])

指定表中各个列的转换表达式；其中 data_type 指定该列在表达式参数中的数据类型；transform_expr 为目标表达式，返回与表中目标列数据类型一致的结果值，表达式可参考表达式。

COPY FROM 能够识别的特殊反斜杠序列如下所示。

- \b：反斜杠 （ASCII 8）
- \f：换页（ASCII 12）
- \n：换行符 （ASCII 10）
- \r：回车符 （ASCII 13）
- \t：水平制表符 （ASCII 9）
- \v：垂直制表符 （ASCII 11）
- \digits：反斜杠后面跟着一到三个八进制数，表示 ASCII 值为该数的字符。
- \xdigits：反斜杠 x 后面跟着一个或两个十六进制位声明指定数值编码的字符。

示例

```
--将 tpcds.ship_mode 中的数据拷贝到/home/gbase/ds_ship_mode.dat 文件中。
postgres=# COPY tpcds.ship_mode TO '/home/gbase/ds_ship_mode.dat';

--将 tpcds.ship_mode 输出到 stdout。
postgres=# COPY tpcds.ship_mode TO stdout;

--创建 tpcds.ship_mode_t1 表。
postgres=# CREATE TABLE tpcds.ship_mode_t1
(
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
;
```

```
--从 stdin 拷贝数据到表 tpcds.ship_mode_t1。
postgres=# COPY tpcds.ship_mode_t1 FROM stdin;

--从/home/gbase/ds_ship_mode.dat 文件拷贝数据到表 tpcds.ship_mode_t1。
postgres=# COPY tpcds.ship_mode_t1 FROM '/home/gbase/ds_ship_mode.dat';

--从/home/gbase/ds_ship_mode.dat 文件拷贝数据到表 tpcds.ship_mode_t1，应用
TRANSFORM 表达式转换，取 SM_TYPE 列左边 10 个字符插入到表中。
postgres=# COPY tpcds.ship_mode_t1 FROM '/home/gbase/ds_ship_mode.dat'
TRANSFORM (SM_TYPE AS LEFT(SM_TYPE, 10));

--从/home/gbase/ds_ship_mode.dat 文件拷贝数据到表 tpcds.ship_mode_t1，使用参数
如下：导入格式为 TEXT（format 'text'），分隔符为'\t'（delimiter E'\t'），忽略
多余列（ignore_extra_data 'true'），不指定转义（noescaping 'true'）。
postgres=# COPY tpcds.ship_mode_t1 FROM '/home/gbase/ds_ship_mode.dat'
WITH(format 'text', delimiter E'\t', ignore_extra_data 'true', noescaping
'true');

--从/home/gbase/ds_ship_mode.dat 文件拷贝数据到表 tpcds.ship_mode_t1，使用参数
如下：导入格式为 FIXED（FIXED），指定定长格式（FORMATTER(SM_SHIP_MODE_SK(0, 2),
SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10), SM_CARRIER(61,20),
SM_CONTRACT(82,20))），忽略多余列（ignore_extra_data），有数据头（header）。
postgres=# COPY tpcds.ship_mode_t1 FROM '/home/gbase/ds_ship_mode.dat' FIXED
FORMATTER(SM_SHIP_MODE_SK(0, 2), SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30),
SM_CODE(50,10), SM_CARRIER(61,20), SM_CONTRACT(82,20)) header
ignore_extra_data;

--删除 tpcds.ship_mode_t1。
postgres=# DROP TABLE tpcds.ship_mode_t1;
```

3.8.58 CREATE AGGREGATE

功能描述

定义一个新的聚合函数。

语法格式

```
CREATE AGGREGATE name ( input_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
```

```
    [ , SORTOP = sort_operator ]
)

or the old syntax

CREATE AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
    [ , SORTOP = sort_operator ]
)
```

参数说明

- name

  要创建的聚合函数名(可以有模式修饰) 。

- input_data_type

  该聚合函数要处理的输入数据类型。要创建一个零参数聚合函数，可以使用*代替输入数据类型列表。 (count(*)就是这种聚合函数的一个实例。)

- base_type

  在以前的 CREATE AGGREGATE 语法中，输入数据类型是通过 basetype 参数指定的，而不是写在聚合的名称之后。 需要注意的是这种以前语法仅允许一个输入参数。 要创建一个零参数聚合函数，可以将 basetype 指定为“ANY”(而不是*)。

- sfunc

  将在每一个输入行上调用的状态转换函数的名称。 对于有 N 个参数的聚合函数，sfunc 必须有 +1 个参数，其中的第一个参数类型为 state_data_type，其余的匹配已声明的输入数据类型。 函数必须返回一个 state_data_type 类型的值。 这个函数接受当前状态值和当前输入数据，并返回下个状态值。

- state_data_type

  聚合的状态值的数据类型。

- ffunc

在转换完所有输入行后调用的最终处理函数，它计算聚合的结果。 此函数必须接受一个类型为 state_data_type 的参数。 聚合的输出数据 类型被定义为此函数的返回类型。 如果没有声明 ffunc 则使用聚合结果的状态值作为聚合的结果，且输出类型为 state_data_type。

- initial_condition

状态值的初始设置(值)。它必须是一个 state_data_type 类型可以接受的文本常量值。如果没有声明，状态值初始为 NULL 。

- sort_operator

用于 MIN 或 MAX 类型聚合的排序操作符。这个只是一个操作符名 (可以有模式修饰)。这个操作符假设接受和聚合一样的输入数据类型。

示例

```
postgres=# CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
);
```

相关链接

ALTER AGGREGATE、DROP AGGREGATE

3.8.59 CREATE AUDIT POLICY

功能描述

创建统一审计策略。

注意事项

只有 poladmin、sysadmin 或初始用户能进行此操作。

需要开启安全策略开关，即设置 GUC 参数 enable_security_policy=on，审计策略才可以生效。

语法格式

```
CREATE AUDIT POLICY [ IF NOT EXISTS ] policy_name { { privilege_audit_clause |
access_audit_clause } [ filter_group_clause ] [ ENABLE | DISABLE ] };
privilege_audit_clause:
PRIVILEGES { DDL | ALL } [ ON LABEL ( resource_label_name [, ... ] ) ]
```

```
access_audit_clause:
ACCESS { DML | ALL } [ ON LABEL ( resource_label_name [, ... ] ) ]
filter_group_clause:
FILTER ON { ( FILTER_TYPE ( filter_value [, ... ] ) ) [, ... ] }
```

DDL 支持:

```
{ ( ALTER | ANALYZE | COMMENT | CREATE | DROP | GRANT | REVOKE | SET | SHOW |
LOGIN_ACCESS | LOGIN_FAILURE | LOGOUT | LOGIN ) }
```

DML 支持:

```
{ ( COPY | DEALLOCATE | DELETE_P | EXECUTE | REINDEX | INSERT | PREPARE | SELECT
| TRUNCATE | UPDATE ) }
```

FILTER_TYPE 支持:

```
{ APP | ROLES | IP }
```

参数说明

- policy_name

  审计策略名称，需要唯一，不可重复；

  取值范围：字符串，要符合标识符的命名规范。

- DDL

  指的是针对数据库执行如下操作时进行审计，目前支持：CREATE、ALTER、DROP、ANALYZE、COMMENT、GRANT、REVOKE、SET、SHOW、LOGIN_ANY、LOGIN_FAILURE、LOGIN_SUCCESS、LOGOUT。

- ALL

  指的是上述 DDL 支持的所有对数据库的操作。

- resource_label_name

  资源标签名称。

- DML

  指的是针对数据库执行如下操作时进行审计，目前支持：SELECT、COPY、DEALLOCATE、DELETE、EXECUTE、INSERT、PREPARE、REINDEX、TRUNCATE、UPDATE。

- FILTER_TYPE

描述策略过滤的条件类型，包括 IP | APP | ROLES。

- filter_value

  指具体过滤信息内容。

- ENABLE|DISABLE

  可以打开或关闭统一审计策略。若不指定 ENABLE|DISABLE，语句默认为 ENABLE。

示例

```
--创建 dev_audit 和 bob_audit 用户。
postgres=# CREATE USER dev_audit PASSWORD 'dev@1234';
CREATE USER bob_audit password 'bob@1234';

--创建一个表 tb_for_audit
postgres=# CREATE TABLE tb_for_audit(col1 text, col2 text, col3 text);

--创建资源标签
postgres=# CREATE RESOURCE LABEL adt_lb0 add TABLE(tb_for_audit);

--对数据库执行 create 操作创建审计策略
postgres=# CREATE AUDIT POLICY adt1 PRIVILEGES CREATE;

--对数据库执行 select 操作创建审计策略
postgres=# CREATE AUDIT POLICY adt2 ACCESS SELECT;

--仅审计记录用户 dev_audit 和 bob_audit 在执行针对 adt_lb0 资源进行的 create 操作数据库创建审计策略
postgres=# CREATE AUDIT POLICY adt3 PRIVILEGES CREATE ON LABEL(adt_lb0) FILTER ON ROLES(dev_audit, bob_audit);

--仅审计记录用户 dev_audit 和 bob_audit,客户端工具为 psql 和 gsql，IP 地址为 '10.20.30.40'，'127.0.0.0/24'，在执行针对 adt_lb0 资源进行的 select、insert、delete 操作数据库创建审计策略。
postgres=# CREATE AUDIT POLICY adt4 ACCESS SELECT ON LABEL(adt_lb0), INSERT ON LABEL(adt_lb0), DELETE FILTER ON ROLES(dev_audit, bob_audit), APP(psql, gsql), IP('10.20.30.40', '127.0.0.0/24');
```

相关命令

ALTER AUDIT POLICY DROP AUDIT POLICY。

3.8.60 CREATE CAST

功能描述

定义一个用户自定义的转换。

语法格式

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name (argument_type [, ...])
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

参数说明

- source_type：转换的源数据类型。
- target_type：转换的目标数据类型。
- function_name(argument_type [, …])：用于执行转换的函数。这个函数名可以是用模式名修饰的。如果它没有用模式名修饰，那么该函数将从模式搜索路径中找出来。函数的结果数据类型必须匹配转换的目标类型。 它的参数在下面讨论。
- WITHOUT FUNCTION：表明源类型是对目标类型是二进制可强制转换的，所以没有函数需要执行此转换。
- WITH INOUT：表明转换是 I/O 转换，通过调用源数据类型的输出函数来执行，并将结果传给目标数据类型的输入函数。
- AS ASSIGNMENT：表示转换可以在赋值模式下隐含调用。
- AS IMPLICIT：表示转换可以在任何环境里隐含调用。

转换实现函数可以有一到三个参数。第一个参数的类型必须与转换的源类型相同的，或可以从转换的源类型二进制可强制转换的。第二个参数，如果存在，必须是 integer 类型；

它接收这些与目标类型相关联的类型修饰符，或者若什么都没有则是-1。第三个参数，如果存在，必须是 boolean 类型；若转换是一个显式类型转换则会收到 true，否则是 false。

一个转换函数的返回类型必须是与转换的目标类型相同或者对转换的目标类型二进制可强制转换 。

通常，一个转换必须有不同的源和目标数据类型。然而，若有多于一个参数的转换实现函数，则允许声明一个有相同的源和目标类型的转换。这用于表示系统目录中的特定类型的长度强制函数。命名的函数用于强制一个该类型的值为第二个参数给出的类型修饰符值。

如果一个类型转换的源类型和目标类型不同，并且接收多于一个参数，它就表示从一种类型转换成另外一种类型只用一个步骤，并且同时实施长度转换。如果没有这样的项可用，那么转换成一个使用了类型修饰词的类型将涉及两个步骤，一个是在数据类型之间转换，另外一个是施加修饰词指定的转换。

对域类型的转换目前没有作用。转换一般是针对域相关的所属数据类型。

示例

创建一个从类型 bigint 到类型 int4 指派映射，要通过使用函数 int4(bigint)（已在系统中预先定义）：

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

兼容性

CREATE CAST 指令符合 SQL 标准，除了 SQL 没有为二进制可强制转换类型或者实现函数的额外参数来实现功能。

3.8.61 CREATE CLIENT MASTER KEY

功能描述

创建一个客户端主密钥对象，该对象可用于加密 Column Encryption Key 对象。

注意事项

本语法属于全密态数据库特有语法。

当使用 gsql 连接数据库服务器时，需使用‘-C’参数，打开全密态数据库的开关，才能使用本语法。

由本语法创建的 CMK 对象中，仅存储从独立的密钥管理工具/服务/组件中读取密钥的方法，而不存储密钥本身。

语法格式

```
CREATE CLIENT MASTER KEY client_master_key_name
    [WITH] ( ['KEY_STORE' , 'KEY_PATH' , 'ALGORITHM'] );
```

参数说明

- client_master_key_name

  该参数作为密钥对象名，在同一命名空间下，需满足命名唯一性约束。

  取值范围：字符串，需符合标识符的命名规范。

- KEY_STORE

  指定管理 CMK 的密钥工具或组件；取值：目前仅支持 localkms。

- KEY_PATH

  KEY_STORE 负责管理多个 CMK 密钥，KEY_PATH 选项用于在 KEY_STORE 中唯一标识 CMK。取值类似："key_path_value"。

- ALGORITHM

  由本语法创建的用于加密 COLUMN ENCRYPTION KEY，该参数用于指定加密算法的类型。取值范围：RSA_2048、RSA_3072 和 SM2。

说明：密钥存储路径：默认情况下，localkms 将在$LOCALKMS_FILE_PATH 路径下生成/读取/删除密钥文件，用户可手动配置该环境变量。但是，用户也可以不用单独配置该环境变量，在尝试获取$LOCALKMS_FILE_PATH 失败时，localkms 会尝试获取$GAUSSHOME/etc/localkms/路径，如果该路径存在，则将其作为密钥存储路径。 密钥相关文件名：使用 CREATE CMK 语法时，localkms 将会创建四个与存储密钥相关的文件。示例：当 KEY_PATH = "key_path_value"，四个文件的名称分别为 key_path_value.pub、key_path_value.pub.rand、 key_path_value.priv、 key_path_value.priv.rand。 所以，为了能够成功创建密钥相关文件，在密钥存储路径下，应该保证没有已存在的与密钥相关文件名同名的文件。

示例

(1) 使用普通账户 alice，连接全密态数据库

```
gsql -U alice -h $host -p $port -d $database -C -r
```

(2) 使用本语法创建客户端加密主密钥(CMK)对象

```
postgres=> CREATE CLIENT MASTER KEY a_cmk WITH (KEY_STORE = localkms, KEY_PATH
= "key_path_value", ALGORITHM = RSA_2048);
postgres=> CREATE CLIENT MASTER KEY another_cmk WITH (KEY_STORE = localkms,
KEY_PATH = "another_path_value", ALGORITHM = SM2);
```

3.8.62 CREATE COLUMN ENCRYPTION KEY

功能描述

创建一个列加密密钥，该密钥可用于加密表中指定列。

注意事项

本语法属于全密态数据库特有语法。

当使用 gsql 连接数据库服务器时，需使用‘-C’参数，打开全密态数据库的开关，才能使用本语法。

由该语法创建 CEK 对象可用于列级加密。在定义表中列字段时，可指定一个 CEK 对象，用于加密该列。

语法格式

```
CREATE COLUMN ENCRYPTION KEY column_encryption_key_name
    [WITH] [VALUES] ( ['CLIENT_MASTER_KEY' , 'ALGORITHM'] );
```

参数说明

- column_encryption_key_name

  该参数作为密钥对象名，在同一命名空间下，需满足命名唯一性约束。

  取值范围：字符串，要符合标识符的命名规范。

- CLIENT_MASTER_KEY

  指定用于加密本 CEK 的 CMK，取值为：CMK 对象名，该 CMK 对象由 CREATE CLIENT MASTER KEY 语法创建。

- ALGORITHM

  指定该 CEK 将用于何种加密算法，取值范围为：AEAD_AES_256_CBC_HMAC_SHA256、AEAD_AES_128_CBC_HMAC_SHA256 和 SM4_SM3；

  须知：国密算法约束：由于 SM2、SM3、SM4 等算法属于中国国家密码标准算法，为规避法律风险，需配套使用。即如果将 CEK 用于 SM4_SM3 算法，则仅能使用 SM2 算法来对该 CEK 进行加密。

示例

```
--创建列加密密钥(CEK)
postgres=> CREATE COLUMN ENCRYPTION KEY a_cek WITH VALUES (CLIENT_MASTER_KEY =
a_cmk, ALGORITHM  = AEAD_AES_256_CBC_HMAC_SHA256);
CREATE COLUMN ENCRYPTION KEY
postgres=> CREATE COLUMN ENCRYPTION KEY another_cek WITH VALUES
(CLIENT_MASTER_KEY = a_cmk, ALGORITHM  = SM4_SM3);
CREATE COLUMN ENCRYPTION KEY
```

3.8.63 CREATE DATABASE

功能描述

创建一个新的数据库。缺省情况下新数据库将通过复制标准系统数据库 template0 来创建，且仅支持使用 template0 来创建。

注意事项

只有拥有 CREATEDB 权限的用户才可以创建新数据库，系统管理员默认拥有此权限。

不能在事务块中执行创建数据库语句。

在创建数据库过程中，出现类似“Permission denied”的错误提示，可能是由于文件系统上数据目录的权限不足。出现类似“No space left on device”的错误提示，可能是由于磁盘满引起的。

语法格式

```
CREATE DATABASE database_name
    [ [ WITH ] { [ OWNER [=] user_name ] |
                 [ TEMPLATE [=] template ] |
                 [ ENCODING [=] encoding ] |
                 [ LC_COLLATE [=] lc_collate ] |
                 [ LC_CTYPE [=] lc_ctype ] |
                 [ DBCOMPATIBILITY [=] compatibilty_type ] |
                 [ TABLESPACE [=] tablespace_name ] |
                 [ CONNECTION LIMIT [=] connlimit ]}[...] ];
```

参数说明

- database_name

  数据库名称。

  取值范围：字符串，要符合标识符的命名规范。

- OWNER [=] user_name

  数据库所有者。缺省时，新数据库的所有者是当前用户。

  取值范围：已存在的用户名。

- TEMPLATE [=] template

  模板名。即从哪个模板创建新数据库。GBase 8s 采用从模板数据库复制的方式来创建新的数据库。初始时，GBase 8s 包含两个模板数据库 template0、template1，以及一个默认的用户数据库 postgres。

  取值范围：仅 template0。

- ENCODING [=] encoding

  指定数据库使用的字符编码，可以是字符串（如'SQL_ASCII'）、整数编号。

  不指定时，默认使用模版数据库的编码。模板数据库 template0 和 template1 的编码默认与操作系统环境相关。template1 不允许修改字符编码，因此若要变更编码，请使用 template0 创建数据库。

  常用取值：GBK、UTF8、Latin1。

表 3-25 GBasec 8s 字符集

名称	描述	语言	是否服务器端？	ICU?	字节/字符	别名
BIG5	Big Five	繁体中文	否	否	1-2	WIN950，Windows950
EUC_CN	扩展 UNIX 编码-中国	简体中文	是	是	1-3	-
EUC_JP	扩展 UNIX 编码-日本	日文	是	是	1-3	-

名称	描述	语言	是否服务器端？	ICU?	字节/字符	别名
EUC_JIS_2004	扩展 UNIX 编码 - 日本，JIS X 0213	日文	是	否	1-3	-
EUC_KR	扩展 UNIX 编码-韩国	韩文	是	是	1-3	-
EUC_TW	扩展 UNIX 编码-台湾	繁体中文，台湾话	是	是	1-3	-
GB18030	国家标准	中文	是	否	1-4	-
GBK	扩展国家标准	简体中文	是	否	1-2	WIN936，Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	拉丁语/西里尔语	是	是	1	-
ISO_8859_6	ISO 8859-6, ECMA 114	拉丁语/阿拉伯语	是	是	1	-
ISO_8859_7	ISO 8859-7, ECMA 118	拉丁语/希腊语	是	是	1	-
ISO_8859_8	ISO 8859-8, ECMA 121	拉丁语/希伯	是	是	1	-

名称	描述	语言	是否服务器端？	ICU?	字节/字符	别名
		来语				
JOHAB	JOHAB	韩语	否	否	1-3	-
KOI8R	KOI8-R	西里尔语（俄语）	是	是	1	KOI8
KOI8U	KOI8-U	西里尔语（乌克兰语）	是	是	1	-
LATIN1	ISO 8859-1, ECMA 94	西欧	是	是	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	中欧	是	是	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	南欧	是	是	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	北欧	是	是	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	土耳其语	是	是	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	日耳曼语	是	是	1	ISO885910
LATIN7	ISO 8859-13	波罗的	是	是	1	ISO885913

名称	描述	语言	是否服务器端？	ICU?	字节/字符	别名
		海				
LATIN8	ISO 8859-14	凯尔特语	是	是	1	ISO885914
LATIN9	ISO 8859-15	带欧罗巴和口音的 LATIN 1	是	是	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	罗马尼亚语	是	否	1	ISO885916
MULE_INTERNAL	Mule 内部编码	多语种编辑器	是	否	1-4	-
SJIS	Shift JIS	日语	否	否	1-2	Mskanji，ShiftJIS，WIN932，Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	日语	否	否	1-2	-
SQL_ASCII	未指定（见文本）	任意	是	否	1	-
UHC	统一韩语编码	韩语	否	否	1-2	WIN949，Windows9

名称	描述	语言	是否服务器端？	ICU?	字节/字符	别名
						49
UTF8	Unicode, 8-bit	所有	是	是	1-4	Unicode
WIN866	Windows CP866	西里尔语	是	是	1	ALT
WIN874	Windows CP874	泰语	是	否	1	-
WIN1250	Windows CP1250	中欧	是	是	1	-
WIN1251	Windows CP1251	西里尔语	是	是	1	WIN
WIN1252	Windows CP1252	西欧	是	是	1	-
WIN1253	Windows CP1253	希腊语	是	是	1	-
WIN1254	Windows CP1254	土耳其语	是	是	1	-
WIN1255	Windows CP1255	希伯来语	是	是	1	-
WIN1256	Windows CP1256	阿拉伯语	是	是	1	-
WIN1257	Windows	波罗的	是	是	1	-

名称	描述	语言	是否服务器端？	ICU?	字节/字符	别名
	CP1257	海				
WIN1258	Windows CP1258	越南语	是	是	1	ABC, TCVN ，TCVN5712，VSCII

注意

并非所有的客户端 API 都支持上面列出的字符集。SQL_ASCII 设置与其他设置表现得相当不同。如果服务器字符集是 SQL_ASCII，服务器把字节值 0-127 根据 ASCII 标准解释，而字节值 128-255 则当作无法解析的字符。如果设置为 SQL_ASCII，就不会有编码转换。因此，这个设置基本不是用来声明所使用的指定编码，因为这个声明会忽略编码。在大多数情况下，如果你使用了任何非 ASCII 数据，那么使用 SQL_ASCII 设置都是不明智的，因为 GBase 8s 将无法帮助你转换或者校验非 ASCII 字符。

须知

- 指定新的数据库字符集编码必须与所选择的本地环境中（LC_COLLATE 和 LC_CTYPE）的设置兼容。
- 当指定的字符编码集为 GBK 时，部分中文生僻字无法直接作为对象名。这是因为 GBK 第二个字节的编码范围在 0x40-0x7E 之间时，字节编码与 ASCII 字符@A-Z[\]^_`a-z{|}重叠。其中@[\]^_'{|}是数据库中的操作符，直接作为对象名时，会语法报错。例如“侅”字，GBK16 进制编码为 0x8240，第二个字节为 0x40，与 ASCII “@”符号编码相同，因此无法直接作为对象名使用。如果确实要使用，可以在创建和访问对象时，通过增加双引号来规避这个问题。
- 若客户端编码为 A，服务器端编码为 B，则需要满足数据库中存在编码格式 A 与 B 的转换，例如：若服务器端编码为 gb18030，由于当前数据库不支持 gb18030 与 gbk 的相互转换，所以此时设置客户端编码格式为 gbk 时，会报错“Conversion between

GB18030 and GBK is not supported.”。数据库能够支持的所有的编码格式转换详见系统表 pg_conversion。

- LC_COLLATE [=] lc_collate

  指定新数据库使用的字符集。例如，通过 lc_collate = 'zh_CN.gbk'设定该参数。

  该参数的使用会影响到对字符串的排序顺序（如使用 ORDER BY 执行，以及在文本列上使用索引的顺序）。默认是使用模板数据库的排序顺序。

  取值范围：有效的排序类型。

- LC_CTYPE [=] lc_ctype

  指定新数据库使用的字符分类。例如，通过 lc_ctype = 'zh_CN.gbk'设定该参数。该参数的使用会影响到字符的分类，如大写、小写和数字。默认是使用模板数据库的字符分类。

  取值范围：有效的字符分类。

- DBCOMPATIBILITY [=] compatibility_type

  指定兼容的数据库的类型，默认兼容 O。

  取值范围：A、B、C、PG。分别表示兼容 O、MY、TD 和 POSTGRES。

  说明

  - A 兼容性下，数据库将空字符串作为 NULL 处理，数据类型 DATE 会被替换为 TIMESTAMP(0) WITHOUT TIME ZONE。
  - 将字符串转换成整数类型时，如果输入不合法，B 兼容性会将输入转换为 0，而其它兼容性则会报错。
  - B 和 PG 兼容性下，CHAR 和 VARCHAR 以字符为计数单位，其它兼容性以字节为计数单位。例如，对于 UTF-8 字符集，CHAR(3)在 B 和 PG 兼容性下能存放 3 个中文字符，而在其它兼容性下只能存放 1 个中文字符。

- TABLESPACE [=] tablespace_name

  指定数据库对应的表空间。

  取值范围：已存在表空间名。

- CONNECTION LIMIT [=] connlimit

  数据库可以接受的并发连接数。

须知

- 系统管理员不受此参数的限制。
- connlimit 数据库主节点单独统计，GBase 8s 整体的连接数 = connlimit * 当前正常数据库主节点个数。

取值范围：>=-1 的整数。默认值为-1，表示没有限制。

有关字符编码的一些限制：

- 若区域设置为 C（或 POSIX），则允许所有的编码类型，但是对于其他的区域设置，字符编码必须和区域设置相同。
- 若字符编码方式是 SQL_ASCII，并且修改者为管理员用户时，则字符编码可以和区域设置不相同。
- 编码和区域设置必须匹配模板数据库，除了将 template0 当作模板。 因为其他数据库可能会包含不匹配指定编码的数据，或者可能包含排序顺序受 LC_COLLATE 和 LC_CTYPE 影响的索引。复制这些数据会导致在新数据库中的索引失效。template0 是不包含任何会受到影响的数据或者索引。

示例

```
--创建 jim 和 tom 用户。
postgres=# CREATE USER jim PASSWORD 'xxxxxxxxx';
postgres=# CREATE USER tom PASSWORD 'xxxxxxxxx';

--创建一个 GBK 编码的数据库 music（本地环境的编码格式必须也为 GBK）。
postgres=# CREATE DATABASE music ENCODING 'GBK' template = template0;

--创建数据库 music2，并指定所有者为 jim。
postgres=# CREATE DATABASE music2 OWNER jim;

--用模板 template0 创建数据库 music3，并指定所有者为 jim。
postgres=# CREATE DATABASE music3 OWNER jim TEMPLATE template0;

--设置 music 数据库的连接数为 10。
postgres=# ALTER DATABASE music CONNECTION LIMIT= 10;

--将 music 名称改为 music4。
postgres=# ALTER DATABASE music RENAME TO music4;

--将数据库 music2 的所属者改为 tom。
```

```
postgres=# ALTER DATABASE music2 OWNER TO tom;

--设置 music3 的表空间为 PG_DEFAULT。
postgres=# ALTER DATABASE music3 SET TABLESPACE PG_DEFAULT;

--关闭在数据库 music3 上缺省的索引扫描。
postgres=# ALTER DATABASE music3 SET enable_indexscan TO off;

--重置 enable_indexscan 参数。
postgres=# ALTER DATABASE music3 RESET enable_indexscan;

--删除数据库。
postgres=# DROP DATABASE music2;
postgres=# DROP DATABASE music3;
postgres=# DROP DATABASE music4;

--删除 jim 和 tom 用户。
postgres=# DROP USER jim;
postgres=# DROP USER tom;

--创建兼容 TD 格式的数据库。
postgres=# CREATE DATABASE td_compatible_db DBCOMPATIBILITY 'C';

--创建兼容 A 格式的数据库。
postgres=# CREATE DATABASE ora_compatible_db DBCOMPATIBILITY 'A';

--删除兼容 TD、A 格式的数据库。
postgres=# DROP DATABASE td_compatible_db;
postgres=# DROP DATABASE ora_compatible_db;
```

相关命令

ALTER DATABASE，DROP DATABASE

优化建议

- create database

  事务中不支持创建 database。

- ENCODING LC_COLLATE LC_CTYPE

  当新建数据库 Encoding、LC-Collate 或 LC_Ctype 与模板数据库（SQL_ASCII）不匹配（为'GBK' /'UTF8'/'LATIN1'）时，必须指定 template [=] template0。

3.8.64 CREATE DATA SOURCE

功能描述

创建一个新的外部数据源对象，该对象用于定义 GBase 8s 要连接的目标库信息。

注意事项

Data Source 名称在数据库中需唯一，遵循标识符命名规范，长度限制为 63 字节，过长则会被截断。

只有系统管理员或初始用户才有权限创建 Data Source 对象。且创建该对象的用户为其默认属主。

当在 OPTIONS 中出现 password 选项时，需要保证 GBase 8s 每个节点的 $GAUSSHOME/bin 目录下存在 datasource.key.cipher 和 datasource.key.rand 文件，如果不存在这两个文件，请使用 gs_guc 工具生成并使用 gs_ssh 工具发布到 GBase 8s 每个节点的 $GAUSSHOME/bin 目录下。

语法格式

```
CREATE DATA SOURCE src_name
    [TYPE 'type_str']
    [VERSION {'version_str' | NULL}]
    [OPTIONS (optname 'optvalue' [, ...])];
```

参数说明

- src_name

  新建 Data Source 对象的名称，需在数据库内部唯一。

  取值范围：字符串，要符标识符的命名规范。

- TYPE

  新建 Data Source 对象的类型，可缺省。

  取值范围：空串或非空字符串。

- VERSION

  新建 Data Source 对象的版本号，可缺省或 NULL 值。

  取值范围：空串或非空字符串或 NULL。

- OPTIONS

Data Source 对象的选项字段，创建时可省略，如若指定，其关键字如下：

- optname

  选项名称。

  取值范围：dsn、 username、 password、 encoding。不区分大小写。

  dsn 对应 odbc 配置文件中的 DSN。

  username/password 对应连接目标库的用户名和密码。

  GBase 8s 在后台会对用户输入的 username/password 加密以保证安全性。该加密所需密钥文件需要使用 gs_guc 工具生成并使用 gs_ssh 工具发布到 GBase 8s 每个节点的 $GAUSSHOME/bin 目录下。username/password 不应当包含'encryptOpt'前缀，否则会被认为是加密后的密文。

  encoding 表示与目标库交互的字符串编码方式（含发送的 SQL 语句和返回的字符类型数据），此处创建对象时不检查 encoding 取值的合法性，能否正确编解码取决于用户提供的编码方式是否在数据库本身支持的字符编码范围内。

- optvalue

  选项值。

  取值范围：空或者非空字符串。

示例

```
--创建一个空的 Data Source 对象，不含任何信息。
postgres=# CREATE DATA SOURCE ds_test1;

--创建一个 Data Source 对象，含 TYPE 信息，VERSION 为 NULL。
postgres=# CREATE DATA SOURCE ds_test2 TYPE 'MPPDB' VERSION NULL;

--创建一个 Data Source 对象，仅含 OPTIONS。
postgres=# CREATE DATA SOURCE ds_test3 OPTIONS (dsn 'gbase', encoding 'utf8');

--创建一个 Data Source 对象，含 TYPE, VERSION, OPTIONS。
postgres=# CREATE DATA SOURCE ds_test4 TYPE 'unknown' VERSION '11.2.3' OPTIONS
(dsn 'GBase 8s', username 'userid', password 'pwd@123456', encoding '');

--删除 Data Source 对象。
postgres=# DROP DATA SOURCE ds_test1;
postgres=# DROP DATA SOURCE ds_test2;
```

```
postgres=# DROP DATA SOURCE ds_test3;
postgres=# DROP DATA SOURCE ds_test4;
```

3.8.65CREATE DIRECTORY

功能描述

使用 CREATE DIRECTORY 语句创建一个目录对象，该目录对象定义了服务器文件系统上目录的别名，用于存放用户使用的数据文件。

注意事项

当 enable_access_server_directory=off 时，只允许初始用户创建 directory 对象；当 enable_access_server_directory=on 时，具有 SYSADMIN 权限的用户和继承了内置角色 gs_role_directory_create 权限的用户可以创建 directory 对象。

创建用户默认拥有此路径的 READ 和 WRITE 操作权限。

目录的默认 owner 为创建 directory 的用户。

以下路径禁止创建：

- 路径含特殊字符。
- 路径是相对路径。
- 路径是符号连接。

创建目录时会进行以下合法性校验：

- 创建时会检查添加路径是否为操作系统实际存在路径，如不存在会提示用户使用风险。
- 创建时会校验数据库初始化（gbase）用户对于添加路径的权限(即操作系统目录权限，读/写/执行 - R/W/X)，如果权限不全，会提示用户使用风险。

在 GBase 8s 环境下用户指定的路径需要用户保证各节点上路径的一致性，否则在不同节点上执行会产生找不到路径的问题。

语法格式

```
CREATE [OR REPLACE] DIRECTORY directory_name
AS 'path_name';
```

参数说明

- directory_name

  目录名称。

取值范围：字符串，要符合标识符的命名规范。

- path_name

  操作系统的路径。

  取值范围：有效的操作系统路径。

示例

```
--创建目录。
postgres=# CREATE OR REPLACE DIRECTORY  dir  as '/tmp/';
```

相关命令

ALTER DIRECTORY，DROP DIRECTORY

3.8.66 CREATE EXTENSION

功能描述

安装一个扩展。

注意事项

CREATE EXTENSION 命令安装一个新的扩展到一个数据库中，必须保证没有同名的扩展已经被安装。

安装一个扩展意味着执行一个扩展的脚本文件，这个脚本会创建一个新的 SQL 实体，例如函数、数据类型、操作符、和索引支持的方法。

安装扩展需要有和创建他的组件对象相同的权限。对于大多数扩展这意味着需要超户或者数据库所有者的权限，对于后续的权限检查和该扩展脚本所创建的实体，运行 CREATE EXTENSION 命令的角色将变为扩展的所有者。

语法格式

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
[ WITH ] [ SCHEMA schema_name ]
[ VERSION version ]
[ FROM old_version ]
```

参数说明

- IF NOT EXISTS

  如果系统已经存在一个同名的扩展，不会报错。这种情况下会给出一个提示。请注意该参数不保证系统存在的扩展和现在脚本创建的扩展相同。

- extension_name

  将被安装扩展的名字。

- schema_name

  扩展的实例被安装在该模式下，扩展的内容可以被重新安装。指定的模式必须已经存在，如果没有指定，扩展的控制文件也不指定一个模式，这样将使用默认模式。

  注意

  - 扩展不认为它在任何模式里面：扩展在一个数据库范围内的名字是不受限制的，但是一个扩展的实例是属于一个模式的。

- version

  安装扩展的版本，可以写为一个标识符或者字符串。默认的版本在扩展的控制文件中指定。

- old_version

  如果想升级安装“old style”模块中没有的内容时，必须指定 FROM old_version。这个选项将指定 CREATE EXTENSION 运行一个安装脚本将新的内容安装到扩展中，而不是创建一个新的实体。注意 SCHEMA 参数将指定包括这些已存在实体的模式。

示例

在当前数据库安装 hstore 扩展：

```
CREATE EXTENSION hstore;
```

3.8.67 CREATE EVENT TRIGGER

功能描述

创建一个事件触发器，在指定事件发生发生时执行指定的事件触发器函数。

注意事项

只有超级用户或系统管理员才有权限创建事件触发器。

如果为同一事件定义了多个相同类型的事件触发器，则按事件触发器的名称字母顺序触发它们。

事件触发器会对 DDL 操作的性能有一定影响，影响程度取决于事件触发器的数量和执行函数的复杂程度。

语法格式

```
CREATE EVENT TRIGGER name
ON event
[ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ] EXECUTE PROCEDURE
function_name()
```

参数说明

- name

  事件触发器名称。

- filter_variable

  事件触发器用来做过滤的变量(目前仅支持 TAG)。

- event

  事件触发器支持的事件，目前支持 ddl_command_start、ddl_command_end、 sql_drop、table_ rewrite。

- function_name

  用户定义的函数，必须声明为不带参数并返回类型为 event_trigger，在事件触发器触发时执行。

示例

```
--创建事件触发器函数(用于 ddl_command_start、ddl_command_end 事件)
postgres=# create function test_event_trigger() returns event_trigger as $$
BEGIN
RAISE NOTICE 'test_event_trigger: % %', tg_event, tg_tag;
END
$$ language plpgsql;

--创建事件触发器函数(用于 sql_drop 事件)
postgres=# CREATE OR REPLACE FUNCTION drop_sql_command()
RETURNS event_trigger AS $$
BEGIN
RAISE NOTICE '% - sql_drop', tg_tag;
END;
$$ LANGUAGE plpgsql;
--创建事件触发器函数(用于 table_rewrite 事件)
postgres=# CREATE OR REPLACE FUNCTION test_evtrig_no_rewrite() RETURNS
event_trigger
```

```
LANGUAGE plpgsql AS $$
BEGIN
RAISE EXCEPTION 'rewrites not allowed';
END;
$$;
--创建事件类型为 ddl_command_start 的事件触发器
postgres=# create event trigger regress_event_trigger on ddl_command_start
execute procedure test_event_trigger();
--创建事件类型为 ddl_command_end 的事件触发器
postgres=# create event trigger regress_event_trigger_end on ddl_command_end
execute procedure test_event_trigger();
--创建事件类型为 sql_drop 的事件触发器
postgres=# CREATE EVENT TRIGGER sql_drop_command ON sql_drop
EXECUTE PROCEDURE drop_sql_command();
--创建事件类型为 table_rewrite 的事件触发器
postgres=# create event trigger no_rewrite_allowed on table_rewrite
when tag in ('alter table') execute procedure test_evtrig_no_rewrite();
penGauss=# create role regress_evt_user WITH ENCRYPTED PASSWORD 'EvtUser123';
postgres=# ALTER EVENT TRIGGER regress_event_trigger RENAME TO
regress_event_trigger_start;
--应该失败，事件触发器的 owner 只能为超级用户
postgres=# ALTER EVENT TRIGGER regress_event_trigger_start owner to
regress_evt_user;
postgres=# ALTER EVENT TRIGGER regress_event_trigger_start disable;
postgres=# ALTER EVENT TRIGGER regress_event_trigger_start enable always;
--删除事件触发器
postgres=# DROP EVENT TRIGGER regress_event_trigger_start;
postgres=# DROP EVENT TRIGGER regress_event_trigger_end;
postgres=# DROP EVENT TRIGGER sql_drop_command;
postgres=# DROP EVENT TRIGGER no_rewrite_allowed;
```

3.8.68 CREATE FOREIGN TABLE

功能描述

创建外表。

注意事项

外表中暂不支持使用系统列（如 tableoid、ctid 等），其中 Private 和 Shares 模式的外表，需要初始用户和运维模式下（operation_mode）的运维管理员权限。

语法格式

```
CREATE FOREIGN TABLE [ IF NOT EXISTS  ] table_name
( { column_name type_name POSITION ( offset, length ) [column_constraint ]
        | LIKE source_table | table_constraint } [, ...] )
        SEVER gsmpp_server
        OPTIONS (  { option_name ' value '  }  [, ...] )
        [ { WRITE ONLY  |  READ ONLY  }]
        [ WITH error_table_name | LOG INTO error_table_name ]
        [ REMOTE LOG 'name' ]
        [PER NODE REJECT LIMIT 'value']
        [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ];
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
( { column_name type_name
    [ { [CONSTRAINT constraint_name] NULL |
    [CONSTRAINT constraint_name] NOT NULL |
        column_constraint [...]} ] |
        table_constraint} [, ...] )
        SERVER server_name
        OPTIONS ( { option_name ' value ' } [, ...] )
        DISTRIBUTE BY {ROUNDROBIN | REPLICATION}
        [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
        [ PARTITION BY ( column_name ) [AUTOMAPPED]] ;
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
        ( [ { column_name type_name | LIKE source_table } [, ...] ] )
        SERVER server_name
        OPTIONS ( { option_name ' value ' } [, ...] )
        [ READ ONLY ]
        [ DISTRIBUTE BY {ROUNDROBIN} ]
        [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ];
```

这里 column_constraint 可以是:

```
[ CONSTRAINT constraint_name ]
{ PRIMARY KEY | UNIQUE }
[ NOT ENFORCED [ ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION ] |
ENFORCED ]
```

这里 table_constraint 可以是:

```
[ CONSTRAINT constraint_name ]
{ PRIMARY KEY | UNIQUE } ( column_name )
[ NOT ENFORCED [ ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION ] |
ENFORCED ]
```

参数说明

- IF NOT EXISTS

如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。

- table_name

外表的表名。

取值范围：字符串，要符合标识符的命名规范。

- column_name

外表中的字段名。

取值范围：字符串，要符合标识符的命名规范。

- type_name

字段的数据类型。

- SERVER server_name

外表的 server 名称。默认值为 mot_server。

- OPTIONS (option 'value' [, …])

选项与新外部表或外部表中的字段有关。允许的选项名称和值，是由每一个外部数据封装器指定的。 也是通过外部数据封装器的验证函数来验证。重复的选项名称是不被允许的(尽管表选项和表字段选项可以有相同的名字)。

  - oracle_fdw 支持的 options 包括：

    - table

oracle server 侧的表名。需要同 oracle 系统表中记录的表名完全一致，通常是由大写字符组成。

    - schema

表所对应的 schema（或 owner）。需要同 oracle 系统表中记录的表名完全一致，通常是由大写字符组成。

  - mysql_fdw 支持的 options 包括：

    - dbname

MySQL 的 database 名称。

- table_name

MySQL 侧的表名。

- postgres_fdw 支持的 options 包括：
  - schema_name

远端 server 的 schema 名称。如果不指定的话，将使用外表自身的 schema 名称作为远端的 schema 名称。

  - table_name

远端 server 的表名。如果不指定的话，将使用外表自身的表名作为远端的表名。

  - column_name

远端 server 的表的列名。如果不指定的话，将使用外表自身的列名作为远端的的表的列名。

- file_fdw 支持的 options 包括：
  - filename

指定要读取的文件，必需的参数，且必须是一个绝对路径名。

  - format

远端 server 的文件格式，支持 text/csv/binary/fixed 四种格式，和 COPY 语句的 FORMAT 选项相同。

  - header

指定的文件是否有标题行，与 COPY 语句的 HEADER 选项相同。

  - delimiter

指定文件的分隔符，与 COPY 的 DELIMITER 选项相同。

  - quote

指定文件的引用字符，与 COPY 的 QUOTE 选项相同。

  - escape

指定文件的转义字符，与 COPY 的 ESCAPE 选项相同。

  - null

指定文件的 null 字符串，与 COPY 的 NULL 选项相同。

- encoding

指定文件的编码，与 COPY 的 ENCODING 选项相同。

- force_not_null

这是一个布尔选项。如果为真，则声明字段的值不应该匹配空字符串（也就是，文件级别 null 选项）。与 COPY 的 FORCE_NOT_NULL 选项里的字段相同。

说明：file_fdw 更多使用请参见《GBase 8s V8.8.5_5.0.0_数据库管理指南》中“file_fdw”章节。

相关命令

ALTER FOREIGN TABLE，DROP FOREIGN TABLE

3.8.69 CREATE FUNCTION

功能描述

创建一个函数。

注意事项

如果创建函数时参数或返回值带有精度，不进行精度检测。

创建函数时，函数定义中对表对象的操作建议都显式指定模式，否则可能会导致函数执行异常。

在创建函数时，函数内部通过 SET 语句设置 current_schema 和 search_path 无效。执行完函数 search_path 和 current_schema 与执行函数前的 search_path 和 current_schema 保持一致。

如果函数参数中带有出参，SELECT 调用函数必须缺省出参，CALL 调用函数必须指定出参，对于调用重载的带有 PACKAGE 属性的函数，CALL 调用函数可以缺省出参，具体信息参见 CALL 的示例。

兼容 Postgresql 风格的函数或者带有 PACKAGE 属性的函数支持重载。在指定 REPLACE 的时候，如果参数个数、类型、返回值有变化，不会替换原有函数，而是会建立新的函数。

SELECT 调用可以指定不同参数来进行同名函数调用。由于语法不支持调用不带有 PACKAGE 属性的同名函数。

在创建 function 时，不能在 avg 函数外面嵌套其他 agg 函数或者其他系统函数。

新创建的函数默认会给 PUBLIC 授予执行权限（详见 GRANT）。用户可以选择收回

PUBLIC 默认执行权限，然后根据需要将执行权限授予其他用户，为了避免出现新函数能被所有人访问的时间窗口，应在一个事务中创建函数并且设置函数执行权限。

在函数内部调用其它无参数的函数时，可以省略括号，直接使用函数名进行调用。

兼容 Oracle 风格的函数支持参数注释的查看与导出、导入。

兼容 Oracle 风格的函数支持介于 IS/AS 与 plsql_body 之间的注释的查看与导出、导入。

语法格式

兼容 PostgreSQL 风格的创建自定义函数语法。

```
CREATE [ OR REPLACE  ] FUNCTION function_name
    ( [  { argname [ argmode  ] argtype [  { DEFAULT  | :=  | =  } expression  ]}
[, ...]  ] )
    [ RETURNS rettype [ DETERMINISTIC  ]
        | RETURNS TABLE (  { column_name column_type  }  [, ...] )]
    LANGUAGE lang_name
    [
        {IMMUTABLE  | STABLE  | VOLATILE}
        | {SHIPPABLE | NOT SHIPPABLE}
        | [ NOT  ] LEAKPROOF
        | WINDOW
        | {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
        | {[ EXTERNAL  ] SECURITY INVOKER  | [ EXTERNAL  ] SECURITY DEFINER |
AUTHID DEFINER  | AUTHID CURREN
T_USER}
        | {FENCED | NOT FENCED}
        | {PACKAGE}
        | COST execution_cost
        | ROWS result_rows
        | SET configuration_parameter { {TO | =} value | FROM CURRENT }
    ] [...]
    {
        AS 'definition'
        | AS 'obj_file', 'link_symbol'
    }
```

O 风格的创建自定义函数的语法。

```
CREATE [ OR REPLACE  ] FUNCTION function_name
    ( [  { argname [ argmode  ] argtype [  { DEFAULT | := | =  } expression  ] }
[, ...]  ] )
```

```
    RETURN rettype [ DETERMINISTIC  ]
    [
        {IMMUTABLE  | STABLE  | VOLATILE }
        | {SHIPPABLE | NOT SHIPPABLE}
        | {PACKAGE}
        | [ NOT  ] LEAKPROOF
        | {CALLED ON NULL INPUT  | RETURNS NULL ON NULL INPUT  | STRICT }
        | {[ EXTERNAL  ] SECURITY INVOKER  | [ EXTERNAL  ] SECURITY DEFINER | |
AUTHID DEFINER  | AUTHID CURR
ENT_USER}
        | COST execution_cost
        | ROWS result_rows
        | SET configuration_parameter { {TO | =} value | FROM CURRENT }
     ][...]
     {
        IS | AS
     } plsql_body
/
```

参数说明

- function_name

  要创建的函数名称（可以用模式修饰）。

  取值范围：字符串，要符合标识符的命名规范。且最多为 63 个字符。若超过 63 个字符，数据库会截断并保留前 63 个字符当做函数名称。

- argname

  函数参数的名称。

  取值范围：字符串，要符合标识符的命名规范。且最多为 63 个字符。若超过 63 个字符，数据库会截断并保留前 63 个字符当做函数参数名称。

- argmode

  函数参数的模式。

  取值范围：IN、OUT、INOUT 或 VARIADIC。缺省值是 IN。并且 OUT 和 INOUT 模式的参数不能用在 RETURNS TABLE 的函数定义中。

  说明：VARIADIC 用于声明数组类型的参数。

- argtype

函数参数的类型。可以使用%TYPE 或%ROWTYPE 间接引用变量或表的类型，详细可参考存储过程章节定义变量。

- expression

  参数的默认表达式。

- rettype

  函数返回值的数据类型。

如果存在 OUT 或 INOUT 参数，可以省略 RETURNS 子句。如果存在，该子句必须和输出参数所表示的结果类型一致：如果有多个输出参数，则为 RECORD，否则与单个输出参数的类型相同。

SETOF 修饰词表示该函数将返回一个集合，而不是单独一项。

与 argtype 相同，同样可以使用%TYPE 或%ROWTYPE 间接引用类型。

- column_name

  字段名称。

- column_type

  字段类型。

- definition

  一个定义函数的字符串常量，含义取决于语言。它可以是一个内部函数名称、一个指向某个目标文件的路径、一个 SQL 查询、一个过程语言文本。

- DETERMINISTIC

  SQL 语法兼容接口，未实现功能，不推荐使用。

- LANGUAGE lang_name

  用以实现函数的语言的名称。可以是 SQL、internal 或者是用户定义的过程语言名称。为了保证向下兼容，该名称可以用单引号（包围）。若采用单引号，则引号内必须为大写。

- WINDOW

  表示该函数是窗口函数。替换函数定义时不能改变 WINDOW 属性。

  须知：自定义窗口函数只支持 LANGUAGE 是 internal，并且引用的内部函数必须是窗口函数。

- IMMUTABLE

  表示该函数在给出同样的参数值时总是返回同样的结果。

- STABLE

  表示该函数不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同 SQL 语句之间变化。

- VOLATILE

  表示该函数值可以在一次表扫描内改变，因此不会做任何优化。

- SHIPPABLE|NOT SHIPPABLE

  表示该函数是否可以下推执行。预留接口，不推荐使用。

- FENCED|NOT FENCED

  声明用户定义的 C 函数是在保护模式还是非保护模式下执行。预留接口，不推荐使用。

- PACKAGE

  表示该函数是否支持重载。PostgreSQL 风格的函数本身就支持重载，此参数主要是针对其它风格的函数。

  不允许 package 函数和非 package 函数重载或者替换。

  package 函数不支持 VARIADIC 类型的参数。

  不允许修改函数的 package 属性。

- LEAKPROOF

  指出该函数的参数只包括返回值。LEAKPROOF 只能由系统管理员设置。

- CALLED ON NULL INPUT

  表明该函数的某些参数是 NULL 的时候可以按照正常的方式调用。该参数可以省略。

- RETURNS NULL ON NULL INPUT | STRICT

  STRICT 用于指定如果函数的某个参数是 NULL，此函数总是返回 NULL。如果声明了这个参数，当有 NULL 值参数时该函数不会被执行；而只是自动返回一个 NULL 结果。

  RETURNS NULL ON NULL INPUT 和 STRICT 的功能相同。

- EXTERNAL

  目的是和 SQL 兼容，是可选的，这个特性适合于所有函数，而不仅是外部函数。

- SECURITY INVOKER | AUTHID CURRENT_USER

  表明该函数将带着调用它的用户的权限执行。该参数可以省略。

  SECURITY INVOKER 和 AUTHID CURRENT_USER 的功能相同。

- SECURITY DEFINER | AUTHID DEFINER

  声明该函数将以创建它的用户的权限执行。

  AUTHID DEFINER 和 SECURITY DEFINER 的功能相同。

- COST execution_cost

  用来估计函数的执行成本。

  execution_cost 以 cpu_operator_cost 为单位。

  取值范围：正数

- ROWS result_rows

  估计函数返回的行数。用于函数返回的是一个集合。

  取值范围：正数，默认值是 1000 行。

- configuration_parameter | value

  把指定的数据库会话参数值设置为给定的值。如果 value 是 DEFAULT 或者 RESET，则在新的会话中使用系统的缺省设置。OFF 关闭设置。

  取值范围：字符串

  - DEFAULT
  - OFF
  - RESET

  指定默认值。

- from current

  取当前会话中的值设置为 configuration_parameter 的值。

- plsql_body

  PL/SQL 存储过程体。

  须知

- 当在函数体中创建用户时，日志中会记录密码的明文。因此不建议用户在函数体中创建用户。

示例

```
--定义函数为 SQL 查询。
postgres=# CREATE FUNCTION func_add_sql(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;

--利用参数名用 PL/pgSQL 自增一个整数。
postgres=# CREATE OR REPLACE FUNCTION func_increment_plsql(i integer) RETURNS
integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;

--返回 RECORD 类型
postgres=# CREATE OR REPLACE FUNCTION func_increment_sql(i int, out result_1
bigint, out result_2 bigint)
returns SETOF RECORD
as $$
begin
    result_1 = i + 1;
    result_2 = i * 10;
return next;
end;
$$language plpgsql;

--返回一个包含多个输出参数的记录。
postgres=# CREATE FUNCTION func_dup_sql(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

postgres=# SELECT * FROM func_dup_sql(42);

--计算两个整数的和，并返回结果。如果输入为 null，则返回 null。
postgres=# CREATE FUNCTION func_add_sql2(num1 integer, num2 integer) RETURN
integer
```

```
AS
BEGIN
RETURN num1 + num2;
END;
/
--修改函数 func_add_sql2 的执行规则为 IMMUTABLE，即参数不变时返回相同结果。
postgres=# ALTER FUNCTION func_add_sql2(INTEGER, INTEGER) IMMUTABLE;

--将函数 func_add_sql2 的名称修改为 add_two_number。
postgres=# ALTER FUNCTION func_add_sql2(INTEGER, INTEGER) RENAME TO
add_two_number;

--将函数 add_two_number 的属者改为 gbase。
postgres=# ALTER FUNCTION add_two_number(INTEGER, INTEGER) OWNER TO gbase;

--删除函数。
postgres=# DROP FUNCTION add_two_number;
postgres=# DROP FUNCTION func_increment_sql;
postgres=# DROP FUNCTION func_dup_sql;
postgres=# DROP FUNCTION func_increment_plsql;
postgres=# DROP FUNCTION func_add_sql;
```

相关命令

ALTER FUNCTION，DROP FUNCTION

3.8.70 CREATE GROUP

功能描述

创建一个新用户组。

注意事项

CREATE GROUP 是 CREATE ROLE 的别名，非 SQL 标准语法，不推荐使用，建议用户直接使用 CREATE ROLE 替代。

语法格式

```
CREATE GROUP group_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ]
{ PASSWORD | IDENTIFIED BY } { 'password' [ EXPIRED ] | DISABLE };
```

其中可选项 option 子句语法为：

```
{SYSADMIN | NOSYSADMIN}
    | {MONADMIN | NOMONADMIN}
```

```
    | {OPRADMIN | NOOPRADMIN}
    | {POLADMIN | NOPOLADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | {PERSISTENCE | NOPERSISTENCE}
    | CONNECTION LIMIT connlimit
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
    | NODE GROUP logic_group_name
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
    | DEFAULT TABLESPACE tablespace_name
    | PROFILE DEFAULT
    | PROFILE profile_name
    | PGUSER
```

参数说明

请参考 CREATE ROLE 的参数说明。

相关命令

ALTER GROUP，DROP GROUP，CREATE ROLE

3.8.71 CREATE INCREMENTAL MATERIALIZED VIEW

功能描述

CREATE INCREMENTAL MATERIALIZED VIEW 会创建一个增量物化视图，并且后续可以使用 REFRESH MATERIALIZED VIEW（全量刷新）和 REFRESH INCREMENTAL MATERIALIZED VIEW（增量刷新）刷新物化视图的数据。

CREATE INCREMENTAL MATERIALIZED VIEW 类似于 CREATE TABLE AS，不过它会记住被用来初始化该视图的查询，因此它可以在后续中进行数据刷新。一个物化视图有很多和表相同的属性，但是不支持临时物化视图。

注意事项

- 增量物化视图不可以在临时表或全局临时表上创建。
- 增量物化视图仅支持简单过滤查询和基表 UNION ALL 查询。
- 创建增量物化视图不可指定分布列。
- 创建增量物化视图后，基表中的绝大多数 DDL 操作不再支持。
- 不支持对增量物化视图进行 IUD 操作。
- 增量物化视图创建后，当基表数据发生变化时，需要使用刷新（REFRESH）命令保持物化视图与基表同步。

语法格式

```
CREATE INCREMENTAL MATERIALIZED VIEW mv_name
    [ (column_name [, ...] ) ]
    [ TABLESPACE tablespace_name ]
    AS query;
```

参数说明

- mv_name

  要创建的物化视图的名称（可以被模式限定）。

  取值范围：字符串，要符合标识符的命名规范。

- column_name

  新物化视图中的一个列名。物化视图支持指定列，指定列需要和后面的查询语句结果的列数量保持一致；如果没有提供列名，会从查询的输出列名中获取列名。

  取值范围：字符串，要符合标识符的命名规范。

- TABLESPACE tablespace_name

  指定新建物化视图所属表空间。如果没有声明，将使用默认表空间。

- AS query

  一个 SELECT 或者 TABLE 命令。这个查询将在一个安全受限的操作中运行。

示例

```
--创建一个普通表
CREATE TABLE my_table (c1 int, c2 int);
--创建增量物化视图
CREATE INCREMENTAL MATERIALIZED VIEW my_imv AS SELECT * FROM my_table;
--基表写入数据
INSERT INTO my_table VALUES(1,1),(2,2);
--对增量物化视图 my_imv 进行增量刷新
REFRESH INCREMENTAL MATERIALIZED VIEW my_imv;
```

相关链接

ALTER MATERIALIZED VIEW， CREATE MATERIALIZED VIEW，CREATE TABLE，DROP MATERIALIZED VIEW， REFRESH INCREMENTAL MATERIALIZED VIEW ，REFRESH MATERIALIZED VIEW

3.8.72 CREATE INDEX

功能描述

在指定的表上创建索引。

索引可以用来提高数据库查询性能，但是不恰当的使用将导致数据库性能下降。建议仅在匹配如下某条原则时创建索引：

经常执行查询的字段。

在连接条件上创建索引，对于存在多字段连接的查询，建议在这些字段上建立组合索引。例如， select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b， 可以在 t1 表上的 a、b 字段上建立组合索引。

where 子句的过滤条件字段上（尤其是范围条件）。

在经常出现在 order by、group by 和 distinct 后的字段。

在分区表上创建索引与在普通表上创建索引的语法不太一样，使用时请注意，如分区表上不支持并行创建索引，不支持创建部分索引。

注意事项

索引自身也占用存储空间、消耗计算资源，创建过多的索引将对数据库性能造成负面影响（尤其影响数据导入的性能，建议在数据导入后再建索引）。因此，仅在必要时创建索引。

索引定义里的所有函数和操作符都必须是 immutable 类型的，即它们的结果必须只能依赖于它们的输入参数，而不受任何外部的影响（如另外一个表的内容或者当前时间）。这个限制可以确保该索引的行为是定义良好的。要在一个索引上或 WHERE 中使用用户定义函数，请把它标记为 immutable 类型函数。

分区表索引分为 LOCAL 索引与 GLOBAL 索引，LOCAL 索引与某个具体分区绑定，而 GLOBAL 索引则对应整个分区表。

列存表支持的 PSORT 和 B-tree 索引都不支持创建表达式索引、部分索引，PSORT 不支持创建唯一索引，B-tree 支持创建唯一索引。

列存表支持的 GIN 索引支持创建表达式索引，但表达式不能包含空分词、空列和多列，不支持创建部分索引和唯一索引。

HASH 索引目前仅限于行存表索引、临时表索引和分区表 LOCAL 索引，且不支持创建多字段索引。

被授予 CREATE ANY INDEX 权限的用户，可以在 public 模式和用户模式下创建索引。

语法格式

在表上创建索引。

```
CREATE [ UNIQUE ] [IF NOT EXISTS] INDEX [ CONCURRENTLY ] [ [schema_name.]
index_name ] ON table_name [ USING method ]
    ({ { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC
| DESC ] [ NULLS { FIRST | LAST } ] }[, ...] )
    [ INCLUDE ( { column_name | ( expression ) }[, ...] ) ]
    [ WITH ( {storage_parameter = value} [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ];
```

在分区表上创建索引。

```
CREATE [ UNIQUE ] [IF NOT EXISTS] INDEX [ [schema_name.] index_name ] ON table_name
[ USING method ]
    ( { { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC
| DESC ] [ NULLS LAST ] } [, ...] )
```

```
    [ LOCAL [ ( { PARTITION index_partition_name | SUBPARTITION
index_subpartition_name [ TABLESPACE index_partition_tablespace ] } [, ...] ) ]
| GLOBAL ]
    [ INCLUDE ( { column_name | ( expression ) }[, ...] ) ]
    [ WITH ( { storage_parameter = value } [, ...] ) ]
    [ TABLESPACE tablespace_name ];
```

参数说明

- UNIQUE

创建唯一性索引，每次添加数据时检测表中是否有重复值。如果插入或更新的值会引起重复的记录时，将导致一个错误。

目前只有 B-tree 索引支持唯一索引。

```
CREATE SCHEMA [IF NOT EXISTS] schema_name
[ AUTHORIZATION user_name ] [WITH BLOCKCHAIN] [ schema_element [ ... ] ];
```

- IF NOT EXISTS

若指定该参数，则当一个同名的索引已经存在时，仅发出提示。

- CONCURRENTLY

以不阻塞 DML 的方式创建索引（加 ShareUpdateExclusiveLock 锁）。创建索引时，一般会阻塞其他语句对该索引所依赖表的访问。指定此关键字，可以实现创建过程中不阻塞 DML。

此选项只能指定一个索引的名称。

普通 CREATE INDEX 命令可以在事务内执行，但是 CREATE INDEX CONCURRENTLY 不可以在事务内执行。

列存表、分区表和临时表不支持 CONCURRENTLY 方式创建索引。

说明

- 创建索引时指定此关键字，需要执行先后两次对该表的全表扫描来完成 build，第一次扫描的时候创建索引，不阻塞读写操作；第二次扫描的时候合并更新第一次扫描到目前为止发生的变更。
- 由于需要执行两次对表的扫描和 build，而且必须等待现有的所有可能对该表执行修改的事务结束。这意味着该索引的创建比正常耗时更长，同时因此带来的 CPU 和 I/O 消耗对其他业务也会造成影响。

- 如果在索引构建时发生失败，那会留下一个“不可用”的索引。这个索引会被查询忽略，但它仍消耗更新开销。这种情况推荐的恢复方法是删除该索引并尝试再次 CONCURRENTLY 建索引。
- 由于在第二次扫描之后，索引构建必须等待任何持有早于第二次扫描拿的快照的事务终止，而且建索引时加的 ShareUpdateExclusiveLock 锁（4 级）会和大于等于 4 级的锁冲突，在创建这类索引时，容易引发卡住（hang）或者死锁问题。例如：
- 两个会话对同一个表创建 CONCURRENTLY 索引，会引起死锁问题；
- 两个会话，一个对表创建 CONCURRENTLY 索引，一个 drop table，会引起死锁问题；
- 三个会话，会话 1 先对表 a 加锁，不提交，会话 2 接着对表 b 创建 CONCURRENTLY 索引，会话 3 接着对表 a 执行写入操作，在会话 1 事务未提交之前，会话 2 会一直被阻塞；
- 将事务隔离级别设置成可重复读（默认为读已提交），起两个会话，会话 1 起事务对表 a 执行写入操作，不提交，会话 2 对表 b 创建 CONCURRENTLY 索引，在会话 1 事务未提交之前，会话 2 会一直被阻塞。

- schema_name

  模式的名称。

  取值范围：已存在模式名。

- index_name

  要创建的索引名，索引的模式与表相同。

  取值范围：字符串，要符合标识符的命名规范。

- table_name

  需要为其创建索引的表的名称，可以用模式修饰。

  取值范围：已存在的表名。

- USING method

  指定创建索引的方法。

  取值范围：

  btree：B-tree 索引使用一种类似于 B+树的结构来存储数据的键值，通过这种结构能够

快速的查找索引。btree 适合支持比较查询以及查询范围。

hash：Hash 索引使用 Hash 函数对索引的关键字进行散列。只能处理简单等值比较，比较适合在索引值较长的情况下使用。

gin：GIN 索引是倒排索引，可以处理包含多个键的值（比如数组）。

gist：Gist 索引适用于几何和地理等多维数据类型和集合数据类型。目前支持的数据类型有 box、point、poly、circle、tsvector、tsquery、range。

Psort：Psort 索引。针对列存表进行局部排序索引。

ubtree：仅供 ustore 表使用的多版本 B-tree 索引，索引页面上包含事务信息，能并自主回收页面。

行存表（ASTORE 存储引擎）支持的索引类型：btree（行存表缺省值）、hash、gin、gist。行存表（USTORE 存储引擎）支持的索引类型：ubtree。列存表支持的索引类型：Psort（列存表缺省值）、btree、gin。全局临时表不支持 GIN 索引和 Gist 索引。

说明：列存表对 GIN 索引支持仅限于对于 tsvector 类型的支持，即创建列存 GIN 索引入参需要为 to_tsvector 函数（的返回值）。此方法为 GIN 索引比较普遍的使用方式。

- column_name

表中需要创建索引的列的名称（字段名）。

如果索引方式支持多字段索引，可以声明多个字段。全局索引最多可以声明 31 个字段，其他索引最多可以声明 32 个字段。

- expression

创建一个基于该表的一个或多个字段的表达式索引，通常必须写在圆括弧中。如果表达式有函数调用的形式，圆括弧可以省略。

表达式索引可用于获取对基本数据的某种变形的快速访问。比如，一个在 upper(col)上的函数索引将允许 WHERE upper(col) = 'JIM'子句使用索引。

在创建表达式索引时，如果表达式中包含 IS NULL 子句，则这种索引是无效的。此时，建议用户尝试创建一个部分索引。

- COLLATE collation

COLLATE 子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。排序规则可以使用“select * from pg_collation”命令从 pg_collation 系统表中查询，默认的排序规则为查询结果中以 default 开始的行。

- opclass

操作符类的名称。对于索引的每一列可以指定一个操作符类，操作符类标识了索引那一列的使用的操作符。例如一个 B-tree 索引在一个四字节整数上可以使用 int4_ops；这个操作符类包括四字节整数的比较函数。实际上对于列上的数据类型默认的操作符类是足够用的。操作符类主要用于一些有多种排序的数据。例如，用户想按照绝对值或者实数部分排序一个复数。能通过定义两个操作符类然后当建立索引时选择合适的类。

- ASC

指定按升序排序（默认）。

- DESC

指定按降序排序。

- NULLS FIRST

指定空值在排序中排在非空值之前，当指定 DESC 排序时，本选项为默认的。

- NULLS LAST

指定空值在排序中排在非空值之后，未指定 DESC 排序时，本选项为默认的。

- LOCAL

指定创建的分区索引为 LOCAL 索引。

- GLOBAL

指定创建的分区索引为 GLOBAL 索引，当不指定 LOCAL、GLOBAL 关键字时，默认创建 GLOBAL 索引。

- INCLUDE (column_name [, …])

可选的 INCLUDE 子句指定将一些非键列（non-key columns）包含在索引中。非键列不能用于作为索引扫描的加速搜索条件，同时在检查索引的唯一性约束时会忽略它们。

仅索引扫描 (Index Only Scan) 可以直接返回非键列中的内容，而不必去访问索引所对应的堆表。

将非键列添加为 INCLUDE 列需要保守一些，尤其是对于宽列。如果索引元组超过索引类型允许的最大大小，数据将插入失败。需要注意的是，任何情况下为索引添加非键列都会增加索引的空间占用，从而可能减慢搜索速度。

目前只有 ubtree 索引访问方式支持该特性。非键列会被保存在与堆元组对应的索引叶子

元组中，不会包含在索引上层页面的元组中。

- WITH ({storage_parameter = value} [, …])

  指定索引方法的存储参数。

  取值范围：

  只有 GIN 索引支持 FASTUPDATE、GIN_PENDING_LIST_LIMIT 参数。GIN 和 Psort 之外的索引都支持 FILLFACTOR 参数。只有 UBTREE 索引支持 INDEXSPLIT 参数。

- FILLFACTOR

  一个索引的填充因子（fillfactor）是一个介于 10 和 100 之间的百分数。

  取值范围：10~100

- FASTUPDATE

  GIN 索引是否使用快速更新。

  取值范围：ON，OFF

  默认值：ON

- GIN_PENDING_LIST_LIMIT

  当 GIN 索引启用 fastupdate 时，设置该索引 pending list 容量的最大值。

  取值范围：64~INT_MAX，单位 KB。

  默认值：gin_pending_list_limit 的默认取决于 GUC 中 gin_pending_list_limit 的值（默认为 4MB）

- INDEXSPLIT

  UBTREE 索引选择采取哪种分裂策略。其中 DEFAULT 策略指的是与 BTREE 相同的分裂策略。INSERTPT 策略能在某些场景下显著降低索引空间占用。

  取值范围：INSERTPT，DEAFAULT

  默认值：INSERTPT

- TABLESPACE tablespace_name

  指定索引的表空间，如果没有声明则使用默认的表空间。

  取值范围：已存在的表空间名。

- WHERE predicate

创建一个部分索引。部分索引是一个只包含表的一部分记录的索引，通常是该表中比其他部分数据更有用的部分。例如，有一个表，表里包含已记账和未记账的定单，未记账的定单只占表的一小部分而且这部分是最常用的部分，此时就可以通过只在未记账部分创建一个索引来改善性能。另外一个可能的用途是使用带有 UNIQUE 的 WHERE 强制一个表的某个子集的唯一性。

取值范围：predicate 表达式只能引用表的字段，它可以使用所有字段，而不仅是被索引的字段。目前，子查询和聚集表达式不能出现在 WHERE 子句里。不建议使用 int 等数值类型作为 predicate，因为 int 等数值类型可以隐式转换为 bool 值（非 0 值隐式转换为 true，0 转换为 false），可能导致非预期的结果。

- PARTITION index_partition_name

  索引分区的名称。

  取值范围：字符串，要符合标识符的命名规范。

- SUBPARTITION index_subpartition_name

  索引二级分区的名称。

  取值范围：字符串，要符合标识符的命名规范

- TABLESPACE index_partition_tablespace

  索引分区的表空间。

  取值范围：如果没有声明，将使用分区表索引的表空间 index_tablespace。

- COMPRESSTYPE

索引参数，设置索引压缩算法。1 代表 pglz 算法，2 代表 zstd 算法，默认不压缩。（仅支持 B-TREE 索引）

取值范围：0~2，默认值为 0。

- COMPRESS_LEVEL

索引参数，设置索引压缩算法等级，仅当 COMPRESSTYPE 为 2 时生效。压缩等级越高，索引的压缩效果越好，索引的访问速度越慢。（仅支持 B-TREE 索引）

取值范围：-31~31，默认值为 0。

- COMPRESS_CHUNK_SIZE

索引参数，设置索引压缩 chunk 块大小。chunk 数据块越小，预期能达到的压缩效果越

好，同时数据越离散，影响索引的访问速度。（仅支持 B-TREE 索引）

取值范围：与页面大小有关。在页面大小为 8k 场景，取值范围为：512、1024、2048、4096。

默认值：4096

- COMPRESS_PREALLOC_CHUNKS

索引参数，设置索引压缩 chunk 块预分配数量。预分配数量越大，索引的压缩率相对越差，离散度越小，访问性能越好。（仅支持 B-TREE 索引）

取值范围：0~7，默认值为 0。

当 COMPRESS_CHUNK_SIZE 为 512 和 1024 时，支持预分配设置最大为 7。

当 COMPRESS_CHUNK_SIZE 为 2048 时，支持预分配设置最大为 3。

当 COMPRESS_CHUNK_SIZE 为 4096 时，支持预分配设置最大为 1。

- COMPRESS_BYTE_CONVERT

索引参数，设置索引压缩字节转换预处理。在一些场景下可以提升压缩效果，同时会导致一定性能劣化。

取值范围：布尔值，默认关闭。

- COMPRESS_DIFF_CONVERT

索引参数，设置索引压缩字节差分预处理。只能与 compress_byte_convert 一起使用。在一些场景下可以提升压缩效果，同时会导致一定性能劣化。

取值范围：布尔值，默认关闭。

示例

```
--创建表 tpcds.ship_mode_t1。
postgres=# create schema tpcds;
postgres=# CREATE TABLE tpcds.ship_mode_t1
(
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
)
```

```
;

--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建普通的唯一索引。
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK);

--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建指定 B-tree 索引。
postgres=# CREATE INDEX ds_ship_mode_t1_index4 ON tpcds.ship_mode_t1 USING
btree(SM_SHIP_MODE_SK);

--在表 tpcds.ship_mode_t1 上 SM_CODE 字段上创建表达式索引。
postgres=# CREATE INDEX ds_ship_mode_t1_index2 ON
tpcds.ship_mode_t1(SUBSTR(SM_CODE,1 ,4));

--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建 SM_SHIP_MODE_SK 大于 10
的部分索引。
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index3 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK) WHERE SM_SHIP_MODE_SK>10;

--重命名一个现有的索引。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index1 RENAME TO
ds_ship_mode_t1_index5;

--设置索引不可用。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index2 UNUSABLE;

--重建索引。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index2 REBUILD;

--删除一个现有的索引。
postgres=# DROP INDEX tpcds.ds_ship_mode_t1_index2;

--删除表。
postgres=# DROP TABLE tpcds.ship_mode_t1;

--创建表空间。
postgres=# CREATE TABLESPACE example1 RELATIVE LOCATION
'tablespace1/tablespace_1';
postgres=# CREATE TABLESPACE example2 RELATIVE LOCATION
'tablespace2/tablespace_2';
postgres=# CREATE TABLESPACE example3 RELATIVE LOCATION
'tablespace3/tablespace_3';
```

```
postgres=# CREATE TABLESPACE example4 RELATIVE LOCATION
'tablespace4/tablespace_4';
--创建表 tpcds.customer_address_p1。
postgres=# CREATE TABLE tpcds.customer_address_p1
(
    CA_ADDRESS_SK             INTEGER               NOT NULL,
    CA_ADDRESS_ID             CHAR(16)              NOT NULL,
    CA_STREET_NUMBER          CHAR(10)                      ,
    CA_STREET_NAME            VARCHAR(60)                   ,
    CA_STREET_TYPE            CHAR(15)                      ,
    CA_SUITE_NUMBER           CHAR(10)                      ,
    CA_CITY                   VARCHAR(60)                   ,
    CA_COUNTY                 VARCHAR(30)                   ,
    CA_STATE                  CHAR(2)                       ,
    CA_ZIP                    CHAR(10)                      ,
    CA_COUNTRY                VARCHAR(20)                   ,
    CA_GMT_OFFSET             DECIMAL(5,2)                  ,
    CA_LOCATION_TYPE          CHAR(20)
)
TABLESPACE example1
PARTITION BY RANGE(CA_ADDRESS_SK)
(
   PARTITION p1 VALUES LESS THAN (3000),
   PARTITION p2 VALUES LESS THAN (5000) TABLESPACE example1,
   PARTITION p3 VALUES LESS THAN (MAXVALUE) TABLESPACE example2
)
ENABLE ROW MOVEMENT;
--创建分区表索引 ds_customer_address_p1_index1，不指定索引分区的名称。
postgres=# CREATE INDEX ds_customer_address_p1_index1 ON
tpcds.customer_address_p1(CA_ADDRESS_SK) LOCAL;
--创建分区表索引 ds_customer_address_p1_index2，并指定索引分区的名称。
postgres=# CREATE INDEX ds_customer_address_p1_index2 ON
tpcds.customer_address_p1(CA_ADDRESS_SK) LOCAL
(
    PARTITION CA_ADDRESS_SK_index1,
    PARTITION CA_ADDRESS_SK_index2 TABLESPACE example3,
    PARTITION CA_ADDRESS_SK_index3 TABLESPACE example4
)
TABLESPACE example2;

--创建 GLOBAL 分区索引
```

```
postgres=# CREATE INDEX ds_customer_address_p1_index3 ON
tpcds.customer_address_p1(CA_ADDRESS_ID) GLOBAL;

--不指定关键字，默认创建 GLOBAL 分区索引
postgres=# CREATE INDEX ds_customer_address_p1_index4 ON
tpcds.customer_address_p1(CA_ADDRESS_ID);

--修改分区表索引 CA_ADDRESS_SK_index2 的表空间为 example1。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 MOVE PARTITION
CA_ADDRESS_SK_index2 TABLESPACE example1;

--修改分区表索引 CA_ADDRESS_SK_index3 的表空间为 example2。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 MOVE PARTITION
CA_ADDRESS_SK_index3 TABLESPACE example2;

--重命名分区表索引。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 RENAME PARTITION
CA_ADDRESS_SK_index1 TO CA_ADDRESS_SK_index4;

--删除索引和分区表。
postgres=# DROP INDEX tpcds.ds_customer_address_p1_index1;
postgres=# DROP INDEX tpcds.ds_customer_address_p1_index2;
postgres=# DROP TABLE tpcds.customer_address_p1;
--删除表空间。
postgres=# DROP TABLESPACE example1;
postgres=# DROP TABLESPACE example2;
postgres=# DROP TABLESPACE example3;
postgres=# DROP TABLESPACE example4;

--创建列存表以及列存表 GIN 索引。
postgres=# create table cgin_create_test(a int, b text) with (orientation =
column);
CREATE TABLE
postgres=# create index cgin_test on cgin_create_test using
gin(to_tsvector('ngram', b));
CREATE INDEX
```

相关命令

ALTER INDEX，DROP INDEX

优化建议

create index

建议仅在匹配如下条件之一时创建索引：

经常执行查询的字段。

在连接条件上创建索引，对于存在多字段连接的查询，建议在这些字段上建立组合索引。例如，select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b，可以在 t1 表上的 a、b 字段上建立组合索引。

where 子句的过滤条件字段上（尤其是范围条件）。

在经常出现在 order by、group by 和 distinct 后的字段。

约束限制：

分区表上不支持创建部分索引。

分区表创建 GLOBAL 索引时，存在以下约束条件：

不支持表达式索引、部分索引

不支持列存表

仅支持 B-tree 索引

在相同属性列上，分区 LOCAL 索引与 GLOBAL 索引不能共存。

GLOBAL 索引，最大支持 31 列。

如果 alter 语句不带有 UPDATE GLOBAL INDEX，那么原有的 GLOBAL 索引将失效，查询时将使用其他索引进行查询；如果 alter 语句带有 UPDATE GLOBAL INDEX，原有的 GLOBAL 索引仍然有效，并且索引功能正确。

3.8.73 CREATE LANGUAGE

功能描述

定义一种新的过程语言。单机和集中式暂不支持创建过程语言。

语法格式

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name;
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ];
```

参数说明

- TRUSTED

TRUSTED 说明该语言并不授权没有权限的用户访问数据。如果在注册该语言时忽略这个关键字，则只有超级用户权限可以使用。

- PROCEDURAL

  这是个没有用的字。

- name

  新过程语言的名称。这个名字应该在数据库的所有语言中唯一。

  出于向下兼容的原因，这个名字可以用单引号包围。

- HANDLER call_handler

  call_handler 是一个以前注册过的函数名字，该函数将被用来执行该过程语言的函数。过程语言的调用处理器必须用一种编译语言(比如 C)书写，调用风格必须是版本 1 的调用风格，并且注册为不接受参数并且返回 language_handler 类型的函数。language_handler 是用于将函数声明为调用处理器的占位符。

- INLINE inline_handler

  inline_handler 是以前注册过的函数名字，用来在该语言中执行一个匿名代码块(DO 命令)。如果没有指定 inline_handler 函数，那么该语言不支持匿名代码块。处理器函数必须接受一个 internal 类型的参数，这将是 DO 命令的内部表示，并且它通常返回 void。忽略该处理器的返回值。

- VALIDATOR valfunction

  valfunction 是一个以前注册过的函数名字，在用该语言创建新函数的时候将用它来校验新函数。如果没有声明校验函数，那么建立新函数的时候就不会检查它。校验函数必须接受一个类型为 oid 的参数，它是将要创建的函数的 OID，并且通常会返回 void。

  校验函数通常会检查函数体，看看有没有语法错误，但是它也可以查看函数的其它属性，比如该语言是否不能处理某种参数类型。校验函数应该用 ereport()函数报告错误。该函数的返回值将被忽略。

示例

```
创建标准的过程语言的比较好的方法：
CREATE LANGUAGE plperl;
对于 pg_pltemplate 还不知道的语言，需要下面这样的序列：
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
```

```
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

3.8.74 CREATE MASKING POLICY

功能描述

创建脱敏策略。

注意事项

只有 poladmin、sysadmin 或初始用户能执行此操作。

需要开启安全策略开关，即设置 GUC 参数 enable_security_policy=on，脱敏策略才可以生效。

语法格式

```
CREATE MASKING POLICY policy_name masking_clause [, ... ]
[ policy_filter_clause ] [ ENABLE | DISABLE ];
masking_clause:
masking_function ON LABEL(label_name [, ... ])
masking_function:
```

maskall 不是预置函数，硬编码在代码中，不支持\df 展示。

预置时脱敏方式如下：

```
maskall | randgbaseasking | creditcardmasking | basicemailmasking |
fullemailmasking | shufflemasking | alldigitsmasking | regexpmasking
policy_filter_clause:
FILTER ON FILTER_TYPE(filter_value [,...]*)[,...]*
FILTER ON { ( FILTER_TYPE ( filter_value [, ... ] ) ) [, ... ] }
FILTER_TYPE:
APP | ROLES | IP
```

参数说明

- policy_name

  审计策略名称，需要唯一，不可重复。

  取值范围：字符串，要符合标识符的命名规范。

- label_name

  资源标签名称。

- masking_clause

  指出使用何种脱敏函数对被 label_name 标签标记的数据库资源进行脱敏，支持用 schema.function 的方式指定脱敏函数。

- policy_filter

  指出该脱敏策略对何种身份的用户生效，若为空表示对所用用户生效。

- FILTER_TYPE

  描述策略过滤的条件类型，包括 IP | APP | ROLES。

- filter_value

  指具体过滤信息内容，例如具体的 IP、具体的 APP 名称、具体的用户名。

- ENABLE|DISABLE

  可以打开或关闭脱敏策略。若不指定 ENABLE|DISABLE，语句默认为 ENABLE。

示例

```
--创建 dev_mask 和 bob_mask 用户。
postgres=# CREATE USER dev_mask PASSWORD 'dev@1234';
postgres=# CREATE USER bob_mask PASSWORD 'bob@1234';

--创建一个表 tb_for_masking
postgres=# CREATE TABLE tb_for_masking(col1 text, col2 text, col3 text);

--创建资源标签标记敏感列 col1
postgres=# CREATE RESOURCE LABEL mask_lb1 ADD COLUMN(tb_for_masking.col1);

--创建资源标签标记敏感列 col2
postgres=# CREATE RESOURCE LABEL mask_lb2 ADD COLUMN(tb_for_masking.col2);

--对访问敏感列 col1 的操作创建脱敏策略
postgres=# CREATE MASKING POLICY maskpol1 maskall ON LABEL(mask_lb1);

--创建仅对用户 dev_mask 和 bob_mask,客户端工具为 psql 和 gsql，IP 地址为
'10.20.30.40', '127.0.0.0/24'场景下生效的脱敏策略。
postgres=# CREATE MASKING POLICY maskpol2 randgbaseasking ON LABEL(mask_lb2)
FILTER ON ROLES(dev_mask, bob_mask), APP(psql, gsql), IP('10.20.30.40',
'127.0.0.0/24');
```

相关命令

ALTER MASKING POLICY，DROP MASKING POLICY。

3.8.75 CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW 会创建一个全量物化视图，并且后续可以使用 REFRESH MATERIALIZED VIEW（全量刷新）刷新物化视图的数据。

CREATE MATERIALIZED VIEW 类似于 CREATE TABLE AS，不过它会记住被用来初始化该视图的查询， 因此它可以在后续中进行数据刷新。一个物化视图有很多和表相同的属性，但是不支持临时物化视图。

注意事项

全量物化视图不可以在临时表或全局临时表上创建。

全量物化视图不支持 nodegroup。

创建全量物化视图后，基表中的绝大多数 DDL 操作不再支持。

不支持对全量物化视图进行 IUD 操作。

全量物化视图创建后，当基表数据发生变化时，需要使用刷新（REFRESH）命令保持物化视图与基表同步。

Ustore 引擎不支持物化创建、使用视图。

语法格式

```
CREATE [ INCREMENTAL ] MATERIALIZED VIEW table_name
    [ (column_name [, ...] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
```

参数说明

- mv_name

  要创建的物化视图的名称（可以被模式限定）。

  取值范围：字符串，要符合标识符的命名规范。

- column_name

  新物化视图中的一个列名。物化视图支持指定列，指定列需要和后面的查询语句结果的列数量保持一致；如果没有提供列名，会从查询的输出列名中获取列名。

取值范围：字符串，要符合标识符的命名规范。

- WITH (storage_parameter [= value] [, …])

  这个子句为表或索引指定一个可选的存储参数。详见 CREATE TABLE。

- TABLESPACE tablespace_name

  指定新建物化视图所属表空间。如果没有声明，将使用默认表空间。

- AS query

  一个 SELECT、TABLE 或者 VALUES 命令。这个查询将在一个安全受限的操作中运行。

示例

```
--创建一个普通表
postgres=# CREATE TABLE my_table (c1 int, c2 int);
--创建全量物化视图
postgres=# CREATE MATERIALIZED VIEW my_mv AS SELECT * FROM my_table;
--基表写入数据
postgres=# INSERT INTO my_table VALUES(1,1),(2,2);
--对全量物化视图 my_mv 进行全量刷新
postgres=# REFRESH MATERIALIZED VIEW my_mv;
```

相关命令

ALTER MATERIALIZED VIEW， CREATE INCREMENTAL MATERIALIZED VIEW，CREATE TABLE ， DROP MATERIALIZED VIEW ， REFRESH INCREMENTAL MATERIALIZED VIEW，REFRESH MATERIALIZED VIEW

3.8.76CREATE MODEL

功能描述

训练机器学习模型并保存模型。

注意事项

模型名称具有唯一性约束，注意命名格式。

AI 训练时长波动较大，在部分情况下训练运行时间较长，设置的 GUC 参数 statement_timeout 时长过短会导致训练中断。建议 statement_timeout 设置为 0，不对语句执行时长进行限制。

语法格式

```
CREATE MODEL model_name USING algorithm_name
[FEATURES { {expression [ [ AS ] output_name ]} [, ...] }]
[TARGET { {expression [ [ AS ] output_name ]} [, ...] }]
FROM { table_name | select_query }
WITH hyperparameter_name = { hyperparameter_value | DEFAULT } [, ...] }
```

参数说明

- model_name

  对训练模型进行命名，模型名称具有唯一性约束。

  取值范围：字符串，需要符合标识符的命名规范。

- architecture_name

  训练模型的算法类型。

  取值范围：字符型，当前支持：logistic_regression、linear_regression、svm_classification、kmeans。

- attribute_list

  枚举训练模型的输入列名。

  取值范围：字符型，需要符合数据属性名的命名规范。

- attribute_name

  在监督学习任务重训练模型的目标列名(可进行简单的表达式处理)。

  取值范围：字符型，需要符合数据属性名的命名规范。

- subquery

  数据源。

  取值范围：字符串，符合数据库 SQL 语法。

- hyper_parameter_name

  机器学习模型的超参名称。

  取值范围：字符串，针对不同算法范围不同，详情请参考《GBase 8s V8.8.5_5.0.0_AI 特性指南》“DB4AI-Query：模型训练和推断”章节中的表 算子支持的超参。

- hp_value

  超参数值。

取值范围：字符串，针对不同算法范围不同，详情请参考《GBase 8s V8.8.5_5.0.0_AI 特性指南》“DB4AI-Query：模型训练和推断”章节中的表 超参的默认值以及取值范围。

示例

```
CREATE MODEL price_model USING logistic_regression
 FEATURES size, lot
 TARGET price
 FROM HOUSES
 (WITH learning_rate=0.88, max_iterations=default);
```

相关命令

DROP MODEL，PREDICT BY

3.8.77 CREATE OPERATOR

功能描述

定义一个新操作符。

注意事项

CREATE OPERATOR 定义一个新的 name 操作符。 定义该操作符的用户将成为其所有者。如果给出了一个模式名， 那么该操作符将在指定的模式中创建。否则它会在当前模式中创建。

操作符 name 可包含字符：- * / < > = ~ ! @ # % ^ & | ` ?

选择名字的时候有几个限制：

- –和/*不能在操作符名的任何地方出现， 因为它们会被认为是一个注释的开始。
- 一个多字符的操作符不能以+或-结尾，除非该名字还包含至少下面字符之一：~ ! @ # % ^ & | ` ?
- => 作为一个操作符名的使用已经废弃了。
- 操作符!=在输入时映射成<>， 因此这两个名称总是等价的。
- 至少需要定义一个 LEFTARG 和 RIGHTARG。对于双目操作符来说， 两者都需要定义。对右目操作符来说，只需要定义 LEFTARG， 而对于左目操作符来说，只需要定义 RIGHTARG。
- 同样，function_name 过程必须已经用 CREATE FUNCTION 定义过， 而且必须定义为接受正确数量的指定类型参数(一个或是两个)。

要想能够创建一个操作符，必须在参数类型和返回类型上有 USAGE 权限，还要在底层函数上有 EXECUTE 权限。如果指定了交换或者负操作符，必须拥有这些操作符。

语法格式

```
CREATE OPERATOR name (
    PROCEDURE = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

参数说明

- name

  要定义的操作符。可用的字符见上文。其名字可以用模式修饰， 比如 CREATE OPERATOR myschema.+ (…)。如果没有模式， 则在当前模式中创建操作符。同一个模式中的两个操作符可以有一样的名字， 只要他们操作不同的数据类型。这是一个重载过程。

- function_name

  用于实现该操作符的函数。

- left_type

  操作符左边的参数数据类型，如果存在的话。如果是左目操作符，这个参数可以省略。

- right_type

  操作符右边的参数数据类型，如果存在的话。如果是右目操作符，这个参数可以省略。

- com_op

  该操作符对应的交换操作符。

- neg_op

  该操作符对应的负操作符。

- res_proc

  此操作符约束选择性评估函数。

- join_proc

  此操作符连接选择性评估函数。

- HASHES

  表明此操作符支持 Hash 连接。

- MERGES

  表明此操作符可以支持一个融合连接。

使用 OPERATOR()语法在 com_op 或者其它可选参数里给出一个模式修饰的操作符名，比如：

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

示例

下面命令定义一个新操作符：面积相等，用于 box 数据类型。

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES, MERGES
);
```

3.8.78CREATE PACKAGE

功能描述

创建一个新的 PACKAGE。

注意事项

在 package specification 中声明过的函数或者存储过程，必须在 package body 中找到定义。

在实例化中，无法调用带有 commit/rollback 的存储过程。

不能在 Trigger 中调用 package 函数。

不能在外部 SQL 中直接使用 package 当中的变量。

不允许在 package 外部调用 package 的私有变量和存储过程。

不支持其它存储过程不支持的用法，例如，在 function 中不允许调用 commit/rollback，

则 package 的 function 中同样无法调用 commit/rollback。

不支持 schema 与 package 同名。

只支持 A 风格的存储过程和函数定义。

不支持 package 内有同名变量，包括包内同名参数。

package 的全局变量为 session 级，不同 session 之间 package 的变量不共享。

package 中调用自治事务的函数，不允许使用 package 中的 cursor 变量，以及递归的使用 package 中 cursor 变量的函数。

package 中不支持声明 refcursor 变量。

package 默认为 SECURITY INVOKER 权限，如果想将默认行为改为 SECURITY DEFINER 权限，需要设置 guc 参数 behavior_compat_options='plsql_security_definer'。

被授予 CREATE ANY PACKAGE 权限的用户，可以在 public 模式和用户模式下创建 PACKAGE。

如果需要创建带有特殊字符的 package 名，特殊字符中不能含有空格，并且最好设置 GUC 参数 behavior_compat_options="skip_insert_gs_source",否则可能引起报错。

语法格式

CREATE PACKAGE SPECIFICATION 语法格式。

```
CREATE [ OR REPLACE ] PACKAGE [ schema ] package_name
    [ invoker_rights_clause ] { IS | AS } item_list_1 END package_name;
```

invoker_rights_clause 可以被声明为 AUTHID DEFINER 或者 AUTHID INVOKER，分别为定义者权限和调用者权限。

item_list_1 可以为声明的变量或者存储过程以及函数。

PACKAGE SPECIFICATION(包规格)声明了包内的公有变量、函数、异常等，可以被外部函数或者存储过程调用。在 PACKAGE SPECIFICATION 中只能声明存储过程、函数，不能定义存储过程或者函数。

CREATE PACKAGE BODY 语法格式。

```
CREATE [ OR REPLACE ] PACKAGE BODY [ schema ] package_name
    { IS | AS } declare_section [ initialize_section ] END package_name;
```

PACKAGE BODY(包体内)定义了包的私有变量、函数等。如果变量或者函数没有在 PACKAGE SPECIFICATION 中声明过，那么这个变量或者函数则为私有变量或者函数。

PACKAGE BODY 也可以声明实例化部分，用来初始化 package。

示例

CREATE PACKAGE SPECIFICATION 示例

```
CREATE OR REPLACE PACKAGE emp_bonus IS
var1 int:=1;--公有变量
var2 int:=2;
PROCEDURE testpro1(var3 int);--公有存储过程，可以被外部调用
END emp_bonus;
/
```

CREATE PACKAGE BODY 示例

```
drop table if exists test1;
create or replace package body emp_bonus is
var3 int:=3;
var4 int:=4;
procedure testpro1(var3 int)
is
begin
create table if not exists test1(col1 int);
insert into test1 values(var1);
insert into test1 values(var5);
end;
begin  --实例化开始
var4:=9;
testpro1(var4);
end emp_bonus;
/
```

ALTER PACKAGE OWNER 示例

```
--将 PACKAGE emp_bonus 的所属者改为 gbase
ALTER PACKAGE emp_bonus OWNER TO gbase;
```

调用 PACKAGE 示例

```
--使用 call 调用 package 存储过程
call emp_bonus.testpro1(1);
--使用 select 调用 package 存储过程
select emp_bonus.testpro1(1);
--匿名块里调用 package 存储过程
begin
emp_bonus.testpro1(1);
```

```
end;
/
```

3.8.79 CREATE PROCEDURE

功能描述

创建一个新的存储过程。

注意事项

如果创建存储过程时参数或返回值带有精度，不进行精度检测。

创建存储过程时，存储过程定义中对表对象的操作建议都显示指定模式，否则可能会导致存储过程执行异常。

在创建存储过程时，存储过程内部通过 SET 语句设置 current_schema 和 search_path 无效。执行完函数 search_path 和 current_schema 与执行函数前的 search_path 和 current_schema 保持一致。

如果存储过程参数中带有出参，SELECT 调用存储过程必须缺省出参，CALL 调用存储过程调用非重载函数时必须指定出参，对于重载的 package 函数，out 参数可以缺省，具体信息参见 CALL 的示例。

存储过程指定 package 属性时支持重载。

在创建 procedure 时，不能在 avg 函数外面嵌套其他 agg 函数，或者其他系统函数。

在存储过程内部调用其它无参数的存储过程时，可以省略括号，直接使用存储过程名进行调用。

在存储过程内部调用其他有出参的函数，如果在赋值表达式中调用时，被调函数的出参可以省略，给出了也会被忽略。

存储过程支持参数注释的查看与导出、导入。

存储过程支持介于 IS/AS 与 plsql_body 之间的注释的查看与导出、导入。

存储过程默认为 SECURITY INVOKER 权限，如果想将默认行为改为 SECURITY DEFINER 权限，需要设置 guc 参数 behavior_compat_options='plsql_security_definer'。

被授予 CREATE ANY FUNCTION 权限的用户，可以在用户模式下创建/替换存储过程。

out/inout 参数必须传入变量，不能够传入常量。

集中式环境下，想要调用 in 参数相同，out 参数不同的存储过程，需要设置 guc 参数

behavior_compat_options='proc_outparam_override',并且打开参数后，无论使用 select 还是 call 调用存储过程，都必须加上 out 参数。打开参数后，不支持使用 perform 调用存储过程或函数。

语法格式

```
CREATE [ OR REPLACE ] PROCEDURE procedure_name
    [ ( {[ argmode ] [ argname ] argtype [ { DEFAULT | := | = } expression ]}[,...]) ]
   { IS | AS } plsql_body {NOT FENCED | FENCED}
/
```

参数说明

- OR REPLACE

  当存在同名的存储过程时，替换原来的定义。

- procedure_name

  创建的存储过程名称，可以带有模式名。

  取值范围：字符串，要符合标识符的命名规范。

- argmode

  参数的模式。

   须知：VARIADIC 用于声明数组类型的参数。

  取值范围：IN、OUT、INOUT 或 VARIADIC。缺省值是 IN。只有 OUT 模式的参数能跟在 VARIADIC 参数之后。

- argname

  参数的名称。

  取值范围：字符串，要符合标识符的命名规范。

- argtype

  参数的数据类型。可以使用%TYPE 或%ROWTYPE 间接引用变量或表的类型，详细可参考《GBase 8s V8.8.5_5.0.0_SQL 参考手册》中“存储过程”-“基本语句”-“定义变量”章节。

  取值范围：可用的数据类型。

- configuration_parameter value

把指定的配置参数设置为给定的值。如果 value 是 DEFAULT，则在新的会话中使用系统的缺省设置。OFF 关闭设置。

取值范围：字符串

DEFAULT

OFF

指定默认值。

- from current

取当前会话中的值设置为 configuration_parameter 的值。

- IMMUTABLE、STABLE 等

行为约束可选项。各参数的功能与 CREATE FUNCTION 类似，详细说明见 CREATE FUNCTION

- plsql_body

PL/SQL 存储过程体。

须知

➢ 当在存储过程体中进行创建用户等涉及用户密码相关操作时，系统表及 csv 日志中会记录密码的明文。因此不建议用户在存储过程体中进行涉及用户密码的相关操作。

说明

➢ argname 和 argmode 的顺序没有严格要求，推荐按照 argname、argmode、argtype 的顺序使用。

- NOT FENCED | FENCED

FENCED 或 NOT FENCED:这个子句指定例程是否被认为可以在数据库管理器操作环境的进程或地址空间中“安全地”运行。

相关命令

DROP PROCEDURE

优化建议

analyse | analyze

不支持在事务或匿名块中执行 analyze 。

不支持在函数或存储过程中执行 analyze 操作。

3.8.80 CREATE PUBLICATION

功能描述

向当前数据库添加一个新的发布，发布的名称必须与当前数据库中任何现有发布的名称不同。发布本质上是通过逻辑复制将一组表的数据变更进行复制。

注意事项

如果既没有指定 FOR TABLE，也没有指定 FOR ALL TABLES， 那么这个发布就是以一组空表开始的，可以在后续添加表。

创建发布不会开始复制。它只为未来的订阅者定义一个分组和过滤逻辑。 要创建一个发布，调用者必须拥有当前数据库的 CREATE 权限。（当然，系统管理员不需要这个检查。）

要将表添加到发布中，调用者必须拥有该表的所有权。FOR ALL TABLES 子句要求调用者是具有 SYSADMIN 权限用户。

添加到发布 UPDATE 或 DELETE 操作的发布的表必须已经定义了 REPLICA IDENTITY，否则将在这些表上禁止这些操作。

COPY … FROM 命令是作为 INSERT 操作发布的。不发布 TRUNCATE 和 DDL 操作。

语法格式

```
CREATE PUBLICATION name
    [ FOR TABLE table_name [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [=value] [, ... ] ) ];
```

参数说明

- name

  新发布的名称。

- FOR TABLE

  指定要添加到发布的表的列表。 只有持久基表才能成为发布的一部分，临时表、非日志表、外表、MOT 表、物化视图、常规视图不能被发布。

- FOR ALL TABLES

  将发布标记为复制数据库中所有表的更改，包括在将来创建的表。

- WITH (publication_parameter [= value] [, …])

  该子句指定发布的可选参数。支持下列参数：

- publish (string)

  这个参数决定了哪些 DML 操作可以发布给订阅者。该值是一个用逗号分隔的操作列表，允许的操作是 insert、update 和 delete，不指定则默认发布所有的动作。该选项的默认值是'insert, update, delete'。

示例

```
--创建一个发布，发布两个表中所有更改。
CREATE PUBLICATION mypublication FOR TABLE users, departments;
--创建一个发布，发布所有表中的所有更改。
CREATE PUBLICATION alltables FOR ALL TABLES;
--创建一个发布，只发布一个表中的 INSERT 操作。
CREATE PUBLICATION insert_only FOR TABLE mydata WITH (publish = 'insert');
--修改发布的动作。
ALTER PUBLICATION insert_only SET (publish='insert,update,delete');
--向发布中添加表。
ALTER PUBLICAITON insert_only ADD TABLE mydata2;
--删除发布。
DROP PUBLICATION insert_only;
```

相关命令

ALTER PUBLICATION，DROP PUBLICATION

3.8.81 CREATE RESOURCE LABEL

功能描述

创建资源标签。

注意事项

只有 poladmin、sysadmin 或初始用户能正常执行此操作。

语法格式

```
CREATE RESOURCE LABEL [IF NOT EXISTS] label_name ADD label_item_list[, ...];
label_item_list:
resource_type(resource_path[, ...])
resource_type:
TABLE | COLUMN | SCHEMA | VIEW | FUNCTION
```

参数说明

- label_name

  资源标签名称，创建时要求不能与已有标签重名。

  取值范围：字符串，要符合标识符的命名规范。

- resource_type

  指的是要标记的数据库资源类型。

- resource_path

  指的是描述具体的数据库资源的路径。

示例

```
--创建一个表 tb_for_label
postgres=# CREATE TABLE tb_for_label(col1 text, col2 text, col3 text);

--创建一个模式 schema_for_label
postgres=# CREATE SCHEMA schema_for_label;

--创建一个视图 view_for_label
postgres=# CREATE VIEW view_for_label AS SELECT 1;

--创建一个函数 func_for_label
postgres=# CREATE FUNCTION func_for_label RETURNS TEXT AS $$ SELECT col1 FROM
tb_for_label; $$ LANGUAGE SQL;

--基于表创建资源标签
postgres=# CREATE RESOURCE LABEL IF NOT EXISTS table_label add
TABLE(public.tb_for_label);

--基于列创建资源标签
postgres=# CREATE RESOURCE LABEL IF NOT EXISTS column_label add
COLUMN(public.tb_for_label.col1);

--基于模式创建资源标签
postgres=# CREATE RESOURCE LABEL IF NOT EXISTS schema_label add
SCHEMA(schema_for_label);

--基于视图创建资源标签
```

```
postgres=# CREATE RESOURCE LABEL IF NOT EXISTS view_label add
VIEW(view_for_label);

--基于函数创建资源标签
postgres=# CREATE RESOURCE LABEL IF NOT EXISTS func_label add
FUNCTION(func_for_label);
```

相关命令

ALTER RESOURCE LABEL，DROP RESOURCE LABEL。

3.8.82 CREATE RESOURCE POOL

功能描述

创建一个资源池，并指定此资源池相关联的控制组。

注意事项

只要用户对当前数据库有 CREATE 权限，就可以创建资源池。

语法格式

```
CREATE RESOURCE POOL pool_name
    [WITH ({MEM_PERCENT=pct | CONTROL_GROUP="group_name" |
ACTIVE_STATEMENTS=stmt | MAX_DOP = dop | MEMORY_LIMIT='memory_size' |
io_limits=io_limits | io_priority='priority' | nodegroup='nodegroup_name' |
is_foreign = boolean }[, ... ])];
```

参数说明

- pool_name

  资源池名称。

  资源池名称不能和当前数据库里已有的资源池重名。

  取值范围：字符串，要符合标识符的命名规范。

- group_name

  控制组名称。

  说明：设置控制组名称时，语法可以使用双引号，也可以使用单引号。

  group_name 对大小写敏感。

不指定 group_name 时，默认指定的字符串为 “Medium”，代表指定 DefaultClass 控制组的“Medium” Timeshare 控制组。

若数据库管理员指定自定义 Class 组下的 Workload 控制组，如 control_group 的字符串为：“class1:workload1”；代表此资源池指定到 class1 控制组下的 workload1 控制组。也可同时指定 Workload 控制组的层次，如 control_group 的字符串为：“class1:workload1:1”。

若数据库用户指定 Timeshare 控制组代表的字符串，即“Rush”、“High”、“Medium”或“Low”其中一种，如 control_group 的字符串为“High”；代表资源池指定到 DefaultClass 控制组下的“High” Timeshare 控制组。

取值范围：字符串，要符合说明中的规则，其指定已创建的控制组。

- stmt

  资源池语句执行的最大并发数量。

  取值范围：数值型，-1~2147483647。

- dop

  资源池最大并发度，语句执行时能够创建的最多线程数量。

  取值范围：数值型，1~2147483647

- memory_size

  资源池最大使用内存。

  取值范围：字符串，内容范围 1KB~2047GB

- mem_percent

  资源池可用内存占全部内存或者组用户内存使用的比例。

  在多租户场景下，组用户和业务用户的 mem_percent 范围 1-100，默认为 20。

  在普通场景下，普通用户的 mem_percent 范围为 0-100，默认值为 0。

  说明：mem_percent 和 memory_limit 同时指定时，只有 mem_percent 起作用。

- io_limits

  资源池每秒可触发 IO 次数上限。

  对于行存，以万次为单位计数，而列存则以正常次数计数。

- io_priority

IO 利用率高达 90%时，重消耗 IO 作业进行 IO 资源管控时关联的优先级等级。

包括三档可选：Low、Medium 和 High。不控制时可设置为 None。默认为 None。

说明：io_limits 和 io_priority 的设置都仅对复杂作业有效。包括批量导入（INSERT INTO SELECT、COPY FROM、CREATE TABLE AS 等），单 DN 数据量大约超过 500MB 的复杂查询和 VACUUM FULL 等操作。

- nodegroup

在逻辑集群模式下，指定逻辑集群名称。必须是存在的逻辑集群。

如果逻辑集群名称包含大写字符、特殊符号或以数字开头，SQL 语句中对逻辑集群名称需要加双引号。

- is_foreign

在逻辑集群模式下，指定当前资源池用于控制没有关联本逻辑集群的普通用户的资源。这里的逻辑集群是由资源池 nodegroup 字段指定的。

说明：nodegroup 必须是存在的逻辑集群，不能是 elastic_group 和安装的 nodegroup (group_version1)。

如果指定了 is_foreign 为 true，则资源池不能再关联用户，即不允许通过 CREATE USER … RESOURCE POOL 语句来将该资源池配置给用户。该资源池自动检查用户是否关联到资源池指定的逻辑集群，如果用户没有关联到该逻辑集群，则这些用户在逻辑集群所包含的 DN 上运行将受到该资源池的资源控制。

示例

本示例假定用户已预先成功创建控制组。

```
--创建一个默认资源池，其控制组为"DefaultClass"组下属的"Medium" Timeshare Workload 控制组。
postgres=# CREATE RESOURCE POOL pool1;

-- 创建一个资源池，其控制组指定为"DefaultClass"组下属的"High" Timeshare Workload 控制组。
postgres=# CREATE RESOURCE POOL pool2 WITH (CONTROL_GROUP="High");

-- 创建一个资源池，其控制组指定为"class1"组下属的"Low" Timeshare Workload 控制组。
postgres=# CREATE RESOURCE POOL pool3 WITH (CONTROL_GROUP="class1:Low");
```

```
-- 创建一个资源池，其控制组指定为"class1"组下属的"wg1" Workload 控制组。
postgres=# CREATE RESOURCE POOL pool4 WITH (CONTROL_GROUP="class1:wg1");

-- 创建一个资源池，其控制组指定为"class1"组下属的"wg2" Workload 控制组。
postgres=# CREATE RESOURCE POOL pool5 WITH (CONTROL_GROUP="class1:wg2:3");

--删除资源池。
postgres=# DROP RESOURCE POOL pool1;
postgres=# DROP RESOURCE POOL pool2;
postgres=# DROP RESOURCE POOL pool3;
postgres=# DROP RESOURCE POOL pool4;
postgres=# DROP RESOURCE POOL pool5;
```

相关命令

ALTER RESOURCE POOL，DROP RESOURCE POOL

3.8.83 CREATE ROLE

功能描述

创建角色。

角色是拥有数据库对象和权限的实体。在不同的环境中角色可以认为是一个用户，一个组或者兼顾两者。

注意事项

在数据库中添加一个新角色，角色无登录权限。

创建角色的用户必须具备 CREATE ROLE 的权限或者是系统管理员。

语法格式

```
CREATE ROLE role_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ]
{ PASSWORD | IDENTIFIED BY } { 'password' [EXPIRED] | DISABLE };
```

其中角色信息设置子句 option 语法为：

```
{SYSADMIN | NOSYSADMIN}
    | {MONADMIN | NOMONADMIN}
    | {OPRADMIN | NOOPRADMIN}
    | {POLADMIN | NOPOLADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
```

```
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | {PERSISTENCE | NOPERSISTENCE}
    | CONNECTION LIMIT connlimit
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
    | NODE GROUP logic_cluster_name
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
    | DEFAULT TABLESPACE tablespace_name
    | PROFILE DEFAULT
    | PROFILE profile_name
    | PGUSER
```

参数说明

- role_name

  角色名称。

  取值范围：字符串，要符合标识符的命名规范，且最多为 63 个字符。若超过 63 个字符，数据库会截断并保留前 63 个字符当做角色名称。在创建角色时，数据库的时候会给出提示信息。

  说明：标识符需要为字母、下划线、数字（0-9）或美元符号（$），且必须以字母（a-z）或下划线（_）开头。

- password

  登录密码。

密码规则如下：

- 密码默认不少于 8 个字符。
- 不能与用户名及用户名倒序相同。
- 至少包含大写字母（A-Z）、小写字母（a-z）、数字（0-9）、非字母数字字符（限定为~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。
- 密码也可以是符合格式要求的密文字符串，这种情况主要用于用户数据导入场景，不推荐用户直接使用。如果直接使用密文密码，用户需要知道密文密码对应的明文，并且保证明文密码复杂度，数据库不会校验密文密码复杂度，直接使用密文密码的安全性由用户保证。
- 创建角色时，应当使用双引号或单引号将用户密码括起来。

取值范围：不为空的字符串。

- EXPIRED

在创建用户时可指定 EXPIRED 参数，即创建密码失效用户，该用户不允许执行简单查询和扩展查询。只有在修改自身密码后才可正常执行语句。

- DISABLE

默认情况下，用户可以更改自己的密码，除非密码被禁用。要禁用用户的密码，请指定 DISABLE。禁用某个用户的密码后，将从系统中删除该密码，此类用户只能通过外部认证来连接数据库，例如：kerberos 认证。只有管理员才能启用或禁用密码。普通用户不能禁用初始用户的密码。要启用密码，请运行 ALTER USER 并指定密码。

- ENCRYPTED | UNENCRYPTED

控制密码存储在系统表里的口令是否加密。按照产品安全要求，密码必须加密存储，所以，UNENCRYPTED 在 GBase 8s 中禁止使用。因为系统无法对指定的加密口令字符串进行解密，所以如果目前的口令字符串已经是用 SHA256 加密的格式，则会继续照此存放，而不管是否声明了 ENCRYPTED 或 UNENCRYPTED。这样就允许在 dump/restore 的时候重新加载加密的口令。

- SYSADMIN | NOSYSADMIN

决定一个新角色是否为“系统管理员”，具有 SYSADMIN 属性的角色拥有系统最高权限。

缺省为 NOSYSADMIN。

- MONADMIN | NOMONADMIN

定义角色是否是监控管理员。

缺省为 NOMONADMIN。

- OPRADMIN | NOOPRADMIN

  定义角色是否是运维管理员。

  缺省为 NOOPRADMIN。

- POLADMIN | NOPOLADMIN

  定义角色是否是安全策略管理员。

  缺省为 NOPOLADMIN。

- AUDITADMIN | NOAUDITADMIN

  定义角色是否有审计管理属性。

  缺省为 NOAUDITADMIN。

- CREATEDB | NOCREATEDB

  决定一个新角色是否能创建数据库。

  新角色没有创建数据库的权限。

  缺省为 NOCREATEDB。

- USEFT | NOUSEFT

  该参数为保留参数，暂未启用。

- CREATEROLE | NOCREATEROLE

  决定一个角色是否可以创建新角色（也就是执行 CREATE ROLE 和 CREATE USER）。一个拥有 CREATEROLE 权限的角色也可以修改和删除其他角色。

  缺省为 NOCREATEROLE。

- INHERIT | NOINHERIT

  这些子句决定一个角色是否“继承”它所在组的角色的权限。不推荐使用。

- LOGIN | NOLOGIN

  具有 LOGIN 属性的角色才可以登录数据库。一个拥有 LOGIN 属性的角色可以认为是一个用户。

缺省为 NOLOGIN。

- REPLICATION | NOREPLICATION

定义角色是否允许流复制或设置系统为备份模式。REPLICATION 属性是特定的角色，仅用于复制。

缺省为 NOREPLICATION。

- INDEPENDENT | NOINDEPENDENT

定义私有、独立的角色。具有 INDEPENDENT 属性的角色，管理员对其进行的控制、访问的权限被分离，具体规则如下：

未经 INDEPENDENT 角色授权，系统管理员无权对其表对象进行增、删、查、改、拷贝、授权操作。

若将私有用户表的相关权限授予其他非私有用户，系统管理员也会获得同样的权限。

未经 INDEPENDENT 角色授权，系统管理员和拥有 CREATEROLE 属性的安全管理员无权修改 INDEPENDENT 角色的继承关系。

系统管理员无权修改 INDEPENDENT 角色的表对象的属主。

系统管理员和拥有 CREATEROLE 属性的安全管理员无权去除 INDEPENDENT 角色的 INDEPENDENT 属性。

系统管理员和拥有 CREATEROLE 属性的安全管理员无权修改 INDEPENDENT 角色的数据库口令，INDEPENDENT 角色需管理好自身口令，口令丢失无法重置。

管理员属性用户不允许定义修改为 INDEPENDENT 属性。

- VCADMIN | NOVCADMIN

该版本没有实际意义。

- PERSISTENCE | NOPERSISTENCE

定义永久用户。仅允许初始用户创建、修改和删除具有 PERSISTENCE 属性的永久用户。

- CONNECTION LIMIT

声明该角色可以使用的并发连接数量。

须知：

系统管理员不受此参数的限制。

connlimit 数据库主节点单独统计，GBase 8s 整体的连接数 = connlimit * 当前正常数据库主节点个数。

取值范围：整数，>=-1，缺省值为-1，表示没有限制。

- VALID BEGIN

  设置角色生效的时间戳。如果省略了该子句，角色无有效开始时间限制。

- VALID UNTIL

  设置角色失效的时间戳。如果省略了该子句，角色无有效结束时间限制。

- RESOURCE POOL

  设置角色使用的 resource pool 名称，该名称属于系统表：pg_resource_pool。

- PERM SPACE

  设置用户使用空间的大小。

- TEMP SPACE

  设置用户临时表存储空间限额。

- SPILL SPACE

  设置用户算子落盘空间限额。

- IN ROLE

  新角色立即拥有 IN ROLE 子句中列出的一个或多个现有角色拥有的权限。不推荐使用。

- IN GROUP

  IN GROUP 是 IN ROLE 过时的拼法。不推荐使用。

- ROLE

  ROLE 子句列出一个或多个现有的角色，它们将自动添加为这个新角色的成员，拥有新角色所有的权限。

- ADMIN

  ADMIN 子句类似 ROLE 子句，不同的是 ADMIN 后的角色可以把新角色的权限赋给其他角色。

- USER

  USER 子句是 ROLE 子句过时的拼法。

- SYSID

  SYSID 子句将被忽略，无实际意义。

- DEFAULT TABLESPACE

  DEFAULT TABLESPACE 子句将被忽略，无实际意义。

- PROFILE

  PROFILE 子句将被忽略，无实际意义。

- PGUSER

  当前版本该属性没有实际意义，仅为了语法的前向兼容而保留。

示例

```
--创建一个角色，名为 manager，密码为 xxxxxxxxx。
postgres=# CREATE ROLE manager IDENTIFIED BY 'xxxxxxxxx';

--创建一个角色，从 2015 年 1 月 1 日开始生效，到 2026 年 1 月 1 日失效。
postgres=# CREATE ROLE miriam WITH LOGIN PASSWORD 'xxxxxxxxx' VALID BEGIN
'2015-01-01' VALID UNTIL '2026-01-01';

--修改角色 manager 的密码为 abcd@123。
postgres=# ALTER ROLE manager IDENTIFIED BY 'abcd@123' REPLACE 'xxxxxxxxx';

--修改角色 manager 为系统管理员。
postgres=# ALTER ROLE manager SYSADMIN;

--删除角色 manager。
postgres=# DROP ROLE manager;

--删除角色 miriam。
postgres=# DROP ROLE miriam;
```

相关命令

SET ROLE，ALTER ROLE，DROP ROLE，GRANT

3.8.84 CREATE ROW LEVEL SECURITY POLICY

功能描述

对表创建行访问控制策略。

当对表创建了行访问控制策略，只有打开该表的行访问控制开关(ALTER TABLE … ENABLE ROW LEVEL SECURITY)，策略才能生效。否则不生效。

当前行访问控制影响数据表的读取操作(SELECT、UPDATE、DELETE)，暂不影响数据表的写入操作(INSERT、MERGE INTO)。表所有者或系统管理员可以在 USING 子句中创建表达式，在客户端执行数据表读取操作时，数据库后台在查询重写阶段会将满足条件的表达式拼接并应用到执行计划中。针对数据表的每一条元组，当 USING 表达式返回 TRUE 时，元组对当前用户可见，当 USING 表达式返回 FALSE 或 NULL 时，元组对当前用户不可见。

行访问控制策略名称是针对表的，同一个数据表上不能有同名的行访问控制策略；对不同的数据表，可以有同名的行访问控制策略。

行访问控制策略可以应用到指定的操作(SELECT、UPDATE、DELETE、ALL)，ALL 表示会影响 SELECT、UPDATE、DELETE 三种操作；定义行访问控制策略时，若未指定受影响的相关操作，默认为 ALL。

行访问控制策略可以应用到指定的用户(角色)，也可应用到全部用户(PUBLIC)；定义行访问控制策略时，若未指定受影响的用户，默认为 PUBLIC。

注意事项

支持对行存表、行存分区表、列存表、列存分区表、unlogged 表、hash 表定义行访问控制策略。

不支持外表、本地临时表定义行访问控制策略。

不支持对视图定义行访问控制策略。

同一张表上可以创建多个行访问控制策略，一张表最多创建 100 个行访问控制策略。

系统管理员不受行访问控制影响，可以查看表的全量数据。

通过 SQL 语句、视图、函数、存储过程查询包含行访问控制策略的表，都会受影响。

语法格式

```
CREATE [ ROW LEVEL SECURITY ] POLICY policy_name ON table_name
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC } [, ...] ]
    USING ( using_expression )
```

参数说明

- policy_name

行访问控制策略名称，同一个数据表上行访问控制策略名称不能相同。

- table_name

  行访问控制策略的表名。

- PERMISSIVE | RESTRICTIVE

  PERMISSIVE 指定行访问控制策略为宽容性策略，宽容性策略的条件用 OR 表达式拼接。

  RESTRICTIVE 指定行访问控制策略为限制性策略，限制性策略的条件用 AND 表达式拼接。拼接方式如下：

  (using_expression_permissive_1 OR using_expression_permissive_2 ...) AND (using_expression_restrictive_1 AND using_expression_restrictive_2 ...)

  缺省值为 PERMISSIVE。

- command

  当前行访问控制影响的 SQL 操作，可指定操作包括：ALL、SELECT、UPDATE、DELETE。当未指定时，ALL 为默认值，涵盖 SELECT、UPDATE、DELETE 操作。

  当 command 为 SELECT 时，SELECT 类操作受行访问控制的影响，只能查看到满足条件(using_expression 返回值为 TRUE)的元组数据，受影响的操作包括 SELECT、UPDATE … RETURNING、DELETE … RETURNING。

  当 command 为 UPDATE 时，UPDATE 类操作受行访问控制的影响，只能更新满足条件(using_expression 返回值为 TRUE)的元组数据，受影响的操作包括 UPDATE、 UPDATE … RETURNING、 SELECT … FOR UPDATE/SHARE。

  当 command 为 DELETE 时，DELETE 类操作受行访问控制的影响，只能删除满足条件(using_expression 返回值为 TRUE)的元组数据，受影响的操作包括 DELETE、 DELETE … RETURNING。

  行访问控制策略与适配的 SQL 语法关系参加下表：

表 3-26 ROW LEVEL SECURITY 策略与适配 SQL 语法关系

SQL 语法命令	SELECT/ALL policy	UPDATE/ALL policy	DELETE/ALL policy

SQL 语法命令	SELECT/ALL policy	UPDATE/ALL policy	DELETE/ALL policy
SELECT	Existing row	No	No
SELECT FOR UPDATE/SHARE	Existing row	Existing row	No
UPDATE	No	Existing row	No
UPDATE RETURNING	Existing row	Existing row	No
DELETE	No	No	Existing row
DELETE RETURNING	Existing row	No	Existing row

- role_name

  行访问控制影响的数据库用户。

  当未指定时，PUBLIC 为默认值，PUBLIC 表示影响所有数据库用户，可以指定多个受影响的数据库用户。

  须知：系统管理员不受行访问控制特性影响。

- using_expression

  行访问控制的表达式（返回 boolean 值）。

  条件表达式中不能包含 AGG 函数和窗口（WINDOW）函数。在查询重写阶段，如果数据表的行访问控制开关打开，满足条件的表达式会添加到计划树中。针对数据表的每条元组，会进行表达式计算，只有表达式返回值为 TRUE 时，行数据对用户才可见（SELECT、UPDATE、DELETE）；当表达式返回 FALSE 时，该元组对当前用户不可见，用户无法通过 SELECT 语句查看此元组，无法通过 UPDATE 语句更新此元组，无法通过 DELETE 语句删除此元组。

示例

```
--创建用户 alice
postgres=# CREATE USER alice PASSWORD 'xxxxxxxxx';

--创建用户 bob
postgres=# CREATE USER bob PASSWORD 'xxxxxxxxx';

--创建数据表 all_data
postgres=# CREATE TABLE all_data(id int, role varchar(100), data varchar(100));

--向数据表插入数据
postgres=# INSERT INTO all_data VALUES(1, 'alice', 'alice data');
postgres=# INSERT INTO all_data VALUES(2, 'bob', 'bob data');
postgres=# INSERT INTO all_data VALUES(3, 'peter', 'peter data');

--将表 all_data 的读取权限赋予 alice 和 bob 用户
postgres=# GRANT SELECT ON all_data TO alice, bob;

--打开行访问控制策略开关
postgres=# ALTER TABLE all_data ENABLE ROW LEVEL SECURITY;

--创建行访问控制策略，当前用户只能查看用户自身的数据
postgres=# CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role
= CURRENT_USER);

--查看表 all_data 相关信息
postgres=# \d+ all_data
                              Table "public.all_data"
 Column |          Type          | Modifiers | Storage  | Stats target |
Description
--------+------------------------+-----------+----------+--------------+-----
--------
 id     | integer                |           | plain    |              |
 role   | character varying(100) |           | extended |              |
 data   | character varying(100) |           | extended |              |
Row Level Security Policies:
    POLICY "all_data_rls"
      USING (((role)::name = "current_user"()))
Has OIDs: no
Options: orientation=row, compression=no, enable_rowsecurity=true
```

```
--当前用户执行 SELECT 操作
postgres=# SELECT * FROM all_data;
 id | role  |    data
----+-------+------------
  1 | alice | alice data
  2 | bob   | bob data
  3 | peter | peter data
(3 rows)

postgres=# EXPLAIN(COSTS OFF) SELECT * FROM all_data;
      QUERY PLAN
----------------------
 Seq Scan on all_data
(1 row)

--切换至 alice 用户执行 SELECT 操作
postgres=# SELECT * FROM all_data;
 id | role  |    data
----+-------+------------
  1 | alice | alice data
(1 row)

postgres=# EXPLAIN(COSTS OFF) SELECT * FROM all_data;
 QUERY PLAN
------------------------------------------------------------------
 Seq Scan on all_data
   Filter: ((role)::name = 'alice'::name)
 Notice: This query is influenced by row level security feature
(3 rows)
```

相关命令

DROP ROW LEVEL SECURITY POLICY，ALTER ROW LEVEL SECURITY POLICY

3.8.85CREATE RULE

功能描述

定义一个新的重写规则。

注意事项

为了在表上定义或修改规则，你必须是该表的拥有者。

如果在同一个表定义了多个相同类型的规则，则按规则的名称字母顺序触发它们。

在视图上用于 INSERT、UPDATE、DELETE 的规则中可以添加 RETURNING 子句基于视图的字段返回。如果规则被 INSERT RETURNING、UPDATE RETURNING、DELETE RETURNING 命令触发，这些子句将用来计算输出结果。如果规则被不带 RETURNING 的命令触发，那么规则的 RETURNING 子句将被忽略。目前仅允许无条件的 INSTEAD 规则包含 RETURNING 子句，而且在同一个事件内的所有规则中最多只能有一个 RETURNING 子句。这样就确保只有一个 RETURNING 子句可以用于计算结果。如果在任何有效规则中都不存在 RETURNING 子句，该视图上的 RETURNING 查询将被拒绝。

不建议在 rule 内使用列存表，尤其是一些写操作。因为列存表与行存表的架构实现、事务处理等存在很大差异，因此 rule 的表现也会有很多与行存表不同的地方。

语法格式

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

其中 event 包含 SELECT、INSER、DELETE、UPDATE

参数说明

- name

  创建的规则名。它必须在同一个表上的所有规则名字中唯一。

  取值范围：符合标识符命名规范的字符串，且最大长度不超过 63 个字符。

- table_name

  规则作用的表或者视图的名字（可以有模式修饰）。

- condition

  返回 boolean 的 SQL 条件表达式，决定是否实际执行规则。表达式除了引用 NEW 和 OLD 之外不能引用任何表， 并且不能有聚合函数。

- INSTEAD

  INSTEAD 指示使用该命令替换初始事件。

- ALSO

  ALSO 指示该命令应该在初始事件执行之后执行。如果既没有声明 ALSO 也没有声明 INSTEAD，那么 ALSO 为缺省值。

- command

组成规则动作的命令。有效的命令是 SELECT、 INSERT、UPDATE、 DELETE 语句之一。

示例

```
CREATE RULE "_RETURN" AS
   ON SELECT TO t1
   DO INSTEAD
       SELECT * FROM t2;
```

3.8.86 CREATE SCHEMA

功能描述

创建模式。

访问命名对象时可以使用模式名作为前缀进行访问，如果无模式名前缀，则访问当前模式下的命名对象。创建命名对象时也可用模式名作为前缀修饰。

另外，CREATE SCHEMA 可以包括在新模式中创建对象的子命令，这些子命令和那些在创建完模式后发出的命令没有任何区别。如果使用了 AUTHORIZATION 子句，则所有创建的对象都将被该用户所拥有。

注意事项

只要用户对当前数据库有 CREATE 权限，就可以创建模式。

系统管理员在普通用户同名 schema 下创建的对象，所有者为 schema 的同名用户（非系统管理员）。

语法格式

根据指定的名称创建模式。

```
CREATE SCHEMA [IF NOT EXISTS] schema_name
[ AUTHORIZATION user_name ] [WITH BLOCKCHAIN] [ schema_element [ ... ] ];
```

参数说明

- IF NOT EXISTS

若指定该参数，则当一个同名的模式已经存在时，仅发出提示。

- schema_name

模式名称。

取值范围：字符串，要符合标识符的命名规范。

须知

- 模式名不能和当前数据库里其他的模式重名。
- 模式的名称不可以“pg_”开头。

- AUTHORIZATION user_name

指定模式的所有者。当不指定 schema_name 时，则默认创建一个与 user_name 同名的模式，此时 user_name 只能是角色名。

取值范围：已存在的用户名/角色名。

- WITH BLOCKCHAIN

指定模式的防篡改属性，防篡改模式下的行存普通用户表将自动扩展为防篡改用户表。

- schema_element

在模式里创建对象的 SQL 语句。目前仅支持 CREATE TABLE、CREATE VIEW、CREATE INDEX、CREATE PARTITION、CREATE SEQUENCE、CREATE TRIGGER、GRANT 子句。

子命令所创建的对象都被 AUTHORIZATION 子句指定的用户所拥有。

说明

- 如果当前搜索路径上的模式中存在同名对象时，需要明确指定引用对象所在的模式。可以通过命令 SHOW SEARCH_PATH 来查看当前搜索路径上的模式。

示例

```
--创建一个角色 role1。
postgres=# CREATE ROLE role1 IDENTIFIED BY 'Role,123';

--为用户 role1 创建一个同名 schema，子命令创建的表 films 和 winners 的拥有者为 role1。
postgres=# CREATE SCHEMA AUTHORIZATION role1
gbase-# CREATE TABLE films (title text, release date, awards text[])
gbase-# CREATE VIEW winners AS
gbase-# SELECT title, release FROM films WHERE awards IS NOT NULL;

--删除 schema。
postgres=# DROP SCHEMA role1 CASCADE;
```

```
--删除用户。
postgres=# DROP USER role1 CASCADE;
```

相关命令

ALTER SCHEMA，DROP SCHEMA

3.8.87 CREATE SEQUENCE

功能描述

CREATE SEQUENCE 用于向当前数据库里增加一个新的序列。序列的 Owner 为创建此序列的用户。

序列 Sequence 是用来产生唯一整数的数据库对象。序列的值是按照一定规则自增的整数。因为自增所以不重复，因此说 Sequence 具有唯一标识性。这也是 Sequence 常被用作主键的原因。

通过序列使某字段成为唯一标识符的方法有两种：

一种是声明字段的类型为序列整型，由数据库在后台自动创建一个对应的 Sequence。

另一种是使用 CREATE SEQUENCE 自定义一个新的 Sequence，然后将 nextval('sequence_name')函数读取的序列值，指定为某一字段的默认值，这样该字段就可以作为唯一标识符。

注意事项

Sequence 是一个存放等差数列的特殊表。这个表没有实际意义，通常用于为行或者表生成唯一的标识符。

如果给出一个模式名，则该序列就在给定的模式中创建，否则会在当前模式中创建。序列名必须和同一个模式中的其他序列、表、索引、视图或外表的名称不同。

创建序列后，在表中使用序列的 nextval()函数和 generate_series(1,N)函数对表插入数据，请保证 nextval 的可调用次数大于等于 N+1 次，否则会因为 generate_series()函数会调用 N+1 次而导致报错。

Sequence 默认最大值为 2^63-1，如果使用了 Large 标识则最大值可以支持到 2^127-1。

被授予 CREATE ANY SEQUENCE 权限的用户，可以在 public 模式和用户模式下创建序列。

语法格式

```
CREATE  [ TEMPORARY | TEMP ] [ LARGE ] SEQUENCE name [ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE | NOMINVALUE] [ MAXVALUE maxvalue | NO MAXVALUE
| NOMAXVALUE] [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE | NOCYCLE]
[ OWNED BY { table_name.column_name | NONE } ];
```

参数说明

- [TEMPORARY | TEMP]

如果被指定，只会为这个会话创建序列对象，并且在会话退出时自动 删除它。当临时序列存在时，已有的同名永久序列（在这个会话中） 会变得不可见，不过可以用模式限定的名称来引用同名永久序列。

- name

将要创建的序列名称。

取值范围：仅可以使用小写字母（a~z）、 大写字母（A~Z）、数字和特殊字符“#”，“_”，“$”的组合。

- increment

指定序列的步长。一个正数将生成一个递增的序列，一个负数将生成一个递减的序列。

缺省值为 1。

- MINVALUE minvalue | NO MINVALUE| NOMINVALUE

执行序列的最小值。如果没有声明 minvalue 或者声明了 NO MINVALUE，则递增序列的缺省值为 1，递减序列的缺省值为-263-1。NOMINVALUE 等价于 NO MINVALUE

- MAXVALUE maxvalue | NO MAXVALUE| NOMAXVALUE

执行序列的最大值。如果没有声明 maxvalue 或者声明了 NO MAXVALUE，则递增序列的缺省值为 263-1，递减序列的缺省值为-1。NOMAXVALUE 等价于 NO MAXVALUE

- start

指定序列的起始值。

缺省值：对于递增序列为 minvalue，递减序列为 maxvalue。

- cache

为了快速访问，而在内存中预先存储序列号的个数。

缺省值为 1，表示一次只能生成一个值，也就是没有缓存。

说明

- 不建议同时定义 cache 和 maxvalue 或 minvalue。因为定义 cache 后不能保证序列的连续性，可能会产生空洞，造成序列号段浪费。

- CYCLE

用于使序列达到 maxvalue 或者 minvalue 后可循环并继续下去。

如果声明了 NO CYCLE，则在序列达到其最大值后任何对 nextval 的调用都会返回一个错误。NOCYCLE 的作用等价于 NO CYCLE。

若定义序列为 CYCLE，则不能保证序列的唯一性。

缺省值为 NO CYCLE。

- OWNED BY

将序列和一个表的指定字段进行关联。这样，在删除那个字段或其所在表的时候会自动删除已关联的序列。关联的表和序列的所有者必须是同一个用户，并且在同一个模式中。需要注意的是，通过指定 OWNED BY，仅仅是建立了表的对应列和 sequence 之间关联关系，并不会在插入数据时在该列上产生自增序列。

缺省值为 OWNED BY NONE，表示不存在这样的关联。

须知

- 通过 OWNED BY 创建的 Sequence 不建议用于其他表，如果希望多个表共享 Sequence，该 Sequence 不应该从属于特定表。

示例

示例 1

创建一个名为 serial 的递增序列，从 101 开始：

```
postgres=# CREATE SEQUENCE serial START 101 CACHE 20;
```

从序列中选出下一个数字：

```
postgres=# SELECT nextval('serial');
 nextval
---------
     101
```

从序列中选出下一个数字：

```
postgres=# SELECT nextval('serial');
```

```
 nextval
 ---------
      102
```

创建与表关联的序列：

```
postgres=# CREATE TABLE customer_address
(
    ca_address_sk             integer               not null,
    ca_address_id             char(16)              not null,
    ca_street_number          char(10)                      ,
    ca_street_name            varchar(60)                   ,
    ca_street_type            char(15)                      ,
    ca_suite_number           char(10)                      ,
    ca_city                   varchar(60)                   ,
    ca_county                 varchar(30)                   ,
    ca_state                  char(2)                       ,
    ca_zip                    char(10)                      ,
    ca_country                varchar(20)                   ,
    ca_gmt_offset             decimal(5,2)                  ,
    ca_location_type          char(20)
);

postgres=# CREATE SEQUENCE serial1
 START 101
 CACHE 20
OWNED BY customer_address.ca_address_sk;
--删除表和序列
postgres=# DROP TABLE customer_address;
postgres=# DROP SEQUENCE serial cascade;
postgres=# DROP SEQUENCE serial1 cascade;
```

示例 2

声明字段类型为序列整型来定义标识符字段。例如：

```
CREATE TABLE T1
(
    id    serial,
    name  text
);
```

当结果显示为如下信息，则表示创建成功。

```
NOTICE:  CREATE TABLE will create implicit sequence "t1_id_seq" for serial column
"t1.id"
CREATE TABLE
```

示例 3

创建序列，并通过 nextval('sequence_name')函数指定为某一字段的默认值。

```
CREATE SEQUENCE seq1 cache 100;
```

指定为某一字段的默认值，使该字段具有唯一标识属性。

```
CREATE TABLE T2
(
    id   int not null default nextval('seq1'),
    name text
);
```

指定序列与列的归属关系。将序列和一个表的指定字段进行关联。这样，在删除那个字段或其所在表的时候会自动删除已关联的序列。

```
ALTER SEQUENCE seq1 OWNED BY T2.id;
```

相关命令

DROP SEQUENCE，ALTER SEQUENCE

3.8.88 CREATE SERVER

功能描述

定义一个新的外部服务器。

语法格式

```
CREATE SERVER server_name
    FOREIGN DATA WRAPPER fdw_name
    OPTIONS ( { option_name ' value ' } [, ...] ) ;
```

参数说明

- server_name

  server 的名称。

  取值范围：长度必须小于等于 63。

- fdw_name

  指定外部数据封装器的名称。

取值范围：dist_fdw，hdfs_fdw，log_fdw，file_fdw，mot_fdw。

- OPTIONS ({ option_name ' value ' } [, …])

这个子句为服务器指定选项。这些选项通常定义该服务器的连接细节，但是实际的名称和值取决于该服务器的外部数据包装器。

- oracle_fdw 支持的 options 包括：
  - dbserver

远端 Oracle 数据库的连接字符串。

  - isolation_level （默认值为 serializable）

oracle 数据库的事务隔离级别。

取值范围：serializable、 read_committed 、 read_only

- mysql_fdw 支持的 options 包括：
  - host （默认值为 127.0.0.1）

MySQL Server/MariaDB 的地址。

  - port （默认值为 3306）

MySQL Server/MariaDB 侦听的端口号。

- postgres_fdw 支持的 options 同 libpq 支持的连接参数一致。需要注意的是，以下几个 options 不支持设置：
  - user 和 password

用户名和密码将在创建 user mapping 时指定。

  - client_encoding

将自动获取本地 server 的编码方式并设置该值。

  - application_name

总是设置成 postgres_fdw。

- 用于指定外部服务器的各类参数，详细的参数说明如下所示。
  - encrypt

是否对数据进行加密，该参数仅支持 type 为 OBS 时设置。默认值为 on。

取值范围：

on 表示对数据进行加密，使用 HTTPS 协议通信。

off 表示不对数据进行加密，使用 HTTP 协议通信。

- access_key

OBS 访问协议对应的 AK 值（OBS 云服务界面由用户获取），创建外表时 AK 值会加密保存到数据库的元数据表中。该参数仅支持 type 为 OBS 时设置。

- secret_access_key

OBS 访问协议对应的 SK 值（OBS 云服务界面由用户获取），创建外表时 SK 值会加密保存到数据库的元数据表中。该参数仅支持 type 为 OBS 时设置。

- 除了 libpq 支持的连接参数外，还额外提供 3 个 options：

  - use_remote_estimate

控制 postgres_fdw 是否发出 EXPLAIN 命令以获取运行消耗估算。默认值为 false。

  - fdw_startup_cost

执行一个外表扫描时的启动耗时估算。这个值通常包含建立连接、远端对请求的分析和生成计划的耗时。默认值为 100。

  - fdw_typle_cost

在远端服务器上对每一个元组进行扫描时的额外消耗。这个值通常表示数据在 server 间传输的额外消耗。默认值为 0.01。

示例

创建 server。

```
postgres=# create server my_server foreign data wrapper log_fdw;
CREATE SERVER
```

相关命令

ALTER SERVER，DROP SERVER

3.8.89 CREATE SUBSCRIPTION

功能描述

为当前数据库添加一个新的订阅。订阅名称必须与数据库中任何现有的订阅不同。 订

阅表示到发布者的复制连接。因此，此命令不仅在本地系统表中添加定义，还会在发布端创建复制槽。 在运行此命令的事务提交时，将启动逻辑复制线程以复制新订阅的数据。

注意事项

创建复制槽时（默认行为），CREATE SUBSCRIPTION 不能在事务块内部执行。

语法格式

```
CREATE SUBSCRIPTION subscription_name
    CONNECTION 'conninfo'
    PUBLICATION publication_name [, ...]
    [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

参数说明

- subscription_name

  新订阅的名称。

- CONNECTION 'conninfo'

  连接发布端的字符串。

  如 'host=1.1.1.1,2.2.2.2 port=10000,20000 dbname=postgres user=repusr1 password=password_123'。

- host

  发布端 IP 地址，可以同时指定发布端主机和备机的 IP 地址，如果同时指定了多个 IP，以英文逗号分隔。

- port

  发布端端口，此处的端口不能使用主端口，而应该使用主端口+1 端口，否则会与线程池冲突。可以同时指定发布端主机和备机的端口，如果同时指定了多个端口，以英文逗号分隔。

  注意：host 和 port 的数量要一致，并且要一一对应。

- dbname

  发布所在的数据库。

- user 和 password

  用于连接发布端且具有系统管理员权限(SYSADMIN)或者运维管理员权限(OPRADMIN)

的用户名和密码。password 需要加密，创建订阅前需要在订阅端执行 gs_guc generate -S xxxxxx -D $GAUSSHOME/bin -o subscription。

- PUBLICATION publication_name

  要订阅的发布端的发布名称，一个订阅可以对应多个发布。

- WITH (subscription_parameter [= value] [, …])

  该子句指定订阅的可选参数。支持的参数有：

- enabled (boolean)

  指定订阅是否应该主动复制，或者是否应该只是设置，但尚未启动。默认值是 true。

- slot_name (string)

  要使用的复制插槽的名称。默认使用订阅名称作为复制槽的名称。

  如果创建订阅时设置 enable 为 false，则 slot_name 将被强制设置为 NONE，即空值，即使用户指定了 slot_name 的值，表示复制槽不存在。

- synchronous_commit (enum)

  该参数的值会覆盖 synchronous_commit 设置。 默认值是 off。

  对于逻辑复制使用 off 是安全的，如果订阅端由于缺少同步而丢失事务，数据将从发布者再次发送。进行同步逻辑复制时，一个不同的设置可能是合适的。逻辑复制线程向发布端报告写入和刷新的位置，当使用同步复制时，发布端将等待实际刷新。这意味着，当订阅用于同步复制时，将订阅者的 synchronous_commit 设置为 off 可能会增加发布端服务器上 COMMIT 的延迟。在这种情况下，将 synchronous_commit 设置为 local 或更高是有利的。

- binary (boolean)

  该参数指定是否需要该订阅对应的发布端以二进制格式发送数据，为 true 表示需要以二进制发送，为 false 表示不以二进制格式而知以默认的文本格式发送。默认值 false。

示例

```
--创建一个到远程服务器的订阅，复制发布 mypublication 和 insert_only 中的表，并在提交时立即开始复制。
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb password=xxxx'
        PUBLICATION mypublication, insert_only;
```

```
--创建一个到远程服务器的订阅，复制 insert_only 发布中的表， 并且不开始复制直到稍后启用复制。
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb password=xxxx '
        PUBLICATION insert_only
               WITH (enabled = false);
--修改订阅的连接信息。
ALTER SUBSCRIPTION mysub CONNECTION 'host=192.168.1.51 port=5432 user=foo dbname=foodb password=xxxx';
--激活订阅。
ALTER SUBSCRIPTION mysub SET(enabled=true);
--删除订阅。
DROP SUBSCRIPTION mysub;
```

相关命令

ALTER SUBSCRIPTION，DROP SUBSCRIPTION

3.8.90 CREATE SYNONYM

功能描述

创建一个同义词对象。同义词是数据库对象的别名，用于记录与其他数据库对象名间的映射关系，用户可以使用同义词访问关联的数据库对象。

注意事项

定义同义词的用户成为其所有者。

若指定模式名称，则同义词在指定模式中创建。否则，在当前模式创建。

支持通过同义词访问的数据库对象包括：表、视图、函数和存储过程。

使用同义词时，用户需要具有对关联对象的相应权限。

支持使用同义词的 DML 语句包括：SELECT、INSERT、UPDATE、DELETE、EXPLAIN、CALL。

不建议对临时表创建同义词。如果需要创建的话，需要指定同义词的目标临时表的模式名，负责无法正常使用同义词，并且在当前会话结束前执行 DROP SYNONYM 命令。

删除原对象后，与之关联同义词不会被级联删除，继续访问该同义词会报错，并提示已失效。

语法格式

```
CREATE [ OR REPLACE ] SYNONYM synonym_name
    FOR object_name;
```

参数说明

- synonym

  创建的同义词名字，可以带模式名。

  取值范围：字符串，要符合标识符的命名规范。

- object_name

  关联的对象名字，可以带模式名。

  取值范围：字符串，要符合标识符的命名规范。

  说明：object_name 可以是不存在的对象名称。

示例

```
--创建模式 ot。
postgres=# CREATE SCHEMA ot;

--创建表 ot.t1 及其同义词 t1。
postgres=# CREATE TABLE ot.t1(id int, name varchar2(10));
postgres=# CREATE OR REPLACE SYNONYM t1 FOR ot.t1;

--使用同义词 t1。
postgres=# SELECT * FROM t1;
postgres=# INSERT INTO t1 VALUES (1, 'ada'), (2, 'bob');
postgres=# UPDATE t1 SET t1.name = 'cici' WHERE t1.id = 2;

--创建同义词 v1 及其关联视图 ot.v_t1。
postgres=# CREATE SYNONYM v1 FOR ot.v_t1;
postgres=# CREATE VIEW ot.v_t1 AS SELECT * FROM ot.t1;

--使用同义词 v1。
postgres=# SELECT * FROM v1;

--创建重载函数 ot.add 及其同义词 add。
postgres=# CREATE OR REPLACE FUNCTION ot.add(a integer, b integer) RETURNS integer
AS
$$
SELECT $1 + $2
$$
```

```
LANGUAGE sql;

postgres=# CREATE OR REPLACE FUNCTION ot.add(a decimal(5,2), b decimal(5,2))
RETURNS decimal(5,2) AS
$$
SELECT $1 + $2
$$
LANGUAGE sql;

postgres=# CREATE OR REPLACE SYNONYM add FOR ot.add;

--使用同义词 add。
postgres=# SELECT add(1,2);
postgres=# SELECT add(1.2,2.3);

--创建存储过程 ot.register 及其同义词 register。
postgres=# CREATE PROCEDURE ot.register(n_id integer, n_name varchar2(10))
SECURITY INVOKER
AS
BEGIN
    INSERT INTO ot.t1 VALUES(n_id, n_name);
END;
/

postgres=# CREATE OR REPLACE SYNONYM register FOR ot.register;

--使用同义词 register，调用存储过程。
postgres=# CALL register(3,'mia');

--删除同义词。
postgres=# DROP SYNONYM t1;
postgres=# DROP SYNONYM IF EXISTS v1;
postgres=# DROP SYNONYM IF EXISTS add;
postgres=# DROP SYNONYM register;
postgres=# DROP SCHEMA ot CASCADE;
```

相关命令

ALTER SYNONYM，DROP SYNONYM

3.8.91 CREATE TABLE

功能描述

在当前数据库中创建一个新的空白表，该表由命令执行者所有。

注意事项

列存表支持的数据类型请参考列存表支持的数据类型。

列存表不支持数组。

列存表不支持生成列。

列存表不支持创建全局临时表。

创建列存表的数量建议不超过 1000 个。

表中的主键约束和唯一约束必须包含分布列。

如果在建表过程中数据库系统发生故障，系统恢复后可能无法自动清除之前已创建的、大小为 0 的磁盘文件。此种情况出现概率小，不影响数据库系统的正常运行。

列存表的表级约束只支持 PARTIAL CLUSTER KEY、UNIQUE、PRIAMRY KEY，不支持外键等表级约束。

列存表的字段约束只支持 NULL、NOT NULL、DEFAULT 常量值、UNIQUE 和 PRIMARY KEY。

列存表支持 delta 表，受参数 enable_delta_store 控制是否开启，受参数 deltarow_threshold 控制进入 delta 表的阀值。

使用 JDBC 时，支持通过 PrepareStatement 对 DEFAUTL 值进行参数化设置。

每张表的列数最大为 1600，具体取决于列的类型，所有列的大小加起来不能超过 8192 byte，text、varchar、char 等长度可变的类型除外。

被授予 CREATE ANY TABLE 权限的用户，可以在 public 模式和用户模式下创建表。如果想要创建包含 serial 类型列的表，还需要授予 CREATE ANY SEQUENCE 创建序列的权限。

语法格式

创建表。

```
CREATE [ [ GLOBAL | LOCAL ] [ TEMPORARY | TEMP ] | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name
    ({ column_name data_type [ compress_mode ] [ COLLATE collation ]
[ column_constraint [ ... ] ]
        | table_constraint
        | LIKE source_table [ like_option [...] ] }
        [, ... ])
```

```
    [ INHERITS ( parent_table [, ... ] ) ]
    [ WITH ( {storage_parameter = value} [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
    [ COMPRESS | NOCOMPRESS ]
    [ TABLESPACE tablespace_name ]
    [ DISTRIBUTE BY { REPLICATION | HASH ( column_name [, ...] ) }
    } ]
    [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
    [ COMMENT {=| } 'text' ];
```

其中列约束 column_constraint 为：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  ENCRYPTED WITH ( COLUMN_ENCRYPTION_KEY = column_encryption_key,
ENCRYPTION_TYPE = encryption_type_value ) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

其中列的压缩可选项 compress_mode 为：

```
{ DELTA | PREFIX | DICTIONARY | NUMSTR | NOCOMPRESS }
```

其中表约束 table_constraint 为：

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  PARTIAL CLUSTER KEY ( column_name [, ... ] ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
[, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE
action ] }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

其中 like 选项 like_option 为：

```
{ INCLUDING | EXCLUDING } { DEFAULTS | GENERATED | CONSTRAINTS | INDEXES | STORAGE
| COMMENTS | PARTITION | R
ELOPTIONS | DISTRIBUTION | ALL }
```

其中索引参数 index_parameters 为：

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

- 创建分区表

创建 VALUES LESS THAN 范围分区表**语法格式**

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ]) PARTITION
BY RANGE (partition_key) ( PARTITION partition_name VALUES LESS THAN
(partition_value | MAXVALUE}) [, ... ] );
```

创建 START END 范围分区表**语法格式**

START END 范围分区表有多种表达方式，而且这些方式可以在一个分区表内组合使用。

方式一：START(partition_value) END (partition_value | MAXVALUE)方式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ) PARTITION
BY RANGE (partition_key) ( PARTITION partition_name START(partition_value) END
(partition_value | MAXVALUE) [, ... ] );
```

方式二：START(partition_value)方式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ] ) PARTITION
BY RANGE (partition_key) ( PARTITION partition_name START(partition_value)
[, ... ] );
```

方式三：END(partition_value | MAXVALUE)方式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ] ) PARTITION
BY RANGE (partition_key) ( PARTITION partition_name END(partition_value |
MAXVALUE) [, ... ] );
```

方式四：START(partition_value) END (partition_value) EVERY (interval_value)方式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ] ) PARTITION
BY RANGE (partition_key) ( PARTITION partition_name START(partition_value) END
(partition_value) EVERY (interval_value) [, ... ] );
```

创建列表分区表语法格式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
  [, ... ]
```

```
)
    PARTITION BY LIST (partition_key)
        (
        PARTITION partition_name VALUES (list_values_clause)
        [, ... ]
        );
```

创建间隔分区表语法格式

间隔分区是在范围分区的基础上，增加了间隔值 PARTITION BY RANGE (partition_key) 的定义。

VALUES LESS THAN 间隔分区语法格式：

```
CREATE TABLE partition_table_name
( [column_name data_type ]
  [, ... ]
)
    PARTITION BY RANGE (partition_key)
        (
        INTERVAL ('interval_expr')
        PARTITION partition_name VALUES LESS THAN (partition_value | MAXVALUE})
        [, ... ]
        );
```

START END 间隔分区表语法格式：

方式一：START(partition_value) END (partition_value | MAXVALUE)方式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ) PARTITION BY RANGE (partition_key) ( INTERVAL ('interval_expr') PARTITION partition_name START(partition_value) END (partition_value | MAXVALUE) [, ... ] );
```

方式二：START(partition_value) END (partition_value) EVERY (interval_value)方式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ] ) PARTITION BY RANGE (partition_key) ( PARTITION partition_name START(partition_value) END (partition_value) EVERY (interval_value) [, ... ] );
```

方式三：START(partition_value)方式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ] ) PARTITION BY RANGE (partition_key) ( INTERVAL ('interval_expr') PARTITION partition_name START(partition_value) [, ... ] );
```

方式四：END(partition_value | MAXVALUE)方式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ] ) PARTITION
BY RANGE (partition_key) INTERVAL ('interval_expr') ( PARTITION partition_name
END(partition_value | MAXVALUE) [, ... ] );
```

哈希分区表语法格式

```
CREATE TABLE partition_table_name ( [column_name data_type ] [, ... ] ) PARTITION
BY HASH (partition_key) (PARTITION partition_name ) [, ... ] );
```

参数说明

创建表参数

- UNLOGGED

如果指定此关键字，则创建的表为非日志表。在非日志表中写入的数据不会被写入到预写日志中，这样就会比普通表快很多。但是非日志表在冲突、执行操作系统重启、强制重启、切断电源操作或异常关机后会被自动截断，会造成数据丢失的风险。非日志表中的内容也不会被复制到备服务器中。在非日志表中创建的索引也不会被自动记录。

使用场景：非日志表不能保证数据的安全性，用户应该在确保数据已经做好备份的前提下使用，例如系统升级时进行数据的备份。

故障处理：当异常关机等操作导致非日志表上的索引发生数据丢失时，用户应该对发生错误的索引进行重建。

- GLOBAL | LOCAL

创建临时表时可以在 TEMP 或 TEMPORARY 前指定 GLOBAL 或 LOCAL 关键字。如果指定 GLOBAL 关键字，GBase 8s 会创建全局临时表，否则 GBase 8s 会创建本地临时表。

- TEMPORARY | TEMP

如果指定 TEMP 或 TEMPORARY 关键字，则创建的表为临时表。临时表分为全局临时表和本地临时表两种类型。创建临时表时如果指定 GLOBAL 关键字则为全局临时表，否则为本地临时表。

全局临时表的元数据对所有会话可见，会话结束后元数据继续存在。会话与会话之间的用户数据、索引和统计信息相互隔离，每个会话只能看到和更改自己提交的数据。全局临时表有两种模式：一种是基于会话级别的(ON COMMIT PRESERVE ROWS)，当会话结束时自动清空用户数据；一种是基于事务级别的(ON COMMIT DELETE ROWS)，当执行 commit 或 rollback 时自动清空用户数据。建表时如果没有指定 ON COMMIT 选项，则缺省为会话级别。与本地临时表不同，全局临时表建表时可以指定非 pg_temp_开头的 schema。

本地临时表只在当前会话可见，本会话结束后会自动删除。因此，在除当前会话连接的数据库节点故障时，仍然可以在当前会话上创建和使用临时表。由于临时表只在当前会话创建，对于涉及对临时表操作的 DDL 语句，会产生 DDL 失败的报错。因此，建议 DDL 语句中不要对临时表进行操作。TEMP 和 TEMPORARY 等价。

须知

- 本地临时表通过每个会话独立的以 pg_temp 开头的 schema 来保证只对当前会话可见，因此，不建议用户在日常操作中手动删除以 pg_temp、pg_toast_temp 开头的 schema。
- 如果建表时不指定 TEMPORARY/TEMP 关键字，而指定表的 schema 为当前会话的 pg_temp_开头的 schema，则此表会被创建为临时表。
- ALTER/DROP 全局临时表和索引，如果其它会话正在使用它，禁止操作（ALTER INDEX index_name REBUILD 除外）。
- 全局临时表的 DDL 只会影响当前会话的用户数据和索引。例如 truncate、reindex、analyze 只对当前会话有效。
- 全局临时表功能可以通过设置 GUC 参数 max_active_global_temporary_table 控制是否启用。如果 max_active_global_temporary_table=0，关闭全局临时表功能。
- 临时表只对当前会话可见，因此不支持与\parallel on 并行执行一起使用。
- \parallel on 临时表不支持主备切换。

- IF NOT EXISTS

  如果已经存在相同名称的表，不会报出错误，而会发出通知，告知通知此表已存在。

- table_name

  要创建的表名。

  须知

  物化视图的一些处理逻辑会通过表名的前缀，来识别判断是否为物化视图日志表和物化视图关联表。因此，用户不要自行创建以 mlog_或 matviewmap_为表名前缀的表，否则会影响此表的一些功能。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

- compress_mode

  表字段的压缩选项。该选项指定表字段优先使用的压缩算法。行存表不支持压缩。

  取值范围：DELTA、PREFIX、DICTIONARY、NUMSTR、NOCOMPRESS

- COLLATE collation

COLLATE 子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。排序规则可以使用“select * from pg_collation;”命令从 pg_collation 系统表中查询，默认的排序规则为查询结果中以 default 开始的行。

- LIKE source_table [like_option …]

LIKE 子句声明一个表，新表自动从这个表中继承所有字段名及其数据类型和非空约束。

新表与源表之间在创建动作完毕之后是完全无关的。在源表做的任何修改都不会传播到新表中，并且也不可能在扫描源表的时候包含新表的数据。

被复制的列和约束并不使用相同的名称进行融合。如果明确的指定了相同的名称或者在另外一个 LIKE 子句中，将会报错。

源表上的字段缺省表达式只有在指定 INCLUDING DEFAULTS 时，才会复制到新表中。缺省是不包含缺省表达式的，即新表中的所有字段的缺省值都是 NULL。

源表上的 CHECK 约束仅在指定 INCLUDING CONSTRAINTS 时，会复制到新表中，而其他类型的约束永远不会复制到新表中。非空约束总是复制到新表中。此规则同时适用于表约束和列约束。

如果指定了 INCLUDING INDEXES，则源表上的索引也将在新表上创建，默认不建立索引。

如果指定了 INCLUDING STORAGE，则复制列的 STORAGE 设置会复制到新表中，默认情况下不包含 STORAGE 设置。

如果指定了 INCLUDING COMMENTS，则源表列、约束和索引的注释会复制到新表中。默认情况下，不复制源表的注释。

如果指定了 INCLUDING PARTITION，则源表的分区定义会复制到新表中，同时新表将不能再使用 PARTITION BY 子句。默认情况下，不拷贝源表的分区定义。如果源表上带有索引，可以使用 INCLUDING PARTITION INCLUDING INDEXES 语法实现。如果对分区

表只使用 INCLUDING INDEXES，目标表定义将是普通表，但是索引是分区索引，最后结果会报错，因为普通表不支持分区索引。

如果指定了 INCLUDING RELOPTIONS，则源表的存储参数（即源表的 WITH 子句）会复制到新表中。默认情况下，不复制源表的存储参数。

INCLUDING ALL 包含了 INCLUDING DEFAULTS、INCLUDING CONSTRAINTS、INCLUDING INDEXES、INCLUDING STORAGE、INCLUDING COMMENTS、INCLUDING PARTITION 和 INCLUDING RELOPTIONS 的内容。

须知

如果源表包含 serial、bigserial、smallserial、largeserial 类型，或者源表字段的默认值是 sequence，且 sequence 属于源表（通过 CREATE SEQUENCE … OWNED BY 创建），这些 Sequence 不会关联到新表中，新表中会重新创建属于自己的 sequence。这和之前版本的处理逻辑不同。如果用户希望源表和新表共享 Sequence，需要首先创建一个共享的 Sequence（避免使用 OWNED BY），并配置为源表字段默认值，这样创建的新表会和源表共享该 Sequence。

不建议将其他表私有的 Sequence 配置为源表字段的默认值，尤其是其他表只分布在特定的 NodeGroup 上，这可能导致 CREATE TABLE … LIKE 执行失败。另外，如果源表配置其他表私有的 Sequence，当该表删除时 Sequence 也会连带删除，这样源表的 Sequence 将不可用。如果用户希望多个表共享 Sequence，建议创建共享的 Sequence。

对于分区表 EXCLUDING，需要配合 INCLUDING ALL 使用，如 INCLUDING ALL EXCLUDING DEFAULTS，除源分区表的 DEFAULTS，其它全包含。

- INHERITS (parent_table [, ...])

可选的 INHERITS 子句指定一个表的列表，新表将从其中自动地继承所有列。父表可以是普通表或者外部表。

INHERITS 的使用在新的子表和它的父表之间创建一种持久的关系。对于父表的模式修改通常也会传播到子表，并且默认情况下子表的数据会被包括在对父表的扫描中。

如果在多个父表中存在同名的列，除非父表中每一个这种列的数据类型都能匹配，否则会报告一个错误。如果没有冲突，那么重复列会被融合来形成新表中的一个单一列。如果新表中的列名列表包含一个也是继承而来的列名，该数据类型必须也匹配继承的列，并且列定义会被融合成一个。如果新表显式地为列指定了任何默认值， 这个默认值将覆盖来自该列继承声明中的默认值。 否则，任何父表都必须为该列指定相同的默认值，或者会报告一个错误。

CHECK 约束本质上也采用和列相同的方式被融合：如果多个父表或者新表定义中包含相同的命名 CHECK 约束，这些约束必须全部具有相同的检查表达式，否则将报告一个错误。具有相同名称和表达式的约束将被融合成一份拷贝。一个父表中的被标记为 NO INHERIT 的约束将不会被考虑。注意新表中一个未命名的 CHECK 约束将永远不会被融合， 因为那样总是会为它选择一个唯一的名字。

列的 STORAGE 设置也会从父表复制过来。

如果父表中的列是标识列，那么该属性不会被继承。如果需要，可以将子表中的列声明为标识列。

- [WITH (storage_parameter [= value] [, ...]) | WITH OIDS | WITHOUT OIDS]

  这个子句为表或索引指定一个可选的存储参数。

  WITH OIDS 子句被用来指定新表的行应该具有被分配的 OID（对象标识符）。

  说明

  使用任意精度类型 Numeric 定义列时，建议指定精度 p 以及刻度 s。在不指定精度和刻度时，会按输入的显示出来。

  参数的详细描述如下所示。

- FILLFACTOR

  一个表的填充因子（fillfactor）是一个介于 10 和 100 之间的百分数。100（完全填充）是默认值。如果指定了较小的填充因子，INSERT 操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得 UPDATE 有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为 100 是最佳选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。该参数对于列存表没有意义。

  取值范围：10~100

- ORIENTATION

  指定表数据的存储方式，即行存方式、列存方式、ORC 格式的方式，该参数设置成功后就不再支持修改。

  取值范围：

  ROW，表示表的数据将以行式存储。

  行存储适合于 OLTP 业务，适用于点查询或者增删操作较多的场景。

COLUMN，表示表的数据将以列式存储。

列存储适合于数据仓库业务，此类型的表上会做大量的汇聚计算，且涉及的列操作较少。

默认值：

若指定表空间为普通表空间，默认值为 ROW。

- STORAGE_TYPE

  指定存储引擎类型，该参数设置成功后就不再支持修改。

  取值范围：

  USTORE，表示表支持 Inplace-Update 存储引擎。

  ASTORE，表示表支持 Append-Only 存储引擎。

  默认值：

  不指定表时，默认是 Append-Only 存储。

- INIT_TD

  创建 Ustore 表时，指定初始化的 TD 个数，该参数只在创建 Ustore 表时才能设置生效。

  取值范围：2~128，默认值为 4。

- COMPRESSION

  指定表数据的压缩级别，它决定了表数据的压缩比以及压缩时间。一般来讲，压缩级别越高，压缩比也越大，压缩时间也越长；反之亦然。实际压缩比取决于加载的表数据的分布特征。行存表默认增加 COMPRESSION=NO 字段。

  取值范围：

  列存表的有效值为 YES/NO/LOW/MIDDLE/HIGH，默认值为 LOW。

- COMPRESSLEVEL

  指定表数据同一压缩级别下的不同压缩水平，它决定了同一压缩级别下表数据的压缩比以及压缩时间。对同一压缩级别进行了更加详细的划分，为用户选择压缩比和压缩时间提供了更多的空间。总体来讲，此值越大，表示同一压缩级别下压缩比越大，压缩时间越长；反之亦然。

  取值范围：0~3，默认值为 0。

- COMPRESSTYPE

行存表参数，设置行存表压缩算法。1 代表 pglz 算法，2 代表 zstd 算法，默认不压缩。（仅支持 ASTORE 下的普通表）

取值范围：0~2，默认值为 0。

- COMPRESS_LEVEL

行存表参数，设置行存表压缩算法等级，仅当 COMPRESSTYPE 为 2 时生效。压缩等级越高，表的压缩效果越好，表的访问速度越慢。（仅支持 ASTORE 下的普通表）

取值范围：-31~31，默认值为 0。

- COMPRESS_CHUNK_SIZE

行存表参数，设置行存表压缩 chunk 块大小。chunk 数据块越小，预期能达到的压缩效果越好，同时数据越离散，影响表的访问速度。（仅支持 ASTORE 下的普通表）

取值范围：与页面大小有关。在页面大小为 8k 场景，取值范围为：512、1024、2048、4096。

默认值：4096

- COMPRESS_PREALLOC_CHUNKS

行存表参数，设置行存表压缩 chunk 块预分配数量。预分配数量越大，表的压缩率相对越差，离散度越小，访问性能越好。（仅支持 ASTORE 下的普通表）

取值范围：0~7，默认值为 0。

当 COMPRESS_CHUNK_SIZE 为 512 和 1024 时，支持预分配设置最大为 7。

当 COMPRESS_CHUNK_SIZE 为 2048 时，支持预分配设置最大为 3。

当 COMPRESS_CHUNK_SIZE 为 4096 时，支持预分配设置最大为 1。

- COMPRESS_BYTE_CONVERT

行存表参数，设置行存表压缩字节转换预处理。在一些场景下可以提升压缩效果，同时会导致一定性能劣化。

取值范围：布尔值，默认关闭。

- COMPRESS_DIFF_CONVERT

行存表参数，设置行存表压缩字节差分预处理。只能与 compress_byte_convert 一起使用。在一些场景下可以提升压缩效果，同时会导致一定性能劣化。

取值范围：布尔值，默认关闭。

- MAX_BATCHROW

指定了在数据加载过程中一个存储单元可以容纳记录的最大数目。该参数只对列存表有效。

取值范围：10000~60000，默认 60000。

- PARTIAL_CLUSTER_ROWS

指定了在数据加载过程中进行将局部聚簇存储的记录数目。该参数只对列存表有效。

取值范围：大于等于 MAX_BATCHROW，建议取值为 MAX_BATCHROW 的整数倍。

- DELTAROW_THRESHOLD

指定列存表导入时小于多少行的数据进入 delta 表，只在 GUC 参数 enable_delta_store 开启时生效。该参数只对列存表有效。

取值范围：0￣9999，默认值为 100

- VERSION

指定 ORC 存储格式的版本。

取值范围：0.12，目前支持 ORC 0.12 格式，后续会随着 ORC 格式的发展，支持更多格式。

默认值：0.12

- segment

使用段页式的方式存储。本参数仅支持行存表。不支持列存表、临时表、unlog 表。不支持 ustore 存储引擎。

取值范围：on/off

默认值：off

- hasuids

参数开启：更新表元组时，为元组分配表级唯一标识 id。

取值范围：on/off。

默认值：off。

- hashbucket

创建 hash bucket 存储。本参数仅支持行存表和行存 range 表。

取值范围：on/off

默认值：off

- bucketcnt

创建 bucket 表时，指定表的 bucket 数目。该参数指定的值必须有和其对应的 Child Node Group 存在。

取值范围：32 ~ 16384,，且是 2 的整数次方。

默认值：16384。

- ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP }

ON COMMIT 选项决定在事务中执行创建临时表操作，当事务提交时，此临时表的后续操作。有以下三个选项，当前支持 PRESERVE ROWS 和 DELETE ROWS 选项。

PRESERVE ROWS（缺省值）：提交时不对临时表做任何操作，临时表及其表数据保持不变。

DELETE ROWS：提交时删除临时表中数据。

DROP：提交时删除此临时表。只支持本地临时表，不支持全局临时表。

- COMPRESS | NOCOMPRESS

创建新表时，需要在 CREATE TABLE 语句中指定关键字 COMPRESS，这样，当对该表进行批量插入时就会触发压缩特性。该特性会在页范围内扫描所有元组数据，生成字典、压缩元组数据并进行存储。指定关键字 NOCOMPRESS 则不对表进行压缩。行存表不支持压缩。

缺省值：NOCOMPRESS，即不对元组数据进行压缩。

- TABLESPACE tablespace_name

创建新表时指定此关键字，表示新表将要在指定表空间内创建。如果没有声明，将使用默认表空间。

- DISTRIBUTE BY

指定表如何在节点之间分布或者复制。

取值范围：

- REPLICATION：表的每一行存在所有数据节点（DN）中，即每个数据节点都有完整的表数据。
- HASH (column_name) ：对指定的列进行 Hash，通过映射，把数据分布到指定 DN。
- DISTRIBUTE BY：指定表如何在节点之间分布或者复制；

说明

- 当指定 DISTRIBUTE BY { HASH } (column_name)参数时，创建主键和唯一索引必须包含“ column_name”列。
- 当被参照表指定 DISTRIBUTE BY { HASH } (column_name)参数时，参照表的外键必须包含“ column_name”列。

默认值：HASH(column_name)，column_name 取表的主键列（如果有的话）或首个数据类型支持作为分布列的列。

column_name 的数据类型必须是以下类型之一：

- INTEGER TYPES：TINYINT，SMALLINT，INT，BIGINT，NUMERIC/DECIMAL
- CHARACTER TYPES：CHAR，BPCHAR，VARCHAR，VARCHAR2，NVARCHAR2，TEXT
- DATE/TIME TYPES：DATE，TIME，TIMETZ，TIMESTAMP，TIMESTAMPTZ，INTERVAL，SMALLDATETIME

说明

在建表时，选择分布列和分区键可对 SQL 查询性能产生重大影响。因此，需要根据一定策略选择合适的分布列和分区键。

- 选择合适的分布列

对于采用散列（Hash）方式的数据分布表，一个合适的分布列应将一个表内的数据，均匀分散存储在多个 DN 内，避免出现数据倾斜现象（即多个 DN 内数据分布不均）。请按照如下原则判定合适的分布列：

(1) 判断是否已发生数据倾斜现象。

连接数据库，执行如下语句，查看各 DN 内元组数目。命令中的斜体部分 tablename，

请填入待分析的表名。

```
postgres=# SELECT a.count,b.node_name FROM (SELECT count(*) AS count,xc_node_id
FROM tablename GROUP BY xc_node_id) a, pgxc_node b WHERE a.xc_node_id=b.node_id
ORDER BY a.count DESC;
```

如果各 DN 内元组数目相差较大（如相差数倍、数十倍），则表明已发生数据倾斜现象，请按照下面原则调整分布列。

(2) 重新选择分布列，重新建表。当前不支持通过 ALTER TABLE 语句调整分布列，因此调整分布列时需要重新建表。

选择原则如下：

分布列的列值应比较离散，以便数据能够均匀分布到各个 DN。例如，考虑选择表的主键为分布列，如在人员信息表中选择身份证号码为分布列。

在满足上面原则的情况下，考虑选择查询中的连接条件为分布列，以便 Join 任务能够下推到 DN 中执行，且减少 DN 之间的通信数据量。

- 选择合适的分区键

数据分区功能，可根据表的一列或者多列，将要插入表的记录分为若干个范围（这些范围在不同的分区里没有重叠）。然后为每个范围创建一个分区，用来存储相应的数据。

调整分区键，使每次查询结果尽可能存储在相同或者最少的分区内（称为“分区剪枝”），通过获取连续 I/O 大幅度提升查询性能。

实际业务中，经常将时间作为查询对象的过滤条件，因此，可考虑选择时间列为分区键，键值范围可根据总数据量、一次查询数据量调整。

- RANGE/LIST 分布

当没有为 RANGE/LIST 分布表的分片显示指定 DN 时，数据库内部为分片分配 DN 是采用 roundrobin 的算法。另外，在使用 RANGE/LIST 分布的场景中，考虑到后续扩容的需要，建议用户在建表时定义尽可能多的分片数，因为如果定义的分片数小于扩容前的 DN 节点数，数据重分布时则无法落入新的 DN 节点。需要特别注意的是，由于是由用户自行设计分片规则，在某些极端情况下，扩容也可能无法解决存储空间不足的问题。

- CONSTRAINT constraint_name

列约束或表约束的名称。可选的约束子句用于声明约束，新行或者更新的行必须满足这些约束才能成功插入或更新。

定义约束有两种方法：

列约束：作为一个列定义的一部分，仅影响该列。

表约束：不和某个列绑在一起，可以作用于多个列。

- NOT NULL

字段值不允许为 NULL。

- NULL

字段值允许为 NULL ，这是缺省值。

这个子句只是为和非标准 SQL 数据库兼容。不建议使用。

- CHECK (expression)

CHECK 约束声明一个布尔表达式，每次要插入的新行或者要更新的行的新值必须使表达式结果为真或未知才能成功，否则会抛出一个异常并且不会修改数据库。

声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以引用多个字段。

 说明：expression 表达式中，如果存在“<>NULL”或“！=NULL”，这种写法是无效的，需要写成“is NOT NULL”。

- DEFAULT default_expr

DEFAULT 子句给字段指定缺省值。该数值可以是任何不含变量的表达式(不允许使用子查询和对本表中的其他字段的交叉引用)。缺省表达式的数据类型必须和字段类型匹配。

缺省表达式将被用于任何未声明该字段数值的插入操作。如果没有指定缺省值则缺省值为 NULL 。

- GENERATED ALWAYS AS (generation_expr) STORED

该子句将字段创建为生成列，生成列的值在写入（插入或更新）数据时由 generation_expr 计算得到，STORED 表示像普通列一样存储生成列的值。

说明：- 生成表达式不能以任何方式引用当前行以外的其他数据。生成表达式不能引用其他生成列，不能引用系统列。生成表达式不能返回结果集，不能使用子查询，不能使用聚集函数，不能使用窗口函数。生成表达式调用的函数只能是不可变（IMMUTABLE）函数。

不能为生成列指定默认值。

生成列不能作为分区键的一部分。

生成列不能和 ON UPDATE 约束字句的 CASCADE,SET NULL,SET DEFAULT 动作同时指定。生成列不能和 ON DELETE 约束字句的 SET NULL、SET DEFAULT 动作同时指定。

修改和删除生成列的方法和普通列相同。删除生成列依赖的普通列，生成列被自动删除。不能改变生成列所依赖的列的类型。

生成列不能被直接写入。在 INSERT 或 UPDATE 命令中，不能为生成列指定值，但是可以指定关键字 DEFAULT。

生成列的权限控制和普通列一样。

列存表、内存表 MOT 不支持生成列。外表中仅 postgres_fdw 支持生成列。

- UNIQUE (column_name [, …]) index_parameters

  UNIQUE 约束表示表里的一个字段或多个字段的组合必须在全表范围内唯一。

  对于唯一约束，NULL 被认为是互不相等的。

- PRIMARY KEY (column_name [, …]) index_parameters

  主键约束声明表中的一个或者多个字段只能包含唯一的非 NULL 值。

  一个表只能声明一个主键。

- FOREIGN KEY (column_name [, …]) REFERENCES reftable [(refcolumn [, …])] [MATCH matchtype] [ON DELETE action] [ON UPDATE action] (table constraint)

  指定列（或一组列）中的值必须匹配另一个表的某一行中出现的值。通常一个表中的 FOREIGN KEY 指向另一个表中的 UNIQUE KEY（唯一约束的键），即维护了两个相关表之间的引用完整性。

  外键约束要求新表中一列或多列构成的组应该只包含、匹配被参考表中被参考字段值。若省略 refcolum，则将使用 reftable 的主键。被参考列应该是被参考表中的唯一字段或主键。外键约束不能被定义在临时表和永久表之间。

  参考字段与被参考字段之间存在三种类型匹配，分别是：

  MATCH FULL：不允许一个多字段外键的字段为 NULL，除非全部外键字段都是 NULL。

  MATCH SIMPLE（缺省）：允许任意外键字段为 NULL。

  MATCH PARTIAL：目前暂不支持。

  另外，当被参考表中的数据发生改变时，某些操作也会在新表对应字段的数据上执行。

ON DELETE 子句声明当被参考表中的被参考行被删除时要执行的操作。ON UPDATE 子句声明当被参考表中的被参考字段数据更新时要执行的操作。对于 ON DELETE 子句、ON UPDATE 子句的可能动作：

NO ACTION（缺省）：删除或更新时，创建一个表明违反外键约束的错误。若约束可推迟，且若仍存在任何引用行，那这个错误将会在检查约束的时候产生。

RESTRICT：删除或更新时，创建一个表明违反外键约束的错误。与 NO ACTION 相同，只是动作不可推迟。

CASCADE：删除新表中任何引用了被删除行的行，或更新新表中引用行的字段值为被参考字段的新值。

SET NULL：设置引用字段为 NULL。

SET DEFAULT：设置引用字段为它们的缺省值。

- DEFERRABLE | NOT DEFERRABLE

这两个关键字设置该约束是否可推迟。一个不可推迟的约束将在每条命令之后马上检查。可推迟约束可以推迟到事务结尾使用 SET CONSTRAINTS 命令检查。缺省是 NOT DEFERRABLE。目前，UNIQUE 约束、主键约束、外键约束可以接受这个子句。所有其他约束类型都是不可推迟的。

说明：Ustore 表不支持 DEFERRABLE 以及 INITIALLY DEFERRED 关键字。

- PARTIAL CLUSTER KEY

局部聚簇存储，列存表导入数据时按照指定的列(单列或多列)，进行局部排序。

- INITIALLY IMMEDIATE | INITIALLY DEFERRED

如果约束是可推迟的，则这个子句声明检查约束的缺省时间。

如果约束是 INITIALLY IMMEDIATE（缺省），则在每条语句执行之后就立即检查它；

如果约束是 INITIALLY DEFERRED ，则只有在事务结尾才检查它。

约束检查的时间可以用 SET CONSTRAINTS 命令修改。

USING INDEX TABLESPACE tablespace_name

为 UNIQUE 或 PRIMARY KEY 约束相关的索引声明一个表空间。如果没有提供这个子句，这个索引将在 default_tablespace 中创建，如果 default_tablespace 为空，将使用数据库的缺省表空间。

- ENCRYPTION_TYPE = encryption_type_value

  为 ENCRYPTED WITH 约束中的加密类型，encryption_type_value 的值为 [DETERMINISTIC | RANDOMIZED]

- TO GROUP | TO NODE

  定义表数据存储的节点/节点组；

- [COMMENT {=| } 'text']

  在 create table 语法中进行列定义时，可以通过 comment 关键字方式指定列的注释信息。

创建 VALUES LESS THAN 范围分区表参数说明

- partition_table_name

  分区表的名称。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

- partition_key

  partition_key 为分区键的名称。

  对于从句是 VALUE LESS THAN 的语法格式，范围分区策略的分区键最多支持 4 列。

- partition_name

  partition_name 为范围分区的名称。

- VALUES LESS THAN

  分区中的数值必须小于上边界值。

- partition_value

  partition_value 为范围分区的上边界，取值依赖于 partition_key 的类型。

- MAXVALUE

  MAXVALUE 表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

创建 START END 范围分区表参数说明

- partition_table_name

  分区表的名称。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

- partition_key

  partition_key 为分区键的名称。

  对于从句是 START END 的语法格式，范围分区策略的分区键仅支持 1 列。

- partition_name

  partition_name 为范围分区的名称或者范围分区的名称前缀。

  若该定义是“START(partition_value) END (partition_value) EVERY (interval_value)”从句，假定其中的 partition_name 是 p1，则分区的名称依次为 p1_1, p1_2, …。

  例如对于定义“PARTITION p1 START(1) END(4) EVERY(1)”，则生成的分区是：[1, 2), [2, 3) 和 [3, 4)，名称依次为 p1_1, p1_2 和 p1_3，即此处的 p1 是名称前缀。

  若该定义是第一个分区定义，且该定义有 START 值，则范围（MINVALUE, START）将自动作为第一个实际分区，其名称为 p1_0，然后该定义语义描述的分区名称依次为 p1_1, p1_2, …。

  例如对于完整定义“PARTITION p1 START(1), PARTITION p2 START(2)”，生成的分区是：(MINVALUE, 1), [1, 2) 和 [2, MAXVALUE)，其名称依次为 p1_0, p1_1 和 p2，即此处 p1 是名称前缀，p2 是分区名称。这里 MINVALUE 表示最小值。

  其余的情况都是范围分区名称。

- VALUES LESS THAN

  分区中的数值必须小于上边界值。

- partition_value

  partition_value 为范围分区的端点值（起始或终点），取值依赖于 partition_key 的类型。

- interval_value

对[START，END）表示的范围进行切分，interval_value 是指定切分后每个分区的宽度。如果（END-START）值不能整除以 EVERY 值，则仅最后一个分区的宽度小于 EVERY 值。

- MAXVALUE

MAXVALUE 表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

创建列表分区表参数说明

- partition_table_name

分区表的名称。

- column_name

新表中要创建的字段名。

- data_type

字段的数据类型。

- partition_key

partition_key 为分区键的名称。

列表分区策略的分区键仅支持 1 列。

- partition_name

partition_name 为范围分区的名称。

- list_values_clause

对应分区存在的一个或者多个键值。多个键值之间以逗号分隔。

- VALUES (DEFAULT)

加入的数据如有“list_values_clause”中未列出的键值，存放在 VALUES (DEFAULT)对应的分区。

- MAXVALUE

MAXVALUE 表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

间隔分区表参数说明

- INTERVAL ('interval_expr')

间隔分区定义信息。只支持 TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE 数据类型。

interval_expr 自动创建分区的间隔，例如：

自动创建分区的间隔，例如：1 day、1 month。

- partition_name

  partition_name 为范围分区的名称。

  系统自动建立的分区按照建立的先后顺序，依次命名为：sys_p1、sys_p2、sys_p3…

哈希分区表参数说明

- partition_table_name

  分区表的名称。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

- partition_key

  partition_key 为分区键的名称。哈希分区策略的分区键仅支持 1 列。

- partition_name

  partition_name 为哈希分区的名称。希望创建几个哈希分区就给出几个分区名。

示例

```
--创建简单的表。
postgres=# CREATE TABLE tpcds.warehouse_t1
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
```

```
    W_CITY                    VARCHAR(60)                  ,
    W_COUNTY                  VARCHAR(30)                  ,
    W_STATE                   CHAR(2)                      ,
    W_ZIP                     CHAR(10)                     ,
    W_COUNTRY                 VARCHAR(20)                  ,
    W_GMT_OFFSET              DECIMAL(5,2)
);

postgres=# CREATE TABLE tpcds.warehouse_t2
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                  ,
    W_WAREHOUSE_SQ_FT         INTEGER                      ,
    W_STREET_NUMBER           CHAR(10)                     ,
    W_STREET_NAME             VARCHAR(60),
    W_STREET_TYPE             CHAR(15)                     ,
    W_SUITE_NUMBER            CHAR(10)                     ,
    W_CITY                    VARCHAR(60)                  ,
    W_COUNTY                  VARCHAR(30)                  ,
    W_STATE                   CHAR(2)                      ,
    W_ZIP                     CHAR(10)                     ,
    W_COUNTRY                 VARCHAR(20)                  ,
    W_GMT_OFFSET              DECIMAL(5,2)
);
--创建表，并指定 W_STATE 字段的缺省值为 GA。
postgres=# CREATE TABLE tpcds.warehouse_t3
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                  ,
    W_WAREHOUSE_SQ_FT         INTEGER                      ,
    W_STREET_NUMBER           CHAR(10)                     ,
    W_STREET_NAME             VARCHAR(60)                  ,
    W_STREET_TYPE             CHAR(15)                     ,
    W_SUITE_NUMBER            CHAR(10)                     ,
    W_CITY                    VARCHAR(60)                  ,
    W_COUNTY                  VARCHAR(30)                  ,
    W_STATE                   CHAR(2)             DEFAULT 'GA',
    W_ZIP                     CHAR(10)                     ,
    W_COUNTRY                 VARCHAR(20)                  ,
    W_GMT_OFFSET              DECIMAL(5,2)
```

```
);

--创建表，并在事务结束时检查 W_WAREHOUSE_NAME 字段是否有重复。
postgres=# CREATE TABLE tpcds.warehouse_t4
(
    W_WAREHOUSE_SK            INTEGER                NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)   UNIQUE DEFERRABLE,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)
);
--创建一个带有 70%填充因子的表。
postgres=# CREATE TABLE tpcds.warehouse_t5
(
    W_WAREHOUSE_SK            INTEGER                NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2),
    UNIQUE(W_WAREHOUSE_NAME) WITH(fillfactor=70)
);

--或者用下面的语法。
postgres=# CREATE TABLE tpcds.warehouse_t6
```

```
(
    W_WAREHOUSE_SK            INTEGER                NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)              UNIQUE,
    W_WAREHOUSE_SQ_FT         INTEGER                         ,
    W_STREET_NUMBER           CHAR(10)                        ,
    W_STREET_NAME             VARCHAR(60)                     ,
    W_STREET_TYPE             CHAR(15)                        ,
    W_SUITE_NUMBER            CHAR(10)                        ,
    W_CITY                    VARCHAR(60)                     ,
    W_COUNTY                  VARCHAR(30)                     ,
    W_STATE                   CHAR(2)                         ,
    W_ZIP                     CHAR(10)                        ,
    W_COUNTRY                 VARCHAR(20)                     ,
    W_GMT_OFFSET              DECIMAL(5,2)
) WITH(fillfactor=70);

--创建表，并指定该表数据不写入预写日志。
postgres=# CREATE UNLOGGED TABLE tpcds.warehouse_t7
(
    W_WAREHOUSE_SK            INTEGER                NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                     ,
    W_WAREHOUSE_SQ_FT         INTEGER                         ,
    W_STREET_NUMBER           CHAR(10)                        ,
    W_STREET_NAME             VARCHAR(60)                     ,
    W_STREET_TYPE             CHAR(15)                        ,
    W_SUITE_NUMBER            CHAR(10)                        ,
    W_CITY                    VARCHAR(60)                     ,
    W_COUNTY                  VARCHAR(30)                     ,
    W_STATE                   CHAR(2)                         ,
    W_ZIP                     CHAR(10)                        ,
    W_COUNTRY                 VARCHAR(20)                     ,
    W_GMT_OFFSET              DECIMAL(5,2)
);

--创建表临时表。
postgres=# CREATE TEMPORARY TABLE warehouse_t24
(
    W_WAREHOUSE_SK            INTEGER                NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                     ,
```

```
    W_WAREHOUSE_SQ_FT         INTEGER                    ,
    W_STREET_NUMBER           CHAR(10)                   ,
    W_STREET_NAME             VARCHAR(60)                ,
    W_STREET_TYPE             CHAR(15)                   ,
    W_SUITE_NUMBER            CHAR(10)                   ,
    W_CITY                    VARCHAR(60)                ,
    W_COUNTY                  VARCHAR(30)                ,
    W_STATE                   CHAR(2)                    ,
    W_ZIP                     CHAR(10)                   ,
    W_COUNTRY                 VARCHAR(20)                ,
    W_GMT_OFFSET              DECIMAL(5,2)
);

--创建本地临时表，并指定提交事务时删除该临时表数据。
postgres=# CREATE TEMPORARY TABLE warehouse_t25
(
    W_WAREHOUSE_SK            INTEGER                NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                    ,
    W_WAREHOUSE_SQ_FT         INTEGER                    ,
    W_STREET_NUMBER           CHAR(10)                   ,
    W_STREET_NAME             VARCHAR(60)                ,
    W_STREET_TYPE             CHAR(15)                   ,
    W_SUITE_NUMBER            CHAR(10)                   ,
    W_CITY                    VARCHAR(60)                ,
    W_COUNTY                  VARCHAR(30)                ,
    W_STATE                   CHAR(2)                    ,
    W_ZIP                     CHAR(10)                   ,
    W_COUNTRY                 VARCHAR(20)                ,
    W_GMT_OFFSET              DECIMAL(5,2)
) ON COMMIT DELETE ROWS;

--创建全局临时表，并指定会话结束时删除该临时表数据。
postgres=# CREATE GLOBAL TEMPORARY TABLE gtt1
(
    ID                        INTEGER                NOT NULL,
    NAME                      CHAR(16)               NOT NULL,
    ADDRESS                   VARCHAR(50)                    ,
    POSTCODE                  CHAR(6)
) ON COMMIT PRESERVE ROWS;

--创建表时，不希望因为表已存在而报错。
```

```
postgres=# CREATE TABLE IF NOT EXISTS tpcds.warehouse_t8
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)
);

--创建普通表空间。
postgres=# CREATE TABLESPACE DS_TABLESPACE1 RELATIVE LOCATION
'tablespace/tablespace_1';
--创建表时，指定表空间。
postgres=# CREATE TABLE tpcds.warehouse_t9
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)
) TABLESPACE DS_TABLESPACE1;

--创建表时，单独指定 W_WAREHOUSE_NAME 的索引表空间。
postgres=# CREATE TABLE tpcds.warehouse_t10
```

```
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)           UNIQUE USING INDEX
TABLESPACE DS_TABLESPACE1,
    W_WAREHOUSE_SQ_FT         INTEGER                      ,
    W_STREET_NUMBER           CHAR(10)                     ,
    W_STREET_NAME             VARCHAR(60)                  ,
    W_STREET_TYPE             CHAR(15)                     ,
    W_SUITE_NUMBER            CHAR(10)                     ,
    W_CITY                    VARCHAR(60)                  ,
    W_COUNTY                  VARCHAR(30)                  ,
    W_STATE                   CHAR(2)                      ,
    W_ZIP                     CHAR(10)                     ,
    W_COUNTRY                 VARCHAR(20)                  ,
    W_GMT_OFFSET              DECIMAL(5,2)
);
--创建一个有主键约束的表。
postgres=# CREATE TABLE tpcds.warehouse_t11
(
    W_WAREHOUSE_SK            INTEGER            PRIMARY KEY,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                  ,
    W_WAREHOUSE_SQ_FT         INTEGER                      ,
    W_STREET_NUMBER           CHAR(10)                     ,
    W_STREET_NAME             VARCHAR(60)                  ,
    W_STREET_TYPE             CHAR(15)                     ,
    W_SUITE_NUMBER            CHAR(10)                     ,
    W_CITY                    VARCHAR(60)                  ,
    W_COUNTY                  VARCHAR(30)                  ,
    W_STATE                   CHAR(2)                      ,
    W_ZIP                     CHAR(10)                     ,
    W_COUNTRY                 VARCHAR(20)                  ,
    W_GMT_OFFSET              DECIMAL(5,2)
);

---或是用下面的语法，效果完全一样。
postgres=# CREATE TABLE tpcds.warehouse_t12
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                  ,
```

```
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2),
    PRIMARY KEY(W_WAREHOUSE_SK)
);

--或是用下面的语法，指定约束的名称。
postgres=# CREATE TABLE tpcds.warehouse_t13
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2),
    CONSTRAINT W_CSTR_KEY1 PRIMARY KEY(W_WAREHOUSE_SK)
);

--创建一个有复合主键约束的表。
postgres=# CREATE TABLE tpcds.warehouse_t14
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
```

```
    W_STREET_NAME             VARCHAR(60)                    ,
    W_STREET_TYPE             CHAR(15)                       ,
    W_SUITE_NUMBER            CHAR(10)                       ,
    W_CITY                    VARCHAR(60)                    ,
    W_COUNTY                  VARCHAR(30)                    ,
    W_STATE                   CHAR(2)                        ,
    W_ZIP                     CHAR(10)                       ,
    W_COUNTRY                 VARCHAR(20)                    ,
    W_GMT_OFFSET              DECIMAL(5,2),
    CONSTRAINT W_CSTR_KEY2 PRIMARY KEY(W_WAREHOUSE_SK, W_WAREHOUSE_ID)
);
--创建列存表。
postgres=# CREATE TABLE tpcds.warehouse_t15
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                    ,
    W_WAREHOUSE_SQ_FT         INTEGER                        ,
    W_STREET_NUMBER           CHAR(10)                       ,
    W_STREET_NAME             VARCHAR(60)                    ,
    W_STREET_TYPE             CHAR(15)                       ,
    W_SUITE_NUMBER            CHAR(10)                       ,
    W_CITY                    VARCHAR(60)                    ,
    W_COUNTY                  VARCHAR(30)                    ,
    W_STATE                   CHAR(2)                        ,
    W_ZIP                     CHAR(10)                       ,
    W_COUNTRY                 VARCHAR(20)                    ,
    W_GMT_OFFSET              DECIMAL(5,2)
) WITH (ORIENTATION = COLUMN);

--创建局部聚簇存储的列存表。
postgres=# CREATE TABLE tpcds.warehouse_t16
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                    ,
    W_WAREHOUSE_SQ_FT         INTEGER                        ,
    W_STREET_NUMBER           CHAR(10)                       ,
    W_STREET_NAME             VARCHAR(60)                    ,
    W_STREET_TYPE             CHAR(15)                       ,
    W_SUITE_NUMBER            CHAR(10)                       ,
    W_CITY                    VARCHAR(60)                    ,
```

```
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2),
    PARTIAL CLUSTER KEY(W_WAREHOUSE_SK, W_WAREHOUSE_ID)
) WITH (ORIENTATION = COLUMN);

--定义一个带压缩的列存表。
postgres=# CREATE TABLE tpcds.warehouse_t17
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)
) WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH);


--定义一个检查列约束。
postgres=# CREATE TABLE tpcds.warehouse_t19
(
    W_WAREHOUSE_SK            INTEGER               PRIMARY KEY CHECK
(W_WAREHOUSE_SK > 0),
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)           CHECK (W_WAREHOUSE_NAME IS
NOT NULL),
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
```

```
    W_COUNTY                    VARCHAR(30)                     ,
    W_STATE                     CHAR(2)                         ,
    W_ZIP                       CHAR(10)                        ,
    W_COUNTRY                   VARCHAR(20)                     ,
    W_GMT_OFFSET                DECIMAL(5,2)
);

postgres=# CREATE TABLE tpcds.warehouse_t20
(
    W_WAREHOUSE_SK              INTEGER                 PRIMARY KEY,
    W_WAREHOUSE_ID              CHAR(16)                NOT NULL,
    W_WAREHOUSE_NAME            VARCHAR(20)             CHECK (W_WAREHOUSE_NAME IS
NOT NULL),
    W_WAREHOUSE_SQ_FT           INTEGER                         ,
    W_STREET_NUMBER             CHAR(10)                        ,
    W_STREET_NAME               VARCHAR(60)                     ,
    W_STREET_TYPE               CHAR(15)                        ,
    W_SUITE_NUMBER              CHAR(10)                        ,
    W_CITY                      VARCHAR(60)                     ,
    W_COUNTY                    VARCHAR(30)                     ,
    W_STATE                     CHAR(2)                         ,
    W_ZIP                       CHAR(10)                        ,
    W_COUNTRY                   VARCHAR(20)                     ,
    W_GMT_OFFSET                DECIMAL(5,2),
    CONSTRAINT W_CONSTR_KEY2 CHECK(W_WAREHOUSE_SK > 0 AND W_WAREHOUSE_NAME IS NOT
NULL)
);

--创建一个有外键约束的表。
postgres=# CREATE TABLE tpcds.city_t23
(
    W_CITY              VARCHAR(60)                 PRIMARY KEY,
    W_ADDRESS       TEXT
);
postgres=# CREATE TABLE tpcds.warehouse_t23
(
    W_WAREHOUSE_SK              INTEGER                 NOT NULL,
    W_WAREHOUSE_ID              CHAR(16)                NOT NULL,
    W_WAREHOUSE_NAME            VARCHAR(20)                     ,
    W_WAREHOUSE_SQ_FT           INTEGER                         ,
    W_STREET_NUMBER             CHAR(10)                        ,
    W_STREET_NAME               VARCHAR(60)                     ,
```

```
    W_STREET_TYPE             CHAR(15)                    ,
    W_SUITE_NUMBER            CHAR(10)                    ,
    W_CITY                    VARCHAR(60)           REFERENCES
tpcds.city_t23(W_CITY),
    W_COUNTY                  VARCHAR(30)                 ,
    W_STATE                   CHAR(2)                     ,
    W_ZIP                     CHAR(10)                    ,
    W_COUNTRY                 VARCHAR(20)                 ,
    W_GMT_OFFSET              DECIMAL(5,2)
);
--创建有外键约束的分布表。
--创建普通表 t0
postgres=# CREATE TABLE t0(id INT, str TEXT, PRIMARY KEY(id));
  --创建有外键约束的分布表 ft0
postgres=# CREATE TABLE ft0
(
id1         INT,
 id2        INT,
 str        TEXT,
 PRIMARY KEY(id1,id2),
 FOREIGN KEY(id2) REFERENCE t0(id) DISTRIBUTE BY HASH(id2)
);

--或是用下面的语法，效果完全一样。
postgres=# CREATE TABLE tpcds.warehouse_t23
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                 ,
    W_WAREHOUSE_SQ_FT         INTEGER                     ,
    W_STREET_NUMBER           CHAR(10)                    ,
    W_STREET_NAME             VARCHAR(60)                 ,
    W_STREET_TYPE             CHAR(15)                    ,
    W_SUITE_NUMBER            CHAR(10)                    ,
    W_CITY                    VARCHAR(60)                 ,
    W_COUNTY                  VARCHAR(30)                 ,
    W_STATE                   CHAR(2)                     ,
    W_ZIP                     CHAR(10)                    ,
    W_COUNTRY                 VARCHAR(20)                 ,
    W_GMT_OFFSET              DECIMAL(5,2)                ,
    FOREIGN KEY(W_CITY) REFERENCES tpcds.city_t23(W_CITY)
);
```

```
--或是用下面的语法，指定约束的名称。
postgres=# CREATE TABLE tpcds.warehouse_t23
(
    W_WAREHOUSE_SK            INTEGER                NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                    ,
    W_WAREHOUSE_SQ_FT         INTEGER                        ,
    W_STREET_NUMBER           CHAR(10)                       ,
    W_STREET_NAME             VARCHAR(60)                    ,
    W_STREET_TYPE             CHAR(15)                       ,
    W_SUITE_NUMBER            CHAR(10)                       ,
    W_CITY                    VARCHAR(60)                    ,
    W_COUNTY                  VARCHAR(30)                    ,
    W_STATE                   CHAR(2)                        ,
    W_ZIP                     CHAR(10)                       ,
    W_COUNTRY                 VARCHAR(20)                    ,
    W_GMT_OFFSET              DECIMAL(5,2)                   ,
    CONSTRAINT W_FORE_KEY1 FOREIGN KEY(W_CITY) REFERENCES
tpcds.city_t23(W_CITY)
);

--向 tpcds.warehouse_t19 表中增加一个 varchar 列。
postgres=# ALTER TABLE tpcds.warehouse_t19 ADD W_GOODS_CATEGORY varchar(30);

--给 tpcds.warehouse_t19 表增加一个检查约束。
postgres=# ALTER TABLE tpcds.warehouse_t19 ADD CONSTRAINT W_CONSTR_KEY4 CHECK
(W_STATE IS NOT NULL);

--在一个操作中改变两个现存字段的类型。
postgres=# ALTER TABLE tpcds.warehouse_t19
    ALTER COLUMN W_GOODS_CATEGORY TYPE varchar(80),
    ALTER COLUMN W_STREET_NAME TYPE varchar(100);

--此语句与上面语句等效。
postgres=# ALTER TABLE tpcds.warehouse_t19 MODIFY (W_GOODS_CATEGORY varchar(30),
W_STREET_NAME varchar(60));

--给一个已存在字段添加非空约束。
postgres=# ALTER TABLE tpcds.warehouse_t19 ALTER COLUMN W_GOODS_CATEGORY SET NOT
NULL;
```

```
--移除已存在字段的非空约束。
postgres=# ALTER TABLE tpcds.warehouse_t19 ALTER COLUMN W_GOODS_CATEGORY DROP NOT
NULL;

--如果列存表中还未指定局部聚簇，向在一个列存表中添加局部聚簇列。
postgres=# ALTER TABLE tpcds.warehouse_t17 ADD PARTIAL CLUSTER
KEY(W_WAREHOUSE_SK);

--查看约束的名称，并删除一个列存表中的局部聚簇列。
postgres=# \d+ tpcds.warehouse_t17
                              Table "tpcds.warehouse_t17"
      Column       |         Type          | Modifiers | Storage  | Stats
target | Description
-------------------+-----------------------+-----------+----------+----------
----+-------------
 w_warehouse_sk    | integer               | not null  | plain    |
|
 w_warehouse_id    | character(16)         | not null  | extended |
|
 w_warehouse_name  | character varying(20) |           | extended |
|
 w_warehouse_sq_ft | integer               |           | plain    |
|
 w_street_number   | character(10)         |           | extended |
|
 w_street_name     | character varying(60) |           | extended |
|
 w_street_type     | character(15)         |           | extended |
|
 w_suite_number    | character(10)         |           | extended |
|
 w_city            | character varying(60) |           | extended |
|
 w_county          | character varying(30) |           | extended |
|
 w_state           | character(2)          |           | extended |
|
 w_zip             | character(10)         |           | extended |
|
 w_country         | character varying(20) |           | extended |
|
```

```
 w_gmt_offset       | numeric(5,2)          |            | main     |
|
Partial Cluster :
    "warehouse_t17_cluster" PARTIAL CLUSTER KEY (w_warehouse_sk)
Has OIDs: no
Location Nodes: ALL DATANODES
Options: compression=no, version=0.12
postgres=# ALTER TABLE tpcds.warehouse_t17 DROP CONSTRAINT
warehouse_t17_cluster;

--将表移动到另一个表空间。
postgres=# ALTER TABLE tpcds.warehouse_t19 SET TABLESPACE PG_DEFAULT;
--创建模式 joe。
postgres=# CREATE SCHEMA joe;

--将表移动到另一个模式中。
postgres=# ALTER TABLE tpcds.warehouse_t19 SET SCHEMA joe;

--重命名已存在的表。
postgres=# ALTER TABLE joe.warehouse_t19 RENAME TO warehouse_t23;

--从 warehouse_t23 表中删除一个字段。
postgres=# ALTER TABLE joe.warehouse_t23 DROP COLUMN W_STREET_NAME;

--删除表空间、模式 joe 和模式表 warehouse。
postgres=# DROP TABLE tpcds.warehouse_t1;
postgres=# DROP TABLE tpcds.warehouse_t2;
postgres=# DROP TABLE tpcds.warehouse_t3;
postgres=# DROP TABLE tpcds.warehouse_t4;
postgres=# DROP TABLE tpcds.warehouse_t5;
postgres=# DROP TABLE tpcds.warehouse_t6;
postgres=# DROP TABLE tpcds.warehouse_t7;
postgres=# DROP TABLE tpcds.warehouse_t8;
postgres=# DROP TABLE tpcds.warehouse_t9;
postgres=# DROP TABLE tpcds.warehouse_t10;
postgres=# DROP TABLE tpcds.warehouse_t11;
postgres=# DROP TABLE tpcds.warehouse_t12;
postgres=# DROP TABLE tpcds.warehouse_t13;
postgres=# DROP TABLE tpcds.warehouse_t14;
postgres=# DROP TABLE tpcds.warehouse_t15;
postgres=# DROP TABLE tpcds.warehouse_t16;
postgres=# DROP TABLE tpcds.warehouse_t17;
```

```
postgres=# DROP TABLE tpcds.warehouse_t18;
postgres=# DROP TABLE tpcds.warehouse_t20;
postgres=# DROP TABLE tpcds.warehouse_t21;
postgres=# DROP TABLE tpcds.warehouse_t22;
postgres=# DROP TABLE joe.warehouse_t23;
postgres=# DROP TABLE tpcds.warehouse_t24;
postgres=# DROP TABLE tpcds.warehouse_t25;
postgres=# DROP TABLESPACE DS_TABLESPACE1;
postgres=# DROP SCHEMA IF EXISTS joe CASCADE;
```

相关命令

ALTER TABLE，DROP TABLE，CREATE TABLESPACE

优化建议

UNLOGGED

UNLOGGED 表和表上的索引因为数据写入时不通过 WAL 日志机制，写入速度远高于普通表。因此，可以用于缓冲存储复杂查询的中间结果集，增强复杂查询的性能。

UNLOGGED 表无主备机制，在系统故障或异常断点等情况下，会有数据丢失风险，因此，不可用来存储基础数据。

TEMPORARY | TEMP

临时表只在当前会话可见，会话结束后会自动删除。

LIKE

新表自动从这个表中继承所有字段名及其数据类型和非空约束，新表与源表之间在创建动作完毕之后是完全无关的。

LIKE INCLUDING DEFAULTS

源表上的字段缺省表达式只有在指定 INCLUDING DEFAULTS 时，才会复制到新表中。缺省是不包含缺省表达式的，即新表中的所有字段的缺省值都是 NULL。

LIKE INCLUDING CONSTRAINTS

源表上的 CHECK 约束仅在指定 INCLUDING CONSTRAINTS 时，会复制到新表中，而其他类型的约束永远不会复制到新表中。非空约束总是复制到新表中。此规则同时适用于表约束和列约束。

LIKE INCLUDING INDEXES

如果指定了 INCLUDING INDEXES，则源表上的索引也将在新表上创建，默认不建立索引。

LIKE INCLUDING STORAGE

如果指定了 INCLUDING STORAGE，则复制列的 STORAGE 设置会复制到新表中，默认情况下不包含 STORAGE 设置。

LIKE INCLUDING COMMENTS

如果指定了 INCLUDING COMMENTS，则源表列、约束和索引的注释会复制到新表中。默认情况下，不复制源表的注释。

LIKE INCLUDING PARTITION

如果指定了 INCLUDING PARTITION，则源表的分区定义会复制到新表中，同时新表将不能再使用 PARTITION BY 子句。默认情况下，不拷贝源表的分区定义。

列表/哈希分区表暂不支持 LIKE INCLUDING PARTITION。

LIKE INCLUDING RELOPTIONS

如果指定了 INCLUDING RELOPTIONS，则源表的存储参数（即源表的 WITH 子句）会复制到新表中。默认情况下，不复制源表的存储参数。

LIKE INCLUDING ALL

INCLUDING ALL 包含了 INCLUDING DEFAULTS、INCLUDING CONSTRAINTS、INCLUDING INDEXES、INCLUDING STORAGE、INCLUDING COMMENTS、INCLUDING PARTITION、INCLUDING RELOPTIONS 的内容。

ORIENTATION ROW

创建行存表，行存储适合于 OLTP 业务，此类型的表上交互事务比较多，一次交互会涉及表中的多个列，用行存查询效率较高。

ORIENTATION COLUMN

创建列存表，列存储适合于数据仓库业务，此类型的表上会做大量的汇聚计算，且涉及的列操作较少。

3.8.92 CREATE TABLE AS

功能描述

根据查询结果创建表。

CREATE TABLE AS 创建一个表并且用来自 SELECT 命令的结果填充该表。该表的字段和 SELECT 输出字段的名称及数据类型相关。不过用户可以通过明确地给出一个字段名称列表来覆盖 SELECT 输出字段的名称。

CREATE TABLE AS 对源表进行一次查询，然后将数据写入新表中，而查询视图结果会根据源表的变化而有所改变。相比之下，每次做查询的时候，视图都重新计算定义它的 SELECT 语句。

注意事项

分区表不能采用此方式进行创建。

如果在建表过程中数据库系统发生故障，系统恢复后可能无法自动清除之前已创建的、大小非 0 的磁盘文件。此种情况出现概率小，不影响数据库系统的正常运行。

语法格式

```
CREATE [ UNLOGGED ] TABLE table_name
    [ (column_name [, ...] ) ]
    [ WITH ( {storage_parameter = value} [, ... ] ) ]
    [ COMPRESS | NOCOMPRESS ]
    [ TABLESPACE tablespace_name ]
    [ DISTRIBUTE BY { REPLICATION | { [HASH ] ( column_name ) } } ]
    [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
    AS query
    [ WITH [ NO ] DATA ];
```

参数说明

- UNLOGGED

指定表为非日志表。在非日志表中写入的数据不会被写入到预写日志中，这样就会比普通表快很多。但是，它也是不安全的，非日志表在冲突或异常关机后会被自动删截。非日志表中的内容也不会被复制到备用服务器中。在该类表中创建的索引也不会被自动记录。

使用场景：非日志表不能保证数据的安全性，用户应该在确保数据已经做好备份的前提下使用，例如系统升级时进行数据的备份。

故障处理：当异常关机等操作导致非日志表上的索引发生数据丢失时，用户应该对发生错误的索引进行重建。

- table_name

  要创建的表名。

  取值范围：字符串，要符合标识符的命名规范。

- column_name

  新表中要创建的字段名。

  取值范围：字符串，要符合标识符的命名规范。

- WITH (storage_parameter [= value] [, …])

  这个子句为表或索引指定一个可选的存储参数。参数的详细说明如下所示。

- FILLFACTOR

  一个表的填充因子（fillfactor）是一个介于 10 和 100 之间的百分数。100（完全填充）是默认值。如果指定了较小的填充因子，INSERT 操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得 UPDATE 有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为 100 是最佳选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。该参数只对行存表有效。

  取值范围：10~100

- ORIENTATION

  取值范围：

  COLUMN：表的数据将以列式存储。

  ROW（缺省值）：表的数据将以行式存储。

- COMPRESSION

  指定表数据的压缩级别，它决定了表数据的压缩比以及压缩时间。一般来讲，压缩级别越高，压缩比也越大，压缩时间也越长；反之亦然。实际压缩比取决于加载的表数据的分布特征。

  取值范围：

  列存表的有效值为 YES/NO/LOW/MIDDLE/HIGH，默认值为 LOW。

  行存表不支持压缩。

- MAX_BATCHROW

指定了在数据加载过程中一个存储单元可以容纳记录的最大数目。该参数只对列存表有效。

取值范围：10000~60000

ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP }

ON COMMIT 选项决定在事务中执行创建临时表操作，当事务提交时，此临时表的后续操作。有以下三个选项，当前仅支持 PRESERVE ROWS 和 DELETE ROWS 选项。

PRESERVE ROWS（缺省值）：提交时不对临时表执行任何操作，临时表及其表数据保持不变。

DELETE ROWS：提交时删除临时表中数据。

DROP：提交时删除此临时表。只支持删除本地临时表，不支持删除全局临时表。

- COMPRESS / NOCOMPRESS

创建一个新表时，需要在创建表语句中指定关键字 COMPRESS，这样，当对该表进行批量插入时就会触发压缩特性。该特性会在页范围内扫描所有元组数据，生成字典、压缩元组数据并进行存储。指定关键字 NOCOMPRESS 则不对表进行压缩。行存表不支持压缩。

缺省值：NOCOMPRESS，即不对元组数据进行压缩。

- TABLESPACE tablespace_name

指定新表将要在 tablespace_name 表空间内创建。如果没有声明，将使用默认表空间。

- AS query

一个 SELECT VALUES 命令或者一个运行预备好的 SELECT 或 VALUES 查询的 EXECUTE 命令。

- [WITH [NO] DATA]

创建表时，是否也插入查询到的数据。默认是要数据，选择“NO”参数时，则不要数据。

示例

```
--创建一个表 tpcds.store_returns 表。
postgres=# CREATE TABLE tpcds.store_returns
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
```

```
    sr_item_sk                VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER
);
--创建一个表 tpcds.store_returns_t1 并插入 tpcds.store_returns 表中 sr_item_sk
字段中大于 16 的数值。
postgres=# CREATE TABLE tpcds.store_returns_t1 AS SELECT * FROM
tpcds.store_returns WHERE sr_item_sk > '4795';

--使用 tpcds.store_returns 拷贝一个新表 tpcds.store_returns_t2。
postgres=# CREATE TABLE tpcds.store_returns_t2 AS table tpcds.store_returns;

--删除表。
postgres=# DROP TABLE tpcds.store_returns_t1 ;
postgres=# DROP TABLE tpcds.store_returns_t2 ;
postgres=# DROP TABLE tpcds.store_returns;
```

相关命令

CREATE TABLE，SELECT

3.8.93 CREATE TABLE PARTITION

功能描述

创建分区表。分区表是把逻辑上的一张表根据某种方案分成几张物理块进行存储，这张逻辑上的表称之为分区表，物理块称之为分区。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的。

常见的分区方案有范围分区（Range Partitioning）、间隔分区（Interval Partitioning）、哈希分区（Hash Partitioning）、列表分区（List Partitioning）等。目前行存表支持范围分区、间隔分区、哈希分区、列表分区，列存表仅支持范围分区。

范围分区是根据表的一列或者多列，将要插入表的记录分为若干个范围，这些范围在不同的分区里没有重叠。为每个范围创建一个分区，用来存储相应的数据。

范围分区的分区策略是指记录插入分区的方式。目前范围分区仅支持范围分区策略。

范围分区策略：根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则给出报错和提示信息。这是最常用的分区策略。

间隔分区是一种特殊的范围分区，相比范围分区，新增间隔值定义，当插入记录找不到匹配的分区时，可以根据间隔值自动创建分区。间隔分区只支持基于表的一列分区，并且该

列只支持 TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE 数据类型。

间隔分区策略：根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则根据分区键值和表定义信息自动创建一个分区，然后将记录插入新分区中，新创建的分区数据范围等于间隔值。

哈希分区是根据表的一列，为每个分区指定模数和余数，将要插入表的记录划分到对应的分区中，每个分区所持有的行都需要满足条件：分区键的值除以为其指定的模数将产生为其指定的余数。

哈希分区策略：根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则返回报错和提示信息。

列表分区是根据表的一列，将要插入表的记录通过每一个分区中出现的键值划分到对应的分区中，这些键值在不同的分区里没有重叠。为每组键值创建一个分区，用来存储相应的数据。

列表分区策略：根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则给出报错和提示信息。

分区可以提供若干好处：

某些类型的查询性能可以得到极大提升。特别是表中访问率较高的行位于一个单独分区或少数几个分区上的情况下。分区可以减少数据的搜索空间，提高数据访问效率。

当查询或更新一个分区的大部分记录时，连续扫描那个分区而不是访问整个表可以获得巨大的性能提升。

如果需要大量加载或者删除的记录位于单独的分区上，则可以通过直接读取或删除那个分区以获得巨大的性能提升，同时还可以避免由于大量 DELETE 导致的 VACUUM 超载（仅范围分区）。

注意事项

唯一约束和主键约束的约束键包含所有分区键将为约束创建 LOCAL 索引，否则创建 GLOBAL 索引。

目前哈希分区和列表分区仅支持单列构建分区键，暂不支持多列构建分区键。

只需要有间隔分区表的 INSERT 权限，往该表 INSERT 数据时就可以自动创建分区。

对于分区表 PARTITION FOR (values)语法，values 只能是常量。

对于分区表 PARTITION FOR (values)语法，values 在需要数据类型转换时，建议使用强制类型转换，以防隐式类型转换结果与预期不符。

分区数最大值为 1048575 个，一般情况下业务不可能创建这么多分区，这样会导致内存不足。应参照参数 local_syscache_threshold 的值合理创建分区，分区表使用内存大致为（分区数 * 3 / 1024）MB。理论上分区占用内存不允许大于 local_syscache_threshold 的值，同时还需要预留部分空间以供其他功能使用。

指定分区语句目前不能走全局索引扫描。

语法格式

```
CREATE TABLE [ IF NOT EXISTS ] partition_table_name
( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option [...] ] }
    [, ... ]
] )
    [ WITH ( {storage_parameter = value} [, ... ] ) ]
    [ COMPRESS | NOCOMPRESS ]
    [ TABLESPACE tablespace_name ]
    [ DISTRIBUTE BY { REPLICATION | { [ HASH ] ( column_name ) } } ]
    [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
    PARTITION BY {
        {VALUES (partition_key)} |
        {RANGE (partition_key) [ INTERVAL ('interval_expr') [ STORE IN
( tablespace_name [, ...] ) ] ] ( part
ition_less_than_item [, ... ] )} |
        {RANGE (partition_key) [ INTERVAL ('interval_expr') [ STORE IN
( tablespace_name [, ...] ) ] ] ( part
ition_start_end_item [, ... ] )} |
        {LIST | HASH (partition_key) (PARTITION partition_name [ VALUES
(list_values_clause) ] opt_table_spac
e ) }
        NOTICE: LIST/HASH partition is only avaliable in CENTRALIZED mode!
    } [ { ENABLE | DISABLE } ROW MOVEMENT ];
```

列约束 column_constraint：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
```

```
  CHECK ( expression ) |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

表约束 table_constraint：

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
[, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE
action ] }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

like 选项 like_option：

```
{ INCLUDING | EXCLUDING } { DEFAULTS | GENERATED | CONSTRAINTS | INDEXES | STORAGE
| COMMENTS | RELOPTIONS |
DISTRIBUTION | ALL }
```

索引存储参数 index_parameters：

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

partition_less_than_item：

```
PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE } )
[TABLESPACE tablespace_name]
```

partition_start_end_item：

```
PARTITION partition_name {
        {START(partition_value) END (partition_value) EVERY (interval_value)} |
        {START(partition_value) END ({partition_value | MAXVALUE})} |
        {START(partition_value)} |
        {END({partition_value | MAXVALUE})}
} [TABLESPACE tablespace_name]
```

参数说明

- IF NOT EXISTS

  如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。

- partition_table_name

  分区表的名称。

  取值范围：字符串，要符合标识符的命名规范。

- column_name

  新表中要创建的字段名。

  取值范围：字符串，要符合标识符的命名规范。

- data_type

  字段的数据类型。

- COLLATE collation

  COLLATE 子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。排序规则可以使用“select * from pg_collation;”命令从 pg_collation 系统表中查询，默认的排序规则为查询结果中以 default 开始的行。

- CONSTRAINT constraint_name

  列约束或表约束的名称。可选的约束子句用于声明约束，新行或者更新的行必须满足这些约束才能成功插入或更新。

  定义约束有两种方法：

  列约束：作为一个列定义的一部分，仅影响该列。

  表约束：不和某个列绑在一起，可以作用于多个列。

- LIKE source_table [like_option …]

  LIKE 子句声明一个表，新表自动从这个表里面继承所有字段名及其数据类型和非空约束。

  和 INHERITS 不同，新表与原来的表之间在创建动作完毕之后是完全无关的。在源表做的任何修改都不会传播到新表中，并且也不可能在扫描源表的时候包含新表的数据。

字段缺省表达式只有在声明了 INCLUDING DEFAULTS 之后才会包含进来。缺省是不包含缺省表达式的，即新表中所有字段的缺省值都是 NULL。

如果指定了 INCLUDING GENERATED，则源表列的生成表达式会复制到新表中。默认不复制生成表达式。

非空约束将总是复制到新表中，CHECK 约束则仅在指定了 INCLUDING CONSTRAINTS 的时候才复制，而其他类型的约束则永远也不会被复制。此规则同时适用于表约束和列约束。

和 INHERITS 不同，被复制的列和约束并不使用相同的名称进行融合。如果明确的指定了相同的名称或者在另外一个 LIKE 子句中，将会报错。

如果指定了 INCLUDING INDEXES，则源表上的索引也将在新表上创建，默认不建立索引。

如果指定了 INCLUDING STORAGE，则拷贝列的 STORAGE 设置也将被拷贝，默认情况下不包含 STORAGE 设置。

如果指定了 INCLUDING COMMENTS，则源表列、约束和索引的注释也会被拷贝过来。默认情况下，不拷贝源表的注释。

如果指定了 INCLUDING RELOPTIONS，则源表的存储参数（即源表的 WITH 子句）也将拷贝至新表。默认情况下，不拷贝源表的存储参数。

INCLUDING ALL 包含了 INCLUDING DEFAULTS、INCLUDING CONSTRAINTS、INCLUDING INDEXES、INCLUDING STORAGE、INCLUDING COMMENTS、INCLUDING PARTITION 和 INCLUDING RELOPTIONS 的内容。

- WITH (storage_parameter [= value] [, …])

这个子句为表或索引指定一个可选的存储参数。参数的详细描述如下所示：

- FILLFACTOR

一个表的填充因子（fillfactor）是一个介于 10 和 100 之间的百分数。100（完全填充）是默认值。如果指定了较小的填充因子，INSERT 操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得 UPDATE 有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为 100 是最佳选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。该参数对于列存表没有意义。

取值范围：10~100

- ORIENTATION

  决定了表的数据的存储方式。

  取值范围：

  COLUMN：表的数据将以列式存储。

  ROW（缺省值）：表的数据将以行式存储。

   须知：orientation 不支持修改。

- STORAGE_TYPE

  指定存储引擎类型，该参数设置成功后就不再支持修改。

  取值范围：

  USTORE，表示表支持 Inplace-Update 存储引擎。特别需要注意，使用 USTORE 表，必须要开启 track_counts 和 track_activities 参数，否则会引起空间膨胀。

  ASTORE，表示表支持 Append-Only 存储引擎。

  默认值：

  不指定表时，默认是 Append-Only 存储。

- COMPRESSION

  列存表的有效值为 LOW/MIDDLE/HIGH/YES/NO，压缩级别依次升高，默认值为 LOW。

  行存表不支持压缩。

- MAX_BATCHROW

  指定了在数据加载过程中一个存储单元可以容纳记录的最大数目。该参数只对列存表有效。

  取值范围：10000~60000，默认 60000。

- PARTIAL_CLUSTER_ROWS

  指定了在数据加载过程中进行将局部聚簇存储的记录数目。该参数只对列存表有效。

  取值范围：大于等于 MAX_BATCHROW，建议取值为 MAX_BATCHROW 的整数倍数。

- DELTAROW_THRESHOLD

预留参数。该参数只对列存表有效。

取值范围：0—9999

- segment

使用段页式的方式存储。本参数仅支持行存表。不支持列存表、临时表、unlog 表。不支持 ustore 存储引擎。

取值范围：on/off

默认值：off

- COMPRESS / NOCOMPRESS

创建一个新表时，需要在创建表语句中指定关键字 COMPRESS，这样，当对该表进行批量插入时就会触发压缩特性。该特性会在页范围内扫描所有元组数据，生成字典、压缩元组数据并进行存储。指定关键字 NOCOMPRESS 则不对表进行压缩。行存表不支持压缩。

缺省值为 NOCOMPRESS，即不对元组数据进行压缩。

- TABLESPACE tablespace_name

指定新表将要在 tablespace_name 表空间内创建。如果没有声明，将使用默认表空间。

- PARTITION BY RANGE(partition_key)

创建范围分区。partition_key 为分区键的名称。

（1）对于从句是 VALUES LESS THAN 的语法格式：

 须知：对于从句是 VALUE LESS THAN 的语法格式，范围分区策略的分区键最多支持 4 列。

该情形下，分区键支持的数据类型为：SMALLINT、INTEGER、BIGINT、DECIMAL、NUMERIC、REAL、DOUBLE PRECISION、CHARACTER VARYING(n)、VARCHAR(n)、CHARACTER(n)、CHAR(n)、CHARACTER、CHAR、TEXT、NVARCHAR、NVARCHAR2、NAME、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。

（2）对于从句是 START END 的语法格式：

 须知：对于从句是 START END 的语法格式，范围分区策略的分区键仅支持 1 列。

该情形下，分区键支持的数据类型为：SMALLINT、INTEGER、BIGINT、DECIMAL、

NUMERIC、REAL、DOUBLE PRECISION、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。

（3）对于指定了 INTERVAL 子句的语法格式：

 须知：对于指定了 INTERVAL 子句的语法格式，范围分区策略的分区键仅支持 1 列。

该情形下，分区键支持的数据类型为：TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。

- PARTITION partition_name VALUES LESS THAN ({ partition_value | MAXVALUE })

指定各分区的信息。partition_name 为范围分区的名称。partition_value 为范围分区的上边界，取值依赖于 partition_key 的类型。MAXVALUE 表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

 须知：

每个分区都需要指定一个上边界。

分区上边界的类型应当和分区键的类型一致。

分区列表是按照分区上边界升序排列的，值较小的分区位于值较大的分区之前。

- **PARTITION partition_name {START (partition_value) END (partition_value) EVERY (interval_value)} | **{START (partition_value) END (partition_value|MAXVALUE)} | {START(partition_value)} | {END (partition_value | MAXVALUE)}

指定各分区的信息，各参数意义如下：

partition_name：范围分区的名称或名称前缀，除以下情形外（假定其中的 partition_name 是 p1），均为分区的名称。

若该定义是 START+END+EVERY 从句，则语义上定义的分区的名称依次为 p1_1, p1_2, …。例如对于定义“PARTITION p1 START(1) END(4) EVERY(1)”，则生成的分区是：[1, 2), [2, 3) 和 [3, 4)，名称依次为 p1_1, p1_2 和 p1_3，即此处的 p1 是名称前缀。

若该定义是第一个分区定义，且该定义有 START 值，则范围（MINVALUE, START）将自动作为第一个实际分区，其名称为 p1_0，然后该定义语义描述的分区名称依次为 p1_1, p1_2, …。例如对于完整定义“PARTITION p1 START(1), PARTITION p2 START(2)”，则生成的分区是：(MINVALUE, 1), [1, 2) 和 [2, MAXVALUE)，其名称依次为 p1_0, p1_1 和 p2，即

此处 p1 是名称前缀，p2 是分区名称。这里 MINVALUE 表示最小值。

partition_value：范围分区的端点值（起始或终点），取值依赖于 partition_key 的类型，不可是 MAXVALUE。

interval_value：对[START，END）表示的范围进行切分，interval_value 是指定切分后每个分区的宽度，不可是 MAXVALUE；如果（END-START）值不能整除以 EVERY 值，则仅最后一个分区的宽度小于 EVERY 值。

MAXVALUE：表示最大值，它通常用于设置最后一个范围分区的上边界。

 须知：

在创建分区表若第一个分区定义含 START 值，则范围（MINVALUE，START）将自动作为实际的第一个分区。

START END 语法需要遵循以下限制：- 每个 partition_start_end_item 中的 START 值(如果有的话，下同）必须小于其 END 值。

相邻的两个 partition_start_end_item，第一个的 END 值必须等于第二个的 START 值；

每个 partition_start_end_item 中的 EVERY 值必须是正向递增的，且必须小于（END-START）值；

每个分区包含起始值，不包含终点值，即形如：[起始值，终点值)，起始值是 MINVALUE 时则不包含；

一个 partition_start_end_item 创建的每个分区所属的 TABLESPACE 一样；

partition_name 作为分区名称前缀时，其长度不要超过 57 字节，超过时自动截断；

在创建、修改分区表时请注意分区表的分区总数不可超过最大限制（1048575）；

在创建分区表时 START END 与 LESS THAN 语法不可混合使用。

即使创建分区表时使用 START END 语法，备份(gs_dump)出的 SQL 语句也是 VALUES LESS THAN 语法格式。

- INTERVAL ('interval_expr') [STORE IN (tablespace_name [, …])]

  间隔分区定义信息。

  interval_expr：自动创建分区的间隔，例如：1 day、1 month。

  STORE IN (tablespace_name [, …])：指定存放自动创建分区的表空间列表，如果有指

定，则自动创建的分区从表空间列表中循环选择使用，否则使用分区表默认的表空间。

 须知：列存表不支持间隔分区。

- PARTITION BY LIST(partition_key)

  创建列表分区。partition_key 为分区键的名称。

  对于 partition_key，列表分区策略的分区键仅支持 1 列。

  对于从句是 VALUES (list_values_clause)的语法格式，list_values_clause 中包含了对应分区存在的键值，推荐每个分区的键值数量不超过 64 个。

  分区键支持的数据类型为：INT1、INT2、INT4、INT8、NUMERIC、VARCHAR(n)、CHAR、BPCHAR、NVARCHAR、NVARCHAR2、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。分区个数不能超过 1048575 个。

- PARTITION BY HASH(partition_key)

  创建哈希分区。partition_key 为分区键的名称。

  对于 partition_key，哈希分区策略的分区键仅支持 1 列。

  分区键支持的数据类型为：INT1、INT2、INT4、INT8、NUMERIC、VARCHAR(n)、CHAR、BPCHAR、TEXT、NVARCHAR、NVARCHAR2、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。分区个数不能超过 1048575 个。

- { ENABLE | DISABLE } ROW MOVEMENT

  行迁移开关。

  如果进行 UPDATE 操作时，更新了元组在分区键上的值，造成了该元组所在分区发生变化，就会根据该开关给出报错信息，或者进行元组在分区间的转移。

  取值范围：

  ENABLE（缺省值）：行迁移开关打开。

  DISABLE：行迁移开关关闭。

   须知：列表/哈希分区表暂不支持 ROW MOVEMENT。

- NOT NULL

  字段值不允许为 NULL。ENABLE 用于语法兼容，可省略。

- NULL

  字段值允许 NULL ，这是缺省。

  这个子句只是为和非标准 SQL 数据库兼容。不建议使用。

- CHECK (condition) [NO INHERIT]

  CHECK 约束声明一个布尔表达式，每次要插入的新行或者要更新的行的新值必须使表达式结果为真或未知才能成功，否则会抛出一个异常并且不会修改数据库。

  声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以引用多个字段。

  用 NO INHERIT 标记的约束将不会传递到子表中去。

  ENABLE 用于语法兼容，可省略。

- DEFAULT default_expr

  DEFAULT 子句给字段指定缺省值。该数值可以是任何不含变量的表达式(不允许使用子查询和对本表中的其他字段的交叉引用)。缺省表达式的数据类型必须和字段类型匹配。

  缺省表达式将被用于任何未声明该字段数值的插入操作。如果没有指定缺省值则缺省值为 NULL 。

- GENERATED ALWAYS AS (generation_expr) STORED

  该子句将字段创建为生成列，生成列的值在写入（插入或更新）数据时由 generation_expr 计算得到，STORED 表示像普通列一样存储生成列的值。

  说明：- 生成表达式不能以任何方式引用当前行以外的其他数据。生成表达式不能引用其他生成列，不能引用系统列。生成表达式不能返回结果集，不能使用子查询，不能使用聚集函数，不能使用窗口函数。生成表达式调用的函数只能是不可变（IMMUTABLE）函数。

  不能为生成列指定默认值。

  生成列不能作为分区键的一部分。

  生成列不能和 ON UPDATE 约束字句的 CASCADE,SET NULL,SET DEFAULT 动作同时指定。生成列不能和 ON DELETE 约束字句的 SET NULL、SET DEFAULT 动作同时指定。

  修改和删除生成列的方法和普通列相同。删除生成列依赖的普通列，生成列被自动删除。不能改变生成列所依赖的列的类型。

生成列不能被直接写入。在 INSERT 或 UPDATE 命令中，不能为生成列指定值，但是可以指定关键字 DEFAULT。

生成列的权限控制和普通列一样。

列存表、内存表 MOT 不支持生成列。外表中仅 postgres_fdw 支持生成列。

- UNIQUE index_parameters | UNIQUE (column_name [, …]) index_parameters

  UNIQUE 约束表示表里的一个字段或多个字段的组合必须在全表范围内唯一。

  对于唯一约束，NULL 被认为是互不相等的。

- PRIMARY KEY index_parameters | PRIMARY KEY (column_name [, …]) index_parameters

  主键约束声明表中的一个或者多个字段只能包含唯一的非 NULL 值。

  一个表只能声明一个主键。

- DEFERRABLE | NOT DEFERRABLE

这两个关键字设置该约束是否可推迟。一个不可推迟的约束将在每条命令之后马上检查。可推迟约束可以推迟到事务结尾使用 SET CONSTRAINTS 命令检查。缺省是 NOT DEFERRABLE。目前，UNIQUE 约束、主键约束、外键约束可以接受这个子句。所有其他约束类型都是不可推迟的。

- INITIALLY IMMEDIATE | INITIALLY DEFERRED

  如果约束是可推迟的，则这个子句声明检查约束的缺省时间。

  如果约束是 INITIALLY IMMEDIATE（缺省），则在每条语句执行之后就立即检查它；

  如果约束是 INITIALLY DEFERRED ，则只有在事务结尾才检查它。

  约束检查的时间可以用 SET CONSTRAINTS 命令修改。

- USING INDEX TABLESPACE tablespace_name

为 UNIQUE 或 PRIMARY KEY 约束相关的索引声明一个表空间。如果没有提供这个子句，这个索引将在 default_tablespace 中创建，如果 default_tablespace 为空，将使用数据库的缺省表空间。

示例

示例 1：创建范围分区表 tpcds.web_returns_p1，含有 8 个分区，分区键为 integer 类型。分区的范围分别为：wr_returned_date_sk< 2450815、2450815<= wr_returned_date_sk< 2451179、

2451179<=wr_returned_date_sk< 2451544 、 2451544 <= wr_returned_date_sk< 2451910 、2451910 <= wr_returned_date_sk< 2452275 、 2452275 <= wr_returned_date_sk< 2452640 、2452640 <= wr_returned_date_sk< 2453005、wr_returned_date_sk>=2453005。

```
--创建表 tpcds.web_returns。
postgres=# CREATE TABLE tpcds.web_returns
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)
);
--创建分区表 tpcds.web_returns_p1。
postgres=# CREATE TABLE tpcds.web_returns_p1
(
    WR_RETURNED_DATE_SK       INTEGER                       ,
    WR_RETURNED_TIME_SK       INTEGER                       ,
    WR_ITEM_SK                INTEGER               NOT NULL,
    WR_REFUNDED_CUSTOMER_SK   INTEGER                       ,
    WR_REFUNDED_CDEMO_SK      INTEGER                       ,
    WR_REFUNDED_HDEMO_SK      INTEGER                       ,
    WR_REFUNDED_ADDR_SK       INTEGER                       ,
    WR_RETURNING_CUSTOMER_SK  INTEGER                       ,
    WR_RETURNING_CDEMO_SK     INTEGER                       ,
    WR_RETURNING_HDEMO_SK     INTEGER                       ,
    WR_RETURNING_ADDR_SK      INTEGER                       ,
    WR_WEB_PAGE_SK            INTEGER                       ,
    WR_REASON_SK              INTEGER                       ,
    WR_ORDER_NUMBER           BIGINT                NOT NULL,
    WR_RETURN_QUANTITY        INTEGER                       ,
    WR_RETURN_AMT             DECIMAL(7,2)                  ,
```

```
    WR_RETURN_TAX             DECIMAL(7,2)                ,
    WR_RETURN_AMT_INC_TAX     DECIMAL(7,2)                ,
    WR_FEE                    DECIMAL(7,2)                ,
    WR_RETURN_SHIP_COST       DECIMAL(7,2)                ,
    WR_REFUNDED_CASH          DECIMAL(7,2)                ,
    WR_REVERSED_CHARGE        DECIMAL(7,2)                ,
    WR_ACCOUNT_CREDIT         DECIMAL(7,2)                ,
    WR_NET_LOSS               DECIMAL(7,2)
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(2451910),
        PARTITION P5 VALUES LESS THAN(2452275),
        PARTITION P6 VALUES LESS THAN(2452640),
        PARTITION P7 VALUES LESS THAN(2453005),
        PARTITION P8 VALUES LESS THAN(MAXVALUE)
);

--从示例数据表导入数据。
postgres=# INSERT INTO tpcds.web_returns_p1 SELECT * FROM tpcds.web_returns;

--删除分区 P8。
postgres=# ALTER TABLE tpcds.web_returns_p1 DROP PARTITION P8;

--增加分区 WR_RETURNED_DATE_SK 介于 2453005 和 2453105 之间。
postgres=# ALTER TABLE tpcds.web_returns_p1 ADD PARTITION P8 VALUES LESS THAN
(2453105);

--增加分区 WR_RETURNED_DATE_SK 介于 2453105 和 MAXVALUE 之间。
postgres=# ALTER TABLE tpcds.web_returns_p1 ADD PARTITION P9 VALUES LESS THAN
(MAXVALUE);

--删除分区 P8。
postgres=# ALTER TABLE tpcds.web_returns_p1 DROP PARTITION FOR (2453005);

--分区 P7 重命名为 P10。
postgres=# ALTER TABLE tpcds.web_returns_p1 RENAME PARTITION P7 TO P10;
```

```
--分区 P6 重命名为 P11。
postgres=# ALTER TABLE tpcds.web_returns_p1 RENAME PARTITION FOR (2452639) TO
P11;

--查询分区 P10 的行数。
postgres=# SELECT count(*) FROM tpcds.web_returns_p1 PARTITION (P10);
 count
--------
 0
(1 row)

--查询分区 P1 的行数。
postgres=# SELECT COUNT(*) FROM tpcds.web_returns_p1 PARTITION FOR (2450815);
 count
--------
 0
(1 row)
```

示例 2：创建范围分区表 tpcds.web_returns_p2，含有 8 个分区，分区键类型为 integer 类型，其中第 8 个分区上边界为 MAXVALUE。

八个分区的范围分别为：wr_returned_date_sk< 2450815 、 2450815<= wr_returned_date_sk< 2451179 、 2451179<=wr_returned_date_sk< 2451544 、 2451544 <= wr_returned_date_sk< 2451910 、 2451910 <= wr_returned_date_sk< 2452275 、 2452275 <= wr_returned_date_sk< 2452640 、 2452640 <= wr_returned_date_sk< 2453005 、 wr_returned_date_sk>=2453005。

分区表 tpcds.web_returns_p2 的表空间为 example1；分区 P1 到 P7 没有声明表空间，使用采用分区表 tpcds.web_returns_p2 的表空间 example1；指定分区 P8 的表空间为 example2。

假定数据库节点的数据目录/pg_location/mount1/path1，数据库节点的数据目录/pg_location/mount2/path2，数据库节点的数据目录/pg_location/mount3/path3，数据库节点的数据目录/pg_location/mount4/path4 是 dwsadmin 用户拥有读写权限的空目录。

```
postgres=# CREATE TABLESPACE example1 RELATIVE LOCATION
'tablespace1/tablespace_1';
postgres=# CREATE TABLESPACE example2 RELATIVE LOCATION
'tablespace2/tablespace_2';
postgres=# CREATE TABLESPACE example3 RELATIVE LOCATION
'tablespace3/tablespace_3';
postgres=# CREATE TABLESPACE example4 RELATIVE LOCATION
'tablespace4/tablespace_4';
```

```
postgres=# CREATE TABLE tpcds.web_returns_p2
(
    WR_RETURNED_DATE_SK       INTEGER                       ,
    WR_RETURNED_TIME_SK       INTEGER                       ,
    WR_ITEM_SK                INTEGER                NOT NULL,
    WR_REFUNDED_CUSTOMER_SK   INTEGER                       ,
    WR_REFUNDED_CDEMO_SK      INTEGER                       ,
    WR_REFUNDED_HDEMO_SK      INTEGER                       ,
    WR_REFUNDED_ADDR_SK       INTEGER                       ,
    WR_RETURNING_CUSTOMER_SK  INTEGER                       ,
    WR_RETURNING_CDEMO_SK     INTEGER                       ,
    WR_RETURNING_HDEMO_SK     INTEGER                       ,
    WR_RETURNING_ADDR_SK      INTEGER                       ,
    WR_WEB_PAGE_SK            INTEGER                       ,
    WR_REASON_SK              INTEGER                       ,
    WR_ORDER_NUMBER           BIGINT                 NOT NULL,
    WR_RETURN_QUANTITY        INTEGER                       ,
    WR_RETURN_AMT             DECIMAL(7,2)                  ,
    WR_RETURN_TAX             DECIMAL(7,2)                  ,
    WR_RETURN_AMT_INC_TAX     DECIMAL(7,2)                  ,
    WR_FEE                    DECIMAL(7,2)                  ,
    WR_RETURN_SHIP_COST       DECIMAL(7,2)                  ,
    WR_REFUNDED_CASH          DECIMAL(7,2)                  ,
    WR_REVERSED_CHARGE        DECIMAL(7,2)                  ,
    WR_ACCOUNT_CREDIT         DECIMAL(7,2)                  ,
    WR_NET_LOSS               DECIMAL(7,2)
)
TABLESPACE example1
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(2451910),
        PARTITION P5 VALUES LESS THAN(2452275),
        PARTITION P6 VALUES LESS THAN(2452640),
        PARTITION P7 VALUES LESS THAN(2453005),
        PARTITION P8 VALUES LESS THAN(MAXVALUE) TABLESPACE example2
)
ENABLE ROW MOVEMENT;
```

```
--以 like 方式创建一个分区表。
postgres=# CREATE TABLE tpcds.web_returns_p3 (LIKE tpcds.web_returns_p2
INCLUDING PARTITION);

--修改分区 P1 的表空间为 example2。
postgres=# ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P1 TABLESPACE
example2;

--修改分区 P2 的表空间为 example3。
postgres=# ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P2 TABLESPACE
example3;

--以 2453010 为分割点切分 P8。
postgres=# ALTER TABLE tpcds.web_returns_p2 SPLIT PARTITION P8 AT (2453010) INTO
(
        PARTITION P9,
        PARTITION P10
);

--将 P6, P7 合并为一个分区。
postgres=# ALTER TABLE tpcds.web_returns_p2 MERGE PARTITIONS P6, P7 INTO
PARTITION P8;

--修改分区表迁移属性。
postgres=# ALTER TABLE tpcds.web_returns_p2 DISABLE ROW MOVEMENT;
--删除表和表空间。
postgres=# DROP TABLE tpcds.web_returns_p1;
postgres=# DROP TABLE tpcds.web_returns_p2;
postgres=# DROP TABLE tpcds.web_returns_p3;
postgres=# DROP TABLESPACE example1;
postgres=# DROP TABLESPACE example2;
postgres=# DROP TABLESPACE example3;
postgres=# DROP TABLESPACE example4;
```

示例 3：START END 语法创建、修改 Range 分区表。

假定/home/gbase/startend_tbs1、/home/gbase/startend_tbs2、/home/gbase/startend_tbs3、/home/gbase/startend_tbs4 是 gbase 用户拥有读写权限的空目录。

```
-- 创建表空间
postgres=# CREATE TABLESPACE startend_tbs1 LOCATION
'/home/gbase/startend_tbs1';
```

```
postgres=# CREATE TABLESPACE startend_tbs2 LOCATION
'/home/gbase/startend_tbs2';
postgres=# CREATE TABLESPACE startend_tbs3 LOCATION
'/home/gbase/startend_tbs3';
postgres=# CREATE TABLESPACE startend_tbs4 LOCATION
'/home/gbase/startend_tbs4';

-- 创建临时 schema
postgres=# CREATE SCHEMA tpcds;
postgres=# SET CURRENT_SCHEMA TO tpcds;

-- 创建分区表，分区键是 integer 类型
postgres=# CREATE TABLE tpcds.startend_pt (c1 INT, c2 INT)
TABLESPACE startend_tbs1
PARTITION BY RANGE (c2) (
    PARTITION p1 START(1) END(1000) EVERY(200) TABLESPACE startend_tbs2,
    PARTITION p2 END(2000),
    PARTITION p3 START(2000) END(2500) TABLESPACE startend_tbs3,
    PARTITION p4 START(2500),
    PARTITION p5 START(3000) END(5000) EVERY(1000) TABLESPACE startend_tbs4
)
ENABLE ROW MOVEMENT;

-- 查看分区表信息
postgres=# SELECT relname, boundaries, spcname FROM pg_partition p JOIN
pg_tablespace t ON p.reltablespace=t.oid and
p.parentid='tpcds.startend_pt'::regclass ORDER BY 1;
   relname    | boundaries |    spcname
--------------+------------+---------------
 p1_0         | {1}        | startend_tbs2
 p1_1         | {201}      | startend_tbs2
 p1_2         | {401}      | startend_tbs2
 p1_3         | {601}      | startend_tbs2
 p1_4         | {801}      | startend_tbs2
 p1_5         | {1000}     | startend_tbs2
 p2           | {2000}     | startend_tbs1
 p3           | {2500}     | startend_tbs3
 p4           | {3000}     | startend_tbs1
 p5_1         | {4000}     | startend_tbs4
 p5_2         | {5000}     | startend_tbs4
 startend_pt  |            | startend_tbs1
(12 rows)
```

```
-- 导入数据，查看分区数据量
postgres=# INSERT INTO tpcds.startend_pt VALUES (GENERATE_SERIES(0, 4999),
GENERATE_SERIES(0, 4999));
postgres=# SELECT COUNT(*) FROM tpcds.startend_pt PARTITION FOR (0);
 count
-------
     1
(1 row)

postgres=# SELECT COUNT(*) FROM tpcds.startend_pt PARTITION (p3);
 count
-------
   500
(1 row)

-- 增加分区：[5000, 5300), [5300, 5600), [5600, 5900), [5900, 6000)
postgres=# ALTER TABLE tpcds.startend_pt ADD PARTITION p6 START(5000) END(6000)
EVERY(300) TABLESPACE startend_tbs4;

-- 增加 MAXVALUE 分区：p7
postgres=# ALTER TABLE tpcds.startend_pt ADD PARTITION p7 END(MAXVALUE);

-- 重命名分区 p7 为 p8
postgres=# ALTER TABLE tpcds.startend_pt RENAME PARTITION p7 TO p8;

-- 删除分区 p8
postgres=# ALTER TABLE tpcds.startend_pt DROP PARTITION p8;

-- 重命名 5950 所在的分区为：p71
postgres=# ALTER TABLE tpcds.startend_pt RENAME PARTITION FOR(5950) TO p71;

-- 分裂 4500 所在的分区[4000, 5000)
postgres=# ALTER TABLE tpcds.startend_pt SPLIT PARTITION FOR(4500)
INTO(PARTITION q1 START(4000) END(5000) EVERY(250) TABLESPACE startend_tbs3);

-- 修改分区 p2 的表空间为 startend_tbs4
postgres=# ALTER TABLE tpcds.startend_pt MOVE PARTITION p2 TABLESPACE
startend_tbs4;

-- 查看分区情形
```

```
postgres=# SELECT relname, boundaries, spcname FROM pg_partition p JOIN
pg_tablespace t ON p.reltablespace=t.oid and
p.parentid='tpcds.startend_pt'::regclass ORDER BY 1;
   relname    | boundaries |    spcname
--------------+------------+---------------
 p1_0         | {1}        | startend_tbs2
 p1_1         | {201}      | startend_tbs2
 p1_2         | {401}      | startend_tbs2
 p1_3         | {601}      | startend_tbs2
 p1_4         | {801}      | startend_tbs2
 p1_5         | {1000}     | startend_tbs2
 p2           | {2000}     | startend_tbs4
 p3           | {2500}     | startend_tbs3
 p4           | {3000}     | startend_tbs1
 p5_1         | {4000}     | startend_tbs4
 p6_1         | {5300}     | startend_tbs4
 p6_2         | {5600}     | startend_tbs4
 p6_3         | {5900}     | startend_tbs4
 p71          | {6000}     | startend_tbs4
 q1_1         | {4250}     | startend_tbs3
 q1_2         | {4500}     | startend_tbs3
 q1_3         | {4750}     | startend_tbs3
 q1_4         | {5000}     | startend_tbs3
 startend_pt  |            | startend_tbs1
(19 rows)

-- 删除表和表空间
postgres=# DROP SCHEMA tpcds CASCADE;
postgres=# DROP TABLESPACE startend_tbs1;
postgres=# DROP TABLESPACE startend_tbs2;
postgres=# DROP TABLESPACE startend_tbs3;
postgres=# DROP TABLESPACE startend_tbs4;
```

示例 4：创建间隔分区表 sales，初始包含 2 个分区，分区键为 DATE 类型。 分区的范围分别为：time_id < '2019-02-01 00:00:00'、

'2019-02-01 00:00:00' <= time_id < '2019-02-02 00:00:00' 。

```
--创建表 sales
postgres=# CREATE TABLE sales
(prod_id NUMBER(6),
 cust_id NUMBER,
 time_id DATE,
```

```
 channel_id CHAR(1),
 promo_id NUMBER(6),
 quantity_sold NUMBER(3),
 amount_sold NUMBER(10,2)
)
PARTITION BY RANGE (time_id)
INTERVAL('1 day')
( PARTITION p1 VALUES LESS THAN ('2019-02-01 00:00:00'),
  PARTITION p2 VALUES LESS THAN ('2019-02-02 00:00:00')
);

-- 数据插入分区 p1
postgres=# INSERT INTO sales VALUES(1, 12, '2019-01-10 00:00:00', 'a', 1, 1, 1);

-- 数据插入分区 p2
postgres=# INSERT INTO sales VALUES(1, 12, '2019-02-01 00:00:00', 'a', 1, 1, 1);

-- 查看分区信息
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'sales' AND t1.parttype
= 'p';
 relname | partstrategy |       boundaries
---------+--------------+------------------------
 p1      | r            | {"2019-02-01 00:00:00"}
 p2      | r            | {"2019-02-02 00:00:00"}
(2 rows)

-- 插入数据没有匹配的分区，新创建一个分区，并将数据插入该分区
-- 新分区的范围为 '2019-02-05 00:00:00' <= time_id < '2019-02-06 00:00:00'
postgres=# INSERT INTO sales VALUES(1, 12, '2019-02-05 00:00:00', 'a', 1, 1, 1);

-- 插入数据没有匹配的分区，新创建一个分区，并将数据插入该分区
-- 新分区的范围为 '2019-02-03 00:00:00' <= time_id < '2019-02-04 00:00:00'
postgres=# INSERT INTO sales VALUES(1, 12, '2019-02-03 00:00:00', 'a', 1, 1, 1);

-- 查看分区信息
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'sales' AND t1.parttype
= 'p';
 relname | partstrategy |       boundaries
---------+--------------+------------------------
 sys_p1  | i            | {"2019-02-06 00:00:00"}
```

```
 sys_p2   | i             | {"2019-02-04 00:00:00"}
 p1       | r             | {"2019-02-01 00:00:00"}
 p2       | r             | {"2019-02-02 00:00:00"}
(4 rows)
```

示例 5：创建 LIST 分区表 test_list，初始包含 4 个分区，分区键为 INT 类型。4 个分区的范围分别为：2000、3000、4000、5000。

```
--创建表 test_list
postgres=# create table test_list (col1 int, col2 int)
partition by list(col1)
(
partition p1 values (2000),
partition p2 values (3000),
partition p3 values (4000),
partition p4 values (5000)
);

-- 数据插入
postgres=# INSERT INTO test_list VALUES(2000, 2000);
INSERT 0 1
postgres=# INSERT INTO test_list VALUES(3000, 3000);
INSERT 0 1

-- 查看分区信息
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'test_list' AND
t1.parttype = 'p';
 relname | partstrategy | boundaries
---------+--------------+-----------
 p1      | l            | {2000}
 p2      | l            | {3000}
 p3      | l            | {4000}
 p4      | l            | {5000}
(4 rows)

-- 插入数据没有匹配到分区，报错处理
postgres=# INSERT INTO test_list VALUES(6000, 6000);
ERROR:  inserted partition key does not map to any table partition

-- 添加分区
```

```
postgres=# alter table test_list add partition p5 values (6000);
ALTER TABLE
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'test_list' AND
t1.parttype = 'p';
 relname | partstrategy | boundaries
---------+--------------+------------
 p5      | l            | {6000}
 p4      | l            | {5000}
 p1      | l            | {2000}
 p2      | l            | {3000}
 p3      | l            | {4000}
(5 rows)
postgres=# INSERT INTO test_list VALUES(6000, 6000);
INSERT 0 1

-- 分区表和普通表交换数据
postgres=# create table t1 (col1 int, col2 int);
CREATE TABLE
postgres=# select * from test_list partition (p1);
 col1 | col2
------+------
 2000 | 2000
(1 row)
postgres=# alter table test_list exchange partition (p1) with table t1;
ALTER TABLE
postgres=# select * from test_list partition (p1);
 col1 | col2
------+------
(0 rows)
postgres=# select * from t1;
 col1 | col2
------+------
 2000 | 2000
(1 row)

-- truncate 分区
postgres=# select * from test_list partition (p2);
 col1 | col2
------+------
 3000 | 3000
(1 row)
```

```
postgres=# alter table test_list truncate partition p2;
ALTER TABLE
postgres=# select * from test_list partition (p2);
 col1 | col2
------+------
(0 rows)

-- 删除分区
postgres=# alter table test_list drop partition p5;
ALTER TABLE
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'test_list' AND
t1.parttype = 'p';
 relname | partstrategy | boundaries
---------+--------------+------------
 p4      | 1            | {5000}
 p1      | 1            | {2000}
 p2      | 1            | {3000}
 p3      | 1            | {4000}
(4 rows)

postgres=# INSERT INTO test_list VALUES(6000, 6000);
ERROR:  inserted partition key does not map to any table partition

-- 删除分区表
postgres=# drop table test_list;
```

示例 6：创建 HASH 分区表 test_hash，初始包含 2 个分区，分区键为 INT 类型。

```
--创建表 test_hash
postgres=# create table test_hash (col1 int, col2 int)
partition by hash(col1)
(
partition p1,
partition p2
);

-- 数据插入
postgres=# INSERT INTO test_hash VALUES(1, 1);
INSERT 0 1
postgres=# INSERT INTO test_hash VALUES(2, 2);
INSERT 0 1
postgres=# INSERT INTO test_hash VALUES(3, 3);
```

```
INSERT 0 1
postgres=# INSERT INTO test_hash VALUES(4, 4);
INSERT 0 1

-- 查看分区信息
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'test_hash' AND
t1.parttype = 'p';
 relname | partstrategy | boundaries
---------+--------------+-----------
 p1      | h            | {0}
 p2      | h            | {1}
(2 rows)

-- 查看数据
postgres=# select * from test_hash partition (p1);
 col1 | col2
------+------
    3 |    3
    4 |    4
(2 rows)

postgres=# select * from test_hash partition (p2);
 col1 | col2
------+------
    1 |    1
    2 |    2
(2 rows)

-- 分区表和普通表交换数据
postgres=# create table t1 (col1 int, col2 int);
CREATE TABLE
postgres=# alter table test_hash exchange partition (p1) with table t1;
ALTER TABLE
postgres=# select * from test_hash partition (p1);
 col1 | col2
------+------
(0 rows)
postgres=# select * from t1;
 col1 | col2
------+------
    3 |    3
```

```
    4 |    4
(2 rows)

-- truncate 分区
postgres=# alter table test_hash truncate partition p2;
ALTER TABLE
postgres=# select * from test_hash partition (p2);
 col1 | col2
------+------
(0 rows)

-- 删除分区表
postgres=# drop table test_hash;
```

相关命令

ALTER TABLE PARTITION，DROP TABLE

3.8.94 CREATE TABLE SUBPARTITION

功能描述

创建二级分区表。分区表是把逻辑上的一张表根据某种方案分成几张物理块进行存储，这张逻辑上的表称之为分区表，物理块称之为分区。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的。对于二级分区表，顶层节点表和一级分区都是逻辑表，不存储数据，只有二级分区（叶子节点）存储数据。

二级分区表的分区方案是由两个一级分区的分区方案组合而来的，一级分区的分区方案详见章节 CREATE TABLE PARTITION。

常见的二级分区表组合方案有 Range-Range 分区、Range-List 分区、Range-Hash 分区、List-Range 分区、List-List 分区、List-Hash 分区、Hash-Range 分区、Hash-List 分区、Hash-Hash 分区等。目前二级分区仅支持行存表。

注意事项

二级分区表有两个分区键，每个分区键只能支持 1 列。

唯一约束和主键约束的约束键包含所有分区键将为约束创建 LOCAL 索引，否则创建 GLOBAL 索引。

二级分区表的二级分区（叶子节点）个数不能超过 1048575 个，一级分区无限制，但一级分区下面至少有一个二级分区。

二级分区表只支持行存，不支持列存、段页式、hashbucket。

不支持 Upsert、Merge into。

指定分区查询时，如 select * from tablename partition/subpartition (partitionname)，关键字 partition 和 subpartition 注意不要写错。如果写错，查询不会报错，这时查询会变为对表起别名进行查询。

不支持对二级分区 subpartition for (values)查询。如 select * from tablename subpartition for (values)。

不支持密态数据库、账本数据库和行级访问控制。

对于二级分区表 PARTITION FOR (values)语法，values 只能是常量。

对于分区表 PARTITION/SUBPARTITION FOR (values)语法，values 在需要数据类型转换时，建议使用强制类型转换，以防隐式类型转换结果与预期不符。

指定分区语句目前不能走全局索引扫描。

语法格式

```
CREATE TABLE [ IF NOT EXISTS ] subpartition_table_name
( { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option [...] ] }
    [, ... ]
)
    [ WITH ( {storage_parameter = value} [, ... ] ) ]
    [ COMPRESS | NOCOMPRESS ]
    [ TABLESPACE tablespace_name ]
    PARTITION BY {RANGE | LIST | HASH} (partition_key) SUBPARTITION BY {RANGE |
LIST | HASH} (subpartition_ke
y)
    (
      PARTITION partition_name1 [ VALUES LESS THAN (val1) | VALUES (val1[, ...]) ]
[ TABLESPACE tablespace ]
      (
        { SUBPARTITION subpartition_name1 [ VALUES LESS THAN (val1_1) | VALUES
(val1_1[, ...])]  [ TABLESPACE
 tablespace ] } [, ...]
      )
      [, ...]
    ) [ { ENABLE | DISABLE } ROW MOVEMENT ];
```

列约束 column_constraint：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

表约束 table_constraint：

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
[, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE
action ] }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

like 选项 like_option：

```
{ INCLUDING | EXCLUDING } { DEFAULTS | GENERATED | CONSTRAINTS | INDEXES | STORAGE
| COMMENTS | RELOPTIONS| ALL }
```

索引存储参数 index_parameters：

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

参数说明

- IF NOT EXISTS

如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。

- subpartition_table_name

二级分区表的名称。

取值范围：字符串，要符合标识符的命名规范。

- column_name

新表中要创建的字段名。

取值范围：字符串，要符合标识符的命名规范。

- data_type

字段的数据类型。

- COLLATE collation

COLLATE 子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。排序规则可以使用“select * from pg_collation;”命令从 pg_collation 系统表中查询，默认的排序规则为查询结果中以 default 开始的行。

- CONSTRAINT constraint_name

列约束或表约束的名称。可选的约束子句用于声明约束，新行或者更新的行必须满足这些约束才能成功插入或更新。

定义约束有两种方法：

列约束：作为一个列定义的一部分，仅影响该列。

表约束：不和某个列绑在一起，可以作用于多个列。

- LIKE source_table [like_option …]

二级分区表暂不支持该功能。

- WITH (storage_parameter [= value] [, …])

这个子句为表或索引指定一个可选的存储参数。参数的详细描述如下所示：

- FILLFACTOR

一个表的填充因子（fillfactor）是一个介于 10 和 100 之间的百分数。100（完全填充）是默认值。如果指定了较小的填充因子，INSERT 操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得 UPDATE 有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为 100 是最佳选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。该参数对于列存表没有意义。

取值范围：10~100

- ORIENTATION

  决定了表的数据的存储方式。

  取值范围：

  COLUMN：表的数据将以列式存储。

  ROW（缺省值）：表的数据将以行式存储。

   须知：orientation 不支持修改。

- COMPRESSION

  列存表的有效值为 LOW/MIDDLE/HIGH/YES/NO，压缩级别依次升高，默认值为 LOW。

  行存表不支持压缩。

- MAX_BATCHROW

  指定了在数据加载过程中一个存储单元可以容纳记录的最大数目。该参数只对列存表有效。

  取值范围：10000~60000，默认 60000。

- PARTIAL_CLUSTER_ROWS

  指定了在数据加载过程中进行将局部聚簇存储的记录数目。该参数只对列存表有效。

  取值范围：大于等于 MAX_BATCHROW，建议取值为 MAX_BATCHROW 的整数倍数。

- DELTAROW_THRESHOLD

  预留参数。该参数只对列存表有效。

  取值范围：0—9999

- COMPRESS / NOCOMPRESS

  创建一个新表时，需要在创建表语句中指定关键字 COMPRESS，这样，当对该表进行批量插入时就会触发压缩特性。该特性会在页范围内扫描所有元组数据，生成字典、压缩元组数据并进行存储。指定关键字 NOCOMPRESS 则不对表进行压缩。行存表不支持压缩。

  缺省值为 NOCOMPRESS，即不对元组数据进行压缩。

- TABLESPACE tablespace_name

  指定新表将要在 tablespace_name 表空间内创建。如果没有声明，将使用默认表空间。

- PARTITION BY {RANGE | LIST | HASH} (partition_key)

  对于 partition_key，分区策略的分区键仅支持 1 列。

  分区键支持的数据类型和一级分区表约束保持一致。

- SUBPARTITION BY {RANGE | LIST | HASH} (subpartition_key)

  对于 subpartition_key，分区策略的分区键仅支持 1 列。

  分区键支持的数据类型和一级分区表约束保持一致。

- { ENABLE | DISABLE } ROW MOVEMENT

  行迁移开关。

  如果进行 UPDATE 操作时，更新了元组在分区键上的值，造成了该元组所在分区发生变化，就会根据该开关给出报错信息，或者进行元组在分区间的转移。

  取值范围：

  ENABLE（缺省值）：行迁移开关打开。

  DISABLE：行迁移开关关闭。

- NOT NULL

  字段值不允许为 NULL。ENABLE 用于语法兼容，可省略。

  NULL

  字段值允许 NULL ，这是缺省。

  这个子句只是为和非标准 SQL 数据库兼容。不建议使用。

- CHECK (condition) [NO INHERIT]

  CHECK 约束声明一个布尔表达式，每次要插入的新行或者要更新的行的新值必须使表达式结果为真或未知才能成功，否则会抛出一个异常并且不会修改数据库。

  声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以引用多个字段。

  用 NO INHERIT 标记的约束将不会传递到子表中去。

  ENABLE 用于语法兼容，可省略。

- DEFAULT default_expr

  DEFAULT 子句给字段指定缺省值。该数值可以是任何不含变量的表达式(不允许使用子

查询和对本表中的其他字段的交叉引用)。缺省表达式的数据类型必须和字段类型匹配。

缺省表达式将被用于任何未声明该字段数值的插入操作。如果没有指定缺省值则缺省值为 NULL 。

- GENERATED ALWAYS AS (generation_expr) STORED

该子句将字段创建为生成列，生成列的值在写入(插入或更新)数据时由 generation_expr 计算得到，STORED 表示像普通列一样存储生成列的值。

说明：

生成表达式不能以任何方式引用当前行以外的其他数据。生成表达式不能引用其他生成列，不能引用系统列。生成表达式不能返回结果集，不能使用子查询，不能使用聚集函数，不能使用窗口函数。生成表达式调用的函数只能是不可变（IMMUTABLE）函数。

不能为生成列指定默认值。

生成列不能作为分区键的一部分。

生成列不能和 ON UPDATE 约束字句的 CASCADE、SET NULL、SET DEFAULT 动作同时指定。生成列不能和 ON DELETE 约束字句的 SET NULL、SET DEFAULT 动作同时指定。

修改和删除生成列的方法和普通列相同。删除生成列依赖的普通列，生成列被自动删除。不能改变生成列所依赖的列的类型。

生成列不能被直接写入。在 INSERT 或 UPDATE 命令中，不能为生成列指定值，但是可以指定关键字 DEFAULT。

生成列的权限控制和普通列一样。

不能为生成列指定默认值。

生成列不能作为分区键的一部分。

生成列不能和 ON UPDATE 约束字句的 CASCADE、SET NULL、SET DEFAULT 动作同时指定。生成列不能和 ON DELETE 约束字句的 SET NULL、SET DEFAULT 动作同时指定。

修改和删除生成列的方法和普通列相同。删除生成列依赖的普通列，生成列被自动删除。不能改变生成列所依赖的列的类型。

生成列不能被直接写入。在 INSERT 或 UPDATE 命令中，不能为生成列指定值，但是可

以指定关键字 DEFAULT。

生成列的权限控制和普通列一样。

列存表、内存表 MOT 不支持生成列。外表中仅 postgres_fdw 支持生成列。

- UNIQUE index_parameters | UNIQUE (column_name [, …]) index_parameters

  UNIQUE 约束表示表里的一个字段或多个字段的组合必须在全表范围内唯一。

  对于唯一约束，NULL 被认为是互不相等的。

- PRIMARY KEY index_parameters | PRIMARY KEY (column_name [, …]) index_parameters

  主键约束声明表中的一个或者多个字段只能包含唯一的非 NULL 值。

  一个表只能声明一个主键。

- DEFERRABLE | NOT DEFERRABLE

  这两个关键字设置该约束是否可推迟。一个不可推迟的约束将在每条命令之后马上检查。可推迟约束可以推迟到事务结尾使用 SET CONSTRAINTS 命令检查。缺省是 NOT DEFERRABLE。目前，UNIQUE 约束、主键约束、外键约束可以接受这个子句。所有其他约束类型都是不可推迟的。

- INITIALLY IMMEDIATE | INITIALLY DEFERRED

  如果约束是可推迟的，则这个子句声明检查约束的缺省时间。

  如果约束是 INITIALLY IMMEDIATE（缺省），则在每条语句执行之后就立即检查它；

  如果约束是 INITIALLY DEFERRED ，则只有在事务结尾才检查它。

  约束检查的时间可以用 SET CONSTRAINTS 命令修改。

- USING INDEX TABLESPACE tablespace_name

  为 UNIQUE 或 PRIMARY KEY 约束相关的索引声明一个表空间。如果没有提供这个子句，这个索引将在 default_tablespace 中创建，如果 default_tablespace 为空，将使用数据库的缺省表空间。

示例

示例 1：创建各种组合类型的二级分区表

```
CREATE TABLE list_list
(
```

```
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY LIST (month_code) SUBPARTITION BY LIST (dept_code)
(
  PARTITION p_201901 VALUES ( '201902' )
  (
    SUBPARTITION p_201901_a VALUES ( '1' ),
    SUBPARTITION p_201901_b VALUES ( '2' )
  ),
  PARTITION p_201902 VALUES ( '201903' )
  (
    SUBPARTITION p_201902_a VALUES ( '1' ),
    SUBPARTITION p_201902_b VALUES ( '2' )
  )
);
insert into list_list values('201902', '1', '1', 1);
insert into list_list values('201902', '2', '1', 1);
insert into list_list values('201902', '1', '1', 1);
insert into list_list values('201903', '2', '1', 1);
insert into list_list values('201903', '1', '1', 1);
insert into list_list values('201903', '2', '1', 1);
select * from list_list;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 1         | 1       |         1
 201902     | 2         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 1         | 1       |         1
(6 rows)

drop table list_list;
CREATE TABLE list_hash
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
```

```
)
PARTITION BY LIST (month_code) SUBPARTITION BY HASH (dept_code)
(
  PARTITION p_201901 VALUES ( '201902' )
  (
    SUBPARTITION p_201901_a,
    SUBPARTITION p_201901_b
  ),
  PARTITION p_201902 VALUES ( '201903' )
  (
    SUBPARTITION p_201902_a,
    SUBPARTITION p_201902_b
  )
);
insert into list_hash values('201902', '1', '1', 1);
insert into list_hash values('201902', '2', '1', 1);
insert into list_hash values('201902', '3', '1', 1);
insert into list_hash values('201903', '4', '1', 1);
insert into list_hash values('201903', '5', '1', 1);
insert into list_hash values('201903', '6', '1', 1);
select * from list_hash;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 4         | 1       |         1
 201903     | 5         | 1       |         1
 201903     | 6         | 1       |         1
 201902     | 2         | 1       |         1
 201902     | 3         | 1       |         1
 201902     | 1         | 1       |         1
(6 rows)

drop table list_hash;
CREATE TABLE list_range
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY LIST (month_code) SUBPARTITION BY RANGE (dept_code)
(
  PARTITION p_201901 VALUES ( '201902' )
```

```
  (
    SUBPARTITION p_201901_a values less than ('4'),
    SUBPARTITION p_201901_b values less than ('6')
  ),
  PARTITION p_201902 VALUES ( '201903' )
  (
    SUBPARTITION p_201902_a values less than ('3'),
    SUBPARTITION p_201902_b values less than ('6')
  )
);
insert into list_range values('201902', '1', '1', 1);
insert into list_range values('201902', '2', '1', 1);
insert into list_range values('201902', '3', '1', 1);
insert into list_range values('201903', '4', '1', 1);
insert into list_range values('201903', '5', '1', 1);
insert into list_range values('201903', '6', '1', 1);
ERROR:  inserted partition key does not map to any table partition
select * from list_range;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 4         | 1       |         1
 201903     | 5         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 2         | 1       |         1
 201902     | 3         | 1       |         1
(5 rows)

drop table list_range;
CREATE TABLE range_list
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY RANGE (month_code) SUBPARTITION BY LIST (dept_code)
(
  PARTITION p_201901 VALUES LESS THAN( '201903' )
  (
    SUBPARTITION p_201901_a values ('1'),
    SUBPARTITION p_201901_b values ('2')
  ),
```

```
  PARTITION p_201902 VALUES LESS THAN( '201904' )
  (
    SUBPARTITION p_201902_a values ('1'),
    SUBPARTITION p_201902_b values ('2')
  )
);
insert into range_list values('201902', '1', '1', 1);
insert into range_list values('201902', '2', '1', 1);
insert into range_list values('201902', '1', '1', 1);
insert into range_list values('201903', '2', '1', 1);
insert into range_list values('201903', '1', '1', 1);
insert into range_list values('201903', '2', '1', 1);
select * from range_list;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 2         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 1         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 1         | 1       |         1
(6 rows)

drop table range_list;
CREATE TABLE range_hash
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY RANGE (month_code) SUBPARTITION BY HASH (dept_code)
(
  PARTITION p_201901 VALUES LESS THAN( '201903' )
  (
    SUBPARTITION p_201901_a,
    SUBPARTITION p_201901_b
  ),
  PARTITION p_201902 VALUES LESS THAN( '201904' )
  (
    SUBPARTITION p_201902_a,
    SUBPARTITION p_201902_b
```

```
  )
);
insert into range_hash values('201902', '1', '1', 1);
insert into range_hash values('201902', '2', '1', 1);
insert into range_hash values('201902', '1', '1', 1);
insert into range_hash values('201903', '2', '1', 1);
insert into range_hash values('201903', '1', '1', 1);
insert into range_hash values('201903', '2', '1', 1);
select * from range_hash;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 2         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 1         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 1         | 1       |         1
(6 rows)

drop table range_hash;
CREATE TABLE range_range
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY RANGE (month_code) SUBPARTITION BY RANGE (dept_code)
(
  PARTITION p_201901 VALUES LESS THAN( '201903' )
  (
    SUBPARTITION p_201901_a VALUES LESS THAN( '2' ),
    SUBPARTITION p_201901_b VALUES LESS THAN( '3' )
  ),
  PARTITION p_201902 VALUES LESS THAN( '201904' )
  (
    SUBPARTITION p_201902_a VALUES LESS THAN( '2' ),
    SUBPARTITION p_201902_b VALUES LESS THAN( '3' )
  )
);
insert into range_range values('201902', '1', '1', 1);
insert into range_range values('201902', '2', '1', 1);
```

```
insert into range_range values('201902', '1', '1', 1);
insert into range_range values('201903', '2', '1', 1);
insert into range_range values('201903', '1', '1', 1);
insert into range_range values('201903', '2', '1', 1);
select * from range_range;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 1         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 2         | 1       |         1
 201903     | 1         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
(6 rows)

drop table range_range;
CREATE TABLE hash_list
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY hash (month_code) SUBPARTITION BY LIST (dept_code)
(
  PARTITION p_201901
  (
    SUBPARTITION p_201901_a VALUES ( '1' ),
    SUBPARTITION p_201901_b VALUES ( '2' )
  ),
  PARTITION p_201902
  (
    SUBPARTITION p_201902_a VALUES ( '1' ),
    SUBPARTITION p_201902_b VALUES ( '2' )
  )
);
insert into hash_list values('201901', '1', '1', 1);
insert into hash_list values('201901', '2', '1', 1);
insert into hash_list values('201901', '1', '1', 1);
insert into hash_list values('201903', '2', '1', 1);
insert into hash_list values('201903', '1', '1', 1);
insert into hash_list values('201903', '2', '1', 1);
```

```
select * from hash_list;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 1         | 1       |         1
 201901     | 2         | 1       |         1
 201901     | 1         | 1       |         1
 201901     | 1         | 1       |         1
(6 rows)

drop table hash_list;
CREATE TABLE hash_hash
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY hash (month_code) SUBPARTITION BY hash (dept_code)
(
  PARTITION p_201901
  (
    SUBPARTITION p_201901_a,
    SUBPARTITION p_201901_b
  ),
  PARTITION p_201902
  (
    SUBPARTITION p_201902_a,
    SUBPARTITION p_201902_b
  )
);
insert into hash_hash values('201901', '1', '1', 1);
insert into hash_hash values('201901', '2', '1', 1);
insert into hash_hash values('201901', '1', '1', 1);
insert into hash_hash values('201903', '2', '1', 1);
insert into hash_hash values('201903', '1', '1', 1);
insert into hash_hash values('201903', '2', '1', 1);
select * from hash_hash;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 2         | 1       |         1
```

```
 201903     | 2          | 1        |          1
 201903     | 1          | 1        |          1
 201901     | 2          | 1        |          1
 201901     | 1          | 1        |          1
 201901     | 1          | 1        |          1
(6 rows)

drop table hash_hash;
CREATE TABLE hash_range
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY hash (month_code) SUBPARTITION BY range (dept_code)
(
  PARTITION p_201901
  (
    SUBPARTITION p_201901_a VALUES LESS THAN ( '2' ),
    SUBPARTITION p_201901_b VALUES LESS THAN ( '3' )
  ),
  PARTITION p_201902
  (
    SUBPARTITION p_201902_a VALUES LESS THAN ( '2' ),
    SUBPARTITION p_201902_b VALUES LESS THAN ( '3' )
  )
);
insert into hash_range values('201901', '1', '1', 1);
insert into hash_range values('201901', '2', '1', 1);
insert into hash_range values('201901', '1', '1', 1);
insert into hash_range values('201903', '2', '1', 1);
insert into hash_range values('201903', '1', '1', 1);
insert into hash_range values('201903', '2', '1', 1);
select * from hash_range;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 1          | 1        |          1
 201903     | 2          | 1        |          1
 201903     | 2          | 1        |          1
 201901     | 1          | 1        |          1
 201901     | 1          | 1        |          1
```

```
 201901     | 2         | 1       |         1
(6 rows)
```

示例 2：对二级分区表进行 truncate 操作

```
CREATE TABLE list_list
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY LIST (month_code) SUBPARTITION BY LIST (dept_code)
(
  PARTITION p_201901 VALUES ( '201902' )
  (
    SUBPARTITION p_201901_a VALUES ( '1' ),
    SUBPARTITION p_201901_b VALUES ( default )
  ),
  PARTITION p_201902 VALUES ( '201903' )
  (
    SUBPARTITION p_201902_a VALUES ( '1' ),
    SUBPARTITION p_201902_b VALUES ( '2' )
  )
);
insert into list_list values('201902', '1', '1', 1);
insert into list_list values('201902', '2', '1', 1);
insert into list_list values('201902', '1', '1', 1);
insert into list_list values('201903', '2', '1', 1);
insert into list_list values('201903', '1', '1', 1);
insert into list_list values('201903', '2', '1', 1);
select * from list_list;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 1         | 1       |         1
 201902     | 2         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 1         | 1       |         1
(6 rows)

select * from list_list partition (p_201901);
```

```
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 2         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 1         | 1       |         1
(3 rows)

alter table list_list truncate partition p_201901;
select * from list_list partition (p_201901);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

select * from list_list partition (p_201902);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 1         | 1       |         1
(3 rows)

alter table list_list truncate partition p_201902;
select * from list_list partition (p_201902);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

select * from list_list;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

insert into list_list values('201902', '1', '1', 1);
insert into list_list values('201902', '2', '1', 1);
insert into list_list values('201902', '1', '1', 1);
insert into list_list values('201903', '2', '1', 1);
insert into list_list values('201903', '1', '1', 1);
insert into list_list values('201903', '2', '1', 1);
select * from list_list subpartition (p_201901_a);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 1         | 1       |         1
```

```
 201902     | 1          | 1       |          1
(2 rows)

alter table list_list truncate subpartition p_201901_a;
select * from list_list subpartition (p_201901_a);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

select * from list_list subpartition (p_201901_b);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 2          | 1       |          1
(1 row)

alter table list_list truncate subpartition p_201901_b;
select * from list_list subpartition (p_201901_b);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

select * from list_list subpartition (p_201902_a);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 1          | 1       |          1
(1 row)

alter table list_list truncate subpartition p_201902_a;
select * from list_list subpartition (p_201902_a);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

select * from list_list subpartition (p_201902_b);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 2          | 1       |          1
 201903     | 2          | 1       |          1
(2 rows)

alter table list_list truncate subpartition p_201902_b;
select * from list_list subpartition (p_201902_b);
```

```
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

select * from list_list;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

drop table list_list;
```

示例 3：对二级分区表进行 split 操作

```
CREATE TABLE list_list
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,
    dept_code  VARCHAR2 ( 30 ) NOT NULL ,
    user_no    VARCHAR2 ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY LIST (month_code) SUBPARTITION BY LIST (dept_code)
(
  PARTITION p_201901 VALUES ( '201902' )
  (
    SUBPARTITION p_201901_a VALUES ( '1' ),
    SUBPARTITION p_201901_b VALUES ( default )
  ),
  PARTITION p_201902 VALUES ( '201903' )
  (
    SUBPARTITION p_201902_a VALUES ( '1' ),
    SUBPARTITION p_201902_b VALUES ( default )
  )
);
insert into list_list values('201902', '1', '1', 1);
insert into list_list values('201902', '2', '1', 1);
insert into list_list values('201902', '1', '1', 1);
insert into list_list values('201903', '2', '1', 1);
insert into list_list values('201903', '1', '1', 1);
insert into list_list values('201903', '2', '1', 1);
select * from list_list;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
```

```
 201903     | 2          | 1       |          1
 201903     | 2          | 1       |          1
 201903     | 1          | 1       |          1
 201902     | 2          | 1       |          1
 201902     | 1          | 1       |          1
 201902     | 1          | 1       |          1
(6 rows)

select * from list_list subpartition (p_201901_a);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 1          | 1       |          1
 201902     | 1          | 1       |          1
(2 rows)

select * from list_list subpartition (p_201901_b);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 2          | 1       |          1
(1 row)

alter table list_list split subpartition p_201901_b values (2) into
(
 subpartition p_201901_b,
 subpartition p_201901_c
);
select * from list_list subpartition (p_201901_a);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 1          | 1       |          1
 201902     | 1          | 1       |          1
(2 rows)

select * from list_list subpartition (p_201901_b);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 2          | 1       |          1
(1 row)

select * from list_list subpartition (p_201901_c);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
```

```
(0 rows)

select * from list_list partition (p_201901);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 2         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 1         | 1       |         1
(3 rows)

select * from list_list subpartition (p_201902_a);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 1         | 1       |         1
(1 row)

select * from list_list subpartition (p_201902_b);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
(2 rows)

alter table list_list split subpartition p_201902_b values (3) into
(
 subpartition p_201902_b,
 subpartition p_201902_c
);
select * from list_list subpartition (p_201902_a);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 1         | 1       |         1
(1 row)

select * from list_list subpartition (p_201902_b);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
(0 rows)

select * from list_list subpartition (p_201902_c);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
```

```
 201903     | 2          | 1        |          1
 201903     | 2          | 1        |          1
(2 rows)

drop table list_list;
```

3.8.95CREATE TABLESPACE

功能描述

在数据库中创建一个新的表空间。

注意事项

系统管理员或者继承了内置角色 gs_roles_tablespace 权限的用户可以创建表空间。

不允许在一个事务块内部执行 CREATE TABLESPACE。

执行 CREATE TABLESPACE 失败，如果内部创建目录（文件）操作成功了就会产生残留的目录（文件），重新创建时需要用户手动清理表空间指定的目录下残留的内容。如果在创建过程中涉及到数据目录下的表空间软连接残留，需要先将软连接的残留文件删除，再重新执行 OM 相关操作。

CREATE TABLESPACE 不支持两阶段事务，如果部分节点执行失败，不支持回滚。

创建表空间前的准备工作参考下述参数说明。

在 HCS 等场景下一般不建议用户使用自定义的表空间。原因：用户自定义表空间通常配合主存（即默认表空间所在的存储设备，如磁盘）以外的其它存储介质使用，以隔离不同业务可以使用的 IO 资源，而在 HCS 等场景下，存储设备都是采用标准化的配置，无其它可用的存储介质，自定义表空间使用不当不利于系统长稳运行以及影响整体性能，因此建议使用默认表空间即可。

语法格式

```
CREATE TABLESPACE tablespace_name
    [ OWNER user_name ] [RELATIVE] LOCATION 'directory' [ MAXSIZE 'space_size' ]
    [with_option_clause];
```

其中普通表空间的 with_option_clause 为：

```
WITH ( filesystem= { 'systemtype '| " systemtype " | systemtype }
 [ { , address = { ' ip:port [ , ... ] ' | " ip:port [ , ... ] "} } ]
, cfgpath = { 'path '| " path " } ,storepath = { 'rootpath '| " rootpath "}
[{, random_page_cost = { 'value '| " value " | value }}]
```

```
[{,seq_page_cost = { 'value '| " value " | value }}])
```

参数说明

- tablespace_name

  要创建的表空间名称。

  表空间名称不能和 GBase 8s 中的其他表空间重名，且名称不能以“pg”开头，这样的名称留给系统表空间使用。

  取值范围：字符串，要符合标识符的命名规范。

- OWNER user_name

  指定该表空间的所有者。缺省时，新表空间的所有者是当前用户。

  只有系统管理员可以创建表空间，但是可以通过 OWNER 子句把表空间的所有权赋给其他非系统管理员。

  取值范围：字符串，已存在的用户。

- RELATIVE

  使用相对路径，LOCATION 目录是相对于各个数据库节点数据目录下的。

  目录层次：数据库节点的数据目录/pg_location/相对路径

  相对路径最多指定两层。

- LOCATION directory

  用于表空间的目录，对于目录有如下要求：

  GBase 8s 系统用户必须对该目录拥有读写权限，并且目录为空。如果该目录不存在，将由系统自动创建。

  目录必须是绝对路径，目录中不得含有特殊字符（如$、> 等）。

  目录不允许指定在数据库数据目录下。

  目录需为本地路径。

  取值范围：字符串，有效的目录。

- MAXSIZE 'space_size'

  指定表空间在单个数据库节点上的最大值。

  取值范围：字符串格式为正整数+单位，单位当前支持 K/M/G/T/P。解析后的数值以 K

为单位，且范围不能够超过 64 比特表示的有符号整数，即 1KB~9007199254740991KB。

- random_page_cost

  指定随机读取 page 的开销。

  取值范围：0~1.79769e+308。

  默认值：使用 GUC 参数 random_page_cost 的值。

- seq_page_cost

  指定顺序读取 page 的开销。

  取值范围：0~1.79769e+308。

  默认值：使用 GUC 参数 seq_page_cost 的值。

示例

```
--创建表空间。
postgres=# CREATE TABLESPACE ds_location1 RELATIVE LOCATION
'tablespace/tablespace_1';

--创建用户 joe。
postgres=# CREATE ROLE joe IDENTIFIED BY 'xxxxxxxxx';

--创建用户 jay。
postgres=# CREATE ROLE jay IDENTIFIED BY 'xxxxxxxxx';

--创建表空间，且所有者指定为用户 joe。
postgres=# CREATE TABLESPACE ds_location2 OWNER joe RELATIVE LOCATION
'tablespace/tablespace_1';

--把表空间 ds_location1 重命名为 ds_location3。
postgres=# ALTER TABLESPACE ds_location1 RENAME TO ds_location3;

--改变表空间 ds_location2 的所有者。
postgres=# ALTER TABLESPACE ds_location2 OWNER TO jay;

--删除表空间。
postgres=# DROP TABLESPACE ds_location2;
postgres=# DROP TABLESPACE ds_location3;

--删除用户。
```

```
postgres=# DROP ROLE joe;
postgres=# DROP ROLE jay;
```

相关命令

CREATE DATABASE，CREATE TABLE，CREATE INDEX，DROP TABLESPACE，ALTER TABLESPACE

优化建议

create tablespace

不建议在事务内部创建表空间。

3.8.96 CREATE TEXT SEARCH CONFIGURATION

功能描述

创建新的文本搜索配置。一个文本搜索配置声明一个能将一个字符串划分成符号的文本搜索解析器，加上可以用于确定搜索对哪些标记感兴趣的字典。

注意事项

若仅声明分析器，那么新的文本搜索配置初始没有从符号类型到词典的映射， 因此会忽略所有的单词。后面必须调用 ALTER TEXT SEARCH CONFIGURATION 命令创建映射使配置生效。如果声明了 COPY 选项，那么会自动拷贝指定的文本搜索配置的解析器、映射、配置选项等信息。

若模式名称已给出，那么文本搜索配置会在声明的模式中创建。否则会在当前模式创建。

定义文本搜索配置的用户成为其所有者。

PARSER 和 COPY 选项是互相排斥的，因为当一个现有配置被复制，其分析器配置也被复制了。

若仅声明分析器，那么新的文本搜索配置初始没有从符号类型到词典的映射， 因此会忽略所有的单词。

语法格式

```
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
    COPY = source_config
) [ WITH ( {configuration_option = value} [, ...] )];
```

参数说明

- name

  要创建的文本搜索配置的名称。该名称可以有模式修饰。

- parser_name

  用于该配置的文本搜索分析器的名称。

- source_config

  要复制的现有文本搜索配置的名称。

- configuration_option

  文本搜索配置的配置参数，主要是针对 parser_name 执行的解析器或者 source_config 隐含的解析器而言的。

  取值范围：目前共支持 default、ngram 两种类型的解析器，其中 default 类型的解析器没有对应的 configuration_option、ngram 类型解析器对应的 configuration_option 如下表所示。

表 3-27 ngram 类型解析器对应的配置参数

解析器	配置参数	参数描述	取值范围
ngram	gram_size	分词长度。	正整数，1~4 默认值：2
	punctuation_ignore	是否忽略标点符号。	true（默认值）：忽略标点符号。 false：不忽略标点符号。
	grapsymbol_ignore	是否忽略图形化字符。	true：忽略图形化字符。 false（默认值）：不忽略图形化字符。

示例

```
--创建文本搜索配置。
```

```
postgres=# CREATE TEXT SEARCH CONFIGURATION ngram2 (parser=ngram) WITH (gram_size
= 2, grapsymbol_ignore = false);

--创建文本搜索配置。
postgres=# CREATE TEXT SEARCH CONFIGURATION ngram3 (copy=ngram2) WITH (gram_size
= 2, grapsymbol_ignore = false);

--添加类型映射。
postgres=# ALTER TEXT SEARCH CONFIGURATION ngram2 ADD MAPPING FOR multisymbol WITH
simple;

--创建用户 joe。
postgres=# CREATE USER joe IDENTIFIED BY 'xxxxxxxxx';

--修改文本搜索配置的所有者。
postgres=# ALTER TEXT SEARCH CONFIGURATION ngram2 OWNER TO joe;

--修改文本搜索配置的 schema。
postgres=# ALTER TEXT SEARCH CONFIGURATION ngram2 SET SCHEMA joe;

--重命名文本搜索配置。
postgres=# ALTER TEXT SEARCH CONFIGURATION joe.ngram2 RENAME TO ngram_2;

--删除类型映射。
postgres=# ALTER TEXT SEARCH CONFIGURATION joe.ngram_2 DROP MAPPING IF EXISTS FOR
multisymbol;

--删除文本搜索配置。
postgres=# DROP TEXT SEARCH CONFIGURATION joe.ngram_2;
postgres=# DROP TEXT SEARCH CONFIGURATION ngram3;

--删除 Schema 及用户 joe。
postgres=# DROP SCHEMA IF EXISTS joe CASCADE;
postgres=# DROP ROLE IF EXISTS joe;
```

相关命令

ALTER TEXT SEARCH CONFIGURATION， DROP TEXT SEARCH CONFIGURATION

3.8.97CREATE TEXT SEARCH DICTIONARY

功能描述

创建一个新的全文检索词典。词典是一种指定在全文检索时识别特定词并处理的方法。

词典的创建依赖于预定义模板（在系统表 PG_TS_TEMPLATE 中定义），支持创建五种类型的词典，分别是 Simple、Ispell、Synonym、Thesaurus 以及 Snowball，每种类型的词典可以完成不同的任务。

注意事项

具有 SYSADMIN 权限的用户可以执行创建词典操作，创建该词典的用户自动成为其所有者。

临时模式（pg_temp）下不允许创建词典。

创建或修改词典之后，任何对于用户自定义的词典定义文件的修改，将不会影响到数据库中的词典。如果需要在数据库中使用这些修改，需使用 ALTER 语句更新对应词典的定义文件。

语法格式

```
CREATE TEXT SEARCH DICTIONARY name
     ( TEMPLATE = template_name | COPY = source_config
     [, option = value [, ...] ] );
```

参数说明

- name

  要创建的词典的名称（可指定模式名，否则在当前模式下创建）。

  取值范围：符合标识符命名规范的字符串，且最大长度不超过 63 个字符。

- template

  模板名。

  取值范围：系统表 PG_TS_TEMPLATE 中定义的模板：Simple/Synonym/Thesaurus/Ispell/Snowball。

- option

  参数名。与 template 值对应，不同的词典模板具有不同的参数列表，且与指定顺序无关。

  Simple 词典对应的 option

- STOPWORDS

  停用词表文件名，默认后缀名为 stop。停用词文件格式为一组 word 列表，每行定义一

个停用词。词典处理时，文件中的空行和空格会被忽略，并将 stopword 词组转换为小写形式。

- ACCEPT

  是否将非停用词设置为已识别。默认值为 true。

  当 Simple 词典设置参数 ACCEPT=true 时，将不会传递任何 token 给后继词典，此时建议将其放置在词典列表的最后。反之，当 ACCEPT=false 时，建议将该 Simple 词典放置在列表中的至少一个词典之前。

- FILEPATH

  词典文件所在目录。目录可以指定为本地目录和 OBS 目录(只能在安全模式下指定 OBS 目录，通过启动时添加 securitymode 选项进入安全模式）。其中，本地目录格式为“file://absolute_path”，OBS 目录格式为“obs://bucket_name/path accesskey=ak secretkey=sk region=rg”。默认值为预定义词典文件所在目录。FILEPATH 参数必须和 STOPWORDS 参数同时指定，不允许单独指定。

  Synonym 词典对应的 option

- SYNONYM

  同义词词典的定义文件名，默认后缀名为 syn。

  文件格式为一组同义词列表，每行格式为“token synonym”，即 token 和其对应的 synonym，中间以空格相连。

- CASESENSITIVE

  设置是否大小写敏感，默认值为 false，此时词典文件中的 token 和 synonym 均会转为小写形式处理。如果设置为 true，则不会进行小写转换。

- FILEPATH

  同义词词典文件所在目录。目录可以指定为本地目录和 OBS 目录两种形式（只能在安全模式下指定 OBS 目录，通过启动时添加 securitymode 选项进入安全模式）。其中，本地目录格式为“file://absolute_path”，OBS 目录格式为“obs://bucket_name/path accesskey=ak secretkey=sk region=rg”。默认值为预定义词典文件所在目录。

  Thesaurus 词典对应的 option

- DICTFILE

  词典定义文件名，默认后缀名为 ths。

文件格式为一组同义词列表，每行格式为“sample words ：indexed words”，中间冒号（:）作为短语和其替换词间的分隔符。TZ 词典处理时，如果有多个匹配的 sample words，将选择最长匹配输出。

- DICTIONARY

用于词规范化的子词典名，必须且仅能定义一个。该词典必须是已经存在的，在检查短语匹配之前使用，用于识别和规范输入文本。

如果子词典无法识别输入词，将会报错。此时，需要移除该词或者更新子词典使其识别。此外，可在 indexed words 的开头放上一个星号（*）来跳过在其上应用子词典，但是所有 sample words 必须可以被子词典识别。

如果词典文件定义的 sample words 中，含有子词典中定义的停用词，需要用问号（？）替代停用词。假设 a 和 the 是子词典中所定义的停用词，如下：

- ? one ? two ： swsw

上述同义词组定义会匹配“a one the two”以及“the one a two”，这两个短语均会被 swsw 替代输出。

- FILEPATH

词典定义文件所在目录。目录可以指定为本地目录和 OBS 目录两种形式（只能在安全模式下指定 OBS 目录，通过启动时添加 securitymode 选项进入安全模式）。其中，本地目录格式为“file://absolute_path”，OBS 目录格式为“obs://bucket_name/path accesskey=ak secretkey=sk region=rg”。默认值为预定义词典文件所在目录。

Ispell 词典

- DICTFILE

词典定义文件名，默认后缀名为 dict。

- AFFFILE

词缀文件名，默认后缀名为 affix。

- STOPWORDS

停用词文件名，默认后缀名为 stop，文件格式要求与 Simple 类型词典的停用词文件相同。

- FILEPATH

词典文件所在目录。可以指定为本地目录和 OBS 目录两种形式（只能在安全模式下指定 OBS 目录，通过启动时添加 securitymode 选项进入安全模式）。其中，本地目录格式为“file://absolute_path”，OBS 目录格式为“obs://bucket_name/path accesskey=ak secretkey=sk region=rg”。默认值为预定义词典文件所在目录。

Snowball 词典

- LANGUAGE

语言名，标识使用哪种语言的词干分析算法。算法按照对应语言中的拼写规则，缩减输入词的常见变体形式为一个基础词或词干。

- STOPWORDS

停用词表文件名，默认后缀名为 stop，文件格式要求与 Simple 类型词典的停用词文件相同。

- FILEPATH

词典定义文件所在目录。可以指定为本地目录或者 OBS 目录（只能在安全模式下指定 OBS 目录，通过启动时添加 securitymode 选项进入安全模式）。其中，本地目录格式为“file://absolute_path”，OBS 目录格式为“obs://bucket_name/path accesskey=ak secretkey=sk region=rg”。默认值为预定义词典文件所在目录。FILEPATH 参数必须和 STOPWORDS 参数同时指定，不允许单独指定。

说明：词典定义文件的文件名仅支持小写字母、数据、下划线混合。

- value

参数值。如果不是简单的标识符或数字，则参数值必须加单引号（标示符和数字同样可以加上单引号）。

示例

请参见配置示例一节的示例。

相关命令

ALTER TEXT SEARCH DICTIONARY，CREATE TEXT SEARCH DICTIONARY

3.8.98 CREATE TRIGGER

功能描述

创建一个触发器。 触发器将与指定的表或视图关联，并在特定条件下执行指定的函数。

注意事项

当前仅支持在普通行存表上创建触发器，不支持在列存表、临时表、unlogged 表等类型表上创建触发器。

如果为同一事件定义了多个相同类型的触发器，则按触发器的名称字母顺序触发它们。

触发器常用于多表间数据关联同步场景，对 SQL 执行性能影响较大，不建议在大数据量同步及对性能要求高的场景中使用。

语法格式

```
CREATE [ CONSTRAINT ] TRIGGER trigger_name { BEFORE | AFTER | INSTEAD OF } { event
[ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY
DEFERRED } }
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments );
```

其中 event 包含以下几种：

```
    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

参数说明

- CONSTRAINT

可选项，指定此参数将创建约束触发器，即触发器作为约束来使用。除了可以使用 SET CONSTRAINTS 调整触发器触发的时间之外，这与常规触发器相同。 约束触发器必须是 AFTER ROW 触发器。

- trigger_name

触发器名称，该名称不能限定模式，因为触发器自动继承其所在表的模式，且同一个表的触发器不能重名。 对于约束触发器，使用 SET CONSTRAINTS 修改触发器行为时也使用此名称。

取值范围：符合标识符命名规范的字符串，且最大长度不超过 63 个字符。

- BEFORE

  触发器函数是在触发事件发生前执行。

- AFTER

  触发器函数是在触发事件发生后执行，约束触发器只能指定为 AFTER。

- INSTEAD OF

  触发器函数直接替代触发事件。

- event

  启动触发器的事件，取值范围包括：INSERT、UPDATE、DELETE 或 TRUNCATE，也可以通过 OR 同时指定多个触发事件。

  对于 UPDATE 事件类型，可以使用下面语法指定列：

- UPDATE OF column_name1 [, column_name2 ...]

  表示当这些列作为 UPDATE 语句的目标列时，才会启动触发器，但是 INSTEAD OF UPDATE 类型不支持指定列信息。

- table_name

  需要创建触发器的表名称。

  取值范围：数据库中已经存在的表名称。

- referenced_table_name

  约束引用的另一个表的名称。 只能为约束触发器指定，常见于外键约束。

  取值范围：数据库中已经存在的表名称。

- DEFERRABLE | NOT DEFERRABLE

  约束触发器的启动时机，仅作用于约束触发器。这两个关键字设置该约束是否可推迟。详细介绍请参见 CREATE TABLE。

- INITIALLY IMMEDIATE | INITIALLY DEFERRED

  如果约束是可推迟的，则这个子句声明检查约束的缺省时间，仅作用于约束触发器。详细介绍请参见 CREATE TABLE。

- FOR EACH ROW | FOR EACH STATEMENT

  触发器的触发频率。

FOR EACH ROW 是指该触发器是受触发事件影响的每一行触发一次。

FOR EACH STATEMENT 是指该触发器是每个 SQL 语句只触发一次。

未指定时默认值为 FOR EACH STATEMENT。约束触发器只能指定为 FOR EACH ROW。

- condition

决定是否实际执行触发器函数的条件表达式。当指定 WHEN 时，只有在条件返回 true 时才会调用该函数。

在 FOR EACH ROW 触发器中，WHEN 条件可以通过分别写入 OLD.column_name 或 NEW.column_name 来引用旧行或新行值的列。 当然，INSERT 触发器不能引用 OLD 和 DELETE 触发器不能引用 NEW。

INSTEAD OF 触发器不支持 WHEN 条件。

WHEN 表达式不能包含子查询。

对于约束触发器，WHEN 条件的评估不会延迟，而是在执行更新操作后立即发生。 如果条件返回值不为 true，则触发器不会排队等待延迟执行。

- function_name

用户定义的函数，必须声明为不带参数并返回类型为触发器，在触发器触发时执行。

- arguments

执行触发器时要提供给函数的可选的以逗号分隔的参数列表。参数是文字字符串常量，简单的名称和数字常量也可以写在这里，但它们都将被转换为字符串。 请检查触发器函数的实现语言的描述，以了解如何在函数内访问这些参数。

说明：关于触发器种类：

INSTEAD OF 的触发器必须标记为 FOR EACH ROW，并且只能在视图上定义。

BEFORE 和 AFTER 触发器作用在视图上时，只能标记为 FOR EACH STATEMENT。

TRUNCATE 类型触发器仅限 FOR EACH STATEMENT。

表 3-28 表和视图上支持的触发器种类：

触发时机	触发事件	行级	语句级

触发时机	触发事件	行级	语句级
BEFORE	INSERT/UPDATE/DELETE	表	表和视图
	TRUNCATE	不支持	表
AFTER	INSERT/UPDATE/DELETE	表	表和视图
	TRUNCATE	不支持	表
INSTEAD OF	INSERT/UPDATE/DELETE	视图	不支持
	TRUNCATE	不支持	不支持

表 3-29 PLPGSQL 类型触发器函数特殊变量：

变量名	变量含义
NEW	INSERT 及 UPDATE 操作涉及 tuple 信息中的新值，对 DELETE 为空。
OLD	UPDATE 及 DELETE 操作涉及 tuple 信息中的旧值，对 INSERT 为空。
TG_NAME	触发器名称。
TG_WHEN	触发器触发时机（BEFORE/AFTER/INSTEAD OF）。

变量名	变量含义
TG_LEVEL	触发频率（ROW/STATEMENT）。
TG_OP	触发操作（INSERT/UPDATE/DELETE/TRUNCATE）。
TG_RELID	触发器所在表 OID。
TG_RELNAME	触发器所在表名（已废弃，现用 TG_TABLE_NAME 替代）。
TG_TABLE_NAME	触发器所在表名。
TG_TABLE_SCHEMA	触发器所在表的 SCHEMA 信息。
TG_NARGS	触发器函数参数个数。
TG_ARGV[]	触发器函数参数列表。

示例

```
--创建源表及触发表
postgres=# CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);
postgres=# CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);

--创建触发器函数
postgres=# CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
           $$
           DECLARE
           BEGIN
                    INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2,
NEW.id3);
                    RETURN NEW;
```

```
            END
            $$ LANGUAGE PLPGSQL;

postgres=# CREATE OR REPLACE FUNCTION tri_update_func() RETURNS TRIGGER AS
            $$
            DECLARE
            BEGIN
                    UPDATE test_trigger_des_tbl SET id3 = NEW.id3 WHERE
id1=OLD.id1;
                    RETURN OLD;
            END
            $$ LANGUAGE PLPGSQL;

postgres=# CREATE OR REPLACE FUNCTION TRI_DELETE_FUNC() RETURNS TRIGGER AS
            $$
            DECLARE
            BEGIN
                    DELETE FROM test_trigger_des_tbl WHERE id1=OLD.id1;
                    RETURN OLD;
            END
            $$ LANGUAGE PLPGSQL;

--创建 INSERT 触发器
postgres=# CREATE TRIGGER insert_trigger
            BEFORE INSERT ON test_trigger_src_tbl
            FOR EACH ROW
            EXECUTE PROCEDURE tri_insert_func();

--创建 UPDATE 触发器
postgres=# CREATE TRIGGER update_trigger
            AFTER UPDATE ON test_trigger_src_tbl
            FOR EACH ROW
            EXECUTE PROCEDURE tri_update_func();

--创建 DELETE 触发器
postgres=# CREATE TRIGGER delete_trigger
            BEFORE DELETE ON test_trigger_src_tbl
            FOR EACH ROW
            EXECUTE PROCEDURE tri_delete_func();

--执行 INSERT 触发事件并检查触发结果
postgres=# INSERT INTO test_trigger_src_tbl VALUES(100,200,300);
```

```
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效。

--执行 UPDATE 触发事件并检查触发结果
postgres=# UPDATE test_trigger_src_tbl SET id3=400 WHERE id1=100;
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效

--执行 DELETE 触发事件并检查触发结果
postgres=# DELETE FROM test_trigger_src_tbl WHERE id1=100;
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效

--修改触发器
postgres=# ALTER TRIGGER delete_trigger ON test_trigger_src_tbl RENAME TO
delete_trigger_renamed;

--禁用 insert_trigger 触发器
postgres=# ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER insert_trigger;

--禁用当前表上所有触发器
postgres=# ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER ALL;

--删除触发器
postgres=# DROP TRIGGER insert_trigger ON test_trigger_src_tbl;
postgres=# DROP TRIGGER update_trigger ON test_trigger_src_tbl;
postgres=# DROP TRIGGER delete_trigger_renamed ON test_trigger_src_tbl;
```

相关命令

ALTER TRIGGER，DROP TRIGGER，ALTER TABLE

3.8.99 CREATE TYPE

功能描述

在当前数据库中定义一种新的数据类型。定义数据类型的用户将成为该数据类型的拥有者。类型只适用于行存表

有五种形式的 CREATE TYPE，分别为：复合类型、基本类型、shell 类型、枚举类型和集合类型。

- 复合类型

复合类型由一个属性名和数据类型的列表指定。如果属性的数据类型是可排序的，也可以指定该属性的排序规则。复合类型本质上和表的行类型相同，但是如果只想定义一种类型，使用 CREATE TYPE 避免了创建一个实际的表。单独的复合类型也是很有用的，例如可以作为函数的参数或者返回类型。

为了能够创建复合类型，必须拥有在其所有属性类型上的 USAGE 特权。

- 基本类型

用户可以自定义一种新的基本类型（标量类型）。通常来说这些函数必须是底层语言所编写。

- shell 类型

shell 类型是一种用于后面要定义的类型的占位符，通过发出一个不带除类型名之外其他参数的 CREATE TYPE 命令可以创建这种类型。在创建基本类型时，需要 shell 类型作为一种向前引用。

- 枚举类型

由若干个标签构成的列表，每一个标签值都是一个非空字符串，且字符串长度必须不超过 63 个字节。

- 集合类型

类似数组，但是没有长度限制，主要在存储过程中使用。

被授予 CREATE ANY TYPE 权限的用户，可以在 public 模式和用户模式下创建类型。

注意事项

如果给定一个模式名，那么该类型将被创建在指定的模式中。否则它会被创建在当前模式中。类型名称必须与同一个模式中任何现有的类型或者域相区别（因为表具有相关的数据类型，类型名称也必须与同一个模式中任何现有表的名称不同）。

语法格式

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name (
    INPUT = input_function,
```

```
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength |VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name

CREATE TYPE name AS ENUM
    ( [ 'lable' [, ... ] ] )

CREATE TYPE name AS TABLE OF data_type
```

参数说明

复合类型

- name

  要创建的类型的名称（可以被模式限定）。

- attribute_name

  复合类型的一个属性（列）的名称。

- data_type

  要成为复合类型的一个列的现有数据类型的名称。可以使用%ROWTYPE 间接引用表的类型，或者使用%TYPE 间接引用表或复合类型中某一列的类型。

- collation

要关联到复合类型的一列的现有排序规则的名称。排序规则可以使用“select * from pg_collation”命令从 pg_collation 系统表中查询，默认的排序规则为查询结果中以 default 开始的行。

基本类型

自定义基本类型时，参数可以以任意顺序出现，input_function 和 output_function 为必选参数，其它为可选参数。

- input_function

将数据从类型的外部文本形式转换为内部形式的函数名。

输入函数可以被声明为有一个 cstring 类型的参数，或者有三个类型分别为 cstring、oid、integer 的参数。

cstring 参数是以 C 字符串存在的输入文本。

oid 参数是该类型自身的 OID（对于数组类型则是其元素类型的 OID）。

integer 参数是目标列的 typmod（如果知道，不知道则将传递 -1）。

输入函数必须返回一个该数据类型本身的值。通常，一个输入函数应该被声明为 STRICT。 如果不是这样，在读到一个 NULL 输入值时，调用输入函数时第一个参数会是 NULL。在这种情况下，该函数必须仍然返回 NULL，除非调用函数发生了错误（这种情况主要是想支持域输入函数，域输入函数可能需要拒绝 NULL 输入）。

说明：输入和输出函数能被声明为具有新类型的结果或参数是因为：必须在创建新类型之前创建这两个函数。而新类型应该首先被定义为一种 shell type，它是一种占位符类型，除了名称和拥有者之外它没有其他属性。这可以通过不带额外参数的命令 CREATE TYPE name 做到。然后用 C 写的 I/O 函数可以被定义为引用这种 shell type。最后，用带有完整定义的 CREATE TYPE 把该 shell type 替换为一个完全的、合法的类型定义，之后新类型就可以正常使用了。

- output_function

将数据从类型的内部形式转换为外部文本形式的函数名。

输出函数必须被声明为有一个新数据类型的参数。输出函数必须返回类型 cstring。对于 NULL 值不会调用输出函数。

- receive_function

可选参数。将数据从类型的外部二进制形式转换成内部形式的函数名。

如果没有该函数，该类型不能参与到二进制输入中。二进制表达转换成内部形式代价更低，然而却更容易移植（例如，标准的整数数据类型使用网络字节序作为外部二进制表达，而内部表达是机器本地的字节序）。receive_function 应该执行足够的检查以确保该值是有效的。

接收函数可以被声明为有一个 internal 类型的参数，或者有三个类型分别为 internal、oid、integer 的参数。

internal 参数是一个指向 StringInfo 缓冲区的指针，其中保存着接收到的字节串。

oid 和 integer 参数和文本输入函数的相同。

接收函数必须返回一个该数据类型本身的值。通常，一个接收函数应该被声明为 STRICT。如果不是这样，在读到一个 NULL 输入值时调用接收函数时第一个参数会是 NULL。在这种情况下，该函数必须仍然返回 NULL，除非接收函数发生了错误（这种情况主要是想支持域接收函数，域接收函数可能需要拒绝 NULL 输入）。

- send_function

  可选参数。将数据从类型的内部形式转换为外部二进制形式的函数名。

  如果没有该函数，该类型将不能参与到二进制输出中。发送函数必须被声明为有一个新数据类型的参数。发送函数必须返回类型 bytea。对于 NULL 值不会调用发送函数。

- type_modifier_input_function

  可选参数。将类型的修饰符数组转换为内部形式的函数名。

- type_modifier_output_function

  可选参数。将类型的修饰符的内部形式转换为外部文本形式的函数名。

说明：如果该类型支持修饰符（附加在类型声明上的可选约束，例如，char(5)或 numeric(30,2)），则需要可选的 type_modifier_input_function 以及 type_modifier_output_function。GBase 8s 允许用户定义的类型有一个或者多个简单常量或者标识符作为修饰符。不过，为了存储在系统目录中，该信息必须能被打包到一个非负整数值中。所声明的修饰符会被以 cstring 数组的形式传递给 type_modifier_input_function。type_modifier_input_function 必须检查该值的合法性（如果值错误就抛出一个错误），如果值正确，要返回一个非负 integer 值，该值将被存储在“typmod”列中。如果类型没有 type_modifier_input_function 则类型修饰符将被拒绝。type_modifier_output_function 把内部

的整数 typmod 值转换回正确的形式用于用户显示。type_modifier_output_function 必须返回一个 cstring 值，该值就是追加到类型名称后的字符串。例如，numeric 的函数可能会返回(30,2)。如果默认的显示格式就是只把存储的 typmod 整数值放在圆括号内，则允许省略 type_modifier_output_function。

- analyze_function

  可选参数。为该数据类型执行统计分析的函数名的可选参数。

  默认情况下，如果该类型有一个默认的 B-tree 操作符类，ANALYZE 将尝试用类型的“equals”和“less-than”操作符来收集统计信息。这种行为对于非标量类型并不合适，因此可以通过指定一个自定义分析函数来覆盖这种行为。分析函数必须被声明为有一个类型为 internal 的参数，并且返回一个 boolean 结果。

- internallength

  可选参数。一个数字常量，用于指定新类型的内部表达的字节长度。默认为变长。

  虽然只有 I/O 函数和其他为该类型创建的函数才知道新类型的内部表达的细节， 但是内部表达的一些属性必须被向 GBase 8s 声明。其中最重要的是 internallength。基本数据类型可以是定长的（这种情况下 internallength 是一个正整数）或者是变长的（把 internallength 设置为 VARIABLE，在内部通过把 typlen 设置为-1 表示）。所有变长类型的内部表达都必须以一个 4 字节整数开始，internallength 定义了总长度。

- PASSEDBYVALUE

  可选参数。表示这种数据类型的值需要被传值而不是传引用。传值的类型必须是定长的，并且它们的内部表达不能超过 Datum 类型（某些机器上是 4 字节，其他机器上是 8 字节）的尺寸。

- alignment

  可选参数。该参数指定数据类型的存储对齐需求。如果被指定，必须是 char、int2、int4 或者 double。默认是 int4。

  允许的值等同于以 1、2、4 或 8 字节边界对齐。要注意变长类型的 alignment 参数必须至少为 4，因为它们需要包含一个 int4 作为它们的第一个组成部分。

- storage

  可选参数。该数据类型的存储策略。

  如果被指定，必须是 plain、external、extended 或者 main。 默认是 plain。

plain 指定该类型的数据将总是被存储在线内并且不会被压缩。(对定长类型只允许 plain)

extended 指定系统将首先尝试压缩一个长的数据值，并且将在数据仍然太长的情况下把值移出主表行。

external 允许值被移出主表， 但是系统将不会尝试对它进行压缩。

main 允许压缩，但是不鼓励把值移出主表（如果没有其他办法让行的大小变得合适，具有这种存储策略的数据项仍将被移出主表，但比起 extended 以及 external 项来，这种存储策略的数据项会被优先考虑保留在主表中）。

除 plain 之外所有的 storage 值都暗示该数据类型的函数能处理被 TOAST 过的值。指定的值仅仅是决定一种可 TOAST 数据类型的列的默认 TOAST 存储策略，用户可以使用 ALTER TABLE SET STORAGE 为列选取其他策略。

- like_type

可选参数。与新类型具有相同表达的现有数据类型的名称。会从这个类型中复制 internallength、 passedbyvalue、 alignment 以及 storage 的值（ 除非在这个 CREATE TYPE 命令的其他地方用显式说明覆盖）。

当新类型的低层实现是以一种现有的类型为参考时，用这种方式指定表达特别有用。

- category

可选参数。这种类型的分类码（一个 ASCII 字符）。 默认是“用户定义类型”的'U'。为了创建自定义分类， 也可以选择其他 ASCII 字符。

- preferred

可选参数。如果这种类型是其类型分类中的优先类型则为 TRUE，否则为 FALSE。默认为假。在一个现有类型分类中创建一种新的优先类型要非常谨慎， 因为这可能会导致很大的改变。

说明：category 和 preferred 参数可以被用来帮助控制在混淆的情况下应用哪一种隐式造型。每一种数据类型都属于一个用单个 ASCII 字符命名的分类，并且每一种类型可以是其所属分类中的“首选”。当有助于解决重载函数或操作符时，解析器将优先造型到首选类型（但是只能从同类的其他类型造型）。对于没有隐式转换到或来自任意其他类型的类型，让这些设置保持默认即可。不过，对于有隐式转换的相关类型的组，把它们都标记为属于同一个类别并且选择一种或两种“最常用”的类型作为该类别的首选通常是很有用的。在把一种用户定义的类型增加到一个现有的内建类别（例如，数字或者字符串类型）中时，

category 参数特别有用。不过，也可以创建新的全部是用户定义类型的类别。对这样的类别，可选择除大写字母之外的任何 ASCII 字符。

- default

  可选参数。数据类型的默认值。如果被省略，默认值是空。

  如果用户希望该数据类型的列被默认为某种非空值，可以指定一个默认值。默认值可以用 DEFAULT 关键词指定（这样一个默认值可以被附加到一个特定列的显式 DEFAULT 子句覆盖）。

- element

  可选参数。被创建的类型是一个数组，element 指定了数组元素的类型。例如，要定义一个 4 字节整数的数组（int4），应指定 ELEMENT = int4。

- delimiter

  可选参数。指定这种类型组成的数组中分隔值的定界符。

  可以把 delimiter 设置为一个特定字符，默认的定界符是逗号（,）。注意定界符是与数组元素类型相关的，而不是数组类型本身相关。

- collatable

  可选参数。如果这个类型的操作可以使用排序规则信息，则为 TRUE。默认为 FALSE。

  如果 collatable 为 TRUE，这种类型的列定义和表达式可能通过使用 COLLATE 子句携带有排序规则信息。在该类型上操作的函数的实现负责真正利用这些信息，仅把类型标记为可排序的并不会让它们自动地去使用这类信息。

- lable

  可选参数。与枚举类型的一个值相关的文本标签，其值为长度不超过 63 个字符的非空字符串。

说明：在创建用户定义类型的时候，GBase 8s 会自动创建一个与之关联的数组类型，其名称由该元素类型的名称前缀一个下划线组成。

示例

```
--创建一种复合类型，建表并插入数据以及查询。
postgres=# CREATE TYPE compfoo AS (f1 int, f2 text);
postgres=# CREATE TABLE t1_compfoo(a int, b compfoo);
postgres=# CREATE TABLE t2_compfoo(a int, b compfoo);
```

```
postgres=# INSERT INTO t1_compfoo values(1,(1,'demo'));
postgres=# INSERT INTO t2_compfoo select * from t1_compfoo;
postgres=# SELECT (b).f1 FROM t1_compfoo;
postgres=# SELECT * FROM t1_compfoo t1 join t2_compfoo t2 on (t1.b).f1=(t1.b).f1;

--重命名数据类型。
postgres=# ALTER TYPE compfoo RENAME TO compfoo1;

--要改变一个用户定义类型 compfoo1 的所有者为 usr1。
postgres=# CREATE USER usr1 PASSWORD 'xxxxxxxxx';
postgres=# ALTER TYPE compfoo1 OWNER TO usr1;

--把用户定义类型 compfoo1 的模式改变为 usr1。
postgres=# ALTER TYPE compfoo1 SET SCHEMA usr1;

--给一个数据类型增加一个新的属性。
postgres=# ALTER TYPE usr1.compfoo1 ADD ATTRIBUTE f3 int;

--删除 compfoo1 类型。
postgres=# DROP TYPE usr1.compfoo1 cascade;

--删除相关表和用户。
postgres=# DROP TABLE t1_compfoo;
postgres=# DROP TABLE t2_compfoo;
postgres=# DROP SCHEMA usr1;
postgres=# DROP USER usr1;

--创建一个枚举类型。
postgres=# CREATE TYPE bugstatus AS ENUM ('create', 'modify', 'closed');

--添加一个标签值。
postgres=# ALTER TYPE bugstatus ADD VALUE IF NOT EXISTS 'regress' BEFORE 'closed';

--重命名一个标签值。
postgres=# ALTER TYPE bugstatus RENAME VALUE 'create' TO 'new';

--创建一个集合类型
postgres=# CREATE TYPE compfoo_table AS TABLE OF compfoo;
```

相关命令

ALTER TYPE，DROP TYPE

3.8.100CREATE USER

功能描述

创建一个用户。

注意事项

通过 CREATE USER 创建的用户，默认具有 LOGIN 权限。

通过 CREATE USER 创建用户的同时，系统会在执行该命令的数据库中，为该用户创建一个同名的 SCHEMA。

系统管理员在普通用户同名 schema 下创建的对象，所有者为 schema 的同名用户（非系统管理员）。

语法格式

```
CREATE USER user_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ]
{ PASSWORD | IDENTIFIED BY } { 'password' [EXPIRED] | DISABLE };
```

其中 option 子句用于设置权限及属性等信息。

```
{SYSADMIN | NOSYSADMIN}
    | {MONADMIN | NOMONADMIN}
    | {OPRADMIN | NOOPRADMIN}
    | {POLADMIN | NOPOLADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | {PERSISTENCE | NOPERSISTENCE}
    | CONNECTION LIMIT connlimit
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
```

```
    | NODE GROUP logic_cluster_name
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
    | DEFAULT TABLESPACE tablespace_name
    | PROFILE DEFAULT
    | PROFILE profile_name
    | PGUSER
```

参数说明

- user_name

  用户名称。

  取值范围：字符串，要符合标识符的命名规范。且最大长度不超过 63 个字符。

- password

  登录密码。

  密码规则如下：

  - 密码默认不少于 8 个字符。
  - 不能与用户名及用户名倒序相同。
  - 至少包含大写字母（A-Z）、小写字母（a-z）、数字（0-9）、非字母数字字符（限定为~!@#$%^&*()-_=+验\|[{}];:,<.>/?）四类字符中的三类字符。
  - 密码也可以是符合格式要求的密文字符串，这种情况主要用于用户数据导入场景，不推荐用户直接使用。如果直接使用密文密码，用户需要知道密文密码对应的明文，并且保证明文密码复杂度，数据库不会校密文密码复杂度，直接使用密文密码的安全性由用户保证。
  - 创建用户时，应当使用双引号或单引号将用户密码括起来。

  取值范围：字符串。

  CREATE USER 的其他参数值请参考 CREATE ROLE。

示例

```
--创建用户 jim，登录密码为 xxxxxxxxx。
```

```
postgres=# CREATE USER jim PASSWORD 'xxxxxxxxx';

--下面语句与上面的等价。
postgres=# CREATE USER kim IDENTIFIED BY 'xxxxxxxxx';

--如果创建有“创建数据库”权限的用户，则需要加 CREATEDB 关键字。
postgres=# CREATE USER dim CREATEDB PASSWORD 'xxxxxxxxx';

--将用户 jim 的登录密码由 xxxxxxxxx 修改为 Abcd@123。
postgres=# ALTER USER jim IDENTIFIED BY 'Abcd@123' REPLACE 'xxxxxxxxx';

--为用户 jim 追加 CREATEROLE 权限。
postgres=# ALTER USER jim CREATEROLE;

--将 enable_seqscan 的值设置为 on， 设置成功后，在下一会话中生效。
postgres=# ALTER USER jim SET enable_seqscan TO on;

--重置 jim 的 enable_seqscan 参数。
postgres=# ALTER USER jim RESET enable_seqscan;

--锁定 jim 帐户。
postgres=# ALTER USER jim ACCOUNT LOCK;

--删除用户。
postgres=# DROP USER kim CASCADE;
postgres=# DROP USER jim CASCADE;
postgres=# DROP USER dim CASCADE;
```

相关命令

ALTER USER，CREATE ROLE，DROP USER

3.8.101CREATE USER MAPPING

功能描述

定义一个用户到一个外部服务器的新映射。

注意事项

当在 OPTIONS 中出现 password 选项时，需要保证 GBase 8s 数据库集群每个节点的 $GAUSSHOME/bin 目录下存在 usermapping.key.cipher 和 usermapping.key.rand 文件，如果不存在这两个文件，请使用 gs_guc 工具生成并发布到每个节点的$GAUSSHOME/bin 目录下。

语法格式

```
CREATE USER MAPPING FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

参数说明

- user_name

  要映射到外部服务器的一个现有用户的名称。

  CURRENT_USER 和 USER 匹配当前用户的名称。 当 PUBLIC 被指定时，一个公共映射会被创建，当没有特定用户的映射可用时将会使用它。

- server_name

  将为其创建用户映射的现有服务器的名称。

- OPTIONS ({ option_name ' value ' } [, ···])

  这个子句指定用户映射的选项。这些选项通常定义该映射实际的用户名和口令。选项名必须唯一。允许的选项名和值与该服务器的外部数据包装器有关。

  说明：

  用户的口令会加密后保存到系统表 PG_USER_MAPPING 中，加密时需要使用 usermapping.key.cipher 和 usermapping.key.rand 作为加密密码文件和加密因子。首次使用前需要通过如下命令创建这两个文件，并将这两个文件放入各节点目录$GAUSSHOME/bin，且确保具有读权限。gs_ssh 工具可以协助您快速将文件放入各节点对应目录下。

```
gs_ssh -c "gs_guc generate -o usermapping -S default -D $GAUSSHOME/bin"
```

  其中-S 参数指定 default 时会随机生成密码，用户也可为-S 参数指定密码，此密码用于保证生成密码文件的安全性和唯一性，用户无需保存或记忆。其他参数详见《GBase 8s V8.8.5 5.0.0_工具参考手册》中 gs_guc 工具说明。

- oracle_fdw 支持的 options

  - user

  oracle server 的用户名。

  - password

  oracle 用户对应的密码。

- mysql_fdw 支持的 options
  - username

  MySQL Server/MariaDB 的用户名。

  - password

  MySQL Server/MariaDB 用户对应的密码。

- postgres_fdw 支持的 options
  - user

  远端数据库的用户名。

  - password

  远端用户对应的密码。

  说明：

  在后台会对用户输入的 password 加密以保证安全性。该加密所需密钥文件需要生成并发布到每个节点的$GAUSSHOME/bin 目录下。password 不应当包含'encryptOpt'前缀，否则会被认为是加密后的密文。

相关命令

ALTER USER MAPPING，DROP USER MAPPING

3.8.102CREATE VIEW

功能描述

创建一个视图。视图与基本表不同，是一个虚拟的表。数据库中仅存放视图的定义，而不存放视图对应的数据，这些数据仍存放在原来的基本表中。若基本表中的数据发生变化，从视图中查询出的数据也随之改变。从这个意义上讲，视图就像一个窗口，透过它可以看到数据库中用户感兴趣的数据及变化。

注意事项

被授予 CREATE ANY TABLE 权限的用户，可以在 public 模式和用户模式下创建视图。

语法格式

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW view_name [ ( column_name [, ...] ) ]
    [ WITH ( {view_option_name [= view_option_value]} [, ... ] ) ]
    AS query;
```

说明：创建视图时使用 WITH(security_barrier)可以创建一个相对安全的视图，避免攻击者利用低成本函数的 RAISE 语句打印出隐藏的基表数据。

参数说明

- OR REPLACE

  如果视图已存在，则重新定义。

- TEMP | TEMPORARY

  创建临时视图。

- view_name

  要创建的视图名称。可以用模式修饰。

  取值范围：字符串，符合标识符命名规范。

- column_name

  可选的名称列表，用作视图的字段名。如果没有给出，字段名取自查询中的字段名。

  取值范围：字符串，符合标识符命名规范。

- view_option_name [= view_option_value]

  该子句为视图指定一个可选的参数。

  目前 view_option_name 支持的参数仅有 security_barrier，当 VIEW 试图提供行级安全时，应使用该参数。

  取值范围：Boolean 类型，TRUE、FALSE

- query

  为视图提供行和列的 SELECT 或 VALUES 语句。

示例

```
--创建字段 spcname 为 pg_default 组成的视图。
postgres=# CREATE VIEW myView AS
    SELECT * FROM pg_tablespace WHERE spcname = 'pg_default';

--查看视图。
postgres=# SELECT * FROM myView ;

--删除视图 myView。
```

```
postgres=# DROP VIEW myView;
```

相关命令

ALTER VIEW，DROP VIEW

3.8.103CREATE WEAK PASSWORD DICTIONARY

功能描述

向 gs_global_config 表中插入一个或者多个弱口令。

注意事项

只有初始用户、系统管理员和安全管理员拥有权限执行本语法。

弱口令字典中的口令存放在 gs_global_config 系统表中。

弱口令字典默认为空，用户通过本语法可以新增一条或多条弱口令。

当用户尝试通过本语法插入 gs_global_config 表中已存在的弱口令时，会只在表中保留一条该弱口令。

语法格式

```
CREATE WEAK PASSWORD DICTIONARY
       [WITH VALUES] ( {'weak_password'} [, ...] );
```

参数说明

- weak_password

  弱口令。

  范围：字符串。

示例

```
--向 gs_global_config 系统表中插入单个弱口令。
postgres=# CREATE WEAK PASSWORD DICTIONARY WITH VALUES ('password1');

--向 gs_global_config 系统表中插入多个弱口令。
postgres=# CREATE WEAK PASSWORD DICTIONARY WITH VALUES
('password2'),('password3');

--清空 gs_global_config 系统表中所有弱口令。
postgres=# DROP WEAK PASSWORD DICTIONARY;
```

```
--查看现有弱口令。
postgres=# SELECT * FROM gs_global_config WHERE NAME LIKE 'weak_password';
```

相关命令

DROP WEAK PASSWORD DICTIONARY

3.8.104CURSOR

功能描述

CURSOR 命令定义一个游标，用于在一个大的查询里面检索少数几行数据。

为了处理 SQL 语句，存储过程进程分配一段内存区域来保存上下文联系。游标是指向上下文区域的句柄或指针。借助游标，存储过程可以控制上下文区域的变化。

注意事项

游标命令只能在事务块里使用。

通常游标和 SELECT 一样返回文本格式。因为数据在系统内部是用二进制格式存储的，系统必须对数据做一定转换以生成文本格式。一旦数据是以文本形式返回，客户端应用需要把它们转换成二进制进行操作。使用 FETCH 语句，游标可以返回文本或二进制格式。

应该小心使用二进制游标。文本格式一般都比对应的二进制格式占用的存储空间大。二进制游标返回内部二进制形态的数据，可能更易于操作。如果想以文本方式显示数据，则以文本方式检索会为用户节约很多客户端的工作。比如，如果查询从某个整数列返回 1，在缺省的游标里将获得一个字符串 1，但在二进制游标里将得到一个 4 字节的包含该数值内部形式的数值（大端顺序）。

语法格式

```
CURSOR cursor_name
    [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    FOR query ;
```

参数说明

- cursor_name

  将要创建的游标名。

  取值范围：遵循数据库对象命名规范。

- BINARY

  指明游标以二进制而不是文本格式返回数据。

- [NO] SCROLL

  SCROLL：若指定，那么游标可以反向滚动。

  NO SCROLL：声明该游标不能用于以倒序的方式检索数据行。

  未声明:根据执行计划的不同,自动判断该游标是否可以用于以倒序的方式检索数据行。

- WITH HOLD | WITHOUT HOLD

  声明当创建游标的事务结束后，游标是否能继续使用。

  WITH HOLD：声明该游标在创建它的事务结束后仍可继续使用。

  WITHOUT HOLD：声明该游标在创建它的事务之外不能再继续使用，此游标将在事务结束时被自动关闭。

  如果不指定 WITH HOLD 或 WITHOUT HOLD，默认行为是 WITHOUT HOLD。

  跨节点事务不支持 WITH HOLD(例如在多 DBnode 部署 GBase 8s 中所创建的含有 DDL 的事务属于跨节点事务）。

- query

  使用 SELECT 或 VALUES 子句指定游标返回的行。

  取值范围：SELECT 或 VALUES 子句。

示例

请参考 FETCH 的示例。

相关命令

FETCH

3.8.105DEALLOCATE

功能描述

DEALLOCATE 用于删除前面编写的预备语句。如果用户没有明确删除一个预备语句，那么它将在会话结束的时候被删除。

PREPARE 关键字总被忽略。

注意事项

无。

语法格式

```
DEALLOCATE [ PREPARE ] { name | ALL };
```

参数说明

- name

  将要删除的预备语句。

- ALL

  删除所有预备语句。

示例

无。

3.8.106DECLARE

功能描述

DECLARE 命令既可以定义一个游标，用于在一个大的查询里面检索少数几行数据，也可以作为一个匿名块的开始。

本节主要描述定义为游标的用法，开启匿名块的用法见 BEGIN。

为了处理 SQL 语句，存储过程进程分配一段内存区域来保存上下文联系。游标是指向上下文区域的句柄或指针。借助游标，存储过程可以控制上下文区域的变化。

通常游标和 SELECT 一样返回文本格式。因为数据在系统内部是用二进制格式存储的，系统必须对数据做一定转换以生成文本格式。一旦数据是以文本形式返回，客户端应用需要把它们转换成二进制进行操作。使用 FETCH 语句，游标可以返回文本或二进制格式。

注意事项

游标命令只能在事务块里使用。

应该小心使用二进制游标。文本格式一般都比对应的二进制格式占用的存储空间大。二进制游标返回内部二进制形态的数据，可能更易于操作。如果想以文本方式显示数据，则以文本方式检索会为用户节约很多客户端的工作。比如，如果查询从某个整数列返回 1，在缺省的游标里将获得一个字符串 1，但在二进制游标里将得到一个 4 字节的包含该数值内部形式的数值（大端顺序）。

语法格式

定义游标

```
DECLARE cursor_name [ BINARY ] [[NO] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ]
FOR query ;
```

开启匿名块

```
[DECLARE [declare_statements]]
BEGIN
execution_statements
END;
/
```

参数说明

- cursor_name

  将要创建的游标名。

  取值范围：遵循数据库对象命名规范。

- BINARY

  指明游标以二进制而不是文本格式返回数据。

- NO SCROLL

  声明游标检索数据行的方式。

  SCROLL：若指定，那么游标可以反向滚动。

  NO SCROLL：声明该游标不能用于以倒序的方式检索数据行。

  未声明:根据执行计划的不同,自动判断该游标是否可以用于以倒序的方式检索数据行。

- WITH HOLD | WITHOUT HOLD

  声明当创建游标的事务结束后，游标是否能继续使用。

  WITH HOLD：声明该游标在创建它的事务结束后仍可继续使用。

  WITHOUT HOLD：声明该游标在创建它的事务之外不能再继续使用，此游标将在事务结束时被自动关闭。

  如果不指定 WITH HOLD 或 WITHOUT HOLD，默认行为是 WITHOUT HOLD。

- query

  使用 SELECT 或 VALUES 子句指定游标返回的行。

  取值范围：SELECT 或 VALUES 子句。

- declare_statements

  声明变量，包括变量名和变量类型，如“sales_cnt int”。

- execution_statements

  匿名块中要执行的语句。

  取值范围：已存在的函数名称。

示例

参考 FETCH 的示例。

相关命令

BEGIN，FETCH

3.8.107DELETE

功能描述

DELETE 从指定的表里删除满足 WHERE 子句的行。如果 WHERE 子句不存在，将删除表中所有行，结果只保留表结构。

注意事项

表的所有者、被授予了表 DELETE 权限的用户或被授予 DELETE ANY TABLE 权限的用户有权删除表中数据，系统管理员默认拥有此权限。同时也必须有 USING 子句引用的表以及 condition 上读取的表的 SELECT 权限。

对于列存表，暂时不支持 RETURNING 子句。

语法格式

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE [/*+ plan_hint */] FROM [ ONLY ] table_name [partition_clause] [ * ] [ [ AS ]
alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ] [ LIMIT row_count ]
    [ RETURNING { * | { output_expr [ [ AS ] output_name ] } [, ...] } ];
```

参数说明

- WITH [RECURSIVE] with_query [, …]

  用于声明一个或多个可以在主查询中通过名称引用的子查询，相当于临时表。

如果声明了 RECURSIVE，那么允许 SELECT 子查询通过名称引用它自己。

其中 with_query 的详细格式为：

```
with_query_name [ ( column_name [, ...] ) ] AS [ [ NOT ] MATERIALIZED]
```

- ({select | values | insert | update | delete})

with_query_name 指定子查询生成的结果集名称，在查询中可使用该名称访问子查询的结果集。

column_name 指定子查询结果集中显示的列名。

每个子查询可以是 SELECT、VALUES、INSERT、UPDATE 或 DELETE 语句。

用户可以使用 MATERIALIZED / NOT MATERIALIZED 对 CTE 进行修饰。

如果声明为 MATERIALIZED，WITH 查询将被物化，生成一个子查询结果集的拷贝，在引用处直接查询该拷贝，因此 WITH 子查询无法和主干 SELECT 语句进行联合优化（如谓词下推、等价类传递等），对于此类场景可以使用 NOT MATERIALIZED 进行修饰，如果 WITH 查询语义上可以作为子查询内联执行，则可以进行上述优化。

如果用户没有显示声明物化属性则遵守以下规则：如果 CTE 只在所属主干语句中被引用一次，且语义上支持内联执行，则会被改写为子查询内联执行，否则以 CTE Scan 的方式物化执行。

- plan_hint 子句

以/*+ */的形式在 DELETE 关键字后，用于对 DELETE 对应的语句块生成的计划进行 hint 调优，详细用法请参见章节使用 Plan Hint 进行调优。每条语句中只有第一个/*+ plan_hint */注释块会作为 hint 生效，里面可以写多条 hint。

- ONLY

如果指定 ONLY 则只有该表被删除；如果没有声明，则该表和它的所有子表将都被删除。

- table_name

目标表的名称（可以有模式修饰）。

取值范围：已存在的表名。

- partition_clause

指定分区删除操作

- PARTITION { (partition_name) | FOR (partition_value [, …]) } |

  SUBPARTITION { (subpartition_name) | FOR (subpartition_value [, …]) }

  关键字详见 SELECT 一节介绍

  示例详见 CREATE TABLE SUBPARTITION

- alias

  目标表的别名。

  取值范围：字符串，符合标识符命名规范。

- using_list

  using 子句。

- condition

  一个返回 Boolean 值的表达式，用于判断哪些行需要被删除。不建议使用 int 等数值类型作为 condition，因为 int 等数值类型可以隐式转换为 bool 值（非 0 值隐式转换为 true，0 转换为 false），可能导致非预期的结果。

- WHERE CURRENT OF cursor_name

  当前不支持，仅保留语法接口。

- output_expr

  DELETE 命令删除行之后计算输出结果的表达式。该表达式可以使用表的任意字段。可以使用*返回被删除行的所有字段。

- output_name

  一个字段的输出名称。

  取值范围：字符串，符合标识符命名规范。

示例

```
--创建表 tpcds.customer_address_bak。
postgres=# CREATE TABLE tpcds.customer_address_bak AS TABLE
tpcds.customer_address;

--删除 tpcds.customer_address_bak 中 ca_address_sk 小于 14888 的职员。
postgres=# DELETE FROM tpcds.customer_address_bak WHERE ca_address_sk < 14888;
```

```
--删除 tpcds.customer_address_bak 中所有数据。
postgres=# DELETE FROM tpcds.customer_address_bak;

--删除 tpcds.customer_address_bak 表。
postgres=# DROP TABLE tpcds.customer_address_bak;
```

优化建议

- delete

  如果要删除表中的所有记录，建议使用 truncate 语法。

3.8.108DO

功能描述

执行匿名代码块。

代码块被看做是没有参数的一段函数体，返回值类型是 void。它的解析和执行是同一时刻发生的。

注意事项

程序语言在使用之前，必须通过命令 CREATE LANGUAGE 安装到当前的数据库中。plpgsql 是默认的安装语言，其它语言安装时必须指定。

如果语言是不受信任的，用户必须有使用程序语言的 USAGE 权限，或者是系统管理员。

语法格式

```
DO [ LANGUAGE lang_name ] code;
```

参数说明

- lang_name

  用来解析代码的程序语言的名称，如果缺省，默认的语言是 plpgsql。

- code

  程序语言代码可以被执行的。程序语言必须指定为字符串才行。

示例

```
--创建用户 webuser。
postgres=# CREATE USER webuser PASSWORD 'xxxxxxxxx';

--授予用户 webuser 对模式 tpcds 下视图的所有操作权限。
postgres=# DO $$DECLARE r record;
```

```
BEGIN
    FOR r IN SELECT c.relname table_name,n.nspname table_schema FROM pg_class
c,pg_namespace n
             WHERE c.relnamespace = n.oid AND n.nspname = 'tpcds' AND relkind IN
('r','v')
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;


--删除用户 webuser。
postgres=# DROP USER webuser CASCADE;
```

3.8.109DROP AGGREGATE

功能描述

删除一个聚合函数。

注意事项

DROP AGGREGATE 删除一个现存的聚合函数，执行这条命令的用户必须是该聚合函数的所有者。

语法格式

```
DROP AGGREGATE [ IF EXISTS ] name ( argtype [ , ... ] ) [ CASCADE | RESTRICT ]
```

参数说明

- IF EXISTS

  如果指定的聚合不存在，那么发出一个 notice 而不是抛出一个错误。

- name

  现存的聚合函数名（可以有模式修饰）。

- argtype

  聚合函数操作的输入数据类型，要引用一个零参数聚合函数，请用*代替输入数据类型列表。

- CASCADE

级联删除依赖于这个聚合函数的对象。

- RESTRICT

  如果有任何依赖对象，则拒绝删除这个聚合函数。这是缺省处理。

示例

将 integer 类型的聚合函数 myavg 删除：

```
DROP AGGREGATE myavg(integer);
```

兼容性

SQL 标准里没有 DROP AGGREGATE 语句。

3.8.110 DROP AUDIT POLICY

功能描述

删除一个审计策略。

注意事项

只有 poladmin、sysadmin 或初始用户才能进行此操作。

语法格式

```
DROP AUDIT POLICY [IF EXISTS] policy_name;
```

参数说明

- policy_name

  审计策略名称，需要唯一，不可重复。

  取值范围：字符串，要符合标识符的命名规范。

示例

请参考 CREATE AUDIT POLICY 的示例。

相关命令

ALTER AUDIT POLICY，CREATE AUDIT POLICY。

3.8.111 DROP CAST

功能描述

删除一个类型转换。

注意事项

DROP CAST 删除一个先前定义过的类型转换。

要能删除一个类型转换，你必须拥有源或者目的数据类型。这是和创建一个类型转换相同的权限。

语法格式

```
DROP CAST [ IF EXISTS ] (source_type AS target_type) [ CASCADE | RESTRICT ]
```

参数说明

- IF EXISTS

  如果指定的转换不存在，那么发出一个 notice 而不是抛出一个错误。

- source_type

  类型转换里的源数据类型。

- target_type

  类型转换里的目标数据类型。

- CASCADE | RESTRICT

  这些键字没有任何效果，因为在类型转换上没有依赖关系。

示例

删除从 text 到 int 的转换：

```
DROP CAST (text AS int);
```

兼容性

DROP CAST 遵循 SQL 标准。

3.8.112 DROP CLIENT MASTER KEY

功能描述

删除一个客户端加密主密钥（CMK）。

注意事项

只有客户端加密主密钥所有者或者被授予了 DROP 权限的用户有权限执行命令，系统管理员默认拥有此权限。

该命令不仅删除数据库中的密钥对象，还会同时删除客户端指定路径下该密钥对象对应的密钥文件。

语法格式

```
DROP CLIENT MASTER KEY [ IF EXISTS ] client_master_key_name [, ...];
```

参数说明

- IF EXISTS

  如果指定的客户端加密主密钥不存在，则发出一个 notice 而不是抛出一个错误。

- client_master_key_name

  要删除的客户端加密主密钥名称。

  取值范围：字符串，已存在的客户端加密主密钥对象的名称。

 须知：在执行本语法的生命周期中，同时需要客户端和服务端更改状态，发生异常时可能存在服务端已删除密钥信息，但客户端未删除密钥文件的情况。 此时，客户端并不会在执行下一条语法的生命周期中，检查是否有期望被删除但却因发生异常而未被删除的密钥文件，而是需要用户定期检查密钥文件夹，对未被使用的密钥文件进行确认并处理。

示例

```
--删除客户端加密主密钥对象。
postgres=# DROP CLIENT MASTER KEY ImgCMK CASCADE;
NOTICE:  drop cascades to column setting: imgcek
DROP CLIENT MASTER KEY
```

3.8.113 DROP COLUMN ENCRYPTION KEY

功能描述

删除一个列加密密钥（CEK）。

注意事项

只有列加密密钥所有者或者被授予了 DROP 权限的用户有权限执行命令，系统管理员默认拥有此权限。

语法格式

```
DROP COLUMN ENCRYPTION KEY [ IF EXISTS ] client_column_key_name [, ...];
```

参数说明

- IF EXISTS

  如果指定的列加密密钥不存在，则发出一个 notice 而不是抛出一个错误。

- column_encryption_key_name

  要删除的列加密密钥名称。

  取值范围：字符串，已存在的列加密密钥名称。

示例

```
--删除列加密密钥。
postgres=# DROP COLUMN ENCRYPTION KEY ImgCEK CASCADE;
ERROR:  cannot drop column setting: imgcek cascadely because encrypted column
depend on it.
HINT:  we have to drop encrypted column: name, ... before drop column setting:
imgcek cascadely.
```

3.8.114 DROP DATABASE

功能描述

删除一个数据库。

注意事项

只有数据库所有者或者被授予了数据库 DROP 权限的用户有权限执行 DROP DATABASE 命令，系统管理员默认拥有此权限。

不能对系统默认安装的三个数据库（POSTGRES、TEMPLATE0 和 TEMPLATE1）执行删除操作，系统做了保护。如果想查看当前服务中有哪几个数据库，可以用 gsql 的\l 命令查看。

如果有用户正在与要删除的数据库连接，则删除操作失败。

不能在事务块中执行 DROP DATABASE 命令。

如果执行 DROP DATABASE 失败，事务回滚，需要再次执行一次 DROP DATABASE IF EXISTS。

须知

DROP DATABASE 一旦执行将无法撤销，请谨慎使用。

语法格式

```
DROP DATABASE [ IF EXISTS ] database_name ;
```

参数说明

- IF EXISTS

  如果指定的数据库不存在，则发出一个 notice 而不是抛出一个错误。

- database_name

  要删除的数据库名称。

  取值范围：字符串，已存在的数据库名称。

示例

请参见 CREATE DATABASE 的示例。

相关命令

CREATE DATABASE

优化建议

- drop database

  不支持在事务中删除 database。

3.8.115 DROP DATA SOURCE

功能描述

删除一个 Data Source 对象。

注意事项

只有属主/系统管理员/初始用户才可以删除一个 Data Source 对象。

语法格式

```
DROP DATA SOURCE [IF EXISTS] src_name [CASCADE | RESTRICT];
```

参数说明

- src_name

  待删除的 Data Source 对象名称。

  取值范围：字符串，符合标识符命名规范。

- IF EXISTS

  如果指定的 Data Source 不存在，则发出一个 notice 而不是报错。

- CASCADE | RESTRICT

  CASCADE：表示允许级联删除依赖于 Data Source 的对象。

  RESTRICT（缺省值）：表示有依赖于该 Data Source 的对象存在，则该 Data Source 无法删除。

  目前 Data Source 对象没有被依赖的对象，CASCADE 和 RESTRICT 效果一样，保留此选项是为了向后兼容性。

示例

```
--创建 Data Source 对象。
postgres=# CREATE DATA SOURCE ds_tst1;

--删除 Data Source 对象。
postgres=# DROP DATA SOURCE ds_tst1 CASCADE;
postgres=# DROP DATA SOURCE IF EXISTS ds_tst1 RESTRICT;
```

相关命令

CREATE DATA SOURCE，ALTER DATA SOURCE

3.8.116 DROP DIRECTORY

功能描述

删除指定的 directory 对象。

注意事项

当 enable_access_server_directory=off 时，只允许初始用户删除 directory 对象；当 enable_access_server_directory=on 时，具有 SYSADMIN 权限的用户、directory 对象的属主、被授予了该 directory 的 DROP 权限的用户或者继承了内置角色 gs_rloe_directory_drop 权限的用户可以删除 directory。

语法格式

```
DROP DIRECTORY [ IF EXISTS ] directory_name;
```

参数说明

- directory_name

  目录名称。

  取值范围：已经存在的目录名。

示例

```
--创建目录。
postgres=# CREATE OR REPLACE DIRECTORY  dir  as '/tmp/';

--删除目录。
postgres=# DROP DIRECTORY dir;
```

相关命令

CREATE DIRECTORY，ALTER DIRECTORY

3.8.117 DROP EXTENSION

功能描述

删除一个扩展。

注意事项

DROP EXTENSION 命令从数据库中删除一个扩展。在删除扩展的过程中，构成扩展的组件也会一起删除。

必须是扩展的拥有者才能够使用 DROP EXTENSION 命令。

语法格式

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

参数说明

- IF EXISTS

  当使用 IF EXISTS 参数，如果扩展不存在时，不会抛出错误，而是产生一个通知。

- name

  已经安装的扩展模块的名称。

- CASCADE

  自动删除依赖于该扩展的对象。

- RESTRICT

  如果有依赖于扩展的对象，则不允许删除次扩展（除非它所有的成员对象和其它扩展对象在一条 DROP 命令一起删除）。这是缺省行为。

示例

从当前数据库中删除扩展 hstore

DROP EXTENSION hstore;

在当前数据库中，如果有使用 hstore 的对象的，这条命令就会失败，比如任一表中的字段使用 hstore 类型。这时增加 CASCADE 选项会强制删除扩展和依赖于扩展的对象。

3.8.118 DROP EVENT TRIGGER

功能描述

删除事件触发器。

注意事项

只有超级用户或者系统管理员才有权限删除事件触发器。

语法格式

```
DROP EVENT TRIGGER [ IF EXISTS ] name [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的事件触发器不存在，则发出一个 notice 而不是抛出一个错误。

- name

  要删除的事件触发器名称。

  取值范围：已存在的事件触发器。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖此触发器的对象。

  RESTRICT：如果有依赖对象存在，则拒绝删除此触发器。此选项为缺省值。

示例

请参见 CREATE EVENT TRIGGER 的示例。

3.8.119 DROP FOREIGN TABLE

功能描述

删除指定的外表。

注意事项

DROP FOREIGN TABLE 会强制删除指定的表，删除表后，依赖该表的索引会被删除，因此引用该表的函数和存储过程将无法执行。

语法格式

```
DROP FOREIGN TABLE [ IF EXISTS ]
    table_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的表不存在，则发出一个 notice 而不是抛出一个错误。

- table_name

  表名称。

  取值范围：已存在的表名。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖于表的对象（比如视图）。

  RESTRICT：如果存在依赖对象，则拒绝删除该表。这个是缺省。

相关命令

ALTER FOREIGN TABLE，CREATE FOREIGN TABLE

3.8.120DROP FUNCTION

功能描述

删除一个已存在的函数。

注意事项

如果函数中涉及对临时表相关操作，则无法使用 DROP FUNCTION 删除函数。

只有函数的所有者或者被授予了函数 DROP 权限的用户才能执行 DROP FUNCTION 命令，系统管理员默认拥有该权限。

语法格式

```
DROP FUNCTION [ IF EXISTS ] function_name
[ ( [ {[ argname ] [ argmode ] argtype} [, ...] ] ) [ CASCADE | RESTRICT ] ];
```

参数说明

- IF EXISTS

  IF EXISTS 表示，如果函数存在则执行删除操作，函数不存在也不会报错，只是发出一个 notice。

- function_name

  要删除的函数名称。

  取值范围：已存在的函数名。

- argmode

  函数参数的模式。

- argname

  函数参数的名称。

- argtype

  函数参数的类型。

示例

请参见的示例。

相关命令

ALTER FUNCTION，CREATE FUNCTION

3.8.121 DROP GLOBAL CONFIGURATION

功能描述

删除系统表 gs_global_config 中的参数值。

注意事项

仅支持数据库初始用户运行此命令。

不支持删除关键字为 weak_password。

语法格式

```
DROP GLOBAL CONFIGURATION paraname, paraname...;
```

参数说明

参数名称是 gs_global_config 中已经存在的参数，删除不存在的参数将报错。

3.8.122DROP GROUP

功能描述

删除用户组。

DROP GROUP 是 DROP ROLE 的别名。

注意事项

DROP GROUP 是 GBase 8s 管理工具封装的接口，用来实现 GBase 8s 管理。该接口不建议用户直接使用，以免对 GBase 8s 状态造成影响。

语法格式

```
DROP GROUP [ IF EXISTS ] group_name [, ...];
```

参数说明

请参见 DROP ROLE 的参数说明。

相关命令

CREATE GROUP，ALTER GROUP，DROP ROLE

3.8.123DROP INDEX

功能描述

删除索引。

注意事项

只有索引的所有者或者拥有索引所在表的 INDEX 权限的用户有权限执行 DROP INDEX 命令，系统管理员默认拥有此权限。

语法格式

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ]
    index_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- CONCURRENTLY

以不加锁的方式删除索引。删除索引时，一般会阻塞其他语句对该索引所依赖表的访问。加此关键字，可实现删除过程中不做阻塞。

此选项只能指定一个索引的名称，并且 CASCADE 选项不支持。

普通 DROP INDEX 命令可以在事务内执行，但是 DROP INDEX CONCURRENTLY 不可以在事务内执行。

- IF EXISTS

  如果指定的索引不存在，则发出一个 notice 而不是抛出一个错误。

- index_name

  要删除的索引名。

  取值范围：已存在的索引。

- CASCADE | RESTRICT

  CASCADE：表示允许级联删除依赖于该索引的对象。

  RESTRICT（缺省值）：表示有依赖与此索引的对象存在，则该索引无法被删除。

示例

请参见 CREATE INDEX 的示例。

相关命令

ALTER INDEX，CREATE INDEX

3.8.124DROP LANGUAGE

功能描述

删除一个过程语言。单机和集中式暂不支持删除过程语言。

语法格式

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

参数说明

- IF EXISTS

  如果指定的过程语言不存在，那么发出一个 notice 而不是抛出一个错误。

- name

  现存语言的名称。出于向下兼容的考虑，这个名字可以用单引号包围。

- CASCADE

  级联删除依赖于该语言的对象（比如该语言写的函数）。

- RESTRICT

  如果存在依赖对象，则拒绝删除。这个是缺省。

示例

下面命令删除 plsample 语言：

```
DROP LANGUAGE plsample;
```

兼容性

SQL 标准里没有 DROP LANGUAGE 语句。

3.8.125DROP MASKING POLICY

功能描述

删除脱敏策略。

注意事项

只有 poladmin、sysadmin 或初始用户才能执行此操作。

语法格式

```
DROP MASKING POLICY [IF EXISTS] policy_name;
```

参数说明

- policy_name

  审计策略名称，需要唯一，不可重复。

  取值范围：字符串，要符合标识符的命名规范。

示例

```
--删除一个脱敏策略。
postgres=# DROP MASKING POLICY IF EXISTS maskpol1;
--删除一组脱敏策略。
postgres=# DROP MASKING POLICY IF EXISTS maskpol1, maskpol2, maskpol3;
```

相关命令

ALTER MASKING POLICY，CREATE MASKING POLICY。

3.8.126DROP MATERIALIZED VIEW

功能描述

强制删除数据库中已有的物化视图。

注意事项

只有物化视图的所有者有权限执行 DROP MATERIALIZED VIEW 命令，系统管理员默认拥有此权限。

语法格式

```
DROP MATERIALIZED VIEW [ IF EXISTS ] mv_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的物化视图不存在，则发出一个 notice 而不是抛出一个错误。

- mv_name

  要删除的物化视图名称。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖此物化视图的对象。

  RESTRICT：如果有依赖对象存在，则拒绝删除此物化视图。此选项为缺省值。

示例

```
--删除名为 my_mv 的物化视图。
postgres=# DROP MATERIALIZED VIEW my_mv;
```

相关命令

ALTER MATERIALIZED VIEW，CREATE INCREMENTAL MATERIALIZED VIEW，CREATE MATERIALIZED VIEW，CREATE TABLE，REFRESH INCREMENTAL MATERIALIZED VIEW，REFRESH MATERIALIZED VIEW

3.8.127DROP MODEL

功能描述

删除一个已训练完成保存的模型对象。

注意事项

所删除模型可在系统表 gs_model_warehouse 中查看到。

语法格式

```
DROP MODEL model_name;
```

参数说明

- model_name

  模型名称

  取值范围：字符串，需要符合标识符的命名规范。

相关命令

CREATE MODEL，PREDICT BY

3.8.128DROP OPERATOR

功能描述

删除一个操作符。

语法格式

```
DROP OPERATOR [ IF EXISTS ] name ( { left_type | NONE } , { right_type | NONE } )
[ CASCADE | RESTRICT ]
```

参数说明

- left_type

  操作符左边的参数数据类型，如果存在的话。如果是左目操作符，这个参数可以省略。

- right_type

  操作符右边的参数数据类型，如果存在的话。如果是右目操作符，这个参数可以省略。

3.8.129DROP OWNED

功能描述

删除一个数据库角色所拥有的数据库对象。

注意事项

所有该角色在当前数据库里和共享对象（数据库、表空间）上的所有对象上的权限都将被撤销。

DROP OWNED 常常被用来为移除一个或者多个角色做准备。因为 DROP OWNED 只影响当前数据库中的对象，通常需要在包含将被移除角色所拥有的对象的每一个数据库中都执行这个命令。

使用 CASCADE 选项可能导致这个命令递归去删除由其他用户所拥有的对象。

角色所拥有的数据库、表空间将不会被移除。

语法格式

```
DROP OWNED BY name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- name

  角色名。

- CASCADE | RESTRICT

  CASCADE：级联删除所有依赖于被删除对象的对象。

  RESTRICT（缺省值）：拒绝删除那些有任何依赖对象存在的对象。

相关命令

REASSIGN OWNED , DROP ROLE

3.8.130DROP PACKAGE

功能描述

删除已存在的 PACKAGE。

语法格式

```
DROP PACKAGE [ IF EXISTS ] package_name;
```

3.8.131DROP PROCEDURE

功能描述

删除已存在的存储过程。

注意事项

无。

语法格式

```
DROP PROCEDURE [ IF EXISTS  ] procedure_name ;
```

参数说明

- IF EXISTS

如果指定的存储过程不存在，发出一个 notice 而不是抛出一个错误。

- procedure_name

  要删除的存储过程名称。

  取值范围：已存在的存储过程名。

相关命令

CREATE PROCEDURE

3.8.132DROP RESOURCE LABEL

功能描述

删除资源标签。

注意事项

只有 poladmin、sysadmin 或初始用户才能执行此操作。

语法格式

```
DROP RESOURCE LABEL [IF EXISTS] policy_name[, ...];
```

参数说明

- label_name

  资源标签名称；

  取值范围：字符串，要符合标识符的命名规范。

示例

```
--删除一个资源标签。
postgres=# DROP RESOURCE LABEL IF EXISTS res_label1;
--删除一组资源标签。
postgres=# DROP RESOURCE LABEL IF EXISTS res_label1, res_label2, res_label3;
```

相关命令

ALTER RESOURCE LABEL，CREATE RESOURCE LABEL

3.8.133DROP RESOURCE POOL

功能描述

删除一个资源池。

说明

如果某个角色已关联到该资源池，无法删除。

注意事项

只要用户对当前数据库有 DROP 权限，就可以删除资源池。

语法格式

```
DROP RESOURCE POOL [ IF EXISTS ] pool_name;
```

参数说明

- IF EXISTS

  如果指定的资源池不存在，发出一个 notice 而不是抛出一个错误。

- pool_name

  已创建过的资源池名称。

  取值范围：字符串，要符合标识符的命名规范。

示例

请参见 CREATE RESOURCE POOL 的示例。

相关命令

ALTER RESOURCE POOL，CREATE RESOURCE POOL

3.8.134DROP ROLE

功能描述

删除指定的角色。

注意事项

无。

语法格式

```
DROP ROLE [ IF EXISTS ] role_name [, ...];
```

参数说明

- IF EXISTS

  如果指定的角色不存在，则发出一个 notice 而不是抛出一个错误。

- role_name

  要删除的角色名称。

  取值范围：已存在的角色。

示例

请参见 CREATE ROLE 的示例。

相关命令

CREATE ROLE，ALTER ROLE，SET ROLE

3.8.135DROP ROW LEVEL SECURITY POLICY

功能描述

删除表上某个行访问控制策略。

注意事项

仅表的所有者或者管理员用户才能删除表的行访问控制策略。

语法格式

```
DROP [ ROW LEVEL SECURITY ] POLICY [ IF EXISTS ] policy_name ON table_name [ CASCADE
| RESTRICT ]
```

参数说明

- IF EXISTS

  如果指定的行访问控制策略不存在，发出一个 notice 而不是抛出一个错误。

- policy_name

  要删除的行访问控制策略的名称。

- table_name

  行访问控制策略所在的数据表名。

- CASCADE/RESTRICT

  仅适配此语法，无对象依赖于该行访问控制策略，CASCADE 和 RESTRICT 效果相同。

示例

```
--创建数据表 all_data
postgres=# CREATE TABLE all_data(id int, role varchar(100), data varchar(100));
```

```
--创建行访问控制策略
postgres=# CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role
= CURRENT_USER);
--删除行访问控制策略
postgres=# DROP ROW LEVEL SECURITY POLICY all_data_rls ON all_data;
```

相关命令

ALTER ROW LEVEL SECURITY POLICY，CREATE ROW LEVEL SECURITY POLICY

3.8.136DROP RULE

功能描述

删除一个重写规则。

语法格式

```
DROP RULE [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

参数说明

- IF EXISTS

  如果该规则不存在，会抛出一个 NOTICE。

- name

  要删除的现存规则名称。

- table_name

  该规则应用的表名。

- CASCADE

  自动级联删除依赖于此规则的对象。

- RESTRICT

  缺省情况下，如果有任何依赖对象，则拒绝删除此规则。

示例

```
--删除重写规则 newrule
DROP RULE newrule ON mytable;
```

相关命令

CREATE RULE

3.8.137DROP PUBLICATION

功能描述

从数据库中删除一个现有的发布。

注意事项

发布只能被其属主或系统管理员删除。

语法格式

```
DROP PUBLICATION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

参数说明

- IF EXISTS

  如果发布不存在，不要抛出一个错误，而是发出一个提示。

- name

  现有发布的名称。

- CASCADE|RESTRICT

  当前这些关键词没有任何作用，因为发布没有依赖关系。

示例

请参见示例。

相关命令

ALTER PUBLICATION，CREATE PUBLICATION

3.8.138DROP SCHEMA

功能描述

从数据库中删除模式。

注意事项

只有模式的所有者或者被授予了模式 DROP 权限的用户有权限执行 DROP SCHEMA 命令，系统管理员默认拥有此权限。

语法格式

```
DROP SCHEMA [ IF EXISTS ] schema_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的模式不存在，发出一个 notice 而不是抛出一个错误。

- schema_name

  模式的名称。

  取值范围：已存在模式名。

- CASCADE | RESTRICT

  CASCADE：自动删除包含在模式中的对象。

  RESTRICT：如果模式包含任何对象，则删除失败（缺省行为）。

  须知：不要随意删除 pg_temp 或 pg_toast_temp 开头的模式，这些模式是系统内部使用的，如果删除，可能导致无法预知的结果。

  说明：无法删除当前模式。如果要删除当前模式，须切换到其他模式下。

示例

请参见 CREATE SCHEMA 的示例。

相关命令

ALTER SCHEMA，CREATE SCHEMA。

3.8.139DROP SEQUENCE

功能描述

从当前数据库里删除序列。

注意事项

只有序列的所有者或者被授予了序列 DROP 权限的用户才能删除，系统管理员默认拥有该权限。

如果 SEQUENCE 被创建时使用了 LARGE 标识，DROP 时也需要使用 LARGE 标识。

语法格式

```
DROP  [ LARGE ]  SEQUENCE [ IF EXISTS ] {[schema.]sequence_name} [ , ... ]
[ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的序列不存在，则发出一个 notice 而不是抛出一个错误。

- name

  序列名称。

- CASCADE

  级联删除依赖序列的对象。

- RESTRICT

  如果存在任何依赖的对象，则拒绝删除序列。此项是缺省值。

示例

```
--创建一个名为 serial 的递增序列，从 101 开始。
postgres=# CREATE SEQUENCE serial START 101;
--删除序列。
postgres=# DROP SEQUENCE serial;
```

相关命令

ALTER SEQUENCE， CREATE SEQUENCE

3.8.140DROP SERVER

功能描述

删除现有的一个数据服务器。

注意事项

只有 server 的所有者或者被授予了 server 的 DROP 权限的用户才可以删除，系统管理员默认拥有该权限。

语法格式

```
DROP SERVER [ IF EXISTS ] server_name [ {CASCADE | RESTRICT} ] ;
```

参数说明

- IF EXISTS

如果指定的数据服务器不存在，则发出一个 notice 而不是抛出一个错误。

- server_name

  服务器名称。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖于 server 的对象 。

  RESTRICT（缺省值）：如果存在依赖对象，则拒绝删除该 server。

相关命令

ALTER SERVER，CREATE SERVER

3.8.141DROP SUBSCRIPTION

功能描述

删除数据库实例中的一个订阅。

注意事项

只有系统管理员才可以删除订阅。

如果该待删除订阅与复制槽相关联，就不能在事务块内部执行 DROP SUBSCRIPTION。

语法格式

```
DROP SUBSCRIPTION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

参数说明

- name

  要删除的订阅的名称。

- CASCADE|RESTRICT

  当前这些关键词没有任何作用，因为订阅没有依赖关系。

示例

请参见示例。

相关命令

ALTER SUBSCRIPTION，CREATE SUBSCRIPTION

3.8.142 DROP SYNONYM

功能描述

删除指定的 SYNONYM 对象。

注意事项

只有 SYNONYM 的所有者有权限执行 DROP SYNONYM 命令，系统管理员默认拥有此权限。

语法格式

```
DROP SYNONYM [ IF EXISTS ] synonym_name [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的同义词不存在，则发出一个 notice 而不是抛出一个错误。

- synonym_name

  同义词名字，可以带模式名。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖同义词的对象（比如视图）。

  RESTRICT：如果有依赖对象存在，则拒绝删除同义词。此选项为缺省值。

示例

请参考 CREATE SYNONYM 的示例。

相关命令

ALTER SYNONYM，CREATE SYNONYM

3.8.143 DROP TABLE

功能描述

删除指定的表。

注意事项

DROP TABLE 会强制删除指定的表，删除表后，依赖该表的索引会被删除，而使用到该表的函数和存储过程将无法执行。删除分区表，会同时删除分区表中的所有分区。

表的所有者、被授予了表的DROP权限的用户或被授予DROP ANY TABLE权限的用户，有权删除指定表 ，系统管理员默认拥有该权限。

语法格式

```
DROP TABLE [ IF EXISTS ]
    { [schema.]table_name } [, ...] [ CASCADE | RESTRICT ] [ PURGE ]};
```

参数说明

- IF EXISTS

  如果指定的表不存在，则发出一个 notice 而不是抛出一个错误。

- schema

  模式名称。

- table_name

  表名称。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖于表的对象（比如视图）。

  RESTRICT（缺省项）：如果存在依赖对象，则拒绝删除该表。这个是缺省。

- PURGE

  该参数表示即使开启回收站功能，DROP 表时，也会直接物理删除表，而不是将其放入回收站中。

示例

请参考 CREATE TABLE 的示例。

相关命令

ALTER TABLE，CREATE TABLE

3.8.144DROP TABLESPACE

功能描述

删除一个表空间。

注意事项

只有表空间所有者或者被授予了表空间 DROP 权限的用户有权限执行 DROP

TABLESPACE 命令，系统管理员默认拥有此权限。

在删除一个表空间之前，表空间里面不能有任何数据库对象，否则会报错。

DROP TABLESPACE 不支持回滚，因此，不能出现在事务块内部。

执行 DROP TABLESPACE 操作时，如果有另外的会话执行\db 查询操作，可能会由于 tablespace 事务的原因导致查询失败，请重新执行\db 查询操作。

如果执行 DROP TABLESPACE 失败，需要再次执行一次 DROP TABLESPACE IF EXISTS。

语法格式

```
DROP TABLESPACE [ IF EXISTS ] tablespace_name;
```

参数说明

- IF EXISTS

  如果指定的表空间不存在，则发出一个 notice 而不是抛出一个错误。

- tablespace_name

  表空间的名称。

  取值范围：已存在的表空间的名称。

示例

请参见 CREATE TABLESPACE 的示例。

相关命令

ALTER TABLESPACE， CREATE TABLESPACE

优化建议

drop database

不支持在事务中删除 database。

3.8.145DROP TEXT SEARCH CONFIGURATION

功能描述

删除已有文本搜索配置。

注意事项

要执行这个命令，用户必须是该配置的所有者。

语法格式

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的文本搜索配置不存在，那么发出一个 notice 而不是抛出一个错误。

- name

  要删除的文本搜索配置名称（可有模式修饰）。

- CASCADE

  级联删除依赖文本搜索配置的对象。

- RESTRICT

  若有任何对象依赖文本搜索配置则拒绝删除它。这是默认情况。

示例

请参见 CREATE TEXT SEARCH CONFIGURATION 的示例。

相关命令

ALTER TEXT SEARCH CONFIGURATION，CREATE TEXT SEARCH CONFIGURATION

3.8.146DROP TEXT SEARCH DICTIONARY

功能描述

删除全文检索词典。

注意事项

预定义词典不支持 DROP 操作。

只有词典的所有者可以执行 DROP 操作，系统管理员默认拥有此权限。

谨慎执行 DROP…CASCADE 操作，该操作将级联删除使用该词典的文本搜索配置（TEXT SEARCH CONFIGURATION）。

语法格式

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

参数说明

- IF EXISTS

  如果指定的全文检索词典不存在，那么发出一个 Notice 而不是报错。

- name

  要删除的词典名称（可指定模式名，否则默认在当前模式下）。

  取值范围：已存在的词典名。

- CASCADE

  自动删除依赖于该词典的对象，并依次删除依赖于这些对象的所有对象。

  如果存在任何一个使用该词典的文本搜索配置，此 DROP 命令将不会成功。可添加 CASCADE 以删除引用该词典的所有文本搜索配置以及词典。

  RESTRICT

  如果任何对象依赖词典，则拒绝删除该词典。这是缺省值。

示例

```
--删除词典 english

postgres=# DROP TEXT SEARCH DICTIONARY english;
```

相关命令

ALTER TEXT SEARCH DICTIONARY，CREATE TEXT SEARCH DICTIONARY

3.8.147DROP TRIGGER

功能描述

删除触发器。

注意事项

只有触发器的所有者可以执行 DROP TRIGGER 操作，系统管理员默认拥有此权限。

语法格式

```
DROP TRIGGER [ IF EXISTS ] trigger_name ON table_name [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的触发器不存在，则发出一个 notice 而不是抛出一个错误。

- trigger_name

  要删除的触发器名称。

  取值范围：已存在的触发器。

- table_name

  要删除的触发器所在的表名称。

  取值范围：已存在的含触发器的表。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖此触发器的对象。

  RESTRICT：如果有依赖对象存在，则拒绝删除此触发器。此选项为缺省值。

示例

请参见 CREATE TRIGGER 的示例。

相关命令

CREATE TRIGGER，ALTER TRIGGER，ALTER TABLE

3.8.148DROP USER

功能描述

删除用户。同时删除与用户同名的 schema。

注意事项

- 须使用 CASCADE 级联删除依赖用户的对象（除数据库外）。当删除用户的级联对象时，如果级联对象处于锁定状态，则此级联对象无法被删除，直到对象被解锁或锁定级联对象的进程被杀死。
- GBase 8s 数据库中存在 enable_kill_query 配置参数，此参数在配置文件 postgresql.conf 中。此参数影响级联删除用户对象的行为：
  - 当参数 enable_kill_query 为 on ，且使用 CASCADE 模式删除用户时，会自动 kill 锁定用户级联对象的进程，并删除用户。

- 当参数 enable_kill_query 为 off，且使用 CASCADE 模式删除用户时，会等待锁定级联对象的进程结束之后再删除用户。

- 在数据库中删除用户时，如果依赖用户的对象在其他数据库中或者依赖用户的对象是其他数据库，请用户先手动删除其他数据库中的依赖对象或直接删除依赖数据库，再删除用户。即 drop user 不支持跨数据库进行级联删除。
- 如果该用户被 DATA SOURCE 对象依赖时，无法直接级联删除该用户，需要手动删除对应的 DATA SOURCE 对象之后再删除该用户。

语法格式

```
DROP USER [ IF EXISTS ] user_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的用户不存在，发出一个 notice 而不是抛出一个错误。

- user_name

  待删除的用户名。

  取值范围：已存在的用户名。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖用户的对象。

  RESTRICT：如果用户还有任何依赖的对象，则拒绝删除该用户（缺省行为）。

示例

请参考 CREATE USER 的示例。

相关命令

ALTER USER，CREATE USER

3.8.149DROP USER MAPPING

功能描述

移除一个用于外部服务器的用户映射。

语法格式

```
DROP USER MAPPING [ IF EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC }
SERVER server_name;
```

参数说明

- IF EXISTS

  如果该用户映射不存在则不要抛出一个错误，而是发出一个提示。

- user_name

  该映射的用户名。

  CURRENT_USER 和 USER 匹配当前用户的名称。PUBLIC 被用来匹配系统中所有现存和未来的用户名。

- server_name

  用户映射的服务器名。

相关命令

ALTER USER MAPPING，CREATE USER MAPPING

3.8.150DROP TYPE

功能描述

删除一个用户定义的数据类型。

注意事项

只有类型的所有者或者被授予了类型 DROP 权限的用户有权限执行 DROP TYPE 命令，系统管理员默认拥有此权限。

语法格式

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

参数说明

- IF EXISTS

  如果指定的类型不存在，那么发出一个 notice 而不是抛出一个错误。

- name

  要删除的类型名(可以有模式修饰)。

- CASCADE

级联删除依赖该类型的对象(比如字段、函数、操作符等)。

- RESTRICT

  如果有依赖对象，则拒绝删除该类型（缺省行为）。

示例

请参考 CREATE TYPE 的示例。

相关命令

CREATE TYPE，ALTER TYPE

3.8.151DROP VIEW

功能描述

数据库中强制删除已有的视图。

注意事项

视图的所有者或者被授予了视图 DROP 权限的用户或拥有 DROP ANY TABLE 权限的用户，有权限执行 DROP VIEW 的命令，系统管理员默认拥有此权限。

语法格式

```
DROP VIEW [ IF EXISTS ] view_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- IF EXISTS

  如果指定的视图不存在，则发出一个 notice 而不是抛出一个错误。

- view_name

  要删除的视图名称。

  取值范围：已存在的视图。

- CASCADE | RESTRICT

  CASCADE：级联删除依赖此视图的对象（比如其他视图）。

  RESTRICT：如果有依赖对象存在，则拒绝删除此视图。此选项为缺省值。

示例

请参见 CREATE VIEW 的示例。

相关命令

ALTER VIEW，CREATE VIEW

3.8.152 DROP WEAK PASSWORD DICTIONARY

功能描述

清空 gs_global_config 中的所有弱口令。

注意事项

只有初始用户、系统管理员和安全管理员拥有权限执行本语法。

语法格式

```
DROP WEAK PASSWORD DICTIONARY;
```

参数说明

无。

示例

参见 CREATE WEAK PASSWORD DICTIONARY 的示例。

相关命令

CREATE WEAK PASSWORD DICTIONARY

3.8.153 EXECUTE

功能描述

执行一个前面准备好的预备语句。因为一个预备语句只在会话的生命期里存在，那么预备语句必须是在当前会话的前些时候用 PREPARE 语句创建的。

注意事项

如果创建预备语句的 PREPARE 语句声明了一些参数，那么传递给 EXECUTE 语句的必须是一个兼容的参数集，否则就会生成一个错误。

语法格式

```
EXECUTE name [ ( parameter [, ...] ) ];
```

参数说明

- name

要执行的预备语句的名称。

- parameter

给预备语句的一个参数的具体数值。它必须是一个和生成与创建这个预备语句时指定参数的数据类型相兼容的值的表达式。

示例

```
--创建表 reason。
postgres=# CREATE TABLE tpcds.reason (
    CD_DEMO_SK           INTEGER          NOT NULL,
    CD_GENDER            character(16)             ,
    CD_MARITAL_STATUS    character(100)
)
;
--插入数据。
postgres=# INSERT INTO tpcds.reason VALUES(51, 'AAAAAAAADDAAAAAA', 'reason 51');
--创建表 reason_t1。
postgres=# CREATE TABLE tpcds.reason_t1 AS TABLE tpcds.reason;
--为一个 INSERT 语句创建一个预备语句然后执行它。
postgres=# PREPARE insert_reason(integer,character(16),character(100)) AS
INSERT INTO tpcds.reason_t1 VALUES($1,$2,$3);
postgres=# EXECUTE insert_reason(52, 'AAAAAAAADDAAAAAA', 'reason 52');
--删除表 reason 和 reason_t1。
postgres=# DROP TABLE tpcds.reason;
postgres=# DROP TABLE tpcds.reason_t1;
```

3.8.154EXPLAIN

功能描述

显示 SQL 语句的执行计划。

执行计划将显示 SQL 语句所引用的表会采用什么样的扫描方式，如：简单的顺序扫描、索引扫描等。如果引用了多个表，执行计划还会显示用到的 JOIN 算法。

执行计划的最关键的部分是语句的预计执行开销，这是计划生成器估算执行该语句将花费多长的时间。

若指定了 ANALYZE 选项，则该语句会被执行，然后根据实际的运行结果显示统计数据，包括每个计划节点内时间总开销（毫秒为单位）和实际返回的总行数。这对于判断计划生成器的估计是否接近现实非常有用。

注意事项

在指定 ANALYZE 选项时，语句会被执行。如果用户想使用 EXPLAIN 分析 INSERT、UPDATE、DELETE、CREATE TABLE AS 或 EXECUTE 语句，而不想改动数据（执行这些语句会影响数据），请使用如下方法。

```
START TRANSACTION;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

由于参数 DETAIL、NODES、NUM_NODES 在单机模式中是被禁止使用的。假如使用，会产生如下错误。

```
postgres=# create table student(id int, name char(20));
CREATE TABLE
postgres=# explain (nodes true) insert into student values(5,'a'),(6,'b');
ERROR:  unrecognized EXPLAIN option "nodes"
postgres=# explain (num_nodes true) insert into student values(5,'a'),(6,'b');
ERROR:  unrecognized EXPLAIN option "num_nodes"
```

语法格式

显示 SQL 语句的执行计划，支持多种选项，对选项顺序无要求。

```
EXPLAIN [ (  option  [, ...] )  ] statement;
```

其中选项 option 子句的语法为。

```
ANALYZE [ boolean ] |
    ANALYSE [ boolean ] |
    VERBOSE [ boolean ] |
    COSTS [ boolean ] |
    CPU [ boolean ] |
    DETAIL [ boolean ] |(不可用)
    NODES [ boolean ] |(不可用)
    NUM_NODES [ boolean ] |(不可用)
    BUFFERS [ boolean ] |
    TIMING [ boolean ] |
    PLAN [ boolean ] |
    FORMAT { TEXT | XML | JSON | YAML }
```

显示 SQL 语句的执行计划，且要按顺序给出选项。

```
EXPLAIN  { [  { ANALYZE  | ANALYSE  }  ] [ VERBOSE  ]  | PERFORMANCE  }
statement;
```

参数说明

- statement

  指定要分析的 SQL 语句。

- ANALYZE boolean | ANALYSE boolean

  显示实际运行时间和其他统计数据。

  取值范围：

  TRUE（缺省值）：显示实际运行时间和其他统计数据。

  FALSE：不显示。

- VERBOSE boolean

  显示有关计划的额外信息。

  取值范围：

  TRUE（缺省值）：显示额外信息。

  FALSE：不显示。

- COSTS boolean

  包括每个规划节点的估计总成本，以及估计的行数和每行的宽度。

  取值范围：

  TRUE（缺省值）：显示估计总成本和宽度。

  FALSE：不显示。

- CPU boolean

  打印 CPU 的使用情况的信息。

  取值范围：

  TRUE（缺省值）：显示 CPU 的使用情况。

  FALSE：不显示。

- DETAIL boolean（不可用）

  打印数据库节点上的信息。

  取值范围：

  TRUE（缺省值）：打印数据库节点的信息。

FALSE：不打印。

- NODES boolean（不可用）

  打印 query 执行的节点信息。

  取值范围：

  TRUE（缺省值）：打印执行的节点的信息。

  FALSE：不打印。

- NUM_NODES boolean（不可用）

  打印执行中的节点的个数信息。

  取值范围：

  TRUE（缺省值）：打印数据库节点个数的信息。

  FALSE：不打印。

- BUFFERS boolean

  包括缓冲区的使用情况的信息。

  取值范围：

  TRUE：显示缓冲区的使用情况。

  FALSE（缺省值）：不显示。

- TIMING boolean

  包括实际的启动时间和花费在输出节点上的时间信息。

  取值范围：

  TRUE（缺省值）：显示启动时间和花费在输出节点上的时间信息。

  FALSE：不显示。

- PLAN

  是否将执行计划存储在 plan_table 中。当该选项开启时，会将执行计划存储在 PLAN_TABLE 中，不打印到当前屏幕，因此该选项为 on 时，不能与其他选项同时使用。

  取值范围：

  ON（缺省值）：将执行计划存储在 plan_table 中，不打印到当前屏幕。执行成功返回

EXPLAIN SUCCESS。

OFF：不存储执行计划，将执行计划打印到当前屏幕。

- FORMAT

  指定输出格式。

  取值范围：TEXT、XML、JSON 和 YAML。

  默认值：TEXT。

- PERFORMANCE

  使用此选项时，即打印执行中的所有相关信息。

示例

```
--创建一个表 tpcds.customer_address_p1。
postgres=# CREATE TABLE tpcds.customer_address_p1 AS TABLE
tpcds.customer_address;
--修改 explain_perf_mode 为 normal
postgres=# SET explain_perf_mode=normal;
--显示表简单查询的执行计划。
postgres=# EXPLAIN SELECT * FROM tpcds.customer_address_p1;
QUERY PLAN
------------------------------------------------
Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
Node/s: All dbnodes
(2 rows)
--以 JSON 格式输出的执行计划（explain_perf_mode 为 normal 时）。
postgres=# EXPLAIN(FORMAT JSON) SELECT * FROM tpcds.customer_address_p1;
              QUERY PLAN
-----------------------------------
 [                                 +
   {                               +
     "Plan": {                     +
       "Node Type": "Data Node Scan",+
       "Startup Cost": 0.00,       +
       "Total Cost": 0.00,         +
       "Plan Rows": 0,             +
       "Plan Width": 0,            +
       "Node/s": "All dbnodes"     +
     }                             +
   }                               +
```

```
 ]
(1 row)
--如果有一个索引，当使用一个带索引 WHERE 条件的查询，可能会显示一个不同的计划。
postgres=# EXPLAIN SELECT * FROM tpcds.customer_address_p1 WHERE
ca_address_sk=10000;
QUERY PLAN
------------------------------------------------
Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
Node/s: dn_6005_6006
(2 rows)
--以 YAML 格式输出的执行计划（explain_perf_mode 为 normal 时）。
postgres=# EXPLAIN(FORMAT YAML) SELECT * FROM tpcds.customer_address_p1 WHERE
ca_address_sk=10000;
           QUERY PLAN
--------------------------------
 - Plan:                       +
     Node Type: "Data Node Scan"+
     Startup Cost: 0.00        +
     Total Cost: 0.00          +
     Plan Rows: 0              +
     Plan Width: 0             +
     Node/s: "dn_6005_6006"
(1 row)
--禁止开销估计的执行计划。
postgres=# EXPLAIN(COSTS FALSE)SELECT * FROM tpcds.customer_address_p1 WHERE
ca_address_sk=10000;
       QUERY PLAN
-----------------------
 Data Node Scan
   Node/s: dn_6005_6006
(2 rows)
--带有聚集函数查询的执行计划。
postgres=# EXPLAIN SELECT SUM(ca_address_sk) FROM tpcds.customer_address_p1
WHERE ca_address_sk<10000;
                                    QUERY PLAN
------------------------------------------------------------------------------
----------
 Aggregate  (cost=18.19..14.32 rows=1 width=4)
   ->  Streaming (type: GATHER)  (cost=18.19..14.32 rows=3 width=4)
         Node/s: All dbnodes
         ->  Aggregate  (cost=14.19..14.20 rows=3 width=4)
```

```
               ->  Seq Scan on customer_address_p1  (cost=0.00..14.18 rows=10
width=4)
                     Filter: (ca_address_sk < 10000)
(6 rows)

--创建一个二级分区表。
postgres=# CREATE TABLE range_list
postgres=# (
postgres=#     month_code VARCHAR2 ( 30 ) NOT NULL ,
postgres=#     dept_code  VARCHAR2 ( 30 ) NOT NULL ,
postgres=#     user_no    VARCHAR2 ( 30 ) NOT NULL ,
postgres=#     sales_amt  int
postgres=# )
postgres=# PARTITION BY RANGE (month_code) SUBPARTITION BY LIST (dept_code)
postgres=# (
postgres=#   PARTITION p_201901 VALUES LESS THAN( '201903' )
postgres=#   (
postgres=#    SUBPARTITION p_201901_a values ('1'),
postgres=#     SUBPARTITION p_201901_b values ('2')
postgres=#   ),
postgres=#   PARTITION p_201902 VALUES LESS THAN( '201910' )
postgres=# #   (
postgres=#     SUBPARTITION p_201902_a values ('1'),
postgres=#     SUBPARTITION p_201902_b values ('2')
postgres=#   )
postgres=# );
CREATE TABLE
--执行带有二级分区表的查询语句。
--Iterations 和 Sub Iterations 分别标识遍历了几个一级分区和二级分区。
--Selected Partitions 标识哪些一级分区被实际扫描，Selected Subpartitions: (p:s)
标识第 p 个一级分区下 s 个二级分区被实际扫描,如果一级分区下所有二级分区都被扫描
则 s 显示为 ALL。
postgres=# EXPLAIN SELECT * FROM range_list WHERE dept_code = '1';
                                  QUERY PLAN
------------------------------------------------------------------------------
--
 Partition Iterator  (cost=0.00..13.81 rows=2 width=238)
   Iterations: 2, Sub Iterations: 2
   ->  Partitioned Seq Scan on range_list  (cost=0.00..13.81 rows=2 width=238)
         Filter: ((dept_code)::text = '1'::text)
         Selected Partitions:  1..2
         Selected Subpartitions:  1:1, 2:1
```

```
(6 rows)
--删除表 tpcds.customer_address_p1。
postgres=# DROP TABLE tpcds.customer_address_p1;
```

相关命令

ANALYZE | ANALYSE

3.8.155EXPLAIN PLAN

功能描述

通过 EXPLAIN PLAN 命令可以将查询执行的计划信息存储于 PLAN_TABLE 表中。与 EXPLAIN 命令不同的是，EXPLAIN PLAN 仅将计划信息进行存储，而不会打印到屏幕。

语法格式

```
EXPLAIN PLAN
[ SET STATEMENT_ID = string ]
FOR statement ;
```

参数说明

EXPLAIN 中的 PLAN 选项表示需要将计划信息存储于 PLAN_TABLE 中，存储成功将返回“EXPLAIN SUCCESS”。

STATEMENT_ID 用户可以对查询设置标签，输入的标签信息也将存储于 PLAN_TABLE 中。

说明：

用户在执行 EXPLAIN PLAN 时，如果没有进行 SET STATEMENT_ID，则默认为空值。同时，用户可输入的 STATEMENT_ID 最大长度为 30 个字节，超过长度将会产生报错。

statement：SQL 语句

注意事项

对于执行错误的 SQL 无法进行计划信息的收集。

PLAN_TABLE 中的数据是 session 级生命周期并且 session 隔离和用户隔离，用户只能看到当前 session、当前用户的数据。

示例

使用 EXPLAIN PLAN 收集 SQL 语句的执行计划，通常包括以下步骤：

执行 EXPLAIN PLAN。

说明：执行 EXPLAIN PLAN 后会将计划信息自动存储于 PLAN_TABLE 中，不支持对 PLAN_TABLE 进行 INSERT、UPDATE、ANALYZE 等操作。PLAN_TABLE 详细介绍见《GBase 8s V8.8.5_5.0.0_数据库参考手册》中“PLAN_TABLE”章节。

```
explain plan set statement_id='TPCH-Q4' for
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >= '1993-07-01'::date
and o_orderdate < '1993-07-01'::date + interval '3 month'
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
```

查询 PLAN_TABLE。

```
SELECT * FROM PLAN_TABLE;
```

清理 PLAN_TABLE 表中的数据。

```
DELETE FROM PLAN_TABLE WHERE xxx;
```

3.8.156FETCH

功能描述

FETCH 通过已创建的游标来检索数据。

每个游标都有一个供 FETCH 使用的关联位置。游标的关联位置可以在查询结果的第一行之前，或者在结果中的任意行，或者在结果的最后一行之后：

游标刚创建完之后，关联位置在第一行之前的。

在抓取了一些移动行之后，关联位置在检索到的最后一行上。

如果 FETCH 抓取完了所有可用行，它就停在最后一行后面，或者在反向抓取的情况下是停在第一行前面。

FETCH ALL 或 FETCH BACKWARD ALL 将总是把游标的关联位置放在最后一行或者在第一行前面。

注意事项

如果游标定义了 NO SCROLL，则不允许使用例如 FETCH BACKWARD 之类的反向抓取。

NEXT、PRIOR、FIRST、LAST、ABSOLUTE、RELATIVE 形式在恰当地移动游标之后抓取一条记录。如果后面没有数据行，就返回一个空的结果，此时游标就会停在查询结果的最后一行之后（向后查询时）或者第一行之前（向前查询时）。

FORWARD 和 BACKWARD 形式在向前或者向后移动的过程中抓取指定的行数，然后把游标定位在最后返回的行上；或者是，如果 count 大于可用的行数，则在所有行之后（向后查询时）或者之前（向前查询时）。

RELATIVE 0、FORWARD 0、BACKWARD 0 都要求在不移动游标的前提下抓取当前行，也就是重新抓取最近刚抓取过的行。除非游标定位在第一行之前或者最后一行之后，这个动作都应该成功，而在那两种情况下，不返回任何行。

当 FETCH 的游标上涉及列存表时，不支持 BACKWARD、PRIOR 等涉及反向获取操作。

语法格式

```
FETCH [ direction { FROM | IN } ] cursor_name;
其中 direction 子句为可选参数。
NEXT
    | PRIOR
    | FIRST
    | LAST
    | ABSOLUTE count
    | RELATIVE count
    | count
    | ALL
    | FORWARD
    | FORWARD count
    | FORWARD ALL
    | BACKWARD
```

```
| BACKWARD count
| BACKWARD ALL
```

参数说明

- direction_clause

  定义抓取数据的方向。

  取值范围：

  NEXT（缺省值）

  从当前关联位置开始，抓取下一行。

- PRIOR

  从当前关联位置开始，抓取上一行。

- FIRST

  抓取查询的第一行（和 ABSOLUTE 1 相同）。

- LAST

  抓取查询的最后一行（和 ABSOLUTE -1 相同）。

- ABSOLUTE count

  抓取查询中第 count 行。

  ABSOLUTE 抓取不会比用相对位移移动到需要的数据行更快，因为下层的实现必须遍历所有中间的行。

  count 取值范围：有符号的整数

  count 为正数，就从查询结果的第一行开始，抓取第 count 行。

  count 为负数，就从查询结果末尾抓取第 abs(count)行。

  count 为 0 时，定位在第一行之前。

- RELATIVE count

  从当前关联位置开始，抓取随后或前面的第 count 行。

  取值范围：有符号的整数

  count 为正数就抓取当前关联位置之后的第 count 行。

  count 为负数就抓取当前关联位置之前的第 abs(count)行。

如果当前行没有数据的话，RELATIVE 0 返回空。

- count

  抓取随后的 count 行（和 FORWARD count 一样）。

- ALL

  从当前关联位置开始，抓取所有剩余的行（和 FORWARD ALL 一样）。

- FORWARD

  抓取下一行（和 NEXT 一样）。

- FORWARD count

  从当前关联位置开始，抓取随后或前面的 count 行。

- FORWARD ALL

  从当前关联位置开始，抓取所有剩余行。

- BACKWARD

  从当前关联位置开始，抓取前面一行(和 PRIOR 一样) 。

- BACKWARD count

  从当前关联位置开始，抓取前面的 count 行（向后扫描）。

  取值范围：有符号的整数

  count 为正数就抓取当前关联位置之前的 count 行。

  count 为负数就抓取当前关联位置之后的 abs（count）行。

  如果有数据的话，BACKWARD 0 重新抓取当前行。

- BACKWARD ALL

  从当前关联位置开始，抓取所有前面的行（向后扫描） 。

- { FROM | IN } cursor_name

  使用关键字 FROM 或 IN 指定游标名称。

  取值范围：已创建的游标的名称。

示例

```
--SELECT 语句，用一个游标读取一个表。开始一个事务。
```

```
postgres=# START TRANSACTION;
--建立一个名为 cursor1 的游标。
postgres=# CURSOR cursor1 FOR SELECT * FROM tpcds.customer_address ORDER BY 1;
--抓取头 3 行到游标 cursor1 里。
postgres=# FETCH FORWARD 3 FROM cursor1;
 ca_address_sk |  ca_address_id   | ca_street_number |   ca_street_name   |
ca_street_type  | ca_suite_number |     ca_city      |    ca_county     |
ca_state |   ca_zip   |  ca_country   | ca_gmt_offset |   ca_location_type
---------------+------------------+------------------+--------------------+--
--------------+----------------+----------------+----------------+-------
---+-----------+--------------+--------------+----------------------
             1 | AAAAAAAABAAAAAAA | 18               | Jackson            |
Parkway         | Suite 280       | Fairfield       | Maricopa County | AZ
| 86192      | United States |         -7.00 | condo
             2 | AAAAAAAACAAAAAAA | 362              | Washington 6th     | RD
| Suite 80        | Fairview        | Taos County     | NM        | 85709      |
United States |         -7.00 | condo
             3 | AAAAAAAADAAAAAAA | 585              | Dogwood Washington |
Circle          | Suite Q         | Pleasant Valley | York County     | PA
| 12477      | United States |         -5.00 | single family
(3 rows)
--关闭游标并提交事务。
postgres=# CLOSE cursor1;
--结束一个事务。
postgres=# END;
--VALUES 子句，用一个游标读取 VALUES 子句中的内容。开始一个事务。
postgres=# START TRANSACTION;
--建立一个名为 cursor2 的游标。
postgres=# CURSOR cursor2 FOR VALUES(1,2),(0,3) ORDER BY 1;
--抓取头 2 行到游标 cursor2 里。
postgres=# FETCH FORWARD 2 FROM cursor2;
column1 | column2
--------+---------
0 |       3
1 |       2
(2 rows)
--关闭游标并提交事务。
postgres=# CLOSE cursor2;
--结束一个事务。
postgres=# END;
--WITH HOLD 游标的使用，开启事务。
postgres=# START TRANSACTION;
```

```
--创建一个 with hold 游标。
postgres=# DECLARE cursor1 CURSOR WITH HOLD FOR SELECT * FROM
tpcds.customer_address ORDER BY 1;
--抓取头 2 行到游标 cursor1 里。
postgres=# FETCH FORWARD 2 FROM cursor1;
 ca_address_sk |  ca_address_id   | ca_street_number |   ca_street_name   |
ca_street_type  | ca_suite_number |     ca_city     |    ca_county    |
ca_state |   ca_zip   |  ca_country   | ca_gmt_offset |   ca_location_type
---------------+------------------+------------------+--------------------+--
--------------+----------------+----------------+----------------+-------
---+-----------+--------------+--------------+---------------------
             1 | AAAAAAAABAAAAAAA | 18               | Jackson            |
Parkway         | Suite 280       | Fairfield       | Maricopa County | AZ
| 86192      | United States |         -7.00 | condo
             2 | AAAAAAAACAAAAAAA | 362              | Washington 6th     | RD
| Suite 80        | Fairview        | Taos County     | NM       | 85709      |
United States |         -7.00 | condo
(2 rows)
--结束事务。
postgres=# END;
--抓取下一行到游标 cursor1 里。
postgres=# FETCH FORWARD 1 FROM cursor1;
 ca_address_sk |  ca_address_id   | ca_street_number |   ca_street_name   |
ca_street_type  | ca_suite_number |     ca_city     |    ca_county    |
ca_state |   ca_zip   |  ca_country   | ca_gmt_offset |   ca_location_type
---------------+------------------+------------------+--------------------+--
--------------+----------------+----------------+----------------+-------
---+-----------+--------------+--------------+---------------------
             3 | AAAAAAAADAAAAAAA | 585              | Dogwood Washington |
Circle          | Suite Q         | Pleasant Valley | York County     | PA
| 12477      | United States |         -5.00 | single family
(1 row)
--关闭游标。
postgres=# CLOSE cursor1;
```

使用动态游标/自动游标。

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
```

```
END;
' LANGUAGE plpgsql;

-- 需要在一个事务中使用游标。
BEGIN;
SELECT reffunc2();

      reffunc2
------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

相关命令

CLOSE，MOVE

3.8.157GRANT

功能描述

对角色和用户进行授权操作。

使用 GRANT 命令进行用户授权包括以下三种场景：

将系统权限授权给角色或用户

系统权限又称为用户属性，包括 SYSADMIN、CREATEDB、CREATEROLE、AUDITADMIN、MONADMIN、OPRADMIN、POLADMIN 和 LOGIN。

系统权限一般通过 CREATE/ALTER ROLE 语法来指定。其中，SYSADMIN 权限可以通过 GRANT/REVOKE ALL PRIVILEGE 授予或撤销。但系统权限无法通过 ROLE 和 USER 的权限被继承，也无法授予 PUBLIC。

将数据库对象授权给角色或用户

将数据库对象（表和视图、指定字段、数据库、函数、模式、表空间等）的相关权限授予特定角色或用户；

GRANT 命令将数据库对象的特定权限授予一个或多个角色。这些权限会追加到已有的权限上。

关键字 PUBLIC 表示该权限要赋予所有角色，包括以后创建的用户。PUBLIC 可以看做

是一个隐含定义好的组，它总是包括所有角色。任何角色或用户都将拥有通过 GRANT 直接赋予的权限和所属的权限，再加上 PUBLIC 的权限。

如果声明了 WITH GRANT OPTION，则被授权的用户也可以将此权限赋予他人，否则就不能授权给他人。这个选项不能赋予 PUBLIC，这是 GBase 8s 特有的属性。

GBase 8s 会将某些类型的对象上的权限授予 PUBLIC。默认情况下，对表、表字段、序列、外部数据源、外部服务器、模式或表空间对象的权限不会授予 PUBLIC，而以下这些对象的权限会授予 PUBLIC：数据库的 CONNECT 权限和 CREATE TEMP TABLE 权限、函数的 EXECUTE 特权、语言和数据类型（包括域）的 USAGE 特权。当然，对象拥有者可以撤销默认授予 PUBLIC 的权限并专门授予权限给其他用户。为了更安全，建议在同一个事务中创建对象并设置权限，这样其他用户就没有时间窗口使用该对象。另外可参考安全加固指南的权限控制章节，对 PUBLIC 用户组的权限进行限制。这些初始的默认权限可以使用 ALTER DEFAULT PRIVILEGES 命令修改。

对象的所有者缺省具有该对象上的所有权限，出于安全考虑所有者可以舍弃部分权限，但 ALTER、DROP、COMMENT、INDEX、VACUUM 以及对象的可再授予权限属于所有者固有的权限，隐式拥有。

将角色或用户的权限授权给其他角色或用户

将一个角色或用户的权限授予一个或多个其他角色或用户。在这种情况下，每个角色或用户都可视为拥有一个或多个数据库权限的集合。

当声明了 WITH ADMIN OPTION，被授权的用户可以将该权限再次授予其他角色或用户，以及撤销所有由该角色或用户继承到的权限。当授权的角色或用户发生变更或被撤销时，所有继承该角色或用户权限的用户拥有的权限都会随之发生变更。

数据库系统管理员可以给任何角色或用户授予/撤销任何权限。拥有 CREATEROLE 权限的角色可以赋予或者撤销任何非系统管理员角色的权限。

将 ANY 权限授予给角色或用户

将 ANY 权限授予特定的角色和用户，ANY 权限的取值范围参见语法格式。当声明了 WITH ADMIN OPTION，被授权的用户可以将该 ANY 权限再次授予其他角色/用户，或从其他角色/用户处回收该 ANY 权限。ANY 权限可以通过角色被继承，但不能赋予 PUBLIC。初始用户和三权分立关闭时的系统管理员用户可以给任何角色/用户授予或撤销 ANY 权限。

目前支持以下 ANY 权限：CREATE ANY TABLE、ALTER ANY TABLE、DROP ANY TABLE、SELECT ANY TABLE、INSERT ANY TABLE、UPDATE ANY TABLE、DELETE ANY

TABLE、CREATE ANY SEQUENCE、CREATE ANY INDEX、CREATE ANY FUNCTION、EXECUTE ANY FUNCTION、 CREATE ANY PACKAGE、EXECUTE ANY PACKAGE、CREATE ANY TYPE。详细的 ANY 权限范围描述参考表 1。

注意事项

不允许将 ANY 权限授予 PUBLIC，也不允许从 PUBLIC 回收 ANY 权限。

ANY 权限属于数据库内的权限，只对授予该权限的数据库内的对象有效，例如 SELECT ANY TABLE 只允许用户查看当前数据库内的所有用户表数据，对其他数据库内的用户表无查看权限。

即使用户被授予 ANY 权限，也不能对私有用户下的对象进行访问操作（INSERT、DELETE、UPDATE、SELECT）。

ANY 权限与原有的权限相互无影响。

如果用户被授予 CREATE ANY TABLE 权限，在同名 schema 下创建表的属主是该 schema 的创建者，用户对表进行其他操作时，需要授予相应的操作权限。

语法格式

将表或视图的访问权限赋予指定的用户或角色。

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
    ALTER | DROP | COMMENT | INDEX | VACUUM } [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将表中字段的访问权限赋予指定的用户或角色。

```
GRANT { {{ SELECT | INSERT | UPDATE | REFERENCES | COMMENT } ( column_name [, ...] )}
[, ...]
      | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将序列的访问权限赋予指定的用户或角色，LARGE 字段属性可选，赋权语句不区分序列是否为 LARGE。

```
GRANT { { SELECT | UPDATE | USAGE | ALTER | DROP | COMMENT } [, ...]
      | ALL [ PRIVILEGES ] }
```

```
    ON { [ [ LARGE ] SEQUENCE ] sequence_name [, ...]
       | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将数据库的访问权限赋予指定的用户或角色。

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP | ALTER | DROP | COMMENT } [, ...]
| ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将域的访问权限赋予指定的用户或角色。

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

说明：本版本暂时不支持赋予域的访问权限。

将客户端加密主密钥 CMK 的访问权限赋予指定的用户或角色。

```
GRANT { { USAGE | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON CLIENT_MASTER_KEY client_master_key [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将列加密密钥 CEK 的访问权限赋予指定的用户或角色。

```
GRANT { { USAGE | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON COLUMN_ENCRYPTION_KEY column_encryption_key [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将外部数据源的访问权限赋予给指定的用户或角色。

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将外部服务器的访问权限赋予给指定的用户或角色。

```
GRANT { { USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
```

```
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将函数的访问权限赋予给指定的用户或角色。

```
GRANT { { EXECUTE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON { FUNCTION {function_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )}
[, ...]
       | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将存储过程的访问权限赋予给指定的用户或角色。

```
GRANT { { EXECUTE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
  ON { PROCEDURE {proc_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )}
[, ...]
     | ALL PROCEDURE IN SCHEMA schema_name [, ...] }
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

将过程语言的访问权限赋予给指定的用户或角色。

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将大对象的访问权限赋予指定的用户或角色。

```
GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

 说明：本版本暂时不支持大对象。

将模式的访问权限赋予指定的用户或角色。

```
GRANT { { CREATE | USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

 说明：将模式中的表或者视图对象授权给其他用户时，需要将表或视图所属的模式的 USAGE 权限同时授予该用户，若没有该权限，则只能看到这些对象的名称，并不能实际

进行对象访问。 同名模式下创建表的权限无法通过此语法赋予，可以通过将角色的权限赋予其他用户或角色的语法，赋予同名模式下创建表的权限。

将表空间的访问权限赋予指定的用户或角色。

```
GRANT { { CREATE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将类型的访问权限赋予指定的用户或角色。

```
GRANT { { USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

 说明：本版本暂时不支持赋予类型的访问权限。

将 NODE GROUP 对象的权限赋予指定的用户或角色。

```
GRANT { { CREATE | USAGE | COMPUTE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON NODE GROUP group_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将 Data Source 对象的权限赋予指定的角色。

```
GRANT { USAGE | ALL [PRIVILEGES]}
   ON DATA SOURCE src_name [, ...]
   TO { [GROUP] role_name | PUBLIC } [, ...]
   [WITH GRANT OPTION];
```

将 directory 对象的权限赋予指定的角色。

```
GRANT { { READ | WRITE | ALTER | DROP } [, ...] | ALL [PRIVILEGES] }
   ON DIRECTORY directory_name [, ...]
   TO { [GROUP] role_name | PUBLIC } [, ...]
   [WITH GRANT OPTION];
```

将 package 对象的权限赋予指定的角色。

```
GRANT { { EXECUTE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON { PACKAGE package_name [, ...]
       | ALL PACKAGES IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

将角色的权限赋予其他用户或角色的语法。

```
GRANT role_name [, ...]
   TO role_name [, ...]
   [ WITH ADMIN OPTION ];
```

将 sysadmin 权限赋予指定的角色。

```
GRANT ALL { PRIVILEGES | PRIVILEGE }
   TO role_name;
```

将 ANY 权限赋予其他用户或角色的语法。

```
GRANT { CREATE ANY TABLE | ALTER ANY TABLE | DROP ANY TABLE | SELECT ANY TABLE
| INSERT ANY TABLE |
    UPDATE ANY TABLE | DELETE ANY TABLE | CREATE ANY SEQUENCE | CREATE ANY INDEX
|
    CREATE ANY FUNCTION | EXECUTE ANY FUNCTION | CREATE ANY PACKAGE |
    EXECUTE ANY PACKAGE | CREATE ANY TYPE } [, ...]
    TO [ GROUP ] role_name [, ...]
    [ WITH ADMIN OPTION ];
```

参数说明

GRANT 的权限分类如下所示。

- SELECT

允许对指定的表、视图、序列执行 SELECT 命令，update 或 delete 时也需要对应字段上的 select 权限。

- INSERT

允许对指定的表执行 INSERT 命令。

- UPDATE

允许对声明的表中任意字段执行 UPDATE 命令。通常，update 命令也需要 select 权限来查询出哪些行需要更新。SELECT… FOR UPDATE 和 SELECT… FOR SHARE 除了需要 SELECT 权限外，还需要 UPDATE 权限。

- DELETE

允许执行 DELETE 命令删除指定表中的数据。通常，delete 命令也需要 select 权限来查询出哪些行需要删除。

- TRUNCATE

允许执行 TRUNCATE 语句删除指定表中的所有记录。

- REFERENCES

  创建一个外键约束，必须拥有参考表和被参考表的 REFERENCES 权限。

- CREATE

  对于数据库，允许在数据库里创建新的模式。

  对于模式，允许在模式中创建新的对象。如果要重命名一个对象，用户除了必须是该对象的所有者外，还必须拥有该对象所在模式的 CREATE 权限。

  对于表空间，允许在表空间中创建表，允许在创建数据库和模式的时候把该表空间指定为缺省表空间。

- CONNECT

  允许用户连接到指定的数据库。

- EXECUTE

  允许使用指定的函数，以及利用这些函数实现的操作符。

- USAGE

  对于过程语言，允许用户在创建函数的时候指定过程语言。

  对于模式，USAGE 允许访问包含在指定模式中的对象，若没有该权限，则只能看到这些对象的名称。

  对于序列，USAGE 允许使用 nextval 函数。

  对于 Data Source 对象，USAGE 是指访问权限，也是可赋予的所有权限，即 USAGE 与 ALL PRIVILEGES 等价。

- ALTER

  允许用户修改指定对象的属性，但不包括修改对象的所有者和修改对象所在的模式。

- DROP

  允许用户删除指定的对象。

- COMMENT

  允许用户定义或修改指定对象的注释。

- INDEX

允许用户在指定表上创建索引，并管理指定表上的索引，还允许用户对指定表执行 REINDEX 和 CLUSTER 操作。

- VACUUM

  允许用户对指定的表执行 ANALYZE 和 VACUUM 操作。

- ALL PRIVILEGES

  一次性给指定用户/角色赋予所有可赋予的权限。只有系统管理员有权执行 GRANT ALL PRIVILEGES。

GRANT 的参数说明如下所示。

- role_name

  已存在用户名称。

- table_name

  已存在表名称。

- column_name

  已存在字段名称。

- schema_name

  已存在模式名称。

- database_name

  已存在数据库名称。

- function_name

  已存在函数名称。

- procedure_name

  已存在存储过程名称。

- sequence_name

  已存在序列名称。

- domain_name

  已存在域类型名称。

- fdw_name

  已存在外部数据包名称。

- lang_name

  已存在语言名称。

- type_name

  已存在类型名称。

- src_name

  已存在的 Data Source 对象名称。

- argmode

  参数模式。

  取值范围：字符串，要符合标识符命名规范。

- arg_name

  参数名称。

  取值范围：字符串，要符合标识符命名规范。

- arg_type

  参数类型。

  取值范围：字符串，要符合标识符命名规范。

- loid

  包含本页的大对象的标识符。

  取值范围：字符串，要符合标识符命名规范。

- tablespace_name

  表空间名称。

- client_master_key

  客户端加密主密钥的名称。

  取值范围：字符串，要符合标识符命名规范。

- column_encryption_key

列加密密钥的名称。

取值范围：字符串，要符合标识符命名规范。

- directory_name

目录名称。

取值范围：字符串，要符合标识符命名规范。

- WITH GRANT OPTION

如果声明了 WITH GRANT OPTION，则被授权的用户也可以将此权限赋予他人，否则就不能授权给他人。这个选项不能赋予 PUBLIC。

非对象所有者给其他用户授予对象权限时，命令按照以下规则执行：

如果用户没有该对象上指定的权限，命令立即失败。

如果用户有该对象上的部分权限，则 GRANT 命令只授予他有授权选项的权限。

如果用户没有可用的授权选项，GRANT ALL PRIVILEGES 形式将发出一个警告信息，其他命令形式将发出在命令中提到的且没有授权选项的相关警告信息。

说明：数据库系统管理员可以访问所有对象，而不会受对象的权限设置影响。这个特点类似 Unix 系统的 root 的权限。和 root 一样，除了必要的情况外，建议不要总是以系统管理员身份进行操作。 不允许对表分区进行 GRANT 操作，对分区表进行 GRANT 操作会引起告警。

- WITH ADMIN OPTION

对于角色，当声明了 WITH ADMIN OPTION，被授权的用户可以将该角色再授予其他角色/用户，或从其他角色/用户回收该角色。

对于 ANY 权限，当声明了 WITH ADMIN OPTION，被授权的用户可以将该 ANY 权限再授予其他角色/用户，或从其他角色/用户回收该 ANY 权限。

表 3-30 ANY 权限列表

系统权限名称	描述
CREATE ANY	用户能够在 public 模式和用户模式下创建表或视图。如果想要创建

系统权限名称	描述
TABLE	serial 类型列的表，还需要授予创建序列的权限。
ALTER ANY TABLE	用户拥有对 public 模式和用户模式下表或视图的 ALTER 权限。如果想要修改表的唯一索引为表增加主键约束或唯一约束，还需要授予该表的索引权限。
DROP ANY TABLE	用户拥有对 public 模式和用户模式下表或视图的 DROP 权限。
SELECT ANY TABLE	用户拥有对 public 模式和用户模式下表或视图的 SELETCT 权限，仍然受行级访问控制限制。
UPDATE ANY TABLE	用户拥有对 public 模式和用户模式下表或视图的 UPDATE 权限，仍然受行级访问控制限制。
INSERT ANY TABLE	用户拥有对 public 模式和用户模式下表或视图的 INSERT 权限。
DELETE ANY TABLE	用户拥有对 public 模式和用户模式下表或视图的 DELETE 权限，仍然受行级访问控制限制。
CREATE ANY FUNCTION	用户能够在用户模式下创建函数或存储过程。
EXECUTE ANY FUNCTION	用户拥有在 public 模式和用户模式下函数或存储过程的 EXECUTE 权限。
CREATE ANY	用户能够在 public 模式和用户模式下创建 PACKAGE。

系统权限名称	描述
PACKAGE	
EXECUTE ANY PACKAGE	用户拥有在 public 模式和用户模式下 PACKAGE 的 EXECUTE 权限。
CREATE ANY TYPE	用户能够在 public 模式和用户模式下创建类型。
CREATE ANY SEQUENCE	用户能够在 public 模式和用户模式下创建序列。
CREATE ANY INDEX	用户能够在 public 模式和用户模式下创建索引。如果在某表空间创建分区表索引，需要授予用户该表空间的创建权限。

说明：用户被授予任何一种 ANY 权限后，用户对 public 模式和用户模式具有 USAGE 权限，对上表中除 public 之外的系统模式没有 USAGE 权限。

示例

(1) 示例：将系统权限授权给用户或者角色。

创建名为 joe 的用户，并将 sysadmin 权限授权给他。

```
postgres=# CREATE USER joe PASSWORD 'xxxxxxxxx';
postgres=# GRANT ALL PRIVILEGES TO joe;
```

授权成功后，用户 joe 会拥有 sysadmin 的所有权限。

(2) 示例：将对象权限授权给用户或者角色。

撤销 joe 用户的 sysadmin 权限，然后将模式 tpcds 的使用权限和表 tpcds.reason 的所有权限授权给用户 joe。

```
postgres=# REVOKE ALL PRIVILEGES FROM joe;
postgres=# GRANT USAGE ON SCHEMA tpcds TO joe;
postgres=# GRANT ALL PRIVILEGES ON tpcds.reason TO joe;
```

授权成功后，joe 用户就拥有了 tpcds.reason 表的所有权限，包括增删改查等权限。

将 tpcds.reason 表中 r_reason_sk、r_reason_id、r_reason_desc 列的查询权限，r_reason_desc 的更新权限授权给 joe。

```
postgres=# GRANT select (r_reason_sk,r_reason_id,r_reason_desc),update
(r_reason_desc) ON tpcds.reason TO joe;
```

授权成功后，用户 joe 对 tpcds.reason 表中 r_reason_sk，r_reason_id 的查询权限会立即生效。如果 joe 用户需要拥有将这些权限授权给其他用户的权限，可以通过以下语法对 joe 用户进行授权。

```
postgres=# GRANT select (r_reason_sk, r_reason_id) ON tpcds.reason TO joe WITH
GRANT OPTION;
```

将数据库 GBase 8s 的连接权限授权给用户 joe，并给予其在 GBase 8s 中创建 schema 的权限，而且允许 joe 将此权限授权给其他用户。

```
postgres=# GRANT create,connect on database gbase TO joe WITH GRANT OPTION;
```

创建角色 tpcds_manager，将模式 tpcds 的访问权限授权给角色 tpcds_manager，并授予该角色在 tpcds 下创建对象的权限，不允许该角色中的用户将权限授权给其他人。

```
postgres=# CREATE ROLE tpcds_manager PASSWORD 'xxxxxxxxx';
postgres=# GRANT USAGE,CREATE ON SCHEMA tpcds TO tpcds_manager;
```

将表空间 tpcds_tbspc 的所有权限授权给用户 joe，但用户 joe 无法将权限继续授予其他用户。

```
postgres=# CREATE TABLESPACE tpcds_tbspc RELATIVE LOCATION
'tablespace/tablespace_1';
postgres=# GRANT ALL ON TABLESPACE tpcds_tbspc TO joe;
```

(3) 示例：将用户或者角色的权限授权给其他用户或角色。

创建角色 manager，将 joe 的权限授权给 manager，并允许该角色将权限授权给其他人。

```
postgres=# CREATE ROLE manager PASSWORD 'xxxxxxxxx';
postgres=# GRANT joe TO manager WITH ADMIN OPTION;
```

创建用户 senior_manager，将用户 manager 的权限授权给该用户。

```
postgres=# CREATE ROLE senior_manager PASSWORD 'xxxxxxxxx';
postgres=# GRANT manager TO senior_manager;
```

撤销权限，并清理用户。

```
postgres=# REVOKE manager FROM joe;
```

```
postgres=# REVOKE senior_manager FROM manager;
postgres=# DROP USER manager;
```

(4) 示例：将 CMK 或者 CEK 的权限授权给其他用户或角色。

连接密态数据库。

```
gsql -p 57101 gbase -r -C
postgres=# CREATE CLIENT MASTER KEY MyCMK1 WITH ( KEY_STORE = localkms , KEY_PATH
= "key_path_value" , ALGORITHM = RSA_2048);
CREATE CLIENT MASTER KEY
postgres=# CREATE COLUMN ENCRYPTION KEY MyCEK1 WITH VALUES (CLIENT_MASTER_KEY =
MyCMK1, ALGORITHM = AEAD_AES_256_CBC_HMAC_SHA256);
CREATE COLUMN ENCRYPTION KEY
```

创建角色 newuser，将密钥的权限授权给 newuser。

```
postgres=# CREATE USER newuser PASSWORD 'xxxxxxxxx';
CREATE ROLE
postgres=# GRANT ALL ON SCHEMA public TO newuser;
GRANT
postgres=# GRANT USAGE ON COLUMN_ENCRYPTION_KEY MyCEK1 to newuser;
GRANT
postgres=# GRANT USAGE ON CLIENT_MASTER_KEY MyCMK1 to newuser;
GRANT
```

设置该用户连接数据库，使用该 CEK 创建加密表。

```
postgres=# SET SESSION AUTHORIZATION newuser PASSWORD 'xxxxxxxxx';
postgres=#  CREATE TABLE acltest1 (x int, x2 varchar(50) ENCRYPTED WITH
(COLUMN_ENCRYPTION_KEY = MyCEK1, ENCRYPTION_TYPE = DETERMINISTIC));
CREATE TABLE
postgres=# SELECT has_cek_privilege('newuser', 'MyCEK1', 'USAGE');
 has_cek_privilege
-------------------
 t
(1 row)
```

撤销权限，并清理用户。

```
postgres=# REVOKE USAGE ON COLUMN_ENCRYPTION_KEY MyCEK1 FROM newuser;
postgres=# REVOKE USAGE ON CLIENT_MASTER_KEY MyCMK1 FROM newuser;
postgres=# DROP TABLE newuser.acltest1;
postgres=# DROP COLUMN ENCRYPTION KEY MyCEK1;
postgres=# DROP CLIENT MASTER KEY MyCMK1;
postgres=# DROP SCHEMA IF EXISTS newuser CASCADE;
```

```
postgres=# REVOKE ALL ON SCHEMA public FROM newuser;
postgres=# DROP ROLE IF EXISTS newuser;
```

(5) 示例：撤销上述授予的权限，并清理角色和用户。

```
postgres=# REVOKE ALL PRIVILEGES ON tpcds.reason FROM joe;
postgres=# REVOKE ALL PRIVILEGES ON SCHEMA tpcds FROM joe;
postgres=# REVOKE ALL ON TABLESPACE tpcds_tbspc FROM joe;
postgres=# DROP TABLESPACE tpcds_tbspc;
postgres=# REVOKE USAGE,CREATE ON SCHEMA tpcds FROM tpcds_manager;
postgres=# DROP ROLE tpcds_manager;
postgres=# DROP ROLE senior_manager;
postgres=# DROP USER joe CASCADE;
```

相关命令

REVOKE，ALTER DEFAULT PRIVILEGES

3.8.158INSERT

功能描述

向表中添加一行或多行数据。

注意事项

只有拥有表 INSERT 权限的用户，才可以向表中插入数据。用户被授予 insert any table 权限，相当于用户对除系统模式之外的任何模式具有 USAGE 权限，并且拥有这些模式下表的 INSERT 权限

如果使用 RETURNING 子句，用户必须要有该表的 SELECT 权限。

如果使用 ON DUPLICATE KEY UPDATE，用户必须要有该表的 SELECT、UPDATE 权限，唯一约束（主键或唯一索引）的 SELECT 权限。

如果使用 query 子句插入来自查询里的数据行，用户还需要拥有在查询里使用的表的 SELECT 权限。

当连接到 TD 兼容的数据库时，td_compatible_truncation 参数设置为 on 时，将启用超长字符串自动截断功能，在后续的 insert 语句中（不包含外表的场景下），对目标表中 char 和 varchar 类型的列上插入超长字符串时，系统会自动按照目标表中相应列定义的最大长度对超长字符串进行截断。

说明：如果向字符集为字节类型编码（SQL_ASCII，LATIN1 等）的数据库中插

入多字节字符数据（如汉字等），且字符数据跨越截断位置，这种情况下，按照字节长度自动截断，自动截断后会在尾部产生非预期结果。如果用户有对于截断结果正确性的要求，建议用户采用 UTF8 等能够按照字符截断的输入字符集作为数据库的编码集。

语法格式

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT [/*+ plan_hint */] INTO table_name [partition_clause] [ AS alias ]
[ ( column_name [, ...] ) ]
    { DEFAULT VALUES
    | VALUES {( { expression | DEFAULT } [, ...] ) }[, ...]
    | query }
    [ ON DUPLICATE KEY UPDATE { NOTHING | { column_name = { expression | DEFAULT } }
[, ...] [ WHERE condition ] }]
    [ RETURNING {* | {output_expression [ [ AS ] output_name ] }[, ...]} ]
    [ ON CONFLICT [ conflict_target ] conflict_action ];
```

其中 conflict_target 可以是以下之一：

```
    ( { index_column_name | ( index_expression ) } [ COLLATE collation ]
[ opclass ] [, ...] ) [ WHERE index_predicate ]
    ON CONSTRAINT constraint_name
```

并且 conflict_action 是以下之一：

```
    DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT }
[, ...] ) |
                    ( column_name [, ...] ) = ( sub-SELECT )
                  } [, ...]
              [ WHERE condition ]
```

参数说明

- WITH [RECURSIVE] with_query [, …]

  用于声明一个或多个可以在主查询中通过名称引用的子查询，相当于临时表。

  如果声明了 RECURSIVE，那么允许 SELECT 子查询通过名称引用它自己。

  其中 with_query 的详细格式为：

```
with_query_name [ ( column_name [, ...] ) ] AS [ [ NOT ] MATERIALIZED ]
( {select | values | insert | update | delete} )
```

  – with_query_name 指定子查询生成的结果集名称，在查询中可使用该名称访问子查询

的结果集。

– column_name 指定子查询结果集中显示的列名。

– 每个子查询可以是 SELECT，VALUES，INSERT，UPDATE 或 DELETE 语句。

– 用户可以使用 MATERIALIZED / NOT MATERIALIZED 对 CTE 进行修饰。

如果声明为 MATERIALIZED，WITH 查询将被物化，生成一个子查询结果集的拷贝，在引用处直接查询该拷贝，因此 WITH 子查询无法和主干 SELECT 语句进行联合优化（如谓词下推、等价类传递等），对于此类场景可以使用 NOT MATERIALIZED 进行修饰，如果 WITH 查询语义上可以作为子查询内联执行，则可以进行上述优化。

如果用户没有显示声明物化属性则遵守以下规则：如果 CTE 只在所属主干语句中被引用一次，且语义上支持内联执行，则会被改写为子查询内联执行，否则以 CTE Scan 的方式物化执行。

说明：INSERT ON DUPLICATE KEY UPDATE 不支持 WITH 及 WITH RECURSIVE 子句。

- plan_hint 子句

以/*+ */的形式在 INSERT 关键字后，用于对 INSERT 对应的语句块生成的计划进行 hint 调优，详细用法请参见章节使用 Plan Hint 进行调优。每条语句中只有第一个/*+ plan_hint */注释块会作为 hint 生效，里面可以写多条 hint。

- table_name

  要插入数据的目标表名。

  取值范围：已存在的表名。

- partition_clause

  指定分区插入操作

- PARTITION { (partition_name) | FOR (partition_value [, ...]) } |

  SUBPARTITION { (subpartition_name) | FOR (subpartition_value [, ...]) }

  关键字详见 SELECT 一节介绍

  如果 value 子句的值和指定分区不一致，会抛出异常。

  示例详见 CREATE TABLE SUBPARTITION

- column_name

  目标表中的字段名：

  字段名可以有子字段名或者数组下标修饰。

  没有在字段列表中出现的每个字段，将由系统默认值，或者声明时的默认值填充，若都没有则用 NULL 填充。例如，向一个复合类型中的某些字段插入数据的话，其他字段将是 NULL。

  目标字段（column_name）可以按顺序排列。如果没有列出任何字段，则默认全部字段，且顺序为表声明时的顺序。

  如果 value 子句和 query 中只提供了 N 个字段，则目标字段为前 N 个字段。

  value 子句和 query 提供的值在表中从左到右关联到对应列。

  取值范围：已存在的字段名。

- expression

  赋予对应 column 的一个有效表达式或值：

  如果是 INSERT ON DUPLICATE KEY UPDATE 语句下，expression 可以为 VALUES(column_name) 或 EXCLUDED.column_name 用来表示引用冲突行对应的 column_name 字段的值。需注意，其中 VALUES(column_name)不支持嵌套在表达式中（例如 VALUES(column_name)+1），但 EXCLUDED 不受此限制。

  向表中字段插入单引号“'”时需要使用单引号自身进行转义。

  如果插入行的表达式不是正确的数据类型，系统试图进行类型转换，若转换不成功，则插入数据失败，系统返回错误信息。

- DEFAULT

  对应字段名的缺省值。如果没有缺省值，则为 NULL。

- query

  一个查询语句（SELECT 语句），将查询结果作为插入的数据。

- RETURNING

  返回实际插入的行，RETURNING 列表的语法与 SELECT 的输出列表一致。注意：INSERT ON DUPLICATE KEY UPDATE 不支持 RETURNING 子句。

- output_expression

INSERT 命令在每一行都被插入之后用于计算输出结果的表达式。

取值范围：该表达式可以使用 table 的任意字段。可以使用*返回被插入行的所有字段。

- output_name

字段的输出名称。

取值范围：字符串，符合标识符命名规范。

- ON DUPLICATE KEY UPDATE

对于带有唯一约束（UNIQUE INDEX 或 PRIMARY KEY）的表，如果插入数据违反唯一约束，则对冲突行执行 UPDATE 子句完成更新，对于不带唯一约束的表，则仅执行插入。UPDATE 时，若指定 NOTHING 则忽略此条插入，可通过“EXCLUDE.” 或者 “VALUES()” 来选择源数据相应的列。

支持触发器，触发器执行顺序由实际执行流程决定：

执行 insert：触发 before insert、after insert 触发器。

执行 update：触发 before insert、before update、after update 触发器。

执行 update nothing：触发 before insert 触发器。

不支持延迟生效（DEFERRABLE）的唯一约束或主键。

如果表中存在多个唯一约束，如果所插入数据违反多个唯一约束，对于检测到冲突的第一行进行更新，其他冲突行不更新（检查顺序与索引维护具有强相关性，一般先创建的索引先进行冲突检查）。

如果插入多行，这些行均与表中同一行数据存在唯一约束冲突，则按照顺序，第一条执行插入或更新，之后依次执行更新。

主键、唯一索引列不允许 UPDATE。

不支持列存，不支持外表、内存表。

expression 支持使用子查询表达式，其语法与功能同 UPDATE。子查询表达式中支持使用“EXCLUDED.”来选择源数据相应的列。

ON CONFLICT 子句

可选的 ON CONFLICT 子句为出现唯一性违背或排除 约束违背错误时提供另一种可供选择的动作。对于每一个要插入的行， 不管是插入进行下去还是由 conflict_target 指定的一个仲裁者约束或者索引被违背，都会 采取可供选择的 conflict_action。ON CONFLICT DO

NOTHING 简单地把避免插入行。 ON CONFLICT DO UPDATE 则会 更新与要插入的行冲突的已有行。

conflict_target 可以执行 唯一索引推断。在执行推断时，它由一个或者多个 index_column_name 列或者 index_expression 表达式以及一个可选的 index_predicate 构成。所有刚好包含 conflict_target 指定的列/表达式的 table_name 唯一索引（不管顺序）都 会被推断为（选择为）仲裁者索引。如果指定了 index_predicate，它 必须满足仲裁者索引（也是推断过程的一个进一步的要求）。注意这意味着如果 有一个满足其他条件的非部分唯一索引 (没有谓词的唯一索引) 可用，它将被 推断为仲裁者 (并且会被 ON CONFLICT 使用)。如果推断 尝试不成功，则会发生一个错误。

ON CONFLICT DO UPDATE 保证一个原子的 INSERT 或者 UPDATE 结果。在没有无关错误的前提下，这两种 结果之一可以得到保证，即使在很高的并发度也能保证。这也可以被称作 UPSERT — “UPDATE 或 INSERT”。

conflict_target：通过选择仲裁者索引来指定哪些行与 ON CONFLICT 在其上采取可替代动作的行相冲突。要么执行唯一索引推断，要么显式命名一个 约束。对于 ON CONFLICT DO NOTHING 来说， 它对于指定一个 conflict_target 是可选的。 在被省略时，与所有有效约束（以及唯一索引）的冲突都会被处理。对于 ON CONFLICT DO UPDATE，必须 提供一个 conflict_target。

conflict_action：conflict_action 指定一个可替换的 ON CONFLICT 动作。它可以是 DO NOTHING，也可以是一个指定在冲突情况下 要被执行的 UPDATE 动作细节的 DO UPDATE 子句。ON CONFLICT DO UPDATE 中的 SET 和 WHERE 子句能够使用该表的名称（或者别名） 访问现有的行，并且可以用特殊的被排除 表访问要插入的行。这个动作要求被排除列所在目标表的任何列上的 SELECT 特权。

注意所有行级 BEFORE INSERT 触发器的效果都会 反映在被排除值中，因为那些效果可能会 让该行避免被插入。

index_column_name：一个 table_name 列 的名称。它被用来推断仲裁者索引。它遵循 CREATE INDEX 格式。这要求 index_column_name 上的 SELECT 特权。

index_expression：和 index_column_name 类似，但是 被用来推断出现在索引定义中的 table_name 列（非简单列）上的 表达式。遵循 CREATE INDEX 格式。这要求 任何出现在 index_expression 中的列上的 SELECT 特权。

collation：指定时，强制相应的 index_column_name 或 index_expression 使用一种特定的排序规则以便在推断期间能被匹配上。通常 会被省略，因为排序规则通常不会影响约束

违背的发生。遵循 CREATE INDEX 格式。

opclass：指定时，强制相应的 index_column_name 或 index_expression 使用特定的操作符类以便在推断期间能被匹配上。通常会被省略， 因为相等语义在一种类型的操作符类 之间都是等价的，或者因为足以信任已定义的唯一索引具有适当的 相等定义。遵循 CREATE INDEX 格式。

index_predicate：用于允许推断部分唯一索引。任何满足该谓词（不一定需要真的是 部分索引）的索引都能被推断。遵循 CREATE INDEX 格式。这要求任何出现在 index_predicate 中的列上 的 SELECT 特权。

constraint_name：用名称显式地指定一个仲裁者约束， 而不是推断一个约束或者索引。

condition：一个能返回 boolean 值的表达式。只有让这个表达式返回 true 的行才将被更新，不过在采用 ON CONFLICT DO UPDATE 动作时所有的行都会被锁定。 注意 condition 会被最后计算，即一个冲突 被标识为要更新的候选对象之后。

注意不支持把排除约束作为 ON CONFLICT DO UPDATE 的 仲裁者。在所有的情况中，只支持 NOT DEFERRABLE 约束和 唯一索引作为仲裁者。

带有 ON CONFLICT DO UPDATE 子句的 INSERT 是一种“确定性的” 语句。这表明不允许该命令影响任何单个现有行超过一次，如果发生则会 发生一个基数违背错误。要插入的行不应该在仲裁者索引或约束所限制的 属性上相重复。

注意，当前不支持用分区表上的 INSERT 的 ON CONFLICT DO UPDATE 子句更新冲突行的分区键，因为那样会让行移动到新的分区中。

示例

```
--创建表 tpcds.reason_t2。
postgres=# CREATE TABLE tpcds.reason_t2
(
  r_reason_sk    integer,
  r_reason_id    character(16),
  r_reason_desc  character(100)
);
--向表中插入一条记录。
postgres=# INSERT INTO tpcds.reason_t2(r_reason_sk, r_reason_id, r_reason_desc)
VALUES (1, 'AAAAAAAABAAAAAAA', 'reason1');
--向表中插入一条记录，和上一条语法等效。
postgres=# INSERT INTO tpcds.reason_t2 VALUES (2, 'AAAAAAAABAAAAAAA',
'reason2');
```

```
--向表中插入多条记录。
postgres=# INSERT INTO tpcds.reason_t2 VALUES (3,
'AAAAAAAACAAAAAAA','reason3'),(4, 'AAAAAAAADAAAAAAA', 'reason4'),(5,
'AAAAAAAAEAAAAAAA','reason5');
--向表中插入 tpcds.reason 中 r_reason_sk 小于 5 的记录。
postgres=# INSERT INTO tpcds.reason_t2 SELECT * FROM tpcds.reason WHERE
r_reason_sk <5;
--对表创建唯一索引
postgres=# CREATE UNIQUE INDEX reason_t2_u_index ON
tpcds.reason_t2(r_reason_sk);
--向表中插入多条记录，如果冲突则更新冲突数据行中 r_reason_id 字段为
'BBBBBBBBCAAAAAAA'。
postgres=# INSERT INTO tpcds.reason_t2 VALUES (5,
'BBBBBBBBCAAAAAAA','reason5'),(6, 'AAAAAAAADAAAAAAA', 'reason6') ON DUPLICATE
KEY UPDATE r_reason_id = 'BBBBBBBBCAAAAAAA';
--删除表 tpcds.reason_t2。
postgres=# DROP TABLE tpcds.reason_t2;
```

优化建议

VALUES

通过 insert 语句批量插入数据时，建议将多条记录合并入一条语句中执行插入，以提高数据加载性能。例如，INSERT INTO sections VALUES (30, 'Administration', 31, 1900)、(40, 'Development', 35, 2000)、 (50, 'Development' , 60 , 2001)。

3.8.159LOCK

功能描述

LOCK TABLE 获取表级锁。

GBase 8s 在为一个引用了表的命令自动请求锁时，尽可能选择最小限制的锁模式。如果用户需要一种更为严格的锁模式，可以使用 LOCK 命令。例如，一个应用是在 Read committed 隔离级别上运行事务，并且它需要保证表中的数据在事务的运行过程中不被修改。为实现这个目的，则可以在查询之前对表使用 SHARE 锁模式进行锁定。这样将防止数据不被并发修改，从而保证后续的查询可以读到已提交的持久化的数据。因为 SHARE 锁模式与任何写操作需要的 ROW EXCLUSIVE 模式冲突，并且 LOCK TABLE name IN SHARE MODE 语句将等到所有当前持有 ROW EXCLUSIVE 模式锁的事务提交或回滚后才能执行。因此，一旦获得该锁，就不会存在未提交的写操作，并且其他操作也只能等到该锁释放之后才能开始。

注意事项

LOCK TABLE 只能在一个事务块的内部有用，因为锁在事务结束时就会被释放。出现在任意事务块外面的 LOCK TABLE 都会报错。

如果没有声明锁模式，缺省为最严格的模式 ACCESS EXCLUSIVE。

LOCK TABLE … IN ACCESS SHARE MODE 需要在目标表上有 SELECT 权限。所有其他形式的 LOCK 需要 UPDATE 和/或 DELETE 权限。

没有 UNLOCK TABLE 命令，锁总是在事务结束时释放。

LOCK TABLE 只处理表级的锁，因此那些带“ROW”字样的锁模式都是有歧义的。这些模式名称通常可理解为用户试图在一个被锁定的表中获取行级的锁。同样，ROW EXCLUSIVE 模式也是一个可共享的表级锁。注意，只要是涉及到 LOCK TABLE ，所有锁模式都有相同的语意，区别仅在于规则中锁与锁之间是否冲突，规则请参见表 1。

如果没有打开 xc_maintenance_mode 参数，那么对系统表申请 ACCESS EXCLUSIVE 级别锁将报错。

语法格式

```
LOCK [ TABLE ] {[ ONLY ] name [, ...]| {name [ * ]} [, ...]}
    [ IN {ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE
| SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE} MODE ]
    [ NOWAIT ];
```

参数说明

表 3-31 冲突的锁模式

请求的锁模式 / 当前锁模式	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHAR	-	-	-	-	-	-	-	X

请求的锁模式 / 当前锁模式	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
E								
ROW SHARE	-	-	-	-	-	-	X	X
ROW EXCLUSIVE	-	-	-	-	X	X	X	X
SHARE UPDATE EXCLUSIVE	-	-	-	X	X	X	X	X
SHARE	-	-	X	X	-	X	X	X
SHARE ROW EXCL	-	-	X	X	X	X	X	X

请求的锁模式 / 当前锁模式	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
USIVE								
EXCLUSIVE	-	X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

LOCK 的参数说明如下所示：

- name

  要锁定的表的名称，可以有模式修饰。

  LOCK TABLE 命令中声明的表的顺序就是上锁的顺序。

  取值范围：已存在的表名。

- ONLY

  如果指定 ONLY，只有该表被锁定。如果没有声明，该表和他的所有子表将都被锁定。

- ACCESS SHARE

  ACCESS 锁只允许对表进行读取，而禁止对表进行修改。所有对表进行读取而不修改的 SQL 语句都会自动请求这种锁。例如，SELECT 命令会自动在被引用的表上请求一个这种锁。

- ROW SHARE

  与 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。

SELECT FOR UPDATE 和 SELECT FOR SHARE 命令会自动在目标表上请求 ROW SHARE 锁（且所有被引用但不是 FOR SHARE/FOR UPDATE 的其他表上，还会自动加上 ACCESS SHARE 锁）。

- ROW EXCLUSIVE

与 ROW SHARE 锁相同，ROW EXCLUSIVE 允许并发读取表，但是禁止修改表中数据。UPDATE，DELETE，INSERT 命令会自动在目标表上请求这个锁（且所有被引用的其他表上还会自动加上的 ACCESS SHARE 锁）。通常情况下，所有会修改表数据的命令都会请求表的 ROW EXCLUSIVE 锁。

- SHARE UPDATE EXCLUSIVE

这个模式保护一个表的模式不被并发修改，以及禁止在目标表上执行垃圾回收命令（VACUUM ）。

VACUUM（不带 FULL 选项），ANALYZE，CREATE INDEX CONCURRENTLY 命令会自动请求这样的锁。

- SHARE

SHARE 锁允许并发的查询，但是禁止对表进行修改。

CREATE INDEX（不带 CONCURRENTLY 选项）语句会自动请求这种锁。

- SHARE ROW EXCLUSIVE

SHARE ROW EXCLUSIVE 锁禁止对表进行任何的并发修改，而且是独占锁，因此一个会话中只能获取一次。

任何 SQL 语句都不会自动请求这个锁模式。

- EXCLUSIVE

EXCLUSIVE 锁允许对目标表进行并发查询，但是禁止任何其他操作。

这个模式只允许并发加 ACCESS SHARE 锁，也就是说，只有对表的读动作可以和持有这个锁模式的事务并发执行。

任何 SQL 语句都不会在用户表上自动请求这个锁模式。然而在某些操作的时候，会在某些系统表上请求它。

- ACCESS EXCLUSIVE

这个模式保证其所有者（事务）是可以访问该表的唯一事务。

ALTER TABLE，DROP TABLE，TRUNCATE，REINDEX 命令会自动请求这种锁。

在 LOCK TABLE 命令没有明确声明需要的锁模式时，它是缺省锁模式。

- NOWAIT

声明 LOCK TABLE 不去等待任何冲突的锁释放，如果无法立即获取该锁，该命令退出并且发出一个错误信息。

在不指定 NOWAIT 的情况下获取表级锁时，如果有其他互斥锁存在的话，则等待其他锁的释放。

示例

```
--在执行删除操作时对一个有主键的表进行 SHARE ROW EXCLUSIVE 锁。
postgres=# CREATE TABLE tpcds.reason_t1 AS TABLE tpcds.reason;

postgres=# START TRANSACTION;

postgres=# LOCK TABLE tpcds.reason_t1 IN SHARE ROW EXCLUSIVE MODE;

postgres=# DELETE FROM tpcds.reason_t1 WHERE r_reason_desc IN(SELECT
r_reason_desc FROM tpcds.reason_t1 WHERE r_reason_sk < 6 );

postgres=# DELETE FROM tpcds.reason_t1 WHERE r_reason_sk = 7;

postgres=# COMMIT;

--删除表 tpcds.reason_t1。
postgres=# DROP TABLE tpcds.reason_t1;
```

3.8.160MERGE INTO

功能描述

通过 MERGE INTO 语句，将目标表和源表中数据针对关联条件进行匹配，若关联条件匹配时对目标表进行 UPDATE，无法匹配时对目标表执行 INSERT。此语法可以很方便地用来合并执行 UPDATE 和 INSERT，避免多次执行。

注意事项

进行 MERGE INTO 操作的用户需要同时拥有目标表的 UPDATE 和 INSERT 权限，以及源表的 SELECT 权限。

语法格式

```
MERGE [/*+ plan_hint */] INTO table_name [ partition_clause ] [ [ AS ] alias ]
USING { { table_name | view_name } | subquery } [ [ AS ] alias ]
ON ( condition )
[
  WHEN MATCHED THEN
  UPDATE SET { column_name = { expression | subquery | DEFAULT } |
           ( column_name [, ...] ) = ( { expression | subquery | DEFAULT }
[, ...] ) } [, ...]
  [ WHERE condition ]
]
[
  WHEN NOT MATCHED THEN
  INSERT { DEFAULT VALUES |
  [ ( column_name [, ...] ) ] VALUES ( { expression | subquery | DEFAULT } [, ...] )
[, ...] [ WHERE condition ] }
];
NOTICE: 'subquery' in the UPDATE and INSERT clauses are only avaliable in
CENTRALIZED mode!
```

参数说明

- plan_hint 子句

  以/*+ */的形式在 MERGE 关键字后，用于对 MERGE 对应的语句块生成的计划进行 hint 调优，详细用法请参见章节使用 Plan Hint 进行调优。每条语句中只有第一个/*+ plan_hint */注释块会作为 hint 生效，里面可以写多条 hint。

- INTO 子句

  指定正在更新或插入的目标表。

- talbe_name

  目标表的表名。

- alias

  目标表的别名。

  取值范围：字符串，符合标识符命名规范。

- partition_clause

  指定分区 MERGE 操作：

```
  PARTITION { ( partition_name ) | FOR ( partition_value [, ...] ) } |
```

```
SUBPARTITION { ( subpartition_name ) | FOR ( subpartition_value [, ...] ) }
```

关键字详见 SELECT 一节介绍。

如果 value 子句的值和指定分区不一致，会抛出异常。

示例详见 CREATE TABLE SUBPARTITION。

- alias

  目标表的别名。

  取值范围：字符串，符合标识符命名规范。

- USING 子句

  指定源表，源表可以为表、视图或子查询。

- ON 子句

  关联条件，用于指定目标表和源表的关联条件。不支持更新关联条件中的字段。

- WHEN MATCHED 子句

  当源表和目标表中数据针对关联条件可以匹配上时，选择 WHEN MATCHED 子句进行 UPDATE 操作。

  不支持更新系统表、系统列。

- WHEN NOT MATCHED 子句

  当源表和目标表中数据针对关联条件无法匹配时，选择 WHEN NOT MATCHED 子句进行 INSERT 操作。

  不支持 INSERT 子句中包含多个 VALUES。

  WHEN MATCHED 和 WHEN NOT MATCHED 子句顺序可以交换，可以缺省其中一个，但不能同时缺省，不支持同时指定两个 WHEN MATCHED 或 WHEN NOT MATCHED 子句。

- DEFAULT

  用对应字段的缺省值填充该字段。

  如果没有缺省值，则为 NULL。

- WHERE condition

  UPDATE 子句和 INSERT 子句的条件，只有在条件满足时才进行更新操作，可缺省。不支持 WHERE 条件中引用系统列。不建议使用 int 等数值类型作为 condition，因为 int 等数

值类型可以隐式转换为 bool 值（非 0 值隐式转换为 true，0 转换为 false），可能导致非预期的结果。

示例

```
-- 创建目标表 products 和源表 newproducts，并插入数据
CREATE TABLE products
(
product_id INTEGER,
product_name VARCHAR2(60),
category VARCHAR2(60)
);
INSERT INTO products VALUES (1501, 'vivitar 35mm', 'electrncs');
INSERT INTO products VALUES (1502, 'olympus is50', 'electrncs');
INSERT INTO products VALUES (1600, 'play gym', 'toys');
INSERT INTO products VALUES (1601, 'lamaze', 'toys');
INSERT INTO products VALUES (1666, 'harry potter', 'dvd');
CREATE TABLE newproducts
(
product_id INTEGER,
product_name VARCHAR2(60),
category VARCHAR2(60)
);
INSERT INTO newproducts VALUES (1502, 'olympus camera', 'electrncs');
INSERT INTO newproducts VALUES (1601, 'lamaze', 'toys');
INSERT INTO newproducts VALUES (1666, 'harry potter', 'toys');
INSERT INTO newproducts VALUES (1700, 'wait interface', 'books');
-- 进行 MERGE INTO 操作
MERGE INTO products p
USING newproducts np
ON (p.product_id = np.product_id)
WHEN MATCHED THEN
  UPDATE SET p.product_name = np.product_name, p.category = np.category WHERE
p.product_name != 'play gym'
WHEN NOT MATCHED THEN
  INSERT VALUES (np.product_id, np.product_name, np.category) WHERE np.category
= 'books';
MERGE 4
-- 查询更新后的结果
SELECT * FROM products ORDER BY product_id;
 product_id |  product_name  | category
------------+----------------+-----------
```

```
      1501 | vivitar 35mm   | electrncs
      1502 | olympus camera | electrncs
      1600 | play gym       | toys
      1601 | lamaze         | toys
      1666 | harry potter   | toys
      1700 | wait interface | books
(6 rows)
-- 删除表
DROP TABLE products;
DROP TABLE newproducts;
```

3.8.161 MOVE

功能描述

MOVE 在不检索数据的情况下重新定位一个游标。MOVE 的作用类似于 FETCH 命令，但只是重定位游标而不返回行。

注意事项

无。

语法格式

```
MOVE [ direction [ FROM | IN ] ] cursor_name;
```

其中 direction 子句为可选参数。

```
 NEXT
| PRIOR
| FIRST
| LAST
| ABSOLUTE count
| RELATIVE count
| count
| ALL
| FORWARD
| FORWARD count
| FORWARD ALL
| BACKWARD
| BACKWARD count
| BACKWARD ALL
```

参数说明

MOVE 命令的参数与 FETCH 的相同，详细请参见 FETCH 的参数说明。

说明：成功完成时，MOVE 命令将返回一个“MOVE count”的标签，count 是一个使用相同参数的 FETCH 命令会返回的行数（可能为零）。

示例

```
--开始一个事务。
postgres=# START TRANSACTION;
--定义一个名为 cursor1 的游标。
postgres=# CURSOR cursor1 FOR SELECT * FROM tpcds.reason;
--忽略游标 cursor1 的前 3 行。
postgres=# MOVE FORWARD 3 FROM cursor1;
--抓取游标 cursor1 的前 4 行。
postgres=# FETCH 4 FROM cursor1;
 r_reason_sk |   r_reason_id    |
r_reason_desc
-------------+------------------+-----------------------------------------
-----------------------------------------------------
           4 | AAAAAAAAEAAAAAAA | Not the product that was ordred
           5 | AAAAAAAAFAAAAAAA | Parts missing
           6 | AAAAAAAAGAAAAAAA | Does not work with a product that I have
           7 | AAAAAAAAHAAAAAAA | Gift exchange
(4 rows)
--关闭游标。
postgres=# CLOSE cursor1;
--结束一个事务。
postgres=# END;
```

相关命令

CLOSE，FETCH

3.8.162 PREDICT BY

功能描述

利用完成训练的模型进行推测任务。

注意事项

调用的模型名称在系统表 gs_model_warehouse 中可查看到。

语法格式

```
PREDICT BY model_name [ (FEATURES attribute [, attribute] +]) ]
```

参数说明

- model_name

  用于推测任务的模型名称。

  取值范围：字符串，需要符合标识符的命名规则。

- attribute

  推测任务的输入特征列名。

  取值范围：字符串，需要符合标识符的命名规则。

示例

```
SELECT id, PREDICT BY price_model (FEATURES size,lot), price
FROM houses;
```

相关命令

CREATE MODEL，DROP MODEL

3.8.163 PREPARE

功能描述

创建一个预备语句。

预备语句是服务端的对象，可以用于优化性能。在执行 PREPARE 语句的时候，指定的查询被解析、分析、重写。当随后发出 EXECUTE 语句的时候，预备语句被规划和执行。这种设计避免了重复解析、分析工作。PREPARE 语句创建后在整个数据库会话期间一直存在，一旦创建成功，即便是在事务块中创建，事务回滚，PREPARE 也不会删除。只能通过显式调用 DEALLOCATE 进行删除，会话结束时，PREPARE 也会自动删除。

注意事项

无。

语法格式

```
PREPARE name [ ( data_type [, ...] ) ] AS statement;
```

参数说明

- name

  指定预备语句的名称。它必须在该会话中是唯一的。

- data_type

  参数的数据类型。

- statement

  是 SELECT INSERT、UPDATE、DELETE、MERGE INTO 或 VALUES 语句之一。

示例

请参见 EXECUTE 的示例。

相关命令

DEALLOCATE

3.8.164PREPARE TRANSACTION

功能描述

为当前事务做两阶段提交的准备。

在命令之后，事务就不再和当前会话关联了；它的状态完全保存在磁盘上，它被提交成功的可能性非常高，即使是在请求提交之前数据库发生了崩溃也如此。

一旦准备好了，一个事务就可以在稍后用 COMMIT PREPARED 或 ROLLBACK PREPARED 命令分别进行提交或者回滚。这些命令可以从任何会话中发出，而不光是最初执行事务的那个会话。

从发出命令的会话的角度来看，PREPARE TRANSACTION 不同于 ROLLBACK：在执行它之后，就不再有活跃的当前事务了，并且预备事务的效果无法见到（在事务提交的时候其效果会再次可见）。

如果 PREPARE TRANSACTION 因为某些原因失败，那么它就会变成一个 ROLLBACK，当前事务被取消。

注意事项

事务功能由数据库自动维护，不应显式使用事务功能。

在运行 PREPARE TRANSACTION 命令时，必须在 postgresql.conf 配置文件中增大 max_prepared_transactions 的数值。建议至少将其设置为等于 max_connections，这样每个会话都可以有一个等待中的预备事务。

语法格式

```
PREPARE TRANSACTION transaction_id;
```

参数说明

- transaction_id

待提交事务的标识符，用于后面在 COMMIT PREPARED 或 ROLLBACK PREPARED 的时候标识这个事务。它不能和任何当前预备事务已经使用了的标识符同名。

取值范围：标识符必须以字符串文本的方式书写，并且必须小于 200 字节长。

相关命令

COMMIT PREPARED，ROLLBACK PREPARED

3.8.165PURGE

功能描述

使用 PURGE 语句可以实现如下功能：

从回收站中清理表或索引，并释放对象相关的全部空间。

清理回收站。

清理回收站中指定表空间的对象。

注意事项

清除（PURGE）操作支持：表（PURGE TABLE）、索引（PURGE INDEX）、回收站（PURGE RECYCLEBIN）。

执行 PURGE 操作的权限要求如下：

PURGE TABLE：用户必须是表的所有者，且用户必须拥有表所在模式的 USAGE 权限，系统管理员默认拥有此权限。

PURGE INDEX：用户必须是索引的所有者，用户必须拥有索引所在模式的 USAGE 权限，系统管理员默认拥有此权限。

PURGE RECYCLEBIN：普通用户只能清理回收站中当前用户拥有的对象，且用户必须拥有对象所在模式的 USAGE 权限，系统管理员默认可以清理回收站所有对象。

语法格式

```
PURGE { TABLE [schema_name.]table_name
        | INDEX index_name
        | RECYCLEBIN
      }
```

参数说明

- schema_name

  模式名。

- TABLE [schema_name.] table_name

  清空回收站中指定的表。

- INDEX index_name

  清空回收站中指定的索引。

- RECYCLEBIN

  清空回收站中的对象。

示例

```
-- 创建表空间 reason_table_space
postgres=# CREATE TABLESPACE REASON_TABLE_SPACE1 owner tpcds RELATIVE location
'tablespace/tsp_reason1';
-- 在表空间创建表 tpcds.reason_t1
postgres=# CREATE TABLE tpcds.reason_t1
 (
  r_reason_sk    integer,
  r_reason_id    character(16),
  r_reason_desc  character(100)
  ) tablespace reason_table_space1;
-- 在表空间创建表 tpcds.reason_t2
postgres=# CREATE TABLE tpcds.reason_t2
 (
  r_reason_sk    integer,
  r_reason_id    character(16),
  r_reason_desc  character(100)
  ) tablespace reason_table_space1;
-- 在表空间创建表 tpcds.reason_t3
postgres=# CREATE TABLE tpcds.reason_t3
 (
  r_reason_sk    integer,
  r_reason_id    character(16),
  r_reason_desc  character(100)
  ) tablespace reason_table_space1;
-- 对表 tpcds.reason_t1 创建索引
```

```
postgres=# CREATE INDEX index_t1 on tpcds.reason_t1(r_reason_id);
postgres=# DROP TABLE tpcds.reason_t1;
postgres=# DROP TABLE tpcds.reason_t2;
postgres=# DROP TABLE tpcds.reason_t3;
--查看回收站
postgres=# SELECT rcyname,rcyoriginname,rcytablespace FROM GS_RECYCLEBIN;
        rcyname         | rcyoriginname | rcytablespace
------------------------+---------------+---------------
 BIN$16409$2CEE988==$0 | reason_t1     |         16408
 BIN$16412$2CF2188==$0 | reason_t2     |         16408
 BIN$16415$2CF2EC8==$0 | reason_t3     |         16408
 BIN$16418$2CF3EC8==$0 | index_t1      |             0
(4 rows)
--PURGE 清除表
postgres=# PURGE TABLE tpcds.reason_t3;
postgres=# SELECT rcyname,rcyoriginname,rcytablespace FROM GS_RECYCLEBIN;
        rcyname         | rcyoriginname | rcytablespace
------------------------+---------------+---------------
 BIN$16409$2CEE988==$0 | reason_t1     |         16408
 BIN$16412$2CF2188==$0 | reason_t2     |         16408
 BIN$16418$2CF3EC8==$0 | index_t1      |             0
(3 rows)
--PURGE 清除索引
postgres=# PURGE INDEX tindex_t1;
postgres=# SELECT rcyname,rcyoriginname,rcytablespace FROM GS_RECYCLEBIN;
        rcyname         | rcyoriginname | rcytablespace
------------------------+---------------+---------------
 BIN$16409$2CEE988==$0 | reason_t1     |         16408
 BIN$16412$2CF2188==$0 | reason_t2     |         16408
(2 rows)
--PURGE 清除回收站所有对象
postgres=# PURGE recyclebin;
postgres=# SELECT rcyname,rcyoriginname,rcytablespace FROM GS_RECYCLEBIN;
        rcyname         | rcyoriginname | rcytablespace
------------------------+---------------+---------------
(0 rows)
```

3.8.166REASSIGN OWNED

功能描述

修改数据库对象的属主。

REASSIGN OWNED 要求系统将所有 old_roles 拥有的数据库对象的属主更改为 new_role。

注意事项

REASSIGN OWNED 常用于在删除角色之前的准备工作。

执行 REASSIGN OWNED 需要有原角色和目标角色上的权限。

语法格式

REASSIGN OWNED BY old_role [, ...] TO new_role;

参数说明

- old_role

  旧属主的角色名。

- new_role

  将要成为这些对象属主的新角色的名称。

示例

无。

3.8.167REFRESH INCREMENTAL MATERIALIZED VIEW

功能描述

REFRESH INCREMENTAL MATERIALIZED VIEW 会以增量刷新的方式对物化视图进行刷新。

注意事项

增量刷新仅支持增量物化视图。

刷新物化视图需要当前用户拥有基表的 SELECT 权限。

语法格式

```
REFRESH [ INCREMENTAL ] MATERIALIZED VIEW name;
```

参数说明

- mv_name

  要刷新的物化视图的名称。

示例

```
--创建一个普通表
postgres=# CREATE TABLE my_table (c1 int, c2 int);
--创建增量物化视图
postgres=# CREATE INCREMENTAL MATERIALIZED VIEW my_imv AS SELECT * FROM my_table;
--基表写入数据
postgres=# INSERT INTO my_table VALUES(1,1),(2,2);
--对增量物化视图 my_imv 进行增量刷新
postgres=# REFRESH INCREMENTAL MATERIALIZED VIEW my_imv;
```

相关命令

ALTER MATERIALIZED VIEW，CREATE INCREMENTAL MATERIALIZED VIEW，CREATE MATERIALIZED VIEW，CREATE TABLE， DROP MATERIALIZED VIEW，REFRESH MATERIALIZED VIEW

3.8.168REFRESH MATERIALIZED VIEW

功能描述

REFRESH MATERIALIZED VIEW 会以全量刷新的方式对物化视图进行刷新。

注意事项

全量刷新既可以对全量物化视图执行，也可以对增量物化视图执行。

刷新物化视图需要当前用户拥有基表的 SELECT 权限。

语法格式

```
REFRESH [ INCREMENTAL ] MATERIALIZED VIEW name;
```

参数说明

- mv_name

  要刷新的物化视图的名称。

示例

```
--创建一个普通表
postgres=# CREATE TABLE my_table (c1 int, c2 int);
--创建全量物化视图
postgres=# CREATE MATERIALIZED VIEW my_mv AS SELECT * FROM my_table;
--创建增量物化视图
postgres=# CREATE INCREMENTAL MATERIALIZED VIEW my_imv AS SELECT * FROM my_table;
```

```
--基表写入数据
postgres=# INSERT INTO my_table VALUES(1,1),(2,2);
--对全量物化视图 my_mv 进行全量刷新
postgres=# REFRESH MATERIALIZED VIEW my_mv;
--对增量物化视图 my_imv 进行全量刷新
postgres=# REFRESH MATERIALIZED VIEW my_imv;
```

相关命令

ALTER MATERIALIZED VIEW，CREATE INCREMENTAL MATERIALIZED VIEW，CREATE MATERIALIZED VIEW，CREATE TABLE， DROP MATERIALIZED VIEW，REFRESH INCREMENTAL MATERIALIZED VIEW

3.8.169REINDEX

功能描述

为表中的数据重建索引。

在以下几种情况下需要使用 REINDEX 重建索引：

索引崩溃，并且不再包含有效的数据。

索引变得“臃肿”，包含大量的空页或接近空页。

为索引更改了存储参数（例如填充因子），并且希望这个更改完全生效。

使用 CONCURRENTLY 选项创建索引失败，留下了一个“非法”索引。

注意事项

REINDEX DATABASE 和 SYSTEM 这种形式的重建索引不能在事务块中执行。

REINDEX CONCURRENTLY 这种形式的重建索引不能在事务块中执行。

语法格式

重建普通索引。

```
REINDEX { INDEX | [INTERNAL] TABLE | DATABASE | SYSTEM } name [ FORCE ];
```

重建索引分区。

```
REINDEX  { INDEX | [INTERNAL] TABLE  } name
    PARTITION partition_name [ FORCE  ];
```

参数说明

- INDEX

重新建立指定的索引。

- INTERNAL TABLE

重建列存表或 Hadoop 内表的 Desc 表的索引，如果表有从属的“TOAST”表，则这个表也会重建索引。

- TABLE

重新建立指定表的所有索引，如果表有从属的“TOAST”表，则这个表也会重建索引。如果表上有索引已经被 alter unusable 失效，则这个索引无法被重新创建。

- DATABASE

重建当前数据库里的所有索引。

- SYSTEM

在当前数据库上重建所有系统表上的索引。不会处理在用户表上的索引。

- CONCURRENTLY

以不阻塞 DML 的方式重建索引（加 ShareUpdateExclusiveLock 锁）。重建索引时，一般会阻塞其他语句对该索引所依赖表的访问。指定此关键字，可以实现重建过程中不阻塞 DML。

此选项只能指定一个索引的名称。

普通 REINDEX 命令可以在事务内执行，但是 REINDEX CONCURRENTLY 不可以在事务内执行。

列存表、全局分区表和临时表不支持 CONCURRENTLY 方式重建索引。

REINDEX SYSTEM CONCURRENTLY 不会执行任何操作，因为系统表不支持在线重建索引。

说明：- 重建索引时指定此关键字，需要执行先后两次对该表的全表扫描来完成 build，第一次扫描的时候创建新索引，不阻塞读写操作；第二次扫描的时候合并更新第一次扫描到目前为止发生的变更。

因为需要执行两次对表的扫描和 build，且必须等待现有的所有可能对该表执行修改的事务结束，所以该索引的重建比正常耗时更长，同时带来的 CPU 和 I/O 消耗对其他业务也会造成影响。

如果在索引构建时发生失败，那会留下一个“不可用”的索引。这个索引会被查询忽略，

但它仍消耗更新开销。这种情况推荐的恢复方法是删除该索引并尝试再次 CONCURRENTLY 重建索引。

由于在第二次扫描之后，索引构建必须等待任何持有早于第二次扫描拿的快照的事务终止，而且建索引时加的 ShareUpdateExclusiveLock 锁（4 级）会和大于等于 4 级的锁冲突，因此在创建这类索引时，容易引发卡住（hang）或者死锁问题。例如：

两个会话对同一个表重建 CONCURRENTLY 索引，会引起死锁问题；

两个会话，一个对表重建 CONCURRENTLY 索引，一个 drop table，会引起死锁问题；

三个会话，会话 1 先对表 a 加锁，不提交，会话 2 接着对表 b 重建 CONCURRENTLY 索引，会话 3 接着对表 a 执行写入操作，在会话 1 事务未提交之前，会话 2 会一直被阻塞；

将事务隔离级别设置成可重复读（默认为读已提交），起两个会话，会话 1 起事务对表 a 执行写入操作，不提交，会话 2 对表 b 重建 CONCURRENTLY 索引，在会话 1 事务未提交之前，会话 2 会一直被阻塞。

- name

  需要重建索引的索引、表、数据库的名称。表和索引可以有模式修饰。

  说明：REINDEX DATABASE 和 SYSTEM 只能重建当前数据库的索引，所以 name 必须和当前数据库名称相同。

- FORCE

  无效选项，会被忽略。

- partition_name

  需要重建索引的分区的名称或者索引分区的名称。

  取值范围：

  如果前面是 REINDEX INDEX，则这里应该指定索引分区的名称；

  如果前面是 REINDEX TABLE，则这里应该指定分区的名称；

  如果前面是 REINDEX INTERNAL TABLE，则这里应该指定列存分区表的分区的名称。

  须知：REINDEX DATABASE 和 SYSTEM 这种形式的重建索引不能在事务块中执行。

示例

```
--创建一个行存表 tpcds.customer_t1，并在 tpcds.customer_t1 表上的 c_customer_sk
字段创建索引。
postgres=# CREATE TABLE tpcds.customer_t1
(
    c_customer_sk             integer               not null,
    c_customer_id             char(16)              not null,
    c_current_cdemo_sk        integer                       ,
    c_current_hdemo_sk        integer                       ,
    c_current_addr_sk         integer                       ,
    c_first_shipto_date_sk    integer                       ,
    c_first_sales_date_sk     integer                       ,
    c_salutation              char(10)                      ,
    c_first_name              char(20)                      ,
    c_last_name               char(30)                      ,
    c_preferred_cust_flag     char(1)                       ,
    c_birth_day               integer                       ,
    c_birth_month             integer                       ,
    c_birth_year              integer                       ,
    c_birth_country           varchar(20)                   ,
    c_login                   char(13)                      ,
    c_email_address           char(50)                      ,
    c_last_review_date        char(10)
)
WITH (orientation = row)
postgres=# CREATE INDEX tpcds_customer_index1 ON tpcds.customer_t1
(c_customer_sk);
postgres=# INSERT INTO tpcds.customer_t1 SELECT * FROM tpcds.customer WHERE
c_customer_sk < 10;
--重建一个单独索引。
postgres=# REINDEX INDEX tpcds.tpcds_customer_index1;
--实时重建一个单独索引。
postgres=# REINDEX INDEX CONCURRENTLY tpcds.tpcds_customer_index1;
--重建表 tpcds.customer_t1 上的所有索引。
postgres=# REINDEX TABLE tpcds.customer_t1;
--实时重建表 tpcds.customer_t1 上的所有索引。
postgres=# REINDEX TABLE CONCURRENTLY tpcds.customer_t1;
--删除 tpcds.customer_t1 表。
postgres=# DROP TABLE tpcds.customer_t1;
```

优化建议

INTERNAL TABLE

此种情况大多用于故障恢复，不建议进行并发操作。

DATABASE

不能在事务中 reindex database。

SYSTEM

不能在事务中 reindex 系统表。

3.8.170RELEASE SAVEPOINT

功能描述

RELEASE SAVEPOINT 删除一个当前事务先前定义的保存点。

把一个保存点删除就令其无法作为回滚点使用，除此之外它没有其它用户可见的行为。它并不能撤销在保存点建立起来之后执行的命令的影响。要撤销那些命令可以使用 ROLLBACK TO SAVEPOINT 。在不再需要的时候删除一个保存点可以令系统在事务结束之前提前回收一些资源。

RELEASE SAVEPOINT 也删除所有在指定的保存点建立之后的所有保存点。

注意事项

不能 RELEASE 一个没有定义的保存点，语法上会报错。

如果事务在回滚状态，则不能释放保存点。

如果多个保存点拥有同样的名称，只有最近定义的那个才被释放。

语法格式

```
RELEASE [ SAVEPOINT ] savepoint_name;
```

参数说明

- savepoint_name

  要删除的保存点的名称。

示例

```
--创建一个新表。
CREATE TABLE tpcds.table1(a int);

--开启事务。
START TRANSACTION;
```

```
--插入数据。
INSERT INTO tpcds.table1 VALUES (3);

--建立保存点。
SAVEPOINT my_savepoint;

--插入数据。
INSERT INTO tpcds.table1 VALUES (4);

--删除保存点。
RELEASE SAVEPOINT my_savepoint;

--提交事务。
COMMIT;

--查询表的内容，会同时看到 3 和 4。
SELECT * FROM tpcds.table1;

--删除表。
DROP TABLE tpcds.table1;
```

相关链接

SAVEPOINT，ROLLBACK TO SAVEPOINT

3.8.171RESET

功能描述

RESET 将指定的运行时参数恢复为缺省值。这些参数的缺省值是指 postgresql.conf 配置文件中所描述的参数缺省值。

RESET 命令与如下命令的作用相同：

SET configuration_parameter TO DEFAULT

注意事项

RESET 的事务性行为和 SET 相同：它的影响将会被事务回滚撤销。

语法格式

```
RESET {configuration_parameter | CURRENT_SCHEMA | TIME ZONE | TRANSACTION
ISOLATION LEVEL | SESSION AUTHORIZATION | ALL };
```

参数说明

- configuration_parameter

  运行时参数的名称。

  取值范围：可以使用 SHOW ALL 命令查看运行时参数。

- CURRENT_SCHEMA

  当前模式。

- TIME ZONE

  时区。

- TRANSACTION ISOLATION LEVEL

  事务的隔离级别。

- SESSION AUTHORIZATION

  当前会话的用户标识符。

- ALL

  所有运行时参数。

示例

```
--把 timezone 设为缺省值。
postgres=# RESET timezone;
--把所有参数设置为缺省值。
postgres=# RESET ALL;
```

相关命令

SET，SHOW

3.8.172 REVOKE

功能描述

REVOKE 用于撤销一个或多个角色的权限。

注意事项

非对象所有者试图在对象上 REVOKE 权限，命令按照以下规则执行：

如果授权用户没有该对象上的权限，则命令立即失败。

如果授权用户有部分权限，则只撤销那些有授权选项的权限。

如果授权用户没有授权选项，REVOKE ALL PRIVILEGES 形式将发出一个错误信息，而对于其他形式的命令而言，如果是命令中指定名称的权限没有相应的授权选项，该命令将发出一个警告。

不允许对表分区进行 REVOKE 操作，对分区表进行 REVOKE 操作会引起告警。

语法格式

回收指定表或视图上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
    ALTER | DROP | COMMENT | INDEX | VACUUM } [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收表上指定字段权限。

```
REVOKE [ GRANT OPTION FOR ]
    { {{ SELECT | INSERT | UPDATE | REFERENCES | COMMENT } ( column_name
[, ...] )}[, ...]
    | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定序列上权限，LARGE 字段属性可选，回收语句不区分序列是否为 LARGE。

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE | ALTER | DROP | COMMENT }[, ...]
    | ALL [ PRIVILEGES ] }
     ON { [ SEQUENCE ] sequence_name [, ...]
        | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定数据库上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP | ALTER | DROP | COMMENT } [, ...]
| ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
```

```
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定域上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定客户端加密主密钥上的权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | DROP } [, ...] | ALL [PRIVILEGES] }
    ON CLIENT_MASTER_KEYS client_master_keys_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定列加密密钥上的权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | DROP } [, ...] | ALL [PRIVILEGES]}
    ON COLUMN_ENCRYPTION_KEYS column_encryption_keys_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定目录上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { READ | WRITE | ALTER |DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON DIRECTORY directory_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定外部数据源上权限。

```
REVOKE [ GRANT OPTION FOR ]
   { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定外部服务器上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
```

```
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定函数上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { EXECUTE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON {FUNCTION {function_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )}
[, ...]
        | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定过程语言上权限。

```
REVOKE [ GRANT OPTION FOR ]
   { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定 NODE GROUP 权限。

```
REVOKE [ GRANT OPTION FOR ]
    { {CREATE | USAGE | COMPUTE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON NODE GROUP group_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定大对象上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定模式上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定表空间上权限。

```
REVOKE [ GRANT OPTION FOR ]
```

```
    { { CREATE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收指定类型上权限。

```
REVOKE [ GRANT OPTION FOR ]
   { { USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

回收 Data Source 对象上的权限。

```
REVOKE [ GRANT OPTION FOR ]
   { USAGE | ALL [PRIVILEGES] }
    ON DATA SOURCE src_name [, ...]
    FROM {[GROUP] role_name | PUBLIC} [, ...]
```

回收 package 对象的权限。

```
REVOKE [ GRANT OPTION FOR ]
   { { EXECUTE | ALTER | DROP | COMMENT } [, ...] | ALL [PRIVILEGES] }
   ON PACKAGE package_name [, ...]
   FROM {[GROUP] role_name | PUBLIC} [, ...]
   [ CASCADE | RESTRICT ];
```

按角色回收角色上的权限。

```
REVOKE [ ADMIN OPTION FOR ]
    role_name [, ...] FROM role_name [, ...]
    [ CASCADE | RESTRICT ];
```

回收角色上的 sysadmin 权限。

```
REVOKE ALL { PRIVILEGES | PRIVILEGE } FROM role_name;
```

回收 ANY 权限。

```
REVOKE [ ADMIN OPTION FOR ]
    { CREATE ANY TABLE | ALTER ANY TABLE | DROP ANY TABLE | SELECT ANY TABLE |
INSERT ANY TABLE |
    UPDATE ANY TABLE | DELETE ANY TABLE | CREATE ANY SEQUENCE | CREATE ANY INDEX
|
    CREATE ANY FUNCTION | EXECUTE ANY FUNCTION | CREATE ANY PACKAGE |
    EXECUTE ANY PACKAGE | CREATE ANY TYPE } [, ...]
    FROM [ GROUP ] role_name [, ...];
```

参数说明

关键字 PUBLIC 表示一个隐式定义的拥有所有角色的组。

权限类别和参数说明，请参见 GRANT 的参数说明。

任何特定角色拥有的特权包括直接授予该角色的特权、从该角色作为其成员的角色中得到的权限以及授予给 PUBLIC 的权限。因此，从 PUBLIC 收回 SELECT 特权并不一定会意味着所有角色都会失去在该对象上的 SELECT 特权，那些直接被授予的或者通过另一个角色被授予的角色仍然会拥有它。类似地，从一个用户收回 SELECT 后，如果 PUBLIC 仍有 SELECT 权限，该用户还是可以使用 SELECT。

指定 GRANT OPTION FOR 时，只撤销对该权限授权的权力，而不撤销该权限本身。

如用户 A 拥有某个表的 UPDATE 权限，及 WITH GRANT OPTION 选项，同时 A 把这个权限赋予了用户 B，则用户 B 持有的权限称为依赖性权限。当用户 A 持有的权限或者授权选项被撤销时，必须声明 CASCADE，将所有依赖性权限都撤销。

一个用户只能撤销由它自己直接赋予的权限。例如，如果用户 A 被指定授权（WITH ADMIN OPTION）选项，且把一个权限赋予了用户 B，然后用户 B 又赋予了用户 C，则用户 A 不能直接将 C 的权限撤销。但是，用户 A 可以撤销用户 B 的授权选项，并且使用 CASCADE。这样，用户 C 的权限就会自动被撤销。另外一个例子：如果 A 和 B 都赋予了 C 同样的权限，则 A 可以撤销他自己的授权选项，但是不能撤销 B 的，因此 C 仍然拥有该权限。

如果执行 REVOKE 的角色持有的权限是通过多层成员关系获得的，则具体是哪个包含的角色执行的该命令是不确定的。在这种场合下，最好的方法是使用 SET ROLE 成为特定角色，然后执行 REVOKE，否则可能导致删除了不想删除的权限，或者是任何权限都没有删除。

示例

请参考 GRANT 的示例。

相关命令

GRANT

3.8.173ROLLBACK

功能描述

回滚当前事务并取消当前事务中的所有更新。

在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤销，数据库状态回到事务开始时。

注意事项

如果不在一个事务内部发出 ROLLBACK 不会有问题，但是将抛出一个 NOTICE 信息。

语法格式

```
ROLLBACK [ WORK | TRANSACTION ];
```

参数说明

- WORK | TRANSACTION

  可选关键字。除了增加可读性，没有任何其他作用。

示例

```
--开启一个事务
postgres=# START TRANSACTION;
--取消所有更改
postgres=# ROLLBACK;
```

相关命令

COMMIT | END

3.8.174ROLLBACK PREPARED

功能描述

取消一个先前为两阶段提交准备好的事务。

注意事项

该功能仅在维护模式(GUC 参数 xc_maintenance_mode 为 on 时)下可用。该模式谨慎打开，一般供维护人员排查问题使用，一般用户不应使用该模式。

要想回滚一个预备事务，必须是最初发起事务的用户，或者是系统管理员。

事务功能由数据库自动维护，不应显式使用事务功能。

语法格式

```
ROLLBACK PREPARED transaction_id ;
```

参数说明

- transaction_id

待提交事务的标识符。它不能和任何当前预备事务已经使用了的标识符同名。

相关命令

COMMIT PREPARED，PREPARE TRANSACTION。

3.8.175ROLLBACK TO SAVEPOINT

功能描述

ROLLBACK TO SAVEPOINT 用于回滚到一个保存点，隐含地删除所有在该保存点之后建立的保存点。

回滚所有指定保存点建立之后执行的命令。保存点仍然有效，并且需要时可以再次回滚到该点。

注意事项

不能回滚到一个未定义的保存点，语法上会报错。

在保存点方面，游标有一些非事务性的行为。任何在保存点里打开的游标都会在回滚掉这个保存点之后关闭。如果一个前面打开了的游标在保存点里面，并且游标被一个 FETCH 命令影响，而这个保存点稍后回滚了，那么这个游标的位置仍然在 FETCH 让它指向的位置（也就是 FETCH 不会被回滚）。关闭一个游标的行为也不会被回滚给撤消掉。如果一个游标的操作导致事务回滚，那么这个游标就会置于不可执行状态，所以，尽管一个事务可以用 ROLLBACK TO SAVEPOINT 重新恢复，但是游标不能再使用了。

使用 ROLLBACK TO SAVEPOINT 回滚到一个保存点。使用 RELEASE SAVEPOINT 删除一个保存点，但是保留该保存点建立后执行的命令的效果。

语法格式

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name;
```

参数说明

- savepoint_name。

  回滚截至的保存点。

示例

```
--撤销 my_savepoint 建立之后执行的命令的影响。
START TRANSACTION;
SAVEPOINT my_savepoint;
ROLLBACK TO SAVEPOINT my_savepoint;
```

```
--游标位置不受保存点回滚的影响。
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
SAVEPOINT foo;
FETCH 1 FROM foo;
 ?column?
----------
        1
ROLLBACK TO SAVEPOINT foo;
FETCH 1 FROM foo;
 ?column?
----------
        2
RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

相关链接

SAVEPOINT，RELEASE SAVEPOINT

3.8.176SAVEPOINT

功能描述

SAVEPOINT 用于在当前事务里建立一个新的保存点。

保存点是事务中的一个特殊记号，它允许将那些在它建立后执行的命令全部回滚，把事务的状态恢复到保存点所在的时刻。

注意事项

使用 ROLLBACK TO SAVEPOINT 回滚到一个保存点。使用 RELEASE SAVEPOINT 删除一个保存点，但是保留该保存点建立后执行的命令的效果。

保存点只能在一个事务块里面建立。在一个事务里面可以定义多个保存点。

由于节点故障或者通信故障引起的节点线程或进程退出导致的报错，以及由于 COPY FROM 操作中源数据与目标表的表结构不一致导致的报错，均不能正常回滚到保存点之前，而是整个事务回滚。

SQL 标准要求，使用 savepoint 建立一个同名保存点时，需要自动删除前面那个同名保存点。在 GBase 8s 数据库里，我们将保留旧的保存点，但是在回滚或者释放的时候，只使用最近的那个。释放了新的保存点将导致旧的再次成为 ROLLBACK TO SAVEPOINT 和 RELEASE SAVEPOINT 可以访问的保存点。除此之外，SAVEPOINT 是完全符合 SQL 标准的。

语法格式

```
SAVEPOINT savepoint_name;
```

参数说明

- savepoint_name

  新建保存点的名称。

示例

```
--创建一个新表。
postgres=# CREATE TABLE table1(a int);
--开启事务。
postgres=# START TRANSACTION;
--插入数据。
postgres=# INSERT INTO table1 VALUES (1);
--建立保存点。
postgres=# SAVEPOINT my_savepoint;
--插入数据。
postgres=# INSERT INTO table1 VALUES (2);
--回滚保存点。
postgres=# ROLLBACK TO SAVEPOINT my_savepoint;
--插入数据。
postgres=# INSERT INTO table1 VALUES (3);
--提交事务。
postgres=# COMMIT;
--查询表的内容，会同时看到 1 和 3,不能看到 2，因为 2 被回滚。
postgres=# SELECT * FROM table1;
--删除表。
postgres=# DROP TABLE table1;
--创建一个新表。
postgres=# CREATE TABLE table2(a int);
--开启事务。
postgres=# START TRANSACTION;
--插入数据。
postgres=# INSERT INTO table2 VALUES (3);
--建立保存点。
postgres=# SAVEPOINT my_savepoint;
--插入数据。
postgres=# INSERT INTO table2 VALUES (4);
--回滚保存点。
postgres=# RELEASE SAVEPOINT my_savepoint;
```

```
--提交事务。
postgres=# COMMIT;
--查询表的内容，会同时看到 3 和 4。
postgres=# SELECT * FROM table2;
--删除表。
postgres=# DROP TABLE table2;
```

相关命令

RELEASE SAVEPOINT，ROLLBACK TO SAVEPOINT

3.8.177SELECT

功能描述

SELECT 用于从表或视图中取出数据。

SELECT 语句就像叠加在数据库表上的过滤器，利用 SQL 关键字从数据表中过滤出用户需要的数据。

注意事项

必须对每个在 SELECT 命令中使用的字段有 SELECT 权限。

使用 FOR UPDATE，FOR NO KEY UPDATE，FOR SHARE 或 FOR KEY SHARE 还要求 UPDATE 权限。

语法格式

查询数据

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [/*+ plan_hint */] [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ [ START WITH condition ] CONNECT BY [NOCYCLE] condition [ ORDER SIBLINGS BY
expression ] ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW {window_name AS ( window_definition )} [, ...] ]
[ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
[ ORDER BY {expression [ [ ASC | DESC | USING operator ] |
nlssort_expression_clause ] [ NULLS { FIRST | LAST } ]} [, ...] ]
[ LIMIT { [offset,] count | ALL } ]
[ OFFSET start [ ROW | ROWS ] ]
```

```
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
[ {FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ]
[ NOWAIT | WAIT N ]} [...] ];
```

说明

- condition 和 expression 中可以使用 targetlist 中表达式的别名。
- 只能同一层引用。
- 只能引用 targetlist 中的别名。
- 只能是后面的表达式引用前面的表达式。
- 不能包含 volatile 函数。
- 不能包含 Window function 函数。
- 不支持在 join on 条件中引用别名。
- targetlist 中有多个要应用的别名则报错。

其中子查询 with_query 为：

```
with_query_name [ ( column_name [, ...] ) ]
    AS [ [ NOT ] MATERIALIZED ] ( {select | values | insert | update | delete} )
```

其中指定查询源 from_item 为：

```
{[ ONLY ] table_name [ * ] [ partition_clause ] [ [ AS ] alias [ ( column_alias
[, ...] ) ] ]
[ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
[TIMECAPSULE {TIMESTAMP|CSN} expression]
|( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
|with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] |
column_definition [, ...] ) ]
|function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
|from_item [ NATURAL ] join_type from_item [ ON join_condition | USING
( join_column [, ...] ) ]}
```

其中 group 子句为：

```
( )
| expression
| ( expression [, ...] )
```

```
| ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
| CUBE ( { expression | ( expression [, ...] ) } [, ...] )
| GROUPING SETS ( grouping_element [, ...] )
```

其中指定分区 partition_clause 为：

```
PARTITION { ( partition_name ) | FOR (  partition_value [, ...] ) }
SUBPARTITION { ( subpartition_name ) | FOR (  subpartition_value [, ...] )}
```

说明

- 指定分区只适合普通表。

其中设置排序方式 nlssort_expression_clause 为：

```
NLSSORT ( column_name, ' NLS_SORT = { SCHINESE_PINYIN_M | generic_m_ci } ' )
```

其中，第二个参数可选 generic_m_ci，仅支持纯英文不区分大小写排序。

简化版查询语法，功能相当于 select * from table_name。

```
TABLE { ONLY {(table_name)| table_name} | table_name [ * ]};
```

参数说明

- WITH [RECURSIVE] with_query [, …]

用于声明一个或多个可以在主查询中通过名称引用的子查询，相当于临时表。

如果声明了 RECURSIVE，那么允许 SELECT 子查询通过名称引用它自己。

其中 with_query 的详细格式为：with_query_name [(column_name [, …])] AS({select | values | insert | update | delete})

with_query_name 指定子查询生成的结果集名称，在查询中可使用该名称访问子查询的结果集。

column_name 指定子查询结果集中显示的列名。

每个子查询可以是 SELECT、VALUES、INSERT、UPDATE 或 DELETE 语句。

- plan_hint 子句

以/*+ */的形式在 SELECT 关键字后，用于对 SELECT 对应的语句块生成的计划进行 hint 调优，详细用法请参见章节使用 Plan Hint 进行调优。每条语句中只有第一个/*+ plan_hint */注释块会作为 hint 生效，里面可以写多条 hint。

- ALL

声明返回所有符合条件的行，是默认行为，可以省略该关键字。

DISTINCT [ON (expression [, …])]

从 SELECT 的结果集中删除所有重复的行，使结果集中的每行都是唯一的。

ON (expression [, …]) 只保留那些在给出的表达式上运算出相同结果的行集合中的第一行。

须知

- DISTINCT ON 表达式是使用与 ORDER BY 相同的规则进行解释的。除非使用了 ORDER BY 来保证需要的行首先出现，否则，“第一行” 是不可预测的。

- SELECT 列表

  指定查询表中列名，可以是部分列或者是全部（使用通配符*表示）。

  通过使用子句 AS output_name 可以为输出字段取个别名，这个别名通常用于输出字段的显示。支持关键字 name、value 和 type 作为列别名。

  列名可以用下面几种形式表达：

  手动输入列名，多个列之间用英文逗号（,）分隔。

  可以是 FROM 子句里面计算出来的字段。

- FROM 子句

  为 SELECT 声明一个或者多个源表。

  FROM 子句涉及的元素如下所示。

- table_name

  表名或视图名，名称前可加上模式名，如：schema_name.table_name。

- alias

  给表或复杂的表引用起一个临时的表别名，以便被其余的查询引用。

  别名用于缩写或者在自连接中消除歧义。如果提供了别名，它就会完全隐藏表的实际名称。

- TABLESAMPLE sampling_method (argument [, …]) [REPEATABLE (seed)]

  _table_name_之后的 TABLESAMPLE 子句表示应该用指定的_sampling_method_来检索表中行的子集。

可选的 REPEATABLE 子句指定一个用于产生采样方法中随机数的_种子_数。种子值可以是任何非空常量值。如果查询时表没有被更改，指定相同种子和_argument_值的两个查询将会选择该表相同的采样。但是不同的种子值通常将会产生不同的采样。如果没有给出 REPEATABLE，则会基于一个系统产生的种子为每一个查询选择一个新的随机采样。

- TIMECAPSULE { TIMESTAMP | CSN } expression

查询指定 CSN 点或者指定时间点表的内容。

目前不支持闪回查询的表：系统表、列存表、内存表、DFS 表、全局临时表、本地临时表、UNLOGGED 表、分区表、视图、序列表、Hbkt 表、共享表、继承表、带有 PARTIAL CLUSTER KEY 约束的表。

- TIMECAPSULE TIMESTAMP

关键字，闪回查询的标识，根据 date 日期，闪回查找指定时间点的结果集。date 日期必须是一个过去有效的时间戳。

- TIMECAPSULE CSN

关键字，闪回查询的标识，根据表的 CSN 闪回查询指定 CSN 点的结果集。其中 CSN 可从 gs_txn_snapshot 记录的 snpcsn 号查得。

 说明：

- 闪回查询不能跨越影响表结构或物理存储的语句，否则会报错。即闪回点和当前点之间，如果执行过修改表结构或影响物理存储的语句（DDL、DCL、VACUUM FULL），则闪回失败，报错:ERROR：The table definition of T1 has been changed。

闪回点过旧时，因闪回版本被回收等导致无法获取旧版本会导致闪回失败，报错：Restore point too old。可通过将 version_retention_age 和 vacuum_defer_cleanup_age 设置成同值，配置闪回功能旧版本保留期限，取值范围是 0~1000000，值为 0 表示 VACUUM 不会延迟清除无效的行存记录。

通过时间方式指定闪回点，闪回数据和实际时间点最多偏差为 3 秒。

- column_alias

列别名。

- PARTITION

查询分区表的某个分区的数据。

- partition_name

  分区名。

- partition_value

  指定的分区键值。在创建分区表时，如果指定了多个分区键，可以通过 PARTITION FOR 子句指定的这一组分区键的值，唯一确定一个分区。

- subquery

  FROM 子句中可以出现子查询，创建一个临时表保存子查询的输出。

- with_query_name

  WITH 子句同样可以作为 FROM 子句的源，可以通过 WITH 查询的名称对其进行引用。

- function_name

  函数名称。函数调用也可以出现在 FROM 子句中。

- join_type

  有 5 种类型，如下所示。

  ■ [INNER] JOIN

  一个 JOIN 子句组合两个 FROM 项。可使用圆括弧以决定嵌套的顺序。如果没有圆括弧，JOIN 从左向右嵌套。

  在任何情况下，JOIN 都比逗号分隔的 FROM 项绑定得更紧。

  ■ LEFT [OUTER] JOIN

  返回笛卡尔积中所有符合连接条件的行，再加上左表中通过连接条件没有匹配到右表行的那些行。这样，左边的行将扩展为生成表的全长，方法是在那些右表对应的字段位置填上 NULL。请注意，只在计算匹配的时候，才使用 JOIN 子句的条件，外层的条件是在计算完毕之后施加的。

  ■ RIGHT [OUTER] JOIN

  返回所有内连接的结果行，加上每个不匹配的右边行（左边用 NULL 扩展）。

  这只是一个符号上的方便，因为总是可以把它转换成一个 LEFT OUTER JOIN，只要把左边和右边的输入互换位置即可。

  ■ FULL [OUTER] JOIN

返回所有内连接的结果行，加上每个不匹配的左边行（右边用 NULL 扩展），再加上每个不匹配的右边行（左边用 NULL 扩展）。

- CROSS JOIN

CROSS JOIN 等效于 INNER JOIN ON（TRUE），即没有被条件删除的行。这种连接类型只是符号上的方便，因为它们与简单的 FROM 和 WHERE 的效果相同。

说明

- 必须为 INNER 和 OUTER 连接类型声明一个连接条件，即 NATURAL ON、join_condition、USING (join_column [， …]) 之一。但是它们不能出现在 CROSS JOIN 中。

其中 CROSS JOIN 和 INNER JOIN 生成一个简单的笛卡尔积，和在 FROM 的顶层列出两个项的结果相同。

- ON join_condition

  连接条件，用于限定连接中的哪些行是匹配的。如：ON left_table.a = right_table.a。

- USING(join_column[，…])

  ON left_table.a = right_table.a AND left_table.b = right_table.b … 的简写。要求对应的列必须同名。

- NATURAL

  NATURAL 是具有相同名称的两个表的所有列的 USING 列表的简写。

- from item

  用于连接的查询源对象的名称。

- WHERE 子句

  WHERE 子句构成一个行选择表达式，用来缩小 SELECT 查询的范围。condition 是返回值为布尔型的任意表达式，任何不满足该条件的行都不会被检索。

  WHERE 子句中可以通过指定“(+)”操作符的方法将表的连接关系转换为外连接。但是不建议用户使用这种用法，因为这并不是 SQL 的标准语法，在做平台迁移的时候可能面临语法兼容性的问题。同时，使用“(+)”有很多限制：

  “(+)”只能出现在 where 子句中。

如果 from 子句中已经有指定表连接关系，那么不能再在 where 子句中使用“(+)”。

“(+)”只能作用在表或者视图的列上，不能作用在表达式上。

如果表 A 和表 B 有多个连接条件，那么必须在所有的连接条件中指定“(+)”，否则“(+)”将不会生效，表连接会转化成内连接，并且不给出任何提示信息。

“(+)”作用的连接条件中的表不能跨查询或者子查询。如果“(+)”作用的表，不在当前查询或者子查询的 from 子句中，则会报错。如果“(+)”作用的对端的表不存在，则不报错，同时连接关系会转化为内连接。

“(+)”作用的表达式不能直接通过“OR”连接。

如果“(+)”作用的列是和一个常量的比较关系， 那么这个表达式会成为 join 条件的一部分。

同一个表不能对应多个外表。

“(+)”只能出现“比较表达式”，“NOT 表达式”，“ANY 表达式”，“ALL 表达式”，“IN 表达式”，“NULLIF 表达式”，“IS DISTINCT FROM 表达式”，“IS OF 表达式”。“(+)”不能出现在其他类型表达式中，并且这些表达式中不允许出现通过“AND”和“OR”连接的表达式。

“(+)”只能转化为左外连接或者右外连接，不能转化为全连接，即不能在一个表达式的两个表上同时指定“(+)”。

须知

- 对于 WHERE 子句的 LIKE 操作符，当 LIKE 中要查询特殊字符“%”、“_”、“\”的时候需要使用反斜杠“\”来进行转义。

- START WITH 子句

START WITH 子句通常与 CONNECT BY 子句同时出现，数据进行层次递归遍历查询，START WITH 代表递归的初始条件。若省略该子句，单独使用 CONNECT BY 子句，则表示以表中的所有行作为初始集合。

- CONNECT BY 子句

CONNECT BY 代表递归连接条件，CONNECT BY 条件中可以对列指定 PRIOR 关键字代表以这列为递归键进行递归。当前约束只能对表中的列指定 PRIOR，不支持对表达式、类型转换指定 PRIOR 关键字。若在递归连接条件前加 NOCYCLE，则表示遇到循环记录时停止递归。（注：含 START WITH .. CONNECT BY 子句的 SELECT 语句不支持使用 FOR

SHARE/UPDATE 锁）。

- GROUP BY 子句

  将查询结果按某一列或多列的值分组，值相等的为一组。

- CUBE ({ expression | (expression [, ···]) } [, ···])

  CUBE 是自动对 group by 子句中列出的字段进行分组汇总，结果集将包含维度列中各值的所有可能组合，以及与这些维度值组合相匹配的基础行中的聚合值。它会为每个分组返回一行汇总信息， 用户可以使用 CUBE 来产生交叉表值。比如，在 CUBE 子句中给出三个表达式（n = 3），运算结果为 2n = 23 = 8 组。 以 n 个表达式的值分组的行称为常规行，其余的行称为超级聚集行。

- GROUPING SETS (grouping_element [, ···])

  GROUPING SETS 子句是 GROUP BY 子句的进一步扩展，它可以使用户指定多个 GROUP BY 选项。 这样做可以通过裁剪用户不需要的数据组来提高效率。 当用户指定了所需的数据组时，数据库不需要执行完整 CUBE 或 ROLLUP 生成的聚合集合。

  须知

  - 如果 SELECT 列表的表达式中引用了那些没有分组的字段，则会报错，除非使用了聚集函数，因为对于未分组的字段，可能返回多个数值。

- HAVING 子句

  与 GROUP BY 子句配合用来选择特殊的组。HAVING 子句将组的一些属性与一个常数值比较，只有满足 HAVING 子句中的逻辑表达式的组才会被提取出来。

- WINDOW 子句

  一般形式为 WINDOW window_name AS (window_definition) [, ···]，window_name 是可以被随后的窗口定义所引用的名称，window_definition 可以是以下的形式：

```
[ existing\_window\_name \]

[ PARTITION BY expression \[, ...\] \]

[ ORDER BY expression \[ ASC | DESC | USING operator \] \[ NULLS \{ FIRST | LAST \} \] \[, ...\] \]

[ frame\_clause \]
```

frame_clause 为窗函数定义一个窗口框架 window frame，窗函数（并非所有）依赖于框架，window frame 是当前查询行的一组相关行。frame_clause 可以是以下的形式：

```
[ RANGE | ROWS \] frame\_start

[ RANGE | ROWS \] BETWEEN frame\_start AND frame\_end
```

frame_start 和 frame_end 可以是：

```
UNBOUNDED PRECEDING
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING
```

须知

- 对列存表的查询目前只支持 row_number 窗口函数，不支持 frame_clause。

- UNION 子句

  UNION 计算多个 SELECT 语句返回行集合的并集。

  UNION 子句有如下约束条件：

  除非声明了 ALL 子句，否则缺省的 UNION 结果不包含重复的行。

  同一个 SELECT 语句中的多个 UNION 操作符是从左向右计算的，除非用圆括弧进行了标识。

  FOR UPDATE，FOR NO KEY UPDATE，FOR SHARE 和 FOR KEY SHARE 不能在 UNION 的结果或输入中声明。

  一般表达式：

```
select_statement UNION [ALL] select_statement
```

  select_statement 可以是任何没有 ORDER BY、LIMIT、FOR UPDATE，FOR NO KEY UPDATE，FOR SHARE 或 FOR KEY SHARE 子句的 SELECT 语句。

  如果用圆括弧包围，ORDER BY 和 LIMIT 可以附着在子表达式里。

- INTERSECT 子句

  INTERSECT 计算多个 SELECT 语句返回行集合的交集，不含重复的记录。

  INTERSECT 子句有如下约束条件：

同一个 SELECT 语句中的多个 INTERSECT 操作符是从左向右计算的，除非用圆括弧进行了标识。

当对多个 SELECT 语句的执行结果进行 UNION 和 INTERSECT 操作的时候，会优先处理 INTERSECT。

一般形式：

```
select_statement INTERSECT select_statement
```

select_statement 可以是任何没有 FOR UPDATE，FOR NO KEY UPDATE，FOR SHARE 或 FOR KEY SHARE 子句的 SELECT 语句。

- EXCEPT 子句

EXCEPT 子句有如下的通用形式：

```
select_statement EXCEPT [ ALL ] select_statement
```

select_statement 是任何没有 FOR UPDATE，FOR NO KEY UPDATE，FOR SHARE 或 FOR KEY SHARE 子句的 SELECT 表达式。

EXCEPT 操作符计算存在于左边 SELECT 语句的输出而不存在于右边 SELECT 语句输出的行。

EXCEPT 的结果不包含任何重复的行，除非声明了 ALL 选项。使用 ALL 时，一个在左边表中有 m 个重复而在右边表中有 n 个重复的行将在结果中出现 max(m-n,0) 次。

除非用圆括弧指明顺序，否则同一个 SELECT 语句中的多个 EXCEPT 操作符是从左向右计算的。EXCEPT 和 UNION 的绑定级别相同。

目前，不能给 EXCEPT 的结果或者任何 EXCEPT 的输入声明 FOR UPDATE，FOR NO KEY UPDATE，FOR SHARE 和 FOR KEY SHARE 子句。

- MINUS 子句

与 EXCEPT 子句具有相同的功能和用法。

- ORDER BY 子句

对 SELECT 语句检索得到的数据进行升序或降序排序。对于 ORDER BY 表达式中包含多列的情况：

首先根据最左边的列进行排序，如果这一列的值相同，则根据下一个表达式进行比较，依此类推。

如果对于所有声明的表达式都相同，则按随机顺序返回。

在与 DISTINCT 关键字一起使用的情况下，ORDER BY 中排序的列必须包括在 SELECT 语句所检索的结果集的列中。

在与 GROUP BY 子句一起使用的情况下，ORDER BY 中排序的列必须包括在 SELECT 语句所检索的结果集的列中。

在与 GROUP BY 子句一起使用的情况下，ORDER BY 中排序的列必须包括在 SELECT 语句所检索的结果集的列中。

须知

- 如果要支持中文拼音排序，需要在初始化数据库时指定编码格式为 UTF-8、GB18030 或 GBK。命令如下:

```
initdb  -E UTF8  -D ../data  -locale=zh_CN.UTF-8、initdb -E GB18030 -D ../data -locale=zh_CN.GB18030 或 initdb  -E GBK  -D ../data  -locale=zh_CN.GBK。
```

- LIMIT 子句

LIMIT 子句由两个独立的子句组成:

```
LIMIT { count | ALL }
```

OFFSET start count 声明返回的最大行数，而 start 声明开始返回行之前忽略的行数。如果两个都指定了，会在开始计算 count 个返回行之前先跳过 start 行。

- OFFSET 子句

SQL：2008 开始提出一种不同的语法:

```
OFFSET start { ROW | ROWS }
```

start 声明开始返回行之前忽略的行数。

- FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY

如果不指定 count，默认值为 1，FETCH 子句限定返回查询结果从第一行开始的总行数。

锁定子句

FOR UPDATE 子句将对 SELECT 检索出来的行进行加锁。这样避免它们在当前事务结束前被其他事务修改或者删除，即其他企图 UPDATE、 DELETE、 SELECT FOR UPDATE、SELECT FOR NO KEY UPDATE, SELECT FOR SHARE 或 SELECT FOR KEY SHARE 这些行的事务将被阻塞，直到当前事务结束。任何在一行上的 DELETE 命令也会获得 FOR

UPDATE 锁模式，在非主键列上修改值的 UPDATE 也会获得该锁模式。反过来，SELECT FOR UPDATE 将等待已经在相同行上运行以上这些命令的并发事务，并且接着锁定并且返回被更新的行（或者没有行，因为行可能已被删除）。

FOR NO KEY UPDATE 行为与 FOR UPDATE 类似，不过获得的锁较弱：这种锁将不会阻塞尝试在相同行上获得锁的 SELECT FOR KEY SHARE 命令。任何不获取 FOR UPDATE 锁的 UPDATE 也会获得这种锁模式。

FOR SHARE 的行为类似，只是它在每个检索出来的行上要求一个共享锁，而不是一个排他锁。一个共享锁阻塞其它事务执行 UPDATE、DELETE、SELECT FOR UPDATE 或者 SELECT FOR NO KEY UPDATE，不阻塞 SELECT FOR SHARE 或者 SELECT FOR KEY SHARE。

FOR KEY SHARE 行为与 FOR SHARE 类似，不过锁较弱：SELECT FOR UPDATE 会被阻塞，但是 SELECT FOR NO KEY UPDATE 不会被阻塞。一个键共享锁会阻塞其他事务执行修改键值的 DELETE 或者 UPDATE，但不会阻塞其他 UPDATE，也不会阻止 SELECT FOR NO KEY UPDATE、SELECT FOR SHARE 或者 SELECT FOR KEY SHARE。

为了避免操作等待其他事务提交，可使用 NOWAIT 选项，如果被选择的行不能立即被锁住，将会立即汇报一个错误，而不是等待。

如果在锁定子句中明确指定了表名称，则只有这些指定的表被锁定，其他在 SELECT 中使用的表将不会被锁定。否则，将锁定该命令中所有使用的表。

如果锁定子句应用于一个视图或者子查询，它同样将锁定所有该视图或子查询中使用到的表。

多个锁定子句可以用于为不同的表指定不同的锁定模式。

如果一个表中同时出现（或隐含同时出现）在多个子句中，则按照最强的锁处理。类似的，如果影响一个表的任意子句中出现了 NOWAIT，该表将按照 NOWAIT 处理。

须知

- 对列存表的查询不支持 for update/share。

- NLS_SORT

指定某字段按照特殊方式排序。目前仅支持中文拼音格式排序和不区分大小写排序。如果要支持此排序方式，在创建数据库时需要指定编码格式为“UTF8”、”GB18030”或“GBK”；如果指定为其他编码，例如 SQL_ASCII，则可能报错或者排序无效。

取值范围：

SCHINESE_PINYIN_M，按照中文拼音排序。

generic_m_ci，不区分大小写排序（可选，仅支持纯英文不区分大小写排序）。

- PARTITION 子句

  查询某个分区表中相应分区的数据。

示例

```
--先通过子查询得到一张临时表 temp_t，然后查询表 temp_t 中的所有数据。
postgres=# WITH temp_t(name,isdba) AS (SELECT usename,usesuper FROM pg_user)
SELECT * FROM temp_t;
--查询 tpcds.reason 表的所有 r_reason_sk 记录，且去除重复。
postgres=# SELECT DISTINCT(r_reason_sk) FROM tpcds.reason;
--LIMIT 子句示例：获取表中一条记录。
postgres=# SELECT * FROM tpcds.reason LIMIT 1;
--查询所有记录，且按字母升序排列。
postgres=# SELECT r_reason_desc FROM tpcds.reason ORDER BY r_reason_desc;
--通过表别名，从 pg_user 和 pg_user_status 这两张表中获取数据。
postgres=# SELECT a.usename,b.locktime FROM pg_user a,pg_user_status b WHERE
a.usesysid=b.roloid;
--FULL JOIN 子句示例：将 pg_user 和 pg_user_status 这两张表的数据进行全连接显示，
即数据的合集。
postgres=# SELECT a.usename,b.locktime,a.usesuper FROM pg_user a FULL JOIN
pg_user_status b on a.usesysid=b.roloid;
--GROUP BY 子句示例：根据查询条件过滤，并对结果进行分组。
postgres=# SELECT r_reason_id, AVG(r_reason_sk) FROM tpcds.reason GROUP BY
r_reason_id HAVING AVG(r_reason_sk) > 25;
--GROUP BY CUBE 子句示例：根据查询条件过滤，并对结果进行分组汇总。
postgres=# SELECT r_reason_id,AVG(r_reason_sk) FROM tpcds.reason GROUP BY
CUBE(r_reason_id,r_reason_sk);
--GROUP BY GROUPING SETS 子句示例:根据查询条件过滤，并对结果进行分组汇总。
postgres=# SELECT r_reason_id,AVG(r_reason_sk) FROM tpcds.reason GROUP BY
GROUPING SETS((r_reason_id,r_reason_sk),r_reason_sk);
--UNION 子句示例：将表 tpcds.reason 里 r_reason_desc 字段中的内容以 W 开头和以 N
开头的进行合并。
postgres=# SELECT r_reason_sk, tpcds.reason.r_reason_desc
    FROM tpcds.reason
    WHERE tpcds.reason.r_reason_desc LIKE 'W%'
UNION
SELECT r_reason_sk, tpcds.reason.r_reason_desc
```

```
    FROM tpcds.reason
    WHERE tpcds.reason.r_reason_desc LIKE 'N%';
--NLS_SORT 子句示例：中文拼音排序。
postgres=# SELECT * FROM tpcds.reason ORDER BY NLSSORT( r_reason_desc, 'NLS_SORT
= SCHINESE_PINYIN_M');

--不区分大小写排序（可选，仅支持纯英文不区分大小写排序）：
postgres=# SELECT * FROM tpcds.reason ORDER BY NLSSORT( r_reason_desc, 'NLS_SORT
= generic_m_ci');
--创建分区表 tpcds.reason_p
postgres=# CREATE TABLE tpcds.reason_p
(
  r_reason_sk integer,
  r_reason_id character(16),
  r_reason_desc character(100)
)
PARTITION BY RANGE (r_reason_sk)
(
  partition P_05_BEFORE values less than (05),
  partition P_15 values less than (15),
  partition P_25 values less than (25),
  partition P_35 values less than (35),
  partition P_45_AFTER values less than (MAXVALUE)
)
;
--插入数据。
postgres=# INSERT INTO tpcds.reason_p values(3,'AAAAAAAABAAAAAAA','reason
1'),(10,'AAAAAAAABAAAAAAA','reason 2'),(4,'AAAAAAAABAAAAAAA','reason
3'),(10,'AAAAAAAABAAAAAAA','reason 4'),(10,'AAAAAAAABAAAAAAA','reason
5'),(20,'AAAAAAAACAAAAAAA','reason 6'),(30,'AAAAAAAACAAAAAAA','reason 7');

--PARTITION 子句示例：从 tpcds.reason_p 的表分区 P_05_BEFORE 中获取数据。
postgres=# SELECT * FROM tpcds.reason_p PARTITION (P_05_BEFORE);
 r_reason_sk |   r_reason_id    |   r_reason_desc
-------------+------------------+----------------------------------
           4 | AAAAAAAABAAAAAAA | reason 3
           3 | AAAAAAAABAAAAAAA | reason 1
(2 rows)

--GROUP BY 子句示例：按 r_reason_id 分组统计 tpcds.reason_p 表中的记录数。
postgres=# SELECT COUNT(*),r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id;
 count |   r_reason_id
```

```
-------+-----------------
     2 | AAAAAAAACAAAAAAA
     5 | AAAAAAAABAAAAAAA
(2 rows)

--GROUP BY CUBE 子句示例：根据查询条件过滤，并对查询结果分组汇总。
postgres=# SELECT * FROM tpcds.reason GROUP BY  CUBE
(r_reason_id,r_reason_sk,r_reason_desc);

--GROUP BY GROUPING SETS 子句示例：根据查询条件过滤，并对查询结果分组汇总。
postgres=# SELECT * FROM tpcds.reason GROUP BY  GROUPING SETS
((r_reason_id,r_reason_sk),r_reason_desc);

--HAVING 子句示例：按 r_reason_id 分组统计 tpcds.reason_p 表中的记录，并只显示
r_reason_id 个数大于 2 的信息。
postgres=# SELECT COUNT(*) c,r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id
HAVING c>2;
 c |    r_reason_id
---+-----------------
 5 | AAAAAAAABAAAAAAA
(1 row)

--IN 子句示例：按 r_reason_id 分组统计 tpcds.reason_p 表中的 r_reason_id 个数，并
只显示 r_reason_id 值为 AAAAAAAABAAAAAAA 或 AAAAAAAADAAAAAAA 的个数。
postgres=# SELECT COUNT(*),r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id
HAVING r_reason_id IN('AAAAAAAABAAAAAAA','AAAAAAAADAAAAAAA');
count |    r_reason_id
-------+-----------------
     5 | AAAAAAAABAAAAAAA
(1 row)

--INTERSECT 子句示例：查询 r_reason_id 等于 AAAAAAAABAAAAAAA，并且 r_reason_sk
小于 5 的信息。
postgres=# SELECT * FROM tpcds.reason_p WHERE r_reason_id='AAAAAAAABAAAAAAA'
INTERSECT SELECT * FROM tpcds.reason_p WHERE r_reason_sk<5;
 r_reason_sk |    r_reason_id    |      r_reason_desc
-------------+-------------------+---------------------------------
           4 | AAAAAAAABAAAAAAA | reason 3
           3 | AAAAAAAABAAAAAAA | reason 1
(2 rows)
```

```
--EXCEPT 子句示例：查询 r_reason_id 等于 AAAAAAAABAAAAAAA，并且去除 r_reason_sk
小于 4 的信息。
postgres=# SELECT * FROM tpcds.reason_p WHERE r_reason_id='AAAAAAAABAAAAAAA'
EXCEPT SELECT * FROM tpcds.reason_p WHERE r_reason_sk<4;
r_reason_sk |    r_reason_id    |       r_reason_desc
------------+-------------------+----------------------------------
          10 | AAAAAAAABAAAAAAA | reason 2
          10 | AAAAAAAABAAAAAAA | reason 5
          10 | AAAAAAAABAAAAAAA | reason 4
           4 | AAAAAAAABAAAAAAA | reason 3
(4 rows)

--通过在 where 子句中指定"(+)"来实现左连接。
postgres=# select t1.sr_item_sk ,t2.c_customer_id from store_returns t1,
customer t2 where t1.sr_customer_sk  = t2.c_customer_sk(+)
order by 1 desc limit 1;
 sr_item_sk | c_customer_id
------------+---------------
      18000 |
(1 row)

--通过在 where 子句中指定"(+)"来实现右连接。
postgres=# select t1.sr_item_sk ,t2.c_customer_id from store_returns t1,
customer t2 where t1.sr_customer_sk(+)  = t2.c_customer_sk
order by 1 desc limit 1;
 sr_item_sk |  c_customer_id
------------+-----------------
            | AAAAAAAAJNGEBAAA
(1 row)

--通过在 where 子句中指定"(+)"来实现左连接，并且增加连接条件。
postgres=# select t1.sr_item_sk ,t2.c_customer_id from store_returns t1,
customer t2 where t1.sr_customer_sk  = t2.c_customer_sk(+) and
t2.c_customer_sk(+) < 1 order by 1  limit 1;
 sr_item_sk | c_customer_id
------------+---------------
          1 |
(1 row)

--不支持在 where 子句中指定"(+)"的同时使用内层嵌套 AND/OR 的表达式。
```

```
postgres=# select t1.sr_item_sk ,t2.c_customer_id from store_returns t1,
customer t2 where not(t1.sr_customer_sk  = t2.c_customer_sk(+) and
t2.c_customer_sk(+) < 1);
ERROR:  Operator "(+)" can not be used in nesting expression.
LINE 1: ...tomer_id from store_returns t1, customer t2 where not(t1.sr_...
                                                             ^

--where 子句在不支持表达式宏指定"(+)"会报错。
postgres=# select t1.sr_item_sk ,t2.c_customer_id from store_returns t1,
customer t2 where (t1.sr_customer_sk  = t2.c_customer_sk(+))::bool;
ERROR:  Operator "(+)" can only be used in common expression.

--where 子句在表达式的两边都指定"(+)"会报错。
postgres=# select t1.sr_item_sk ,t2.c_customer_id from store_returns t1,
customer t2 where t1.sr_customer_sk(+)  = t2.c_customer_sk(+);
ERROR:  Operator "(+)" can't be specified on more than one relation in one join
condition
HINT:  "t1", "t2"...are specified Operator "(+)" in one condition.

--删除表。
postgres=# DROP TABLE tpcds.reason_p;

--闪回查询示例
--创建表 tpcds.time_table
postgres=# create table tpcds.time_table(idx integer, snaptime timestamp,
snapcsn bigint, timeDesc character(100));
--向表 tpcds.time_table 中插入记录
postgres=# INSERT INTO tpcds.time_table select 1, now(),int8in(xidout(next_csn)),
'time1' from gs_get_next_xid_csn();
postgres=# INSERT INTO tpcds.time_table select 2, now(),int8in(xidout(next_csn)),
'time2' from gs_get_next_xid_csn();
postgres=# INSERT INTO tpcds.time_table select 3, now(),int8in(xidout(next_csn)),
'time3' from gs_get_next_xid_csn();
postgres=# INSERT INTO tpcds.time_table select 4, now(),int8in(xidout(next_csn)),
'time4' from gs_get_next_xid_csn();
postgres=# select * from tpcds.time_table;

 idx |          snaptime          | snapcsn |
timedesc
-----+----------------------------+---------+-----------------------------
-------------------------------------------------------------------------
   1 | 2021-04-25 17:50:05.360326 |  107322 | time1
   2 | 2021-04-25 17:50:10.886848 |  107324 | time2
```

```
   3 | 2021-04-25 17:50:16.12921  |  107327 | time3
   4 | 2021-04-25 17:50:22.311176 |  107330 | time4
(4 rows)
postgres=# delete tpcds.time_table;
DELETE 4
postgres=# SELECT * FROM tpcds.time_table TIMECAPSULE TIMESTAMP
to_timestamp('2021-04-25 17:50:22.311176','YYYY-MM-DD HH24:MI:SS.FF');
 idx |           snaptime           | snapcsn |
timedesc
-----+------------------------------+---------+-------------------------------
-------------------------------------------------------------------------
   1 | 2021-04-25 17:50:05.360326 |  107322 | time1
   2 | 2021-04-25 17:50:10.886848 |  107324 | time2
   3 | 2021-04-25 17:50:16.12921  |  107327 | time3
(3 rows)
postgres=# SELECT * FROM tpcds.time_table TIMECAPSULE CSN 107330;
 idx |           snaptime           | snapcsn |
timedesc
-----+------------------------------+---------+-------------------------------
-------------------------------------------------------------------------
   1 | 2021-04-25 17:50:05.360326 |  107322 | time1
   2 | 2021-04-25 17:50:10.886848 |  107324 | time2
   3 | 2021-04-25 17:50:16.12921  |  107327 | time3
(3 rows)
```

3.8.178SELECT INTO

功能描述

SELECT INTO 用于根据查询结果创建一个新表，并且将查询到的数据插入到新表中。

数据并不返回给客户端，这一点和普通的 SELECT 不同。新表的字段具有和 SELECT 的输出字段相同的名称和数据类型。

注意事项

CREATE TABLE AS 的作用和 SELECT INTO 类似，且提供了 SELECT INTO 所提供功能的超集。建议使用 CREATE TABLE AS 语法替代 SELECT INTO，因为 SELECT INTO 不能在存储过程中使用。

语法格式

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
```

```
    { * | {expression [ [ AS ] output_name ]} [, ...] }
    INTO [ UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW {window_name AS ( window_definition )} [, ...] ]
    [ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
    [ ORDER BY {expression [ [ ASC | DESC | USING operator ] |
nlssort_expression_clause ] [ NULLS { FIRST | LAST } ]} [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ {FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT | WAIT N ]} [...] ];
```

参数说明

- new_table

  new_table 指定新建表的名称。

- UNLOGGED

  指定表为非日志表。在非日志表中写入的数据不会被写入到预写日志中，这样就会比普通表快很多。但是，它也是不安全的，非日志表在冲突或异常关机后会被自动删截。非日志表中的内容也不会被复制到备用服务器中。在该类表中创建的索引也不会被自动记录。

  使用场景：非日志表不能保证数据的安全性，用户应该在确保数据已经做好备份的前提下使用，例如系统升级时进行数据的备份。

  故障处理：当异常关机等操作导致非日志表上的索引发生数据丢失时，用户应该对发生错误的索引进行重建。

- GLOBAL | LOCAL

  创建临时表时可以在 TEMP 或 TEMPORARY 前指定 GLOBAL 或 LOCAL 关键字。如果指定 GLOBAL 关键字，GBase 8s 会创建全局临时表，否则 GBase 8s 会创建本地临时表。

- TEMPORARY | TEMP

  如果指定 TEMP 或 TEMPORARY 关键字，则创建的表为临时表。临时表分为全局临时表和本地临时表两种类型。创建临时表时如果指定 GLOBAL 关键字则为全局临时表，否则为本地临时表。

  全局临时表的元数据对所有会话可见，会话结束后元数据继续存在。会话与会话之间的

用户数据、索引和统计信息相互隔离，每个会话只能看到和更改自己提交的数据。全局临时表有两种模式：一种是基于会话级别的（ON COMMIT PRESERVE ROWS），当会话结束时自动清空用户数据；一种是基于事务级别的（ON COMMIT PRESERVE ROWS），当执行 commit 或 rollback 时自动清空用户数据。建表时如果没有指定 ON COMMIT 选项，则缺省为会话级别。与本地临时表不同，全局临时表建表时可以指定非 pg_temp_开头的 schema。

由于临时表只在当前会话创建，对于涉及对临时表操作的 DDL 语句，会产生 DDL 失败的报错。因此，建议 DDL 语句中不要对临时表进行操作。TEMP 和 TEMPORARY 等价。

 须知：

本地临时表通过每个会话独立的以 pg_temp 开头的 schema 来保证只对当前会话可见，因此，不建议用户在日常操作中手动删除以 pg_tem、pg_toast_temp 开头的 schema。

如果建表时不指定 TEMPORARY/TEMP 关键字，而指定表的 schema 为当前会话的 pg_temp_开头的 schema，则此表会被创建为临时表。

ALTER/DROP 全局临时表和索引，如果其它会话正在使用它，禁止操作。

全局临时表的 DDL 只会影响当前会话的用户数据和索引。例如 truncate、reindex、analyze 只对当前会话有效。

 说明：SELECT INTO 的其它参数可参考 SELECT 的参数说明。

示例

```
--将 tpcds.reason 表中 r_reason_sk 小于 5 的值加入到新建表中。
postgres=# SELECT * INTO tpcds.reason_t1 FROM tpcds.reason WHERE r_reason_sk <
5;
INSERT 0 6
--删除 tpcds.reason_t1 表。
postgres=# DROP TABLE tpcds.reason_t1;
```

相关命令

SELECT

优化建议

- DATABASE

  不建议在事务中 reindex database。

- SYSTEM

不建议在事务中 reindex 系统表。

3.8.179SET

功能描述

用于修改运行时配置参数。

注意事项

大多数运行时参数都可以用 SET 在运行时设置，但有些则在服务运行过程中或会话开始之后不能修改。

语法格式

设置所处的时区。

```
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT };
```

设置所属的模式。

```
SET [ SESSION | LOCAL ]
    {CURRENT_SCHEMA { TO | = } { schema | DEFAULT }
    | SCHEMA 'schema'};
```

设置客户端编码集。

```
SET [ SESSION | LOCAL ] NAMES encoding_name;
```

设置 XML 的解析方式。

```
SET [ SESSION | LOCAL ] XML OPTION { DOCUMENT | CONTENT };
```

设置其他运行时参数。

```
SET [ LOCAL | SESSION ]
    { {config_parameter { { TO | = } { value | DEFAULT }
                        | FROM CURRENT }}};
```

参数说明

- SESSION

声明的参数只对当前会话起作用。如果 SESSION 和 LOCAL 都没出现，则 SESSION 为缺省值。

如果在事务中执行了此命令，命令的产生影响将在事务回滚之后消失。如果该事务已提交，影响将持续到会话的结束，除非被另外一个 SET 命令重置参数。

- LOCAL

声明的参数只在当前事务中有效。在 COMMIT 或 ROLLBACK 之后，会话级别的设置将再次生效。

不论事务是否提交，此命令的影响只持续到当前事务结束。一个特例是：在一个事务里面，即有 SET 命令，又有 SET LOCAL 命令，且 SET LOCAL 在 SET 后面，则在事务结束之前，SET LOCAL 命令会起作用，但事务提交之后，则是 SET 命令会生效。

- TIME ZONE timezone

  用于指定当前会话的本地时区。

  取值范围：有效的本地时区。该选项对应的运行时参数名称为 TimeZone，DEFAULT 缺省值为 PRC。

- CURRENT_SCHEMA

  schema

  CURRENT_SCHEMA 用于指定当前的模式。

  取值范围：已存在模式名称。如果模式名不存在，会导致 CURRENT_SCHEMA 值为空。

- SCHEMA schema

  同 CURRENT_SCHEMA。此处的 schema 是个字符串。

  例如：set schema 'public'。

- NAMES encoding_name

  用于设置客户端的字符编码。等价于 set client_encoding to encoding_name。

  取值范围：有效的字符编码。该选项对应的运行时参数名称为 client_encoding，默认编码为 UTF8。

- XML OPTION option

  用于设置 XML 的解析方式。

  取值范围：CONTENT（缺省）、DOCUMENT。

- config_parameter

  可设置的运行时参数的名称。可用的运行时参数可以使用 SHOW ALL 命令查看。

  说明：部分通过 SHOW ALL 查看的参数不能通过 SET 设置。如 max_datanodes。

- value

config_parameter 的新值。可以声明为字符串常量、标识符、数字，或者逗号分隔的列表。DEFAULT 用于把这些参数设置为它们的缺省值。

示例

```
--设置模式搜索路径。
postgres=# SET search_path TO tpcds, public;
--把日期时间风格设置为传统的 POSTGRES 风格(日在月前)。
postgres=# SET datestyle TO postgres,dmy;
```

相关命令

RESET，SHOW

3.8.180SET CONSTRAINTS

功能描述

SET CONSTRAINTS 设置当前事务检查行为的约束条件。

IMMEDIATE 约束是在每条语句后面进行检查。DEFERRED 约束一直到事务提交时才检查。每个约束都有自己的模式。

从创建约束条件开始，一个约束总是设定为 DEFERRABLE INITIALLY DEFERRED、DEFERRABLE INITIALLY IMMEDIATE、NOT DEFERRABLE 三个特性之一。第三种总是 IMMEDIATE，并且不会受 SET CONSTRAINTS 影响。前两种以指定的方式启动每个事务，但是其行为可以在事务里用 SET CONSTRAINTS 改变。

带着一个约束名列表的 SET CONSTRAINTS 改变这些约束的模式(都必须是可推迟的)。如果有多个约束匹配某个名称，则所有都会被影响。SET CONSTRAINTS ALL 改变所有可推迟约束的模式。

当 SET CONSTRAINTS 把一个约束从 DEFERRED 改成 IMMEDIATE 的时候，新模式反作用式地起作用：任何将在事务结束准备进行的数据修改都将在 SET CONSTRAINTS 的时候执行检查。如果违反了任何约束，SET CONSTRAINTS 都会失败（并且不会修改约束模式）。因此，SET CONSTRAINTS 可以用于强制在事务中某一点进行约束检查。

检查约束总是不可推迟的。

注意事项

SET CONSTRAINTS 只在当前事务里设置约束的行为。因此，如果用户在事务块之外（START TRANSACTION/COMMIT 对）执行这个命令，它将没有任何作用。

语法格式

```
SET CONSTRAINTS { ALL | { name } [, ...] } { DEFERRED | IMMEDIATE } ;
```

参数说明

- name

  约束名。

  取值范围：已存在的约束名。可以在系统表 pg_constraint 中查到。

- ALL

  所有约束。

- DEFERRED

  约束一直到事务提交时才检查。

- IMMEDIATE

  约束在每条语句后进行检查。

示例

```
--设置所有约束在事务提交时检查。
postgres=# SET CONSTRAINTS ALL DEFERRED;
```

3.8.181SET ROLE

功能描述

设置当前会话的当前用户标识符。

注意事项

当前会话的用户必须是指定的 rolename 角色的成员，但系统管理员可以选择任何角色。

使用这条命令，它可能会增加一个用户的权限，也可能会限制一个用户的权限。如果会话用户的角色有 INHERITS 属性，则它自动拥有它能 SET ROLE 变成的角色的所有权限；在这种情况下，SET ROLE 实际上是删除了所有直接赋予会话用户的权限，以及它的所属角色的权限，只剩下指定角色的权限。另一方面，如果会话用户的角色有 NOINHERITS 属性，SET ROLE 删除直接赋予会话用户的权限，而获取指定角色的权限。

语法格式

设置当前会话的当前用户标识符。

```
SET [ SESSION | LOCAL ] ROLE role_name PASSWORD 'password';
```

重置当前用户标识为当前会话用户标识符。

```
RESET ROLE;
```

参数说明

- SESSION

  声明这个命令只对当前会话起作用，此参数为缺省值。

- LOCAL

  声明该命令只在当前事务中有效。

- role_name

  角色名。

  取值范围：字符串，要符合标识符的命名规范。

- password

  角色的密码。要求符合密码的命名规则。

- RESET ROLE

  用于重置当前用户标识。

示例

```
--创建角色 paul。
postgres=# CREATE ROLE paul IDENTIFIED BY 'xxxxxxxxx';
--设置当前用户为 paul。
postgres=# SET ROLE paul PASSWORD 'xxxxxxxxx';
--查看当前会话用户，当前用户。
postgres=# SELECT SESSION_USER, CURRENT_USER;
--重置当前用户。
postgres=# RESET role;
--删除用户。
postgres=# DROP USER paul;
```

3.8.182 SET SESSION AUTHORIZATION

功能描述

把当前会话里的会话用户标识和当前用户标识都设置为指定的用户。

注意事项

只有在初始会话用户有系统管理员权限的时候，会话用户标识符才能改变。否则，只有在指定了被认证的用户名的情况下，系统才接受该命令。

语法格式

为当前会话设置会话用户标识符和当前用户标识符。

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION role_name PASSWORD 'password';
```

重置会话和当前用户标识符为初始认证的用户名。

```
{SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
    | RESET SESSION AUTHORIZATION};
```

参数说明

- SESSION

  声明这个命令只对当前会话起作用。

- LOCAL

  声明该命令只在当前事务中有效。

- role_name

  用户名。

  取值范围：字符串，要符合标识符的命名规范。

- password

  角色的密码。要求符合密码的命名规则。

- DEFAULT

  重置会话和当前用户标识符为初始认证的用户名。

示例

```
--创建角色 paul。
postgres=# CREATE ROLE paul IDENTIFIED BY 'xxxxxxxxx';
--设置当前用户为 paul。
postgres=# SET SESSION AUTHORIZATION paul password 'xxxxxxxxx';
--查看当前会话用户，当前用户。
postgres=# SELECT SESSION_USER, CURRENT_USER;
--重置当前用户。
postgres=# RESET SESSION AUTHORIZATION;
--删除用户。
postgres=# DROP USER paul;
```

相关命令

SET ROLE

3.8.183SET TRANSACTION

功能描述

为事务设置特性。事务特性包括事务隔离级别、事务访问模式(读/写或者只读)。可以设置当前事务的特性（LOCAL），也可以设置会话的默认事务特性(SESSION)。

注意事项

设置当前事务特性需要在事务中执行（即执行 SET TRANSACTION 之前需要执行 START TRANSACTION 或者 BEGIN），否则设置不生效。

语法格式

设置事务的隔离级别、读写模式。

```
{SET [ LOCAL ] TRANSACTION|SET SESSION CHARACTERISTICS AS TRANSACTION}
  { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED }
  | { READ WRITE | READ ONLY | SERIALIZABLE | REPEATABLE READ }
  } [, ...]
SET TRANSACTION SNAPSHOT snapshot_id;
```

参数说明

- LOCAL

  声明该命令只在当前事务中有效。

- SESSION

  声明这个命令只对当前会话起作用。

  取值范围：字符串，要符合标识符的命名规范。

- ISOLATION_LEVEL

  指定事务隔离级别，该参数决定当一个事务中存在其他并发运行事务时能够看到什么数据。

  说明：在事务中第一个数据修改语句（SELECT、INSERT、DELETE、UPDATE、FETCH、COPY）执行之后，当前事务的隔离级别就不能再次设置。

  取值范围：

  READ COMMITTED：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提

交的数据。这是缺省值。

REPEATABLE READ：可重复读隔离级别，仅仅能看到事务开始之前提交的数据，不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。

SERIALIZABLE：GBase 8s 目前功能上不支持此隔离级别，等价于 REPEATABLE READ。

- READ WRITE | READ ONLY

  指定事务访问模式（读/写或者只读）。

示例

```
--开启一个事务，设置事务的隔离级别为 READ COMMITTED，访问模式为 READ ONLY。
postgres=# START TRANSACTION;
postgres=# SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED READ ONLY;
postgres=# COMMIT;
```

3.8.184SHOW

功能描述

SHOW 将显示当前运行时参数的数值。

注意事项

无。

语法格式

```
SHOW { configuration_parameter | CURRENT_SCHEMA | TIME ZONE | TRANSACTION
ISOLATION LEVEL | SESSION AUTHORIZATION | ALL };
```

参数说明

显示变量的参数请参见 RESET 的参数说明。

示例

```
--显示 timezone 参数值。
postgres=# SHOW timezone;
--显示所有参数。
postgres=# SHOW ALL;
--显示参数名中包含”var”的所有参数
postgres=# SHOW VARIABLES LIKE var;
```

相关命令

SET，RESET

3.8.185SHUTDOWN

功能描述

SHUTDOWN 将关闭当前连接的数据库节点。

注意事项

仅拥有管理员权限的用户可以运行此命令。

语法格式

```
SHUTDOWN
{
|
fast  |
immediate
};
```

参数说明

- ""

  不指定关闭模式，默认为 fast。

- fast

  不等待客户端中断连接，将所有活跃事务回滚并且强制断开客户端，然后关闭数据库节点。

- immediate

  强行关闭，在下次重新启动的时候将导致故障恢复。

示例

```
--关闭当前数据库节点。
postgres=# SHUTDOWN;

--使用 fast 模式关闭当前数据库节点。
postgres=# SHUTDOWN FAST;
```

3.8.186SNAPSHOT

功能描述

针对多用户情况下，对数据进行统一的版本控制。

注意事项

本特性 GUC 参数 db4ai_snapshot_ mode，快照存储模型分为 MSS 和 CSS 两种；GUC 参数 db4ai_snapshot_version_delimiter，用于设定版本分隔符，仅接受设定单字节参数值，默认为“@”;GUC 参数 db4ai_snapshot_version_separator，用于设定子版本分隔符，仅接受设定单字节参数值，默认为“.”。

当快照选用增量存储方式时，各个快照中具有依赖关系。删除快照需要按照依赖顺序进行删除。

snapshot 特性用于团队不同成员间维护数据，涉及管理员和普通用户之间的数据转写。

所以在私有用户、三权分立(enableSeparationOfDuty=ON)等状态下，数据库不支持 snapshot 功能特性。

当需要稳定可用的快照用于 AI 训练等任务时，用户需要将快照发布。

语法格式

1. 创建快照。

可以采用“CREATE SNAPSHOT … AS”以及“CREATE SNAPSHOT … FROM”语句创建数 据表快照。

CREATE SNAPSHOT AS

```
CREATE SNAPSHOT <qualified_name> [@ <version | ident | sconst>]
[COMMENT IS <sconst>}
AS query;
```

CREATE SNAPSHOT FROM

```
CREATE SNAPSHOT <qualified_name> [@ <version | ident | sconst>] FROM @ <version
| ident | sconst>
[COMMENT IS <sconst>}
USING (
{ INSERT [INTO SNAPSHOT] …
| UPDATE [SNAPSHOT] [AS <alias>] SET … [FROM …] [WHERE …]
| DELETE [FROM SNAPSHOT] [AS <alias>] [USING …] [WHERE …] | ALTER [SNAPSHOT]
{ ADD … | DROP … } [, …]
} [; …]
);
```

2. 删除快照。

PURGE SNAPSHOT

```
PURGE SNAPSHOT <qualified_name> @ <version | ident | sconst>;
```

3. 快照采样。

SAMPLE SNAPSHOT

```
SAMPLE SNAPSHOT <qualified_name> @ <version | ident | sconst> [STRATIFY BY
attr_list]
{ AS <label> AT RATIO <num> [COMMENT IS <comment>] } [, ...]
```

4. 快照发布。

PUBLISH SNAPSHOT

```
PUBLISH SNAPSHOT <qualified_name> @ <version | ident | sconst>;
```

5. 快照存档。

ARCHIVE SNAPSHOT

```
ARCHIVE SNAPSHOT <qualified_name> @ <version | ident | sconst>;
```

参数说明

- qualified_ name

  创建 snapshot 的名称。

  取值范围：字符串，需要符合标识符命名规则。

  (可省略)snapshot 的版本号，当省略设置。系统会自动顺延编号。 取值范围：字符串，数字编号配合分隔符。

示例

```
create snapshot s1@1.0 comment is 'first version' as select * from t1;
create snapshot s1@3.0 from @1.0 comment is 'inherits from @1.0' using (INSERT
VALUES(6, 'john'), (7, 'tim'); DELETE WHERE id = 1);
SELECT * FROM s1@1.0;
purge snapshot s1@1.0;
sample snapshot s1@2.0 stratify by name as nick at ratio .5;
publish snapshot s1@2.0;
archive snapshot s1@2.0;
```

相关命令

无

3.8.187START TRANSACTION

功能描述

通过 START TRANSACTION 启动事务。如果声明了隔离级别、读写模式，那么新事务就使用这些特性，类似执行了 SET TRANSACTION。

注意事项

无。

语法格式

```
START TRANSACTION
   [ { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED }
   | { READ WRITE | READ ONLY | SERIALIZABLE | REPEATABLE READ }
   } [, ...] ];
```

参数说明

- WORK | TRANSACTION

  BEGIN 格式中的可选关键字，没有实际作用。

- ISOLATION LEVEL

  指定事务隔离级别，它决定当一个事务中存在其他并发运行事务时它能够看到什么数据。

  说明：

  在事务中第一个数据修改语句（SELECT、 INSERT、DELETE、UPDATE、FETCH、COPY）执行之后，事务隔离级别就不能再次设置。

  取值范围：

  READ COMMITTED：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提交的数据。这是缺省值。

  REPEATABLE READ：可重复读隔离级别，仅仅看到事务开始之前提交的数据，它不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。

  SERIALIZABLE：GBase 8s 目前功能上不支持此隔离级别，等价于 REPEATABLE READ。

- READ WRITE | READ ONLY

  指定事务访问模式（读/写或者只读）。

示例

```
--以默认方式启动事务。
postgres=# START TRANSACTION;
```

```
postgres=# SELECT * FROM tpcds.reason;
postgres=# END;
--以默认方式启动事务。
postgres=# BEGIN;
postgres=# SELECT * FROM tpcds.reason;
postgres=# END;
--以隔离级别为 READ COMMITTED，读/写方式启动事务。
postgres=# START TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
postgres=# SELECT * FROM tpcds.reason;
postgres=# COMMIT;
```

相关命令

COMMIT | END，ROLLBACK，SET TRANSACTION

3.8.188TIMECAPSULE TABLE

功能描述

在人为操作或应用程序错误时，使用 TIMECAPSULE TABLE 语句恢复可将表恢复到一个早期状态。

表可以闪回到过去的时间点，这依赖于系统中保存的旧版本数据。此外 GBase 8s 数据库不能恢复到通过 DDL 操作改变了表结构的早期状态。

注意事项

TIMECAPSULE TABLE 语句的用法主要分为两大类：闪回旧版本数据和从回收站中闪回。

TO TIMECAPSULE 和 TO CSN 能够将表闪回到过去的某个版本。

回收站记录了 DROP 和 TRUNCATE 的对象数据。TO BEFORE DROP 和 TO BEFORE TRUNCATE 就是从回收站中闪回。

不支持闪回表的对象类型：系统表、列存表、内存表、DFS 表、全局临时表、本地临时表、UNLOGGED 表、序列表、hashbucket 表。

闪回点和当前点之间，执行过修改表结构或影响物理存储的语句（DDL、DCL、VACUUM FULL），闪回失败。

执行闪回删除需要用户具有如下权限：用户必须具有垃圾对象所在 schema 的 create 和 usage 权限，并且用户必须是 schema 的所有者或者是垃圾对象的所有者。

执行闪回 TRUNCATE 需要用户具有如下权限：用户必须具有垃圾对象所在 schema 的

create 和 usage 权限，并且用户必须是 schema 的所有者或者是垃圾对象的所有者，另外用户必须具有垃圾对象的 TRUNCATE 权限。

不适用闪回 drop/truncate 功能的场景或表：

回收站关闭场景：enable_recyclebin = off；

系统处于维护态（xc_maintenance_mode = on）或升级场景；

多对象删除场景：DROP/TRUNCATE TABLE 命令同时指定多个对象；

系统表、列存表、内存表、DFS 表、全局临时表、本地临时表、UNLOGGED 表、序列表、hashbucket 表。

语法格式

支持回收站闪回，支持从旧版本闪回。

```
TIMECAPSULE TABLE [ schema.]table_name TO {CSN expr | TIMESTAMP expr | BEFORE { DROP [RENAME TO table_name] | TRUNCATE } }
```

支持闪回查询，查询指定 CSN 点或者指定时间点表的内容

```
IMECAPSULE{ TIMESTAMP | CSN } expression。
```

参数说明

- schema_name

  指定模式包含的表。如果缺省，则为当前模式。

- table_name

  指定表名。

- TO CSN

  指定要返回表的时间点对应的事务提交序列号（CSN）。expr 必须计算一个数字，代表有效的 CSN。

- TO TIMESTAMP

  指定要返回表的时间点对应的时间戳。expr 必须计算一个过去有效的时间戳（使用 TO_TIMESTAMP 函数将字符串转换为时间类型）。表将被闪回到指定时间戳大约 3 秒内的时间点。

📖 说明：闪回点过旧时，因旧版本被回收导致无法获取旧版本，会导致闪回失败并报

错：Restore point too old。

- TO BEFORE DROP

  使用这个子句检索回收站中已删除的表及其子对象。

  你可以指定原始用户指定的表的名称，或对象删除时数据库分配的系统生成名称。

  回收站中系统生成的对象名称是唯一的。因此，如果指定系统生成名称，那么数据库检索指定的对象。使用“select * from gs_recyclebin;”语句查看回收站中的内容。

  如果指定了用户指定的名称，且如果回收站中包含多个该名称的对象，然后数据库检索回收站中最近移动的对象。如果想要检索更早版本的表，你可以这样做：

  指定你想要检索的表的系统生成名称。

  执行 TIMECAPSULE TABLE … TO BEFORE DROP 语句，直到你要检索的表。

  恢复 DROP 表时，只恢复基表名，其他子对象名均保持回收站对象名。用户可根据需要，执行 DDL 命令手工调整子对象名。

  回收站对象不支持 DML、DCL、DDL 等写操作，不支持 DQL 查询操作（后续支持）。

  recyclebin_retention_time 配置参数用于设置回收站对象保留时间，超过该时间的回收站对象将被自动清理。

- RENAME TO

  为从回收站中检索的表指定一个新名称。

- TRUNCATE

  闪回到 TRUNCATE 之前。

示例

```
-- 删除表 tpcds.reason_t2
DROP TABLE IF EXISTS tpcds.reason_t2;
-- 创建表 tpcds.reason_t2
postgres=# CREATE TABLE tpcds.reason_t2
 (
  r_reason_sk     integer,
  r_reason_id     character(16),
  r_reason_desc   character(100)
  );
--向表 tpcds.reason_t2 中插入记录
```

```
postgres=# INSERT INTO tpcds.reason_t2 VALUES (1, 'AA', 'reason1'),(2, 'AB',
'reason2'),(3, 'AC', 'reason3');
INSERT 0 3
--清空 tpcds.reason_t2 表中的数据
postgres=# TRUNCATE TABLE tpcds.reason_t2;
--查询 tpcds.reason_t2 表中的数据
postgres=# select * from tpcds.reason_t2;
 r_reason_sk | r_reason_id | r_reason_desc
-------------+-------------+--------------
(0 rows)
--执行闪回 TRUNCATE
postgres=# TIMECAPSULE TABLE tpcds.reason_t2 to BEFORE TRUNCATE;
postgres=# select * from tpcds.reason_t2;
 r_reason_sk |    r_reason_id    |
r_reason_desc
-------------+-------------------+-----------------------------------------
-------------------------------------------------------
           1 | AA                | reason1
           2 | AB                | reason2
           3 | AC                | reason3
(3 rows)
--删除表 tpcds.reason_t2
postgres=# DROP TABLE tpcds.reason_t2;
--执行闪回 DROP
postgres=# TIMECAPSULE TABLE tpcds.reason_t2 to BEFORE DROP;
TimeCapsule Table
```

3.8.189TRUNCATE

功能描述

清理表数据，TRUNCATE 快速地从表中删除所有行。

它和在目标表上进行无条件的DELETE有同样的效果，但由于TRUNCATE不做表扫描，因而快得多。在大表上操作效果更明显。

注意事项

TRUNCATE TABLE 在功能上与不带 WHERE 子句 DELETE 语句相同：二者均删除表中的全部行。

TRUNCATE TABLE 比 DELETE 速度快且使用系统和事务日志资源少：

DELETE 语句每次删除一行，并在事务日志中为所删除每行记录一项。

TRUNCATE TABLE 通过释放存储表数据所用数据页来删除数据，并且只在事务日志中记录页的释放。

TRUNCATE、DELETE、DROP 三者的差异如下：

TRUNCATE TABLE，删除内容，释放空间，但不删除定义。

DELETE TABLE，删除内容，不删除定义，不释放空间。

DROP TABLE，删除内容和定义，释放空间。

语法格式

清理表数据。

```
TRUNCATE [ TABLE ] [ ONLY ] {table_name [ * ]} [, ... ]
[ CONTINUE IDENTITY ] [ CASCADE | RESTRICT][PURGE]};
```

清理表分区的数据。

```
ALTER TABLE [ IF EXISTS  ] { [ ONLY  ] table_name
                        | table_name *
                        | ONLY ( table_name )  }
TRUNCATE PARTITION { partition_name
                   | FOR (  partition_value  [, ...] )  };
```

参数说明

- ONLY

如果声明 ONLY，只有指定的表会被清空。如果没有声明 ONLY，这个表以及其所有子表（若有）会被清空。

- table_name

目标表的名称（可以有模式修饰）。

取值范围：已存在的表名。

- CONTINUE IDENTITY

不改变序列的值。这是缺省值。

- CASCADE | RESTRICT

CASCADE：级联清空所有由于 CASCADE 而被添加到组中的表。

RESTRICT（缺省值）：完全清空。

PURGE：默认将表数据放入回收站中，PURGE 直接清理。

- partition_name

  目标分区表的分区名。

  取值范围：已存在的分区名。

- partition_value

  指定的分区键值。

  通过 PARTITION FOR 子句指定的这一组值，可以唯一确定一个分区。

  取值范围：需要进行删除数据分区的分区键的取值范围。

   须知：使用 PARTITION FOR 子句时，partition_value 所在的整个分区会被清空。

  UPDATE GLOBAL INDEX

  如果使用该参数，则会更新分区表上的所有全局索引，以确保使用全局索引可以查询出正确的数据；如果不使用该参数，则分区表上的所有全局索引将会失效。

示例

```
--创建表。
postgres=# CREATE TABLE tpcds.reason_t1 AS TABLE tpcds.reason;
--清空表 tpcds.reason_t1。
postgres=# TRUNCATE TABLE tpcds.reason_t1;
--删除表。
postgres=# DROP TABLE tpcds.reason_t1;
--创建分区表。
postgres=# CREATE TABLE tpcds.reason_p
(
  r_reason_sk integer,
  r_reason_id character(16),
  r_reason_desc character(100)
)PARTITION BY RANGE (r_reason_sk)
(
  partition p_05_before values less than (05),
  partition p_15 values less than (15),
  partition p_25 values less than (25),
  partition p_35 values less than (35),
  partition p_45_after values less than (MAXVALUE)
);
```

```
--插入数据。
postgres=# INSERT INTO tpcds.reason_p SELECT * FROM tpcds.reason;
--清空分区 p_05_before。
postgres=# ALTER TABLE tpcds.reason_p TRUNCATE PARTITION p_05_before;
--清空分区 p_15。
postgres=# ALTER TABLE tpcds.reason_p TRUNCATE PARTITION for (13);
--清空分区表。
postgres=# TRUNCATE TABLE tpcds.reason_p;
--删除表。
postgres=# DROP TABLE tpcds.reason_p;
```

3.8.190UPDATE

功能描述

更新表中的数据。UPDATE 修改满足条件的所有行中指定的字段值，WHERE 子句声明条件，SET 子句指定的字段会被修改，没有出现的字段则保持它们的原值。

注意事项

表的所有者、拥有表 UPDATE 权限的用户或拥有 UPDATE ANY TABLE 权限的用户，有权更新表中的数据，系统管理员默认拥有此权限。

对 expression 或 condition 条件里涉及到的任何表要有 SELECT 权限。

对于列存表，暂时不支持 RETURNING 子句。

列存表不支持结果不确定的更新(non-deterministic update)。试图对列存表用多行数据更新一行时会报错。

列存表的更新操作，旧记录空间不会回收，需要执行 VACUUM FULL table_name 进行清理。

列存复制表不支持 UPDATE 操作。

语法格式

```
UPDATE [/*+ plan_hint */] [ ONLY ] table_name [ partition_clause ] [ * ] [ [ AS ]
alias ]
    SET {column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = {( { expression | DEFAULT } [, ...] )
|sub_query }
          }[, ...]
    [ FROM from_list] [ WHERE condition ]
    [ RETURNING {* | {output_expression [ [ AS ] output_name ]} [, ...] }];
```

```
where sub_query can be:
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
where partition_clause can be:
PARTITION { ( partition_name ) | FOR ( partition_value [, ...] ) } |
SUBPARTITION { ( subpartition_name ) | FOR ( subpartition_value [, ...] ) }
```

参数说明

- plan_hint 子句

  以/*+ */的形式在 UPDATE 关键字后，用于对 UPDATE 对应的语句块生成的计划进行 hint 调优，详细用法请参见章节使用 Plan Hint 进行调优。每条语句中只有第一个/*+ plan_hint */注释块会作为 hint 生效，里面可以写多条 hint。

- table_name

  要更新的表名，可以使用模式修饰。

  取值范围：已存在的表名称。

- partition_clause

  指定分区更新操作

```
PARTITION { ( partition_name ) | FOR ( partition_value [, …] ) } |
SUBPARTITION { ( subpartition_name ) | FOR ( subpartition_value [, …] ) }
```

  关键字详见 SELECT 介绍。

  示例详见 CREATE TABLE SUBPARTITION

- alias

  目标表的别名。

  取值范围：字符串，符合标识符命名规范。

- column_name

  要修改的字段名。

  支持使用目标表的别名加字段名来引用这个字段。例如：

- UPDATE foo AS f SET f.col_name = 'namecol';

  取值范围：已存在的字段名。

- expression

  赋给字段的值或表达式。

- DEFAULT

  用对应字段的缺省值填充该字段。

  如果没有缺省值，则为 NULL。

- sub_query

  子查询。

  使用同一数据库里其他表的信息来更新一个表可以使用子查询的方法。其中 SELECT 子句具体介绍请参考 SELECT。

- from_list

  一个表的表达式列表，允许在 WHERE 条件里使用其他表的字段。与在一个 SELECT 语句的 FROM 子句里声明表列表类似。

   须知：目标表绝对不能出现在 from_list 里，除非在使用一个自连接（此时它必须以 from_list 的别名出现）。

- condition

  一个返回 Boolean 类型结果的表达式。只有这个表达式返回 true 的行才会被更新。不建议使用 int 等数值类型作为 condition，因为 int 等数值类型可以隐式转换为 bool 值（非 0 值隐式转换为 true，0 转换为 false），可能导致非预期的结果。

- output_expression

  在所有需要更新的行都被更新之后，UPDATE 命令用于计算返回值的表达式。

  取值范围：使用任何 table 以及 FROM 中列出的表的字段。*表示返回所有字段。

- output_name

  字段的返回名称。

示例

```
--创建表 student1。
```

```
postgres=# CREATE TABLE student1
(
   stuno     int,
   classno   int
);
--插入数据。
postgres=# INSERT INTO student1 VALUES(1,1);
postgres=# INSERT INTO student1 VALUES(2,2);
postgres=# INSERT INTO student1 VALUES(3,3);
--查看数据。
postgres=# SELECT * FROM student1;
--直接更新所有记录的值。
postgres=# UPDATE student1 SET classno = classno*2;
--查看数据。
postgres=# SELECT * FROM student1;
--删除表。
postgres=# DROP TABLE student1;
```

3.8.191VACUUM

功能描述

VACUUM 回收表或 B-Tree 索引中已经删除的行所占据的存储空间。在一般的数据库操作里，那些已经 DELETE 的行并没有从它们所属的表中物理删除；在完成 VACUUM 之前它们仍然存在。因此有必要周期地运行 VACUUM，特别是在经常更新的表上。

注意事项

如果没有参数，VACUUM 处理当前数据库里用户拥有相应权限的每个表。如果参数指定了一个表，VACUUM 只处理指定的那个表。

要对一个表进行 VACUUM 操作，通常用户必须是表的所有者或者被授予了指定表 VACUUM 权限的用户，默认系统管理员有该权限。数据库的所有者允许对数据库中除了共享目录以外的所有表进行 VACUUM 操作（该限制意味着只有系统管理员才能真正对一个数据库进行 VACUUM 操作）。VACUUM 命令会跳过那些用户没有权限的表进行垃圾回收操作。

VACUUM 不能在事务块内执行。

建议生产数据库经常清理（至少每晚一次），以保证不断地删除失效的行。尤其是在增删了大量记录之后，对受影响的表执行 VACUUM ANALYZE 命令是一个很好的习惯。这样将更新系统目录为最近的更改，并且允许查询优化器在规划用户查询时有更好地选择。

不建议日常使用 FULL 选项，但是可以在特殊情况下使用。例如在用户删除了一个表的大部分行之后，希望从物理上缩小该表以减少磁盘空间占用。VACUUM FULL 通常要比单纯的 VACUUM 收缩更多的表尺寸。FULL 选项并不清理索引，所以推荐周期性的运行 REINDEX 命令。实际上，首先删除所有索引，再运行 VACUUM FULL 命令，最后重建索引通常是更快的选择。如果执行此命令后所占用物理空间无变化（未减少），请确认是否有其他活跃事务（删除数据事务开始之前开始的事务，并在 VACUUM FULL 执行前未结束）存在，如果有等其他活跃事务退出进行重试。

VACUUM 会导致 I/O 流量的大幅增加，这可能会影响其他活动会话的性能。因此，有时候会建议使用基于开销的 VACUUM 延迟特性。

如果指定了 VERBOSE 选项，VACUUM 将打印处理过程中的信息，以表明当前正在处理的表。各种有关当前表的统计信息也会打印出来。但是对于列存表执行 VACUUM 操作，指定了 VERBOSE 选项，无信息输出。

当含有带括号的选项列表时，选项可以以任何顺序写入。如果没有括号，则选项必须按语法显示的顺序给出。

VACUUM 和 VACUUM FULL 时，会根据参数 vacuum_defer_cleanup_age 延迟清理行存表记录，即不会立即清理刚刚删除的元组。

VACUUM ANALYZE 先执行一个 VACUUM 操作，然后给每个选定的表执行一个 ANALYZE。对于日常维护脚本而言，这是一个很方便的组合。

简单的 VACUUM（不带 FULL 选项）只是简单地回收空间并且令其可以再次使用。这种形式的命令可以和对表的普通读写并发操作，因为没有请求排他锁。VACUUM FULL 执行更广泛的处理，包括跨块移动行，以便把表压缩到最少的磁盘块数目里。这种形式要慢许多并且在处理的时候需要在表上施加一个排他锁。

VACUUM 列存表内部执行的操作包括三个：迁移 delta 表中的数据到主表、VACUUM 主表的 delta 表、VACUUM 主表的 desc 表。该操作不会回收 delta 表的存储空间，如果要回收 delta 表的冗余存储空间，需要对该列存表执行 VACUUM DELTAMERGE。

同时执行多个 VACUUM FULL 可能出现死锁。

如果没有打开 xc_maintenance_mode 参数，那么 VACUUM FULL 操作将跳过所有系统表。

执行 DELETE 后立即执行 VACUUM FULL 命令不会回收空间。执行 DELETE 后再执行 1000 个非 SELECT 事务，或者等待 1s 后再执行 1 个事务，之后再执行 VACUUM FULL

命令空间才会回收。

语法格式

回收空间并更新统计信息，对关键字顺序无要求。

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | {ANALYZE | ANALYSE }} [,...] ) ]
    [ table_name [ (column_name [, ...] ) ] ] [ PARTITION ( partition_name ) |
SUBPARTITION ( subpartition_name ) ];
```

仅回收空间，不更新统计信息。

```
VACUUM [ FULL [COMPACT] ] [ FREEZE ] [ VERBOSE ] [ table_name ]
[ PARTITION ( partition_name )  | SUBPARTITION ( subpartition_name ) ];
```

回收空间并更新统计信息，且对关键字顺序有要求。

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] { ANALYZE | ANALYSE } [ VERBOSE ]
    [ table_name [ (column_name [, ...] ) ] ] [ PARTITION ( partition_name )|
SUBPARTITION ( subpartition_name ) ];
```

针对 HDFS 表，将 delta 表中的数据转移到主表存储。

```
VACUUM DELTAMERGE [ table_name ];
```

针对 HDFS 表，删除 HDFS 表在 HDFS 存储上的空值分区目录。

```
VACUUM HDFSDIRECTORY [ table_name ];
```

参数说明

- FULL

选择“FULL”清理，这样可以恢复更多的空间，但是需要耗时更多，并且在表上施加了排他锁。

说明：使用 FULL 参数会导致统计信息丢失，如果需要收集统计信息，请在 VACUUM FULL 语句中加上 analyze 关键字。

- FREEZE

指定 FREEZE 相当于执行 VACUUM 时将 vacuum_freeze_min_age 参数设为 0。

- VERBOSE

为每个表打印一份详细的清理工作报告。

- ANALYZE | ANALYSE

更新用于优化器的统计信息，以决定执行查询的最有效方法。

- table_name

  要清理的表的名称（可以有模式修饰）。

  取值范围：要清理的表的名称。缺省时为当前数据库中的所有表。

- column_name

  要分析的具体的字段名称，需要配合 analyze 选项使用。

  取值范围：要分析的具体的字段名称。缺省时为所有字段。

- PARTITION

  COMPACT 和 PARTITION 参数不能同时使用。

- partition_name

  要清理的表的一级分区名称。缺省时为所有一级分区。

- subpartition_name

  要清理的表的二级分区名称。缺省时为所有二级分区

- DELTAMERGE

  只针对列存表，将列存表的 delta table 中的数据转移到主表存储上。对列存表而言，此操作受 enable_delta_store 和参数说明中的 deltarow_threshold 控制。

示例

```
--在表 tpcds.reason 上创建索引。
postgres=# CREATE UNIQUE INDEX ds_reason_index1 ON tpcds.reason(r_reason_sk);
--对带索引的表 tpcds.reason 执行 VACUUM 操作。
postgres=# VACUUM (VERBOSE, ANALYZE) tpcds.reason;
--删除索引。
postgres=# DROP INDEX ds_reason_index1 CASCADE;
postgres=# DROP TABLE tpcds.reason;
```

优化建议

- vacuum

  VACUUM 不能在事务块内执行。

  建议生产数据库经常清理（至少每晚一次），以保证不断地删除失效的行。尤其是在增删了大量记录后，对相关表执行 VACUUM ANALYZE 命令。

不建议日常使用 FULL 选项，但是可以在特殊情况下使用。例如，一个例子就是在用户删除了一个表的大部分行之后，希望从物理上缩小该表以减少磁盘空间占用。

执行 VACUUM FULL 操作时，建议首先删除相关表上的所有索引，再运行 VACUUM FULL 命令，最后重建索引。

3.8.192VALUES

功能描述

根据给定的值表达式计算一个或一组行的值。它通常用于在一个较大的命令内生成一个“常数表”。

注意事项

应当避免使用 VALUES 返回数量非常大的结果行，否则可能会遭遇内存耗尽或者性能低下。出现在 INSERT 中的 VALUES 是一个特殊情况，因为目标字段类型可以从 INSERT 的目标表获知，并不需要通过扫描 VALUES 列表来推测，所以在此情况下可以处理非常大的结果行。

如果指定了多行，那么每一行都必须拥有相同的元素个数。

语法格式

```
VALUES {( expression [, ...] )} [, ...]
    [ ORDER BY { sort_expression [ ASC | DESC | USING operator ] } [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ];
```

参数说明

- expression

  用于计算或插入结果表指定地点的常量或者表达式。

  在一个出现在 INSERT 顶层的 VALUES 列表中，expression 可以被 DEFAULT 替换以表示插入目的字段的缺省值。除此以外，当 VALUES 出现在其他场合的时候是不能使用 DEFAULT 的。

- sort_expression

  一个表示如何排序结果行的表达式或者整数常量。

- ASC

指定按照升序排列。

- DESC

  指定按照降序排列。

- operator

  一个排序操作符。

- count

  返回的最大行数。

- OFFSET start { ROW | ROWS }

  声明返回的最大行数，而 start 声明开始返回行之前忽略的行数。

- FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY

  FETCH 子句限定返回查询结果从第一行开始的总行数，count 的缺省值为 1。

示例

请参见 INSERT 的示例。

3.8.193SHRINK

功能描述

将给定的压缩表进行 chunk 碎片化的整理，整理后有利于页面的读写。

注意事项

- shrink 操作只在主机上执行，备机上不能手动执行。
- shrink 操作的时间与当前 CPU 使用率和表的大小相关。
- shrink 本质属于优化操作，若优化时数据库异常，重新拉起后未优化的部分不再执行。

语法格式

```
SHRINK TABLE table_name [nowait];
SHRINK INDEX index_name [nowait];
```

参数说明

- nowait

  表示任务发起后立即返回，不需要等待整理结果，后台线程会定时唤醒对 shrink 添加的任务进行整理。

示例

以下以 SHRINK TABLE 进行举例，SHRINK INDEX 操作与 SHRINK TABLE 相同。

```
--创建表 row_compression
postgres=# CREATE TABLE row_compression
(
id int
) with (compresstype=2, compress_chunk_size = 512, compress_level = 1);

--插入数据
postgres=# Insert into row_compression select generate_series(1,1000);
--查看数据
postgres=# SELECT * FROM row_compression;
postgres=# SHRINK TABLE row_compression;
--删除表
postgres=# DROP TABLE row_compression;
```

3.9 DELIMITER

3.9.1 功能描述

定义一个结束符，表示遇到该结束符的时候，输入命令会结束。该用法可以用在输入语句较多时，并且语句中存在分号，可以指定一个特殊的符号作为结束符。默认情况下，结束符为‘;’。

3.9.2 注意事项

delimiter 符号目前不是自由设定的，结束符范围有限制，目前包含关键字，标识符，字符串，操作符和分号等，其中常见的用法是”//“；具体使用可以看示例。

设置的结束符的级别是会话级别的，仅在 gsql 客户端支持，仅在 B 模式下可用。

3.9.3 语法格式

定义结束符

```
DELIMITER delimiter_str_name END_OF_INPUT
DELIMITER delimiter_str_name END_OF_INPUT_COLON
```

3.9.4 参数说明

- delim_str_name

  可以被定义的结束符种类。

- END_OF_INPUT/END_OF_INPUT_COLON

  结束状态。

3.9.5 示例

```
-定义标识符
postgres=# delimiter abcd
-定义字符串
postgres=# delimiter “sds;”
-定义操作符
postgres=# delimiter + postgres=# delimiter /
-定义默认值
postgres=# delimtier ;
```

3.9.6 相关链接

无。

4 数据类型

GBase 8s 支持某些数据类型间的隐式转换，具体转化关系详见《GBase 8sV8.8.5 5.0.0_数据库参考手册》系统表 PG_CAST。

4.1 数值类型

GBase 8s 数据库所有可用的整数数值类型。数字操作符和相关的内置函数，详见数字操作函数和操作符。

4.1.1 整数类型

表 4-1 整数类型

名称	描述	存储空间	范围

TINYINT	微整数，别名为 INT1。	1 字节	0 ~ 255
SMALLINT	小范围整数，别名为 INT2。	2 字节	-32,768 ~ +32,767
INTEGER	常用的整数，别名为 INT4。	4 字节	-2,147,483,648 ~ +2,147,483,647
BINARY_INTEGER	常用整数类型 INTEGER 的别名。	4 字节	-2,147,483,648 ~ +2,147,483,647
BIGINT	大范围的整数，别名为 INT8。	8 字节	-9,223,372,036,854,775,808 ~ +9,223,372,036,854,775,807
int16	十六字节的大范围证书，目前不支持用户用于建表等使用。	16 字节	-170,141,183,460,469,231,731,687,303,715,884,105,728 ~ +170,141,183,460,469,231,731,687,303,715,884,105,727

示例

```
--创建具有 TINYINT 类型数据的表。
postgres=# CREATE TABLE int_type_t1 (IT_COL1 TINYINT);
CREATE TABLE
--向创建的表中插入数据。
postgres=# INSERT INTO int_type_t1 VALUES(10);
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM int_type_t1;
it_col1
---------
10
(1 row)
--删除表。
postgres=# DROP TABLE int_type_t1;
DROP TABLE
```

```
--创建具有 TINYINT,INTEGER,BIGINT 类型数据的表。
postgres=# CREATE TABLE int_type_t2 (a TINYINT, b TINYINT, c INTEGER, d BIGINT);
CREATE TABLE
--插入数据。
postgres=# INSERT INTO int_type_t2 VALUES(100, 10, 1000, 10000);
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM int_type_t2;
 a | b | c | d
---+---+---+---
100 | 10 | 1000 | 10000
(1 row)
--删除表。
postgres=# DROP TABLE int_type_t2;
DROP TABLE
```

说明

- TINYINT、SMALLINT、INTEGER、BIGINT 和 INT16 类型存储各种范围的数字，也就是整数。试图存储超出范围以外的数值将会导致错误。
- 常用的类型是 INTEGER，因为它提供了在范围、存储空间、性能之间的最佳平衡。一般只有取值范围确定不超过 SMALLINT 的情况下，才会使用 SMALLINT 类型。而只有在 INTEGER 的范围不够的时候才使用 BIGINT，因为前者相对快得多。

4.1.2 任意精度型

表 4-2 任意精度型

名称	描述	存储空间	范围
NUMERIC[(p[,s])], DECIMAL[(p[,s])]	精度 p 取值范围为[1,1000]，标度 s 取值范围为[0,p]。说明 p 为总位数，s 为小数位数。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大 131,072 位，小数点后最大 16,383 位。

NUMBER[(p[,s])]	NUMERIC 类型的别名。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大 131,072 位，小数点后最大 16,383 位。

示例

```
--创建表。
postgres=# CREATE TABLE decimal_type_t1 (DT_COL1 DECIMAL(10,4));
CREATE TABLE
--插入数据。
postgres=# INSERT INTO decimal_type_t1 VALUES(123456.122331);
INSERT 0 1
--查询表中的数据。
postgres=# SELECT * FROM decimal_type_t1;
dt_col1
-------------
 123456.1223
(1 row)
--删除表。
postgres=# DROP TABLE decimal_type_t1;
DROP TABLE
--创建表。
postgres=# CREATE TABLE numeric_type_t1 (NT_COL1 NUMERIC(10,4));
CREATE TABLE
--插入数据。
postgres=# INSERT INTO numeric_type_t1 VALUES(123456.12354);
INSERT 0 1
--查询表中的数据。
postgres=# SELECT * FROM numeric_type_t1;
nt_col1
-------------
 123456.1235
(1 row)
--删除表。
postgres=# DROP TABLE numeric_type_t1;
DROP TABLE
```

说明

- 与整数类型相比，任意精度类型需要更大的存储空间，其存储效率、运算效率以及压缩比效果都要差一些。在进行数值类型定义时，优先选择整数类型。当且仅当数值超出整数可表示最大范围时，再选用任意精度类型。
- 使用 Numeric/Decimal 进行列定义时，建议指定该列的精度 p 以及标度 s。

4.1.3 序列整型

表 4-3 序列整型

名称	描述	存储空间	范围
SMALLSERIAL	二字节序列整型	2 字节	-32,768 ~ +32,767
SERIAL	四字节序列整型	4 字节	-2,147,483,648 ~ +2,147,483,647
BIGSERIAL	八字节序列整型	8 字节	-9,223,372,036,854,775,808 ~ +9,223,372,036,854,775,807
LARGESERIAL	十六字节序列整形	16 字节	-170,141,183,460,469,231,731,687,303,715,884,105,728 ~ +170,141,183,460,469,231,731,687,303,715,884,105,727

示例

```
--创建表。
postgres=# CREATE TABLE smallserial_type_tab(a SMALLSERIAL);
CREATE TABLE
--插入数据。
postgres=# INSERT INTO smallserial_type_tab VALUES(default);
INSERT 0 1
--再次插入数据。
postgres=# INSERT INTO smallserial_type_tab VALUES(default);
INSERT 0 1
```

```
--查看数据。
postgres=# SELECT * FROM smallserial_type_tab;
a
---
1
2
(2 rows)
--创建表。
postgres=# CREATE TABLE serial_type_tab(b SERIAL);
CREATE TABLE
--插入数据。
postgres=# INSERT INTO serial_type_tab VALUES(default);
INSERT 0 1
--再次插入数据。
postgres=# INSERT INTO serial_type_tab VALUES(default);
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM serial_type_tab;
b
---
1
2
(2 rows)
--创建表。
postgres=# CREATE TABLE bigserial_type_tab(c BIGSERIAL);
CREATE TABLE
--插入数据。
postgres=# INSERT INTO bigserial_type_tab VALUES(default);
INSERT 0 1
--插入数据。
postgres=# INSERT INTO bigserial_type_tab VALUES(default);
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM bigserial_type_tab;
c
---
1
2
(2 rows)
--创建表。
postgres=# CREATE TABLE largeserial_type_tab(c LARGESERIAL);
CREATE TABLE
```

```
--插入数据。
postgres=# INSERT INTO largeserial_type_tab VALUES(default);
INSERT 0 1
--插入数据。
postgres=# INSERT INTO largeserial_type_tab VALUES(default);
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM largeserial_type_tab;
c
---
1
2
(2 rows)
--删除表。
postgres=# DROP TABLE smallserial_type_tab;
DROP TABLE
postgres=# DROP TABLE serial_type_tab;
DROP TABLE
postgres=# DROP TABLE bigserial_type_tab;
DROP TABLE
```

说明

SMALLSERIAL，SERIAL，BIGSERIAL 和 LARGESERIAL 类型不是真正的类型，只是为在表中设置唯一标识做的概念上的便利。因此，创建一个整数字段，并且把它的缺省数值安排为从一个序列发生器读取。应用了一个 NOT NULL 约束以确保 NULL 不会被插入。在大多数情况下用户可能还希望附加一个 UNIQUE 或 PRIMARY KEY 约束避免意外地插入重复的数值，但这个不是自动的。最后，将序列发生器从属于那个字段，这样当该字段或表被删除的时候也一并删除它。目前只支持在创建表时候指定 SERIAL 列，不可以在已有的表中，增加 SERIAL 列。另外临时表也不支持创建 SERIAL 列。因为 SERIAL 不是真正的类型，也不可以将表中存在的列类型转化为 SERIAL。

4.1.4 浮点类型

表 4-4 浮点类型

名称	描述	存储空间	范围
REAL,	单精度浮点数，不精准。	4 字节	-3.402E+38~3.402E+38，6 位十进制数字精

FLOAT4			度。
DOUBLE PRECISION, FLOAT8	双精度浮点数，不精准。	8 字节	-1.79E+308~1.79E+308，15 位十进制数字精度。
FLOAT [(p)]	浮点数，不精准。精度 p 取值范围为[1,53]。 说明：p 为精度，表示总位 数。	4 字节或 8 字节	根据精度 p 不同选择 REAL 或 DOUBLE PRECISION 作为内部表示。如不指定精度，内部用 DOUBLE PRECISION 表示。
BINARY_DOUBLE	DOUBLE PRECISION 的别名。	8 字节	-1.79E+308~1.79E+308，15 位十进制数字精度。
DEC[(p[,s])]	精度 p 取值范围为[1,1000]，标度 s 取值范围为[0,p]。说明 p 为总位数，s 为小数位位数。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大 131,072 位，小数点后最大 16,383 位。
INTEGER[(p [,s])]	精度 p 取值范围为[1,1000]，标度 s 取值范围为[0,p]。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	——

示例

```
--创建表。
postgres=# CREATE TABLE float_type_t2 (FT_COL1 INTEGER, FT_COL2 FLOAT4, FT_COL3 FLOAT8, FT_COL4 FLOAT(3), FT_COL5 BINARY_DOUBLE, FT_COL6 DECIMAL(10,4), FT_COL7 INTEGER(6,3));
CREATE TABLE
--插入数据。
postgres=# INSERT INTO float_type_t2 VALUES(10,10.365456,123456.1234,10.3214, 321.321, 123.123654, 123.123654);
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM float_type_t2 ;
ft_col1 | ft_col2 | ft_col3 | ft_col4 | ft_col5 | ft_col6 | ft_col7
--------+---------+------------+---------+---------+----------+---------
10 | 10.3655 | 123456.1234 | 10.3214 | 321.321 | 123.1237 | 123.124
(1 row)
--删除表。
postgres=# DROP TABLE float_type_t2;
DROP TABLE
```

4.2 货币类型

货币类型存储带有固定小数精度的货币金额。

下表显示的范围假设有两位小数。可以以任意格式输入，包括整型、浮点型或者典型的货币格式（如“$1,000.00”）。根据区域字符集，输出一般是最后一种形式。

表 4-5 货币类型

名称	存储容量	描述	范围
money	8 字节	货币金额	-92233720368547758.08 到 +92233720368547758.07

numeric，int 和 bigint 类型的值可以转化为 money 类型。如果从 real 和 double precision 类型转换到 money 类型，可以先转化为 numeric 类型，再转化为 money 类型，例如：

```
postgres=# SELECT '12.34'::float8::numeric::money;
money
--------
 $12.34
```

```
(1 row)
```

这种用法是不推荐使用的。浮点数不应该用来处理货币类型，因为小数点的位数可能会导致错误。

money 类型的值可以转换为 numeric 类型而不丢失精度。转换为其他类型可能丢失精度，并且必须通过以下两步来完成：

```
postgres=# SELECT '52093.89'::money::numeric::float8;
  float8
----------
 52093.89
(1 row)
```

当一个 money 类型的值除以另一个 money 类型的值时，结果是 double precision（也就是，一个纯数字，而不是 money 类型）；在运算过程中货币单位相互抵消。

4.3 布尔类型

表 4-6 布尔类型

名称	描述	存储空间	取值
BOOLEAN	布尔类型	1 字节。	● true：真 ● false：假 ● null：未知

- “真”值的有效文本值是：

  TRUE、't'、'true'、'y'、'yes'、'1' 、'TRUE'、true、整数范围内 1~2^63-1、整数范围内-1~-2^63。

- “假”值的有效文本值是：

  FALSE、'f'、'false'、'n'、'no'、'0'、0、'FALSE'、false。

  使用 TRUE 和 FALSE 是比较规范的用法（也是 SQL 兼容的用法）。

示例

显示用字母 t 和 f 输出 Boolean 值。

```
--创建表。
postgres=# CREATE TABLE bool_type_t1 (BT_COL1 BOOLEAN, BT_COL2 TEXT);
CREATE TABLE
```

```
--插入数据。
postgres=# INSERT INTO bool_type_t1 VALUES (TRUE, 'sic est');
INSERT 0 1
postgres=# INSERT INTO bool_type_t1 VALUES (FALSE, 'non est');
INSERT 0 1

--查看数据。
postgres=# SELECT * FROM bool_type_t1;
 bt_col1 | bt_col2
---------+---------
 t       | sic est
 f       | non est
(2 rows)
postgres=# SELECT * FROM bool_type_t1 WHERE bt_col1 = 't';
 bt_col1 | bt_col2
---------+---------
 t       | sic est
(1 row)
--删除表。
postgres=# DROP TABLE bool_type_t1;
DROP TABLE
```

4.4 字符类型

GBase 8s 支持的字符类型请参见下表。字符串操作符和相关的内置函数请参见字符处理函数和操作符。

表 4-7 字符类型

名称	描述	存储空间
CHAR(n) CHARACTER(n) NCHAR(n)	定长字符串，不足补空格。n 是指字节长度，如不带精度 n，默认精度为 1。	最大为 10MB。
VARCHAR(n) CHARACTER VARYING(n)	变长字符串。PG 兼容模式下，n 是字符长度。其他兼容模式下，n 指代字节长度。	最大为 10MB。

VARCHAR2(n)	变长字符串。是VARCHAR(n)类型的别名。n 指代字节长度。	最大为 10MB。
NVARCHAR(n)	变长字符串。是NVARCHAR2(n)类型的别名。n 指代字符长度。	最大为 10MB。
NVARCHAR2(n)	变长字符串。n 指代字符长度。	最大为 10MB。
TEXT	变长字符串。	最大为 1GB-1， 但还需要考虑到列描述头信息的大小，以及列所在元组的大小限制（也小于 1GB-1），因此 TEXT 类型最大大小可能小于 1GB-1。
CLOB	文本大对象。是 TEXT 类型的别名。	最大为 4GB-1， 但还需要考虑到列描述头信息的大小，以及列所在元组的大小限制（也小于 4GB-1），因此 CLOB 类型最大大小可能小于 4GB-1。

说明

- 除了每列的大小限制以外，每个元组的总大小也不可超过 1GB-1 字节，主要受列的控制头信息、元组控制头信息以及元组中是否存在 NULL 字段等影响。
- NCHAR 为 bpchar 类型的别名，NCHAR(n)为 VARCHAR(n)类型的别名。
- 超过 1GB 的 clob 只有 dbe_lob 相关高级包支持，系统函数不支持大于 1GBclob。
- 在 GBase 8s 里另外还有两种定长字符类型。在下表里显示特殊字符类型。name 类型只用在内部 系统表中，作为存储标识符，不建议普通用户使用。该类型长度当

前定为 64 字节（63 可用字符加结束符）。类型“char”只用了一个字节的存储空间。他在系统内部主要用于系统表，主要作为简单化的枚举类型使用。

名称	描述	存储空间
name	用于对象名的内部类型。	64 字节。
"char"	单字节内部类型。	1 字节。

示例

```
--创建表。
postgres=# CREATE TABLE char_type_t1 (CT_COL1 CHARACTER(4));
CREATE TABLE
--插入数据。
postgres=# INSERT INTO char_type_t1 VALUES ('ok');
INSERT 0 1
--查询表中的数据。
postgres=# SELECT ct_col1, char_length(ct_col1) FROM char_type_t1;
ct_col1 | char_length
--------+-------------
ok      |   4
(1 row)
--删除表。
postgres=# DROP TABLE char_type_t1;
DROP TABLE

--创建表。
postgres=# CREATE TABLE char_type_t2 (CT_COL1 VARCHAR(5));
--插入数据。
postgres=# INSERT INTO char_type_t2 VALUES ('ok');
INSERT 0 1
postgres=# INSERT INTO char_type_t2 VALUES ('good');
INSERT 0 1

--插入的数据长度超过类型规定的长度报错。
postgres=# INSERT INTO char_type_t2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
CONTEXT:  referenced column: ct_col1
--明确类型的长度，超过数据类型长度后会自动截断。
postgres=# INSERT INTO char_type_t2 VALUES ('too long'::varchar(5));
```

```
INSERT 0 1
--查询数据。
postgres=# SELECT ct_col1, char_length(ct_col1) FROM char_type_t2;
ct_col1 | char_length
---------+-------------
 ok      |           2
 good    |           4
 too l   |           5
(3 rows)
--删除数据。
postgres=# DROP TABLE char_type_t2;
DROP TABLE
```

4.5 二进制类型

GBase 8s 支持的二进制类型，参见下表。

表 4-8 二进制类型

名称	描述	存储空间
BLOB	二进制大对象 说明：列存不支持 BLOB 类型	最大为 1GB-8203 字节（即 1073733621 字节）。
RAW	变长的十六进制类型 说明：列存不支持 RAW 类型	4 字节加上实际的十六进制字符串。最大为 1GB-8203 字节（即 1073733621 字节）。
BYTEA	变长的二进制字符串	4 字节加上实际的二进制字符串。最大为 1GB-8203 字节（即 1073733621 字节）。
BYTEAWITHOUTORDERWITHEQUALCOL	变长的二进制字符串（密态特性新增的类型，如果加密列的加密类型指定为确定性加密，则该列的实际类型为 BYTEAWITHOUTORDERWITHEQUALCOL），元命令打印	4 字节加上实际的二进制字符串。最大为 1GB 减去 53 字节（即 1073741771 字节）。

	加密表将显示原始数据类型	
BYTEAWITHOUTORDERCOL	变长的二进制字符串（密态特性新增的类型，如果加密列的加密类型指定为随机加密，则该列的实际类型为 BYTEAWITHOUT ORDER COL），元命令打印加密表将显示原始数据类型	4 字节加上实际的二进制字符串。最大为 1GB 减去 53 字节（即 1073741771 字节）。
_BYTEAWITHOUTORDERWITHEQUALCOL	变长的二进制字符串，密态特性新增的类型	4 字节加上实际的二进制字符串。最大为 1GB 减去 53 字节（即 1073741771 字节）。
_BYTEAWITHOUTORDERCOL	变长的二进制字符串，密态特性新增的类型	4 字节加上实际的二进制字符串。最大为 1GB 减去 53 字节（即 1073741771 字节）。

说明

- 除了每列的大小限制以外，每个元组的总大小也不可超过 1GB-8203 字节（即 1073733621 字节）。
- 不支持直接使用 BYTEAWITHOUTORDERWITHEQUALCOL 和 BYTEAWITHOUTORDERCOL，_BYTEAWITHOUTORDERWITHEQUALCOL，_BYTEAWITHOUTORDERCOL 类型创建表。

示例

```
--创建表。
postgres=# CREATE TABLE blob_type_t1 (BT_COL1 INTEGER, BT_COL2 BLOB, BT_COL3 RAW,
BT_COL4 BYTEA) ;
CREATE TABLE
--插入数据。
postgres=# INSERT INTO blob_type_t1 VALUES(10,empty_blob(),
HEXTORAW('DEADBEEF'),E'\\xDEADBEEF');
INSERT 0 1
```

```
--查询表中的数据。
postgres=# SELECT * FROM blob_type_t1;
bt_col1 | bt_col2 | bt_col3  |  bt_col4
--------+---------+----------+------------
      10 |         | DEADBEEF | \xdeadbeef
(1 row)
--删除表。
postgres=# DROP TABLE blob_type_t1;
DROP TABLE
```

4.6 日期/时间类型

4.6.1 日期/时间类型

GBase 8s 支持的日期/时间类型请参见下表。该类型的操作符和内置函数请参见 5.8 时间和日期处理函数和操作符。

说明

如果其他的数据库时间格式和 GBase 8s 数据库的时间格式不一致，可通过修改配置参数 DateStyle 值来保持一致。

表 4-9 日期/时间类型

名称	描述	存储空间
DATE	日期和时间。	4 字节（兼容模式 A 下存储空间大小为 8 字节）
TIME [(p)] [WITHOUT TIME ZONE]	只用于一日内时间。 p 表示小数点后的精度，取值范围为 0~6。	8 字节
TIME [(p)] [WITH TIME ZONE]	只用于一日内时间，带时区。 p 表示小数点后的精度，取值范围为 0~6。	12 字节
TIMESTAMP[(p)]	日期和时间。	8 字节

[WITHOUT TIME ZONE]	p 表示小数点后的精度，取值范围为 0~6。	
TIMESTAMP[(p)] [WITH TIME ZONE]	日期和时间，带时区。TIMESTAMP 的别名为 TIMESTAMPTZ。 p 表示小数点后的精度，取值范围为 0~6。	8 字节
SMALLDATETIME	日期和时间，不带时区。 精确到分钟，秒位大于等于 30 秒进一位。	8 字节
INTERVAL DAY (l) TO SECOND (p)	时间间隔，X 天 X 小时 X 分 X 秒。 l：天数的精度，取值范围为 0~6。兼容性考虑，目前未实现具体功能。 p：秒数的精度，取值范围为 0~6。小数末尾的零不显示。	16 字节
INTERVAL [FIELDS] [(p)]	时间间隔。 fields：可以是 YEAR, MONTH，DAY，HOUR, MINUTE，SECOND, DAY TO HOUR，DAY TO MINUTE，DAY TO SECOND，HOUR TO MINUTE，HOUR TO SECOND，MINUTE TO SECOND。 p：秒数的精度，取值范围为 0~6，且 fields 为 SECOND, DAY TO SECOND, HOUR TO SECOND 或 MINUTE TO SECOND 时，参数 p 才有效。小数末尾的零不显示。	12 字节
reltime	相对时间间隔。格式为： X years X mons X days XX:XX:XX 。	4 字节

	采用儒略历计时，规定一年为 365.25 天，一个月为 30 天，计算输入值对应的相对时间间隔，输出采用 POSTGRES 格式。	
abstime	日期和时间。格式为： YYYY-MM-DD hh:mm:ss+timezone 取值范围为 1901-12-13 20:45:53 GMT~2038-01-18 23:59:59 GMT， 精度为秒。	4 字节

示例

```
--创建表。
postgres=# CREATE TABLE date_type_tab(coll date);
CREATE TABLE
--插入数据。
postgres=# INSERT INTO date_type_tab VALUES (date '5-10-2022');
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM date_type_tab;
coll
---------------------
2022-05-10 00:00:00
(1 row)
--删除表。
postgres=# DROP TABLE date_type_tab;
DROP TABLE
--创建表。
postgres=# CREATE TABLE time_type_tab (da time without time zone ,dai time with time zone,dfgh timestamp without time zone,dfga timestamp with time zone, vbg smalldatetime);
CREATE TABLE
--插入数据。
postgres=# INSERT INTO time_type_tab VALUES ('21:21:21','21:21:21 pst','2010-12-12','2013-12-11 pst','2003-04-12 04:05:06');
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM time_type_tab;
```

```
    da     |     dai     |         dfgh          |          dfga          |
vbg
----------+-------------+-----------------------+------------------------+-----
----------------
 21:21:21 | 21:21:21-08 | 2010-12-12 00:00:00 | 2013-12-11 16:00:00+08 |
2003-04-12 04:05:00
(1 row)
--删除表。
postgres=# DROP TABLE time_type_tab;
DROP TABLE

--创建表。
postgres=# CREATE TABLE day_type_tab (a int,b INTERVAL DAY(3) TO SECOND (4));
CREATE TABLE
--插入数据。
postgres=# INSERT INTO day_type_tab VALUES (1, INTERVAL '3' DAY);
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM day_type_tab;
a |   b
---+--------
 1 | 3 days
(1 row)
--删除表。
postgres=# DROP TABLE day_type_tab;
DROP TABLE

--创建表。
postgres=# CREATE TABLE year_type_tab(a int, b interval year (6));
CREATE TABLE
--插入数据。
postgres=# INSERT INTO year_type_tab VALUES(1,interval '2' year);
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM year_type_tab;
 a | b
---+---------
1 | 2 years
(1 row)
--删除表。
postgres=# DROP TABLE year_type_tab;
DROP TABLE
```

4.6.2 日期输入

日期和时间的输入格式包括 ISO-8601 格式、SQL-兼容格式、 传统 POSTGRES 格式或者其它的形式。系统支持按照日、月、年的顺序自定义日期输入。如果把 DateStyle 参数设置为 MDY，则按照“月-日-年”解析；设置为 DMY，则按照“日-月-年”解析；设置为 YMD，则按照 “年-月-日” 解析。

日期的文本输入需要用单引号括起来，语法如下：

```
type [ ( p ) ] 'value'
```

可选精度 p 为整数，表示在秒域中小数部分的位数。下表列出 date 类型的输入方式。

表 4-10 日期输入

例子	描述
1999-01-08	ISO 8601 格式(建议格式)，任何方式下都是 1999 年 1 月 8 号。
January 8, 1999	在任何 datestyle 输入模式下都无歧义。
1/8/1999	有歧义，在 MDY 模式下是一月八号，在 DMY 模式下是八月一号。
1/18/1999	MDY 模式下是一月十八日，其它模式下被拒绝。
01/02/03	MDY 模式下的 2003 年 1 月 2 日。 DMY 模式下的 2003 年 2 月 1 日。 YMD 模式下的 2001 年 2 月 3 日。
1999-Jan-08	任何模式下都是 1 月 8 日。
Jan-08-1999	任何模式下都是 1 月 8 日。
08-Jan-1999	任何模式下都是 1 月 8 日。
99-Jan-08	YMD 模式下是 1 月 8 日，否则错误。

08-Jan-99	一月八日，除了在 YMD 模式下是错误的之外。
Jan-08-99	一月八日，除了在 YMD 模式下是错误的之外。
19990108	ISO 8601；任何模式下都是 1999 年 1 月 8 日。
990108	ISO 8601；任何模式下都是 1999 年 1 月 8 日。
1999.008	年和年里的第几天。
J2451187	儒略日。
January 8, 99 BC	公元前 99 年。

示例

```
--创建表。
postgres=# CREATE TABLE date_type_tab(coll date);
CREATE TABLE
--插入数据。
postgres=# INSERT INTO date_type_tab VALUES (date '5-10-2022');
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM date_type_tab;
coll
---------------------
2022-05-10 00:00:00
(1 row)
--查看日期格式。
postgres=# SHOW datestyle;
DateStyle
-----------
ISO, MDY
(1 row)
--设置日期格式。
postgres=# SET datestyle='YMD';
SET
--插入数据。
postgres=# INSERT INTO date_type_tab VALUES(date '2022-5-11');
INSERT 0 1
--查看数据。
```

```
postgres=# SELECT * FROM date_type_tab;
coll
---------------------
2022-05-10 00:00:00
2022-05-11 00:00:00
(2 rows)
--删除表。
postgres=# DROP TABLE date_type_tab;
DROP TABLE
```

4.6.3 时间输入

时间类型包括 time [(p)] without time zone 和 time [(p)] with time zone。如果只写 time，等效于 time without time zone。如果在 time without time zone 类型的输入中声明了时区，则会忽略这个时区。

时间输入类型的详细信息请参见下表，时区输入类型的详细信息请参见下表。

表 4-11 时间输入

例子	描述
05:06.8	ISO 8601
4:05:06	ISO 8601
4:05	ISO 8601
40506	ISO 8601
4:05 AM	与 04:05 一样，AM 不影响数值
4:05 PM	与 16:05 一样，输入小时数必须<= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601

04:05:06 PST	缩写的时区
2003-04-12 04:05:06 America/New_York	用名称声明的时区

表 时区输入

例子	描述
PST	太平洋标准时间（Pacific Standard Time）
America/New_York	完整时区名称
-8:00	ISO 8601 与 PST 的偏移
-800	ISO 8601 与 PST 的偏移
-8	ISO 8601 与 PST 的偏移

示例

```
postgres=# SELECT time '04:05:06';
time
----------
04:05:06
(1 row)
postgres=# SELECT time '04:05:06 PST';
time
----------
04:05:06
(1 row)
postgres=# SELECT time with time zone '04:05:06 PST';
timetz
-------------
04:05:06-08
(1 row)
```

4.6.4 特殊值

GBase 8s 还支持特殊值，在读取的时候，转换成普通的日期/时间值。

表 4-12 特殊值

输入字符串	适用类型	描述
epoch	date，timestamp	1970-01-01 00:00:00+00（Unix 系统零时）
infinity	timestamp	比任何其他时间戳都晚
-infinity	timestamp	比任何其他时间戳都早
now	date，time， timestamp	当前事务的开始时间
today	date，timestamp	今日午夜
tomorrow	date，timestamp	明日午夜
yesterday	date，timestamp	昨日午夜
allballs	time	00:00:00.00 UTC

4.6.5 时间段输入

reltime 的输入方式可以采用任何合法的时间段文本格式，包括数字形式（含负数和小数）及时间形式，其中时间形式的输入支持 SQL 标准格式、ISO-8601 格式、POSTGRES 格式等。另外，文本输入需要加单引号。

表 4-13 时间段输入详细信息

输入示例	输出结果	描述
60	2 mons	采用数字表示时间段，默认单位是 day，可以是小数或负数。特别的，负数时间段，在语义上，可以理解为“早于多久”。
31.25	1 mons 1 days 06:00:00	
-365	-12 mons -5 days	
1 years 1 mons 8 days	1 years 1 mons 8 days	采用 POSTGRES 格式表示

12:00:00	12:00:00	时间段，可以正负混用，不区分大小写，输出结果为将输入时间段计算并转换得到的简化 POSTGRES 格式时间段。
-13 months -10 hours	-1 years -25 days -04:00:00	
-2 YEARS +5 MONTHS 10 DAYS	-1 years -6 mons -25 days -06:00:00	
P-1.1Y10M	-3 mons -5 days -06:00:00	采用 ISO-8601 格式表示时间段，可以正负混用，不区分大小写，输出结果为将输入时间段计算并转换得到的简化 POSTGRES 格式时间段。
-12H	-12:00:00	

示例

```
--创建表。
postgres=# CREATE TABLE reltime_type_tab(col1 character(30), col2 reltime);
CREATE TABLE
--插入数据。
postgres=# INSERT INTO reltime_type_tab VALUES ('90', '90');
INSERT 0 1
postgres=# INSERT INTO reltime_type_tab VALUES ('-366', '-366');
INSERT 0 1
postgres=# INSERT INTO reltime_type_tab VALUES ('1975.25', '1975.25');
INSERT 0 1
postgres=# INSERT INTO reltime_type_tab VALUES ('-2 YEARS +5 MONTHS 10 DAYS',
'-2 YEARS +5 MONTHS 10 DAYS');
INSERT 0 1
postgres=# INSERT INTO reltime_type_tab VALUES ('30 DAYS 12:00:00', '30 DAYS
12:00:00');
INSERT 0 1
postgres=# INSERT INTO reltime_type_tab VALUES ('P-1.1Y10M', 'P-1.1Y10M');
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM reltime_type_tab;
              col1               |                col2
---------------------------------+-----------------------------------
```

```
 90                                | 3 mons
 -366                              | -1 years -18:00:00
 1975.25                           | 5 years 4 mons 29 days
 -2 YEARS +5 MONTHS 10 DAYS        | -1 years -6 mons -25 days -06:00:00
 30 DAYS 12:00:00                  | 1 mon 12:00:00
 P-1.1Y10M                         | -3 mons -5 days -06:00:00
(6 rows)
--删除表。
postgres=# DROP TABLE reltime_type_tab;
DROP TABLE
```

4.7 几何类型

GBase 8s 支持的几何类型，参见下表。最基本的类型：点，是其它类型的基础。

表 4-14 几何类型

名称	存储空间	说明	表现形式
point	16 字节	平面中的点	(x,y)
lseg	32 字节	（有限）线段	((x1,y1),(x2,y2))
box	32 字节	矩形	((x1,y1),(x2,y2))
path	16+16n 字节	闭合路径（与多边形类似）	((x1,y1),...)
path	16+16n 字节	开放路径	[(x1,y1),...]
polygon	40+16n 字节	多边形（与闭合路径相似）	((x1,y1),...)
circle	24 字节	圆	<(x,y),r> （圆心和半径）

GBase 8s 数据库提供一系列的函数和操作符，用于进行各种几何计算，如拉伸、转换、旋转、计算相交等。详细信息参考几何函数和操作符。

4.7.1 点

点是几何类型的基本二维构造单位。用下面语法描述 point 的数值：

```
( x , y ) x ,
y
```

x 和 y 是用浮点数表示的点的坐标。点输出使用第一种语法。

4.7.2 线段

线段（lseg）是用一对点来代表的。用下面的语法描述 lseg 的数值：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1 , x2 , y2
```

(x1,y1)和(x2,y2)表示线段的端点。线段输出使用第一种语法。

4.7.3 矩形

矩形是用一对对角点来表示的。用下面的语法描述 box 的值：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1 , x2 , y2
```

(x1,y1)和(x2,y2)表示矩形的一对对角点。矩形的输出使用第二种语法。

任何两个对角都可以出现在输入中，但按照那样的顺序，右上角和左下角的值会被重新排序以存储。

4.7.4 路径

路径由一系列连接的点组成。路径可能是开放的，也就是认为列表中第一个点和最后一个点没有连接，也可能是闭合的，这时认为第一个和最后一个点连接起来。

用下面的语法描述 path 的数值：

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1    , ... ,    xn , yn )
x1 , y1 , ... , xn , yn
```

点表示组成路径的线段的端点。方括弧（[]）表明一个开放的路径，圆括弧（()）表明一个闭合的路径。当最外层的括号被省略，如在第三至第五语法，会假定一个封闭的路径。

路径的输出使用第一种或第二种语法输出。

4.7.5 多边形

多边形由一系列点代表（多边形的顶点）。多边形可以认为与闭合路径一样，但是存储方式不一样而且有自己的一套支持函数。

用下面的语法描述 polygon 的数值：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1    , ... ,    xn , yn )
x1 , y1 , ... , xn , yn
```

点表示多边形的端点。多边形输出使用第一种语法。

4.7.6 圆

圆由一个圆心和半径标识。用下面的语法描述 circle 的数值：

```
< ( x , y ) , r > ( ( x , y ) , r ) ( x , y ) , r
x , y , r
```

(x,y)表示圆心，r 表示半径。圆的输出用第一种格式。

4.8 网络地址类型

GBase 8s 数据库提供用于存储 IPv4、IPv6、MAC 地址的数据类型。当存储网络地址时，这些数据类型可以提供输入错误检查和特殊的操作或功能（参见网络地址函数和操作符），因此优于纯文本类型。

表 4-15 网络地址类型

名称	存储空间	描述
cidr	7 或 19 字节	IPv4 或 IPv6 网络
inet	7 或 19 字节	IPv4 或 IPv6 主机和网络
macaddr	6 字节	MAC 地址

在对 inet 或 cidr 数据类型进行排序的时候，IPv4 地址总是排在 IPv6 地址前面，包括那些封装或者是映射在 IPv6 地址里的 IPv4 地址，比如::10.2.3.4 或::ffff:10.4.3.2。

4.8.1 cidr

cidr（无类别域间路由，Classless Inter-Domain Routing）类型，保存一个 IPv4 或 IPv6 网络地址。声明网络格式为 address/y，address 表示 IPv4 或者 IPv6 地址，y 表示子网掩码的二进制位数。如果省略 y，则掩码部分使用已有类别的网络编号系统进行计算，但要求输入的数据已经包括了确定掩码所需的所有字节。

名称	存储空间	描述
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

4.8.2 inet

inet 类型在一个数据区域内保存主机的 IPv4 或 IPv6 地址，以及一个可选子网。主机地址 中网络地址的位数表示子网（“子网掩码”）。如果子网掩码是 32 并且地址是 IPv4， 则这个值不表示任何子网，只表示一台主机。在 IPv6 里，地址长度是 128 位，因此 128 位表示唯一的主机地址。

该类型的输入格式是 address/y，address 表示 IPv4 或者 IPv6 地址，y 是子网掩码的二进制位数。如果省略/y，则子网掩码对 IPv4 是 32，对 IPv6 是 128，所以该值表示只有一台 主机。如果该值表示只有一台主机，/y 将不会显示。

inet 和 cidr 类型之间的基本区别是 inet 接受子网掩码，而 cidr 不接受。

4.8.3 macaddr

macaddr 类型存储 MAC 地址，也就是以太网卡硬件地址（尽管 MAC 地址还用于其它用途）。可以接受下列格式：

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03' '08002b:010203' '08002b-010203' '0800.2b01.0203'
'08002b010203'
```

这些**示例**都表示同一个地址。对于数据位 a 到 f，大小写都行。输出时都是以第一种形式展示。

4.9 位串类型

位串就是一串 1 和 0 的字符串。它们可以用于存储位掩码。

GBase 8s 支持两种位串类型：bit(n)、bit varying(n)，其中 n 为正整数。bit 类型的数据必须准确匹配长度 n，如果存储短或者长的数据都会报错。bit varying 类型的数据是最长为 n 的变长类型，超过 n 的类型会被拒绝。一个没有长度的 bit 等效于 bit(1)，没有长度的 bit varying 表示没有长度限制。

说明

如果用户明确地把一个位串值转换成 bit(n)，则此位串右边的内容将被截断或者在右边补齐零，直到刚好 n 位，而不会抛出任何错误。

如果用户明确地把一个位串数值转换成 bit varying(n)，如果它超过了 n 位，则它的右边将被截断。

```
--创建表。
```

```
postgres=# CREATE TABLE bit_type_t1 (BT_COL1 INTEGER, BT_COL2  BIT(3), BT_COL3
BIT VARYING(5)) ;
CREATE TABLE
--插入数据。
postgres=# INSERT INTO bit_type_t1 VALUES(1, B'101', B'00');
INSERT 0 1
--插入数据的长度不符合类型的标准会报错。
postgres=# INSERT INTO bit_type_t1 VALUES(2, B'10', B'101');
ERROR: bit string length 2 does not  match  type  bit(3)
CONTEXT: referenced column: bt_col2
--将不符合类型长度的数据进行转换。
postgres=# INSERT INTO bit_type_t1 VALUES(2, B'10'::bit(3), B'101');
INSERT 0 1
--查看数据。
postgres=# SELECT * FROM bit_type_t1;
bt_col1 | bt_col2 | bt_col3
--------+---------+---------
1 | 101     | 00
2 | 100     | 101
(2 rows)
--删除表。
postgres=# DROP TABLE bit_type_t1;
DROP TABLE
```

4.10 文本搜索类型

GBase 8s 提供了两种用于全文检索的数据类型。tsvector 类型用于为文本搜索优化的文件格式，tsquery 类型用于文本查询。

4.10.1 tsvector

tsvector 类型表示一个检索单元，通常是一个数据库表中一行的文本字段或者这些字段的组合。to_tsvector 函数通常用于解析和标准化文档字符串。

tsvector 类型的值是一个唯一标准词位的有序列表。把同一个词的变型体都进行标准化得到同样的标准词，在输入的同时，tsvector 会自动排序和消除重复。例如：

```
postgres=# SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
(1 row)
```

从上面的例子可以看出，tsvector 格式中，字符串按照空格进行分词，并按照长短和字母排序。但是如果词条中需要包含空格或标点符号，可以用引号标记。例如：

```
postgres=# SELECT $$the lexeme ' ' contains spaces$$::tsvector;
                tsvector
-------------------------------------
 ' ' 'contains' 'lexeme' 'spaces' 'the'
(1 row)
```

如果在词条中使用引号，可以使用双$$符号作为标记。例如：

```
postgres=# SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
---------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
(1 row)
```

词条位置常量也可以放到词汇中：

```
postgres=# SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11
rat:12'::tsvector;
                                    tsvector
-------------------------------------------------------------------------------
--
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
(1 row)
```

位置常量通常表示文档中源字的位置，可以用于排名。位置常量的范围是 1 到 16383，最大值默认是 16383。相同词的重复位会被忽略掉。

拥有位置的词汇可以使用权标记，这个权可以是 A、B、C 或 D。默认为 D。因此输出中不会显示权为 D：

```
postgres=# SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
(1 row)
```

权可以用来反映文档结构，如：标记标题与主体文字的区别。全文检索排序函数可以为不同的权标记分配不同的优先级。

以下示例为 tsvector 类型标准用法：

```
postgres=# SELECT 'The Fat Rats'::tsvector;
      tsvector
```

```
------------------
 'Fat' 'Rats' 'The'
(1 row)
```

但是对于英文全文检索应用来说，以上单词会判定为非规范化的，所以需要通过 to_tsvector 函数对这些单词进行规范化处理：

```
postgres=# SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
(1 row)
```

4.10.2 tsquery

tsquery 类型表示检索条件，存储用于检索的词汇，并且使用布尔操作符&（AND），|(OR)和!(NOT)来组合他们，括号用来强调操作符的分组。to_tsquery 函数及 plainto_tsquery 函数会将单词转换为 tsquery 类型前进行规范化处 理。

```
postgres=# SELECT 'fat & rat'::tsquery;
    tsquery
--------------
 'fat' & 'rat'
(1 row)
postgres=# SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )
(1 row)
postgres=# SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
----------------------
 'fat' & 'rat' & !'cat'
(1 row)
```

在没有括号的情况下，！（非）结合的最紧密，而&（和）结合的比 |（或）紧密。

tsquery 中的词汇可以用一个或多个权字母来标记，这些权字母限制这些词汇只能与带有匹配权的 tsvector 词汇进行匹配。

```
postgres=# SELECT 'fat:ab & cat'::tsquery;
     tsquery
------------------
 'fat':AB & 'cat'
```

```
(1 row)
```

同样，tsquery 中的词汇可以用*标记来指定前缀匹配。例如，匹配 tsvector 中以 super 开始的任意单词：

```
postgres=# SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
(1 row)
```

需要注意的是，文本搜索分词器会首先处理前缀。例如：

```
postgres=# SELECT to_tsvector('postgraduate') @@ to_tsquery('postgres:*')
AS RESULT;
result
--------
 t
(1 row)
```

例如，将 postgres 处理后得到 postgr，则可以匹配 postgraduate：

```
postgres=# SELECT to_tsquery('postgres:*');
to_tsquery
------------
 'postgr':*
(1 row)
```

例如，将'Fat:ab & Cats'规范化转为 tsquery 类型：

```
postgres=# SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
(1 row)
```

4.11 UUID 类型

UUID 是 ISO/IEF 9834-8:2005 以及相关标准定义的通用唯一标识符。UUID 数据类型可用于存储 RFC 4122。这个标识符是一个由算法产生的 128 位标识符，确保在已知的模块中使用相同算法不可能产生相同的标识符。

UUID 是一个小写十六进制数字的序列，由分字符分为：一组 8 位数字+三组 4 位数字+一组 12 位数字，总共 32 个数字代表 128 位。标准的 UUID 示例如下：

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

GBase 8s 还支持其他的输入方式：大写字母和数字、由花括号包围的标准格式、省略部分或所有连字符、在任意一组四位数字之后加一个连字符。**示例**：

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
```

一般是以标准格式输出。

4.12 JSON/JSONB 类型

JSON(JavaScript Object Notation)数据，可以是单独的一个标量，也可以是一个数组，也可以是一个键值对象。其中数组和对象可以统称容器(container)：

- 标量(scalar)：单一的数字、bool、string、null 都可以叫做标量。
- 数组(array)：[]结构，里面存放的元素可以是任意类型的 JSON，并且不要求数组内所有元素都是同一类型。
- 对象(object)：{}结构，存储 key:value 的键值对，其键只能是用“”包裹起来的字符串，值可以是任意类型的 JSON，对于重复的键，按最后一个键值对为准。

GBase 8s 支持两种存储 JSON 数据的类型：JSON 和 JSONB。其中 JSON 是对输入的字符串的完整拷贝，使用时再去解析，所以它会保留输入的空格、重复键以及顺序等；JSONB 解析输入后保存的二进制，它在解析时会删除语义无关的细节和重复的键，对键值也会进行排序，使用时不用再次解析。

二者接受输入相同的字符串。但差别在于处理效率：JSON 数据类型存储输入文本的精确拷贝，处理函数必须在每个执行上重新解析；而 JSONB 数据类型以分解的二进制格式存储，这使得由于转换机制而在输入过程略慢些，但是在处理过程明显更快，因为不需要重新解析。同时，由于 JSONB 类型存在解析后的格式归一化等操作，同等语义下只会有一种格式。因此，JSONB 能够更好地支持一些额外操作，如按照特定规则进行大小比较等。此外，JSONB 还支持索引。

输入格式

输入必须是一个符合 JSON 数据格式的字符串，此字符串用单引号"声明。

null (null-json)：仅 null，全小写。

```
postgres=# select 'null'::json;
```

```
json
------
 null
(1 row)
postgres=# select 'NULL'::jsonb;
ERROR:  invalid input syntax for type json
LINE 1: select 'NULL'::jsonb;
               ^
DETAIL:  Token "NULL" is invalid.
CONTEXT:  JSON data, line 1: NULL
referenced column: jsonb
```

数字 (num-json)：正负整数、小数、0，支持科学计数法。

```
postgres=# select '1'::json;
json
------
 1
(1 row)
postgres=# select '-1.5'::json;
json
------
 -1.5
(1 row)
postgres=# select '-1.5e-5'::jsonb, '-1.5e+2'::jsonb;
  jsonb   | jsonb
----------+-------
 -.000015 | -150
(1 row)
postgres=# select '001'::json, '+15'::json, 'NaN'::json;
ERROR:  invalid input syntax for type json
LINE 1: select '001'::json, '+15'::json, 'NaN'::json;
               ^
DETAIL:  Token "001" is invalid.
CONTEXT:  JSON data, line 1: 001
referenced column: json
-- 结果表明，JSON 数据类型格式不支持多余的前导 0，正数的+号，以及 NaN 和 infinity。
```

布尔(bool-json)：仅 true、false，全小写。

```
postgres=# select 'true'::json;
json
------
 true
```

```
(1 row)
postgres=# select 'false'::jsonb;
jsonb
-------
 false
(1 row)
```

字符串(str-json)：必须是加双引号的字符串。

```
postgres=# select '"a"'::json;
json
------
 "a"
(1 row)
postgres=# select '"abc"'::jsonb;
jsonb
-------
 "abc"
(1 row)
```

数组(array-json)：使用中括号[]包裹，满足数组书写条件。数组内元素类型可以是任意合法的 JSON，且不要求类型一致。

```
postgres=# select '[1, 2, "foo", null]'::json;
        json
--------------------
 [1, 2, "foo", null]
(1 row)
postgres=# select '[]'::json;
json
------
 []
(1 row)
postgres=# select '[1, 2, "foo", null, [[]], {}]'::jsonb;
             jsonb
-------------------------------
 [1, 2, "foo", null, [[]], {}]
(1 row)
```

对象(object-json)：使用大括号{}包裹，键必须是满足 JSON 字符串规则的字符串，值可以是任意合法的 JSON。

```
postgres=# select '{}'::json;
json
```

```
------
 {}
(1 row)
postgres=# select '{"a": 1, "b": {"a": 2, "b": null}}'::json;
                json
-----------------------------------
 {"a": 1, "b": {"a": 2, "b": null}}
(1 row)
postgres=# select '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::jsonb;
                          jsonb
-------------------------------------------------
 {"foo": [true, "bar"], "tags": {"a": 1, "b": null}}
(1 row)
```

注意

- 'null'::json 和 null::json 是两个不同的概念，类似字符串 str="" 和 str=null 的区别。
- 对于数字，当使用科学计数法的时候，jsonb 类型会将其展开，而 json 会精准拷贝输入。

JSONB 高级特性

注意事项

- 不支持列存。
- 不支持作为分区键。
- 不支持外表、mot。

JSON 和 JSONB 的主要差异在于存储方式上的不同，JSONB 存储的是解析后的二进制，能够体现 JSON 的层次结构，更方便直接访问等，因此 JSONB 会有很多 JSON 所不具有 的高级特性。

- 格式归一化
  - 对于输入的 object-json 字符串，解析成 jsonb 二进制后，会丢弃语义上无关紧要的细节，比如空格。如下示例：

```
postgres=# select '   [1, " a ", {"a"   :1  }] '::jsonb;
        jsonb
---------------------
```

```
 [1, " a ", {"a": 1}]
(1 row)
```

- 对于 object-json，会删除重复的键值，只保留最后一个出现的。如下示例：

```
postgres=# select '{"a" : 1, "a" : 2}'::jsonb;
  jsonb
----------
 {"a": 2}
(1 row)
```

- 对于 object-json，键值会重新进行排序，排序规则：长度长的在后、长度相等则 ascii 码大的在后。如：

```
postgres=# select '{"aa" : 1, "b" : 2, "a" : 3}'::jsonb;
           jsonb
--------------------------
 {"a": 3, "b": 2, "aa": 1}
(1 row)
```

- 大小比较

由于经过了格式归一化，保证了同一种语义下的 JSONB 只会有一种存在形式。因此可以按照制定的规则，进行大小比较。比较类型排序规则为： object-jsonb > array-jsonb > bool-jsonb > num-jsonb > str- jsonb > null-jsonb

同类型则比较内容：

- str-json 类型：依据 text 比较的方法，使用数据库默认排序规则进行比较，返回值正数代表大于，负数代表小于，0 表示相等。
- num-json 类型：数值比较
- bool-json 类型：true > false
- array-jsonb 类型：长度长的 > 长度短的，长度相等则依次比较每个元素。
- object-jsonb 类型：长度长的 > 长度短的，长度相等则依次比较每个键值对，先比较键，在比较值。

注意

object-jsonb 类型内比较时，是对格式整理后的结果进行比较，因此相对于直接输入的

形式，未必会很直观。

- 创建索引、主外键

  - BTREE 索引

JSONB 类型支持创建 btree 索引，支持创建主键、外键。

  - GIN 索引

GIN 索引可以用来有效的搜索出现在大量 JSONB 文档（datums）中的键或者键/值对。提供了两个 GIN 操作符类(jsonb_ops、jsonb_hash_ops)，提供了不同的性能和灵活性取舍。缺省的 GIN 操作符类支持使用@>、<@、?、 ?&和?| 操作符查询，非缺省的 GIN 操作符类 jsonb_path_ops 只支持索引@>、<@操作符。

相关的操作符请参见 JSON/JSONB 函数和操作符。

- 包含存在

查询 JSON 中是否包含某些元素，或者某些元素是否存在于某个 JSON 中，是 JSONB 类型的一个重要能力。

```
-- 简单的标量/原始值只包含相同的值:
postgres=# SELECT '"foo"'::jsonb @> '"foo"'::jsonb;
?column?
----------
 t
(1 row)
-- 左侧数组包含了右侧字符串。
postgres=# SELECT '[1, "aa", 3]'::jsonb ? 'aa';
ERROR:  invalid input syntax for type json
LINE 1: SELECT '[1, "aa", 3]'::jsonb ? 'aa';
               ^
DETAIL:  Token "“aa”" is invalid.
CONTEXT:  JSON data, line 1: [1, "aa"...

-- 左侧数组包含了右侧的数组所有元素，顺序、重复不重要。
postgres=# SELECT '[1, 2, 3]'::jsonb @> '[1, 3, 1]'::jsonb;
?column?
----------
 t
(1 row)
-- 左侧 object-json 包含了右侧 object-json 的所有键值对。
```

```
postgres=# SELECT '{"product": "PostgreSQL", "version": 9.4,
"jsonb":true}'::jsonb @> '{"version":9.4}'::jsonb;
?column?
----------
 t
(1 row)
-- 左侧数组并没有包含右侧的数组所有元素，因为左侧数组的三个元素为 1、2、[1,3]，
右侧的为 1、3。
postgres=# SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;
?column?
----------
 f
(1 row)
-- 比较相似值但不一致，结果值应返回为 false。
postgres=# SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;
?column?
----------
 f
(1 row)
```

相关的操作符请参见 5.13 JSON/JSONB 函数和操作符。

- 函数和操作符

JSON/JSONB 类型相关支持的函数和操作符，参见 5.13 JSON/JSONB 函数和操作符。

4.13 HLL 数据类型

HLL（Hyper Loglog）是一种用于统计数据集中唯一值个数的高效近似算法，具有计算速度快、节省空间的特点，不需要直接存储集合本身，而是存储 HLL 数据结构。每当有新数据合入统计时，只需要把数据经过哈希计算，并插入到 HLL 中，最后根据 HLL 就可以得到统计结果。

HLL 与其他算法的比较，参见下表。

表 4-16 算法比较

比较指标	Sort 算法	Hash 算法	HLL
时间复杂度	O(nlogn)	O(n)	O(n)
空间复杂度	O(n)	O(n)	log(logn)

误差率	0	0	≈0.8%
所需存储空间	原始数据大小	原始数据大小	默认规格下最大为 16KB

由上表可知，HLL 在计算速度和所占存储空间上都占优势。在时间复杂度上，Sort 算法至少需要 O(nlogn)的时间，Hash 算法、HLL 需要 O(n)的时间就可以得出结果；在存储空间上，Sort 算法和 Hash 算法都需要先把原始数据存起来再进行统计，会导致存储空间消耗巨大，而对 HLL 来说，不需要存原始数据，只需要维护 HLL 数据结构，故占用空间有很大的压缩。默认规格下 HLL 数据结构的最大空间约为 16KB。

须知

- 当前默认规格下，可计算最大 distinct 值的数量约为 1.1e+15 个，误差率为 0.8%。需要注意的是，如果计算结果超过当前规格下 distinct 最大值，会导致计算结果误差率变大，或导致计算结果失败并报错。
- 用户在首次使用该特性时，应该对业务的 distinct value 做评估，选取适当的配置参数并做验证，以确保精度符合要求：
  - 当前默认参数下，可以计算的 distinct 值为 1.1e+15，如果计算得到的 distinct 值为 NaN，需要调整 log2m，或者采用其他算法计算 distinct 值。
  - 虽然 hash 算法存在极低的 hash collision 概率，但是建议用户在首次使用时，选取 2-3 个 hash seed 验证，如果得到的 distinct value 相差不大，则可以从该组 seed 中任选一个作为 hash seed。

HLL 中主要的数据结构，请参见下表。

表 4-17 HLL 数据类型

数据类型	功能描述
HLL	HLL 头部为 27 字节长度字段，默认规格下数据段长度 0~16KB，可直接计算得到 distinct 值。

创建 HLL 数据类型时，可以支持 0~4 个参数入参。当入参输入值为-1 时，会采用默认

值设定 HLL 的参数。可以通过\d 或\d+查看 HLL 类型的参数。具体的参数含义与参数规格同函数 hll_empty 一致：

- 第一个参数为 log2m，表示分桶数的对数值，取值范围 10~16；
- 第二个参数为 log2explicit，表示 Explicit 模式的阈值大小，取值范围 0~12；
- 第三个参数为 log2sparse，表示 Sparse 模式的阈值大小，取值范围 0~14；
- 第四个参数为 duplicatecheck，表示是否启用 duplicatecheck，取值范围为 0~1。

说明

创建 HLL 数据类型时，根据入参的行为不同，结果不同：

- 创建 HLL 类型时对应入参不输入或输入-1，采用默认值设定对应的 HLL 参数。
- 输入合法范围的入参，对应 HLL 参数采用输入值。
- 输入不合法范围的入参，创建 HLL 类型报错。

```
-- 创建 hll 类型的表，不指定入参
postgres=# CREATE TABLE t1(id integer, set hll);
CREATE TABLE
postgres=# \d t1
      Table "public.t1"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
 set    | hll     |

-- 创建 hll 类型的表，指定前两个入参，后两个采用默认值
postgres=# CREATE TABLE t2 (id integer, set hll(12,4));
CREATE TABLE
postgres=# \d t2
          Table "public.t2"
 Column |      Type      | Modifiers
--------+----------------+-----------
 id     | integer        |
 set    | hll(12,4,12,0) |

--创建 hll 类型的表，指定第三个入参，其余采用默认值
postgres=# CREATE TABLE t3(id int, set hll(-1,-1,8,-1));
```

```
CREATE TABLE
postgres=# \d t3
          Table "public.t3"
 Column |      Type      | Modifiers
--------+----------------+-----------
 id     | integer        |
 set    | hll(14,10,8,0) |

--创建 hll 类型的表，指定入参不合法报错
postgres=# CREATE TABLE t4(id int, set hll(5,-1));
ERROR:  log2m = 5 is out of range, it should be in range 10 to 16, or set -1 as
default
LINE 1: CREATE TABLE t4(id int, set hll(5,-1));

postgres=# DROP TABLE t1,t2,t3;
DROP TABLE
```

说明

对含有 HLL 类型的表插入 HLL 对象时，HLL 类型的设定参数须同插入对象的设定参数一致，否则报错。

```
-- 创建带有 hll 类型的表
postgres=# CREATE TABLE t1(id integer, set hll(14));
CREATE TABLE
-- 向表中插入 hll 对象,参数一致，成功
postgres=# insert into t1 values (1, hll_empty(14,-1));
INSERT 0 1
-- 向表中插入 hll 对象，参数不一致，失败
postgres=# insert into t1(id, set) values (1, hll_empty(14,5));
ERROR:  log2explicit does not match: source is 5 and dest is 10
CONTEXT:  referenced column: set
postgres=# DROP TABLE t1;
DROP TABLE
```

HLL 的应用场景

场景 1：通过下面的示例说明如何使用 HLL 数据类型：

```
-- 创建带有 hll 类型的表
postgres=# create table helloworld (id integer, set hll);
CREATE TABLE
-- 向表中插入空的 hll
postgres=# insert into helloworld(id, set) values (1, hll_empty());
```

```
INSERT 0 1
-- 把整数经过哈希计算加入到 hll 中
postgres=# update helloworld set set = hll_add(set, hll_hash_integer(12345))
where id = 1;
UPDATE 1
-- 把字符串经过哈希计算加入到 hll 中
postgres=# update helloworld set set = hll_add(set, hll_hash_text('hello world'))
where id = 1;
UPDATE 1
-- 得到 hll 中的 distinct 值
postgres=# select hll_cardinality(set) from helloworld where id = 1;
hll_cardinality
-----------------
              2
(1 row)
-- 删除表
postgres=# drop table helloworld;
DROP TABLE
```

场景 2：网站访客数量统计。通过下面的示例说明 hll 如何统计在一段时间内访问网站的不同用户数量：

```
-- 创建原始数据表，表示某个用户在某个时间访问过网站。
postgres=# create table facts ( date date, user_id integer);
CREATE TABLE
-- 构造数据，表示一天中有哪些用户访问过网站。
postgres=# insert into facts values ('2019-02-20', generate_series(1,100));
INSERT 0 100
postgres=# insert into facts values ('2019-02-21', generate_series(1,200));
INSERT 0 100
postgres=# insert into facts values ('2019-02-22', generate_series(1,300));
INSERT 0 100
postgres=# insert into facts values ('2019-02-23', generate_series(1,400));
INSERT 0 100
postgres=# insert into facts values ('2019-02-24', generate_series(1,500));
INSERT 0 100
postgres=# insert into facts values ('2019-02-25', generate_series(1,600));
INSERT 0 100
postgres=# insert into facts values ('2019-02-26', generate_series(1,700));
INSERT 0 100
postgres=# insert into facts values ('2019-02-27', generate_series(1,800));
INSERT 0 100
```

```
-- 创建表并指定列为 hll。
postgres=# create table daily_uniques ( date date UNIQUE,users hll);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "daily_uniques_date_key" for table
"daily_uniques"
CREATE TABLE
-- 根据日期把数据分组，并把数据插入到 hll 中。
postgres=# insert into daily_uniques(date, users) select date,
hll_add_agg(hll_hash_integer(user_id)) from facts group by 1;
INSERT 0 8
-- 计算每一天访问网站不同用户数量
postgres=# select date, hll_cardinality(users) from daily_uniques order by date;
        date          | hll_cardinality
----------------------+------------------
 2019-02-20 00:00:00 |              100
 2019-02-21 00:00:00 | 200.217913059312
 2019-02-22 00:00:00 |  301.76494508014
 2019-02-23 00:00:00 | 400.862858326446
 2019-02-24 00:00:00 | 502.626933349694
 2019-02-25 00:00:00 | 601.922606454213
 2019-02-26 00:00:00 | 696.602316769498
 2019-02-27 00:00:00 | 798.111731634412
(8 rows)
-- 计算在 2019.02.20 到 2019.02.26 一周中有多少不同用户访问过网站
postgres=# select hll_cardinality(hll_union_agg(users)) from daily_uniques
where date >= '2019-02-20'::date and date <= '2019-02-26'::date;
hll_cardinality
------------------
 696.602316769498
(1 row)
-- 计算昨天访问过网站而今天没访问网站的用户数量。
postgres=# SELECT date, (#hll_union_agg(users) OVER two_days) - #users AS
lost_uniques FROM daily_uniques WINDOW two_days AS (ORDER BY date ASC ROWS 1
PRECEDING);
        date          | lost_uniques
----------------------+--------------
 2019-02-20 00:00:00 |            0
 2019-02-21 00:00:00 |            0
 2019-02-22 00:00:00 |            0
 2019-02-23 00:00:00 |            0
 2019-02-24 00:00:00 |            0
 2019-02-25 00:00:00 |            0
 2019-02-26 00:00:00 |            0
```

```
 2019-02-27 00:00:00 |            0
(8 rows)
-- 删除表
postgres=# drop table facts;
DROP TABLE
postgres=# drop table daily_uniques;
DROP TABLE
```

场景 3：当用户给 hll 类型的字段插入数据的时候，必须保证插入的数据满足 hll 数据结构要求，如果解析后不满足就会报错。**示例**：插入数据'E\\1234'时，该数据不满足 hll 数据结构，不能解析成功因此失败报错。

```
postgres=# create table test(id integer, set hll);
CREATE TABLE
postgres=# insert into test values(1, 'E\\1234');
ERROR:  not a hll type, size=6 is not enough
LINE 1: insert into test values(1, 'E\\1234');
                                   ^

CONTEXT:  referenced column: set
postgres=# drop table test;
DROP TABLE
```

4.14 范围类型

范围类型是表达某种元素类型值的范围的数据类型，这种元素类型称为称为范围的 subtype。例如，timestamp 的范围，可用来表达一个会议室被保留的时间范围。在这种情况下，范围的数据类型是 tsrange（timestamp range 的简写），范围 subtype 的数据类型为 timestamp。subtype 必须具有某种总体的顺序，保证元素值是在范围值之内。范围值界限是清楚的。

范围类型可以表达一种单一范围值中的多个元素值，并且可以清晰地表达诸如范围重叠等概念。例如，用于时间安排的时间和日期范围、价格范围、仪器的量程等方面，都可使用范围类型。

4.14.1 内建范围类型

GBase 8s 提供如下内建范围类型：

- int4range — integer 的范围
- int8range — bigint 的范围
- numrange — numeric 的范围

- tsrange — 不带时区的 timestamp 的范围
- tstzrange — 带时区的 timestamp 的范围
- daterange — date 的范围

此外，用户也可以定义自己的范围类型，详见 CREATE TYPE。

示例

```
postgres=# CREATE TABLE reservation (room int, during tsrange);
CREATE TABLE
postgres=# INSERT INTO reservation VALUES (1108, '[2010-01-01 14:30, 2010-01-01
15:30)');
INSERT 0 1
-- 包含
postgres=# SELECT int4range(10, 20) @> 3;
f
-- 重叠
postgres=# SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);
t
-- 抽取上界
postgres=# SELECT upper(int8range(15, 25));
25
-- 计算交集
postgres=# SELECT int4range(10, 20) * int4range(15, 25);
[15,20)
-- 范围为空吗？
postgres=# SELECT isempty(numrange(1, 5));
f
```

范围类型上的操作符和函数的完整列表，参见 5.17 范围函数和操作符。

4.14.2 包含和排除边界

每一个非空范围都有两个界限：下界和上界。上下界之间的所有值都被包括在范围内。包含界限意味着边界点本身也被包括在范围内，而排除边界则意味边界点不被包括在范围内。

在范围的文本形式中，包含下界用"["表示，排除下界用"("表示。同理，包含上界用"]"表示，排除上界用")"表示。详见范围输入/输出。

用户可以通过函数 lower_inc 和 upper_inc，分别测试范围值的上下界。

4.14.3 无限（无界）范围

当忽略范围的下界时，所有小于上界的值都被包括在范围中，例如(,3]。同理，如果忽略范围的上界，那么所有比上界大的值都包括在范围中。如果上下界都被忽略，该元素类型的所有值都包含在该范围中。在 GBase 8s 数据库中，缺省的包含界限自动转换为排除界限，例如：[,]转换为(,)。

具有无限（infinity）概念的元素类型，可以使用 infinity 作为显式边界值。例如，在时间戳范围，[today,infinity)不包括特殊的 timestamp 值 infinity。如果使用[today,infinity]，则包含特殊值 infinity。尽管这二者的区别，好比[today,)和[today,]，范围值差别不大。

用户可以使用函数 lower_inf 和 upper_inf，分别测试范围的无限上下界。

4.14.4 范围输入/输出

范围值的输入必须遵循下列模式之一：

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

圆括号或方括号指示上下界是否为被排除或被包含的。empty 表示空范围。

lower-bound 可以是作为 subtype 的合法输入的一个字符串，或者是空表示没有下界。同样，upper-bound 可以是作为 subtype 的合法输入的一个字符串，或者是空表示没有上界。

- 每个界限值可以使用"（双引号）字符引用。如果界限值包含圆括号、方括号、逗号、双引号或反斜线时，必须使用双引号。否则那些字符认定为范围语法，而非界限值内容。
- 如需把双引号或反斜线放在被引用的界限值中，则需在前面添加使用一个反斜线。
- 在双引号引用的界限值中的一对双引号，表示为双引号字符，这与 SQL 字符串中的单引号规则类似。
- 用户可以避免引用，并且使用反斜线转义来保护所有数据字符，否则它们会被当做返回语法的一部分。
- 如需写一个空字符串的界限值，则可以写成""。什么都不写，默认表示一个无限界限。

- 范围值前后允许有空格，但是圆括号或方括号之间的任何空格，认定为上下界值的一部分。

示例

```
-- 包括 3，不包括 7，并且包括 3 和 7 之间的所有点
postgres=# SELECT  '[3,7)'::int4range;
[3,7)
-- 既不包括 3 也不包括 7，但是包括之间的所有点
postgres=# SELECT '(3,7)'::int4range;
[4,7)
-- 只包括单独一个点 4
postgres=# SELECT  '[4,4]'::int4range;
[4,5)
-- 不包括点（并且将被标准化为 '空'）
postgres=# SELECT '[4,4)'::int4range;
empty
```

4.14.5 构造范围

每一种范围类型都有一个与其同名的构造器函数。使用构造器函数常常比写一个范围文字常数更方便，因为它避免了对界限值的额外引用。构造器函数接受两个或三个参数。两个参数的形式以标准的形式构造一个范围（下界是包含的，上界是排除的）， 而三个参数的形式按照第三个参数指定的界限形式构造一个范围。第三个参数必须是下列字符串之一：“()”、 “(]”、 “[)”或者 “[]”。 例如：

```
-- 完整形式是：下界、上界以及指示界限包含性/排除性的文本参数。
postgres=# SELECT numrange(1.0, 14.0, '(]');
(1.0,14.0]
-- 如果第三个参数被忽略，则假定为 '[)'。
postgres=# SELECT numrange(1.0, 14.0);
[1.0,14.0)
-- 尽管这里指定了 '(]'，显示时该值将被转换成标准形式，因为 int8range 是一种离散范围类型（见下文）。
postgres=# SELECT int8range(1, 14, '(]');
[2,15)
-- 为一个界限使用 NULL 导致范围在那一边是无界的。
postgres=# SELECT numrange(NULL, 2.2);
(,2.2)
```

4.14.6 离散范围类型

一种范围的元素类型具有一个良定义的“步长”，例如 integer 或 date。在这些类型中，如果两个元素之间没有合法值，它们可以被说成是相邻。这与连续范围相反，连续范围中总是（或者几乎总是）可以在两个给定值之间标识其他元素值。例如， numeric 类型之上的一个范围就是连续的， timestamp 上的范围也是（尽管 timestamp 具有有限的精度，并且在理论上可以被当做离散的，最好认为它是连续的，因为通常并不关心它的步长）。

另一种考虑离散范围类型的方法是对每一个元素值都有一种清晰的“下一个”或“上一个”值。了解了这种思想之后，通过选择原来给定的下一个或上一个元素值来取代它，就可以在一个范围界限的包含和排除表达之间转换。例如，在一个整数范围类型中，[4,8]和(3,9)表示相同的值集合，但是对于 numeric 上的范围就不是这样。

一个离散范围类型应该具有一个正规化函数，它知道元素类型期望的步长。正规化函数负责把范围类型的相等值转换成具有相同的表达，特别是与包含或者排除界限一 致。如果没有指定一个正规化函数，那么具有不同格式的范围将总是会被当作不等， 即使它们实际上是表达相同的一组值。

内建的范围类型 int4range、int8range 和 daterange 都使用一种正规的形式，该形式包括下界并且排除上界，也就是[)。不过，用户定义的范围类型可以使用其他习惯。

4.14.7 定义新的范围类型

用户可以定义他们自己的范围类型。这样做最常见的原因是为了使用内建范围类型中没有提供的 subtype 上的范围。例如，要创建一个 subtype float8 的范围类型：

```
postgres=# CREATE TYPE floatrange AS RANGE ( subtype = float8,subtype_diff =
float8mi);
CREATE TYPE
postgres=# SELECT '[1.234, 5.678]'::floatrange;
[1.234,5.678]
```

因为 float8 没有有意义的“步长”，我们在这个例子中没有定义一个正规化函数。

定义自己的范围类型也允许你指定使用一个不同的子类型 B-树操作符类或者集合，以便更改排序顺序来决定哪些值会落入到给定的范围中。

如果 subtype 被认为是具有离散值而不是连续值，CREATE TYPE 命令应当指定一个 canonical 函数。正规化函数接收一个输入的范围值，并且必须返回一个可能具有不同界限和格式的等价的范围值。对于两个表示相同值集合的范围（例如[1, 7]和[1, 8)），正规的输

出必须一样。选择哪一种表达作为正规的没有关系，只要两个具有不同格式的等价值总是能被映射到具有相同格式的相同值就行。除了调整包含/排除界限格式外，假使期望的补偿比 subtype 能够存储的要大，一个正规化函数可能会舍入边界值。例如，一个 timestamp 之上的范围类型可能被定义为具有一个一小时的步长， 这样正规化函数可能需要对不是一小时的倍数的界限进行舍入，或者可能直接抛出一个错误。

另外，任何打算要和 GiST 或 SP-GiST 索引一起使用的范围类型应当定一个 subtype 差异或 subtype_diff 函数（没有 subtype_diff 时索引仍然能工作，但是可能效率不如提供了差异函数时高）。subtype 差异函数采用两个 subtype 输入值，并且返回表示为一个 float8 值的差（即 X 减 Y）。在我们上面的例子中，可以使用常规 float8 减法操作符 之下的函数。但是对于任何其他 subtype，可能需要某种类型转换。还可能需要一些关于如何把差异表达为数字的创新型想法。为了最大的可扩展性，subtype_diff 函数应 该同意选中的操作符类和排序规则所蕴含的排序顺序，也就是说，只要它的第一个参数根据排序顺序大于第二个参数，它的结果就应该是正值。

示例

```
postgres=# CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;
CREATE FUNCTION
postgres=# CREATE TYPE timerange AS RANGE ( subtype = time,subtype_diff =
time_subtype_diff);
CREATE FUNCTION
postgres=# SELECT '[11:10, 23:00]'::timerange;
[11:10:00,23:00:00]
```

更多关于创建范围类型的信息，参考 14.94 CREATE TYPE。

4.14.8索引

可以为范围类型的表列，创建 GiST 和 SP-GiST 索引。例如，创建 GiST 索引：

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

一个 GiST 或 SP-GiST 索引可以加速，涉及以下范围操作符的查询：=、 &&、 <@、 @>、 <<、 >>、 -|-、 &<以及 &> （详见 5.17 范围函数和操作符）。

此外，B-树和哈希索引可以在范围类型的表列上创建。对于这些索引类型，基本上唯一有用的范围操作就是等值。使用相应的< 和 >操作符，对于范围值定义有一种 B-树排序顺序，但是该顺序相当任意并且在真实世界中通常不怎么有用。范围类型的 B-树和哈希支持主要是为了允许在查询内部进行排序和哈希，而不是创建真正的索引。

4.15 对象标识符类型

GBase 8s 内部使用对象标识符（OID），作为系统表的主键。系统不会给用户自创的表增加 OID 字段。OID 类型代表对象标识符。

目前 OID 类型用一个四字节的无符号整数实现。因此，不建议在创建的表中使用 OID 字段做主键。

表 4-18 对象标识符类型

名称	引用	描述	示例
OID	——	数字化的对象标识符。	564182
CID	——	命令标识符。GBase 8s 系统字段 cmin 和 cmax 的数据类型。长度为 32 位。	——
XID	——	事务标识符。GBase 8s 系统字段 xmin 和 xmax 的数据类型。长度为 64 位。	——
TID	——	行标识符。GBase 8s 系统表字段 ctid 的数据类型。TID 是一对数值(块号,块内的行索引),标识该行在其所在表内的物理位置。	——
REGCONFIG	pg_ts_conf ig	文本搜索配置	english
REGDICTIONARY	pg_ts_dict	文本搜索字典	simple
REGOPER	pg_operat or	操作符名	——
REGOPERA	pg_operat or	带参数类型的操作符	*(integer,integer) 或 -

TOR			(NONE,integer)
REGPROC	pg_proc	函数名称	sum
REGPROCEDURE	pg_proc	带参数类型的函数	sum(int4)
REGCLASS	pg_class	关系名	pg_type
REGTYPE	pg_type	数据类型名	integer

OID 类型：主要作为数据库系统表中字段使用。示例：

```
postgres=# SELECT oid FROM pg_class WHERE relname = 'pg_type';
1247
```

OID 别名类型 REGCLASS：主要用于对象 OID 值的简化查找。示例：

```
postgres=# SELECT attrelid,attname,atttypid,attstattarget FROM pg_attribute
WHERE attrelid = 'pg_type'::REGCLASS;
 1247 | xc_node_id     |       23 |            0
 1247 | tableoid       |       26 |            0
 1247 | cmax           |       29 |            0
 1247 | xmax           |       28 |            0
 1247 | cmin           |       29 |            0
 1247 | xmin           |       28 |            0
 1247 | oid            |       26 |            0
 1247 | ctid           |       27 |            0
 1247 | typname        |       19 |           -1
 1247 | typnamespace   |       26 |           -1
 1247 | typowner       |       26 |           -1
 1247 | typlen         |       21 |           -1
 1247 | typbyval       |       16 |           -1
 1247 | typtype        |       18 |           -1
 1247 | typcategory    |       18 |           -1
 1247 | typispreferred |       16 |           -1
 1247 | typisdefined   |       16 |           -1
 1247 | typdelim       |       18 |           -1
 1247 | typrelid       |       26 |           -1
 1247 | typelem        |       26 |           -1
 1247 | typarray       |       26 |           -1
 1247 | typinput       |       24 |           -1
 1247 | typoutput      |       24 |           -1
```

```
1247 | typreceive    |  24 | -1
1247 | typsend       |  24 | -1
1247 | typmodin      |  24 | -1
1247 | typmodout     |  24 | -1
1247 | typanalyze    |  24 | -1
1247 | typalign      |  18 | -1
1247 | typstorage    |  18 | -1
1247 | typnotnull    |  16 | -1
1247 | typbasetype   |  26 | -1
1247 | typtypmod     |  23 | -1
1247 | typndims      |  23 | -1
1247 | typcollation  |  26 | -1
1247 | typdefaultbin | 194 | -1
1247 | typdefault    |  25 | -1
1247 | typacl        | 1034 | -1
```

4.16 伪类型

GBase 8s 支持伪类型，是一系列具有特殊用途的类型。伪类型不能作为字段的数据类型，但可以用于声明函数参数或者结果类型。GBase 8s 支持的伪类型。

表 4-19 伪类型

名称	描述
any	表示函数接受任何输入数据类型。
anyelement	表示函数接受任何数据类型。
anyarray	表示函数接受任意数组数据类型。
anynonarray	表示函数接受任意非数组数据类型。
anyenum	表示函数接受任意枚举数据类型。
anyrange	表示函数接受任意范围数据类型。
cstring	表示函数接受或者返回一个空结尾的 C 字符串。
internal	表示函数接受或者返回一种服务器内部的数据类型。

language_handler	声明一个过程语言调用句柄返回 language_handler。
fdw_handler	声明一个外部数据封装器返回 fdw_handler。
record	标识函数返回一个未声明的行类型。
trigger	声明一个触发器函数返回 trigger。
void	表示函数不返回数值。

用 C 编写的函数，无论内置或动态装载的，都可以接受或者返回任何伪数据类型。当伪类型作为参数类型使用时，用户需要保证函数的正常运行。

用过程语言编写的函数，只能使用实现语言允许的伪类型。目前，过程语言都不允许将伪类型作为参数类型，并且只允许使用 void 和 record 作为结果类型。一些多态的函数还支持使用 anyelement、anyarray、anynonarray anyenum 和 anyrange 类型。

伪类型 internal 用于声明只能在数据库内部调用而不能在 SQL 查询调用的函数。如果函数有至少一个 internal 类型的参数，则不能从 SQL 里直接调用。建议：除非有参数要求 internal 类型，否则不要创建任何声明返回 internal 的函数。

示例：

```
--创建表
postgres=# create table t1 (a int);
CREATE TABLE
--插入两条数据
postgres=# insert into t1 values(1),(2);
INSERT 0 2
--创建函数 showall()。
postgres=# CREATE OR REPLACE FUNCTION showall() RETURNS SETOF record AS $$ SELECT
count(*) from t1;
gbase$# $$
gbase-# LANGUAGE SQL;
CREATE FUNCTION
--调用函数 showall()。
postgres=# SELECT showall();
 (2)
--删除函数。
postgres=# DROP FUNCTION showall();
DROP FUNCTION
```

```
--删除表
postgres=# DROP TABLE t1;
DROP TABLE
```

4.17 列存表支持的数据类型

GBase 8s 数据库列存表支持的数据类型。

表 4-20 列存表支持的数据类型

类别	数据类型	长度	是否支持
数值类型（Numeric）	smallint	2	支持
	integer	4	支持
	bigint	8	支持
	decimal	-1	支持
	numeric	-1	支持
	real	4	支持
	double precision	8	支持
	smallserial	2	支持
	serial	4	支持
	bigserial	8	支持
	largeserial	-1	支持
货币类型（Monetary）	money	8	支持
字符类型（Character）	character varying(n), varchar(n)	-1	支持

	character(n), char(n)	n	支持
	character、char	1	支持
	text	-1	支持
	nvarchar	-1	支持
	nvarchar2	-1	支持
	name	64	不支持
	clob	-1	支持
日期/时间类型(Date/Time)	timestamp with time zone	8	支持
	timestamp without time zone	8	支持
	date	4	支持
	time without time zone	8	支持
	time with time zone	12	支持
	interval	16	支持

4.18 XML 类型

GBase 8s 支持 XML 类型，使用示例如下。

```
postgres=# CREATE TABLE xmltest ( id int, data xml );
CREATE TABLE
postgres=# INSERT INTO xmltest VALUES (1,'one');
INSERT 0 1
postgres=# INSERT INTO xmltest VALUES (2,'two');
INSERT 0 1
```

```
postgres=# SELECT * FROM xmltest ORDER BY 1;
id | data
----+--------------------
   | one
   | two
(2 rows)
postgres=# SELECT xmlconcat('', NULL, '');
xmlconcat
------------

(1 row)
postgres=# SELECT xmlconcat('', NULL, '');
xmlconcat
-------------

(1 row)
```

4.19 账本数据库使用的数据类型

账本数据库使用 HASH16 数据类型，存储行级 hash 摘要或表级 hash 摘要；使用 HASH32 数据类型，来存储全局 hash 摘要或者历史表校验 hash。

表 4-21 账本数据库数据类型

名称	描述	存储空间	范围
HASH16	以无符号 64 位整数存储。	8 字节	0 ~ +18446744073709551615
HASH32	以包含 16 个的无符号整形元素数的组存储。	16 字节	16 个元素的无符号整形数组能够包含的取值范围

HASH16 数据类型用来在账本数据库中存储行级或表级 hash 摘要，在获得长度为 16 个字符串的十六进制字符串的 hash 序列后，系统将调用 hash16in 函数将该序列转换为一个无符号 64 位整数存储进 HASH16 类型变量中。示例如下：

```
十六进制字符串：e697da2eaa3a775b 对应的无符号 64 位整数：16615989244166043483
十六进制字符串：ffffffffffffffff 对应的无符号 64 位整数：18446744073709551615
```

HASH32 数据类型用来在账本数据库中存储全局 hash 摘要或者历史表校验 hash，在获得长度为 32 个字符串的十六进制字符串的 hash 序列后，系统将调用 hash32in 函数将该 序列转换到一个包含 16 个无符号整形元素的数组中。示例如下：

十六进制字符串：685847ed1fe38e18f6b0e2b18s00edee
对应的 HASH32 数组：[104, 88, 71, 237, 31, 227, 142, 24, 246, 176, 226, 177, 140, 0, 237, 238]

4.20 SET 类型

SET 类型是一种包含字符串成员的集合类型，在表字段创建时定义。

4.20.1 规格描述

(1) SET 类型成员个数最大为 64 个，最小为 1 个。不能定义为空集。

(2) 成员名称长度最大为 255 个字符，允许使用空字符串作为成员名称。成员名称必须是字符常量，且不能是是计算后得到的字符常量，如 SET(‘a’ || ‘b’, ‘c’)。

(3) 成员名称不能包含逗号，成员名称不能重复。

(4) 不支持创建 SET 类型的数组和域类型。

(5) 只有在 sql_compatibility 参数值为 B 兼容模式下支持 SET 类型。

(6) 不支持 SET 类型作为列存表字段的数据类型。

(7) 不支持 SET 类型作为分区表的分区键。

(8) DROP TYPE 删除 SET 类型时，需要使用 CASCADE 方式删除，且关联的表字段也会被同时删除。

(9) 对于 USTORE 存储方式的表，如果表中包含 SET 类型的字段，且已经开启回收站功能，表被删除时，不会进入到回收站中，会直接删除。

(10) ALTER TABLE 不支持将 SET 类型字段的数据类型修改为其他 SET 类型。

(11) 表或者 SET 类型关联的表字段被删除时，或者表字段的 SET 类型修改为其他类型时，SET 数据类型也会被同步删除。

(12) 不支持以 CREATE TABLE { AS | LIKE } 的方式创建包含 SET 类型的表。

(13) SET 类型是随表字段创建的，其名称是组合而成的。如果 schema 中已经存在同名的数据类型，创建 SET 类型会失败。

(14) SET 类型支持与 int2、int4、int8、text 类型的=、<、>、<、<=、>、>=比较。

(15) SET 类型支持与 int2、int4、int8、float4、float8、numeric、char、varchar、text、nvarchar2 数据类型的转换。

4.20.2注意事项

SET 类型的表字段值必须是 SET 类型定义的集合的子集。如：

```
CREATE TABLE employee (
  name text,
  site SET('beijing','shanghai','nanjing','wuhan')
);
```

site 字段的值必须是上述集合定义中的子集，可以是空集合，如果提供的值在 SET 定义中的成员中不存在，会报错。如：

```
postgres=# INSERT INTO employee values('zhangsan', 'nanjing,beijing');
INSERT 0 1
postgres=# insert into employee values ('zhangsan', 'hangzhou');
ERROR:  invalid input value for set employee_site_set: 'hangzhou'
LINE 1: insert into employee values ('zhangsan', 'hangzhou');
                                                 ^

CONTEXT:  referenced column: site
postgres=#
```

INSERT 时无论用户提供的成员值顺序是怎样的，INSERT 成功后，查询到的 SET 类型的值，其成员的都是按照定义时的顺序输出的。

```
postgres=# select * from employee;
   name   |      site
----------+----------------
 zhangsan | beijing,nanjing
(1 rows)
```

SET 类型是以 bitmap 的方式存储的。SET 类型的成员按照定义时的顺序，赋予不同的值。如：SET(‘beijing’，‘shanghai’，‘nanjing’，‘wuhan’) 的类型，对应的值如下：

表 4-22 SET 成员与其对应的数值

SET 成员	SET 成员	SET 成员
‘beijing’	‘beijing’	‘beijing’
‘shanghai’	‘shanghai’	‘shanghai’
‘nanjing’	‘nanjing’	‘nanjing’

‘wuhan’	‘wuhan’	‘wuhan’

因此，如果给 SET 类型的字段赋值为数值时，会转换为对应的子集。如：9 对应的二进制值为 1001，对应的是子集是 ‘beijing,wuhan’。

```
postgres=# INSERT INTO employee values('lisi', 9);
INSERT 0 1
postgres=# select * from employee;
   name    |      site
-----------+----------------
 zhangsan | beijing,nanjing
 lisi     | beijing,wuhan
(2 rows)
```

5 函数和操作符

5.1 逻辑操作符

常用的逻辑操作符有 AND、OR 和 NOT。运算结果可能为：TRUE、FALSE 和 NULL，其中 NULL 代表未知。运算优先级顺序为：NOT>AND>OR。运算规则参见下表，表中的 a 和 b 代表逻辑表达式。

表 5-1 运算规则表

a	b	a AND b 的结果	a OR b 的结果	NOT a 的结果
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	NULL	NULL	NULL	NULL

操作符 AND 和 OR 具有交换性，即交换左右两个操作数，不影响其结果。

5.2 比较操作符

大部分数据类型都可以使用比较操作符，进行比较并返回一个布尔类型的值。

比较操作符均为双目操作符，待比较的两个数据类型必须是相同的数据类型或者是可以进行隐式转换的类型。

GBase 8s 提供的比较操作符，参见下表。

表 5-2 比较操作符

操作符	描述

<	小于
>	大于
<=	小于或等于
>=	大于或等于
=	等于
<> 或 !=或^=	不等于

比较操作符可以用于所有相关的数据类型。所有比较操作符都是双目操作符，返回布尔类型数值。当输入的数据不同且无法隐式转换时,比较操作将会失败。例如，1<2<3 表达式是非法的，因为 1 在表达式中视为布尔值，等同于 TRUE，无法与数值类型通过小于号（<）比较。

5.3 字符处理函数和操作符

GBase 8s 提供的字符处理函数和操作符，主要用于字符串与字符串、字符串与非字符串之间的连接，以及字符串的模式匹配操作。字符串处理函数除了 length 相关函数，其他函数和操作符的参数不支持大于 1GB 的 CLOB 类型。按首字母顺序排序，显示 GBase 8s 支持的字符处理函数。

注意

部分函数需指定相应模式名，才能调用。

使用时设置 search_path 为相应模式名，或语句中通过模式名.函数名格式调用。例如如 regexp_count 函数，设置 search_path 为 oracle 或使用 oracle.regexp_count。

5.3.1 A-G

- ascii(string)

  描述：参数 string 的第一个字符的 ASCII 码。

  返回值类型：integer

  示例：

```
postgres=# SELECT ascii('xyz');
```

```
ascii
-------
120
(1 row)
```

- bit_length(string)

  描述：字符串的位数。

  返回值类型：integer

  示例：

```
postgres=# SELECT bit_length('world');
bit_length
------------
40
(1 row)
```

- btrim(string text [, characters text])

  描述：从 string 开头和结尾删除只包含 characters 中字符（缺省是空白）的最长字符串。

  返回值类型：text

  示例：

```
postgres=# SELECT btrim('sring' , 'ing');
btrim
-------
sr
(1 row)
```

- char_length(string)或 character_length(string)

  描述：字符串中的字符个数。

  返回值类型：int

  示例：

```
postgres=# SELECT char_length('hello');
char_length
-------------
5
(1 row)
```

- chr(integer)

描述：给出 ASCII 码的字符。

返回值类型：varchar

示例：

```
postgres=# SELECT chr(65);
chr
-----
A
(1 row)
```

- concat(str1,str2)

描述：将字符串 str1 和 str2 连接并返回。

须知

数据库 SQL 兼容模式设置为 MY 的情况下，参数 str1 或 str2 为 NULL 会导致返回结果为 NULL。

返回值类型：varchar

示例：

```
postgres=# SELECT concat('Hello', ' World!');
    concat
--------------
 Hello World!
(1 row)
postgres=# SELECT concat('Hello', NULL);
concat
--------
 Hello
(1 row)
```

- concat_ws(sep text, str"any" [, str"any" [, ...]])

描述：以第一个参数为分隔符，链接第二个以后的所有参数。NULL 参数被忽略。

须知

- 如果第一个参数值是 NULL，会导致返回结果为 NULL。
- 如果第一个参数值是空字符串（''），且数据库 SQL 兼容模式设置为 A 的情况 下，会导致返回结果为 NULL。这是因为 A 兼容模式>将''作为 NULL 处理，避免 此类

行为，可以将数据库 SQL 兼容模式改为 B、C 或者 PG。

返回值类型：text

示例：

```
postgres=# SELECT concat_ws(',', 'ABCDE', 2, NULL, 22);
concat_ws
------------
ABCDE,2,22
(1 row)
```

- convert(string bytea, src_encoding name, dest_encoding name)

描述：以 dest_encoding 指定的目标编码方式转化字符串 bytea。src_encoding 指定源编码方式，在该编码下，string 必须是合法的。

返回值类型：bytea

示例：

```
postgres=# SELECT convert('text_in_utf8', 'UTF8', 'GBK');
          convert
----------------------------
 \x746578745f696e5f75746638
(1 row)
```

如 GBK 和 LATIN1 之间的转换规则是不存在的，具体转换规则可以通过查看系统表 pg_conversion 获得。

示例：

```
postgres=# show server_encoding;
server_encoding
-----------------
 SQL_ASCII
(1 row)
postgres=# SELECT convert_from('some text', 'GBK');
convert_from
--------------
 some text
(1 row)
postgres=# SELECT convert_to('some text', 'GBK');
convert_to
----------------------
```

```
\x736f6d652074657874
(1 row)
db_latin1=# SELECT convert('some text', 'GBK', 'LATIN1');
convert
----------------------
\x736f6d652074657874
(1 row)
```

- convert_from(string bytea, src_encoding name)

  描述：以数据库的编码方式转化字符串 bytea。

  src_encoding 指定源编码方式，在该编码下，string 必须是合法的。

  返回值类型：text

  示例：

```
postgres=# SELECT convert_from('text_in_utf8', 'UTF8');
convert_from
--------------
text_in_utf8
(1 row)
```

- convert_to(string text, dest_encoding name)

  描述：将字符串转化为 dest_encoding 的编码格式。

  返回值类型：bytea

  示例：

```
postgres=# SELECT convert_to('some text', 'UTF8');
convert_to
----------------------
\x736f6d652074657874
(1 row)
```

- decode(string text, format text)

  描述：将二进制数据从文本数据中解码。

  返回值类型：bytea

  示例：

```
postgres=# SELECT decode('MTIzAAE=', 'base64');
decode
```

```
--------------
\x3132330001
(1 row)
```

- encode(data bytea, format text)

  描述：将二进制数据编码为文本数据。

  返回值类型：text

  示例：

```
postgres=# SELECT encode(E'123\\000\\001', 'base64');
encode
----------
MTIzAAE=
(1 row)
```

  说明

  若字符串中存在换行符，如字符串由一个换行符和一个空格组成，在 GBase 8s 中 LENGTH 和 LENGTHB 的值为 2。

  对于 CHAR(n)类型，n 是指字符个数。因此，对于多字节编码的字符集，LENGTHB 函数返回的长度可能大于 n。

GBase 8s 目前支持多种类型的数据库，目前有 4 种，分别是 A 类型，B 类型，C 类型以 PG 类型。不指定数据库类型时，默认为 A 类型。A 的词法分析器与另外三种不一样，在 A 中空字符串会被当作是 NULL。所以，当使用 A 类型的数据库时，假如上述字符操作函数中有空字符串作为参数，会出现没有输出的情况。例如：

```
postgres=# SELECT translate('12345','123','');
translate
-----------


(1 row)
```

这是因为内核在调用相应的函数进行处理前，会判断所输入的参数中是否含有 NULL，假如有，则不会调用相应的函数，因此会没有输出。而在 PG 模式下，字符串的处理方式与 postgresql 保持一致，因此不会有上述问题产生。

- format(formatstr text [, str"any" [, ...]])

  描述：格式化字符串。

返回值类型：text

示例：

```
postgres=# SELECT format('Hello %s, %1$s', 'World');
      format
-------------------
 Hello World, World
(1 row)
```

5.3.2 H-N

- instr(text,text,int,int)

描述： instr(string1,string2,int1,int2)返回在 string1 中从 int1 位置开始匹配到第 int2 次 string2 的位置，第一个 int 表示开始匹配起始位置，第二个 int 表示匹配的次数。

返回值类型：int 示例：

```
postgres=# SELECT instr( 'abcdabcdabcd', 'bcd', 2, 2 );
instr
-------
     6
(1 row)
```

- instr(string,substring[,position,occurrence])

描述：从字符串 string 的 position（缺省时为 1）所指的位置开始查找并返回第 occurrence（缺省时为 1）次出现子串 substring 的位置的值。本函数以字符为计算单位，如一个汉字为一个字符。

  - 当 position 为 0 时，返回 0。
  - 当 position 为负数时，从字符串倒数第 n 个字符往前逆向搜索。n 为 position 的绝对值。

返回值类型：integer

示例：

```
postgres=# SELECT instr('corporate floor','or', 3);
instr
-------
5
(1 row)
```

```
postgres=# SELECT instr('corporate floor','or',-3,2);
instr
-------
2
(1 row)
```

- initcap(string)

  描述：将字符串中的每个单词的首字母转化为大写，其他字母转化为小写。

  返回值类型：text

  示例：

```
postgres=# SELECT initcap('hi THOMAS');
initcap
-----------
Hi Thomas
(1 row)
```

- lengthb(text/bpchar)

  描述：获取指定字符串的字节数。

  返回值类型：int

  示例：

```
postgres=# SELECT lengthb('hello');
lengthb
---------
5
(1 row)
```

- lengthb(string)

  描述：获取参数 string 中字节的数目。与字符集有关，同样的中文字符，在 GBK 与 UTF8 中，返回的字节数不同。

  返回值类型：integer

  示例：

```
postgres=# SELECT lengthb('Chinese');
lengthb
---------
7
```

```
(1 row)
```

- left(str text, n int)

  描述：返回字符串的前 n 个字符。当 n 是负数时，返回除最后|n|个字符以外的所有字符。

  返回值类型：text

  示例：

```
postgres=# SELECT left('abcde', 2);
left
------
ab
(1 row)
```

- length(string bytea, encoding name)

  描述：指定 encoding 编码格式的 string 的字符数。在这个编码格式中，string 必须是有效的。

  返回值类型：int 示例：

```
postgres=# SELECT length('jose', 'UTF8');
length
--------
4
(1 row)
```

  说明

  如果是查询 bytea 类型的长度，指定 utf8 编码时，最大长度只能为 536870888。

- length(string)

  描述：获取参数 string 中字符的数目。

  返回值类型：integer

  示例：

```
postgres=# SELECT length('abcd');
length
--------
4
(1 row)
```

- lower(string)

描述：把字符串转化为小写。

返回值类型：varchar

示例：

```
postgres=# SELECT lower('TOM');
lower
-------
tom
(1 row)
```

- lpad(string text, length int [, fill text])

描述：通过填充字符 fill（缺省时为空白），把 string 填充为 length 长度。如果 string 已经比 length 长则将其尾部截断。

返回值类型：text

示例：

```
postgres=# SELECT lpad('hi', 5, 'xyza');
lpad
-------
 xyzhi
(1 row)
```

- lpad(string varchar, length int[, repeat_string varchar])

描述：在 string 的左侧添上一系列的 repeat_string（缺省为空白）来组成一个总长度为 n 的新字符串。

如果 string 本身的长度比指定的长度 length 长，则本函数将把 string 截断并把前面长度为 length 的字符串内容返回。

返回值类型：varchar

示例：

```
postgres=# SELECT lpad('PAGE 1',15,'*.');
      lpad
-----------------
 *.*.*.*.*PAGE 1
(1 row)
postgres=# SELECT lpad('hello world',5,'abcd');
lpad
```

```
-------
hello
(1 row)
```

- ltrim(string [, characters])

描述：从字符串 string 的开头删除只包含 characters 中字符（缺省是一个空白）的最长的字符串。

返回值类型：varchar

示例：

```
postgres=# SELECT ltrim('xxxxTRIM','x');
ltrim
-------
TRIM
(1 row)
```

- md5(string)

描述：将 string 使用 MD5 加密，并以 16 进制数作为返回值。

说明

MD5 加密算法安全性低，存在安全风险，不建议使用。

返回值类型：text

示例：

```
postgres=# SELECT md5('ABC');
               md5
----------------------------------
 902fbdd2b1df0c4f70b4a5d23525e932
(1 row)
```

- notlike(x bytea name text, y bytea text)

描述：比较 x 和 y 是否不一致。

返回值类型：bool

示例：

```
postgres=# SELECT notlike(1,2);
notlike
--------------
```

```
t
(1 row)
postgres=# SELECT notlike(1,1);
notlike
--------------
f
(1 row)
```

- nlssort(string text, sort_method text)

描述：以 sort_method 指定的排序方式返回字符串在该排序模式下的编码值，该编 码值可用于排序，其决定了 string 在这种排序模式下的先后位置。目前支持的 sort_method 为'nls_sort=schinese_pinyin_m'和'nls_sort=generic_m_ci'。其中， 'nls_sort=generic_m_ci'仅支持纯英文不区分大小写排序。

string 类型 ： text sort_method 类型：

text 返回值类型：text

示例：

```
postgres=# SELECT nlssort('A', 'nls_sort=schinese_pinyin_m');
    nlssort
----------------
 01EA0000020006
(1 row)
postgres=# SELECT nlssort('A', 'nls_sort=generic_m_ci');
  nlssort
------------
 01EA000002
(1 row)
```

5.3.3 O-T

- octet_length(string)

描述：字符串中的字节数。

返回值类型：int

示例：

```
postgres=# SELECT octet_length('jose');
octet_length
--------------
```

```
4
(1 row)
```

- overlay(string placing string FROM int [for int])

描述：替换子字符串。FROM int 表示从第一个 string 的第几个字符开始替换，for int 表示替换第一个 string 的字符数目。

返回值类型：text

示例：

```
postgres=# SELECT overlay('hello' placing 'world' from 2 for 3 );
overlay
---------
hworldo
(1 row)
```

- position(substring in string)

描述：指定子字符串的位置。字符串区分大小写。

返回值类型：int，字符串不存在时返回 0。

示例：

```
postgres=# SELECT position('ing' in 'string');
position
----------
4
(1 row)
```

- pg_client_encoding()

描述：当前客户端编码名称。

返回值类型：name

示例：

```
postgres=# SELECT pg_client_encoding();
pg_client_encoding
--------------------
UTF8
(1 row)
```

- quote_ident(string text)

描述：返回适用于 SQL 语句的标识符形式（使用适当的引号进行界定）。只有在必要的时候才会添加引号（字符串包含非标识符字符或者会转换大小写的字符）。返回值中嵌入的引号都写了两次。

返回值类型：text

示例：

```
postgres=# SELECT quote_ident('hello world');
quote_ident
--------------
"hello world"
(1 row)
```

- quote_literal(string text)

描述：返回适用于在 SQL 语句里当作文本使用的形式（使用适当的引号进行界定）。

返回值类型：text

示例：

```
postgres=# SELECT quote_literal('hello');
quote_literal
---------------
'hello'
(1 row)
```

如果出现如下写法，text 文本将进行转义。

```
postgres=# SELECT quote_literal(E'0\'hello');
quote_literal
---------------
'0''hello'
(1 row)
```

如果出现如下写法，反斜杠会写入两次。

```
postgres=# SELECT quote_literal('0\hello');
quote_literal
---------------
E'0\\hello'
(1 row)
```

如果参数为 NULL，返回空。如果参数可能为 null，通常使用函数 quote_nullable 更适用。

```
postgres=# SELECT quote_literal(NULL);
quote_literal
---------------


(1 row)
```

- quote_literal(value anyelement)

  描述：将给定的值强制转换为 text，加上引号作为文本。

  返回值类型：text

  示例：

```
postgres=# SELECT quote_literal(42.5);
quote_literal
---------------
'42.5'
(1 row)
```

  如果出现如下写法，定值将进行转义。

```
postgres=# SELECT quote_literal(E'O\'42.5');
quote_literal
---------------
'O''42.5'
(1 row)
```

  如果出现如下写法，反斜杠会写入两次。

```
postgres=# SELECT quote_literal('O\42.5');
quote_literal
---------------
E'O\\42.5'
(1 row)
```

- quote_nullable(string text)

  描述：返回适用于在 SQL 语句里当作字符串使用的形式（使用适当的引号进行界定）。

  返回值类型：text 示例：

```
postgres=# SELECT quote_nullable('hello');
quote_nullable
----------------
'hello'
(1 row)
```

如果出现如下写法，text 文本将进行转义。

```
postgres=# SELECT quote_nullable(E'O\'hello');
quote_nullable
----------------
'O''hello'
(1 row)
```

如果出现如下写法，反斜杠会写入两次。

```
postgres=# SELECT quote_nullable('O\hello');
quote_nullable
----------------
E'O\\hello'
(1 row)
```

如果参数为 NULL，返回 NULL。

```
postgres=# SELECT quote_nullable(NULL);
quote_nullable
----------------
NULL
(1 row)
```

- quote_nullable(value anyelement)

  描述：将给定的参数值转化为 text，加上引号作为文本。

  返回值类型：text

  示例：

```
postgres=# SELECT quote_nullable(42.5);
quote_nullable
----------------
'42.5'
(1 row)
```

如果出现如下写法，定值将进行转义。

```
postgres=# SELECT quote_nullable(E'O\'42.5');
quote_nullable
----------------
'O''42.5'
(1 row)
```

如果出现如下写法，反斜杠会写入两次。

```
postgres=# SELECT quote_nullable('0\42.5');
quote_nullable
----------------
E'0\\42.5'
(1 row)
```

如果参数为 NULL，返回 NULL。

```
postgres=# SELECT quote_nullable(NULL);
quote_nullable
----------------
NULL
(1 row)
```

- rawcat(raw,raw)

  描述：字符串拼接函数。

  返回值类型：raw

  示例：

```
postgres=# SELECT rawcat('ab','cd');
rawcat
--------
ABCD
(1 row)
```

- regexp_like(text,text,text)

  描述：正则表达式的模式匹配函数。

  返回值类型：bool

  示例：

```
postgres=# SELECT regexp_like('str','[ac]');
regexp_like
-------------
 f
(1 row)
```

- replace(string varchar, search_string varchar, replacement_string varchar)

  描述：把字符串 string 中所有子字符串 search_string 替换成子字符串 replacement_string 。

  返回值类型：varchar

示例：

```
postgres=# SELECT replace('jack and jue','j','bl');
replace
-----------------
black and blue
(1 row)
```

- rpad(string varchar, length int [, fill varchar])

描述：使用填充字符 fill（缺省时为空白），把 string 填充到 length 长度。如果 string 已经比 length 长则将其从尾部截断。length 参数在 GBase 8s 中表示字符长度。一个汉字长度计算为一个字符。

返回值类型：varchar

示例：

```
postgres=# SELECT rpad('hi',5,'xyza');
rpad
-------
hixyz
(1 row)
postgres=# SELECT rpad('hi',5,'abcdefg');
rpad
-------
hiabc
(1 row)
```

- regexp_substr(source_char, pattern)

描述：正则表达式的抽取子串函数。SQL 语法兼容 A 和 B 的情况下，GUC 参数 behavior_compat_options 的值包含 aformat_regexp_match 时，.不能匹配 '\n' 字符；不包含 aformat_regexp_match 时，.能够匹配'\n'字符。

返回值类型：text

示例：

```
postgres=# SELECT regexp_substr('500 Hello World, Redwood Shores, CA', ',[^,]+,')
"REGEXPR_SUBSTR";
REGEXPR_SUBSTR
-------------------
, Redwood Shores,
(1 row)
```

- regexp_replace(string, pattern, replacement [,flags])

描述：替换匹配 POSIX 正则表达式的子字符串。如果没有匹配 pattern，那么返回不加修改的 string 串。如果有匹配，则返回的 string 串里面的匹配子串将被 replacement 串替换掉。

replacement 串可以包含\n，其中\n 是 1 到 9，表明 string 串里匹配模式里第 n 个圆括号子表达式的子串应该被插入，并且它可以包含\&表示应该插入匹配整个模式的子串。

可选的 flags 参数包含零个或多个改变函数行为的单字母标记。i 表示进行大小写无关的匹配，g 表示替换每一个匹配的子字符串而不仅仅是第一个。m 表示按照多行模式匹配。SQL 语法兼容 A 和 B 的情况下，n 选项在 GUC 参数 behavior_compat_options 的值包含 aformat_regexp_match 时，表示 . 能够匹配'\n' 字符，flags 中没有指定 n 时，默认不能匹配'\n'字符；值不包含 aformat_regexp_match 时，. 默认能匹配'\n'字符。n 选项的含义与 m 选项一致。

返回值类型：varchar

示例：

```
postgres=# SELECT regexp_replace('Thomas', '.[mN]a.', 'M');
regexp_replace
----------------
ThM
(1 row)
postgres=# SELECT regexp_replace('foobarbaz','b(..)', E'X\\1Y', 'g') AS RESULT;
result
-------------
fooXarYXazY
(1 row)
```

- string [NOT] LIKE pattern [ESCAPE escape-character]

描述：模式匹配函数。

如果 pattern 不包含百分号或者下划线，该模式只代表它本身，这时候 LIKE 的行为 就像等号操作符。在 pattern 里的下划线（_）匹配任何单个字符；而一个百分号（%）匹配零或多个任何字符。

要匹配下划线或者百分号本身，在 pattern 里相应的字符必须前导逃逸字符。缺省的逃逸字符是反斜杠，但是用户可以用 ESCAPE 子句指定一个。要匹配逃逸字符本身，写两个逃逸字符。

返回值类型：Boolean

示例：

```
postgres=# SELECT 'AA_BBCC' LIKE '%A@_B%' ESCAPE '@' AS RESULT;
result
--------
t
(1 row)
postgres=# SELECT 'AA_BBCC' LIKE '%A@_B%' AS RESULT;
result
--------
f
(1 row)
postgres=# SELECT 'AA@_BBCC' LIKE '%A@_B%' AS RESULT;
result
--------
t
(1 row)
```

- repexp_replace(string text, pattern text [, replacement text [, position int [, occurrence int [, flags text]]]])

描述：替换匹配 POSIX 正则表达式的子字符串。如果没有匹配 pattern，那么返回不加修改的 string 串。如果有匹配，则返回的 string 串里面的匹配子串将被 replacement 串替换掉。

参数说明：

- -- string：用于匹配的源字符串
- -- pattern：用于匹配的正则表达式模式串
- -- replacement：可选参数，用于替换匹配子串的字符串。如果不给定参数值或 者为 null，表示用空串替换。
- -- position：可选参数，表示从源字符串的第几个字符开始匹配，默认值为 1。
- -- occurrence：可选参数，表示替换第 occurrence 个匹配的子串。默认值为 0， 表示替换所有匹配到的子串。
- -- flags：可选参数，包含零个或多个改变函数匹配行为的单字母标记。其中：m 表示按照多行模式匹配。SQL 语法兼容 A 和 B 的情况下，n 选项在 GUC 参数 behavior_compat_options 值包含 aformat_regexp_match 时，表示 . 能够匹配 '\n' 字

符，flags 中没有指定 n 时，默认.不能匹配 '\n' 字符；值不包含 aformat_regexp_match 时，. 默认能匹配'\n'字符。n 选项的含义与 m 选项一致。

返回值类型：text

示例：

```
postgres=# SELECT regexp_replace('Thomas','.[mN]a.','M');
regexp_replace
----------------
 ThM
(1 row)
postgres=# SELECT regexp_replace('footbarbaz','b(..)',E'X\\1Y','g') AS RESULT;
    result
--------------
 footXarYXazY
(1 row)
```

- regexp_substr(string text, pattern text [, position int [, occurrence int [, flags text]]])

描述：正则表达式的抽取子串函数。与 substr 功能相似，正则表达式出现多个并列的括号时，也全部处理。

参数说明：

- -- string：用于匹配的源字符串。
- -- pattern：用于匹配的正则表达式模式串。
- -- position：可选参数，表示从源字符串的第几个字符开始匹配，默认值为 1。
- -- occurrence：可选参数，表示抽取第几个满足匹配的子串，为，默认值为 1。
- -- flags：可选参数，包含零个或多个改变函数匹配行为的单字母标记。其中：m 表示按照多行模式匹配。SQL 语法兼容 A 和 B 的情况下，n 选项在 GUC 参数 behavior_compat_options 值包含 aformat_regexp_match 时，表示 . 能够匹配 '\n' 字符，flags 中没有指定 n 时，默认.不能匹配 '\n' 字符；值不包含 aformat_regexp_match 时，. 默认能匹配'\n'字符。n 选项的含义与 m 选项一致。

返回值类型：text

示例：

```
postgres=# SELECT regexp_substr('str','[ac]');
regexp_substr
---------------


(1 row)
postgres=# SELECT regexp_substr('foobarbaz', 'b(..)', 3, 2) AS RESULT;
result
--------
baz
(1 row)
```

- regexp_count(string text, pattern text [, position int [, flags text]])

  描述：获取满足匹配的子串个数。

  参数说明：

  - -- string：用于匹配的源字符串。
  - -- pattern：用于匹配的正则表达式模式串。
  - -- position：表示从源字符串的第几个字符开始匹配，为可选参数，默认值为 1。
  - -- flags：可选参数，包含零个或多个改变函数匹配行为的单字母标记。其中：m 表示按照多行模式匹配。SQL 语法兼容 A 和 B 的情况下，n 选项在 GUC 参数 behavior_compat_options 值包含 aformat_regexp_match 时，表示 . 能够匹配 '\n' 字符，flags 中没有指定 n 时，默认.不能匹配 '\n' 字符；值不包含 aformat_regexp_match 时，. 默认能匹配'\n'字符。n 选项的含义与 m 选项一致。

  返回值类型：int

  示例：

```
postgres=# SELECT regexp_count('foobarbaz','b(..)', 5) AS RESULT;
result
--------
1
(1 row)
```

- regexp_instr(string text, pattern text [, position int [, occurrence int [, return_opt int [, flags text]]]])

  描述：获取满足匹配条件的子串位置（从 1 开始）。如果没有匹配的子串，则返回 0。

参数说明:

- -- string：用于匹配的源字符串。
- -- pattern：用于匹配的正则表达式模式串。
- -- position：可选参数，表示从源字符串的第几个字符开始匹配，默认值为 1。
- -- occurrence：可选参数，表示获取第 occurrence 个匹配子串的位置，默认值为 1。
- -- return_opt：可选参数，用于控制返回匹配子串的首字符位置还是尾字符位 置。取值为 0 时，返回匹配子串的第一个字符的位置（从 1 开始计算），取值为大于 0 的值时，返回匹配子串的尾字符的下一个字符的位置。默认值为 0。
- -- flags：可选参数，包含零个或多个改变函数匹配行为的单字母标记。其中：m 表示按照多行模式匹配。SQL 语法兼容 A 和 B 的情况下，n 选项在 GUC 参数 behavior_compat_options 值包含 aformat_regexp_match 时，表示 . 能够匹配 '\n' 字符，flags 中没有指定 n 时，默认.不能匹配 '\n' 字符；值不包含 aformat_regexp_match 时，. 默认能匹配'\n'字符。n 选项的含义与 m 选项一致。

返回值类型：int

示例:

```
postgres=# SELECT regexp_instr('foobarbaz','b(..)', 1, 1, 0) AS RESULT;
 result
--------
 4
(1 row)
postgres=# SELECT regexp_instr('foobarbaz','b(..)', 1, 2, 0) AS RESULT;
 result
--------
  7
(1 row)
```

- regexp_matches(string text, pattern text [, flags text])

描述：返回 string 中所有匹配 POSIX 正则表达式的子字符串。如果 pattern 不匹配， 该函数不返回行。如果模式不包含圆括号子表达式，则每一个被返回的行都是一个单一元素的文本数组，其中包括匹配整个模式的子串。如果模式包含圆括号子表达式，该函数返回一个

文本数组，它的第 n 个元素是匹配模式的第 n 个圆括号子表达式的子串。

flags 参数为可选参数，包含零个或多个改变函数行为的单字母标记。i 表示进行大小写无关的匹配，g 表示替换每一个匹配的子字符串而不仅仅是第一个。

须知

如果提供了最后一个参数，但参数值是空字符串（''），且数据库 SQL 兼容模式设置为 A 的情况下，会导致返回结果为空集。这是因为 A 兼容模式将''作为 NULL 处理，避免此类行为的方式有如下几种：

-- 将数据库 SQL 兼容模式改为 C；

-- 不提供最后一个参数，或最后一个参数不为空字符串。

返回值类型：setof text[]

示例：

```
postgres=# SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');
regexp_matches
----------------
 {bar,beque}
(1 row)
postgres=# SELECT regexp_matches('foobarbequebaz', 'barbeque');
regexp_matches
----------------
{barbeque}
(1 row)
postgres=# SELECT regexp_matches('foobarbequebazilbarfbonk',
'(b[^b]+)(b[^b]+)', 'g');
regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

- regexp_split_to_array(string text, pattern text [, flags text])

描述：用 POSIX 正则表达式作为分隔符，分隔 string。和 regexp_split_to_table 相 同，不过 regexp_split_to_array 会把它的结果以一个 text 数组的形式返回。

返回值类型：text[]

示例：

```
postgres=# SELECT regexp_split_to_array('hello world', E'\\s+');
 regexp_split_to_array
-----------------------
{hello,world}
(1 row)
```

- regexp_split_to_table(string text, pattern text [, flags text])

描述：用 POSIX 正则表达式作为分隔符，分隔 string。如果没有与 pattern 的匹配， 该函数返回 string。如果有至少有一个匹配，对每一个匹配它都返回从上一个匹配的末尾（或者串的开头）到这次匹配开头之间的文本。当没有更多匹配时，它返回从上一次匹配的末尾到串末尾之间的文本。

flags 参数包含零个或多个改变函数行为的单字母标记。i 表示进行大小写无关的匹配。

返回值类型：setof text

示例：

```
postgres=# SELECT regexp_split_to_table('hello world', E'\\s+');
regexp_split_to_table
-----------------------
hello
world
(2 rows)
```

- repeat(string text, number int)

描述：将 string 重复 number 次。

返回值类型：text。

示例：

```
postgres=# SELECT repeat('Pg', 4);
repeat
----------
PgPgPgPg
(1 row)
```

说明

由于数据库内存分配机制限制单次内存分配不可超过 1GB，因此 number 最大值不应超过(1G-x)/lengthb(string) - 1。x 为头信息长度，通常大于 4 字节，其具体值在不同的场景下存在差异。

- replace(string text, from text, to text)

  描述：把字符串 string 里出现地所有子字符串 from 的内容替换成子字符串 to 的内容。

  返回值类型：text

  示例：

```
postgres=# SELECT replace('abcdefabcdef', 'cd', 'XXX');
    replace
---------------
 abXXXefabXXXef
(1 row)
```

- replace(string, substring)

  描述：删除字符串 string 里出现的所有子字符串 substring 的内容。

  string 类型：text substring 类型：

  text 返回值类型：text

  示例：

```
postgres=# SELECT replace('abcdefabcdef', 'cd');
replace
---------------
abefabef
(1 row)
```

- reverse(str)

  描述：返回颠倒的字符串。

  返回值类型：text

  示例：

```
postgres=# SELECT reverse('abcde');
reverse
---------
edcba
(1 row)
```

- right(str text, n int)

  描述：返回字符串中的后 n 个字符。当 n 是负值时，返回除前|n|个字符以外所有字符。

返回值类型：text

示例：

```
postgres=# SELECT right('abcde', 2);
right
-------
de
(1 row)
postgres=# SELECT right('abcde', -2);
right
-------
cde
(1 row)
```

- rpad(string text, length int [, fill text])

描述：使用填充字符 fill（缺省时为空白），把 string 填充到 length 长度。如果 string 已经比 length 长则将其从尾部截断。

返回值类型：text

示例：

```
postgres=# SELECT rpad('hi', 5, 'xy');
rpad
-------
hixyx
(1 row)
```

- rtrim(string text [, characters text])

描述：从字符串 string 的结尾删除只包含 characters 中字符（缺省是个空白）的最长的字符串。

返回值类型：text

示例：

```
postgres=# SELECT rtrim('trimxxxx', 'x');
rtrim
-------
trim
(1 row)
```

- rtrim(string [, characters])

描述：从字符串 string 的结尾删除只包含 characters 中字符（缺省是个空白）的最长的字符串。

返回值类型：varchar

示例：

```
postgres=# SELECT rtrim('TRIMxxxx','x');
rtrim
-------
TRIM
(1 row)
```

- REGEXP_LIKE(source_string, pattern [, match_parameter])

描述：正则表达式的模式匹配函数。

source_string 为源字符串，pattern 为正则表达式匹配模式。 match_parameter 为匹配选项，可取值为：

- -- 'i'：大小写不敏感。
- -- 'c'：大小写敏感。
- -- 'n'：允许正则表达式元字符“.”匹配换行符。
- -- 'm'：将 source_string 视为多行。
- -- 若忽略 match_parameter 选项，默认为大小写敏感，“.”不匹配换行符，source_string 视为单行。

返回值类型：Boolean

示例：

```
postgres=# SELECT regexp_like('ABC', '[A-Z]');
regexp_like
-------------
t
(1 row)
postgres=# SELECT regexp_like('ABC', '[D-Z]');
regexp_like
-------------
f
(1 row)
```

```
postgres=# SELECT regexp_like('ABC', '[a-z]','i');
regexp_like
-------------
t
(1 row)
```

- substring_inner(string [from int] [for int])

  描述：截取子字符串，from int 表示从第几个字符开始截取，for int 表示截取几个字节。

  返回值类型：text

  示例：

```
postgres=# select substring_inner('adcde', 2,3);
substring_inner
-----------------
dcd
(1 row)
```

- substring(string [from int] [for int])

  描述：截取子字符串，from int 表示从第几个字符开始截取，for int 表示截取几个字节。

  返回值类型：text

  示例：

```
postgres=# SELECT substring('Thomas' from 2 for 3);
substring
-----------
hom
(1 row)
```

- substring(string from pattern)

  描述：截取匹配 POSIX 正则表达式的子字符串。如果没有匹配它返回空值，否则返回文本中匹配模式的那部分。

  返回值类型：text

  示例：

```
postgres=# SELECT substring('Thomas' from '...$');
substring
-----------
mas
```

```
(1 row)
postgres=# SELECT substring('foobar' from 'o(.)b');
result
--------
o
(1 row)
postgres=# SELECT substring('foobar' from '(o(.)b)');
result
--------
oob
(1 row)
```

说明

如果 POSIX 正则表达式模式包含任何圆括号，那么将返回匹配第一对子表达式（对应第一个左圆括号的）的文本。如果需要使用圆括号而不产生以上结果，可以在整个表达式外边放上一对圆括号。

- substring(string from pattern for escape)

描述：截取匹配 SQL 正则表达式的子字符串。声明的模式必须匹配整个数据串， 否则函数失败并返回空值。为了标识在成功的时候应该返回的模式部分，模式必须包含逃逸字符的两次出现，并且后面要跟上双引号（"）。匹配这两个标记之间的模式的文本将被返回。

返回值类型：text

示例：

```
postgres=# SELECT substring('Thomas' from '%#"o_a#"_' for '#');
substring
-----------
oma
(1 row)
```

- substrb(text,int,int)

描述：提取子字符串，第一个 int 表示提取的起始位置，第二个表示提取几位字符。

返回值类型：text

示例：

```
postgres=# SELECT substrb('string',2,3);
substrb
---------
```

```
tri
(1 row)
```

- substrb(text,int)

  描述：提取子字符串，int 表示提取的起始位置。

  返回值类型：text

  示例：

```
postgres=# SELECT substrb('string',2);
substrb
---------
tring
(1 row)
```

- substr(bytea,from,count)

  描述：从参数 bytea 中抽取子字符串。from 表示抽取的起始位置，count 表示抽取的子字符串长度。

  返回值类型：text

  示例：

```
postgres=# SELECT substr('string',2,3);
substr
--------
tri
(1 row)
```

- string || string

  描述：连接字符串。

  返回值类型：text

  示例：

```
postgres=# SELECT 'MPP'||'DB' AS RESULT;
result
--------
MPPDB
(1 row)
```

- string || non-string 或 non-string || string

描述：连接字符串和非字符串。

返回值类型：text

示例：

```
postgres=# SELECT 'Value: '||42 AS RESULT;
result
-----------
Value: 42
(1 row)
```

- split_part(string text, delimiter text, field int)

描述：根据 delimiter 分隔 string 返回生成的第 field 个子字符串（从出现第一个 delimiter 的 text 为基础）。

返回值类型：text

示例：

```
postgres=# SELECT split_part('abc~@~def~@~ghi', '~@~', 2);
split_part
------------
def
(1 row)
```

- strpos(string, substring)

描述：指定的子字符串的位置。和 position(substring in string)一样，不过参数顺序相反。

返回值类型：int

示例：

```
postgres=# SELECT strpos('source', 'rc');
strpos
--------
4
(1 row)
```

- substr(string,from)

描述：

从参数 string 中抽取子字符串。from 表示抽取的起始位置。from 为 0 时，按 1 处理。from 为正数时，抽取从 from 到末尾的所有字符。from 为负数时，抽取字符串的后 n 个字符，n

为 from 的绝对值。

返回值类型：varchar

示例：

from 为正数时：

```
postgres=# SELECT substr('ABCDEF',2);
substr
--------
BCDEF
(1 row)
```

from 为负数时：

```
postgres=# SELECT substr('ABCDEF',-2);
substr
--------
EF
(1 row)
```

- substr(string,from,count)

描述：

从参数 string 中抽取子字符串。from 表示抽取的起始位置。count 表示抽取的子字符串长度。from 为 0 时，按 1 处理。from 为正数时，抽取从 from 开始的 count 个字符。from 为负数时，抽取从倒数第 n 个开始的 count 个字符，n 为 from 的绝对值。count 小于 1 时，返回 null。

返回值类型：varchar

示例：

from 为正数时：

```
postgres=# SELECT substr('ABCDEF',2,2);
substr
--------
BC
(1 row)
```

from 为负数时：

```
postgres=# SELECT substr('ABCDEF',-3,2);
substr
```

```
--------
DE
(1 row)
```

- substrb(string,from)

  描述：该函数和 SUBSTR(string,from)函数功能一致，但是计算单位为字节。

  返回值类型：bytea

  示例：

```
postgres=# SELECT substrb('ABCDEF',-2);
substrb
---------
EF
(1 row)
```

- substrb(string,from,count)

  描述：该函数和 SUBSTR(string,from,count)函数功能一致，但是计算单位为字节。

  返回值类型：bytea 示例：

```
postgres=# SELECT substrb('ABCDEF',2,2);
substrb
---------
BC
(1 row)
```

- similar_escape(pat text, esc text)

  描述：将一个 SQL:2008 风格的正则表达式转换为 POSIX 风格。

  返回值类型：text

  示例：

```
postgres=# SELECT similar_escape('\s+ab','2');
similar_escape
----------------
^(?:\\s+ab)$
(1 row)
```

- svals(hstore)

  描述：获取 hstore 中的值。

返回值类型：SETOF text

示例：

```
postgres=# SELECT svals('"aa"=>"bb"');
svals
-------
bb
(1 row)
```

- tconvert(key text, value text)

描述：将字符串转换为 hstore 格式。

返回值类型：hstore

示例：

```
postgres=# SELECT tconvert('aa', 'bb');
tconvert
------------
"aa"=>"bb"
(1 row)
```

- to_hex(number int or bigint)

描述：把 number 转换成十六进制表现形式

。返回值类型：text

示例：

```
postgres=# SELECT to_hex(2147483647);
to_hex
----------
7fffffff
(1 row)
```

- translate(string text, from text, to text)

描述：把在 string 中包含的任何匹配 from 中字符的字符转化为对应的在 to 中的字符。如果 from 比 to 长，删掉在 from 中出现的额外的字符。

返回值类型：text

示例：

```
postgres=# SELECT translate('12345', '143', 'ax');
```

```
translate
-----------
a2x5
(1 row)
```

- trim([leading |trailing |both] [characters] from string)

描述：从字符串 string 的开头、结尾或两边删除只包含 characters 中字符（缺省是空白）的最长的字符串。

返回值类型：varchar

示例：

```
postgres=# SELECT trim(BOTH 'x' FROM 'xTomxx');
btrim
-------
Tom
(1 row)
postgres=# SELECT trim(LEADING 'x' FROM 'xTomxx');
ltrim
-------
Tomxx
(1 row)
postgres=# SELECT trim(TRAILING 'x' FROM 'xTomxx');
rtrim
-------
xTom
(1 row)
```

5.3.4 U-Z

- upper(string)

描述：把字符串转化为大写。

返回值类型：varchar

示例：

```
postgres=# SELECT upper('tom');
upper
-------
TOM
(1 row)
```

5.4 二进制字符串函数和操作符

5.4.1 字符串操作符

SQL 定义了一些字符串函数，在这些函数里使用关键字而不是逗号来分隔参数。

- octet_length(string)

  描述：二进制字符串中的字节数。返回值类型：int

  示例：

```
postgres=# SELECT octet_length(E'jo\\000se'::bytea) AS RESULT;
result
--------
      5
(1 row)
```

- overlay(string placing string from int [for int])

  描述：替换子串。返回值类型：bytea

  示例：

```
postgres=# SELECT overlay(E'Th\\000omas'::bytea placing E'\\002\\003'::bytea
from 2 for 3) AS RESULT;
result
----------------
\x5402036d6173
(1 row)
```

- position(substring in string) 描述：特定子字符串的位置。返回值类型：int

  示例：

```
postgres=# SELECT position(E'\\000om'::bytea in E'Th\\000omas'::bytea) AS
RESULT;
result
--------
3
(1 row)
```

- substring(string [from int] [for int])

  描述：截取子串。返回值类型：bytea

示例：

```
postgres=# SELECT substring(E'Th\\000omas'::bytea from 2 for 3) AS RESULT;
 result
----------
\x68006f
(1 row)
```

- substr(string, from int [, for int])

  描述：截取子串。返回值类型：bytea

  示例：

```
postgres=# SELECT substr(E'Th\\000omas'::bytea,2, 3) AS RESULT;
result
----------
\x68006f
(1 row)
```

- trim([both] bytes from string)

  描述：从 string 的开头和结尾删除只包含 bytes 中字节的最长字符串。返回值类型：bytea

  示例：

```
postgres=# SELECT trim(E'\\000'::bytea from E'\\000Tom\\000'::bytea) AS RESULT;
result
----------
\x546f6d
(1 row)
```

5.4.2 二进制字符串函数

GBase 8s 提供了常用的函数调用语法。

- btrim(string bytea,bytes bytea)

  描述：从 string 的开头和结尾删除只包含 bytes 中字节的最长的字符串。返回值类型：bytea

  示例：

```
postgres=# SELECT btrim(E'\\000trim\\000'::bytea, E'\\000'::bytea) AS RESULT;
result
------------
\x7472696d
```

```
(1 row)
```

- get_bit(string, offset)

  描述：从字符串中抽取位。返回值类型：int

  示例：

```
postgres=# SELECT get_bit(E'Th\\000omas'::bytea, 45) AS RESULT;
result
--------
1
(1 row)
```

- get_byte(string, offset)

  描述：从字符串中抽取字节。返回值类型：int

  示例：

```
postgres=# SELECT get_byte(E'Th\\000omas'::bytea, 4) AS RESULT;
result
--------
109
(1 row)
```

- rawcmp

  描述：raw 数据类型比较函数。参数：raw, raw

  返回值类型：integer

- raweq

  描述：raw 数据类型比较函数。参数：raw, raw

  返回值类型：boolean

- rawge

  描述：raw 数据类型比较函数。参数：raw, raw

  返回值类型：boolean

- rawgt

  描述：raw 数据类型比较函数。参数：raw, raw

返回值类型：boolean

- rawin

  描述：raw 数据类型解析函数。参数：cstring

  返回值类型：bytea

- rawle

  描述：raw 数据类型解析函数。参数：raw, raw

  返回值类型：boolean

- rawlike

  描述：raw 数据类型解析函数。参数：raw, raw

  返回值类型：boolean

- rawlt

  描述：raw 数据类型解析函数。参数：raw, raw

  返回值类型：boolean

- rawne

  描述：比较 raw 类型是否一样。参数：raw, raw

  返回值类型：boolean

- rawnlike

  描述：比较 raw 类型与模式是否不匹配。参数：raw, raw

  返回值类型：boolean

- rawout

  描述：RAW 类型的输出接口。参数：bytea

  返回值类型：cstring

- rawsend

  描述：转换 bytea 为二进制类型。参数：raw

  返回值类型：bytea

- rawtohex

  描述：raw 格式转换为十六进制。参数：text

  返回值类型：text

- set_bit(string,offset, newvalue)

  描述：设置字符串中的位。返回值类型：bytea

  示例：

```
postgres=# SELECT set_bit(E'Th\\000omas'::bytea, 45, 0) AS RESULT;
result
------------------
\x5468006f6d4173
(1 row)
```

- set_byte(string,offset, newvalue)

  描述：设置字符串中的字节。返回值类型：bytea

  示例：

```
postgres=# SELECT set_byte(E'Th\\000omas'::bytea, 4, 64) AS RESULT;
result
------------------
\x5468006f406173
(1 row)
```

5.5 位串函数和操作符

位串操作符

除了常用的比较操作符之外，还可以使用以下的操作符。&，|和#的位串操作数必须等长。在位移的时候，保留原始的位串长度（并以 0 填充）。

- ||

  描述：位串之间进行连接。示例：

```
postgres=# SELECT B'10001' || B'011' AS RESULT;
result
----------
10001011
(1 row)
```

说明

单字段内部连续连接操作不建议超过 180 次。如果超过 180 次，需拆分为多个连续连接的字符串，在它们之间再执行连接操作。例如：str1||str2||str3||str4 拆分为 (str1||str2)||(str3||str4)。

- &

描述：位串之间进行“与”操作。**示例**：

```
postgres=# SELECT B'10001' & B'01101' AS RESULT;
result
--------
00001
(1 row)
```

- |

描述：位串之间进行“或”操作。**示例**：

```
postgres=# SELECT B'10001' | B'01101' AS RESULT;
result
--------
11101
(1 row)
```

- #

描述：位串之间如果不一致进行“或”操作。如果两个位串中对应位置都为 1 或者则该位置返回为 0。**示例**：

```
postgres=# SELECT B'10001' # B'01101' AS RESULT;
result
--------
11100
(1 row)
```

- ~

描述：位串之间进行“非”操作。**示例**：

```
postgres=# SELECT ~B'10001'AS RESULT;
result
----------
01110
(1 row)
```

- <<

描述：位串进行左移操作。**示例**：

```
postgres=# SELECT B'10001' << 3 AS RESULT;
result
----------
01000
(1 row)
```

- >>

描述：位串进行右移操作。**示例**：

```
postgres=# SELECT B'10001' >> 2 AS RESULT;
result
----------
00100
(1 row)
```

下面的 SQL 标准函数除了可以用于字符串之外，也可以用于位串：length，bit_length，octet_length，position，substring，overlay。

下面的函数用于位串和二进制字符串：get_bit，set_bit。当用于位串时，这些函数位数从字符串的第一位（最左边）作为 0 位。

另外，可以在整数和 bit 之间来回转换。**示例**：

```
postgres=# SELECT 44::bit(10) AS RESULT;
result
------------
0000101100
(1 row)
postgres=# SELECT 44::bit(3) AS RESULT;
result
--------
100
(1 row)
postgres=# SELECT cast(-44 as bit(12)) AS RESULT;
result
--------------
111111010100
(1 row)
postgres=# SELECT '1110'::bit(4)::integer AS RESULT;
result
```

```
--------
14
(1 row)
postgres=# SELECT substring('10101111'::bit(8), 2);
substring
-----------
0101111
(1 row)
```

说明

只是转换为“bit”的意思是转换成 bit(1)，因此只会转换成整数的最低位。

5.6 模式匹配操作符

数据库提供了三种独立的实现模式匹配的方法：SQL LIKE 操作符、SIMILAR TO 操作符和 POSIX-风格的正则表达式。除了这些基本的操作符外，还有一些函数可用于提取或替换匹配子串并在匹配位置分离一个串。

5.6.1 LIKE

描述：判断字符串是否能匹配上 LIKE 后的模式字符串。如果字符串与提供的模式匹配，则 LIKE 表达式返回为真（NOT LIKE 表达式返回假），否则返回为假（NOT LIKE 表达式返回真）。

匹配规则：

a. 此操作符只有在它的模式匹配整个串的时候才能成功。如果要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。

b. 下划线 （_）代表（匹配）任何单个字符； 百分号（%）代表任意串的通配符。

c. 要匹配文本里的下划线或者百分号，在提供的模式里相应字符必须前导逃逸字符。逃逸字符的作用是禁用元字符的特殊含义，缺省的逃逸字符是反斜 线，也可以用 ESCAPE 子句指定一个不同的逃逸字符。

d. 要匹配逃逸字符本身，写两个逃逸字符。例如要写一个包含反斜线的模式常量，那你就要在 SQL 语句里写两个反斜线。

e. 关键字 ILIKE 可以替换 LIKE，区别是 LIKE 大小写敏感，而 ILIKE 大小写不敏感。

f. 操作符~~等效于 LIKE，操作符~~*等效于 ILIKE。

说明

- 参数 standard_conforming_strings 设置为 off 时，在文串常量中的任何反斜线需要被双写。因此，写一个匹配单个反斜线的模式，实际上在语句需要四个反斜线（可以用 ESCAPE 选择一个不同的逃逸字符，来避免这种情况，这样反斜线就不再是 LIKE 的特殊字符了。但仍然是字符文本分析器的特殊字符，所以仍需要写两个反斜线）。
- 在兼容 MYSQL 数据模式时，也可以通过写 ESCAPE "的方式不选择逃逸字符，这样可以有效地禁用逃逸机制。但是无法关闭下划线和百分号在模式中的特殊含义。

示例：

```
postgres=# SELECT 'abc' LIKE 'abc' AS RESULT;
result
-----------
t
(1 row)
postgres=# SELECT 'abc' LIKE 'a%' AS RESULT;
result
-----------
t
(1 row)
postgres=# SELECT 'abc' LIKE '_b_' AS RESULT;
result
-----------
t
(1 row)
postgres=# SELECT 'abc' LIKE 'c' AS RESULT;
result
-----------
f
(1 row)
```

5.6.2 SIMILAR TO

描述：SIMILAR TO 操作符根据自己的模式是否匹配给定串而返回真或者假。他和 LIKE 非常类似，只不过他使用 SQL 标准定义的正则表达式理解模式。

匹配规则：

a. 和 LIKE 一样，此操作符只有在它的模式匹配整个串的时候才能成功。如果要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。

b. 下划线 （_）代表（匹配）任何单个字符； 百分号（%）代表任意串的通配符。

c. SIMILAR TO 也支持下面这些从 POSIX 正则表达式借用的模式匹配元字符，如下表所示。

表 5-3 元字符及其含义

元字符	含义
|	表示选择（两个候选之一）
*	表示重复前面的项零次或更多次
+	表示重复前面的项一次或更多次
?	表示重复前面的项零次或一次
{m}	表示重复前面的项刚好 m 次
{m,}	表示重复前面的项 m 次或更多次
{m,n}	表示重复前面的项至少 m 次并且不超过 n 次
()	把多个项组合成一个逻辑项
[...]	声明一个字符类，就像 POSIX 正则表达式一样

d. 前导逃逸字符可以禁止所有这些元字符的特殊含义。逃逸字符的使用规则和 LIKE 一样。

正则表达式函数：

支持使用函数 substring(string from pa...截取匹配 SQL 正则表达式的子字符串。

示例：

```
postgres=# SELECT 'abc' SIMILAR TO 'abc' AS RESULT;
result
```

```
-----------
t
(1 row)
postgres=# SELECT 'abc' SIMILAR TO 'a' AS RESULT;
result
-----------
f
(1 row)
postgres=# SELECT 'abc' SIMILAR TO '%(b|d)%' AS RESULT;
result
-----------
t
(1 row)
postgres=# SELECT 'abc' SIMILAR TO '(b|c)%' AS RESULT;
result
-----------
f
(1 row)
```

5.6.3 POSIX 正则表达式

描述：正则表达式是一个字符序列，它是定义一个串集合（一个正则集）的缩写。如果一个串属于正则表达式描述的正则集时，则称该串匹配该正则表达式。POSIX 正则表达式提供了比 LIKE 和 SIMILAR TO 操作符更强大的含义。下表列出了所有可用于 POSIX 正则表达式模式匹配的操作符。

表 5-4 正则表达式匹配操作符

操作符	描述	例子
~	匹配正则表达式，大小写敏感	'thomas' ~ '.*thomas.*'
~*	匹配正则表达式，大小写不敏感	'thomas' ~* '.*Thomas.*'
!~	不匹配正则表达式，大小写敏感	'thomas' !~ '.*Thomas.*'
!~*	不匹配正则表达式，大小写不敏感	'thomas' !~* '.*vadim.*'

匹配规则：

a. 与 LIKE 不同，正则表达式允许匹配串里的任何位置，除非该正则表达式显式地挂接在串的开头或者结尾。

b. 除了上文提到的元字符外，POSIX 正则表达式还支持下列模式匹配元字符，如下表所示。

表 5-5 元字符及其含义

元字符	含义
^	表示串开头的匹配
$	表示串末尾的匹配
.	匹配任意单个字符

正则表达式函数：

POSIX 正则表达式支持下面函数。

- substring(string from pa...函数提供了抽取一个匹配 POSIX 正则表达式模式的子串的方法。
- regexp_count(string tex...函数提供了获取匹配 POSIX 正则表达式模式的子串数量的功能。
- regexp_instr(string tex...函数提供了获取匹配 POSIX 正则表达式模式子串位置的功能。
- regexp_substr(string te...函数提供了抽取一个匹配 POSIX 正则表达式模式的子串的方法。
- regexp_replace(string, p...函数提供了将匹配 POSIX 正则表达式模式的子串替换为新文本的功能。
- regexp_matches(string te...函数返回一个文本数组，该数组由匹配一个 POSIX 正则表达式模式得到的所有被捕获子串构成。
- regexp_split_to_table(st...函数把一个 POSIX 正则表达式模式当作一个定界符来分离

一个串。

- regexp_split_to_array(st...和 regexp_split_to_table 类似，是一个正则表达式分离函数，不过它的结果以一个 text 数组的形式返回

说明

正则表达式分离函数会忽略零长度的匹配，这种匹配发生在串的开头或结尾或者正好发生在前一个匹配之后。这和正则表达式匹配的严格定义是相悖的，后者由 regexp_matches 实现，但是通常前者是实际中最常用的行为。

示例

```
postgres=# SELECT 'abc' ~ 'Abc' AS RESULT;
result
--------
f
(1 row)
postgres=# SELECT 'abc' ~* 'Abc' AS RESULT;
result
--------
t
(1 row)
postgres=# SELECT 'abc' !~ 'Abc' AS RESULT;
result
--------
t
(1 row)
postgres=# SELECT 'abc'!~* 'Abc' AS RESULT;
result
--------
f
(1 row)
postgres=# SELECT 'abc' ~ '^a' AS RESULT;
result
--------
t
(1 row)
postgres=# SELECT 'abc' ~ '(b|d)'AS RESULT;
result
--------
t
```

```
(1 row)
postgres=# SELECT 'abc' ~ '^(b|c)'AS RESULT;
result
--------
f
(1 row)
```

虽然大部分的正则表达式搜索都能很快地执行，但是正则表达式仍可能被人为地弄成需要任意长的时间和任意量的内存进行处理。不建议从非安全模式来源接受正则表达式搜索模式，如果必须这样做，建议加上语句超时限制。使用 SIMILAR TO 模式的搜索具有同样的安全性危险， 因为 SIMILAR TO 提供了很多和 POSIX-风格正则表达式相同的能力。LIKE 搜索比其他两种选项简单得多，因此在接受非安全模式来源搜索时要更安全些。

5.7 数字操作函数和操作符

5.7.1 数字操作符

- \+ 加法运算

示例：

```
postgres=# SELECT 2+3 AS RESULT;
result
--------
5
(1 row)
```

- \- 减法运算

示例：

```
postgres=# SELECT 2-3 AS RESULT;
result
--------
-1
(1 row)
```

- * 乘法运算

示例：

```
postgres=# SELECT 2*3 AS RESULT;
result
--------
```

```
 6
(1 row)
```

- / 除法运算

  示例：

```
postgres=# SELECT 4/2 AS RESULT;
result
--------
2
(1 row)
postgres=# SELECT 4/3 AS RESULT;
result
------------------
1.33333333333333
(1 row)
```

- +/- 正/负

  示例：

```
postgres=# SELECT -2 AS RESULT;
result
--------
-2
(1 row)
```

- % 模（求余）运算

  示例：

```
postgres=# SELECT 5%4 AS RESULT;
result
--------
1
(1 row)
```

- @ 取绝对值

  示例：

```
postgres=# SELECT @ -5.0 AS RESULT;
result
--------
 5.0
```

```
(1 row)
```

- ^ 幂运算

  示例：

```
postgres=# SELECT 2.0^3.0 AS RESULT;
result
--------------------
 8.0000000000000000
(1 row)
```

- |/ 求平方根运算

  示例：

```
postgres=# SELECT |/ 25.0 AS RESULT;
result
--------
5
(1 row)
```

- ||/ 求立方根运算

  示例：

```
postgres=# SELECT ||/ 27.0 AS RESULT;
result
--------
 3
(1 row)
```

- ! 阶乘运算

  示例：

```
postgres=# SELECT 5! AS RESULT;
result
--------
 120
(1 row)
```

- !! 阶乘运算（带前缀操作符）

  示例：

```
postgres=# SELECT !!5 AS RESULT;
```

```
result
--------
 120
(1 row)
```

- & 二进制与运算 AND

  示例:

```
postgres=# SELECT 91&15 AS RESULT;
result
--------
 11
(1 row)
```

- | 二进制或运算 OR

  示例:

```
postgres=# SELECT 32|3 AS RESULT;
result
--------
35
(1 row)
```

- # 二进制异或运算 XOR

  示例:

```
postgres=# SELECT 17#5 AS RESULT;
result
--------
 20
(1 row)
```

- ~ 二进制非运算 NOT

  示例:

```
postgres=# SELECT ~1 AS RESULT;
result
--------
-2
(1 row)
```

- << 二进制左移

示例：

```
postgres=# SELECT 1<<4 AS RESULT;
result
--------
16
(1 row)
```

- >> 二进制右移

示例：

```
postgres=# SELECT 8>>2 AS RESULT;
result
--------
 2
(1 row)
```

5.7.2 数字操作函数

- abs(x)

描述：绝对值。

返回值类型：和输入相同。

示例：

```
postgres=# SELECT abs(-17.4);
 abs
------
 17.4
(1 row)
```

- acos(x)

描述：反余弦。

返回值类型：double precision

示例：

```
postgres=# SELECT acos(-1);
acos
------------------
 3.14159265358979
(1 row)
```

- asin(x)

  描述：反正弦。

  返回值类型：double precision

  示例：

```
postgres=# SELECT asin(0.5);
 asin
------------------
.523598775598299
(1 row)
```

- atan(x)

  描述：反正切。

  返回值类型：double precision

  示例：

```
postgres=# SELECT atan(1);
atan
------------------
.785398163397448
(1 row)
```

- atan2(y, x)

  描述：y/x 的反正切。

  返回值类型：double precision

  示例：

```
postgres=# SELECT atan2(2, 1);
 atan2
------------------
 1.10714871779409
(1 row)
```

- bitand(integer, integer)

  描述：计算两个数字与运算(&)的结果。

  返回值类型：bigint 类型数字。

示例：

```
postgres=# SELECT bitand(127, 63);
 bitand
--------
 63
(1 row)
```

- cbrt(dp)

  描述：立方根。

  返回值类型：double precision

  示例：

```
postgres=# SELECT cbrt(27.0);
cbrt
------
    3
(1 row)
```

- ceil(x)

  描述：不小于参数的最小的整数。

  返回值类型：整数。示例：

```
postgres=# SELECT ceil(-42.8);
 ceil
------
-42
(1 row)
```

- ceiling(dp or numeric)

  描述：不小于参数的最小整数（ceil 的别名）。

  返回值类型：dp or numeric，不考虑隐式类型转换的情况下与输入相同。示例：

```
postgres=# SELECT ceiling(-95.3);
ceiling
---------
-95
(1 row)
```

- cos(x)

描述：余弦。

返回值类型：double precision

示例：

```
postgres=# SELECT cos(-3.1415927);
cos
-------------------
-.999999999999999
(1 row)
```

- cot(x)

  描述：余切。

  返回值类型：double precision

  示例：

```
postgres=# SELECT cot(1);
cot
-----------------
.642092615934331
(1 row)
```

- degrees(dp)

  描述：把弧度转为角度。

  返回值类型：double precision

  示例：

```
postgres=# SELECT degrees(0.5);
degrees
-----------------
28.6478897565412
(1 row)
```

- div(y numeric, x numeric)

  描述：y 除以 x 的商的整数部分。

  返回值类型：numeric

  示例：

```
postgres=# SELECT div(9,4);
div
-----
2
(1 row)
```

- exp(x)

  描述：自然指数。

  返回值类型：dp or numeric，不考虑隐式类型转换的情况下与输入相同。

  示例：

```
postgres=# SELECT exp(1.0);
exp
--------------------
2.7182818284590452
(1 row)
```

- floor(x)

  描述：不大于参数的最大整数。

  返回值类型：与输入相同。

  示例：

```
postgres=# SELECT floor(-42.8);
floor
-------
-43
(1 row)
```

- int1(in)

  描述：将传入的 text 参数转换为 int1 类型值并返回。

  返回值类型：int1

  示例：

```
postgres=# SELECT int1('123');
int1
------
123
(1 row)
```

- int2(in)

描述：将传入参数转换为 int2 类型值并返回。支持的入参类型包括 float4，float8，int16，numeric，text。

返回值类型：int2

示例：

```
postgres=# SELECT int2('1234');
int2
------
1234
(1 row)
postgres=# SELECT int2(25.3);
int2
------
25
(1 row)
```

- int4(in)

描述：将传入参数转换为 int4 类型值并返回。支持的入参类型包括 bit，boolean，char，duoble precision，int16，numeric， real，smallint，text。

返回值类型：int4

示例：

```
postgres=# select int4('789');
int4
------
 789
(1 row)
postgres=# select int4(99.9);
int4
------
 100
(1 row)
```

- float4(in)

描述：将传入参数转换为 float4 类型值并返回。支持的入参类型包括：bigint，duoble precision，int16， integer, numeric，smallint，text 。

返回值类型：float4

示例：

```
postgres=# select float4('789');
float4
--------
789
(1 row)
postgres=# select float4(99.9);
float4
--------
 99.9
(1 row)
```

- float8(in)

描述：将传入参数转换为 float8 类型值并返回。支持的入参类型包括：bigint, int16, integer, numeric，real，smallint，text 。

返回值类型：float8

示例：

```
postgres=# select float8('789');
 float8
--------
 789
(1 row)
postgres=# select float8(99.9);
 float8
--------
 99.9
(1 row)
```

- int16(in)

描述：将传入参数转换为 int16 类型值并返回。支持的入参类型包括：bigint，boolean，double precision，integer，numeric，oid，real，smallint，tinyint 。

返回值类型：int16

示例：

```
postgres=# select int16('789');
 int16
```

```
--------
 789
(1 row)
postgres=# select int16(99.9);
 int16
--------
 100
(1 row)
```

- numeric(in)

描述：将传入参数转换为 numeric 类型值并返回。支持的入参类型包括：bigint，boolean，double precision，int16，integer，money，real，smallint 。

返回值类型：numeric

示例：

```
postgres=# select "numeric"('789');
numeric
---------
789
(1 row)
postgres=# select "numeric"(99.9);
numeric
---------
99.9
(1 row)
```

- oid(in)

描述：将传入参数转换为 oid 类型值并返回。支持的入参类型包括：bigint，int16。

返回值类型：oid

- radians(dp)

描述：把角度转为弧度。

返回值类型：double precision

示例：

```
postgres=# SELECT radians(45.0);
 radians
------------------
```

```
.785398163397448
(1 row)
```

- random()

  描述：0.0 到 1.0 之间的随机数。

  返回值类型：double precision

  示例：

```
postgres=# SELECT random();
 random
------------------
.824823560658842
(1 row)
```

- multiply(x double precision or text, y double precision or text)

  描述：x 和 y 的乘积。

  返回值类型：double precision

  示例：

```
postgres=# SELECT multiply(9.0, '3.0');
 multiply
------------------
27
(1 row)
postgres=# SELECT multiply('9.0', 3.0);
 multiply
------------------
27
(1 row)
```

- ln(x)

  描述：自然对数。

  返回值类型：dp or numeric，不考虑隐式类型转换的情况下与输入相同。

  示例：

```
postgres=# SELECT ln(2.0);
 ln
------------------
```

```
.6931471805599453
(1 row)
```

- log(x)

  描述：以 10 为底的对数。

  返回值类型：与输入相同。

  示例：

```
postgres=# SELECT log(100.0);
 log
--------------------
 2.0000000000000000
(1 row)
```

- log(b numeric, x numeric)

- 描述：以 b 为底的对数。

  返回值类型：numeric

  示例：

```
postgres=# SELECT log(2.0, 64.0);
 log
--------------------
 6.0000000000000000
(1 row)
```

- mod(x,y)

  描述：x/y 的余数（模）。如果 x 是 0，则返回 0。返回值类型：与参数类型相同。

  示例：

```
postgres=# SELECT mod(9,4);
 mod
-----
 1
(1 row)
postgres=# SELECT mod(9,0);
mod
-----
9
(1 row)
```

- pi()

  描述：“π”常量。

  返回值类型：double precision

  示例：

```
postgres=# SELECT pi();
 pi
------------------
 3.14159265358979
(1 row)
```

- power(a double precision, b double precision)

  描述：a 的 b 次幂。

  返回值类型：double precision

  示例：

```
postgres=# SELECT power(9.0, 3.0);
 power
----------------------
 729.0000000000000000
(1 row)
```

- round(x)

  描述：离输入参数最近的整数。

  返回值类型：与输入相同。

  示例：

```
postgres=# SELECT round(42.4);
 round
-------
 42
(1 row)
postgres=# SELECT round(42.6);
 round
-------
 43
(1 row)
```

- round(v numeric, s int)

  描述：保留小数点后 s 位，s 后一位进行四舍五入。

  返回值类型：numeric

  示例：

```
postgres=# SELECT round(42.4382, 2);
 round
-------
 42.44
(1 row)
```

- setseed(dp)

  描述：为随后的 random()调用设置种子(-1.0 到 1.0 之间，包含)。返回值类型：void

  示例：

```
postgres=# SELECT setseed(0.54823);
 setseed
---------


(1 row)
```

- sign(x)

  描述：输出此参数的符号。

  返回值类型：-1 表示负数，0 表示 0，1 表示正数。

  示例：

```
postgres=# SELECT sign(-8.4);
sign
------
-1
(1 row)
```

- sin(x)

  描述：正弦。

  返回值类型：double precision

  示例：

```
postgres=# SELECT sin(1.57079);
sin
------------------
.999999999979986
(1 row)
```

- sqrt(x)

  描述：平方根。

  返回值类型：dp or numeric，不考虑隐式类型转换的情况下与输入相同。

  示例：

```
postgres=# SELECT sqrt(2.0);
 sqrt
-------------------
 1.414213562373095
(1 row)
```

- tan(x)

  描述：正切。

  返回值类型：double precision

  示例：

```
postgres=# SELECT tan(20);
 tan
------------------
 2.23716094422474
(1 row)
```

- trunc(x)

  描述：截断（取整数部分）。

  返回值类型：与输入相同。

  示例：

```
postgres=# SELECT trunc(42.8);
 trunc
-------
 42
(1 row)
```

- trunc(v numeric, s int)

  描述：截断为 s 位小数。

  返回值类型：numeric

  示例：

```
postgres=# SELECT trunc(42.4382, 2);
 trunc
-------
 42.43
(1 row)
```

- smgrne(a smgr, b smgr)

  描述：比较两个 smgr 类型整数是否不相等。

  返回值类型：bool

- smgreq(a smgr, b smgr)

  描述：比较两个 smgr 类型整数是否相等。

  返回值类型：bool

- int1abs

  描述：返回 uint8 类型数据的绝对值。

  参数：tinyint

  返回值类型：tinyint

- int1and

  描述：返回两个 uint8 类型数据按位与的结果。

  参数：tinyint, tinyint

  返回值类型：tinyint

- int1cmp

  描述：返回两个 uint8 类型数据比较的结果，若第一个参数大，则返回 1；若第二个参数大，则返回-1；若相等，则返回 0。

  参数：tinyint, tinyint

返回值类型：integer

- int1div

  描述：返回两个 uint8 类型数据相除的结果，结果为 float8 类型。

  参数：tinyint, tinyint

  返回值类型：tinyint

- int1eq

  描述：比较两个 uint8 类型数据是否相等。

  参数：tinyint, tinyint

  返回值类型：boolean

- int1ge

  描述：判断两个 uint8 类型数据是否第一个参数大于等于第二个参数。

  参数：tinyint, tinyint

  返回值类型：boolean

- int1gt

  描述：无符号 1 字节整数做大于运算。

  参数：tinyint, tinyint

  返回值类型：boolean

- int1larger

  描述：无符号 1 字节整数求最大值。

  参数：tinyint, tinyint

  返回值类型：tinyint

- int1le

  描述：无符号 1 字节整数做小于等于运算。

  参数：tinyint, tinyint

  返回值类型：boolean

- int1lt

  描述：无符号 1 字节整数做小于运算。

  参数：tinyint, tinyint

  返回值类型：boolean

- int1smaller

  描述：无符号 1 字节整数求最小算。

  参数：tinyint, tinyint

  返回值类型：tinyint

- int1inc

  描述：无符号 1 字节整数加一。

  参数：tinyint

  返回值类型：tinyint

- int1mi

  描述：无符号一字节整数做差运算。

  参数：tinyint, tinyint

  返回值类型：tinyint

- int1mod

  描述：无符号一字节整数做取余运算。

  参数：tinyint, tinyint

  返回值类型：tinyint

- int1mul

  描述：无符号一字节整数做乘法运算。

  参数：tinyint, tinyint

  返回值类型：tinyint

- int1ne

描述：无符号一字节整数不等于运算。

参数：tinyint, tinyint

返回值类型：boolean

- int1pl

  描述：无符号一字节整数加法。

  参数：tinyint, tinyint

  返回值类型：tinyint

- int1um

  描述：无符号一字节数去相反数并返回有符号二字节整数。

  参数：tinyint

  返回值类型：smallint

- int1xor

  描述：无符号一字节整数异或操作。

  参数：tinyint, tinyint

  返回值类型：tinyint

- cash_div_int1

  描述：对 money 类型进行除法运算。

  参数：money, tinyint

  返回值类型：money

- cash_mul_int1

  描述：对 money 类型进行乘法运算。

  参数：money, tinyint

  返回值类型：money

- int1not

  描述：无符号一字节整数二进制位翻转。

参数：tinyint

返回值类型：tinyint

- int1or

描述：无符号一字节整数或运算。

参数：tinyint, tinyint

返回值类型：tinyint

- int1shl

描述：无符号一字节整数左移指定位数。

参数：tinyint, integer

返回值类型：tinyint

- width_bucket(op numeric, b1 numeric, b2 numeric, count int)

描述：返回一个桶，这个桶是在一个有 count 个桶，上界为 b1 下界为 b2 的等深柱 图中 operand 将被赋予的那个桶。

返回值类型：int

示例：

```
postgres=# SELECT width_bucket(5.35, 0.024, 10.06, 5);
width_bucket
--------------
3
(1 row)
```

- width_bucket(op dp, b1 dp, b2 dp, count int)

描述：返回一个桶，这个桶是在一个有 count 个桶，上界为 b1 下界为 b2 的等深柱 图中 operand 将被赋予的那个桶。

返回值类型：int

示例：

```
postgres=# SELECT width_bucket(5.35, 0.024, 10.06, 5);
width_bucket
--------------
3
```

```
(1 row)
```

5.8 时间和日期处理函数和操作符

5.8.1 时间日期操作符

在使用时间和日期操作符时，需要使用明确的类型前缀修饰对应的操作数，以确保数据库解析准确。

例如，没有明确数据类型，就会出现异常错误。

```
postgres=# SELECT date '2001-10-01' - '7' AS RESULT;
ERROR:  invalid input syntax for type timestamp: "7"
LINE 1: SELECT date '2001-10-01' - '7' AS RESULT
                                   ^

CONTEXT:  referenced column: result
```

表 5-6 时间和日期操作符

操作符	示例
+	postgres=# SELECT date '2001-9-28' + integer '7' AS RESULT; result --------------------- 2001-10-05 00:00:00 (1 row)
	postgres=# SELECT date '2001-09-28' + interval '1 hour' AS RESULT; result --------------------- 2001-09-28 01:00:00 (1 row)
	postgres=# SELECT date '2001-09-28' + time '03:00' AS RESULT; result --------------------- 2001-09-28 03:00:00 (1 row)
	postgres=# SELECT interval '1 day' + interval '1 hour' AS RESULT; result --------------------- 1 day 01:00:00 (1 row)
	postgres=# SELECT timestamp '2001-09-28 01:00' + interval '23

	hours' AS RESULT; result --- 2001-09-29 00:00:00 (1 row)
	postgres=# SELECT time '01:00' + interval '3 hours' AS RESULT; result --- 04:00:00 (1 row)
-	postgres=# SELECT date '2001-10-01' - date '2001-09-28' AS RESULT; result --- 3days (1 row)
	postgres=# SELECT date '2001-10-01' - integer '7' AS RESULT; result --- 2001-09-24 00:00:00 (1 row)
	postgres=# SELECT date '2001-09-28' - interval '1 hour' AS RESULT; result --- 2001-09-27 23:00:00 (1 row)
	postgres=# SELECT time '05:00' - time '03:00' AS RESULT; result --- 02:00:00 (1 row)
	postgres=# SELECT time '05:00' - interval '2 hours' AS RESULT; result --- 03:00:00 (1 row)
	postgres=# SELECT timestamp '2001-09-28 23:00' - interval '23 hours' AS RESULT; result --- 2001-09-28 00:00:00 (1 row)

	postgres=# SELECT interval '1 day' - interval '1 hour' AS RESULT; result ---------- 23:00:00 (1 row)
	postgres=# SELECT timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' AS RESULT; result ---------- 1 day 15:00:00 (1 row)
*	postgres=# SELECT 900 * interval '1 second' AS RESULT; result ---------- 00:15:00 (1 row)
	postgres=# SELECT 21 * interval '1 day' AS RESULT; result ---------- 21 days (1 row)
	postgres=# SELECT double precision '3.5' * interval '1 hour' AS RESULT; result ---------- 03:30:00 (1 row)
/	postgres=# SELECT interval '1 hour' / double precision '1.5' AS RESULT; result ---------- 00:40:00 (1 row)
	postgres=# SELECT time '05:00' - interval '2 hours' AS RESULT; result ---------- 03:00:00 (1 row)

5.8.2 时间/日期函数

- age(timestamp, timestamp)

描述：将两个参数相减，并以年、月、日作为返回值。若相减值为负，则函数返回亦为负。两个参数类型必须相同，可以都带 timezone，或都不带 timezone。

返回值类型：interval

示例：

```
postgres=# SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');
          age
-------------------------
 43 years 9 mons 27 days
(1 row)
```

- age(timestamp)

描述：当前时间和参数相减，入参可以带或者不带 timezone。

返回值类型：interval

示例：

```
postgres=# SELECT age(timestamp '1957-06-13');
          age
------------------------
 64 years 11 mons 4 days
(1 row)
```

- clock_timestamp()

描述：实时时钟的当前时间戳。

返回值类型：timestamp with time zone

示例：

```
postgres=# SELECT clock_timestamp();
        clock_timestamp
-------------------------------
 2022-05-17 16:34:14.629575+08
(1 row)
```

- current_date

描述：当前时间。

返回值类型：date

示例：

```
postgres=# SELECT current_date;
    date
------------
 2022-05-17
(1 row)
```

- current_time

描述：当前时间。

返回值类型：time with time zone

示例：

```
postgres=# SELECT current_time;
       timetz
--------------------
 16:35:00.099149+08
(1 row)
```

- current_timestamp

描述：当前日期及时间。

返回值类型：timestamp with time zone

示例：

```
postgres=# SELECT current_timestamp;
        pg_systimestamp
-------------------------------
 2022-05-17 16:35:08.018834+08
(1 row)
```

- date_part(text, timestamp)

描述：获取日期/时间值中子域的值，例如年或者小时的值。等效于 extract(field from timestamp)。

timestamp 类型：abstime、date、interval、reltime、time with time zone、time without time zone、timestamp with time zone、timestamp without time zone。

返回值类型：date

示例：

```
postgres=# SELECT date_part('hour', timestamp '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

- date_part(text, interval)

  描述：获取日期/时间值中子域的值。获取月份值时，如果月份值大于 12，则取与 12 的模。等效于 extract(field from timestamp)。

  返回值类型：double precision

  示例：

```
postgres=# SELECT date_part('month', interval '2 years 3 months');
date_part
-----------
3
(1 row)
```

- date_trunc(text, timestamp)

  描述：截取到参数 text 指定的精度。

  返回值类型：interval、timestamp with time zone、timestamp without time zone

  示例：

```
postgres=# SELECT date_trunc('hour', timestamp '2001-02-16 20:38:40');
 date_trunc
---------------------
 2001-02-16 20:00:00
(1 row)
```

- trunc(timestamp)

  描述：默认按天截取。示例：

```
postgres=# SELECT trunc(timestamp '2001-02-16 20:38:40');
trunc
---------------------
2001-02-16 00:00:00
```

```
(1 row)
```

- trunc(arg1, arg2)

  描述：截取到 arg2 指定的精度。

  arg1 类型：interval、timestamp with time zone、timestamp without time zone

  arg2 类型：text

  返回值类型：interval、timestamp with time zone、timestamp without time zone

  示例：

```
postgres=# SELECT trunc(timestamp '2001-02-16 20:38:40', 'hour');
trunc
---------------------
 2001-02-16 20:00:00
(1 row)
```

- daterange(arg1, arg2)

  描述：获取时间边界信息。arg1 和 arg2 的类型为 date。

  返回值类型：daterange

  示例：

```
postgres=# select daterange('2000-05-06','2000-08-08');
 daterange
------------------------
[2000-05-06,2000-08-08)
(1 row)
```

- daterange(arg1, arg2, text)

  描述：获取时间边界信息。arg1 和 arg2 的类型为 date，text 类型为 text。返回值类型：daterange

  示例：

```
postgres=# select daterange('2000-05-06','2000-08-08','[]');
 daterange
------------------------
[2000-05-06,2000-08-09)
(1 row)
```

- extract(field from timestamp)

描述：获取小时的值。

返回值类型：double precision

示例：

```
postgres=# SELECT extract(hour from timestamp '2001-02-16 20:38:40');
 date_part
-----------
20
(1 row)
```

- extract(field from interval)

  描述：获取月份的值。如果大于 12，则取与 12 的模。返回值类型：double precision

  示例：

```
postgres=# SELECT extract(month from interval '2 years 3 months');
 date_part
-----------
3
(1 row)
```

- isfinite(date)

  描述：测试是否为有效日期。返回值类型：Boolean

  示例：

```
postgres=# SELECT isfinite(date '2001-02-16');
 isfinite
----------
t
(1 row)
```

- isfinite(timestamp)

  描述：测试判断是否为有效时间。返回值类型：Boolean

  示例：

```
postgres=# SELECT isfinite(timestamp '2001-02-16 21:28:30');
isfinite
----------
t
(1 row)
```

- isfinite(interval)

  描述：测试是否为有效区间。返回值类型：Boolean

  示例：

```
postgres=# SELECT isfinite(interval '4 hours');
isfinite
----------
t
(1 row)
```

- justify_days(interval)

  描述：将时间间隔以月（30 天为一月）为单位。返回值类型：interval

  示例：

```
postgres=# SELECT justify_days(interval '35 days');
 justify_days
--------------
1 mon 5 days
(1 row)
```

- justify_hours(interval)

  描述：将时间间隔以天（24 小时为一天）为单位。返回值类型：interval

  示例：

```
postgres=# SELECT JUSTIFY_HOURS(INTERVAL '27 HOURS');
justify_hours
---------------
1 day 03:00:00
(1 row)
```

- justify_interval(interval)

  描述：结合 justify_days 和 justify_hours，调整 interval。 返回值类型：interval

  示例：

```
postgres=# SELECT JUSTIFY_INTERVAL(INTERVAL '1 MON -1 HOUR');
justify_interval
------------------
29 days 23:00:00
(1 row)
```

- localtime

  描述：当前时间。

  返回值类型：time

  示例：

```
postgres=# SELECT localtime AS RESULT;
     result
----------------
 16:39:21.245672
(1 row)
```

- localtimestamp

  描述：当前日期及时间。返回值类型：timestamp 示例：

```
postgres=# SELECT localtimestamp;
         timestamp
---------------------------
 2022-05-17 16:39:41.292909
(1 row)
```

- now()

  描述：当前日期及时间。

  返回值类型：timestamp with time zone

  示例：

```
postgres=# SELECT now();
              now
-------------------------------
 2022-05-17 16:47:22.123899+08
(1 row)
```

- timenow

  描述：当前日期及时间。

  返回值类型：timestamp with time zone

  示例：

```
postgres=# select timenow();
        timenow
```

```
-----------------------
 2022-05-17 16:55:05+08
(1 row)
```

- numtodsinterval(num, interval_unit)

描述：将数字转换为 interval 类型。num 为 numeric 类型数字，interval_unit 为固定格式字符串（'DAY' | 'HOUR' | 'MINUTE' | 'SECOND'）。

可以通过设置参数 IntervalStyle 为 a，兼容该函数 interval 输出格式。

示例：

```
postgres=# SELECT numtodsinterval(100, 'HOUR');
numtodsinterval
-----------------
100:00:00
(1 row)
postgres=# SET intervalstyle = a;
SET
postgres=# SELECT numtodsinterval(100, 'HOUR');
numtodsinterval
------------------------------
+000000004 04:00:00.000000000
(1 row)
```

- pg_sleep(seconds)

描述：服务器线程延迟时间，单位为秒。

返回值类型：void

示例：

```
postgres=# SELECT pg_sleep(10);
pg_sleep
----------
(1 row)
```

- statement_timestamp()

描述：当前日期及时间。

返回值类型：timestamp with time zone

示例：

```
postgres=# SELECT statement_timestamp();
     statement_timestamp
-------------------------------
 2022-05-17 16:56:15.024107+08
(1 row)
```

- sysdate

  描述：当前日期及时间。

  返回值类型：timestamp

  示例：

```
postgres=# SELECT sysdate;
       sysdate
---------------------
 2022-05-17 16:56:35
(1 row)
```

- timeofday()

  描述：当前日期及时间（像 clock_timestamp，但是返回时为 text）。

  返回值类型：text

  示例：

```
postgres=# SELECT timeofday();
              timeofday
-------------------------------------
 Tue May 17 16:56:58.381310 2022 CST
(1 row)
```

- transaction_timestamp()

  描述：当前日期及时间，与 current_timestamp 等效。

  返回值类型：timestamp with time zone

  示例：

```
postgres=# SELECT transaction_timestamp();
     transaction_timestamp
-------------------------------
 2022-05-17 16:57:15.156131+08
(1 row)
```

- add_months(d,n)

  描述：用于计算时间点 d 再加上 n 个月的时间。返回值类型：timestamp

  示例：

```
postgres=# SELECT add_months(to_date('2017-5-29', 'yyyy-mm-dd'), 11) FROM
sys_dummy;
     add_months
---------------------
 2018-04-29 00:00:00
(1 row)
```

- last_day(d)

  描述：用于计算时间点 d 当月最后一天的时间。返回值类型：timestamp

  示例：

```
postgres=# select last_day(to_date('2017-01-01', 'YYYY-MM-DD')) AS cal_result;
    cal_result
---------------------
2017-01-31 00:00:00
(1 row)
```

- next_day(x,y)

  描述：用于计算时间点 x 开始的下一个星期几（y）的时间。返回值类型：timestamp

  示例：

```
postgres=# select next_day(timestamp '2017-05-25 00:00:00','Sunday')AS
cal_result;
   cal_result
---------------------
2017-05-28 00:00:00
(1 row)
```

- tinterval(abstime, abstime)

  描述：用两个绝对时间创建时间间隔。

  返回值类型：tinterval

  示例：

```
postgres=# call tinterval(abstime 'May 10, 1947 23:59:12', abstime 'Mon May 1
00:30:30 1995');
                      tinterval
-----------------------------------------------------
 ["1947-05-10 23:59:12+09" "1995-05-01 00:30:30+08"]
(1 row)
```

- tintervalend(tinterval)

  描述：返回 tinteval 的结束时间。

  返回值类型：abstime

  示例：

```
postgres=# select tintervalend('["Sep 4, 1983 23:59:12" "Oct4, 1983
23:59:12"]');
      tintervalend
-----------------------
 1983-10-04 23:59:12+08
(1 row)
```

- tintervalrel(tinterval)

  描述：计算并返回 tinterval 的相对时间。

  返回值类型：reltime

  示例：

```
postgres=# select tintervalrel('["Sep 4, 1983 23:59:12" "Oct4, 1983
23:59:12"]');
tintervalrel
--------------
 0-1
(1 row)
```

- tz_offset

  描述：将时区别名转换为以 UTC 为标准的 OFFSET。

  返回值类型：varchar

  示例：

```
postgres=# select tz_offset('US/Eastern');
 tz_offset
```

```
-----------
 -04:00:00
(1 row)
```

- smalldatetime_ge

  描述：判断是否第一个参数大于等于第二个参数。

  参数：smalldatetime, smalldatetime

  返回值类型：boolean

- smalldatetime_cmp

  描述：对比 smalldatetime 是否相等。

  参数：smalldatetime,　smalldatetime

  返回值类型：integer

- smalldatetime_eq

  描述：对比 smalldatetime 是否相等。

  参数：smalldatetime,　smalldatetime

  返回值类型：boolean。

- smalldatetime_gt

  描述：判断是否第一个参数大于第二个参数。

  参数：smalldatetime, smalldatetime

  返回值类型：boolean

- smalldatetime_hash

  描述：计算 timestamp 对应的哈希值。

  参数：smalldatetime

  返回值类型：integer

- smalldatetime_in

  描述：输入 timestamp。

  参数：cstring, oid, integer

返回值类型：smalldatetime

- smalldatetime_larger

  描述：返回较大的 timestamp。

  参数：smalldatetime, smalldatetime

  返回值类型：smalldatetime

- smalldatetime_le

  描述：判断是否第一个参数小于等于第二个参数。

  参数：smalldatetime, smalldatetime

  返回值类型：boolean

- smalldatetime_lt

  描述：判断是否第一个参数小于第二个参数。

  参数：smalldatetime, smalldatetime

  返回值类型：boolean

- smalldatetime_ne

  描述：比较两个 timestamp 是否不相等。

  参数：smalldatetime, smalldatetime

  返回值类型：boolean

- smalldatetime_out

  描述：timestamp 转换为外部形式。

  参数：smalldatetime

  返回值类型：cstring

- smalldatetime_send

  描述：timestamp 转换为二进制格式。

  参数：smalldatetime

  返回值类型：bytea

- smalldatetime_smaller

  描述：返回较小的一个 smalldatetime。

  参数：smalldatetime, smalldatetime

  返回值类型：smalldatetime

- smalldatetime_to_abstime

  描述：smalldatetime 转换为 abstime。

  参数：smalldatetime

  返回值类型：abstime

- smalldatetime_to_time

  描述：smalldatetime 转换为 time。

  参数：smalldatetime

  返回值类型：time without time zone

- smalldatetime_to_timestamp

  描述：smalldatetime 转换为 timestamp。

  参数：smalldatetime

  返回值类型：timestamp without time zone

- smalldatetime_to_timestamptz

  描述：smalldatetime 转换为 timestamptz。

  参数：smalldatetime

  返回值类型：timestamp with time zone

- smalldatetime_to_varchar2

  描述：smalldatetime 转换为 varchar2。

  参数：smalldatetime

  返回值类型：character varying

  说明

获取当前时间有多种方式，请根据实际业务从场景选择合适的接口：

(1) 以下接口按照当前事务的开始时刻返回值：

```
CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision) LOCALTIME LOCALTIMESTAMP LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

其中 CURRENT_TIME 和 CURRENT_TIMESTAMP 传递带有时区的值；LOCALTIME 和 LOCALTIMESTAMP 传递的值不带时区。CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME 和 LOCALTIMESTAMP 可以有选择地接受一个精度参数，该精度导致结果的秒域被园整为指定小数位。如果没有精度参数，结果将被给予所能得到的全部精度。

因为这些函数全部都按照当前事务的开始时刻返回结果，所以它们的值在事务运行的整个期间内都不改变。我们认为这是一个特性：目的是为了允许一个事务在“当前”时间上有一致的概念，这样在同一个事务里的多个修改可以保持同样的时间戳。

(2) 以下接口返回当前语句开始时间：

```
transaction_timestamp() statement_timestamp() now()
```

其中 transaction_timestamp()等价于 CURRENT_TIMESTAMP，但是其命名清楚地反映了它的返回值。statement_timestamp()返回当前语句的开始时刻（更准确的说是收到客户端最后一条命令的时间）。statement_timestamp()和 transaction_timestamp()在一个事务的第一条命令期间返回值相同，但是在随后的命令中却不一定相同。now()等效于 transaction_timestamp()。

a. 以下接口返回函数被调用时的真实当前时间：

```
clock_timestamp() timeofday()
```

clock_timestamp()返回真正的当前时间，因此它的值甚至在同一条 SQL 命令中都会变化。timeofday()和 clock_timestamp()相似，timeofday()也返回真实的当前时间，但是它的结果是一个格式化的 text 串，而不是 timestamp with time zone 值。

下表显示了可以用于截断日期和时间值的模板。

表 5-7 用于日期/时间截断的模式

类别	模式	描述
微秒	MICROSECON	截断日期/时间，精确到微

	US	秒（000000 - 999999）
	USEC	
	USECOND	
毫秒	MILLISECON	截断日期/时间，精确到毫秒（000 - 999）
	MS	
	MSEC	
	MSECOND	
秒	S	截断日期/时间，精确到秒（00 - 59）
	SEC	
	SECOND	
分钟	M	截断日期/时间，精确到分钟（00 - 59）
	MI	
千年	MIL	截断日期/时间，精确到千年（本千年的第一天）
	MILLENNIA	
	MILLENNIUM	

5.8.3 TIMESTAMPDIFF

- TIMESTAMPDIFF(unit , timestamp_expr1, timestamp_expr2)

timestampdiff 函数是计算两个日期时间之间(timestamp_expr2-timestamp_expr1)的差值，并以 unit 形式返回结果。timestamp_expr1，timestamp_expr2 必须是一个 timestamp、timestamptz、date 类型的值表达式。unit 表示的是两个日期差的单位。

说明

仅在兼容 MY 类型时（即创建数据库时指定 dbcompatibility 'B'）有效，其他类型不支持

该函数。

- year ： 年份

```
postgres=# SELECT TIMESTAMPDIFF(YEAR, '2018-01-01', '2020-01-01');
timestamp_diff
---------------
2
(1 row)
```

- quarter ： 季度

```
postgres=# SELECT TIMESTAMPDIFF(QUARTER, '2018-01-01', '2020-01-01');
timestamp_diff
---------------
8
(1 row)
```

- month：月份

```
postgres=# SELECT TIMESTAMPDIFF(MONTH, '2018-01-01', '2020-01-01');
timestamp_diff
---------------
24
(1 row)
```

- week：星期

```
postgres=# SELECT TIMESTAMPDIFF(WEEK, '2018-01-01', '2020-01-01');
timestamp_diff
---------------
104
(1 row)
```

- day ： 天

```
postgres=# SELECT TIMESTAMPDIFF(DAY, '2018-01-01', '2020-01-01');
timestamp_diff
---------------
730
(1 row)
```

- hour ： 小时

```
postgres=# SELECT TIMESTAMPDIFF(HOUR, '2020-01-01 10:10:10', '2020-01-01
11:11:11');
```

```
timestamp_diff
---------------
1
(1 row)
```

- minute ： 分钟

```
postgres=# SELECT TIMESTAMPDIFF(MINUTE, '2020-01-01 10:10:10', '2020-01-01
11:11:11');
timestamp_diff
---------------
61
(1 row)
```

- second ： 秒

```
postgres=# SELECT TIMESTAMPDIFF(SECOND, '2020-01-01 10:10:10', '2020-01-01
11:11:11');
timestamp_diff
---------------
3661
(1 row)
```

- microseconds ： 秒域（百万级别），可包含小数。

```
postgres=# SELECT TIMESTAMPDIFF(MICROSECOND, '2020-01-01 10:10:10.000000',
'2020-01-01 10:10:10.111111');
timestamp_diff
---------------
111111
(1 row)
```

- timestamp_expr 含有时区

```
postgres=# SELECT TIMESTAMPDIFF(HOUR,'2020-05-01 10:10:10-01','2020-05-01
10:10:10-03');
timestamp_diff
---------------
2
(1 row)
```

5.8.4 EXTRACT

- EXTRACT(field FROM source)

extract 函数从日期或时间的数值里抽取子域，比如年、小时等。source 必须是一个 timestamp、time 或 interval 类型的值表达式（类型为 date 的表达式转换为 timestamp，因此也可以用）。field 是一个标识符或者字符串，它指定从源数据中抽取的域。extract 函数返回类型为 double precision 的数值。field 的取值范围如下所示。

- century

世纪。第一个世纪从 0001-01-01 00:00:00 AD 开始。这个定义适用于所有使用阳历的国家。没有 0 世纪，直接从公元前 1 世纪到公元 1 世纪。

示例：

```
postgres=# SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
date_part
-----------
20
(1 row)
```

- day
  - 如果 source 为 timestamp，表示月份里的日期（1-31）。

```
postgres=# SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
16
(1 row)
```

  - 如果 source 为 interval，表示天数。

```
postgres=# SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
 date_part
-----------
40
(1 row)
```

- decade

年份除以 10。

```
postgres=# SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
200
(1 row)
```

- dow

每周的星期几，星期天（0）到星期六（6）。

```
postgres=# SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
5
(1 row)
```

- doy

一年的第几天。取值范围为 1~365/366。

```
postgres=# SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
47
(1 row)
```

- epoch

如果 source 为 timestamp with time zone，表示自 1970-01-01 00:00:00-00 UTC 以来的秒数（结果可能是负数）；

如果 source 为 date 和 timestamp，表示自 1970-01-01 00:00:00-00 当地时间以来的秒数；

如果 source 为 interval，表示时间间隔的总秒数。

```
postgres=# SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16
20:38:40.12-08');
date_part
--------------
 982384720.12
(1 row)
postgres=# SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
date_part
-----------
 442800
(1 row)
```

将 epoch 值转换为时间戳的方法。

```
postgres=# SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * INTERVAL '1
second' AS RESULT;
result
```

```
--------------------------
2001-02-17 12:38:40.12+08
(1 row)
```

- hour

  小时域，取值范围为 0-23。

```
postgres=# SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
20
(1 row)
```

- isodow

  一周的第几天，取值范围为 1-7。1 表示星期一，7 表示星期天。

  说明

  除了星期天外，都与 dow 相同。

```
postgres=# SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
date_part
-----------
7
(1 row)
```

- isoyear

  日期中的 ISO 8601 标准年（不适用于间隔）。

  每个带有星期一开始的周中包含 1 月 4 日的 ISO 年，所以在年初的 1 月或 12 月下旬的 ISO 年可能会不同于阳历的年。详细信息请参见后续的 week 描述。

```
postgres=# SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
date_part
-----------
2005
(1 row)
postgres=# SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
date_part
-----------
2006
(1 row)
```

- microseconds

  秒域（百万级别），可包含小数。

```
postgres=# SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
date_part
-----------
28500000
(1 row)
```

- millennium

  第 n 个千年。

  20 世纪年份在第二个千年里。第三个千年从 2001 年 1 月 1 日零时开始算起，至今。

```
postgres=# SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
3
(1 row)
```

- milliseconds

  秒域（千秒级别），可包含小数。

```
postgres=# SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
date_part
-----------
28500
(1 row)
```

- minute

  分钟域。取值范围为 0-59。

```
postgres=# SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
38
(1 row)
```

- month

  如果 source 为 timestamp，表示一年里的月份数（1-12）。

```
postgres=# SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
```

```
date_part
-----------
2
(1 row)
```

如果 source 为 interval，表示月的数目，然后对 12 取模（0-11）。

```
postgres=# SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
date_part
-----------
1
(1 row)
```

- quarter

该天所在的该年的季度。取值范围为 1-4。

```
postgres=# SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
1
(1 row)
```

- second

秒域，可包含小数。取值范围为 0-59。

```
postgres=# SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
date_part
-----------
28.5
(1 row)
```

- timezone

与 UTC 的时区偏移量，单位为秒。正数对应 UTC 东边的时区，负数对应 UTC 西边的时区。

- timezone_hour

时区偏移量的小时部分。

- timezone_minute

时区偏移量的分钟部分。

- week

该天在所在的年份里是第几周。

ISO 8601 定义一年的第一周包含该年的一月四日（ISO-8601 的周从星期一开始）。换句话说，一年的第一个星期四在第一周。在 ISO 定义里，一月的头几天可能是前一年的第 52 或者第 53 周，十二月的后几天可能是下一年第一周。比如，2005-01-01 是 2004 年的第 53 周，而 2006-01-01 是 2005 年的第 52 周，2012-12-31 是 2013 年的第一周。建议 isoyear 字段和 week 一起 使用以得到一致的结果。

```
postgres=# SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
7
(1 row)
```

- year

年份域

```
postgres=# SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
2001
(1 row)
```

5.8.5 data_part

date_part 函数基于传统 Ingres 函数，等效于 SQL 标准函数 extract。

- date_part('field', source)

这里的 field 参数必须是一个字符串，而不是一个名称。有效的 field 与 extract 一样，详细信息请参见 EXTRACT。

示例：

```
postgres=# SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
date_part
-----------
16
(1 row)
postgres=# SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
date_part
-----------
4
```

```
(1 row)
```

下表显示了日期和时间值的规范格式。

表 5-8 日期/时间规范格式

类别	模式	描述
小时	HH	一天的小时数，取值范围为 01-12
	HH12	一天的小时数，取值范围为 01-12
	HH24	一天的小时数，取值范围为 00-23
分钟	MI	分钟，取值范围为 00-59
秒	SS	秒，取值范围为 00-59
	FF	微秒，取值范围为 000000-999999
	SSSSS	午夜后的秒，取值范围为 0-86399
上、下午	AM 或 A.M.	上午标识
	PM 或 P.M.	下午标识
年	Y,YYY	带逗号的年（4 位及以上）
	SYYYY	公元前四位年
	YYYY	年（4 位及以上）
	YYY	年的后三位
	YY	年的后两位
	Y	年的最后一位
	IYYY	ISO 年（4 位及以上）
	IYY	ISO 年的最后三位

	IY	ISO 年的最后两位
	I	ISO 年的最后一位
	RR	年的后两位（可在 21 世纪存储 20 世纪的年份）
	RRRR	可接收 2 位或 4 位。若是两位，则和 RR 的返回值相同，若是四位，则和 YYYY 相同。
	BC 或 B.C. AD 或 A.D.	纪元标识。BC（公元前），AD（公元后）。
月	MONTH	全长大写月份名（空白填充为 9 字符）
	MON	大写缩写月份名（3 字符）
	MM	月份数，取值范围为 01-12
	RM	罗马数字的月份（I-XII ；I=JAN）（大写）
天	DAY	全长大写日期名（空白填充为 9 字符）
	DY	缩写大写日期名（3 字符）
	DDD	一年里的日，取值范围为 001-366
	DD	一个月里的日，取值范围为 01-31
	D	一周里的日，取值范围为 1-7（从周日开始计算，1 表示周日）
周	W	一个月里的周数，取值范围为 1-5（第一周从该月第一天开始）
	WW	一年里的周数，取值范围为 1-53。（第一

		周从该年的第一天开始）
	IW	ISO 一年里的周数（第一个星期四在第一周里）
世纪	CC	世纪 (2 位）(21 世纪从 2001-01-01 开始）
儒略日	J	儒略日（自公元前 4712 年 1 月 1 日来的天数）
季度	Q	季度

说明

上表中 RR 计算年的规则如下：

- 输入的两位年份在 00~49 之间：

当前年份的后两位在 00~49 之间，返回值年份的前两位和当前年份的前两位相同； 当前年份的后两位在 50~99 之间，返回值年份的前两位是当前年份的前两位加 1。

- 输入的两位年份在 50~99 之间：

当前年份的后两位在 00~49 之间，返回值年份的前两位是当前年份的前两位减 1； 当前年份的后两位在 50~99 之间，返回值年份的前两位和当前年份的前两位相同。

5.9 类型转换函数

5.9.1 类型转换函数

- cash_words(money)

描述：类型转换函数，将 money 转换成 text。

示例：

```
postgres=# SELECT cash_words('1.23');
            cash_words
-----------------------------------
 One dollar and twenty three cents
(1 row)
```

- cast(x as y)

  描述：类型转换函数，将 x 转换成 y 指定的类型。

  示例：

```
postgres=# SELECT cast('22-oct-1997' as timestamp);
      timestamp
---------------------
 1997-10-22 00:00:00
(1 row)
```

- hextoraw(raw)

  描述：将一个十六进制构成的字符串转换为 raw 类型。

  返回值类型：raw

  示例：

```
postgres=# SELECT hextoraw('7D');
 hextoraw
----------
 7D
(1 row)
```

- numtoday(numeric)

  描述：将数字类型的值转换为指定格式的时间戳。

  返回值类型：timestamp

  示例：

```
postgres=# SELECT numtoday(2);
           numtoday
-------------------------------
 +000000002 00:00:00.000000000
(1 row)
```

- pg_systimestamp()

  描述：获取系统时间戳。

  返回值类型：timestamp with time zone

  示例：

```
postgres=# SELECT pg_systimestamp();
        pg_systimestamp
-------------------------------
 2022-05-17 17:07:51.267568+08
(1 row)
```

- rawtohex(string)

描述：将一个二进制构成的字符串转换为十六进制的字符串。结果为输入字符的 ACSII 码，以十六进制表示。

返回值类型：varchar

示例：

```
码 postgres=# SELECT rawtohex('1234567');
    rawtohex
----------------
 31323334353637
(1 row)
```

- to_bigint(varchar)

描述：将字符类型转换为 bigint 类型。

返回值类型：bigint

示例：

```
postgres=# SELECT to_bigint('123364545554455');
    to_bigint
---------------
 123364545554455
(1 row)
```

- to_char(datetime/interval [, fmt])

描述：将一个 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE 或者 TIMESTAMP WITH LOCAL TIME ZONE 类型的 DATETIME 或者 INTERVAL 值按照 fmt 指定的格式转换为 VARCHAR 类型。

可选参数 fmt 可以为以下几类：日期、时间、星期、季度和世纪。每类都可以有不同的模板，模板之间可以合理组合，常见的模板有：HH、MI、SS、YYYY、MM、DD。

模板可以有修饰词，常用的修饰词是 FM，可以用来抑制前导的零或尾随的空白。

返回值类型：varchar

示例：

```
postgres=# SELECT to_char(current_timestamp,'HH12:MI:SS');
 to_char
----------
 05:08:27
(1 row)
postgres=# SELECT to_char(current_timestamp,'FMHH12:FMMI:FMSS');
to_char
---------
 5:8:34
(1 row)
```

- to_char(double precision/real, text)

描述：将浮点类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
postgres=# SELECT to_char(125.8::real, '999D99');
 to_char
---------
  125.80
(1 row)
```

- to_char(numeric/smallint/integer/bigint/double precision/real[, fmt])

描述：将一个整型或者浮点类型的值转换为指定格式的字符串。

可选参数 fmt 可以为以下几类：十进制字符、“分组”符、正负号和货币符号，每类都可以有不同的模板，模板之间可以合理组合，常见的模板有：9、0、,（千分隔符）、.（小数点）。

模板可以有类似 FM 的修饰词，但 FM 不抑制由模板 0 指定而输出的 0。

要将整型类型的值转换成对应 16 进制值的字符串，使用模板 X 或 x。

返回值类型：varchar

示例：

```
postgres=# SELECT to_char(1485,'9,999');
 to_char
```

```
---------
  1,485
(1 row)
postgres=# SELECT to_char( 1148.5,'9,999.999');
  to_char
------------
  1,148.500
(1 row)
postgres=# SELECT to_char(148.5,'990999.909');
   to_char
-------------
    0148.500
(1 row)
postgres=# SELECT to_char(123,'XXX');
 to_char
---------
   7B
(1 row)
```

- to_char(interval, text)

  描述：将时间间隔类型的值转换为指定格式的字符串。

  返回值类型：text

  示例：

```
postgres=# SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');
 to_char
----------
 15:02:12
(1 row)
```

- to_char(int, text)

  描述：将整数类型的值转换为指定格式的字符串。

  返回值类型：text

  示例：

```
postgres=# SELECT to_char(125, '999');
 to_char
---------
  125
```

```
(1 row)
```

- to_char(numeric, text)

  描述：将数字类型的值转换为指定格式的字符串。

  返回值类型：text

  示例：

```
postgres=# SELECT to_char(-125.8, '999D99S');
 to_char
---------
 125.80-
(1 row)
```

- to_char(string)

  描述：将 CHAR、VARCHAR、VARCHAR2、CLOB 类型转换为 VARCHAR 类型。

  如使用该函数对 CLOB 类型进行转换，且待转换 CLOB 类型的值超出目标类型的范围，则返回错误。

  返回值类型：varchar

  示例：

```
postgres=# SELECT to_char('01110');
 to_char
---------
 01110
(1 row)
```

- to_char(timestamp, text)

  描述：将时间戳类型的值转换为指定格式的字符串。

  返回值类型：text

  示例：

```
postgres=# SELECT to_char(current_timestamp, 'HH12:MI:SS');
 to_char
----------
 10:55:59
(1 row)
```

- to_clob(char/nchar/varchar/varchar2/nvarchar/nvarchar2/text/raw)

描述：将 RAW 类型或者文本字符集类型 CHAR、NCHAR、VARCHAR、VARCHAR2、NVARCHAR、NVARCHAR2、TEXT 转成 CLOB 类型。

返回值类型：clob

示例：

```
postgres=# SELECT to_clob('ABCDEF'::RAW(10));
 to_clob
---------
 ABCDEF
(1 row)
 postgres=# SELECT to_clob('hello111'::CHAR(15));
 to_clob
----------
 hello111
(1 row)
postgres=# SELECT to_clob('gauss123'::NCHAR(10));
 to_clob
----------
 gauss123
(1 row)
postgres=# SELECT to_clob('gauss234'::VARCHAR(10));
 to_clob
----------
 gauss234
(1 row)
postgres=# SELECT to_clob('gauss345'::VARCHAR2(10));
 to_clob
----------
 gauss345
(1 row)
postgres=# SELECT to_clob('gauss456'::NVARCHAR2(10));
 to_clob
----------
 gauss456
(1 row)
postgres=# SELECT to_clob('World222!'::TEXT);
  to_clob
-----------
 World222!
(1 row)
```

- to_date(text)

  描述：将文本类型的值转换为指定格式的时间戳。目前只支持两类格式。

  - 格式一：无分隔符日期，如 20150814，需要包括完整的年月日。
  - 格式二：带分隔符日期，如 2014-08-14，分隔符可以是单个任意非数字字符。

  返回值类型：timestamp without time zone

  示例：

```
postgres=# SELECT to_date('2015-08-14');
       to_date
---------------------
 2015-08-14 00:00:00
(1 row)
```

- to_date(text, text)

  描述：将字符串类型的值转换为指定格式的日期。

  返回值类型：timestamp without time zone

  示例：

```
postgres=# SELECT to_date('05 Dec 2000', 'DD Mon YYYY');
       to_date
---------------------
 2000-12-05 00:00:00
(1 row)
```

- to_number (expr [, fmt])

  描述：将 expr 按指定格式转换为一个 NUMBER 类型的值。

  类型转换格式请参考表 5-8。

  - 转换十六进制字符串为十进制数字时，最多支持 16 个字节的十六进制字符串转换为无符号数。
  - 转换十六进制字符串为十进制数字时，格式字符串中不允许出现除'x'或'X'以外的其他字符，否则报错。

  返回值类型：number

示例：

```
postgres=# SELECT to_number('12,454.8-', '99G999D9S');
 to_number
-----------
  -12454.8
(1 row)
```

- to_number(text, text)

  描述：将字符串类型的值转换为指定格式的数字。

  返回值类型：numeric

  示例：

```
postgres=# SELECT to_number('12,454.8-', '99G999D9S');
 to_number
-----------
  -12454.8
(1 row)
```

- to_timestamp(double precision)

  描述：把 UNIX 纪元转换成时间戳。

  返回值类型：timestamp with time zone

  示例：

```
postgres=# SELECT to_timestamp(1284352323);
      to_timestamp
------------------------
 2010-09-13 12:32:03+08
(1 row)
```

- to_timestamp(string [,fmt])

  描述：将字符串 string 按 fmt 指定的格式转换成时间戳类型的值。不指定 fmt 时，按参数 nls_timestamp_format 所指定的格式转换。

  如果输入的年份 YYYY=0，系统报错。如果输入的年份 YYYY<0，在 fmt 中指定 SYYYY，则正确输出公元前绝对值 n 的年份。fmt 中出现的字符必须与日期/时间格式化的模式相匹配，否则报错。

  返回值类型：timestamp without time zone

示例：

```
postgres=# SHOW nls_timestamp_format;
    nls_timestamp_format
-----------------------------
 DD-Mon-YYYY HH:MI:SS.FF AM
(1 row)

postgres=# SELECT to_timestamp('12-sep-2014');
    to_timestamp
---------------------
 2014-09-12 00:00:00
(1 row)
postgres=# SELECT to_timestamp('12-Sep-10 14:10:10.123000','DD-Mon-YY
HH24:MI:SS.FF');
      to_timestamp
-------------------------
 2010-09-12 14:10:10.123
(1 row)
postgres=# SELECT to_timestamp('-1','SYYYY');
      to_timestamp
-----------------------
 0001-01-01 00:00:00 BC
(1 row)
postgres=# SELECT to_timestamp('98','RR');
    to_timestamp
---------------------
 1998-01-01 00:00:00
(1 row)
postgres=# SELECT to_timestamp('01','RR');
    to_timestamp
---------------------
 2001-01-01 00:00:00
(1 row)
```

- to_timestamp(text, text)

  描述：将字符串类型的值转换为指定格式的时间戳。

  返回值类型：timestamp

  示例：

```
postgres=# SELECT to_timestamp('05 Dec 2000', 'DD Mon YYYY');
```

```
    to_timestamp
---------------------
 2000-12-05 00:00:00
(1 row)
```

表 5-9 数值格式化的模版模式

模式	描述
9	带有指定数值位数的值
0	带前导零的值
.（句点）	小数点
,（逗号）	分组（千）分隔符
PR	尖括号内负值
S	带符号的数值（使用区域设置）
L	货币符号（使用区域设置）
D	小数点（使用区域设置）
G	分组分隔符（使用区域设置）
MI	在指明的位置的负号（如果数字 <0）
PL	在指明的位置的正号（如果数字 >0）
SG	在指明的位置的正/负号
RN	罗马数字（输入在 1 和 3999 之间）
TH 或 th	序数后缀
V	移动指定位（小数）

- abstime_text

描述：将 abstime 类型转为 text 类型输出。

参数：abstime

返回值类型：text

- abstime_to_smalldatetime

  描述：将 abstime 类型转为 smalldatatime 类型。

  参数：abstime

  返回值类型：smalldatetime

- bigint_tid

  描述：将 bigint 转为 tid。

  参数：bigint

  返回值类型：tid

- bool_int1

  描述：将 bool 转为 int1。

  参数：boolean

  返回值类型：tinyint

- bool_int2

  描述：将 bool 转为 int2。

  参数：boolean

  返回值类型：smallint

- bool_int8

  描述：将 bool 转为 int8。

  参数：boolean

  返回值类型：bigint

- bpchar_date

  描述：将字符串转为日期。

参数：character

返回值类型：date

- bpchar_float4

  描述：将字符串转为 float4。

  参数：character

  返回值类型：real

- bpchar_float8

  描述：将字符串转为 float8。

  参数：character

  返回值类型：double precision

- bpchar_int4

  描述：将字符串转为 int4。

  参数：character

  返回值类型：integer

- bpchar_int8

  描述：将字符串转为 int8。

  参数：character

  返回值类型：bigint

- bpchar_numeric

  描述：将字符串转为 numeric。

  参数：character

  返回值类型：numeric

- bpchar_timestamp

  描述：将字符串转为时间戳。

  参数：character

返回值类型：timestamp without time zone

- bpchar_to_smalldatetime

  描述：将字符串转为 smalldatetime。

  参数：character

  返回值类型：smalldatetime

- cupointer_bigint

  描述：将列存 CU 指针类型转为 bigint 类型。

  参数：text

  返回值类型：bigint

- date_bpchar

  描述：将 date 类型转换为 bpchar 类型。

  参数：date

  返回值类型：character

- date_text

  描述：将 date 类型转换为 text 类型。

  参数：date

  返回值类型：text

- date_varchar

  描述：将 date 类型转换为 varchar 类型。

  参数：date

  返回值类型：character varying

- f4toi1

  描述：把 float4 类型强转为 uint8 类型。

  参数：real

  返回值类型：tinyint

- f8toi1

  描述：把 float8 类型强转为 uint8 类型。

  参数：double precision

  返回值类型：tinyint

- float4_bpchar

  描述：float4 转换为 bpchar。

  参数：real

  返回值类型：character

- float4_text

  描述：float4 转换为 text。

  参数：real

  返回值类型：text

- float4_varchar

  描述：float4 转换为 varchar。

  参数：real

  返回值类型：character varying

- float8_bpchar

  描述：float8 转换为 bpchar。

  参数：double precision

  返回值类型：character

- float8_interval

  描述：float8 转换为 interval。

  参数：double precision

  返回值类型：interval

- float8_text

描述：float8 转换为 text。

参数：double precision

返回值类型：text

- float8_varchar

  描述：float8 转换为 varchar。

  参数：double precision

  返回值类型：character varying

- i1tof4

  描述：uint8 转换为 float4。

  参数：tinyint

  返回值类型：real

- i1tof8

  描述：uint8 转换为 float8。

  参数：tinyint

  返回值类型：double precision

- i1toi2

  描述：uint8 转换为 int16。

  参数：tinyint

  返回值类型：smallint

- i1toi4

  描述：uint8 转换为 int32。

  参数：tinyint

  返回值类型：integer

- i1toi8

  描述：uint8 转换为 int64。

参数：tinyint

返回值类型：bigint

- i2toi1

  描述：int16 转换为 uint8。

  参数：smallint

  返回值类型：tinyint

- i4toi1

  描述：int32 转换为 uint8。

  参数：integer

  返回值类型：tinyint

- i8toi1

  描述：int64 转换为 uint8。

  参数：bigint

  返回值类型：tinyint

- int1_avg_accum

  描述：将第二个 uint8 类型参数，加入到第一个参数中，一个参数为 bigint 类型数组。

  参数：bigint[], tinyint

  返回值类型：bigint[]

- int1_bool

  描述：uint8 转换为 bool。

  参数：tinyint

  返回值类型：boolean

- int1_bpchar

  描述：uint8 转换为 bpchar。

  参数：tinyint

返回值类型：character

- int1_mul_cash

  描述：返回一个 int8 类型参数和一个 cash 类型参数的乘积，返回值为 cash 类型。

  参数：tinyint, money

  返回值类型：money

- int1_numeric

  描述：uint8 转换为 numeric。

  参数：tinyint

  返回值类型：numeric

- int1_nvarchar2

  描述：uint8 转换为 nvarchar2。

  参数：tinyint

  返回值类型：nvarchar2

- int1_text

  描述：uint8 转换为 text。

  参数：tinyint

  返回值类型：text

- int1_varchar

  描述：uint8 转换为 varchar。

  参数：tinyint

  返回值类型：character varying

- int1in

  描述：字符串转化为无符号一字节整数。

  参数：cstring

  返回值类型：tinyint

- int1out

  描述：无符号一字节整数转化为字符串。

  参数：tinyint

  返回值类型：cstring

- int1up

  描述：输入整数转化为无符号一字节整数。

  参数：tinyint

  返回值类型：tinyint

- int2_bool

  描述：将有符号二字节整数转化为 bool 型。

  参数：smallint

  返回值类型：boolean

- int2_bpchar

  描述：将有符号二字节整数转化为 BpChar。

  参数：smallint

  返回值类型：character

- int2_text

  描述：有符号二字节整数转化为 text 类型。

  参数：smallint

  返回值类型：text

- int2_varchar

  描述：有符号二字节整数转化为 varchar 类型。

  参数：smallint

  返回值类型：character varying

- int8_text

描述：有符号八字节整数转化为 text 类型。

参数：bigint

返回值类型：text

- int8_varchar

  描述：有符号八字节整数转化为 varchar。

  参数：bigint

  返回值类型：character varying

- intervaltonum

  描述：将内部数据类型日期转化为 numeric 类型。

  参数：interval

  返回值类型：numeric

- numeric_bpchar

  描述：numeric 转化为 bpchar。

  参数：numeric

  返回值类型：character

- numeric_int1

  描述：numeric 转化为有符号 1 字节整数。

  参数：numeric

  返回值类型：tinyint

- numeric_text

  描述：numeric 转化为 text。

  参数：numeric

  返回值类型：text

- numeric_varchar

  描述：numeric 转化为 varchar。

参数：numeric

返回值类型：character varying

- nvarchar2in

  描述：将 c 字符串转化为 varchar。

  参数：cstring, oid, integer

  返回值类型：nvarchar2

- nvarchar2out

  描述：将 text 转化为 c 字符串。

  参数：nvarchar2

  返回值类型：cstring

- nvarchar2send

  描述：将 varchar 转化为二进制。

  参数：nvarchar2

  返回值类型：bytea

- oidvectorin_extend

  描述：将字符串转化为 oidvector。

  参数：cstring

  返回值类型：oidvector_extend

- oidvectorout_extend

  描述：将 oidvector 转化为字符串。

  参数：oidvector_extend

  返回值类型：cstring

- oidvectorsend_extend

  描述：将 oidvector 转化为字符串。

  参数：oidvector_extend

返回值类型：bytea

- reltime_text

  描述：reltime 转换为 text。

  参数：reltime

  返回值类型：text

- text_date

  描述：text 类型转换为 date 类型。

  参数：text

  返回值类型：date

- text_float4

  描述：text 类型转换为 float4 类型。

  参数：text

  返回值类型：real

- text_float8

  描述：text 类型转换为 float8 类型。

  参数：text

  返回值类型：double precision

- text_int1

  描述：text 类型转换为 int1 类型。

  参数：text

  返回值类型：tinyint

- text_int2

  描述：text 类型转换为 int2 类型。

  参数：text

  返回值类型：smallint

- text_int4

  描述：text 类型转换为 int4 类型。

  参数：text

  返回值类型：integer

- text_int8

  描述：text 类型转换为 int8 类型。

  参数：text

  返回值类型：bigint

- text_numeric

  描述：text 类型转换为 numeric 类型。

  参数：text

  返回值类型：numeric

- text_timestamp

  描述：text 类型转换为 timestamp 类型。

  参数：text

  返回值类型：timestamp without time zone

- time_text

  描述：time 类型转换为 text 类型。

  参数：time without time zone

  返回值类型：text

- timestamp_text

  描述：timestamp 类型转换为 text 类型。

  参数：timestamp without time zone

  返回值类型：text

- timestamp_to_smalldatetime

描述：timestamp 类型转换为 smalldatetime 类型。

参数：timestamp without time zone

返回值类型：smalldatetime

- timestamp_varchar

  描述：timestamp 类型转换为 varchar 类型。

  参数：timestamp without time zone

  返回值类型：character varying

- timestamptz_to_smalldatetime

  描述：timestamptz 类型转换为 smalldatetime。

  参数：timestamp with time zone

  返回值类型：smalldatetime

- timestampzone_text

  描述：timestampzone 类型转换为 text 类型。

  参数：timestamp with time zone

  返回值类型：text

- timetz_text

  描述：timetz 类型转换为 text 类型。

  参数：time with time zone

  返回值类型：text

- to_integer

  描述：转换为 integer 类型。

  参数：character varying

  返回值类型：integer

- to_interval

  描述：转换为 interval 类型。

参数：character varying

返回值类型：interval

- to_numeric

  描述：转换为 numeric 类型。

  参数：character varying

  返回值类型：numeric

- to_nvarchar2

  描述：转换为 nvarchar2 类型。

  参数：numeric

  返回值类型：nvarchar2

- to_text

  描述：转换为 text 类型。

  参数：smallint

  返回值类型：text

- to_ts

  描述：转换为 ts 类型。

  参数：character varying

  返回值类型：timestamp without time zone

- to_varchar2

  描述：转换为 varchar2 类型。

  参数：timestamp without time zone

  返回值类型：character varying

- varchar_date

  描述：varchar 类型转换为 date。

  参数：character varying

返回值类型：date

- varchar_float4

  描述：varchar 类型转换为 float4。

  参数：character varying

  返回值类型：real

- varchar_float8

  描述：varchar 类型转换为 float8。

  参数：character varying

  返回值类型：double precision

- varchar_int4

  描述：varchar 类型转换为 int4。

  参数：character varying

  返回值类型：integer

- varchar_int8

  描述：varchar 类型转换为 int8。

  参数：character varying

  返回值类型：bigint

- varchar_numeric

  描述：varchar 类型转换为 numeric。

  参数：character varying

  返回值类型：numeric

- varchar_timestamp

  描述：varchar 类型转换为 timestamp。

  参数：character varying

  返回值类型：timestamp without time zone

- varchar2_to_smlldatetime

  描述：varchar2 类型转换为 smlldatetime。

  参数：character varying

  返回值类型：smalldatetime

- xidout4

  描述：xid 输出为 4 字节数字。

  参数：xid32

  返回值类型：cstring

- xidsend4

  描述：xid 转换为二进制格式。

  参数：xid32

  返回值类型：bytea

5.9.2 编码类型转换

- convert_to_nocase(text, text)

  描述：将字符串转换为指定的编码类型。

  返回值类型：bytea

  示例：

```
postgres=# SELECT convert_to_nocase('12345', 'GBK');
 convert_to_nocase
-------------------
 \x3132333435
(1 row)
```

5.10 几何函数和操作符

5.10.1 几何操作符

- +

描述：平移。

示例：

```
postgres=# SELECT box '((0,0),(1,1))' + point '(2.0,0)' AS RESULT;
   result
-------------
 (3,1),(2,0)
(1 row)
```

- -

描述：平移。

示例：

```
postgres=# SELECT box '((0,0),(1,1))' - point '(2.0,0)' AS RESULT;
    result
---------------
 (-1,1),(-2,0)
(1 row)
```

- *

描述：伸展/旋转。

示例：

```
postgres=# SELECT box '((0,0),(1,1))' * point '(2.0,0)' AS RESULT;
   result
-------------
 (2,2),(0,0)
(1 row)
```

- /

描述：收缩/旋转。

示例：

```
 postgres=# SELECT box '((0,0),(2,2))' / point '(2.0,0)' AS RESULT;
   result
-------------
 (1,1),(0,0)
(1 row)
```

- #

描述：两个图形交面。

示例：

```
postgres=# SELECT box '((1,-1),(-1,1))' # box '((1,1),(-2,-2))' AS RESULT;
 result
---------------
 (1,1),(-1,-1)
(1 row)
```

- #

描述：图形的路径数目或多边形顶点数。

示例：

```
postgres=# SELECT # path'((1,0),(0,1),(-1,0))' AS RESULT;
 result
--------
      3
(1 row)
```

- @-@

描述：图形的长度或者周长。

示例：

```
 postgres=# SELECT @-@ path '((0,0),(1,0))' AS RESULT;
 result
--------
      2
(1 row)
```

- @@

描述：图形的中心。

示例：

```
postgres=# SELECT @@ circle '((0,0),10)' AS RESULT;
 result
--------
 (0,0)
(1 row)
```

- <->

描述：两个图形之间的距离。

示例：

```
postgres=# SELECT circle '((0,0),1)' <-> circle '((5,0),1)' AS RESULT;
 result
--------
      3
(1 row)
```

- &&

描述：两个图形是否重叠（有一个共同点就为真）。

示例：

```
postgres=# SELECT box '((0,0),(1,1))' && box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

- <<

描述：图形是否全部在另一个图形的左边（没有相同的横坐标）。

示例：

```
postgres=# SELECT circle '((0,0),1)' << circle '((5,0),1)' AS RESULT;
 result
--------
 t
(1 row)
```

- >>

描述：图形是否全部在另一个图形的右边（没有相同的横坐标）。

示例：

```
postgres=# SELECT circle '((5,0),1)' >> circle '((0,0),1)' AS RESULT;
 result
--------
 t
(1 row)
```

- &<

描述：图形的最右边是否不超过在另一个图形的最右边。

示例：

```
postgres=# SELECT box '((0,0),(1,1))' &< box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

- &>

描述：图形的最左边是否不超过在另一个图形的最左边。

示例：

```
postgres=# SELECT box '((0,0),(3,3))' &> box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

- <<|

描述：图形是否全部在另一个图形的下边（没有相同的纵坐标）。

示例：

```
postgres=# SELECT box '((0,0),(3,3))' <<| box '((3,4),(5,5))' AS RESULT;
 result
--------
 t
(1 row)
```

- |>>

描述：图形是否全部在另一个图形的上边（没有相同的纵坐标）。

示例：

```
postgres=# SELECT box '((3,4),(5,5))' |>> box '((0,0),(3,3))' AS RESULT;
 result
--------
 t
(1 row)
```

- &<|

描述：图形的最上边是否不超过另一个图形的最上边。

示例：

```
postgres=# SELECT box '((0,0),(1,1))' &<| box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

- |&>

描述：图形的最下边是否不超过另一个图形的最下边。

示例：

```
 postgres=# SELECT box '((0,0),(3,3))' |&> box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

- <^

描述：图形是否低于另一个图形（允许两个图形有接触）。

示例：

```
postgres=# SELECT box '((0,0),(-3,-3))' <^ box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

- >^

描述：图形是否高于另一个图形（允许两个图形有接触）。

示例：

```
postgres=# SELECT box '((0,0),(2,2))' >^ box '((0,0),(-3,-3))'  AS RESULT;
 result
--------
 t
(1 row)
```

- ?#

描述：两个图形是否相交。

示例：

```
postgres=# SELECT lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

- ?-

描述：图形是否处于水平位置。

示例：

```
postgres=# SELECT ?- lseg '((-1,0),(1,0))' AS RESULT;
 result
--------
 t
(1 row)
```

- ?-

描述：图形是否水平对齐。

示例：

```
postgres=# SELECT point '(1,0)' ?- point '(0,0)' AS RESULT;
 result
--------
 t
(1 row)
```

- ?|

描述：图形是否处于竖直位置。

示例：

```
postgres=# SELECT ?| lseg '((-1,0),(1,0))' AS RESULT;
 result
--------
 f
(1 row)
```

- ?|

描述：图形是否竖直对齐。

示例：

```
postgres=# SELECT point '(0,1)' ?| point '(0,0)' AS RESULT;
 result
--------
 t
(1 row)
```

- ?-|

  描述：两条线是否垂直。

  示例：

```
postgres=# SELECT lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))' AS RESULT;
 result
--------
 t
(1 row)
```

- ?||

  描述：两条线是否平行。

  示例：

```
postgres=# SELECT lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))' AS RESULT;
 result
--------
 t
(1 row)
```

- @>

  描述：图形是否包含另一个图形。

  示例：

```
postgres=# SELECT circle '((0,0),2)' @> point '(1,1)' AS RESULT;
 result
--------
 t
(1 row)
```

- <@

描述：图形是否被包含于另一个图形。

示例：

```
postgres=# SELECT point '(1,1)' <@ circle '((0,0),2)' AS RESULT;
 result
--------
 t
(1 row)
```

- ~=

  描述：两个图形是否相同。

  示例：

```
postgres=# SELECT polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))' AS RESULT;
 result
--------
 t
(1 row)
```

5.10.2 几何函数

- area(object)

  描述：计算图形的面积。

  返回类型：double precision

  示例：

```
postgres=# SELECT area(box '((0,0),(1,1))') AS RESULT;
 result
--------
 t
(1 row)
```

- center(object)

  描述：计算图形的中心。

  返回类型：point

  示例：

```
postgres=# SELECT center(box '((0,0),(1,2))') AS RESULT;
```

```
 result
---------
 (0.5,1)
(1 row)
```

- diameter(circle)

  描述：计算圆的直径。

  返回类型：double precision

  示例：

```
postgres=# SELECT diameter(circle '((0,0),2.0)') AS RESULT;
 result
--------
      4
(1 row)
```

- height(box)

  描述：矩形的竖直高度。

  返回类型：double precision

  示例：

```
postgres=# SELECT height(box '((0,0),(1,1))') AS RESULT;
 result
--------
      1
(1 row)
```

- isclosed(path)

  描述：图形是否为闭合路径。

  返回类型：Boolean

  示例：

```
postgres=# SELECT isclosed(path '((0,0),(1,1),(2,0))') AS RESULT;
 result
--------
 t
(1 row)
```

- isopen(path)

描述：图形是否为开放路径。

返回类型：Boolean

示例：

```
postgres=# SELECT isopen(path '[(0,0),(1,1),(2,0)]') AS RESULT;
 result
--------
 t
(1 row)
```

- length(object)

  描述：计算图形的长度。

  返回类型：double precision

  示例：

```
postgres=# SELECT length(path '((-1,0),(1,0))') AS RESULT;
 result
--------
      4
(1 row)
```

- npoints(path)

  描述：计算路径的顶点数。

  返回类型：int

  示例：

```
postgres=# SELECT npoints(path '[(0,0),(1,1),(2,0)]') AS RESULT;
 result
--------
      3
(1 row)
```

- npoints(polygon)

  描述：计算多边形的顶点数。

  返回类型：int

  示例：

```
postgres=# SELECT npoints(polygon '((1,1),(0,0))') AS RESULT;
 result
--------
      2
(1 row)
```

- pclose(path)

  描述：把路径转换为闭合路径。

  返回类型：path

  示例：

```
postgres=# SELECT pclose(path '[(0,0),(1,1),(2,0)]') AS RESULT;
       result
---------------------
 ((0,0),(1,1),(2,0))
(1 row)
```

- popen(path)

  描述：把路径转换为开放路径。

  返回类型：path

  示例：

```
postgres=# SELECT popen(path '((0,0),(1,1),(2,0))') AS RESULT;
       result
---------------------
 [(0,0),(1,1),(2,0)]
(1 row)
```

- radius(circle)

  描述：计算圆的半径。

  返回类型：double precision

  示例：

```
 postgres=# SELECT radius(circle '((0,0),2.0)') AS RESULT;
 result
--------
      2
(1 row)
```

- width(box)

  描述：计算矩形的水平尺寸。

  返回类型：double precision

  示例：

```
postgres=# SELECT width(box '((0,0),(1,1))') AS RESULT;
 result
--------
      1
(1 row)
```

5.10.3 几何类型转换函数

- box(circle)

  描述：将圆转换成矩形

  返回类型：box

  示例：

```
postgres=# SELECT box(circle '((0,0),2.0)') AS RESULT;
                                  result
-------------------------------------------------------------------------
 (1.41421356237309,1.41421356237309),(-1.41421356237309,-1.41421356237309)
(1 row)
```

- box(point, point)

  描述：将点转换成矩形

  返回类型：box

  示例：

```
postgres=# SELECT box(point '(0,0)', point '(1,1)') AS RESULT;
   result
-------------
 (1,1),(0,0)
(1 row)
```

- box(polygon)

  描述：将多边形转换成矩形

返回类型：box

示例：

```
postgres=# SELECT box(polygon '((0,0),(1,1),(2,0))') AS RESULT;
   result
-------------
 (2,1),(0,0)
(1 row)
```

- circle(box)

  描述：矩形转换成圆

  返回类型：circle

  示例：

```
postgres=# SELECT circle(box '((0,0),(1,1))') AS RESULT;
            result
-----------------------------
 <(0.5,0.5),0.707106781186548>
(1 row)
```

- circle(point, double precision)

  描述：将圆心和半径转换成圆

  返回类型：circle

  示例：

```
postgres=# SELECT circle(point '(0,0)', 2.0) AS RESULT;
  result
-----------
 <(0,0),2>
(1 row)
```

- circle(polygon)

  描述：将多边形转换成圆

  返回类型：circle

  示例：

```
postgres=# SELECT circle(polygon '((0,0),(1,1),(2,0))') AS RESULT;
                  result
```

```
-----------------------------------------
<(1,0.333333333333333),0.924950591148529>
(1 row)
```

- lseg(box)

  描述：矩形对角线转化成线段

  返回类型：lseg

  示例：

```
postgres=# SELECT lseg(box '((-1,0),(1,0))') AS RESULT;
    result
----------------
[(1,0),(-1,0)]
(1 row)
```

- lseg(point, point)

  描述：点转换成线段

  返回类型：lseg

  示例：

```
postgres=# SELECT lseg(point '(-1,0)', point '(1,0)') AS RESULT;
    result
----------------
[(-1,0),(1,0)]
(1 row)
```

- slope(point, point)

  描述：计算两个点构成直线的斜率

  返回类型：double

  示例：

```
postgres=# SELECT slope(point '(1,1)', point '(0,0)') AS RESULT;
result
--------
     1
(1 row)
```

- path(polygon)

描述：多边形转换成路径

返回类型：path

示例：

```
postgres=# SELECT path(polygon '((0,0),(1,1),(2,0))') AS RESULT;
      result
--------------------
 ((0,0),(1,1),(2,0))
(1 row)
```

- point(double precision, double precision)

  描述：节点

  返回类型：point

  示例：

```
postgres=# SELECT point(23.4, -44.5) AS RESULT;
   result
--------------
 (23.4,-44.5)
(1 row)
```

- point(box)

  描述：矩形的中心

  返回类型：point

  示例：

```
postgres=# SELECT point(box '((-1,0),(1,0))') AS RESULT;
 result
--------
 (0,0)
(1 row)
```

- point(circle)

  描述：圆心

  返回类型：point

  示例：

```
postgres=# SELECT point(circle '((0,0),2.0)') AS RESULT;
 result
--------
 (0,0)
(1 row)
```

- point(lseg)

  描述：线段的中心

  返回类型：point

  示例：

```
postgres=# SELECT point(lseg '((-1,0),(1,0))') AS RESULT;
 result
--------
 (0,0)
(1 row)
```

- point(polygon)

  描述：多边形的中心

  返回类型：point

  示例：

```
postgres=# SELECT point(polygon '((0,0),(1,1),(2,0))') AS RESULT;
        result
----------------------
 (1,0.333333333333333)
(1 row)
```

- polygon(box)

  描述：矩形转换成 4 点多边形

  返回类型：polygon

  示例：

```
postgres=# SELECT polygon(box '((0,0),(1,1))') AS RESULT;
          result
--------------------------
 ((0,0),(0,1),(1,1),(1,0))
(1 row)
```

- polygon(circle)

  描述：圆转换成 12 点多边形

  返回类型：polygon

  示例：

```
postgres=# SELECT polygon(circle '((0,0),2.0)') AS RESULT;

result

------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------------------------------------------------

((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),
(1,1.73205080756888),(1.73205080756888,1),(2,2.44929359829471e-16),(1.7320508
0756888,-0.999999999999999),(1,-1.73205080756888),(3.67394039744206e-16,-2),(
-0.999999999999999,-1.73205080756888),(-1.73205080756888,-1))
(1 row)
```

- polygon(npts, circle)

  描述：圆转换成 npts 点多边形

  返回类型：polygon

  示例：

```
postgres=# SELECT polygon(12, circle '((0,0),2.0)') AS RESULT;

result

------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------------------------------------------------

((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),
(1,1.73205080756888),(1.73205080756888,1),(2,2.44929359829471e-16),(1.7320508
0756888,-0.999999999999999),(1,-1.73205080756888),(3.67394039744206e-16,-2),(
-0.999999999999999,-1.73205080756888),(-1.73205080756888,-1))
(1 row)
```

- polygon(path)

  描述：路径转换成多边形

  返回类型：polygon

  示例：

```
postgres=# SELECT polygon(path '((0,0),(1,1),(2,0))') AS RESULT;
      result
--------------------
 ((0,0),(1,1),(2,0))
(1 row)
```

5.11 网络地址函数和操作符

5.11.1 cidr 和 inet 操作符

操作符<<、<<=、>>、>>=对子网进行测试。它们只考虑两个地址的网络部分（忽略任何主机部分），然后判断其中一个网络是等于另外一个网络，还是另外一个网络的子网。

- <

  描述：小于

  示例：

```
postgres=# SELECT inet '192.168.1.5' < inet '192.168.1.6' AS RESULT;
 result
--------
 t
(1 row)
```

- <=

  描述：小于或等于

  示例：

```
postgres=# SELECT inet '192.168.1.5' <= inet '192.168.1.5' AS RESULT;
 result
--------
 t
(1 row)
```

- =

描述：等于

示例：

```
postgres=# SELECT inet '192.168.1.5' = inet '192.168.1.5' AS RESULT;
 result
--------
 t
(1 row)
```

- >=

描述：大于或等于

示例：

```
postgres=# SELECT inet '192.168.1.5' >= inet '192.168.1.5' AS RESULT;
 result
--------
 t
(1 row)
```

- >

描述：大于

示例：

```
postgres=# SELECT inet '192.168.1.5' > inet '192.168.1.4' AS RESULT;
 result
--------
 t
(1 row)
```

- <>

描述：不等于

示例：

```
postgres=# SELECT inet '192.168.1.5' <> inet '192.168.1.4' AS RESULT;
 result
--------
 t
(1 row)
```

- <<

描述：包含于

示例：

```
postgres=# SELECT inet '192.168.1.5' << inet '192.168.1/24' AS RESULT;
 result
--------
 t
(1 row)
```

- <<=

描述：包含于或等于

示例：

```
postgres=# SELECT inet '192.168.1/24' <<= inet '192.168.1/24' AS RESULT;
 result
--------
 t
(1 row)
```

- >>

描述：包含

示例：

```
postgres=# SELECT inet '192.168.1/24' >> inet '192.168.1.5' AS RESULT;
 result
--------
 t
(1 row)
```

- >>=

描述：包含或等于

示例：

```
postgres=# SELECT inet '192.168.1/24' >>= inet '192.168.1/24' AS RESULT;
 result
--------
 t
(1 row)
```

- ~

描述：位非

示例：

```
postgres=# SELECT ~ inet '192.168.1.6' AS RESULT;
   result
---------------
 63.87.254.249
(1 row)
```

- &

描述：两个网络地址的每一位都进行“与”操作

示例：

```
postgres=# SELECT inet '192.168.1.6' & inet '10.0.0.0' AS RESULT;
 result
---------
 0.0.0.0
(1 row)
```

- |

描述：两个网络地址的每一位都进行“或”操作

示例：

```
postgres=# SELECT inet '192.168.1.6' | inet '10.0.0.0' AS RESULT;
  result
-------------
 202.168.1.6
(1 row)
```

- +

描述：加

示例：

```
postgres=# SELECT inet '192.168.1.6' + 25 AS RESULT;
   result
--------------
 192.168.1.31
(1 row)
```

- -

描述：减

示例：

```
postgres=# SELECT inet '192.168.1.43' - 36 AS RESULT;
  result
-------------
 192.168.1.7
(1 row)
```

- -

描述：减

示例：

```
postgres=# SELECT inet '192.168.1.43' - inet '192.168.1.19' AS RESULT;
 result
--------
     24
(1 row)
```

5.11.2 cidr 和 inet 函数

函数 abbrev、host、text 主要是为了提供可选的显示格式。

- abbrev(inet)

描述：缩写显示格式文本。

返回类型：text

示例：

```
postgres=# SELECT abbrev(inet '10.1.0.0/16') AS RESULT;
  result
-------------
 10.1.0.0/16
(1 row)
```

- abbrev(cidr)

描述：缩写显示格式文本。

返回类型：text

示例：

```
postgres=# SELECT abbrev(cidr '10.1.0.0/16') AS RESULT;
 result
---------
 10.1/16
(1 row)
```

- broadcast(inet)

  描述：网络广播地址。

  返回类型：inet

  示例：

```
postgres=# SELECT broadcast('192.168.1.5/24') AS RESULT;
      result
------------------
 192.168.1.255/24
(1 row)
```

- family(inet)

  描述：抽取地址族，4 为 IPv4，6 为 IPv6。

  返回类型：int

  示例：

```
postgres=# SELECT family('127.0.0.1') AS RESULT;
 result
--------
      4
(1 row)
```

- host(inet)

  描述：将主机地址类型抽出为文本。

  返回类型：text

  示例：

```
postgres=# SELECT host('192.168.1.5/24') AS RESULT;
   result
-------------
 192.168.1.5
(1 row)
```

- hostmask(inet)

  描述：为网络构造主机掩码。

  返回类型：inet

  示例：

```
postgres=# SELECT hostmask('192.168.23.20/30') AS RESULT;
 result
---------
 0.0.0.3
(1 row)
```

- masklen(inet)

  描述：抽取子网掩码长度。

  返回类型：int

  示例：

```
postgres=# SELECT masklen('192.168.1.5/24') AS RESULT;
 result
--------
     24
(1 row)
```

- netmask(inet)

  描述：为网络构造子网掩码。

  返回类型：inet

  示例：

```
postgres=# SELECT netmask('192.168.1.5/24') AS RESULT;
    result
---------------
 255.255.255.0
(1 row)
```

- network(inet)

  描述：抽取地址的网络部分。

  返回类型：cidr

示例：

```
postgres=# SELECT network('192.168.1.5/24') AS RESULT;
    result
---------------
 192.168.1.0/24
(1 row)
```

- set_masklen(inet, int)

  描述：为 inet 数值设置子网掩码长度。

  返回类型：inet

  示例：

```
postgres=# SELECT set_masklen('192.168.1.5/24', 16) AS RESULT;
    result
---------------
 192.168.1.5/16
(1 row)
```

- set_masklen(cidr, int)

  描述：为 cidr 数值设置子网掩码长度。

  返回类型：cidr

  示例：

```
postgres=# SELECT set_masklen('192.168.1.0/24'::cidr, 16) AS RESULT;
    result
---------------
 192.168.0.0/16
(1 row)
```

- text(inet)

  描述：把 IP 地址和掩码长度抽取为文本。

  返回类型：text

  示例：

```
postgres=# SELECT text(inet '192.168.1.5') AS RESULT;
    result
---------------
 192.168.1.5/32
```

```
(1 row)
```

任何 cidr 值都能以显式或者隐式的方式转换为 inet 值，因此上述能够操作 inet 值的函数也同样能够操作 cidr 值。inet 值也可以转换为 cidr 值，此时 inet 子网掩码右侧的所有位都将转换为零，以创建一个有效的 cidr 值。另外，用户还可以使用常规的类型转换语法将一个文本字符串转换为 inet 或 cidr 值。例如：inet(expression)或 colname::cidr。

5.11.3 macaddr 函数

函数 trunc(macaddr)返回一个 MAC 地址，该地址的最后三个字节设置为零。

- trunc(macaddr)

  描述：把后三个字节置为零。

  返回类型：macaddr

  示例：

```
postgres=# SELECT trunc(macaddr '12:34:56:78:90:ab') AS RESULT;
     result
-------------------
 12:34:56:00:00:00
(1 row)
```

macaddr 类型还支持标准关系操作符（>、<=等）用于词法排序，和按位运算符（~、&和|）非、与和或。

5.12 文本检索函数和操作符

5.12.1 文本检索操作符

- @@

  描述：tsvector 类型的词汇与 tsquery 类型的词汇是否匹配

  示例：

```
postgres=# SELECT to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')
AS RESULT;
 result
--------
 t
(1 row)
```

- @@@

  描述：@@的同义词

  示例：

```
postgres=# SELECT to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')
AS RESULT;
 result
--------
 t
(1 row)
```

- ||

  描述：连接两个 tsvector 类型的词汇

  示例：

```
postgres=# SELECT 'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector AS RESULT;
          result
---------------------------
 'a':1 'b':2,5 'c':3 'd':4
(1 row)
```

- &&

  描述：将两个 tsquery 类型的词汇进行“与”操作

  示例：

```
postgres=# SELECT 'fat | rat'::tsquery && 'cat'::tsquery AS RESULT;
          result
---------------------------
 ( 'fat' | 'rat' ) & 'cat'
(1 row)
```

- ||

  描述：将两个 tsquery 类型的词汇进行“或”操作

  示例：

```
postgres=# SELECT 'fat | rat'::tsquery || 'cat'::tsquery AS RESULT;
          result
---------------------------
 ( 'fat' | 'rat' ) | 'cat'
```

```
(1 row)
```

- !!

  描述：tsquery 类型词汇的非关系

  示例：

```
postgres=# SELECT !! 'cat'::tsquery AS RESULT;
 result
--------
 !'cat'
(1 row)
```

- @>

  描述：一个 tsquery 类型的词汇是否包含另一个 tsquery 类型的词汇

  示例：

```
postgres=# SELECT 'cat'::tsquery @> 'cat & rat'::tsquery AS RESULT;
 result
--------
 f
(1 row)
```

- <@

  描述：一个 tsquery 类型的词汇是否被包含另一个 tsquery 类型的词汇

  示例：

```
postgres=# SELECT 'cat'::tsquery <@ 'cat & rat'::tsquery AS RESULT;
 result
--------
 t
(1 row)
```

除了上述的操作符，还为 tsvector 类型和 tsquery 类型的数据定义了普通的 B-tree 比较操作符（=、<等）。

5.12.2 文本检索函数

- get_current_ts_config()

  描述：获取文本检索的默认配置。

返回类型：regconfig

示例：

```
postgres=# SELECT get_current_ts_config();
get_current_ts_config
----------------------
english
(1 row)
```

- length(tsvector)

  描述：tsvector 类型词汇的单词数。

  返回类型：integer

  示例：

```
postgres=# SELECT length('fat:2,4 cat:3 rat:5A'::tsvector);
length
--------
      3
(1 row)
```

- numnode(tsquery)

  描述：tsquery 类型的单词加上操作符的数量。

  返回类型：integer

  示例：

```
postgres=# SELECT numnode('(fat & rat) | cat'::tsquery);
numnode
---------
       5
(1 row)
```

- plainto_tsquery([config regconfig ,] query text)

  描述：产生 tsquery 类型的词汇，并忽略标点。

  返回类型：tsquery

  示例：

```
postgres=# SELECT plainto_tsquery('english', 'The Fat Rats');
plainto_tsquery
```

```
----------------
'fat' & 'rat'
(1 row)
```

- querytree(query tsquery)

  描述：获取 tsquery 类型的词汇可加索引的部分。

  返回类型：text

  示例：

```
postgres=# SELECT querytree('foo & ! bar'::tsquery);
 querytree
-----------
'foo'
(1 row)
```

- setweight(tsvector, “char”)

  描述：给 tsvector 类型的每个元素分配权值。

  返回类型：tsvector

  示例：

```
postgres=# SELECT setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A');
          setweight
------------------------------
'cat':3A 'fat':2A,4A 'rat':5A
(1 row)
```

- strip(tsvector)

  描述：删除 tsvector 类型单词中的 position 和权值。

  返回类型：tsvector

  示例：

```
postgres=# SELECT strip('fat:2,4 cat:3 rat:5A'::tsvector);
      strip
------------------
'cat' 'fat' 'rat'
(1 row)
```

- to_tsquery([config regconfig ,] query text)

描述：标准化单词，并转换为 tsquery 类型。

返回类型：tsquery

示例：

```
postgres=# SELECT to_tsquery('english', 'The & Fat & Rats');
 to_tsquery
---------------
'fat' & 'rat'
(1 row)
```

- to_tsvector([config regconfig ,] document text)

  描述：去除文件信息，并转换为 tsvector 类型。

  返回类型：tsvector

  示例：

```
postgres=# SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
'fat':2 'rat':3
(1 row)
```

- to_tsvector_for_batch([config regconfig ,] document text)

  描述：去除文件信息，并转换为 tsvector 类型。

  返回类型：tsvector

  示例：

```
postgres=# SELECT to_tsvector_for_batch('english', 'The Fat Rats');
   to_tsvector
-----------------
'fat':2 'rat':3
(1 row)
```

- ts_headline([config regconfig,] document text, query tsquery [, options text])

  描述：高亮显示查询的匹配项。

  返回类型：text

  示例：

```
postgres=# SELECT ts_headline('x y z', 'z'::tsquery);
 ts_headline
--------------
 x y <b>z</b>
(1 row)
```

- ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer])

  描述：文档查询排名。

  返回类型：float4

  示例：

```
postgres=# SELECT ts_rank('hello world'::tsvector, 'world'::tsquery);
 ts_rank
----------
 .0607927
(1 row)
```

- ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer])

  描述：排序文件查询使用覆盖密度。

  返回类型：float4

  示例：

```
postgres=# SELECT ts_rank_cd('hello world'::tsvector, 'world'::tsquery);
ts_rank_cd
------------
          0
(1 row)
```

- ts_rewrite(query tsquery, target tsquery, substitute tsquery)

  描述：替换目标 tsquery 类型的单词。

  返回类型：tsquery

  示例：

```
postgres=# SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery,
'foo|bar'::tsquery);
       ts_rewrite
-----------------------
 'b' & ( 'foo' | 'bar' )
```

```
(1 row)
```

- ts_rewrite(query tsquery, select text)

  描述：使用 SELECT 命令的结果替代目标中 tsquery 类型的单词。

  返回类型：tsquery

  示例：

```
postgres=# SELECT ts_rewrite('world'::tsquery, 'select ''world''::tsquery,
''hello''::tsquery');
 ts_rewrite
------------
 'hello'
(1 row)
```

5.12.3 文本检索调试函数

- ts_debug([config regconfig,] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])

  描述：测试一个配置。

  返回类型：setof record

  示例：

```
postgres=# SELECT ts_debug('english', 'The Brightest supernovaes');
                                    ts_debug
-------------------------------------------------------------------------------
------
 (asciiword,"Word, all ASCII",The,{english_stem},english_stem,{})
 (blank,"Space symbols"," ",{},,)
 (asciiword,"Word, all
ASCII",Brightest,{english_stem},english_stem,{brightest})
 (blank,"Space symbols"," ",{},,)
 (asciiword,"Word, all
ASCII",supernovaes,{english_stem},english_stem,{supernova})
(5 rows)
```

- ts_lexize(dict regdictionary, token text)

  描述：测试一个数据字典。

返回类型：text[]

示例：

```
postgres=# SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}
(1 row)
```

- ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)

  描述：测试一个解析。

  返回类型：setof record

  示例：

```
postgres=# SELECT ts_parse('default', 'foo - bar');
 ts_parse
-----------
 (1,foo)
 (12," ")
 (12,"- ")
 (1,bar)
(4 rows)
```

- ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)

  描述：测试一个解析。

  返回类型：setof record

  示例：

```
postgres=# SELECT ts_parse(3722, 'foo - bar');
 ts_parse
-----------
 (1,foo)
 (12," ")
 (12,"- ")
 (1,bar)
(4 rows)
```

- ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description text)

  描述：获取分析器定义的记号类型。

返回类型：setof record

示例：

```
postgres=# SELECT ts_token_type('default');
                        ts_token_type
------------------------------------------------------------
(1,asciiword,"Word, all ASCII")
(2,word,"Word, all letters")
(3,numword,"Word, letters and digits")
(4,email,"Email address")
(5,url,URL)
(6,host,Host)
(7,sfloat,"Scientific notation")
(8,version,"Version number")
(9,hword_numpart,"Hyphenated word part, letters and digits")
(10,hword_part,"Hyphenated word part, all letters")
(11,hword_asciipart,"Hyphenated word part, all ASCII")
(12,blank,"Space symbols")
(13,tag,"XML tag")
(14,protocol,"Protocol head")
(15,numhword,"Hyphenated word, letters and digits")
(16,asciihword,"Hyphenated word, all ASCII")
(17,hword,"Hyphenated word, all letters")
(18,url_path,"URL path")
(19,file,"File or path name")
(20,float,"Decimal notation")
(21,int,"Signed integer")
(22,uint,"Unsigned integer")
(23,entity,"XML entity")
(23 rows)
```

- ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description text)

  描述：获取分析器定义的记号类型。

  返回类型：setof record

  示例：

```
postgres=# SELECT ts_token_type(3722);
                        ts_token_type
------------------------------------------------------------
(1,asciiword,"Word, all ASCII")
```

```
 (2,word,"Word, all letters")
 (3,numword,"Word, letters and digits")
 (4,email,"Email address")
 (5,url,URL)
 (6,host,Host)
 (7,sfloat,"Scientific notation")
 (8,version,"Version number")
 (9,hword_numpart,"Hyphenated word part, letters and digits")
 (10,hword_part,"Hyphenated word part, all letters")
 (11,hword_asciipart,"Hyphenated word part, all ASCII")
 (12,blank,"Space symbols")
 (13,tag,"XML tag")
 (14,protocol,"Protocol head")
 (15,numhword,"Hyphenated word, letters and digits")
 (16,asciihword,"Hyphenated word, all ASCII")
 (17,hword,"Hyphenated word, all letters")
 (18,url_path,"URL path")
 (19,file,"File or path name")
 (20,float,"Decimal notation")
 (21,int,"Signed integer")
 (22,uint,"Unsigned integer")
 (23,entity,"XML entity")
(23 rows)
```

- ts_stat(sqlquery text, [weights text,] OUT word text, OUT ndoc integer, OUT nentry integer)

  描述：获取 tsvector 列的统计数据。

  返回类型：setof record

  示例：

```
postgres=# SELECT ts_stat('select ''hello world''::tsvector');
  ts_stat
-------------
 (world,1,1)
 (hello,1,1)
(2 rows)
```

5.13 JSON/JSONB 函数和操作符

JSON/JSONB 数据类型参考 JSON/JSONB 类型。

表 5-10 JSON/JSONB 通用操作符

操作符	左操作数类型	右操作数类型	返回类型	描述	例子	例子结果
->	Array-json(b)	int	json(b)	获得array-json元素。下标不存在返回空。	'[{“a”:“foo”},{“b”:“bar”},{“c”:“baz”}]'::json->2	{“c”:“baz”}
->	object-json(b)	text	json(b)	通过键获得值。不存在则返回空。	'{“a” : {“b”:“foo”}}'::json->'a'	{“b”:“foo”}
->>	Array-json(b)	int	text	获得JSON数组元素。下标不存在返回空。	'[1,2,3]'::json->>2	3
->>	object-json(b)	text	text	通过键获得值。不存在则返回空。	'{“a”:1,“b”:2}'::json->>'b'	2
#>	container-json (b)	text[]	json(b)	获取在指定路径的JSON	'{“a” : {“b”:{“c” : “foo”}}}	{“c” : “foo”}

				对象，路径不存在则返回空。	'::json #>'{a,b}'	
#>>	container -json (b)	text[]	text	获取在指定路径的 JSON 对象，路径不存在则返回空。	'{“a”:[1, 2,3],“b”: [4,5,6]}': :json #>>'{a,2}'	3

⚠ 注意

对于 #> 和 #>> 操作符，当给出的路径无法查找到数据时，不会报错，会返回空。

表 5-11 JSONB 额外支持操作符

操作符	右操作数类型	描述	例子
@>	jsonb	左边的 JSON 的顶层是否包含右边 JSON 的顶层所有项。	'{“a”:1, “b”:2}'::jsonb @> '{“b”:2}'::jsonb
<@	jsonb	左边的 JSON 的所有项是否全部存在于右边 JSON 的顶层。	'{“b”:2}'::jsonb <@ '{“a”:1, “b”:2}'::jsonb
?	text	键/元素的字符串是否存在于 JSON 值的顶层。	'{“a”:1, “b”:2}'::jsonb ? 'b'
?|	text[]	这些数组字符串中的任何一个是否做为顶层键存在。	'{“a”:1,“b”:2,“c”:3}' ::jsonb ? | array['b', 'c']
?&	text[]	是否所有这些数组字符串都	'[“a”, “b”]'::jsonb ?&

		作为顶层键存在。	array['a', 'b']
=	jsonb	判断两个 jsonb 的大小关系，同函数 jsonb_eq。	/
<>	jsonb	判断两个 jsonb 的大小关系，同函数 jsonb_ne。	/
<	jsonb	判断两个 jsonb 的大小关系，同函数 jsonb_lt。	/
>	jsonb	判断两个 jsonb 的大小关系，同函数 jsonb_gt。	/
<=	jsonb	判断两个 jsonb 的大小关系，同函数 jsonb_le。	/
>=	jsonb	判断两个 jsonb 的大小关系，同函数 jsonb_ge。	/

JSON/JSONB 支持的函数

- array_to_json\(anyarray \[, pretty_bool\]\)

描述：返回 JSON 类型的数组。一个多维数组成为一个 JSON 数组的数组。如果 pretty_bool 为 true，将在一维元素之间添加换行符。

返回类型：json

示例：

```
postgres=# SELECT array_to_json('{{1,5},{99,100}}'::int[]);
   array_to_json
   ------------------
   [[1,5],[99,100]]
   (1 row)
```

- row_to_json\(record \[, pretty_bool\]\)

描述：返回 JSON 类型的行。如果 pretty_bool 为 true，将在第一级元素之间添加换行符。

返回类型：json

示例：

```
postgres=# SELECT row_to_json(row(1,'foo'));
    row_to_json
---------------------
 {"f1":1,"f2":"foo"}
(1 row)
```

- json_array_element(array-json, integer)、jsonb_array_element(array-jsonb, integer)

  描述：同操作符`->`，返回数组中指定下标的元素。

  返回类型：json、jsonb

  示例：

```
postgres=# select json_array_element('[1,true,[1,[2,3]],null]',2);
 json_array_element
 --------------------
 [1,[2,3]]
 (1 row)
```

- json_array_element_text(array-json, integer)、jsonb_array_element_text(array-jsonb, integer)

  描述：同操作符`->>`，返回数组中指定下标的元素。

  返回类型：text、text

  示例：

```
postgres=# select json_array_element_text('[1,true,[1,[2,3]],null]',2);
json_array_element_text
-----------------------
 [1,[2,3]]
 (1 row)
```

- json_object_field(object-json, text)、jsonb_object_field(object-jsonb, text)

  描述：同操作符`->`，返回对象中指定键对应的值。

  返回类型：json、json

  示例：

```
  postgres=# select json_object_field('{"a": {"b":"foo"}}','a');
  json_object_field
```

```
------------------
{"b":"foo"}
(1 row)
```

- json_object_field_text(object-json, text)、jsonb_object_field_text(object-jsonb, text)

  描述：同操作符`->>`，返回对象中指定键对应的值。

  返回类型：text、text

  示例：

```
 postgres=# select json_object_field_text('{"a": {"b":"foo"}}','a');
json_object_field_text
----------------------
{"b":"foo"}
(1 row)
```

- json_extract_path(json, VARIADIC text[])、jsonb_extract_path((jsonb, VARIADIC text[])

  描述：等价于操作符`#>`。根据$2 所指的路径，查找 json，并返回。

  返回类型：json、jsonb

  示例：

```
  postgres=# select
json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}',
'f4','f6');
   json_extract_path
------------------
 "stringy"
(1 row)
```

- json_extract_path_op(json, text[])、jsonb_extract_path_op(jsonb, text[])

  描述：同操作符`#>`。根据$2 所指的路径，查找 json，并返回。

  返回类型：json、jsonb

  示例：

```
   postgres=# select
json_extract_path_op('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}',
'f4','f6');
   json_extract_path_op
--------------------
```

```
 "stringy"
(1 row)
```

- json_extract_path_text(json, VARIADIC text[])、jsonb_extract_path_text((jsonb, VARIADIC text[])

  描述：等价于操作符`#>>`。根据$2 所指的路径，查找 json，并返回。

  返回类型：text、text

  示例：

```
 postgres=# select
json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}',
'f4','f6');
  json_extract_path_text
------------------------
 "stringy"
(1 row)
```

- json_extract_path_text_op(json, text[])、jsonb_extract_path_text_op(jsonb, text[])

  描述：同操作符`#>>`。根据$2 所指的路径，查找 json，并返回。

  返回类型：text、text

  示例：

```
 postgres=# select json_extract_path_text_op
('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}', 'f4','f6');
  json_extract_path_text_op
---------------------------
 "stringy"
(1 row)
```

- json_array_elements(array-json)、jsonb_array_elements(array-jsonb)

  描述：拆分数组，每一个元素返回一行。

  返回类型：json、jsonb

  示例：

```
 postgres=# select json_array_elements('[1,true,[1,[2,3]],null]');
 json_array_elements
---------------------
 1
```

```
 true
 [1,[2,3]]
 null
(4 rows)
```

- json_array_elements_text(array-json)、jsonb_array_elements_text(array-jsonb)

  描述：拆分数组，每一个元素返回一行。

  返回类型：text、text

  示例：

```
postgres=# select * from
json_array_elements_text('[1,true,[1,[2,3]],null]');
    value
-----------
 1
 true
 [1,[2,3]]

(4 rows)
```

- json_array_length(array-json)、jsonb_array_length(array-jsonb)

  描述：返回数组长度。

  返回类型：integer

  示例：

```
   postgres=# SELECT
json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4,null]');
 json_array_length
-------------------
                 6
(1 row)
```

- json_each(object-json)、jsonb_each(object-jsonb)

  描述：将对象的每个键值对拆分转换成一行两列。

  返回类型：setof(key text, value json)、setof(key text, value jsonb)

  示例：

```
 postgres=# select * from
json_each('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
 key |  value
-----+----------
 f1  | [1,2,3]
 f2  | {"f3":1}
 f4  | null
(3 rows)
```

- json_each_text(object-json)、jsonb_each_text(object-jsonb)

  描述：将对象的每个键值对拆分转换成一行两列。

  返回类型：setof(key text, value text)、setof(key text, value text)

  示例：

```
 postgres=# select * from
json_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
 key |  value
-----+----------
 f1  | [1,2,3]
 f2  | {"f3":1}
 f4  |
(3 rows)
```

- json_object_keys(object-json)、jsonb_object_keys(object-jsonb)

  描述：返回对象中顶层的所有键。

  返回类型：SETOF text

  示例：

```
 postgres=# select json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"},
"f1":"abcd"}');
 json_object_keys
------------------
 f1
 f2
 f1
(3 rows)
```

  jsonb 中会有去重操作

```
 postgres=# select jsonb_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"},
"f1":"abcd"}');
   jsonb_object_keys
  -------------------
   f1
   f2
  (2 rows)
```

- json_populate_record(anyelement, object-json [, bool])、jsonb_populate_record(anyelement, object-jsonb [, bool])

描述：$1 必须是一个复合类型的参数。将会把 object-json 里的每个对键值进行拆分，以键当做列名，与$1 中的列名进行匹配查找，并填充到$1 的格式中。

返回类型：anyelement、anyelement

示例：

```
 postgres=# create type jpop as (a text, b int, c bool);
  CREATE TYPE
postgres=# select * from
json_populate_record(null::jpop,'{"a":"blurfl","x":43.2}');
   a    | b | c
--------+---+---
 blurfl |   |
(1 row)
postgres=# select * from
json_populate_record((1,1,null)::jpop,'{"a":"blurfl","x":43.2}');
   a    | b | c
--------+---+---
 blurfl | 1 |
(1 row)
postgres=# DROP TYPE jpop;
DROP TYPE
```

- json_populate_record_set(anyelement, array-json [, bool])、

  jsonb_populate_record_set(anyelement, array-jsonb [, bool])

描述：参考上述函数 json_populate_record、jsonb_populate_record，对$2 数组的每一个元素进行上述参数函数的操作，因此这也要求$2 数组的每个元素都是 object-json 类型的。

返回类型：setof anyelement、setof anyelement

示例：

```
 postgres=# create type jpop as (a text, b int, c bool);
  CREATE TYPE
 postgres=# select * from json_populate_recordset(null::jpop,
'[{"a":1,"b":2},{"a":3,"b":4}]');
   a | b | c
  ---+---+---
   1 | 2 |
   3 | 4 |
  (2 rows)
```

- json_typeof(json)、jsonb_typeof(jsonb)

  描述：检测 json 类型

  返回类型：text、text

  示例：

```
  postgres=# select value, json_typeof(value)
  gbase-# from (values (json '123.4'), (json '"foo"'), (json 'true'), (json
'null'), (json '[1, 2, 3]'), (json '{"x":"foo", "y":123}'), (NULL::json)) as
data(value);
          value          | json_typeof
  -----------------------+-------------
   123.4                 | number
   "foo"                 | string
   true                  | boolean
   null                  | null
   [1, 2, 3]             | array
   {"x":"foo", "y":123}  | object
                         |
  (7 rows)
```

- json_build_array([VARIADIC “any”])

  描述：从一个可变参数列表构造出一个 JSON 数组。

  返回类型：array-json

  示例：

```
   postgres=# select
json_build_array('a',1,'b',1.2,'c',true,'d',null,'e',json '{"x": 3, "y":
[1,2,3]}','');
```

```
                                json_build_array

-------------------------------------------------------------------------------
   ["a", 1, "b", 1.2, "c", true, "d", null, "e", {"x": 3, "y": [1,2,3]}, ""]
(1 row)
```

- json_build_object([VARIADIC “any”])

描述：从一个可变参数列表构造出一个 JSON 对象，其入参必须为偶数个，两两一组组成键值对。注意键不可为 null。

返回类型：object-json

示例：

```
 postgres=# select json_build_object(1,2);
 json_build_object
-------------------
 {"1" : 2}
(1 row)
```

- json_to_record(object-json, bool)

描述：正如所有返回 record 的函数一样，调用者必须用一个 AS 子句显式地定义记录的结构。会将 object-json 的键值对进行拆分重组，把键当做列名，去匹配填充 as 显示指定的记录的结构。

返回类型：record

示例：

```
postgres=# select * from json_to_record('{"a":1,"b":"foo","c":"bar"}',true) as
x(a int, b text, d text);
 a |  b  | d
---+-----+---
 1 | foo |
(1 row)
```

- json_to_recordset(array-json, bool)

描述：参考函数 json_to_record，对数组内个每个元素，执行上述函数的操作，因此这要求数组内的每个元素都得是 object-json。

返回类型：setof record

示例：

```
postgres=# select * from json_to_recordset(
  gbase(# '[{"a":1,"b":"foo","d":false},{"a":2,"b":"bar","c":true}]',
  gbase(# false
  gbase(# ) as x(a int, b text, c boolean);
   a |  b  | c
  ---+-----+---
   1 | foo |
   2 | bar | t
  (2 rows)
```

- json_object(text[])、json_object(text[], text[])

描述：从一个文本数组构造一个 object-json。这是个重载函数，当入参为一个文本数组的时候，其数组长度必须为偶数，成员被当做交替出现的键/值对。两个文本数组的时候，第一个数组认为是键，第二个认为是值，两个数组长度必须相等。键不可为 null。

返回类型：object-json

示例：

```
postgres=# select json_object('{a,1,b,2,3,NULL,"d e f","a b c"}');
                      json_object
 ------------------------------------------------
  {"a" : "1", "b" : "2", "3" : null, "d e f" : "a b c"}
 (1 row)
postgres=# select json_object('{a,b,"a b c"}', '{a,1,1}');
               json_object
  ---------------------------------
   {"a" : "a", "b" : "1", "a b c" : "1"}
  (1 row)
```

- json_agg(any)

描述：将值聚集为 json 数组。

返回类型：array-json

示例：

```
postgres=# create table classes (name varchar, score int8);
CREATE TABLE
postgres=# insert into classes values('A',2),('A',3),('D',5),('D',NULL);
INSERT 0 5
postgres=# select * from classes;
  name | score
```

```
 -----+-------
 A    |     2
 A    |     3
 D    |     5
 D    |
   (4 rows)
 postgres=# select name, json_agg(score) score from classes group by name order
by name;
 name |      score
 -----+----------------
 A    | [2, 3]
 D    | [5, null]
(2 rows)
```

- json_object_agg(any, any)

  描述：将值聚集为 json 对象。

  返回类型：object-json

  示例：

```
postgres=# select * from classes;
 name | score
 -----+-------
 A    |     2
 A    |     3
 D    |     5
 D    |
 (4 rows)
 postgres=# select json_object_agg(name, score) from classes group by name order
by name;
      json_object_agg
 ------------------------
  { "A" : 2, "A" : 3 }
  { "D" : 5, "D" : null }
 (2 rows)
```

- jsonb_contained(jsonb, jsonb)

  描述：同操作符 `<@`，判断$1 中的所有元素是否在$2 的顶层存在。

  返回类型：bool

  示例：

```
 postgres=# select jsonb_contained('[1,2,3]', '[1,2,3,4]');
 jsonb_contained
-----------------
 t
(1 row)
```

- jsonb_contains(jsonb, jsonb)

  描述：同操作符 `@>`，判断$1 中的顶层所有元素是否包含在$2 的所有元素。

  返回类型：bool

  示例：

```
 postgres=# select jsonb_contains('[1,2,3,4]', '[1,2,3]');
 jsonb_contains
----------------
 t
(1 row)
```

- jsonb_exists(jsonb, text)

  描述：同操作符 `?`，字符串$2 是否存在$1 的顶层以 key\elem\scalar 的形式存在。

  返回类型：bool

  示例：

```
  postgres=# select jsonb_exists('["1",2,3]', '1');
 jsonb_exists
--------------
 t
(1 row)
```

- jsonb_exists_all(jsonb, text[])

  描述：同操作符 `?&`，字符串数组$2 里面，是否所有的元素，都在$1 的顶层以 key\elem\scalar 的形式存在。

  返回类型：bool

  示例：

```
  postgres=# select jsonb_exists_all('["1","2",3]', '{1, 2}');
 jsonb_exists_all
------------------
 t
```

```
(1 row)
```

- jsonb_exists_any(jsonb, text[])

描述：同操作符 `?|`，字符串数组$2 里面，是否存在的元素，在$1 的顶层以 key\elem\scalar 的形式存在。

返回类型：bool

示例：

```
postgres=# select jsonb_exists_any('["1","2",3]', '{1, 2, 4}');
 jsonb_exists_any
------------------
 t
(1 row)
```

- jsonb_cmp(jsonb, jsonb)

描述：比较大小，正数代表大于，负数代表小于，0 表示相等。

返回类型：integer

示例：

```
postgres=# select jsonb_cmp('["a", "b"]', '{"a":1, "b":2}');
 jsonb_cmp
-----------
        -1
(1 row)
```

- jsonb_eq(jsonb, jsonb)

描述：同操作符 `=`，比较两个值的大小。

返回类型：bool

示例：

```
postgres=# select jsonb_eq('["a", "b"]', '{"a":1, "b":2}');
 jsonb_eq
----------
 f
(1 row)
```

- jsonb_ne(jsonb, jsonb)

描述：同操作符 `<>`，比较两个值的大小。

返回类型：bool

示例：

```
 postgres=# select jsonb_ne('["a", "b"]', '{"a":1, "b":2}');
 jsonb_ne
----------
 t
(1 row)
```

- jsonb_gt(jsonb, jsonb)

  描述：同操作符 `>`，比较两个值的大小。

  返回类型：bool

  示例：

```
 postgres=# select jsonb_gt('["a", "b"]', '{"a":1, "b":2}');
 jsonb_gt
----------
 f
(1 row)
```

- jsonb_ge(jsonb, jsonb)

  描述：同操作符 `>=`，比较两个值的大小。

  返回类型：bool

  示例：

```
 postgres=# select jsonb_ge('["a", "b"]', '{"a":1, "b":2}');
 jsonb_ge
----------
 f
(1 row)
```

- jsonb_lt(jsonb, jsonb)

  描述：同操作符 `<`，比较两个值的大小。

  返回类型：bool

  示例：

```
 postgres=# select jsonb_lt('["a", "b"]', '{"a":1, "b":2}');
 jsonb_lt
```

```
----------
 t
(1 row)
```

- jsonb_le(jsonb, jsonb)

  描述：同操作符 `<=`，比较两个值的大小。

  返回类型：bool

  示例：

```
postgres=# select jsonb_le('["a", "b"]', '{"a":1, "b":2}');
 jsonb_le
----------
 t
(1 row)
```

- to_json(anyelement)

  描述：把参数转换为`json`。

  返回类型：json

  示例：

```
postgres=# select to_json('{1,5}'::text[]);
  to_json
-----------
 ["1","5"]
(1 row)
```

- jsonb_hash(jsonb)

  描述：对 jsonb 进行 hash 运算。

  返回类型：integer

  示例：

```
postgres=# select jsonb_hash('[1,2,3]');
 jsonb_hash
------------
 -559968547
(1 row)
```

  其他函数

描述：gin 索引以及 json\jsonb 聚集函数所用到的内部函数，功能不过多赘述。

gin_compare_jsonb、gin_consistent_jsonb、gin_consistent_jsonb_hash、gin_extract_jsonb、gin_extract_jsonb_hash 、 gin_extract_jsonb_query 、 gin_extract_jsonb_query_hash 、gin_triconsistent_jsonb、gin_triconsistent_jsonb_hash、json_agg_transfn、json_agg_finalfn、json_object_agg_transfn、json_object_agg_finalfn 等。

5.14 HLL 函数和操作符

5.14.1 哈希函数

- hll_hash_boolean(bool)

  描述：对 bool 类型数据计算哈希值。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_boolean(FALSE);
 hll_hash_boolean
--------------------
 -5451962507482445012
(1 row)
```

- hll_hash_boolean(bool, int32)

  描述：设置 hash seed（即改变哈希策略）并对 bool 类型数据计算哈希值。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_boolean(FALSE, 10);
 hll_hash_boolean
-------------------
 -1169037589280886076
(1 row)
```

- hll_hash_smallint(smallint)

  描述：对 smallint 类型数据计算哈希值。

  返回值类型：hll_hashval

示例：

```
postgres=# SELECT hll_hash_smallint(100::smallint);
 hll_hash_smallint
--------------------
 962727970174027904
(1 row)
```

说明

数值大小相同的参数使用不同数据类型的哈希函数计算，最后结果会不一样，因为不同类型哈希函数会选取不同的哈希计算策略。

- hll_hash_smallint(smallint, int32)

  描述：设置 hash seed（即改变哈希策略）同时对 smallint 类型数据计算哈希值。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_smallint(100::smallint, 10);
 hll_hash_smallint
--------------------
 -9056177146160443041
(1 row)
```

- hll_hash_integer(integer)

  描述：对 integer 类型数据计算哈希值。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_integer(0);
  hll_hash_integer
---------------------
 5156626420896634997
(1 row)
```

- hll_hash_integer(integer, int32)

  描述：对 integer 类型数据计算哈希值，并设置 hashseed（即改变哈希策略）。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_integer(0, 10);
 hll_hash_integer
------------------
 -5035020264353794276
(1 row)
```

- hll_hash_bigint(bigint)

  描述：对 bigint 类型数据计算哈希值。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_bigint(100::bigint);
  hll_hash_bigint
--------------------
 -2401963681423227794
(1 row)
```

- hll_hash_bigint(bigint, int32)

  描述：对 bigint 类型数据计算哈希值，并设置 hashseed（即改变哈希策略）。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_bigint(100::bigint, 10);
  hll_hash_bigint
--------------------
 -2305749404374433531
(1 row)
```

- hll_hash_bytea(bytea)

  描述：对 bytea 类型数据计算哈希值。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_bytea(E'\\x');
hll_hash_bytea
---------------
 0
(1 row)
```

- hll_hash_bytea(bytea, int32)

  描述：对 bytea 类型数据计算哈希值，并设置 hashseed（即改变哈希策略）。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_bytea(E'\\x', 10);
  hll_hash_bytea
--------------------
 7233188113542599437
(1 row)
```

- hll_hash_text(text)

  描述：对 text 类型数据计算哈希值。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_text('AB');
   hll_hash_text
--------------------
 -5666002586880275174
(1 row)
```

- hll_hash_text(text, int32)

  描述：对 text 类型数据计算哈希值，并设置 hashseed（即改变哈希策略）。

  返回值类型：hll_hashval

  示例：

```
postgres=# SELECT hll_hash_text('AB', 10);
hll_hash_text
--------------------
-2215507121143724132
(1 row)
```

- hll_hash_any(anytype)

  描述：对任意类型数据计算哈希值。

  返回值类型：hll_hashval

示例：

```
postgres=# select hll_hash_any(1);
    hll_hash_any
---------------------
 -1316670585935156930
(1 row)

postgres=# select hll_hash_any('08:00:2b:01:02:03'::macaddr);
    hll_hash_any
---------------------
 -3719950434455589360
(1 row)
```

- hll_hash_any(anytype, int32)

  描述：对任意类型数据计算哈希值，并设置 hashseed（即改变哈希策略）。

  返回值类型：hll_hashval

  示例：

```
postgres=# select hll_hash_any(1, 10);
    hll_hash_any
---------------------
 7048553517657992351
(1 row)
```

- hll_hashval_eq(hll_hashval, hll_hashval)

  描述：比较两个 hll_hashval 类型数据是否相等。

  返回值类型：bool

  示例：

```
postgres=# select hll_hashval_eq(hll_hash_integer(1), hll_hash_integer(1));
 hll_hashval_eq
----------------
 t
(1 row)
```

- hll_hashval_ne(hll_hashval, hll_hashval)

  描述：比较两个 hll_hashval 类型数据是否不相等。

  返回值类型：bool

示例：

```
postgres=# select hll_hashval_ne(hll_hash_integer(1), hll_hash_integer(1));
 hll_hashval_ne
----------------
 f
(1 row)
```

5.14.2 日志函数

hll 主要存在三种模式 Explicit、Sparse、Full。当数据规模比较小的时候会使用 Explicit 模式，这种模式下 distinct 值的计算是没有误差的；随着 distinct 值越来越多，hll 会先后转换为 Sparse 模式和 Full 模式，这两种模式在计算结果上没有任何区别，只影响 hll 函数的计算效率和 hll 对象的存储空间。下面的函数可以用于查看 hll 的一些参数。

- hll_print(hll)

  描述：打印 hll 的一些 debug 参数信息。

  示例：

```
postgres=# select hll_print(hll_empty());
                                    hll_print
------------------------------------------------------------------------------
--
 type=1(HLL_EMPTY), log2m=14, log2explicit=10, log2sparse=12, duplicatecheck=0
(1 row)
```

- hll_type(hll)

  描述：查看当前 hll 的类型。返回值具体含义如下：返回值 0，表示 HLL_UNINIT，未初始化的 hll 对象;返回值 1,表示 HLL_EMPTY,hll 空对象;返回值 2,表示 HLL_EXPLICIT,Explicit 模式的 hll 对象；返回值 3，表示 HLL_SPARSE，Sparse 模式的 hll 对象；返回值 4,表示 HLL_FULL，Full 模式的 hll 对象；返回值 5，表示 HLL_UNDEFINED，不合法的 hll 对象。

  示例：

```
postgres=# select hll_type(hll_empty());
 hll_type
----------
        1
(1 row)
```

- hll_log2m(hll)

描述：查看当前 hll 数据结构中的 log2m 数值，log2m 是分桶数的对数值，此值会影响最后 hll 计算 distinct 误差率，误差率计算公式为 ± 1.04/√(2 ^ log2m)。当显式指定 log2m 的取值为 10-16 之间时，hll 会设置分桶数为 2log2m。当显示指定 log2explicit 为-1 时，会采用内置默认值。

示例：

```
postgres=# select hll_log2m(hll_empty());
 hll_log2m
-----------
        14
(1 row)
postgres=# select hll_log2m(hll_empty(10));
 hll_log2m
-----------
        10
(1 row)
postgres=# select hll_log2m(hll_empty(-1));
 hll_log2m
-----------
        14
(1 row)
```

- hll_log2explicit(hll)

描述：查看当前 hll 数据结构中的 log2explicit 数值。hll 通常会由 Explicit 模式到 Sparse 模式再到 Full 模式，这个过程称为 promotion hierarchy 策略。可以通过调整 log2explicit 值的大小改变策略，比如 log2explicit 为 0 的时候就会跳过 Explicit 模式而直接进入 Sparse 模式。当显式指定 log2explicit 的取值为 1-12 之间时，hll 会在数据段长度超过 2log2explicit 时转为 Sparse 模式。当显示指定 log2explicit 为-1 时，会采用内置默认值。

示例：

```
postgres=# select hll_log2explicit(hll_empty());
 hll_log2explicit
------------------
               10
(1 row)
postgres=# select hll_log2explicit(hll_empty(12, 8));
 hll_log2explicit
```

```
-----------------
               8
(1 row)

postgres=# select hll_log2explicit(hll_empty(12, -1));
 hll_log2explicit
------------------
               10
(1 row)
```

- hll_log2sparse(hll)

描述：查看当前 hll 数据结构中的 log2sparse 数值。hll 通常会由 Explicit 模式到 Sparse 模式再到 Full 模式，这个过程称为 promotion hierarchy 策略。可以通过调整 log2sparse 值的大小改变策略，比如 log2sparse 为 0 的时候就会跳过 Sparse 模式而直接进入 Full 模式。当显式指定 Sparse 的取值为 1-14 之间时，hll 会在数据段长度超过 2log2sparse 时转为 Full 模式。当显示指定 log2sparse 为-1 时，会采用内置默认值。

示例：

```
 postgres=# select hll_log2sparse(hll_empty());
 hll_log2sparse
----------------
             12
(1 row)
postgres=# select hll_log2sparse(hll_empty(12, 8, 10));
 hll_log2sparse
----------------
             10
(1 row)
postgres=# select hll_log2sparse(hll_empty(12, 8, -1));
 hll_log2sparse
----------------
             12
(1 row)
```

- hll_duplicatecheck(hll)

描述：是否启用 duplicatecheck，0 是关闭，1 是开启。默认关闭，对于有较多重复值出现的情况，可以开启以提高效率。当显示指定 duplicatecheck 为-1 时，会采用内置默认值。

示例：

```
postgres=# select hll_duplicatecheck(hll_empty());
 hll_duplicatecheck
--------------------
                  0
(1 row)
postgres=# select hll_duplicatecheck(hll_empty(12, 8, 10, 1));
 hll_duplicatecheck
--------------------
                  1
(1 row)

postgres=# select hll_duplicatecheck(hll_empty(12, 8, 10, -1));
 hll_duplicatecheck
--------------------
                  0
(1 row)
```

5.14.3 功能函数

- hll_empty()

  描述：创建一个空的 hll。

  返回值类型：hll

  示例：

```
postgres=# select hll_empty();
                          hll_empty
-------------------------------------------------------------
 \x484c4c00000000002b050000000000000000000000000000000000000
(1 row)
```

- hll_empty(int32 log2m)

  描述：创建空的 hll 并指定参数 log2m，取值范围是 10 到 16。若输入-1，则采用内置默认值。

  返回值类型：hll

  示例：

```
postgres=# select hll_empty(10);
                          hll_empty
-------------------------------------------------------------
```

```
 \x484c4c00000000002b040000000000000000000000000000000000000000
(1 row)

postgres=# select hll_empty(-1);
                          hll_empty
------------------------------------------------------------
 \x484c4c00000000002b050000000000000000000000000000000000000000
(1 row)
```

- hll_empty(int32 log2m, int32 log2explicit)

描述：创建空的 hll 并依次指定参数 log2m、log2explicit。log2explicit 取值范围是 0 到 12，0 表示直接跳过 Explicit 模式。该参数可以用来设置 Explicit 模式的阈值大小，在数据段长度达到 2log2explicit 后切换为 Sparse 模式或者 Full 模式。若输入-1，则 log2explicit 采用内置默认值。

返回值类型：hll

示例：

```
postgres=# select hll_empty(10, 4);
                          hll_empty
------------------------------------------------------------
 \x484c4c0000000000130400000000000000000000000000000000000000
(1 row)

postgres=# select hll_empty(10, -1);
                          hll_empty
------------------------------------------------------------
 \x484c4c00000000002b040000000000000000000000000000000000000000
(1 row)
```

- hll_empty(int32 log2m, int32 log2explicit, int64 log2sparse)

描述：创建空的 hll 并依次指定参数 log2m、log2explicit、log2sparse。log2sparse 取值范围是 0 到 14，0 表示直接跳过 Sparse 模式。该参数可以用来设置 Sparse 模式的阈值大小，在数据段长度达到 2log2sparse 后切换为 Full 模式。若输入-1，则 log2sparse 采用内置默认值。

返回值类型：hll

示例：

```
postgres=# select hll_empty(10, 4, 8);
                          hll_empty
```

```
------------------------------------------------------------
 \x484c4c0000000000120400000000000000000000000000000000000000
(1 row)

postgres=# select hll_empty(10, 4, -1);
                          hll_empty
------------------------------------------------------------
 \x484c4c0000000000130400000000000000000000000000000000000000
(1 row)
```

- hll_empty(int32 log2m, int32 log2explicit, int64 log2sparse, int32 duplicatecheck)

描述：创建空的 hll 并依次指定参数 log2m、log2explicit、log2sparse、duplicatecheck。duplicatecheck 取 0 或者 1，表示是否开启该模式，默认情况下该模式会关闭。若输入-1，则 duplicatecheck 采用内置默认值。

返回值类型：hll

示例：

```
postgres=# select hll_empty(10, 4, 8, 0);
                          hll_empty
------------------------------------------------------------
 \x484c4c0000000000120400000000000000000000000000000000000000
(1 row)

postgres=# select hll_empty(10, 4, 8, -1);
                          hll_empty
------------------------------------------------------------
 \x484c4c0000000000120400000000000000000000000000000000000000
(1 row)
```

- hll_add(hll, hll_hashval)

描述：把 hll_hashval 加入到 hll 中。

返回值类型：hll

示例：

```
postgres=# select hll_add(hll_empty(), hll_hash_integer(1));
                                  hll_add
------------------------------------------------------------------------
 \x484c4c08000002002b0900000000000000f03f3e2921ff133fbaed3e2921ff133fbaed00
(1 row)
```

- hll_add_rev(hll_hashval, hll)

  描述：把 hll_hashval 加入到 hll 中，和 hll_add 功能一样，只是参数位置进行了交换。

  返回值类型：hll

  示例：

```
postgres=# select hll_add_rev(hll_hash_integer(1), hll_empty());
                                hll_add_rev
-------------------------------------------------------------------------------
 \x484c4c08000002002b0900000000000000f03f3e2921ff133fbaed3e2921ff133fbaed00
(1 row)
```

- hll_eq(hll, hll)

  描述：比较两个 hll 是否相等。

  返回值类型：bool

  示例：

```
postgres=# select hll_eq(hll_add(hll_empty(), hll_hash_integer(1)),
hll_add(hll_empty(), hll_hash_integer(2)));
 hll_eq
--------
 f
(1 row)
```

- hll_ne(hll, hll)

  描述：比较两个 hll 是否不相等。

  返回值类型：bool

  示例：

```
postgres=# select hll_ne(hll_add(hll_empty(), hll_hash_integer(1)),
hll_add(hll_empty(), hll_hash_integer(2)));
 hll_ne
--------
 t
(1 row)
```

- hll_cardinality(hll)

  描述：计算 hll 的 distinct 值。

返回值类型：int

示例：

```
postgres=# select hll_cardinality(hll_empty() || hll_hash_integer(1));
 hll_cardinality
-----------------
               1
(1 row)
```

- hll_union(hll, hll)

描述：把两个 hll 数据结构 union 成一个。

返回值类型：hll

示例：

```
postgres=# select hll_union(hll_add(hll_empty(), hll_hash_integer(1)),
hll_add(hll_empty(), hll_hash_integer(2)));
                                        hll_union
--------------------------------------------------------------------------------
--------------

\x484c4c10002000002b090000000000000000400000000000000000b3ccc49320cca1ae3e292
1ff133fbaed00
(1 row)
```

5.14.4 聚合函数

- hll_add_agg(hll_hashval)

描述：把哈希后的数据按照分组放到 hll 中。

返回值类型：hll

示例：

```
--准备数据
postgres=# create table t_id(id int);
CREATE TABLE
postgres=# insert into t_id values(generate_series(1,500));
INSERT 0 500
postgres=# create table t_data(a int, c text);
CREATE TABLE
postgres=# insert into t_data select mod(id,2), id from t_id;
```

```
INSERT 0 500

--创建表并指定列为 hll
postgres=# create table t_a_c_hll(a int, c hll);
CREATE TABLE
--根据 a 列 group by 对数据分组，把各组数据加到 hll 中
postgres=# insert into t_a_c_hll select a, hll_add_agg(hll_hash_text(c)) from
t_data group by a;
INSERT 0 2
--得到每组数据中 hll 的 Distinct 值
postgres=# select a, #c as cardinality from t_a_c_hll order by a;
 a |    cardinality
---+-----------------
 0 | 247.862354346299
 1 | 250.908710610377
(2 rows)
```

- hll_add_agg(hll_hashval, int32 log2m)

描述：把哈希后的数据按照分组放到 hll 中，并指定参数 log2m，取值范围是 10 到 16。若输入-1 或者 NULL，则采用内置默认值。

返回值类型：hll

示例：

```
 postgres=# select hll_cardinality(hll_add_agg(hll_hash_text(c), 12)) from
t_data;
 hll_cardinality
-----------------
 497.965240179228
(1 row)
```

- hll_add_agg(hll_hashval, int32 log2m, int32 log2explicit)

描述：把哈希后的数据按照分组放到 hll 中，依次指定参数 log2m、log2explicit。log2explicit 取值范围是 0 到 12，0 表示直接跳过 Explicit 模式。该参数可以用来设置 Explicit 模式的阈值大小，在数据段长度达到 2log2explicit 后切换为 Sparse 模式或者 Full 模式。若输入-1 或者 NULL，则 log2explicit 采用内置默认值。

返回值类型：hll

示例：

```
postgres=# select hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1)) from
t_data;
 hll_cardinality
------------------
 498.496062953313
(1 row)
```

- hll_add_agg(hll_hashval, int32 log2m, int32 log2explicit, int64 log2sparse)

描述:把哈希后的数据按照分组放到hll中,依次指定参数log2m、log2explicit、log2sparse。log2sparse 取值范围是 0 到 14，0 表示直接跳过 Sparse 模式。该参数可以用来设置 Sparse 模式的阈值大小，在数据段长度达到 2log2sparse 后切换为 Full 模式。若输入-1 或者 NULL，则 log2sparse 采用内置默认值。

返回值类型：hll

示例:

```
postgres=# select hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 6, 10))
from t_data;
 hll_cardinality
------------------
 498.496062953313
(1 row)
```

- hll_add_agg(hll_hashval, int32 log2m, int32 log2explicit, int64 log2sparse, int32 duplicatecheck)

描述:把哈希后的数据按照分组放到 hll 中，依次制定参数 log2m、log2explicit、log2sparse、duplicatecheck，duplicatecheck 取值范围是 0 或者 1，表示是否开启该模式，默认情况下该模式会关闭。若输入-1 或者 NULL，则 duplicatecheck 采用内置默认值。

返回值类型：hll

示例:

```
postgres=# select hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 6, 10,
-1)) from t_data;
 hll_cardinality
------------------
 498.496062953313
(1 row)
```

- hll_union_agg(hll)

描述：将多个 hll 类型数据 union 成一个 hll。

返回值类型：hll

示例：

```
--将各组中的 hll 数据 union 成一个 hll，并计算 distinct 值。
postgres=# select #hll_union_agg(c) as cardinality from t_a_c_hll;
   cardinality
------------------
 498.496062953313
(1 row)
```

说明

当两个或者多个 hll 数据结构做 union 的时候，必须要保证其中每一个 hll 里面的精度参数一样，否则将不可以进行 union。同样的约束也适用于函数 hll_union(hll,hll)。

5.14.5 废弃函数

此外，还存在一些 HLL 旧函数。可用类似的函数进行替代。

- hll_schema_version(hll)

描述：查看当前 hll 中的 schema version。旧版本 schema version 是常值 1，用来进行 hll 字段的头部校验，重构后的 hll 在头部增加字段“HLL”进行校验，schema version 不再使用。

- hll_regwidth(hll)

描述：查看 hll 数据结构中桶的位数大小。旧版本桶的位数 regwidth 取值 1~5，会存在较大的误差，也限制了基数估计上限。 重构后 regwidth 为固定值 6，不再使用 regwidth 变量。

- hll_expthresh(hll)

描述：得到当前 hll 中 expthresh 大小。采用 hll_log2explicit(hll)替代类似功能。

- hll_sparseon(hll)

描述：是否启用 Sparse 模式。采用 hll_log2sparse(hll)替代类似功能，0 表示关闭 Sparse 模式。

5.14.6 内置函数

HLL（HyperLogLog）有一系列内置函数用于内部对数据进行处理，一般情况下用户不需要熟知这些函数的使用。详见下表。

表 5-12 内置函数

函数名称	功能描述
hll_in	以 string 格式接收 hll 数据。
hll_out	以 string 格式发送 hll 数据。
hll_recv	以 bytea 格式接收 hll 数据。
hll_send	以 bytea 格式发送 hll 数据。
hll_trans_in	以 string 格式接收 hll_trans_type 数据。
hll_trans_out	以 string 格式发送 hll_trans_type 数据。
hll_trans_recv	以 bytea 形式接收 hll_trans_type 数据。
hll_trans_send	以 bytea 形式发送 hll_trans_type 数据。
hll_typmod_in	接收 typmod 类型数据。
hll_typmod_out	发送 typmod 类型数据。
hll_hashval_in	接收 hll_hashval 类型数据。
hll_hashval_out	发送 hll_hashval 类型数据。
hll_add_trans0	类似于 hll_add 所提供的功能，初始化时无指定入参，通常在聚合运算的第一阶段 DN 上使用。
hll_add_trans1	类似于 hll_add 所提供的功能，初始化时指定一个入参，通常在聚合运算的第一阶段 DN 上使用。

hll_add_trans2	类似于 hll_add 所提供的功能，初始化时指定两个入参，通常在聚合运算的第一阶段 DN 上使用。
hll_add_trans3	类似于 hll_add 所提供的功能，初始化时指定三个入参，通常在聚合运算的第一阶段 DN 上使用。
hll_add_trans4	类似于 hll_add 所提供的功能，初始化时指定四个入参，通常在聚合运算的第一阶段 DN 上使用。
hll_union_trans	类似 hll_union 所提供的功能，在聚合运算的第一阶段 DN 上使用。
hll_union_collect	类似于 hll_union 所提供的功能，在聚合运算第二阶段 DN 上使用，汇总各个 DN 上的结果。
hll_pack	在聚合运算第三阶段 DN 上使用，把自定义 hll_trans_type 类型最后转换成 hll 类型。
hll	用于 hll 类型转换成 hll 类型，根据输入参数会设定指定参数。
hll_hashval	用于 bigint 类型转换成 hll_hashval 类型。
hll_hashval_int4	用于 int4 类型转换成 hll_hashval 类型。

5.14.7 操作符

- =

  描述：比较 hll 或 hll_hashval 的值是否相等。

  返回值类型：bool

  示例：

```
--hll
postgres=# select (hll_empty() || hll_hash_integer(1)) = (hll_empty() ||
hll_hash_integer(1));
?column?
----------
```

```
 t
(1 row)
--hll_hashval
postgres=# select hll_hash_integer(1) = hll_hash_integer(1);
 ?column?
----------
 t
(1 row)
```

- <> or !=

  描述：比较 hll 或 hll_hashval 是否不相等。

  返回值类型：bool

  示例：

```
 --hll
postgres=# select (hll_empty() || hll_hash_integer(1)) <> (hll_empty() ||
hll_hash_integer(2));
 ?column?
----------
 t
(1 row)

--hll_hashval
postgres=# select hll_hash_integer(1) <> hll_hash_integer(2);
 ?column?
----------
 t
(1 row)
```

- ||

  描述：可代表 hll_add、hll_union、hll_add_rev 三个函数的功能。

  返回值类型：hll

  示例：

```
 --hll_add
postgres=# select hll_empty() || hll_hash_integer(1);
                                  ?column?
-----------------------------------------------------------------------------
 \x484c4c08000002002b0900000000000000f03f3e2921ff133fbaed3e2921ff133fbaed00
```

```
(1 row)
--hll_add_rev
postgres=# select hll_hash_integer(1) || hll_empty();
                                  ?column?
---------------------------------------------------------------------------
 \x484c4c08000002002b0900000000000000f03f3e2921ff133fbaed3e2921ff133fbaed00
(1 row)
--hll_union
postgres=# select (hll_empty() || hll_hash_integer(1)) || (hll_empty() ||
hll_hash_integer(2));
                                          ?column?
---------------------------------------------------------------------------
---------------

\x484c4c10002000002b0900000000000000040000000000000000b3ccc49320cca1ae3e292
1ff133fbaed00
(1 row)
```

- #

  描述：计算出 hll 的 Dintinct 值，同 hll_cardinality 函数。

  返回值类型：int

  示例：

```
postgres=# select #(hll_empty() || hll_hash_integer(1));
 ?column?
----------
        1
(1 row)
```

5.15 序列函数

序列（SEQUEENCE）函数能够确保用户在序列对象获取后续的序列值时，保证多用户安全。

- nextval(regclass)

  描述：递增序列并返回新值。

  说明

  为了避免从同一个序列获取值的并发事务被阻塞，nextval 操作不会回滚；也就是说，

一旦一个值已经被抓取，那么就认为它已经被用过了，并且不会再被返回。即使该操作处于事务中，当事务之后中断，或者如果调用查询结束不使用该值，也是如此。这种情况将在指定值的顺序中留下未使用的“空洞”。因此，GBase 8s 序列对象不能用于获得“无间隙”序列。

须知

nextval 函数只能在主机上执行，备机不支持执行此函数。

返回类型：numeric

nextval 函数有两种调用方式（其中第二种调用方式目前不支持 Sequence 命名中有特殊字符”.“的情况），如下：

```
调用方式 1。例如：
  postgres=# select nextval('seqDemo');
调用方式 2。例如：
  postgres=# select seqDemo.nextval;
```

- currval(regclass)

返回当前会话里最近一次 nextval 返回的指定的 sequence 的数值。如果当前会话还没有调用过指定的 sequence 的 nextval，那么调用 currval 将会报错。

返回类型：numeric

currval 函数有两种调用方式（其中第二种调用方式目前不支持 Sequence 命名中有特殊字符”.“的情况），如下：

```
调用方式 1。例如：
 postgres=# select currval('seq1');
调用方式 2。例如：
 postgres=# select seq1.currval;
```

- lastval()

描述：返回当前会话里最近一次 nextval 返回的数值。这个函数等效于 currval，只是它不用序列名为参数，它抓取当前会话里面最近一次 nextval 使用的序列。如果当前会话还没有调用过 nextval，那么调用 lastval 将会报错。

返回类型：numeric

示例：

```
postgres=# select lastval();
```

- setval(regclass,numeric)

  描述：设置序列的当前数值。

  返回类型：numeric

  示例：

```
postgres=# select setval('seqDemo',1);
```

- setval(regclass, numeric, Boolean)

  描述：设置序列的当前数值以及 is_called 标志。

  返回类型：numeric

  示例：

```
postgres=# select setval('seqDemo',1,true);
```

  说明

  Setval 后当前会话会立刻生效，但如果其他会话有缓存的序列值，只能等到缓存值用尽才能感知 Setval 的作用。所以为了避免序列值冲突，setval 要谨慎使用。 因为序列是非事务的，setval 造成的改变不会由于事务的回滚而撤销。

  须知

  nextval 函数只能在主机上执行，备机不支持执行此函数。

- pg_sequence_last_value(sequence_oid oid, OUT cache_value int16, OUT last_value int16)

  描述：获取指定 sequence 的参数，包含缓存值，当前值。

  返回类型：int16，int16

5.16 数组函数和操作符

5.16.1 数组操作符

- =

  描述：两个数组是否相等

  示例：

```
postgres=# SELECT ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] AS RESULT ;
result
```

```
--------
 t
(1 row)
```

- <>

  描述：两个数组是否不相等

  示例：

```
postgres=# SELECT ARRAY[1,2,3] <> ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

- <

  描述：一个数组是否小于另一个数组

  示例：

```
postgres=# SELECT ARRAY[1,2,3] < ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

- >

  描述：一个数组是否大于另一个数组

  示例：

```
postgres=# SELECT ARRAY[1,4,3] > ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

- <=

  描述：一个数组是否小于或等于另一个数组

  示例：

```
postgres=# SELECT ARRAY[1,2,3] <= ARRAY[1,2,3] AS RESULT;
 result
```

```
--------
 t
(1 row)
```

- >=

  描述：一个数组是否大于或等于另一个数组

  示例：

```
postgres=# SELECT ARRAY[1,4,3] >= ARRAY[1,4,3] AS RESULT;
 result
--------
 t
(1 row)
```

- @>

  描述：一个数组是否包含另一个数组

  示例：

```
postgres=# SELECT ARRAY[1,4,3] @> ARRAY[3,1] AS RESULT;
 result
--------
 t
(1 row)
```

- <@

  描述：一个数组是否被包含于另一个数组

  示例：

```
postgres=# SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6] AS RESULT;
 result
--------
 t
(1 row)
```

- &&

  描述：一个数组是否和另一个数组重叠（有共同元素）

  示例：

```
postgres=# SELECT ARRAY[1,4,3] && ARRAY[2,1] AS RESULT;
 result
```

```
--------
 t
(1 row)
```

- ||

  描述：数组与数组进行连接

  示例：

```
postgres=# SELECT ARRAY[1,2,3] || ARRAY[4,5,6] AS RESULT;
   result
---------------
 {1,2,3,4,5,6}
(1 row)
postgres=# SELECT ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] AS RESULT;
         result
-------------------------
 {{1,2,3},{4,5,6},{7,8,9}}
(1 row)
```

- ||

  描述：元素与数组进行连接

  示例：

```
postgres=# SELECT 3 || ARRAY[4,5,6] AS RESULT;
 result
-----------
 {3,4,5,6}
(1 row)
```

- ||

  描述：数组与元素进行连接

  示例：

```
postgres=# SELECT ARRAY[4,5,6] || 7 AS RESULT;
 result
-----------
 {4,5,6,7}
(1 row)
```

数组比较是使用默认的 B-tree 比较函数对所有元素逐一进行比较的。多维数组的元素按

照行顺序进行访问。如果两个数组的内容相同但维数不等，决定排序顺序的首要因素是维数。

5.16.2 数组函数

- array_append(anyarray, anyelement)

  描述：向数组末尾添加元素，只支持一维数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_append(ARRAY[1,2], 3) AS RESULT;
 result
---------
 {1,2,3}
(1 row)
```

- array_prepend(anyelement, anyarray)

  描述：向数组开头添加元素，只支持一维数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_prepend(1, ARRAY[2,3]) AS RESULT;
 result
---------
 {1,2,3}
(1 row)
```

- array_cat(anyarray, anyarray)

  描述：连接两个数组，支持多维数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_cat(ARRAY[1,2,3], ARRAY[4,5]) AS RESULT;
   result
-------------
 {1,2,3,4,5}
(1 row)

postgres=# SELECT array_cat(ARRAY[[1,2],[4,5]], ARRAY[6,7]) AS RESULT;
```

```
      result
--------------------
 {{1,2},{4,5},{6,7}}
(1 row)
```

- array_union(anyarray, anyarray)

  描述：连接两个数组，只支持一维数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_union(ARRAY[1,2,3], ARRAY[3,4,5]) AS RESULT;
  result
-------------
 {1,2,3,3,4,5}
(1 row)
```

- array_union_distinct(anyarray, anyarray)

  描述：连接两个数组，并去重，只支持一维数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_union_distinct(ARRAY[1,2,3], ARRAY[3,4,5]) AS RESULT;
  result
-------------
 {1,2,3,4,5}
(1 row)
```

- array_intersect(anyarray, anyarray)

  描述：两个数组取交集，只支持一维数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_intersect(ARRAY[1,2,3], ARRAY[3,4,5]) AS RESULT;
  result
-------------
 {3}
(1 row)
```

- array_intersect_distinct(anyarray, anyarray)

描述：两个数组取交集，并去重，只支持一维数组。

返回类型：anyarray

示例：

```
 postgres=# SELECT array_intersect_distinct(ARRAY[1,2,2], ARRAY[2,2,4,5]) AS
RESULT;
   result
------------
 {2}
(1 row)
```

- array_except(anyarray, anyarray)

  描述：两个数组取差，只支持一维数组。

  返回类型：anyarray

  示例：

```
 postgres=# SELECT array_except(ARRAY[1,2,3], ARRAY[3,4,5]) AS RESULT;
   result
------------
 {1,2}
(1 row)
```

- array_except_distinct(anyarray, anyarray)

  描述：两个数组取差，并去重，只支持一维数组。

  返回类型：anyarray

  示例：

```
 postgres=# SELECT array_except_distinct(ARRAY[1,2,2,3], ARRAY[3,4,5]) AS
RESULT;
   result
------------
 {1,2}
(1 row)
```

- array_ndims(anyarray)

  描述：返回数组的维数。

  返回类型：int

示例：

```
postgres=# SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
 result
--------
      2
(1 row)
```

- array_dims(anyarray)

  描述：返回数组各个维度中的低位下标值和高位下标值。

  返回类型：text

  示例：

```
postgres=# SELECT array_dims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
   result
------------
 [1:2][1:3]
(1 row)
```

- array_length(anyarray, int)

  描述：返回指定数组维度的长度。int 为指定数组维度。

  返回类型：int

  示例：

```
postgres=# SELECT array_length(array[1,2,3], 1) AS RESULT;
 result
--------
      3
(1 row)

postgres=# SELECT array_length(array[[1,2,3],[4,5,6]], 2) AS RESULT;
 result
--------
      3
(1 row)
```

- array_lower(anyarray, int)

  描述：返回指定数组维数的下界。int 为指定数组维度。

  返回类型：int

示例：

```
postgres=# SELECT array_lower('[0:2]={1,2,3}'::int[], 1) AS RESULT;
 result
--------
      0
(1 row)
```

- array_upper(anyarray, int)

  描述：返回指定数组维数的上界。int 为指定数组维度。

  返回类型：int

  示例：

```
postgres=# SELECT array_upper(ARRAY[1,8,3,7], 1) AS RESULT;
 result
--------
      4
(1 row)
```

- array_upper(anyarray, int)

  描述：返回指定数组维数的上界。int 为指定数组维度。

  返回类型：int

  示例：

```
postgres=# SELECT array_upper(ARRAY[1,8,3,7], 1) AS RESULT;
 result
--------
      4
(1 row)
```

- array_remove(anyarray, anyelement)

  描述：移除数组中的所有指定元素。仅支持一维数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_remove(ARRAY[1,8,8,7], 8) AS RESULT;
 result
--------
 {1,7}
```

```
(1 row)
```

- array_to_string(anyarray, text [, text])

  描述：使用第一个 text 作为数组的新分隔符，使用第二个 text 替换数组值为 null 的值。

  返回类型：text

  示例：

```
postgres=# SELECT array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*') AS RESULT;
 result
-----------
 1,2,3,*,5
(1 row)
```

- array_delete(anyarray)

  描述：清空数组中的元素并返回一个同类型的空数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_delete(ARRAY[1,8,3,7]) AS RESULT;
 result
--------
 {}
(1 row)
```

- array_deleteidx(anyarray, int)

  描述：从数组中删除指定下标的元素并返回剩余元素组成的数组。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_deleteidx(ARRAY[1,2,3,4,5], 1) AS RESULT;
 result
-----------
 {2,3,4,5}
(1 row)
```

- array_extendnull(anyarray, int)

  描述：往数组尾部添加指定个数的 null 空元素。

返回类型：anyarray

示例：

```
postgres=# SELECT array_extendnull(ARRAY[1,8,3,7],1) AS RESULT;
   result
--------------
{1,8,3,7,null}
(1 row)
```

- array_trim(anyarray, int)

  描述：从数组尾部删除指定个数个元素。

  返回类型：anyarray

  示例：

```
postgres=# SELECT array_trim(ARRAY[1,8,3,7],1) AS RESULT;
 result
---------
 {1,8,3}
(1 row)
```

- array_exists(anyarray, int)

  描述：检查第二个参数是否是数组的合法下标。

  返回类型：boolean

  示例：

```
postgres=# SELECT array_exists(ARRAY[1,8,3,7],1) AS RESULT;
 result
--------
 t
(1 row)
```

- array_next(anyarray, int)

  描述：根据第二个入参返回数组中指定下标元素的下一个元素的下标。

  返回类型：int

  示例：

```
postgres=# SELECT array_next(ARRAY[1,8,3,7],1) AS RESULT;
 result
```

```
--------
       2
(1 row)
```

- array_prior(anyarray, int)

  描述：根据第二个入参返回数组中指定下标元素的上一个元素的下标。

  返回类型：int

  示例：

```
postgres=# SELECT array_prior(ARRAY[1,8,3,7],2) AS RESULT;
 result
--------
  1
(1 row)
```

- string_to_array(text, text [, text])

  描述：使用第二个 text 指定分隔符，使用第三个可选的 text 作为 NULL 值替换模板，如果分隔后的子串与第三个可选的 text 完全匹配，则将其替换为 NULL。

  返回类型：text[]

  示例：

```
postgres=# SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy') AS RESULT;
    result
--------------
 {xx,NULL,zz}
(1 row)
postgres=# SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'y') AS RESULT;
   result
------------
 {xx,yy,zz}
(1 row)
```

- unnest(anyarray)

  描述：扩大一个数组为一组行。

  返回类型：setof anyelement

  示例：

```
postgres=# SELECT unnest(ARRAY[1,2]) AS RESULT;
```

```
 result
--------
      1
      2
(2 rows)
```

在 string_to_array 中，如果分隔符参数是 NULL，输入字符串中的每个字符将在结果数组中变成一个独立的元素。如果分隔符是一个空白字符串，则整个输入的字符串将变为一个元素的数组。否则输入字符串将在每个分隔字符串处分开。

在 string_to_array 中，如果省略 null 字符串参数或为 NULL，将字符串中没有输入内容的子串替换为 NULL。

在 array_to_string 中，如果省略 null 字符串参数或为 NULL，运算中将跳过在数组中的任何 null 元素，并且不会在输出字符串中出现。

5.17 范围函数和操作符

5.17.1 范围操作符

- =

  描述：等于

  示例：

```
postgres=# SELECT int4range(1,5) = '[1,4]'::int4range AS RESULT;
 result
--------
 t
(1 row)
```

- <>

  描述：不等于

  示例：

```
postgres=# SELECT numrange(1.1,2.2) <> numrange(1.1,2.3) AS RESULT;
 result
--------
 t
(1 row)
```

- <

  描述：小于

  示例：

```
postgres=# SELECT int4range(1,10) < int4range(2,3) AS RESULT;
 result
--------
 t
(1 row)
```

- >

  描述：大于

  示例：

```
postgres=# SELECT int4range(1,10) > int4range(1,5) AS RESULT;
 result
--------
 t
(1 row)
```

- <=

  描述：小于或等于

  示例：

```
postgres=# SELECT numrange(1.1,2.2) <= numrange(1.1,2.2) AS RESULT;
 result
--------
 t
(1 row)
```

- >=

  描述：大于或等于

  示例：

```
postgres=# SELECT numrange(1.1,2.2) >= numrange(1.1,2.0) AS RESULT;
 result
--------
 t
(1 row)
```

- @>

  描述：包含范围

  示例：

```
postgres=# SELECT int4range(2,4) @> int4range(2,3) AS RESULT;
 result
--------
 t
(1 row)
```

- @>

  描述：包含元素

  示例：

```
 postgres=# SELECT '[2011-01-01,2011-03-01)'::tsrange @>
'2011-01-10'::timestamp AS RESULT;
 result
--------
 t
(1 row)
```

- <@

  描述：范围包含于

  示例：

```
postgres=# SELECT int4range(2,4) <@ int4range(1,7) AS RESULT;
 result
--------
 t
(1 row)
```

- <@

  描述：元素包含于

  示例：

```
postgres=# SELECT 42 <@ int4range(1,7) AS RESULT;
 result
--------
 f
```

```
(1 row)
```

- &&

  描述：重叠（有共同点）

  示例：

```
postgres=# SELECT int8range(3,7) && int8range(4,12) AS RESULT;
 result
--------
 t
(1 row)
```

- <<

  描述：范围值是否比另一个范围值的最小值还小（没有交集）。

  示例：

```
postgres=# SELECT int8range(1,10) << int8range(100,110) AS RESULT;
 result
--------
 t
(1 row)
```

- >>

  描述：范围值是否比另一个范围值的最大值还大（没有交集）。

  示例：

```
postgres=# SELECT int8range(50,60) >> int8range(20,30) AS RESULT;
 result
--------
 t
(1 row)
```

- &<

  描述：范围值的最大值是否不超过另一个范围值的最大值。

  示例：

```
postgres=# SELECT int8range(1,20) &< int8range(18,20) AS RESULT;
 result
--------
 t
```

```
(1 row)
```

- &>

  描述：范围值的最小值是否不小于另一个范围值的最小值。

  示例：

```
postgres=# SELECT int8range(7,20) &> int8range(5,10) AS RESULT;
 result
--------
 t
(1 row)
```

- -|-

  描述：相邻

  示例：

```
postgres=# SELECT numrange(1.1,2.2) -|- numrange(2.2,3.3) AS RESULT;
 result
--------
 t
(1 row)
```

- +

  描述：并集

  示例：

```
postgres=# SELECT numrange(5,15) + numrange(10,20) AS RESULT;
 result
--------
 [5,20)
(1 row)
```

- *

  描述：交集

  示例：

```
postgres=# SELECT int8range(5,15) * int8range(10,20) AS RESULT;
 result
---------
 [10,15)
```

```
(1 row)
```

- -

  描述：差集

  示例：

```
postgres=# SELECT int8range(5,15) - int8range(10,20) AS RESULT;
 result
--------
 [5,10)
(1 row)
```

  简单的比较操作符<、>、<=和>=先比较下界，只有下界相等时才比较上界。

  <<、>>和-|-操作符当包含空范围时也会返回 false；也就是，不认为空范围在其他范围之前或之后。

  并集和差集操作符的执行结果无法包含两个不相交的子范围。

5.17.2 范围函数

- numrange(numeric, numeric, [text])

  描述：表示一个范围。

  返回类型：范围元素类型

  示例：

```
postgres=# SELECT numrange(1.1,2.2) AS RESULT;
 result
--------
[1.1,2.2)
(1 row)
postgres=# SELECT numrange(1.1,2.2, '()') AS RESULT;
 result
--------
(1.1,2.2)
(1 row)
```

- lower(anyrange)

  描述：范围的下界。

  返回类型：范围元素类型

示例：

```
postgres=# SELECT lower(numrange(1.1,2.2)) AS RESULT;
 result
--------
    1.1
(1 row)
```

- upper(anyrange)

  描述：范围的上界。

  返回类型：范围元素类型

  示例：

```
postgres=# SELECT upper(numrange(1.1,2.2)) AS RESULT;
 result
--------
    2.2
(1 row)
```

- isempty(anyrange)

  描述：范围是否为空。

  返回类型：Boolean

  示例：

```
postgres=# SELECT isempty(numrange(1.1,2.2)) AS RESULT;
 result
--------
 f
(1 row)
```

- lower_inc(anyrange)

  描述：是否包含下界。

  返回类型：Boolean

  示例：

```
postgres=# SELECT lower_inc(numrange(1.1,2.2)) AS RESULT;
 result
--------
 t
```

```
(1 row)
```

- upper_inc(anyrange)

  描述：是否包含上界。

  返回类型：Boolean

  示例：

```
postgres=# SELECT upper_inc(numrange(1.1,2.2)) AS RESULT;
 result
--------
 f
(1 row)
```

- lower_inf(anyrange)

  描述：下界是否为无穷。

  返回类型：Boolean

  示例：

```
postgres=# SELECT lower_inf('(,)'::daterange) AS RESULT;
 result
--------
 t
(1 row)
```

- upper_inf(anyrange)

  描述：上界是否为无穷。

  返回类型：Boolean

  示例：

```
postgres=# SELECT upper_inf('(,)'::daterange) AS RESULT;
 result
--------
 t
(1 row)
```

如果范围是空或者需要的界限是无穷的，lower 和 upper 函数将返回 null。lower_inc、upper_inc、lower_inf 和 upper_inf 函数均对空范围返回 false。

- elem_contained_by_range(anyelement, anyrange)

描述：判断元素是否在范围内。

返回类型：Boolean

示例：

```
postgres=# SELECT elem_contained_by_range('2', numrange(1.1,2.2));
elem_contained_by_range
-------------------------
t
(1 row)
```

5.18 聚集函数

- sum(expression)

  描述：所有输入行的 expression 总和。

  返回类型：

  通常情况下输入数据类型和输出数据类型是相同的，但以下情况会发生类型转换：

  - 对于 SMALLINT 或 INT 输入，输出类型为 BIGINT。
  - 对于 BIGINT 输入，输出类型为 NUMBER 。
  - 对于浮点数输入，输出类型为 DOUBLE PRECISION。

  示例：

```
postgres=# SELECT SUM(ss_ext_tax) FROM public.STORE_SALES;
 sum
--------------
213267594.69
(1 row)
```

- max(expression)

  描述：所有输入行中 expression 的最大值。

  参数类型：任意数组、数值、字符串、日期/时间类型、IPV4 和 IPV6 地址（INET 型和 CIDR 型）。

  返回类型：与参数数据类型相同

  示例：

```
postgres=# SELECT MAX(inv_quantity_on_hand) FROM public.inventory;
  max
---------------
21000000
(1 row)
```

- min(expression)

描述：所有输入行中 expression 的最小值。

参数类型：任意数组、数值、字符串、日期/时间类型、IPV4 和 IPV6 地址（INET 型和 CIDR 型）。

返回类型：与参数数据类型相同

示例：

```
postgres=# SELECT MIN(inv_quantity_on_hand) FROM public.inventory;
 min
-----
   0
(1 row)
```

- avg(expression)

描述：所有输入值的均值（算术平均）。

返回类型：

对于任何整数类型输入，结果都是 NUMBER 类型。对于任何浮点输入，结果都是 DOUBLE PRECISION 类型。否则和输入数据类型相同。

示例：

```
postgres=# SELECT AVG(inv_quantity_on_hand) FROM public.inventory;
        avg
--------------------
 500.0387129084044604
(1 row)
```

- count(expression)

描述：返回表中满足 expression 不为 NULL 的行数。

返回类型：BIGINT

示例：

```
postgres=# SELECT COUNT(inv_quantity_on_hand) FROM public.inventory;
  count
----------
 11158087
(1 row)
```

- count(*)

  描述：返回表中的记录行数。

  返回类型：BIGINT

  示例：

```
postgres=# SELECT COUNT(*) FROM public.inventory;
  count
----------
 11745000
(1 row)
```

- median(expression) [over (query partition clause)]

  描述：返回表达式的中位数，计算时 NULL 将会被 median 函数忽略。可以使用 distinct 关键字排除表达式中的重复记录。输入 expression 的数据类型可以是数值类型（包括 integer、double、bigint 等），也可以是 interval 类型。其他数据类型不支持求取中位数。

  返回类型：double 或 interval 类型

  示例：

```
postgres=# SELECT MEDIAN(id) FROM (values(1), (2), (3), (4), (null)) test(id);
  median
--------
     2.5
(1 row)
```

- array_agg(expression)

  描述：将所有输入值（包括空）连接成一个数组。

  返回类型：参数类型的数组。

  示例：

```
postgres=# SELECT ARRAY_AGG(sr_fee) FROM public.store_returns WHERE
sr_customer_sk = 2;
   array_agg
```

```
---------------
 {22.18,63.21}
(1 row)
```

- string_agg(expression, delimiter)

  描述：将输入值连接成为一个字符串，用分隔符分开。

  返回类型：和参数数据类型相同。

  示例：

```
 postgres=# SELECT string_agg(sr_item_sk, ',') FROM public.store_returns where
sr_item_sk < 3;
         string_agg
----------------------------------------------------------------------------
----
----------------------------

1,2,1,2,2,1,1,2,2,1,2,1,2,1,1,1,2,1,1,1,1,1,2,1,1,1,1,1,2,2,1,1,1,1,1,1,1,1,
2,2,1,1,1,1,1,1,2,2,1,1,2,1,1,1
(1 row)
```

- listagg(expression [, delimiter]) WITHIN GROUP(ORDER BY order-list)

  描述：将聚集列数据按 WITHIN GROUP 指定的排序方式排列，并用 delimiter 指定的分隔符拼接成一个字符串。

  - expression：必选。指定聚集列名或基于列的有效表达式，不支持 DISTINCT 关键字和 VARIADIC 参数。
  - delimiter：可选。指定分隔符，可以是字符串常数或基于分组列的确定性表达式，缺省时表示分隔符为空。
  - order-list：必选。指定分组内的排序方式。

  返回类型：text

  示例：

  聚集列是文本字符集类型。

```
 postgres=# SELECT deptno, listagg(ename, ',') WITHIN GROUP(ORDER BY ename) AS
employees FROM emp GROUP BY deptno;
 deptno |              employees
```

```
--------+-----------------------------------
     10 | CLARK,KING,MILLER
     20 | ADAMS,FORD,JONES,SCOTT,SMITH
     30 | ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD
(3 rows)
```

聚集列是整型。

```
postgres=# SELECT deptno, listagg(mgrno,',') WITHIN GROUP(ORDER BY mgrno NULLS
FIRST) AS mgrnos FROM emp GROUP BY deptno;
 deptno |            mgrnos
--------+------------------------------
     10 | 7782,7839
     20 | 7566,7566,7788,7839,7902
     30 | 7698,7698,7698,7698,7698,7839
(3 rows)
```

聚集列是浮点类型。

```
postgres=# SELECT job, listagg(bonus,'($);') WITHIN GROUP(ORDER BY bonus DESC)
|| '($)' AS bonus FROM emp GROUP BY job;
    job     |                        bonus
------------+-----------------------------------------------
 CLERK      | 10234.21($); 2000.80($); 1100.00($); 1000.22($)
 PRESIDENT  | 23011.88($)
 ANALYST    | 2002.12($); 1001.01($)
 MANAGER    | 10000.01($); 2399.50($); 999.10($)
 SALESMAN   | 1000.01($); 899.00($); 99.99($); 9.00($)
(5 rows)
```

聚集列是时间类型。

```
postgres=# SELECT deptno, listagg(hiredate,',') WITHIN GROUP(ORDER BY hiredate
DESC) AS hiredates FROM emp GROUP BY deptno;
 deptno |                                                              hiredates
--------+------------------------------------------------------------------------
-------------------------------------------------------
     10 | 1982-01-23 00:00:00, 1981-11-17 00:00:00, 1981-06-09 00:00:00
     20 | 2001-04-02 00:00:00, 1999-12-17 00:00:00, 1987-05-23 00:00:00,
1987-04-19 00:00:00, 1981-12-03 00:00:00
     30 | 2015-02-20 00:00:00, 2010-02-22 00:00:00, 1997-09-28 00:00:00,
1981-12-03 00:00:00, 1981-09-08 00:00:00, 1981-05-01 00:00:00
(3 rows)
```

聚集列是时间间隔类型。

```
postgres=# SELECT deptno, listagg(vacationTime, '; ') WITHIN GROUP(ORDER BY
vacationTime DESC) AS vacationTime FROM emp GROUP BY deptno;
 deptno |                                    vacationtime
--------+-----------------------------------------------------------------------
---------------
     10 | 1 year 30 days; 40 days; 10 days
     20 | 70 days; 36 days; 9 days; 5 days
     30 | 1 year 1 mon; 2 mons 10 days; 30 days; 12 days 12:00:00; 4 days 06:00:00;
24:00:00
(3 rows)
```

分隔符缺省时，默认为空。

```
postgres=# SELECT deptno, listagg(job) WITHIN GROUP(ORDER BY job) AS jobs FROM
emp GROUP BY deptno;
 deptno |                      jobs
--------+-----------------------------------------------
     10 | CLERKMANAGERPRESIDENT
     20 | ANALYSTANALYSTCLERKCLERKMANAGER
     30 | CLERKMANAGERSALESMANSALESMANSALESMANSALESMAN
(3 rows)
```

listagg 作为窗口函数时，OVER 子句不支持 ORDER BY 的窗口排序，listagg 列为对应分组的有序聚集。

```
postgres=# SELECT deptno, mgrno, bonus, listagg(ename,'; ') WITHIN GROUP(ORDER
BY hiredate) OVER(PARTITION BY deptno) AS employees FROM emp;
 deptno | mgrno |  bonus   |                  employees
--------+-------+----------+-------------------------------------------
     10 |  7839 | 10000.01 | CLARK; KING; MILLER
     10 |       | 23011.88 | CLARK; KING; MILLER
     10 |  7782 | 10234.21 | CLARK; KING; MILLER
     20 |  7566 |  2002.12 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7566 |  1001.01 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7788 |  1100.00 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7902 |  2000.80 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7839 |   999.10 | FORD; SCOTT; ADAMS; SMITH; JONES
     30 |  7839 |  2399.50 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |     9.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |  1000.22 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |    99.99 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |  1000.01 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |   899.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
```

```
(14 rows)
```

- covar_pop(Y, X)

  描述：总体协方差。

  返回类型：double precision

  示例：

```
postgres=# SELECT COVAR_POP(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
    covar_pop
------------------
 829.749627587403
(1 row)
```

- covar_samp(Y, X)

  描述：样本协方差。

  返回类型：double precision

  示例：

```
postgres=# SELECT COVAR_SAMP(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
    covar_samp
------------------
 830.052235037289
(1 row)
```

- stddev_pop(expression)

  描述：总体标准差。

  返回类型：对于浮点类型的输入返回 double precision，其他输入返回 numeric。

  示例：

```
postgres=# SELECT STDDEV_POP(inv_quantity_on_hand) FROM public.inventory WHERE
inv_warehouse_sk = 1;
    stddev_pop
------------------
 289.224294957556
(1 row)
```

- stddev_samp(expression)

描述：样本标准差。

返回类型：对于浮点类型的输入返回 double precision，其他输入返回 numeric。

示例：

```
postgres=# SELECT STDDEV_SAMP(inv_quantity_on_hand) FROM public.inventory
WHERE inv_warehouse_sk = 1;
   stddev_samp
------------------
 289.224359757315
(1 row)
```

- var_pop(expression)

描述：总体方差（总体标准差的平方）

返回类型：对于浮点类型的输入返回 double precision 类型，其他输入返回 numeric 类型。

示例：

```
postgres=# SELECT VAR_POP(inv_quantity_on_hand) FROM public.inventory WHERE
inv_warehouse_sk = 1;
      var_pop
--------------------
 83650.692793695475
(1 row)
```

- var_samp(expression)

描述：样本方差（样本标准差的平方）

返回类型：对于浮点类型的输入返回 double precision 类型，其他输入返回 numeric 类型。

示例：

```
postgres=# SELECT VAR_SAMP(inv_quantity_on_hand) FROM public.inventory WHERE
inv_warehouse_sk = 1;
      var_samp
--------------------
 83650.730277028768
(1 row)
```

- bit_and(expression)

描述：所有非 NULL 输入值的按位与(AND)，如果全部输入值皆为 NULL，那么结果也为 NULL 。

返回类型：和参数数据类型相同。

示例：

```
postgres=# SELECT BIT_AND(inv_quantity_on_hand) FROM public.inventory WHERE
inv_warehouse_sk = 1;
 bit_and
---------
       0
(1 row)
```

- bit_or(expression)

描述：所有非 NULL 输入值的按位或(OR)，如果全部输入值皆为 NULL，那么结果也为 NULL。

返回类型：和参数数据类型相同

示例：

```
postgres=# SELECT BIT_OR(inv_quantity_on_hand) FROM public.inventory WHERE
inv_warehouse_sk = 1;
 bit_or
--------
   1023
(1 row)
```

- bool_and(expression)

描述：如果所有输入值都是真，则为真，否则为假。

返回类型：bool

示例：

```
postgres=# SELECT bool_and(100 <2500);
 bool_and
----------
 t
(1 row)
```

- bool_or(expression)

描述：如果所有输入值只要有一个为真，则为真，否则为假。

返回类型：bool

示例：

```
postgres=# SELECT bool_or(100 <2500);
 bool_or
----------
 t
(1 row)
```

- corr(Y, X)

  描述：相关系数

  返回类型：double precision

  示例：

```
postgres=# SELECT CORR(sr_fee, sr_net_loss) FROM public.store_returns WHERE
sr_customer_sk < 1000;
       corr
------------------
 .0381383624904186
(1 row)
```

- every(expression)

  描述：等效于 bool_and。

  返回类型：bool

  示例：

```
postgres=# SELECT every(100 <2500);
 every
-------
 t
(1 row)
```

- regr_avgx(Y, X)

  描述：自变量的平均值 (sum(X)/N)

  返回类型：double precision

  示例：

```
postgres=# SELECT REGR_AVGX(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
    regr_avgx
```

```
-----------------
 578.606576740795
(1 row)
```

- regr_avgy(Y, X)

  描述：因变量的平均值 (sum(Y)/N)

  返回类型：double precision

  示例：

```
 postgres=# SELECT REGR_AVGY(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
    regr_avgy
------------------
 50.0136711629602
(1 row)
```

- regr_count(Y, X)

  描述：两个表达式都不为 NULL 的输入行数。

  返回类型：bigint

  示例：

```
 postgres=# SELECT REGR_COUNT(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
 regr_count
------------
       2743
(1 row)
```

- regr_intercept(Y, X)

  描述：根据所有输入的点(X, Y)按照最小二乘法拟合成一个线性方程，然后返回该直线的 Y 轴截距。

  返回类型：double precision

  示例：

```
 postgres=# SELECT REGR_INTERCEPT(sr_fee, sr_net_loss) FROM
public.store_returns WHERE sr_customer_sk < 1000;
  regr_intercept
------------------
```

```
 49.2040847848607
(1 row)
```

- regr_r2(Y, X)

  描述：相关系数的平方。

  返回类型：double precision

  示例：

```
 postgres=# SELECT REGR_R2(sr_fee, sr_net_loss) FROM public.store_returns WHERE
sr_customer_sk < 1000;
      regr_r2
--------------------
 .00145453469345058
(1 row)
```

- regr_slope(Y, X)

  描述：根据所有输入的点(X, Y)按照最小二乘法拟合成一个线性方程， 然后返回该直线的斜率。

  返回类型：double precision

  示例：

```
 postgres=# SELECT REGR_SLOPE(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
     regr_slope
--------------------
 .00139920009665259
(1 row)
```

- regr_sxx(Y, X)

  描述：sum(X^2) - sum(X)^2/N （自变量的“平方和”）

  返回类型：double precision

  示例：

```
 postgres=# SELECT REGR_SXX(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
     regr_sxx
------------------
 1626645991.46135
```

```
(1 row)
```

- regr_sxy(Y, X)

  描述：sum(X*Y) - sum(X) * sum(Y)/N （自变量和因变量的“乘方积”）

  返回类型：double precision

  示例：

```
postgres=# SELECT REGR_SXY(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
     regr_sxy
------------------
 2276003.22847225
(1 row)
```

- regr_syy(Y, X)

  描述：sum(Y^2) - sum(Y)^2/N（因变量的”平方和”）

  返回类型：double precision

  示例：

```
postgres=# SELECT REGR_SYY(sr_fee, sr_net_loss) FROM public.store_returns
WHERE sr_customer_sk < 1000;
    regr_syy
-----------------
 2189417.6547314
(1 row)
```

- stddev(expression)

  描述：stddev_samp 的别名。

  返回类型：对于浮点类型的输入返回 double precision，其他输入返回 numeric。

  示例：

```
postgres=# SELECT STDDEV(inv_quantity_on_hand) FROM public.inventory WHERE
inv_warehouse_sk = 1;
      stddev
------------------
 289.224359757315
(1 row)
```

- variance(expexpression,ression)

描述：var_samp 的别名。

返回类型：对于浮点类型的输入返回 double precision 类型，其他输入返回 numeric 类型。

示例：

```
 postgres=# SELECT VARIANCE(inv_quantity_on_hand) FROM public.inventory WHERE
inv_warehouse_sk = 1;
      variance
--------------------
 83650.730277028768
(1 row)
```

- delta

描述：返回当前行和前一行的差值。

参数：numeric

返回值类型：numeric

checksum(expression)

描述：返回所有输入值的 CHECKSUM 值。使用该函数可以用来验证 GBase 8s 数据库（不支持 GBase 8s 之外的其他数据库）的备份恢复或者数据迁移操作前后表中的数据是否相同。在备份恢复或者数据迁移操作前后都需要用户通过手工执行 SQL 命令的方式获取执行结果，通过对比获取的执行结果判断操作前后表中的数据是否相同。

说明

对于大表，CHECKSUM 函数可能会需要很长时间。

如果某两表的 CHECKSUM 值不同，则表明两表的内容是不同的。由于 CHECKSUM 函数中使用散列函数不能保证无冲突，因此两个不同内容的表可能会得到相同的 CHECKSUM 值，存在这种情况的可能性较小。对于列进行的 CHECKSUM 也存在相同的情况。

- 对于时间类型 timestamp, timestamptz 和 smalldatetime，计算 CHECKSUM 值时请确保时区设置一致。
- 若计算某列的 CHECKSUM 值，且该列类型可以默认转为 TEXT 类型，则 expression 为列名。
- 若计算某列的 CHECKSUM 值，且该列类型不能默认转为 TEXT 类型，则 expression 为列名::TEXT。

- 若计算所有列的 CHECKSUM 值，则 expression 为表名::TEXT。

可以默认转换为 TEXT 类型的类型包括：char、name、 int8、 int2、 int1、 int4、 raw、 pg_node_tree、 float4、 float8、 bpchar、 varchar、 nvarchar、 nvarchar2、 date、 timestamp、 timestamptz、 numeric、 smalldatetime，其他类型需要强制转换为 TEXT。

返回类型：numeric。

示例：

表中可以默认转为 TEXT 类型的某列的 CHECKSUM 值。

```
postgres=# SELECT CHECKSUM(inv_quantity_on_hand) FROM public.inventory;
      checksum
------------------
 24417258945265247
(1 row)
```

表中不能默认转为 TEXT 类型的某列的 CHECKSUM 值。注意此时 CHECKSUM 参数是列名::TEXT。

```
postgres=# SELECT CHECKSUM(inv_quantity_on_hand::TEXT) FROM
public.inventory;
      checksum
------------------
 24417258945265247
(1 row)
```

表中所有列的 CHECKSUM 值。注意此时 CHECKSUM 参数是表名::TEXT，且表名前不加 Schema。

```
postgres=# SELECT CHECKSUM(inventory::TEXT) FROM public.inventory;
      checksum
------------------
 25223696246875800
(1 row)
```

- first(anyelement)

描述：返回排序后第一个非 NULL 值。

返回类型：anyelement

```
postgres=# select * from tba;
name
-----
```

```
A
A
B
D
(4 rows)

postgres=# select first("name" order by "name") as name from tba;
first
-----
A
(1 rows)
```

- last(anyelement)

  描述：返回排序后最后一个非 NULL 值。

  返回类型：anyelement

```
 postgres=# select * from tba;
name
-----
A
A
B
D
(4 rows)

postgres=# select last("name" order by "name") as name from tba;
last
-----
D
(1 rows)
```

- mode() within group (order by value anyelement)

描述：返回某列中出现频率最高的值，如果多个值频率相同，则返回最小的那个值。排序方式和该列类型的默认排序方式相同。其中 value 为输入参数，可以为任意类型。

返回类型：与输入参数类型相同。

示例：

```
 postgres=# select mode() within group (order by value) from (values(1, 'a'),
(2, 'b'), (2, 'c')) v(value, tag);
 mode
```

```
------
    2
(1 row)
postgres=# select mode() within group (order by tag) from (values(1, 'a'), (2,
'b'), (2, 'c')) v(value, tag);
 mode
------
 a
(1 row)
```

- json_agg(any)

  描述：将值聚集为 json 数组。

  返回类型：array-json

  示例：

```
 postgres=# select * from classes;
name | score
-----+-------
A    |     2
A    |     3
D    |     5
D    |
(4 rows)
 postgres=# select name, json_agg(score) score from classes group by name order
by name;
name |      score
-----+-----------------
A    | [2, 3]
D    | [5, null]
(2 rows)
```

- json_object_agg(any, any)

  描述：将值聚集为 json 对象。

  返回类型：object-json

  示例：

```
 postgres=# select * from classes;
name | score
-----+-------
```

```
A    |     2
A    |     3
D    |     5
D    |
(4 rows)
 postgres=# select json_object_agg(name, score) from classes group by name order by name;
json_object_agg
-----------------------
{ "A" : 2, "A" : 3 }
{ "D" : 5, "D" : null }
(2 rows)
```

5.19 窗口函数

列存表目前只支持 rank(expression)和 row_number(expression)两个函数。

窗口函数与 OVER 语句一起使用。OVER 语句用于对数据进行分组，并对组内元素进行排序。窗口函数用于给组内的值生成序号。

说明

窗口函数中的 order by 后面必须跟字段名，若 order by 后面跟数字，该数字会被按照常量处理，因此对目标列没有起到排序的作用。

- RANK()

描述：RANK 函数为各组内值生成跳跃排序序号，其中，相同的值具有相同序号。

返回值类型：BIGINT

示例：

```
 postgres=# SELECT d_moy, d_fy_week_seq, rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM public.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
  d_moy | d_fy_week_seq | rank
-------+---------------+------
      1 |             1 |    1
      1 |             1 |    1
      1 |             1 |    1
      1 |             1 |    1
      1 |             1 |    1
      1 |             1 |    1
```

```
    1 |             1 |    1
    1 |             2 |    8
    1 |             2 |    8
    1 |             2 |    8
    1 |             2 |    8
    1 |             2 |    8
    1 |             2 |    8
    1 |             2 |    8
    1 |             3 |   15
    1 |             3 |   15
    1 |             3 |   15
    1 |             3 |   15
    1 |             3 |   15
    1 |             3 |   15
    1 |             3 |   15
    1 |             4 |   22
    1 |             4 |   22
    1 |             4 |   22
    1 |             4 |   22
    1 |             4 |   22
    1 |             4 |   22
    1 |             4 |   22
    1 |             5 |   29
    1 |             5 |   29
    2 |             5 |    1
    2 |             5 |    1
    2 |             5 |    1
    2 |             5 |    1
    2 |             5 |    1
    2 |             6 |    6
    2 |             6 |    6
    2 |             6 |    6
    2 |             6 |    6
    2 |             6 |    6
    2 |             6 |    6
    2 |             6 |    6
(42 rows)
```

- ROW_NUMBER()

描述：ROW_NUMBER 函数为各组内值生成连续排序序号，其中，相同的值其序号也不相同。

返回值类型：BIGINT

示例：

```
postgres=# SELECT d_moy, d_fy_week_seq, Row_number() OVER(PARTITION BY d_moy
ORDER BY d_fy_week_seq) FROM public.date_dim  WHERE d_moy < 4 AND d_fy_week_seq
< 7 ORDER BY 1,2;
 d_moy | d_fy_week_seq | row_number
-------+---------------+------------
     1 |             1 |          1
     1 |             1 |          2
     1 |             1 |          3
     1 |             1 |          4
     1 |             1 |          5
     1 |             1 |          6
     1 |             1 |          7
     1 |             2 |          8
     1 |             2 |          9
     1 |             2 |         10
     1 |             2 |         11
     1 |             2 |         12
     1 |             2 |         13
     1 |             2 |         14
     1 |             3 |         15
     1 |             3 |         16
     1 |             3 |         17
     1 |             3 |         18
     1 |             3 |         19
     1 |             3 |         20
     1 |             3 |         21
     1 |             4 |         22
     1 |             4 |         23
     1 |             4 |         24
     1 |             4 |         25
     1 |             4 |         26
     1 |             4 |         27
     1 |             4 |         28
     1 |             5 |         29
     1 |             5 |         30
     2 |             5 |          1
     2 |             5 |          2
     2 |             5 |          3
```

```
     2 |             5 |          4
     2 |             5 |          5
     2 |             6 |          6
     2 |             6 |          7
     2 |             6 |          8
     2 |             6 |          9
     2 |             6 |         10
     2 |             6 |         11
     2 |             6 |         12
(42 rows)
```

- DENSE_RANK()

描述：DENSE_RANK 函数为各组内值生成连续排序序号，其中，相同的值具有相同序号。

返回值类型：BIGINT

示例：

```
 postgres=# SELECT d_moy, d_fy_week_seq, dense_rank() OVER(PARTITION BY d_moy
ORDER BY d_fy_week_seq) FROM public.date_dim WHERE d_moy < 4 AND d_fy_week_seq
< 7 ORDER BY 1,2;
 d_moy | d_fy_week_seq | dense_rank
-------+---------------+------------
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             2 |          2
     1 |             2 |          2
     1 |             2 |          2
     1 |             2 |          2
     1 |             2 |          2
     1 |             2 |          2
     1 |             2 |          2
     1 |             2 |          2
     1 |             3 |          3
     1 |             3 |          3
     1 |             3 |          3
     1 |             3 |          3
```

```
     1 |             3 |           3
     1 |             3 |           3
     1 |             3 |           3
     1 |             4 |           4
     1 |             4 |           4
     1 |             4 |           4
     1 |             4 |           4
     1 |             4 |           4
     1 |             4 |           4
     1 |             4 |           4
     1 |             4 |           4
     1 |             5 |           5
     1 |             5 |           5
     2 |             5 |           1
     2 |             5 |           1
     2 |             5 |           1
     2 |             5 |           1
     2 |             5 |           1
     2 |             6 |           2
     2 |             6 |           2
     2 |             6 |           2
     2 |             6 |           2
     2 |             6 |           2
     2 |             6 |           2
     2 |             6 |           2
(42 rows)
```

- PERCENT_RANK()

描述：PERCENT_RANK 函数为各组内对应值生成相对序号，即根据公式 (rank - 1) / (total rows - 1)计算所得的值。其中 rank 为该值依据 RANK 函数所生成的对应序号，totalrows 为该分组内的总元素个数。

返回值类型：DOUBLE PRECISION

示例：

```
postgres=# SELECT d_moy, d_fy_week_seq, percent_rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM public.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
 d_moy | d_fy_week_seq |   percent_rank
-------+---------------+------------------
     1 |             1 |                0
     1 |             1 |                0
```

```
    1 |             1 |                  0
    1 |             1 |                  0
    1 |             1 |                  0
    1 |             1 |                  0
    1 |             1 |                  0
    1 |             2 | .241379310344828
    1 |             2 | .241379310344828
    1 |             2 | .241379310344828
    1 |             2 | .241379310344828
    1 |             2 | .241379310344828
    1 |             2 | .241379310344828
    1 |             2 | .241379310344828
    1 |             3 | .482758620689655
    1 |             3 | .482758620689655
    1 |             3 | .482758620689655
    1 |             3 | .482758620689655
    1 |             3 | .482758620689655
    1 |             3 | .482758620689655
    1 |             3 | .482758620689655
    1 |             4 | .724137931034483
    1 |             4 | .724137931034483
    1 |             4 | .724137931034483
    1 |             4 | .724137931034483
    1 |             4 | .724137931034483
    1 |             4 | .724137931034483
    1 |             4 | .724137931034483
    1 |             5 |  .96551724137931
    1 |             5 |  .96551724137931
    2 |             5 |                  0
    2 |             5 |                  0
    2 |             5 |                  0
    2 |             5 |                  0
    2 |             5 |                  0
    2 |             6 | .454545454545455
    2 |             6 | .454545454545455
    2 |             6 | .454545454545455
    2 |             6 | .454545454545455
    2 |             6 | .454545454545455
    2 |             6 | .454545454545455
    2 |             6 | .454545454545455
(42 rows)
```

- CUME_DIST()

描述：CUME_DIST 函数为各组内对应值生成累积分布序号。即根据公式(小于等于当前值的数据行数)/(该分组总行数 totalrows)计算所得的相对序号。

返回值类型：DOUBLE PRECISION

示例：

```
postgres=# SELECT d_moy, d_fy_week_seq, cume_dist() OVER(PARTITION BY d_moy
ORDER BY d_fy_week_seq) FROM public.date_dim e_dim WHERE d_moy < 4 AND
d_fy_week_seq < 7 ORDER BY 1,2;
 d_moy | d_fy_week_seq |    cume_dist
-------+---------------+-----------------
     1 |             1 | .233333333333333
     1 |             1 | .233333333333333
     1 |             1 | .233333333333333
     1 |             1 | .233333333333333
     1 |             1 | .233333333333333
     1 |             1 | .233333333333333
     1 |             1 | .233333333333333
     1 |             2 | .466666666666667
     1 |             2 | .466666666666667
     1 |             2 | .466666666666667
     1 |             2 | .466666666666667
     1 |             2 | .466666666666667
     1 |             2 | .466666666666667
     1 |             2 | .466666666666667
     1 |             3 |               .7
     1 |             3 |               .7
     1 |             3 |               .7
     1 |             3 |               .7
     1 |             3 |               .7
     1 |             3 |               .7
     1 |             3 |               .7
     1 |             4 | .933333333333333
     1 |             4 | .933333333333333
     1 |             4 | .933333333333333
     1 |             4 | .933333333333333
     1 |             4 | .933333333333333
     1 |             4 | .933333333333333
     1 |             4 | .933333333333333
     1 |             5 |                 1
```

```
    1 |              5 |                 1
    2 |              5 | .416666666666667
    2 |              5 | .416666666666667
    2 |              5 | .416666666666667
    2 |              5 | .416666666666667
    2 |              5 | .416666666666667
    2 |              6 |                 1
    2 |              6 |                 1
    2 |              6 |                 1
    2 |              6 |                 1
    2 |              6 |                 1
    2 |              6 |                 1
    2 |              6 |                 1
(42 rows)
```

- NTILE(num_buckets integer)

描述：NTILE 函数根据 num_buckets integer 将有序的数据集合平均分配到 num_buckets 所指定数量的桶中，并将桶号分配给每一行。分配时应尽量做到平均分配。

返回值类型：INTEGER

示例：

```
postgres=# SELECT d_moy, d_fy_week_seq, ntile(3) OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM public.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
 d_moy | d_fy_week_seq | ntile
-------+---------------+-------
     1 |             1 |     1
     1 |             1 |     1
     1 |             1 |     1
     1 |             1 |     1
     1 |             1 |     1
     1 |             1 |     1
     1 |             1 |     1
     1 |             2 |     1
     1 |             2 |     1
     1 |             2 |     1
     1 |             2 |     2
     1 |             2 |     2
     1 |             2 |     2
     1 |             2 |     2
```

```
     1 |             3 |     2
     1 |             3 |     2
     1 |             3 |     2
     1 |             3 |     2
     1 |             3 |     2
     1 |             3 |     2
     1 |             3 |     3
     1 |             4 |     3
     1 |             4 |     3
     1 |             4 |     3
     1 |             4 |     3
     1 |             4 |     3
     1 |             4 |     3
     1 |             4 |     3
     1 |             5 |     3
     1 |             5 |     3
     2 |             5 |     1
     2 |             5 |     1
     2 |             5 |     1
     2 |             5 |     1
     2 |             5 |     2
     2 |             6 |     2
     2 |             6 |     2
     2 |             6 |     2
     2 |             6 |     3
     2 |             6 |     3
     2 |             6 |     3
     2 |             6 |     3
(42 rows)
```

- LAG(value any [, offset integer [, default any]])

描述：LAG 函数为各组内对应值生成滞后值。即当前值对应的行数往前偏移 offset 位后所得行的 value 值作为序号。若经过偏移后行数不存在，则对应结果取为 default 值。若无指定，在默认情况下，offset 取为 1，default 值取为 NULL。

返回值类型：与参数数据类型相同。

示例：

```
postgres=# SELECT d_moy, d_fy_week_seq, lag(d_moy,3,null) OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM public.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
```

d_moy	d_fy_week_seq	lag
1	1	
1	1	
1	1	
1	1	1
1	1	1
1	1	1
1	1	1
1	2	1
1	2	1
1	2	1
1	2	1
1	2	1
1	2	1
1	2	1
1	3	1
1	3	1
1	3	1
1	3	1
1	3	1
1	3	1
1	3	1
1	4	1
1	4	1
1	4	1
1	4	1
1	4	1
1	4	1
1	5	1
1	5	1
2	5	
2	5	
2	5	
2	5	2
2	5	2
2	6	2
2	6	2
2	6	2
2	6	2
2	6	2

```
     2 |              6 |    2
     2 |              6 |    2
(42 rows)
```

- LEAD(value any [, offset integer [, default any]])

描述：LEAD 函数为各组内对应值生成提前值。即当前值对应的行数向后偏移 offset 位后所得行的 value 值作为序号。若经过向后偏移后行数超过当前组内的总行数，则对应结果取为 default 值。若无指定，在默认情况下，offset 取为 1，default 值取为 NULL。

返回值类型：与参数数据类型相同。

示例：

```
 postgres=# SELECT d_moy, d_fy_week_seq, lead(d_fy_week_seq,2) OVER(PARTITION
BY d_moy ORDER BY d_fy_week_seq) FROM  public.date_dim WHERE d_moy < 4 AND
d_fy_week_seq < 7 ORDER BY 1,2;
d_moy | d_fy_week_seq | lead
-------+---------------+------
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    2
     1 |              1 |    2
     1 |              2 |    2
     1 |              2 |    2
     1 |              2 |    2
     1 |              2 |    2
     1 |              2 |    2
     1 |              2 |    3
     1 |              2 |    3
     1 |              3 |    3
     1 |              3 |    3
     1 |              3 |    3
     1 |              3 |    3
     1 |              3 |    3
     1 |              3 |    4
     1 |              3 |    4
     1 |              4 |    4
     1 |              4 |    4
     1 |              4 |    4
```

```
     1 |             4 |     4
     1 |             4 |     4
     1 |             4 |     5
     1 |             4 |     5
     1 |             5 |
     1 |             5 |
     2 |             5 |     5
     2 |             5 |     5
     2 |             5 |     5
     2 |             5 |     6
     2 |             5 |     6
     2 |             6 |     6
     2 |             6 |     6
     2 |             6 |     6
     2 |             6 |     6
     2 |             6 |     6
     2 |             6 |
     2 |             6 |
(42 rows)
```

- FIRST_VALUE(value any)

  描述：FIRST_VALUE 函数取各组内的第一个值作为返回结果。

  返回值类型：与参数数据类型相同。

  示例：

```
 postgres=# SELECT d_moy, d_fy_week_seq, first_value(d_fy_week_seq)
OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM public.date_dim WHERE d_moy
< 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
 d_moy | d_fy_week_seq | first_value
-------+---------------+-------------
     1 |             1 |           1
     1 |             1 |           1
     1 |             1 |           1
     1 |             1 |           1
     1 |             1 |           1
     1 |             1 |           1
     1 |             1 |           1
     1 |             2 |           1
     1 |             2 |           1
     1 |             2 |           1
     1 |             2 |           1
```

```
     1 |             2 |             1
     1 |             2 |             1
     1 |             2 |             1
     1 |             3 |             1
     1 |             3 |             1
     1 |             3 |             1
     1 |             3 |             1
     1 |             3 |             1
     1 |             3 |             1
     1 |             3 |             1
     1 |             3 |             1
     1 |             4 |             1
     1 |             4 |             1
     1 |             4 |             1
     1 |             4 |             1
     1 |             4 |             1
     1 |             4 |             1
     1 |             4 |             1
     1 |             5 |             1
     1 |             5 |             1
     2 |             5 |             5
     2 |             5 |             5
     2 |             5 |             5
     2 |             5 |             5
     2 |             5 |             5
     2 |             6 |             5
     2 |             6 |             5
     2 |             6 |             5
     2 |             6 |             5
     2 |             6 |             5
     2 |             6 |             5
     2 |             6 |             5
(42 rows)
```

- LAST_VALUE(value any)

  描述：LAST_VALUE 函数取各组内的最后一个值作为返回结果。

  返回值类型：与参数数据类型相同。

  示例：

```
postgres=# SELECT d_moy, d_fy_week_seq, last_value(d_moy) OVER(PARTITION BY
d_moy ORDER BY d_fy_week_seq) FROM public.date_dim WHERE d_moy < 4 AND
d_fy_week_seq < 6 ORDER BY 1,2;
```

```
 d_moy | d_fy_week_seq | last_value
-------+---------------+------------
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             1 |          1
     1 |             2 |          1
     1 |             2 |          1
     1 |             2 |          1
     1 |             2 |          1
     1 |             2 |          1
     1 |             2 |          1
     1 |             2 |          1
     1 |             2 |          1
     1 |             3 |          1
     1 |             3 |          1
     1 |             3 |          1
     1 |             3 |          1
     1 |             3 |          1
     1 |             3 |          1
     1 |             3 |          1
     1 |             4 |          1
     1 |             4 |          1
     1 |             4 |          1
     1 |             4 |          1
     1 |             4 |          1
     1 |             4 |          1
     1 |             4 |          1
     1 |             5 |          1
     1 |             5 |          1
     2 |             5 |          2
     2 |             5 |          2
     2 |             5 |          2
     2 |             5 |          2
     2 |             5 |          2
(35 rows)
```

- NTH_VALUE(value any, nth integer)

描述：NTH_VALUE 函数返回该组内的第 nth 行作为结果。若该行不存在，则默认返回 NULL。

返回值类型：与参数数据类型相同。

示例：

```
postgres=# SELECT d_moy, d_fy_week_seq, nth_value(d_fy_week_seq,6)
OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM public.date_dim WHERE d_moy
< 4 AND d_fy_week_seq < 6 ORDER BY 1,2;
 d_moy | d_fy_week_seq | nth_value
-------+---------------+-----------
     1 |             1 |         1
     1 |             1 |         1
     1 |             1 |         1
     1 |             1 |         1
     1 |             1 |         1
     1 |             1 |         1
     1 |             1 |         1
     1 |             2 |         1
     1 |             2 |         1
     1 |             2 |         1
     1 |             2 |         1
     1 |             2 |         1
     1 |             2 |         1
     1 |             2 |         1
     1 |             3 |         1
     1 |             3 |         1
     1 |             3 |         1
     1 |             3 |         1
     1 |             3 |         1
     1 |             3 |         1
     1 |             4 |         1
     1 |             4 |         1
     1 |             4 |         1
     1 |             4 |         1
     1 |             4 |         1
     1 |             4 |         1
     1 |             5 |         1
     1 |             5 |         1
```

```
     2 |              5 |
     2 |              5 |
     2 |              5 |
     2 |              5 |
     2 |              5 |
(35 rows)
```

5.20 安全函数

- gs_encrypt_aes128(encryptstr,keystr)

描述：以 keystr 为密钥对 encryptstr 字符串进行加密，返回加密后的字符串。keystr 的长度范围为 8~16 字节，至少包含 3 种字符（大写字母、小写字母、数字、特殊字符）。

返回值类型：text

返回值长度：至少为 92 字节，不超过 4*[(Len+68)/3]字节，其中 Len 为加密前数据长度（单位为字节）。

示例：

```
postgres=# SELECT gs_encrypt_aes128('MPPDB','Asdf1234');
                                      gs_encrypt_aes128
------------------------------------------------------------------------------
----------------


PkKJ0I+o6V83IXB2MbfS34amruD+5hrzsD/IQEU7HL0XfpAb1rfEvkjtKgoypGs7h1JfPpqc20EGx
CpHRQHfexdkn08=
(1 row)
```

说明

由于该函数的执行过程需要传入解密口令，为了安全起见，gsql 工具不会将该函数记录入执行历史。即无法在 gsql 里通过上下翻页功能找到该函数的执行历史。

- gs_encrypt(encryptstr,keystr,encrypttype)

描述：根据 encrypttype，以 keystr 为密钥对 encryptstr 字符串进行加密，返回加密后的字符串。keystr 的长度范围为 8~16 字节，至少包含 3 种字符（大写字母、小写字母、数字、特殊字符），encrypttype 可以是 aes128 或 sm4。

返回值类型：text

示例：

```
postgres=# SELECT gs_encrypt('MPPDB','Asdf1234','sm4');
          gs_encrypt
------------------------------
 UFrm2vuNFSEbDAmFObfNIqW7dbCa
(1 row)
```

说明

由于该函数的执行过程需要传入解密口令，为了安全起见，gsql 工具不会将该函数记录入执行历史。即无法在 gsql 里通过上下翻页功能找到该函数的执行历史。

- gs_decrypt_aes128(decryptstr,keystr)

描述：以 keystr 为密钥对 decrypt 字符串进行解密，返回解密后的字符串。解密使用的 keystr 必须保证与加密时使用的 keystr 一致才能正常解密。keystr 不得为空。

说明

此参数需要结合 gs_encrypt_aes128 加密函数共同使用。

返回值类型：text

示例：

```
-- gs_decrypt_aes128 第一个输入参数为 gs_encrypt_aes128 输出参数；第二个输入参数为 gs_encrypt_aes128 第二个输入参数
postgres=# select
gs_decrypt_aes128('PkKJOI+o6V83IXB2MbfS34amruD+5hrzsD/IQEU7HLOXfpAb1rfEvkjtKgoypGs7hlJfPpqc20EGxCpHRQHfexdknO8=','Asdf1234');
 gs_decrypt_aes128
------------------
 MPPDB
(1 row)
```

说明

由于该函数的执行过程需要传入解密口令，为了安全起见，gsql 工具不会将该函数记录入执行历史；即无法在 gsql 里通过上下翻页功能找到该函数的执行历史。

- gs_decrypt(decryptstr,keystr,decrypttype)

描述：根据 decrypttype，以 keystr 为密钥对 decrypt 字符串进行解密，返回解密后的字符串。解密使用的 decrypttype 及 keystr 必须保证与加密时使用的 encrypttype 及 keystr 一致才能正常解密。keystr 不得为空。decrypttype 可以是 aes128 或 sm4。

此函数需要结合 gs_encrypt 加密函数共同使用。

返回值类型：text

示例：

```
    postgres=# select
gs_decrypt('ZBzOmaGA4Bb+coyucJOB8AkIShqc','Asdf1234','sm4');
   gs_decrypt
  ------------
   MPPDB
  (1 row)
```

说明

由于该函数的执行过程需要传入解密口令，为了安全起见，gsql 工具不会将该函数记录入执行历史；即无法在 gsql 里通过上下翻页功能找到该函数的执行历史。

- gs_password_deadline

  描述：显示当前帐户密码离过期还距离多少天。

  返回值类型：interval

  示例：

```
postgres=# SELECT gs_password_deadline();
 gs_password_deadline
-----------------------
 72 days 16:16:56.315324
(1 row)
```

- gs_password_notifytime

  描述：显示帐户密码到期前提醒的天数。

  返回值类型：int32

- login_audit_messages

  描述：查看登录用户的登录信息。

  返回值类型：元组

  示例：

  查看上一次登录认证通过的日期、时间和 IP 等信息。

```
postgres=# select * from login_audit_messages(true);
```

```
 username | database |       logintime        |    mytype     | result |
client_conninfo
----------+----------+------------------------+---------------+--------+-----
-------------
 gbase    | postgres | 2022-05-18 17:08:46+08 | login_success | ok     |
gs_ctl@[local_ip]
```

查看上一次登录认证失败的日期、时间和 IP 等信息。

```
 postgres=# select * from login_audit_messages(false) ORDER BY logintime desc
limit 1;
username | database | logintime | mytype | result | client_conninfo
----------+----------+-----------+--------+--------+-----------------
(0 rows)
```

查看自从最后一次认证通过以来失败的尝试次数、日期和时间。

```
 postgres=# select * from login_audit_messages(false);
 username | database |       logintime        |    mytype     | result |
client_conninfo
----------+----------+------------------------+---------------+--------+------
-------------
(0 rows)
```

- login_audit_messages_pid

描述：查看登录用户的登录信息。与 login_audit_messages 的区别在于结果基于当前 backendid 向前查找。所以不会因为同一用户的后续登录，而影响本次登录的查询结果。也就是查询不到该用户后续登录的信息。

返回值类型：元组

示例：

查看上一次登录认证通过的日期、时间和 IP 等信息。

```
 postgres=# SELECT * FROM login_audit_messages_pid(true);
username | database |       logintime        |    mytype     | result |
client_conninfo  |    backendid
----------+----------+------------------------+---------------+--------+-----
------------+----------------
 gbase    | postgres | 2022-05-18 17:02:34+08 | login_success | ok     |
gs_ctl@10.0.7.16 | 140573725284096
(1 row)
```

查看上一次登录认证失败的日期、时间和 IP 等信息。

```
postgres=# SELECT * FROM login_audit_messages_pid(false) ORDER BY logintime desc
limit 1;
username | database | logintime | mytype | result | client_conninfo | backendid
----------+----------+-----------+--------+--------+-----------------+-------
----
(0 rows)
```

查看自从最后一次认证通过以来失败的尝试次数、日期和时间。

```
postgres=# SELECT * FROM login_audit_messages_pid(false);
username | database | logintime | mytype | result | client_conninfo | backendid
----------+----------+-----------+--------+--------+-----------------+-------
----
(0 rows)
```

- inet_server_addr

  描述：显示服务器 IP 信息。

  返回值类型：inet

  示例：

```
postgres=# SELECT inet_server_addr();
 inet_server_addr
------------------
 10.10.0.13
(1 row)
```

  说明

  上面是以客户端在 10.10.0.50 上，服务器端在 10.10.0.13 上为例。

  如果是通过本地连接，使用此接口显示为空。

- inet_client_addr

  描述：显示客户端 IP 信息。

  返回值类型：inet

  示例：

```
postgres=# SELECT inet_client_addr();
 inet_client_addr
------------------
 10.10.0.50
(1 row)
```

说明

上面是以客户端在 10.10.0.50 上，服务器端在 10.10.0.13 上为例。

如果是通过本地连接，使用此接口显示为空。

- pg_query_audit

  描述：查看数据库主节点审计日志。

  返回值类型：record

  函数返回字段如下：

名称	类型	描述
time	timestamp with time zone	操作时间
type	text	操作类型
result	text	操作结果
userid	oid	用户 id
username	text	执行操作的用户名
database	text	数据库名称
client_conninfo	text	客户端连接信息
object_name	text	操作对象名称
detail_info	text	执行操作详细信息
node_name	text	节点名称
thread_id	text	线程 id
local_port	text	本地端口
remote_port	text	远端端口

- pg_delete_audit

描述：删除指定时间段的审计日志。

返回值类型：void

5.21 账本数据库的函数

- ledger_hist_check(text, text)

  描述：校验指定防篡改用户表的表级数据 hash 值与其对应历史表 hash 一致性。

  参数类型：text

  返回值类型：Boolean

- ledger_hist_repair(text, text)

  描述：修复指定防篡改用户表对应的历史表 hash 值，使之与用户表 hash 一致，返回 hash 差值。

  参数类型：text

  返回值类型：hash16

- ledger_hist_archive(text, text)

  描述：归档指定防篡改用户表对应的历史表至审计日志目录中 hist_back 文件夹下。

  参数类型：text

  返回值类型：Boolean

- ledger_gchain_check(text, text)

  描述：校验指定防篡改用户表对应的历史表 hash 与全局历史表对应的 relhash 一致性。

  参数类型：text

  返回值类型：Boolean

- ledger_gchain_repair(text, text)

  描述：修复验指定防篡改用户表在全局历史表中的 relhash，使之与其历史表 hash 一致，返回 hash 差值。

  参数类型：text

  返回值类型：hash16

- ledger_gchain_archive(void)

  描述：归档全局历史表至审计日志目录中 hist_back 文件夹下。

  参数类型：void

  返回值类型：Boolean

- hash16in(cstring)

  描述：将输入 16 进制字符串转化成内部 hash16 形式。

  参数类型：cstring

  返回值类型：hash16

- hash16out(hash16)

  描述：将内部 hash16 类型的数据转码转化为 16 进制 cstring 类型。

  参数类型：hash16

  返回值类型：cstring

- hash32in(cstring)

  描述：将输入 16 进制字符串（32 个字符）转化成内部类型 hash32 形式。

  参数类型：cstring

  返回值类型：hash32

- hash32out(hash32)

  描述：将内部 hash32 类型的数据转码转化为 16 进制 cstring 类型。

  参数类型：cstring

  返回值类型：hash32

5.22 密态等值函数

- byteawithoutorderwithequalcolin(cstring)

  描述：将输入转码转化成内部 byteawithoutorderwithequalcol 形式。

  参数类型：cstring

返回值类型：byteawithoutorderwithequalcol

- byteawithoutorderwithequalcolout(byteawithoutorderwithequalcol)

  描述：将内部 byteawithoutorderwithequalcol 类型的数据转码转化为 cstring 类型。

  参数类型：byteawithoutorderwithequalcol

  返回值类型：cstring

- byteawithoutorderwithequalcolsend(byteawithoutorderwithequalcol)

  描述：将 byteawithoutorderwithequalcol 类型的数据转码转化为 bytea 类型。

  参数类型：byteawithoutorderwithequalcol

  返回值类型：bytea

- byteawithoutorderwithequalcolrecv(internal)

  描述：将 byteawithoutorderwithequalcol 类型的数据转码转化为 byteawithoutorderwithequalcol 类型。

  参数类型：internal

  返回值类型：byteawithoutorderwithequalcol

- byteawithoutorderwithequalcoltypmodin(_cstring)

  描述：将 byteawithoutorderwithequalcol 类型的数据转码转化为 byteawithoutorderwithequalcol 类型。

  参数类型：_cstring

  返回值类型：int4

- byteawithoutorderwithequalcoltypmodout(int4)

  描述：将 int4 类型的数据转码转化为 cstring 类型。

  参数类型：int4

  返回值类型：cstring

- byteawithoutordercolin(cstring)

  描述：将输入转码转化成内部 byteawithoutordercolin 形式。

参数类型：cstring

返回值类型：byteawithoutordercol

- byteawithoutordercolout(byteawithoutordercol)

  描述：将内部 byteawithoutordercol 类型的数据转码转化为 cstring 类型。

  参数类型：byteawithoutordercol

  返回值类型：cstring

- byteawithoutordercolsend(byteawithoutordercol)

  描述：将 byteawithoutordercol 类型的数据转码转化为 bytea 类型。

  参数类型：byteawithoutordercol

  返回值类型：bytea

- byteawithoutordercolrecv(internal)

  描述：将 byteawithoutordercol 类型的数据转码转化为 byteawithoutordercol 类型。

  参数类型：internal

  返回值类型：byteawithoutordercol

- byteawithoutorderwithequalcolcmp(byteawithoutorderwithequalcol, byteawithoutorderwithequalcol)

  描述：比较两个 byteawithoutorderwithequalcol 类型的数据大小，若第一个参数小于第二个参数，返回-1，若等于，返回 0，若大于，则返回 1。

  参数类型：byteawithoutorderwithequalcol, byteawithoutorderwithequalcol

  返回值类型：int4

- byteawithoutorderwithequalcolcmpbytear(byteawithoutorderwithequalcol, bytea)

  描述：比较 byteawithoutorderwithequalcol 和 bytea 数据大小，若第一个参数小于第二个参数，返回-1，若等于，返回 0，若大于，则返回 1。

  参数类型：byteawithoutorderwithequalcol, bytea

  返回值类型：int4

- byteawithoutorderwithequalcolcmpbyteal(bytea, byteawithoutorderwithequalcol)

描述：比较 bytea 和 byteawithoutorderwithequalcol 数据大小，若第一个参数小于第二个参数，返回-1，若等于，返回 0，若大于，则返回 1。

参数类型：byteawithoutorderwithequalcol, bytea

返回值类型：int4

- byteawithoutorderwithequalcoleq(byteawithoutorderwithequalcol, byteawithoutorderwithequalcol)

描述：比较两个 byteawithoutorderwithequalcol 类型的数据是否相同，相同则返回 true，否则返回 false。

参数类型：byteawithoutorderwithequalcol, bytea

返回值类型：bool

- byteawithoutorderwithequalcoleqbyteal(bytea, byteawithoutorderwithequalcol)

描述：比较 bytea 和 byteawithoutorderwithequalcol 数据是否相同，相同则返回 true，否则返回 false。

参数类型：bytea，byteawithoutorderwithequalcol

返回值类型：bool

- byteawithoutorderwithequalcoleqbytear(byteawithoutorderwithequalcol, bytea)

描述：比较 byteawithoutorderwithequalcol 和 bytea 数据是否相同，相同则返回 true，否则返回 false。

参数类型：byteawithoutorderwithequalcol, bytea

返回值类型：bool

- byteawithoutorderwithequalcolne(byteawithoutorderwithequalcol, byteawithoutorderwithequalcol)

描述：比较两个 byteawithoutorderwithequalcol 类型的数据是否不相同，不相同则返回 true，否则返回 false。

参数类型：byteawithoutorderwithequalcol, byteawithoutorderwithequalcol

返回值类型：bool

- byteawithoutorderwithequalcolnebyteal(bytea, byteawithoutorderwithequalcol)

描述：比较 bytea 和 byteawithoutorderwithequalcol 数据是否相同，相同则返回 true，否则返回 false。

参数类型：bytea，byteawithoutorderwithequalcol

返回值类型：bool

- byteawithoutorderwithequalcolnebytear(byteawithoutorderwithequalcol, bytea)

描述：比较 byteawithoutorderwithequalcol 和 bytea 数据是否不相同，相同则返回 true，否则返回 false。

参数类型：byteawithoutorderwithequalcol, bytea

返回值类型：bool

- hll_hash_byteawithoutorderwithequalcol(byteawithoutorderwithequalcol)

描述：返回 byteawithoutorderwithequalcol 的 hll 哈希值。

参数类型：byteawithoutorderwithequalcol

返回值类型：hll_hashval

由于 byteawithoutorderwithequalcolin 的实现会对 cek 进行查找，并且判断是否为正常加密后的数据类型。因此如果用户输入数据的格式不为加密后的数据格式，并且在本地不存在对应 cek 的情况下，会返回错误。

```
postgres=# SELECT * FROM
byteawithoutorderwithequalcolsend('\x907219912381298461289346129'::byteawith
outorderwithequalcol);
ERROR:  cek with OID 596711794 not found
LINE 1: SELECT * FROM byteawithoutorderwithequalcolsend('\x907219912...
                                                        ^

postgres=# SELECT * FROM
byteawithoutordercolout('\x907219019999999999999912381298461289346129');
ERROR:  cek with OID 2566986098 not found
LINE 1: SELECT * FROM byteawithoutordercolout('\x9072190199999999999...
                                              ^

postgres=# SELECT * FROM
byteawithoutorderwithequalcolrecv('\x907219019999999999999912381298461289346129'::byteawithoutorderwithequalcol);
ERROR:  cek with OID 2566986098 not found
```

```
LINE 1: SELECT * FROM byteawithoutorderwithequalcolrecv('\x907219019...
                                                        ^

postgres=# SELECT * FROM
byteawithoutorderwithequalcolsend('\x907219019999999999999912381298461289346129'::byteawithoutorderwithequalcol);
ERROR:  cek with OID 2566986098 not found
LINE 1: SELECT * FROM byteawithoutorderwithequalcolsend('\x907219019...
                                                        ^
```

5.23 返回集合的函数

5.23.1 序列号生成函数

- generate_series(start, stop)

  描述：生成一个数值序列，从 start 到 stop，步长为 1。

  参数类型：int、bigint、numeric

  返回值类型：setof int、setof bigint、setof numeric（与参数类型相同）

- generate_series(start, stop, step)

  描述：生成一个数值序列，从 start 到 stop，步长为 step。

  参数类型：int、bigint、numeric

  返回值类型：setof int、setof bigint、setof numeric（与参数类型相同）

- generate_series(start, stop, step interval)

  描述：生成一个数值序列，从 start 到 stop，步长为 step。

  参数类型：timestamp 或 timestamp with time zone

  返回值类型：setof timestamp 或 setof timestamp with time zone（与参数类型相同）

如果 step 是正数且 start 大于 stop，则返回零行。相反，如果 step 是负数且 start 小于 stop，则也返回零行。如果输入是 NULL，同样产生零行。如果 step 为零则是一个错误。

示例：

```
postgres=# SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
```

```
               3
               4
(3 rows)
postgres=# SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)
postgres=# SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)
--这个示例应用于 date-plus-integer 操作符。
postgres=# SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS
s(a);
   dates
------------
 2022-05-18
 2022-05-25
 2022-06-01
(3 rows)
postgres=# SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
'2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

5.23.2 下标生成函数

- generate_subscripts(array anyarray, dim int)

描述：生成一系列包括给定数组的下标。

返回值类型：setof int

- generate_subscripts(array anyarray, dim int, reverse boolean)

描述：生成一系列包括给定数组的下标。当 reverse 为真时，该系列则以相反的顺序返回。

返回值类型：setof int

generate_subscripts 是一个为给定数组中的指定维度生成有效下标集的函数。如果数组中没有所请求的维度或者 NULL 数组，返回零行 (但是会给数组元素为空的返回有效下标)。

示例：

```
 --基本用法。
postgres=# SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)
 --unnest 一个 2D 数组。
postgres=# CREATE OR REPLACE FUNCTION unnest2(anyarray)
gbase-# RETURNS SETOF anyelement AS $$
gbase$# SELECT $1[i][j]
gbase$# FROM generate_subscripts($1,1) g1(i),
gbase$# generate_subscripts($1,2) g2(j);
gbase$# $$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
postgres=# SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
--删除函数。
postgres=# DROP FUNCTION unnest2;
DROP FUNCTION
```

5.24 条件表达式函数

- coalesce(expr1, expr2, …, exprn)

描述：返回参数列表中第一个非 NULL 的参数值。COALESCE(expr1, expr2) 等价于 CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END。

示例：

```
postgres=# SELECT coalesce(NULL,'hello');
coalesce
----------
hello
(1 row)
```

如果表达式列表中的所有表达式都等于 NULL，则本函数返回 NULL。

该函数常用于在显示数据时用缺省值替换 NULL。和 CASE 表达式一样，COALESCE 不会计算不需要用来判断结果的参数；即在第一个非空参数右边的参数不会被计算。

- decode(base_expr, compare1, value1, Compare2,value2, … default)

描述：把 base_expr 与后面的每个 compare(n) 进行比较，如果匹配返回相应的 value(n)。如果没有发生匹配，则返回 default。

示例：

```
postgres=# SELECT decode('A','A',1,'B',2,0);
case
------
1
(1 row)
```

- nullif(expr1, expr2)

描述：当且仅当 expr1 和 expr2 相等时，NULLIF 才返回 NULL，否则它返回 expr1。

nullif(expr1, expr2) 逻辑上等价于 CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END。

示例：

```
postgres=# SELECT nullif('hello','world');
nullif
--------
```

```
 hello
(1 row)
```

如果两个参数的数据类型不同，则：

- 若两种数据类型之间存在隐式转换，则以其中优先级较高的数据类型为基准将另一个参数隐式转换成该类型，转换成功则进行计算，转换失败则返回错误。如：

```
 postgres=# SELECT nullif('1234'::VARCHAR,123::INT4);
 nullif
--------
   1234
(1 row)
 postgres=# SELECT nullif('1234'::VARCHAR,'2012-12-24'::DATE);
ERROR:  invalid input syntax for type timestamp: "1234"
CONTEXT:  referenced column: nullif
```

- 若两种数据类型之间不存在隐式转换，则返回错误 。如：

```
 postgres=# SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE);
ERROR:  operator does not exist: boolean = timestamp without time zone
LINE 1: SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE) FROM sys_dummy;
^
HINT:  No operator matches the given name and argument type(s). You might need
to add explicit type casts.
CONTEXT:  referenced column: nullif
```

- nvl(expr1 , expr2)

描述：

- 如果 expr1 为 NULL，则返回 expr2。
- 如果 expr1 非 NULL，则返回 expr1。

示例：

```
 postgres=# SELECT nvl('hello','world');
  nvl
-------
 hello
(1 row)
```

参数 expr1 和 expr2 可以为任意类型，当 NVL 的两个参数不属于同类型时，看第二个参数是否可以向第一个参数进行隐式转换。如果可以则返回第一个参数类型，否则返回错误。

- greatest(expr1 [, …])

  描述：获取并返回参数列表中值最大的表达式的值。

  返回值类型：

  示例：

```
postgres=# SELECT greatest(1*2,2-3,4-1);
 greatest
----------
        3
(1 row)
postgres=# SELECT greatest('HARRY', 'HARRIOT', 'HAROLD');
 greatest
----------
 HARRY
(1 row)
```

- least(expr1 [, …])

  描述：获取并返回参数列表中值最小的表达式的值。

  示例：

```
postgres=# SELECT least(1*2,2-3,4-1);
 least
-------
    -1
(1 row)
postgres=# SELECT least('HARRY','HARRIOT','HAROLD');
 least
--------
 HAROLD
(1 row)
```

- EMPTY_BLOB()

  描述：使用 EMPTY_BLOB 在 INSERT 或 UPDATE 语句中初始化一个 BLOB 变量，取值为 NULL。

  返回值类型：BLOB

  示例：

```
--新建表
```

```
postgres=# CREATE TABLE blob_tb(b blob,id int);
CREATE TABLE
--插入数据
postgres=# INSERT INTO blob_tb VALUES (empty_blob(),1);
INSERT 0 1
--删除表
postgres=# DROP TABLE blob_tb;
DROP TABLE
```

使用 DBE_LOB.GET_LENGTH 求得的长度为 0。

5.25 系统信息函数

5.25.1 会话信息函数

- current_catalog

  描述：当前数据库的名称（在标准 SQL 中称“catalog”）。

  返回值类型：name

  示例：

```
postgres=# SELECT current_catalog;
 current_database
------------------
 postgres
(1 row)
```

- current_database()

  描述：当前数据库的名称。

  返回值类型：name

  示例：

```
postgres=# SELECT current_database();
 current_database
------------------
 postgres
(1 row)
```

- current_query()

  描述：由客户端提交的当前执行语句（可能包含多个声明）。

返回值类型：text

示例：

```
postgres=# SELECT current_query();
     current_query
------------------------
 SELECT current_query();
(1 row)
```

- current_schema[()]

描述：当前模式的名称。

返回值类型：name

示例：

```
postgres=# SELECT current_schema();
 current_schema
----------------
 public
(1 row)
```

备注：current_schema 返回在搜索路径中第一个顺位有效的模式名。（如果搜索路径为空则返回 NULL，没有有效的模式名也返回 NULL）。如果创建表或者其他命名对象时没有声明目标模式，则将使用这些对象的模式。

- current_schemas(Boolean)

描述：搜索路径中的模式名称。

返回值类型：name[]

示例：

```
postgres=# SELECT current_schemas(true);
  current_schemas
--------------------
 {pg_catalog,public}
(1 row)
```

备注：

current_schemas(Boolean)返回搜索路径中所有模式名称的数组。布尔选项决定像 pg_catalog 这样隐含包含的系统模式是否包含在返回的搜索路径中。

说明

搜索路径可以通过运行时设置更改。命令是：

```
SET search_path TO schema [, schema, ...]
```

- current_user

  描述：当前执行环境下的用户名。

  返回值类型：name

  示例：

```
postgres=# SELECT current_user;
 current_user
--------------
 gbase
(1 row)
```

备注：current_user 是用于权限检查的用户标识，通常表示会话用户，但可以通过 SET ROLE 命令改变。在函数执行的过程中，若 SECURITY DEFINER 属性改变，其值也会改变。

- definer_current_user

  描述：当前执行环境下的用户名。

  返回值类型：name

  示例：

```
postgres=# SELECT definer_current_user();
 definer_current_user
----------------------
gbase
(1 row)
```

- pg_current_sessionid()

  描述：当前执行环境下的会话 ID。

  返回值类型：text

  示例：

```
postgres=# SELECT pg_current_sessionid();
   pg_current_sessionid
---------------------------
```

```
 1652864557.140573668599552
(1 row)
```

备注：pg_current_sessionid()是用于获取当前执行环境下的会话 ID。其组成结构为：时间戳.会话 ID，当线程池模式开启（enable_thread_pool=on）时，会话 ID 为 SessionID；而线程池模式关闭时，会话 ID 为 ThreadID。

- pg_current_sessid

  描述：当前执行环境下的会话 ID。

  返回值类型：text

  示例：

```
 postgres=# select pg_current_sessid();
pg_current_sessid
-------------------
   140573668599552
(1 row)
```

备注：在线程池模式下获得当前会话的会话 ID，非线程池模式下获得当前会话对应的后台线程 ID。

- pg_current_userid

  描述：当前用户 ID。

  返回值类型：text

```
 postgres=# SELECT pg_current_userid();
pg_current_userid
-------------------
10
(1 row)
```

- working_version_num()

  描述：版本序号信息。返回一个系统兼容性有关的版本序号。

  返回值类型：int

  示例：

```
 postgres=# SELECT working_version_num();
 working_version_num
---------------------
```

```
                92605
(1 row)
```

- tablespace_oid_name()

  描述：根据表空间 oid，查找表空间名称。

  返回值类型：text

  示例：

```
postgres=# select tablespace_oid_name(1663);
 tablespace_oid_name
---------------------
 pg_default
(1 row)
```

- inet_client_addr()

  描述：连接的远端地址。inet_client_addr 返回当前客户端的 IP 地址。

  说明

  此函数只有在远程连接模式下有效。

  返回值类型：inet

  示例：

```
postgres=# SELECT inet_client_addr();
 inet_client_addr
------------------
 10.10.0.50
(1 row)
```

- inet_client_port()

  描述：连接的远端端口。inet_client_port 返回当前客户端的端口号。

  说明

  此函数只有在远程连接模式下有效。

  返回值类型：int

  示例：

```
postgres=# SELECT inet_client_port();
```

```
 inet_client_port
------------------
            33143
(1 row)
```

- inet_server_addr()

  描述：连接的本地地址。inet_server_addr 返回服务器接收当前连接用的 IP 地址。

  说明

  此函数只有在远程连接模式下有效。

  返回值类型：inet

  示例：

```
postgres=# SELECT inet_server_addr();
 inet_server_addr
------------------
 10.10.0.13
(1 row)
```

- inet_server_port()

  描述：连接的本地端口。inet_server_port 返回接收当前连接的端口号。如果是通过 Unix-domain socket 连接的，则所有这些函数都返回 NULL。

  说明

  此函数只有在远程连接模式下有效。

  返回值类型：int

  示例：

```
postgres=# SELECT inet_server_port();
 inet_server_port
------------------
             8000
(1 row)
```

- pg_backend_pid()

  描述：当前会话连接的服务进程的进程 ID。

  返回值类型：int

示例：

```
postgres=# SELECT pg_backend_pid();
pg_backend_pid
----------------
 140573668599552
(1 row)
```

- pg_conf_load_time()

  描述：配置加载时间。pg_conf_load_time 返回最后加载服务器配置文件的时间戳。

  返回值类型：timestamp with time zone

  示例：

```
postgres=# SELECT pg_conf_load_time();
      pg_conf_load_time
-------------------------------
 2022-05-05 14:33:02.197289+08
(1 row)
```

- pg_my_temp_schema()

  描述：会话的临时模式的 OID，不存在则为 0。

  返回值类型：oid

  示例：

```
postgres=# SELECT pg_my_temp_schema();
 pg_my_temp_schema
-------------------
                 0
(1 row)
```

备注：pg_my_temp_schema 返回当前会话中临时模式的 OID，如果不存在（没有创建临时表）的话则返回 0。如果给定的 OID 是其它会话中临时模式的 OID，pg_is_other_temp_schema 则返回 true。

- pg_is_other_temp_schema(oid)

  描述：是否为另一个会话的临时模式。

  返回值类型：Boolean

  示例：

```
postgres=# SELECT pg_is_other_temp_schema(25356);
 pg_is_other_temp_schema
-------------------------
 f
(1 row)
```

- pg_listening_channels()

  描述：会话正在侦听的信道名称。

  返回值类型：setof text

  示例：

```
postgres=# SELECT pg_listening_channels();
 pg_listening_channels
-----------------------
(0 rows)
```

  备注：pg_listening_channels 返回当前会话正在侦听的一组信道名称。

- pg_postmaster_start_time()

  描述：服务器启动时间。pg_postmaster_start_time 返回服务器启动时的 timestamp with time zone。

  返回值类型：timestamp with time zone

  示例：

```
postgres=# SELECT pg_postmaster_start_time();
   pg_postmaster_start_time
-------------------------------
 2022-05-05 11:59:14.394307+08
(1 row)
```

- pg_get_ruledef(rule_oid)

  描述：获取规则的 CREATE RULE 命令。

  返回值类型：text

  示例：

```
postgres=# select * from pg_get_ruledef(24828);
pg_get_ruledef
----------------
```

```
-
(1 row)
```

- sessionid2pid()

  描述：从 sessionid 中得到 pid 信息（例如，gs_session_stat 中 sessid 列）。

  返回值类型：int8

  示例：

```
postgres=# select sessionid2pid(sessid::cstring) from gs_session_stat limit 2;
 sessionid2pid
-----------------
 140574146287360
 140574146287360
(2 rows)
```

- pg_trigger_depth()

  描述：触发器的嵌套层次。

  返回值类型：int

  示例：

```
postgres=# SELECT pg_trigger_depth();
pg_trigger_depth
------------------
                0
(1 row)
```

- session_user

  描述：会话用户名。

  返回值类型：name

  示例：

```
postgres=# SELECT session_user;
session_user
--------------
gbase
(1 row)
```

备注：session_user 通常是连接当前数据库的初始用户，不过系统管理员可以用 SET SESSION AUTHORIZATION 修改这个设置。

- user

  描述：等价于 current_user。

  返回值类型：name

  示例：

```
postgres=# SELECT user;
current_user
--------------
gbase
(1 row)
```

- getpgusername()

  描述：获取数据库用户名。

  返回值类型：name

  示例：

```
postgres=# select getpgusername();
getpgusername
---------------
gbase
(1 row)
```

- getdatabaseencoding()

  描述：获取数据库编码方式。

  返回值类型：name

  示例：

```
postgres=# select getdatabaseencoding();
getdatabaseencoding
--------------------
SQL_ASCII
(1 row)
```

- version()

  描述：版本信息。version 返回一个描述服务器版本信息的字符串。

  返回值类型：text

示例：

```
postgres=# select version();
            version

------------------------------------------------------------------
------------------------------------------------------------------
--------------------------------
 single_node (GBase8s 5.0.0B06 build 19f48e93) compiled at 2022-04
-29 14:37:48 commit 0 last mr 36  on x86_64-unknown-linux-gnu, com
piled by g++ (GCC) 7.3.0, 64-bit
(1 row)
```

- opengauss_version()

  描述：兼容 openGauss 的版本信息。

  返回值类型：text

  示例：

```
postgres=# select opengauss_version();
opengauss_version
-------------------
 5.0.0
(1 row)
```

- gs_deployment()

  描述：当前系统的部署形态信息。

  返回值类型：text

  示例：

```
postgres=# select gs_deployment();
    gs_deployment
----------------------
 OpenSourceCentralized
(1 row)
```

- get_hostname()

  描述：返回当前节点的 hostname。

  返回值类型：text

示例：

```
postgres=# SELECT get_hostname();
get_hostname
--------------
 gbase8s_7_16
(1 row)
```

- get_nodename()

  描述：返回当前节点的名字。

  返回值类型：text

  示例：

```
postgres=# SELECT get_nodename();
 get_nodename
--------------
 dn1
(1 row)
```

- get_schema_oid(cstring)

  描述：返回查询 schema 的 oid。

  返回值类型：oid

  示例：

```
postgres=# SELECT get_schema_oid('public');
 get_schema_oid
----------------
           2200
(1 row)
```

- get_client_info()

  描述：返回客户端信息。

  返回值类型：record

5.25.2 访问权限查询函数

DDL 类权限 ALTER、DROP、COMMENT、INDEX、VACUUM 属于所有者固有的权限，隐式拥有。

- has_any_column_privilege(user, table, privilege)

  描述：指定用户是否有访问表任何列的权限。

表 5-13 参数类型说明

参数名	合法入参类型
user	name, oid
table	text, oid
privilege	text

返回类型：Boolean

has_any_column_privilege(table, privilege)

描述：当前用户是否有访问表任何列的权限，合法参数类型见表 5-12。

返回类型：Boolean

备注：has_any_column_privilege 检查用户是否以特定方式访问表的任何列。其参数可能与 has_table_privilege 类似，除了访问权限类型必须是 SELECT、INSERT、UPDATE、COMMENT 或 REFERENCES 的一些组合。

说明

拥有表的表级别权限则隐含的拥有该表每列的列级权限，因此如果与 has_table_privilege 参数相同，has_any_column_privilege 总是返回 true。但是如果授予至少一列的列级权限也返回成功。

has_column_privilege(user, table, column, privilege)

描述：指定用户是否有访问列的权限。

表 5-14 参数类型说明

参数名	合法入参类型
user	name, oid
table	text, oid

column	text, smallint
privilege	text

返回类型：Boolean

- has_column_privilege(table, column, privilege)

描述：当前用户是否有访问列的权限，合法参数类型见表 5-13。

返回类型：Boolean

备注：has_column_privilege 检查用户是否以特定方式访问一列。其参数类似于 has_table_privilege，可以通过列名或属性号添加列。想要的访问权限类型必须是 SELECT、INSERT、UPDATE、COMMENT 或 REFERENCES 的一些组合。

说明

拥有表的表级别权限则隐含的拥有该表每列的列级权限。

- has_cek_privilege(user, cek, privilege)

描述：指定用户是否有访问列加密密钥 CEK 的权限。参数说明如下。

表 5-15 参数类型说明

参数名	合法入参类型	描述	取值范围
user	name，oid	用户	用户名字或 id。
cek	text，oid	列加密密钥	列加密密钥名称或 id。
privilege	text	权限	USAGE：允许使用指定列加密密钥。 DROP：允许删除指定列加密密钥。

返回类型：Boolean

- has_cmk_privilege(user, cmk, privilege)

描述：指定用户是否有访问客户端加密主密钥 CMK 的权限。参数说明如下。

表 5-16 参数类型说明

参数名	合法入参类型	描述	取值范围
user	name，oid	用户	用户名字或 id。
cmk	text，oid	客户端加密主密钥	客户端加密主密钥名称或 id。
privilege	text	权限	USAGE：允许使用指定客户端加密主密钥。 DROP：允许删除指定客户端加密主密钥。

返回类型：Boolean

- has_database_privilege(user, database, privilege)

  描述：指定用户是否有访问数据库的权限。参数说明如下。

表 5-17 参数类型说明

参数名	合法入参类型
user	name, oid
database	text, oid
privilege	text

返回类型：Boolean

- has_database_privilege(database, privilege)

  描述：当前用户是否有访问数据库的权限，合法参数类型请参见表 5-17。

  返回类型：Boolean

  备注：has_database_privilege 检查用户是否能以在特定方式访问数据库。其参数类似 has_table_privilege。访问权限类型必须是 CREATE、CONNECT、TEMPORARY、ALTER、DROP、COMMENT 或 TEMP（等价于 TEMPORARY）的一些组合。

- has_directory_privilege(user, directory, privilege)

  描述：指定用户是否有访问 directory 的权限。

表 5-18 参数类型说明

参数名	合法入参类型
user	name, oid
directory	text, oid
privilege	text

  返回类型：Boolean

- has_directory_privilege(directory, privilege)

  描述：当前用户是否有访问 directory 的权限，合法参数类型请参见表 5-18。

  返回类型：Boolean

- has_foreign_data_wrapper_privilege(user, fdw, privilege)

  描述：指定用户是否有访问外部数据封装器的权限。

表 5-19 参数类型说明

参数名	合法入参类型
user	name, oid
fdw	text, oid
privilege	text

  返回类型：Boolean

- has_foreign_data_wrapper_privilege(fdw, privilege)

  描述：当前用户是否有访问外部数据封装器的权限。合法参数类型请参见表 5-19。

  返回类型：Boolean

备注：has_foreign_data_wrapper_privilege 检查用户是否能以特定方式访问外部数据封装器。其参数类似 has_table_privilege。访问权限类型必须是 USAGE。

- has_function_privilege(user, function, privilege)

  描述：指定用户是否有访问函数的权限。

表 5-20 参数类型说明

参数名	合法入参类型
user	name, oid
function	text, oid
privilege	text

返回类型：Boolean

- has_function_privilege(function, privilege)

  描述：当前用户是否有访问函数的权限。合法参数类型请参见表 5-20。

  返回类型：Boolean

备注：has_function_privilege 检查一个用户是否能以指定方式访问一个函数。其参数类似 has_table_privilege。使用文本字符而不是 OID 声明一个函数时，允许输入的类型和 regprocedure 数据类型一样（请参考对象标识符类型）。访问权限类型必须是 EXECUTE、ALTER、DROP 或 COMMENT。

- has_language_privilege(user, language, privilege)

  描述：指定用户是否有访问语言的权限。

表 5-21 参数类型说明

参数名	合法入参类型
user	name, oid
language	text, oid

privilege	text

返回类型：Boolean

- has_language_privilege(language, privilege)

描述：当前用户是否有访问语言的权限。合法参数类型请参见表 5-21。

返回类型：Boolean

备注：has_language_privilege 检查用户是否能以特定方式访问一个过程语言。其参数类似 has_table_privilege。访问权限类型必须是 USAGE。

- has_nodegroup_privilege(user, nodegroup, privilege)

描述：检查用户是否有数据库节点访问权限。

返回类型：Boolean

表 5-22 参数类型说明

参数名	合法入参类型
user	name, oid
nodegroup	text, oid
privilege	text

- has_nodegroup_privilege(nodegroup, privilege)

描述：检查用户是否有数据库节点访问权限。参数与 has_table_privilege 类似。访问权限类型必须是 USAGE、CREATE、COMPUTE、ALTER 或 CROP。

返回类型：Boolean

- has_schema_privilege(user, schema, privilege)

描述：指定用户是否有访问模式的权限。

返回类型：Boolean

- has_schema_privilege(schema, privilege)

描述：当前用户是否有访问模式的权限。

返回类型：Boolean

备注：has_schema_privilege 检查用户是否能以特定方式访问一个模式。其参数类似 has_table_privilege。访问权限类型必须是 CREATE、USAGE、ALTER、DROP 或 COMMENT 的一些组合。

- has_server_privilege(user, server, privilege)

  描述：指定用户是否有访问外部服务的权限。

  返回类型：Boolean

- has_server_privilege(server, privilege)

  描述：当前用户是否有访问外部服务的权限。

  返回类型：Boolean

  备注：has_server_privilege 检查用户是否能以指定方式访问一个外部服务器。其参数类似 has_table_privilege。访问权限类型必须是 USAGE、ALTER、DROP 或 COMMENT 之一的值。

- has_table_privilege(user, table, privilege)

  描述：指定用户是否有访问表的权限。

  返回类型：Boolean

- has_table_privilege(table, privilege)

  描述：当前用户是否有访问表的权限。

  返回类型：Boolean

  备注：has_table_privilege 检查用户是否以特定方式访问表。用户可以通过名称或 OID（pg_authid.oid）来指定，public 表明 PUBLIC 伪角色，或如果缺省该参数，则使用 current_user。该表可以通过名称或者 OID 声明。如果用名称声明，则在必要时可以用模式进行修饰。如果使用文本字符串来声明所希望的权限类型，这个文本字符串必须是 SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGER、ALTER、DROP、COMMENT、INDEX 或 VACUUM 之一的值。可以给权限类型添加 WITH GRANT OPTION，用来测试权限是否拥有授权选项。也可以用逗号分隔列出的多个权限类型，如果拥有任何所列出的权限，则结果便为 true。

示例：

```
postgres=# SELECT has_table_privilege('public.web_site', 'select');
 has_table_privilege
---------------------
 t
(1 row)

postgres=# SELECT has_table_privilege('gbase', 'public.web_site',
'select,INSERT WITH GRANT OPTION ');
 has_table_privilege
---------------------
 t
(1 row)
```

has_tablespace_privilege(user, tablespace, privilege)

描述：指定用户是否有访问表空间的权限。

返回类型：Boolean

has_tablespace_privilege(tablespace, privilege)

描述：当前用户是否有访问表空间的权限。

返回类型：Boolean

备注：has_tablespace_privilege 检查用户是否能以特定方式访问一个表空间。其参数类似 has_table_privilege。访问权限类型必须是 CREATE、ALTER、DROP 或 COMMENT 之一的值。

pg_has_role(user, role, privilege)

描述：指定用户是否有角色的权限。

返回类型：Boolean

pg_has_role(role, privilege)

描述：当前用户是否有角色的权限。

返回类型：Boolean

备注：pg_has_role 检查用户是否能以特定方式访问一个角色。其参数类似 has_table_privilege，除了 public 不能用做用户名。访问权限类型必须是 MEMBER 或 USAGE 的一些组合。 MEMBER 表示的是角色中的直接或间接成员关系（也就是 SET ROLE 的权

限），而 USAGE 表示无需通过 SET ROLE 也直接拥有角色的使用权限。

has_any_privilege(user, privilege)

描述：指定用户是否有某项 ANY 权限，若同时查询多个权限，只要具有其中一个则返回 true。

返回类型：Boolean

表 5-23 参数类型说明

参数名	合法入参类型	描述	取值范围
user	name	用户	已存在的用户名。
privilege	text	ANY 权限	可选取值： CREATE ANY TABLE [WITH ADMIN OPTION] ALTER ANY TABLE [WITH ADMIN OPTION] DROP ANY TABLE [WITH ADMIN OPTION] SELECT ANY TABLE [WITH ADMIN OPTION] INSERT ANY TABLE [WITH ADMIN OPTION] UPDATE ANY TABLE [WITH ADMIN OPTION] DELETE ANY TABLE [WITH ADMIN OPTION] CREATE ANY SEQUENCE [WITH ADMIN OPTION] CREATE ANY INDEX [WITH ADMIN OPTION] CREATE ANY FUNCTION [WITH ADMIN OPTION] EXECUTE ANY FUNCTION [WITH ADMIN OPTION] CREATE ANY PACKAGE [WITH ADMIN OPTION] EXECUTE ANY PACKAGE [WITH ADMIN

			OPTION] CREATE ANY TYPE [WITH ADMIN OPTION]

5.25.3 模式可见性查询函数

每个函数执行检查数据库对象类型的可见性。对于函数和操作符，如果在前面的搜索路径中没有相同的对象名称和参数的数据类型，则此对象是可见的。对于操作符类，则要同时考虑名称和相关索引的访问方法。

所有这些函数都需要使用 OID 来标识要需要检查的对象。如果用户想通过名称测试对象，则使用 OID 别名类型（regclass、regtype、regprocedure、regoperator、regconfig 或 regdictionary）将会很方便。

比如，如果一个表所在的模式在搜索路径中，并且在前面的搜索路径中没有同名的表，则这个表是可见的。它等效于表可以不带明确模式修饰进行引用。比如，要列出所有可见表的名称：

```
postgres=# SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
                   relname
-----------------------------------------------
pg_type
gs_client_global_keys_args
abc
pg_subscription_oid_index
pg_subscription_subname_index
pgxc_prepared_xacts
pg_shadow
pg_roles
pg_user
pg_group
pg_rules
pg_authid
gs_labels
pg_rlspolicies
gs_auditing_access
......
```

- pg_collation_is_visible(collation_oid)

  描述：该排序是否在搜索路径中可见。

返回类型：Boolean

- pg_conversion_is_visible(conversion_oid)

  描述：该转换是否在搜索路径中可见。

  返回类型：Boolean

- pg_function_is_visible(function_oid)

  描述：该函数是否在搜索路径中可见。

  返回类型：Boolean

- pg_opclass_is_visible(opclass_oid)

  描述：该操作符类是否在搜索路径中可见。

  返回类型：Boolean

- pg_operator_is_visible(operator_oid)

  描述：该操作符是否在搜索路径中可见。

  返回类型：Boolean

- pg_opfamily_is_visible(opclass_oid)

  描述：该操作符族是否在搜索路径中可见。

  返回类型：Boolean

- pg_table_is_visible(table_oid)

  描述：该表是否在搜索路径中可见。

  返回类型：Boolean

- pg_ts_config_is_visible(config_oid)

  描述：该文本检索配置是否在搜索路径中可见。

  返回类型：Boolean

- pg_ts_dict_is_visible(dict_oid)

  描述：该文本检索词典是否在搜索路径中可见。

  返回类型：Boolean

- pg_ts_parser_is_visible(parser_oid)

  描述：该文本搜索解析是否在搜索路径中可见。

  返回类型：Boolean

- pg_ts_template_is_visible(template_oid)

  描述：该文本检索模板是否在搜索路径中可见。

  返回类型：Boolean

- pg_type_is_visible(type_oid)

  描述：该类型（或域）是否在搜索路径中可见。

  返回类型：Boolean

5.25.4 系统表信息函数

- format_type(type_oid, typemod)

  描述：获取数据类型的 SQL 名称。

  返回类型：text

  备注：format_type 通过某个数据类型的类型 OID 以及可能的类型修饰词，返回其 SQL 名称。如果不知道具体的修饰词，则在类型修饰词的位置传入 NULL。类型修饰词一般只对有长度限制的数据类型有意义。format_type 所返回的 SQL 名称中包含数据类型的长度值，其大小是：实际存储长度 len - sizeof(int32)，单位字节。原因是数据存储时需要 32 位的空间来存储用户对数据类型的自定义长度信息，即实际存储长度要比用户定义长度多 4 个字节。在下例中，format_type 返回的 SQL 名称为“character varying(6)”，6 表示 varchar 类型的长度值是 6 字节，因此该类型的实际存储长度为 10 字节。

```
postgres=# SELECT format_type((SELECT oid FROM pg_type WHERE typname='varchar'),
10);
     format_type
---------------------
 character varying(6)
(1 row)
```

- getdistributekey(table_name)

  描述：获取一个 hash 表的分布列。单机环境下不支持分布，该函数返回为空。

- pg_check_authid(role_oid)

  描述：检查是否存在给定 oid 的角色名。

  返回类型：bool

  示例：

```
 postgres=# select pg_check_authid(1);
pg_check_authid
-----------------
f
(1 row)
```

- pg_describe_object(catalog_id, object_id, object_sub_id)

  描述：获取数据库对象的描述。

  返回类型：text

  备注：pg_describe_object 返回由目录 OID，对象 OID 和一个（或许 0 个）子对象 ID 指定的数据库对象的描述。这有助于确认存储在 pg_depend 系统表中对象的身份。

- pg_get_constraintdef(constraint_oid)

  描述：获取约束的定义。

  返回类型：text

- pg_get_constraintdef(constraint_oid, pretty_bool)

  描述：获取约束的定义。

  返回类型：text

  备注：pg_get_constraintdef 和 pg_get_indexdef 分别从约束或索引上使用创建命令进行重构。

- pg_get_expr(pg_node_tree, relation_oid)

  描述：反编译表达式的内部形式，假设其中的任何 Vars 都引用第二个参数指定的关系。

  返回类型：text

- pg_get_expr(pg_node_tree, relation_oid, pretty_bool)

  描述：反编译表达式的内部形式，假设其中的任何 Vars 都引用第二个参数指定的关系。

返回类型：text

备注：pg_get_expr 反编译一个独立表达式的内部形式，比如一个字段的缺省值。在检查系统表的内容的时候很有用。如果表达式可能包含关键字，则指定他们引用相关的 OID 作为第二个参数；如果没有关键字，零就足够了。

- pg_get_functiondef(func_oid)

描述：获取函数的定义。

返回类型：text

示例：

```
 postgres=# select * from pg_get_functiondef(598);
headerlines |                    definition
------------+-------------------------------------------------
           4 | CREATE OR REPLACE FUNCTION pg_catalog.abbrev(inet)+
             |  RETURNS text                                    +
             |  LANGUAGE internal                               +
             |  IMMUTABLE STRICT NOT FENCED NOT SHIPPABLE       +
             | AS $function$inet_abbrev$function$;              +
             |
(1 row)
```

- pg_get_function_arguments(func_oid)

描述：获取函数定义的参数列表（带默认值）。

返回类型：text

备注：pg_get_function_arguments 返回一个函数的参数列表，需要在 CREATE FUNCTION 中使用这种格式。

- pg_get_function_identity_arguments(func_oid)

描述：获取参数列表来确定一个函数（不带默认值）。

返回类型：text

备注：pg_get_function_identity_arguments 返回需要的参数列表用来标识函数，这种形式需要在 ALTER FUNCTION 中使用，并且这种形式省略了默认值。

- pg_get_function_result(func_oid)

描述：获取函数的 RETURNS 子句。

返回类型：text

备注：pg_get_function_result 为函数返回适当的 RETURNS 子句。

- pg_get_indexdef(index_oid)

描述：获取索引的 CREATE INDEX 命令。

返回类型：text

示例：

```
postgres=# select * from pg_get_indexdef(16416);
                            pg_get_indexdef
-------------------------------------------------------------------------
 CREATE INDEX test3_b_idx ON test3 USING btree (b) TABLESPACE pg_default
(1 row)
```

- pg_get_indexdef(index_oid, dump_schema_only)

描述：获取索引的 CREATE INDEX 命令，仅用于 dump 场景。对于包含 local 索引的间隔分区表，当 dump_schema_only 为 true 时，返回的创建索引语句中不包含自动创建的分区的 local 索引信息；当 dump_schema_only 为 false 时，返回的创建索引语句中包含自动创建的分区的 local 索引信息。对于非间隔分区表或者不包含 local 索引的间隔分区分区表，dump_schema_only 参数取值不影响函数返回结果。

返回类型：text

示例：

```
postgres=# CREATE TABLE sales
gbase-# (prod_id NUMBER(6),
gbase(#  cust_id NUMBER,
gbase(#  time_id DATE,
gbase (#  channel_id CHAR(1),
gbase (#  promo_id NUMBER(6),
gbase (#  quantity_sold NUMBER(3),
gbase (#  amount_sold NUMBER(10,2)
gbase (# )
gbase-# PARTITION BY RANGE( time_id) INTERVAL('1 day')
gbase -# (
gbase (# partition p1 VALUES LESS THAN ('2019-02-01 00:00:00'),
gbase (# partition p2 VALUES LESS THAN ('2019-02-02 00:00:00')
gbase (# );
CREATE TABLE
```

```
postgres=# create index index_sales on sales(prod_id) local (PARTITION
idx_p1 ,PARTITION idx_p2);
CREATE INDEX
-- 插入数据没有匹配的分区，新创建一个分区，并将数据插入该分区
postgres=# INSERT INTO sales VALUES(1, 12, '2019-02-05 00:00:00', 'a', 1, 1, 1);
INSERT 0 1
postgres=# select oid from pg_class where relname = 'index_sales';
  oid
-------
 16636
(1 row)
postgres=# select * from pg_get_indexdef(16636, true);
pg_get_indexdef
--------------------------------------------------------------------------------
--------------------------------------------------------------------------
 CREATE INDEX index_sales ON sales USING ubtree (prod_id) LOCAL(PARTITION idx_p1,
PARTITION idx_p2)  WITH (storage_type=USTORE) TABLESPACE pg_default
(1 row)
postgres=# select * from pg_get_indexdef(16636, false);
                pg_get_indexdef

--------------------------------------------------------------------
--------------------------------------------------------------------
-----------------------------------------------
 CREATE INDEX index_sales ON sales USING ubtree (prod_id) LOCAL(PA
RTITION idx_p1, PARTITION idx_p2, PARTITION sys_p1_prod_id_idx)  W
ITH (storage_type=USTORE) TABLESPACE pg_default
(1 row)
```

- pg_get_indexdef(index_oid, column_no, pretty_bool)

描述：获取索引的 CREATE INDEX 命令，或者如果 column_no 不为零，则只获取一个索引字段的定义。

示例：

```
postgres=# select * from pg_get_indexdef(16416, 0, false);
                            pg_get_indexdef
--------------------------------------------------------------------------------
 CREATE INDEX test3_b_idx ON test3 USING btree (b) TABLESPACE pg_default
(1 row)
postgres=# select * from pg_get_indexdef(16416, 1, false);
 pg_get_indexdef
```

```
----------------
 b
(1 row)
```

返回类型：text

备注：pg_get_functiondef 为函数返回一个完整的 CREATE OR REPLACE FUNCTION 语句。

- pg_get_keywords()

  描述：获取 SQL 关键字和类别列表。

  返回类型：setof record

  备注：pg_get_keywords 返回一组关于描述服务器识别 SQL 关键字的记录。word 列包含关键字。catcode 列包含一个分类代码：U 表示通用的，C 表示列名，T 表示类型或函数名，或 R 表示保留。catdesc 列包含了一个可能本地化描述分类的字符串。

- pg_get_userbyid(role_oid)

  描述：获取给定 OID 的角色名。

  返回类型：name

  备注：pg_get_userbyid 通过角色的 OID 抽取对应的用户名。

- pg_check_authid(role_id)

  描述：通过 role_id 检查用户是否存在。

  返回类型：text

  示例：

```
 postgres=# select pg_check_authid(20);
pg_check_authid
----------------
f
(1 row)
```

- pg_get_viewdef(view_name)

  描述：为视图获取底层的 SELECT 命令。

  返回类型：text

- pg_get_viewdef(view_name, pretty_bool)

描述：为视图获取底层的 SELECT 命令，如果 pretty_bool 为 true，行字段可以包含 80 列。

返回类型：text

备注：pg_get_viewdef 重构出定义视图的 SELECT 查询。这些函数大多数都有两种形式，其中带有 pretty_bool 参数，且参数为 true 时，是"适合打印"的结果，这种格式更容易读。另一种是缺省的格式，更有可能被将来的不同版本用同样的方法解释。如果是用于转储，那么尽可能避免使用适合打印的格式。给 pretty-print 参数传递 false 生成的结果和没有这个参数的变种生成的结果是完全一样。

- pg_get_viewdef(view_oid)

描述：为视图获取底层的 SELECT 命令。

返回类型：text

- pg_get_viewdef(view_oid, pretty_bool)

描述：为视图获取底层的 SELECT 命令，如果 pretty_bool 为 true，行字段可以包含 80 列。

返回类型：text

- pg_get_viewdef(view_oid, wrap_column_int)

描述：为视图获取底层的 SELECT 命令；行字段被换到指定的列数，打印是隐含的。

返回类型：text

- pg_get_tabledef(table_oid)

描述：根据 table_oid 获取表定义

示例：

```
postgres=# select * from pg_get_tabledef(16384);
                    pg_get_tabledef
-----------------------------------------------------
 SET search_path = public;                          +
 CREATE  TABLE t1 (                                 +
         c1 bigint DEFAULT nextval('serial'::regclass)+
 )                                                  +
```

```
WITH (orientation=row, compression=no)                +
TO GROUP group1;
(1 row)
```

返回类型：text

- pg_get_tabledef(table_name)

描述：根据 table_name 获取表定义。

示例：

```
postgres=# select * from pg_get_tabledef('classes');
                       pg_get_tabledef
-------------------------------------------------------------
 SET search_path = public;                                   +
 CREATE TABLE classes (                                      +
     name character varying,                                 +
     score bigint                                            +
 )                                                           +
 WITH (orientation=row, compression=no, storage_type=USTORE);
(1 row)
```

返回类型：text

备注：pg_get_tabledef 重构出表定义的 CREATE 语句，包含了表定义本身、索引信息、comments 信息。对于表对象依赖的 group、schema、tablespace、server 等信息，需要用户自己去创建，表定义里不会有这些对象的创建语句。

- pg_options_to_table(reloptions)

描述：获取存储选项名称/值对的集合。

返回类型：setof record

备注：pg_options_to_table 当通过 pg_class.reloptions 或 pg_attribute.attoptions 时返回存储选项名称/值对（option_name/option_value）的集合。

- pg_tablespace_databases(tablespace_oid)

描述：获取在指定的表空间中有对象的数据库 OID 集合。

返回类型：setof oid

备注：pg_tablespace_databases 允许检查表空间的状况，返回在该表空间中保存了对象的数据库 OID 集合。如果这个函数返回数据行，则该表空间就是非空的，因此不能删除。

要显示该表空间中的特定对象，用户需要连接 pg_tablespace_databases 标识的数据库与查询 pg_class 系统表。

- pg_tablespace_location(tablespace_oid)

  描述：获取表空间所在的文件系统的路径。

  返回类型：text

- pg_typeof(any)

  描述：获取任何值的数据类型。

  返回类型：regtype

  备注：pg_typeof 返回传递给他的值的数据类型 OID。这可能有助于故障排除或动态构造 SQL 查询。声明此函数返回 regtype，这是一个 OID 别名类型（请参考对象标识符类型）；这意味着它是一个为了比较而显示类型名称的 OID。

  示例：

```
postgres=# SELECT pg_typeof(33);
 pg_typeof
-----------
 integer
(1 row)

postgres=# SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

- collation for (any)

  描述：获取参数的排序。

  返回类型：text

  备注：表达式 collation for 返回传递给他的值的排序。

  示例：

```
postgres=# SELECT collation for (description) FROM pg_description LIMIT 1;
 pg_collation_for
------------------
```

```
 "default"
(1 row)
```

值可能是引号括起来的并且模式限制的。如果没有为参数表达式排序，则返回一个 null 值。如果参数不是排序的类型，则抛出一个错误。

- pg_extension_update_paths(name)

  描述：返回指定扩展的版本更新路径。

  返回类型：text(source text), text(path text), text(target text)

- pg_get_serial_sequence(tablename, colname)

  描述：获取对应表名和列名上的序列。

  返回类型：text

  示例：

```
postgres=# select * from pg_get_serial_sequence('t1', 'c1');
 pg_get_serial_sequence
------------------------
 public.serial
(1 row)
```

- pg_sequence_parameters(sequence_oid)

  描述：获取指定 sequence 的参数，包含起始值，最小值和最大值，递增值等。

  返回类型：int16, int16,int16, bigint, Boolean

  示例：

```
postgres=# select * from pg_sequence_parameters(16420);
 start_value | minimum_value |    maximum_value    | increment | cycle_option
-------------+---------------+---------------------+-----------+--------------
         101 |             1 | 9223372036854775807 |         1 | f
(1 row)
```

5.25.5 注释信息函数

- col_description(table_oid, column_number)

  描述：获取一个表字段的注释

  返回类型：text

备注：col_description 返回一个表中字段的注释，通过表 OID 和字段号来声明。

- obj_description(object_oid, catalog_name)

  描述：获取一个数据库对象的注释

  返回类型：text

  备注：带有两个参数的 obj_description 返回一个数据库对象的注释，该对象是通过其 OID 和其所属的系统表名称声明。比如，obj_description(123456,'pg_class')将返回 OID 为 123456 的表的注释。只带一个参数的 obj_description 只要求对象 OID。

  obj_description 不能用于表字段，因为字段没有自己的 OID。

- obj_description(object_oid)

  描述：获取一个数据库对象的注释

  返回类型：text

- shobj_description(object_oid, catalog_name)

  描述：获取一个共享数据库对象的注释

  返回类型：text

  备注：shobj_description 和 obj_description 差不多，不同之处仅在于前者用于共享对象。一些系统表是通用于 GBase 8s 中所有数据库的全局表，因此这些表的注释也是全局存储的。

5.25.6 事务 ID 和快照

内部事务 ID 类型（xid）是 64 位。这些函数使用的数据类型 txid_snapshot，存储在特定时刻事务 ID 可见性的信息。其组件描述详见下表。

表 5-24 快照组件

名称	描述
xmin	最早的事务 ID（txid）仍然活动。所有较早事务将是已经提交可见的，或者是直接回滚。
xmax	作为尚未分配的 txid。所有大于或等于此 txids 的都是尚未开始的快照时间，因此不可见。

xip_list	当前快照中活动的 txids。这个列表只包含在 xmin 和 xmax 之间活动的 txids；有可能活动的 txids 高于 xmax。介于大于等于 xmin、小于 xmax，并且不在这个列表中的 txid，在这个时间快照已经完成的，因此按照提交状态查看他是可见还是回滚。这个列表不包含子事务的 txids。

txid_snapshot 的文本表示为：xmin:xmax:xip_list。

示例：10:20:10,14,15 意思为：xmin=10, xmax=20, xip_list=10, 14, 15。

以下的函数在一个输出形式中提供服务器事务信息。这些函数的主要用途是为了确定在两个快照之间有哪个事务提交。

- txid_current()

  描述：获取当前事务 ID。

  返回类型：bigint

- gs_txid_oldestxmin()

  描述：获取当前最小事务 id 的值 oldesxmin。

  返回类型：bigint

- txid_current_snapshot()

  描述：获取当前快照。

  返回类型：txid_snapshot

- txid_snapshot_xip(txid_snapshot)

  描述：在快照中获取正在进行的事务 ID。

  返回类型：setof bigint

- txid_snapshot_xmax(txid_snapshot)

  描述：获取快照的 xmax。

  返回类型：bigint

- txid_snapshot_xmin(txid_snapshot)

  描述：获取快照的 xmin。

返回类型：bigint

- txid_visible_in_snapshot(bigint, txid_snapshot)

描述：在快照中事务 ID 是否可见（不使用子事务 ID）。

返回类型：Boolean

- get_local_prepared_xact()

描述：获取当前节点两阶段残留事务信息，包括事务 id，两阶段 gid 名称，prepared 的时间，owner 的 oid，database 的 oid 及当前节点的 node_name。

返回类型：xid, text, timestamptz, oid, oid，text

- get_remote_prepared_xacts()

描述：获取所有远程节点两阶段残留事务信息，包括事务 id，两阶段 gid 名称，prepared 的时间，owner 的名称，database 的名称及 node_name。

返回类型：xid, text, timestamptz, name, name，text

- global_clean_prepared_xacts(text, text)

描述：并发清理两阶段残留事务，仅 gs_clean 工具可以调用清理，其他用户调用均返回 false。

返回类型：Boolean

- gs_get_next_xid_csn()

描述：返回全局所有节点上的 next_xid 和 next_csn 值。

返回值如下：

表 5-25 gs_get_next_xid_csn 返回参数说明

字段名	描述
nodename	节点名称。
next_xid	当前节点下一个事务 id 号。
next_csn	当前节点下一个 csn 号。

- slice(hstore, text[])

  描述：提取 hstore 的子集。

  返回值：hstore

  示例：

```
postgres=# select slice('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x']);
      slice
-------------------
"b"=>"2", "c"=>"3"
(1 row)
```

- slice_array(hstore, text[])

  描述：提取 hstore 的值的集合。

  返回值：值数组

  示例：

```
postgres=# select slice_array('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x']);
slice_array
-------------
{2,3,NULL}
(1 row)
```

- skeys(hstore)

  描述：返回 hstore 的所有键构成的集合。

  返回值：键的集合。

示例：

```
postgres=# select skeys('a=>1,b=>2');
skeys
-------
a
b
(2 rows)
```

- pg_control_system()

  描述：返回系统控制文件状态。

  返回类型：SETOF record

- pg_control_checkpoint()

  描述：返回系统检查点状态。

  返回类型：SETOF record

- pv_builtin_functions

  描述：查看所有内置系统函数信息。

  参数：nan

  返回值类型： proname name, pronamespace oid, proowner oid, prolang oid, procost real, prorows real, provariadic oid, protransform regproc, proisagg boolean, proiswindow boolean, prosecdef boolean, proleakproof boolean, proisstrict boolean, proretset boolean, provolatile “char”, pronargs smallint, pronargdefaults smallint, prorettype oid, proargtypes oidvector, proallargtypes integer[], proargmodes “char”[], proargnames text[], proargdefaults pg_node_tree, prosrc text, probin text, proconfig text[], proacl aclitem[], prodefaultargpos int2vector, fencedmode boolean, proshippable boolean, propackage boolean, oid oid

- pv_thread_memory_detail

  描述：返回各线程的内存信息。

  参数：nan

  返回值类型： threadid text, tid bigint, thrdtype text, contextname text, level smallint, parent text, totalsize bigint, freesize bigint, usedsize bigint

- pg_relation_compression_ratio

  描述：查询表压缩率，默认返回 1.0。

  参数：text

  返回值类型：real

- pg_relation_with_compression

  描述：查询表是否压缩。

  参数：text

  返回值类型：boolean

- pg_stat_file_recursive

描述：列出路径下所有文件。

参数：location text

- pg_shared_memory_detail

描述：返回所有已产生的共享内存上下文的使用信息，各列描述请参考GS_SHARED_MEMORY_DETAIL。

参数：nan

返回值类型：contextname text, level smallint, parent text, totalsize bigint, freesize bigint, usedsize bigint

- get_gtm_lite_status

描述：返回 GTM 上的 backupXid 和 csn 号，用来支持问题定位，GTM-FREE 模式下不支持使用本系统函数。

- gs_stat_get_wlm_plan_operator_info

描述：从内部哈希表中获取算子计划信息。

参数：oid

返回值类型：datname text, queryid int8, plan_node_id int4, startup_time int8, total_time int8, actual_rows int8, max_peak_memory int4, query_dop int4, parent_node_id int4, left_child_id int4, right_child_id int4, operation text, orientation text, strategy text, options text, condition text, projection text

- pg_stat_get_partition_tuples_hot_updated

描述：返回给定分区 id 的分区热更新元组数的统计。

参数：oid

返回值类型：bigint

- gs_session_memory_detail_tp

描述：返回会话的内存使用情况，参考 gs_session_memory_detail。

参数：nan

返回值类型：sessid text, sesstype text, contextname text, level smallint, parent text, totalsize bigint, freesize bigint, usedsize bigint

- gs_thread_memory_detail

  描述：返回各线程的内存信息。

  参数：nan

  返回值类型：threadid text, tid bigint, thrdtype text, contextname text, level smallint, parent text, totalsize bigint, freesize bigint, usedsize bigint

- pg_stat_get_wlm_realtime_operator_info

  描述：从内部哈希表中获取实时执行计划算子信息。

  参数：nan

  返回值类型：queryid bigint, pid bigint, plan_node_id integer, plan_node_name text, start_time timestamp with time zone, duration bigint, status text, query_dop integer, estimated_rows bigint, tuple_processed bigint, min_peak_memory integer, max_peak_memory integer, average_peak_memory integer, memory_skew_percent integer, min_spill_size integer, max_spill_size integer, average_spill_size integer, spill_skew_percent integer, min_cpu_time bigint, max_cpu_time bigint, total_cpu_time bigint, cpu_skew_percent integer, warning text

- pg_stat_get_wlm_realtime_ec_operator_info

  描述：从内部哈希表中获取 EC 执行计划算子信息。

  参数：nan

  返回值类型：queryid bigint, plan_node_id integer, plan_node_name text, start_time timestamp with time zone, ec_operator integer, ec_status text, ec_execute_datanode text, ec_dsn text, ec_username text, ec_query text, ec_libodbc_type text, ec_fetch_count bigint

- pg_stat_get_wlm_operator_info

  描述：从内部哈希表中获取执行计划算子信息。

  参数：nan

  返回值类型：queryid bigint, pid bigint, plan_node_id integer, plan_node_name text, start_time timestamp with time zone, duration bigint, query_dop integer, estimated_rows bigint, tuple_processed bigint, min_peak_memory integer, max_peak_memory integer, average_peak_memory integer, memory_skew_percent integer, min_spill_size integer, max_spill_size integer, average_spill_size integer, spill_skew_percent integer, min_cpu_time

bigint, max_cpu_time bigint, total_cpu_time bigint, cpu_skew_percent integer, warning text

- pg_stat_get_wlm_node_resource_info

  描述：获取当前节点资源信息。

  参数：nan

  返回值类型：min_mem_util integer, max_mem_util integer, min_cpu_util integer, max_cpu_util integer, min_io_util integer, max_io_util integer, used_mem_rate integer

- pg_stat_get_session_wlmstat

  描述：返回当前会话负载信息。

  参数：pid integer

  返回值类型：datid oid, threadid bigint, sessionid bigint, threadpid integer, usesysid oid, appname text, query text, priority bigint, block_time bigint, elapsed_time bigint, total_cpu_time bigint, skew_percent integer, statement_mem integer, active_points integer, dop_value integer, current_cgroup text, current_status text, enqueue_state text, attribute text, is_plana boolean, node_group text, srespool name

- pg_stat_get_wlm_ec_operator_info

  描述：从内部哈希表中获取 EC 执行计划算子信息。

  参数：nan

  返回值类型：queryid bigint, plan_node_id integer, plan_node_name text, start_time timestamp with time zone, duration bigint, tuple_processed bigint, min_peak_memory integer, max_peak_memory integer, average_peak_memory integer, ec_operator integer, ec_status text, ec_execute_datanode text, ec_dsn text, ec_username text, ec_query text, ec_libodbc_type text, ec_fetch_count bigint

- pg_stat_get_wlm_instance_info

  描述：返回当前实例负载信息。

  参数：nan

  返回值类型：instancename text, timestamp timestamp with time zone, used_cpu integer, free_memory integer, used_memory integer, io_await double precision, io_util double precision, disk_read double precision, disk_write double precision, process_read bigint, process_write bigint,

logical_read bigint, logical_write bigint, read_counts bigint, write_counts bigint

- pg_stat_get_wlm_instance_info_with_cleanup

  描述：返回当前实例负载信息，并且保存到系统表中。

  参数：nan

  返回值类型：instancename text, timestamp timestamp with time zone, used_cpu integer, free_memory integer, used_memory integer, io_await double precision, io_util double precision, disk_read double precision, disk_write double precision, process_read bigint, process_write bigint, logical_read bigint, logical_write bigint, read_counts bigint, write_counts bigint

- pg_stat_get_wlm_realtime_session_info

  描述：返回实时会话负载信息。

  参数：nan

  返回值类型：nodename text, threadid bigint, block_time bigint, duration bigint, estimate_total_time bigint, estimate_left_time bigint, schemaname text, query_band text, spill_info text, control_group text, estimate_memory integer, min_peak_memory integer, max_peak_memory integer, average_peak_memory integer, memory_skew_percent integer, min_spill_size integer, max_spill_size integer, average_spill_size integer, spill_skew_percent integer, min_dn_time bigint, max_dn_time bigint, average_dn_time bigint, dntime_skew_percent integer, min_cpu_time bigint, max_cpu_time bigint, total_cpu_time bigint, cpu_skew_percent integer, min_peak_iops integer, max_peak_iops integer, average_peak_iops integer, iops_skew_percent integer, warning text, query text, query_plan text, cpu_top1_node_name text, cpu_top2_node_name text, cpu_top3_node_name text, cpu_top4_node_name text, cpu_top5_node_name text, mem_top1_node_name text, mem_top2_node_name text, mem_top3_node_name text, mem_top4_node_name text, mem_top5_node_name text, cpu_top1_value bigint, cpu_top2_value bigint, cpu_top3_value bigint, cpu_top4_value bigint, cpu_top5_value bigint, mem_top1_value bigint, mem_top2_value bigint, mem_top3_value bigint, mem_top4_value bigint, mem_top5_value bigint, top_mem_dn text, top_cpu_dn text

- pg_stat_get_wlm_session_iostat_info

  描述：返回会话负载 IO 信息。

  参数：nan

返回值类型：threadid bigint, maxcurr_iops integer, mincurr_iops integer, maxpeak_iops integer, minpeak_iops integer, iops_limits integer, io_priority integer, curr_io_limits integer

- pg_stat_get_wlm_statistics

  描述：返回会话负载统计数据。

  参数：nan

  返回值类型：statement text, block_time bigint, elapsed_time bigint, total_cpu_time bigint, qualification_time bigint, skew_percent integer, control_group text, status text, action text

5.26 系统管理函数

5.26.1 配置设置函数

配置设置函数是可以用于查询以及修改运行时配置参数的函数。

- current_setting(setting_name)

  描述：当前的设置值。

  返回值类型：text

  备注：current_setting 用于以查询形式获取 setting_name 的当前值。和 SQL 语句 SHOW 是等效的。比如：

```
postgres=# SELECT current_setting('datestyle');
current_setting
-----------------
 ISO, MDY
(1 row)
```

- set_working_grand_version_num_manually(tmp_version)

  描述：通过切换授权版本号来更新和升级数据库的新特性。

  返回值类型：void

- shell_in(type)

  描述：为 shell 类型输入路由（那些尚未填充的类型）。

  返回值类型：void

- shell_out(type)

描述：为 shell 类型输出路由（那些尚未填充的类型）。

返回值类型：void

- set_config(setting_name, new_value, is_local)

描述：设置参数并返回新值。

返回值类型：text

备注：set_config 将参数 setting_name 设置为 new_value。如果 is_local 为 true，则 new_value 将只应用于当前事务。如果希望 new_value 应用于当前会话，可以使用 false，和 SQL 语句 SET 是等效的。例如：

```
postgres=# SELECT set_config('log_statement_stats', 'off', false);
set_config
------------
 off
(1 row)
```

5.26.2 通用文件访问函数

通用文件访问函数提供了对数据库服务器上的文件的本地访问接口。只有 GBase 8s 目录和 log_directory 目录里面的文件可以访问。使用相对路径访问 GBase 8s 目录里面的文件，以及匹配 log_directory 配置而设置的路径访问日志文件。只有数据库初始化用户才能使用这些函数。

- pg_ls_dir(dirname text)

描述：列出目录中的文件。

返回值类型：setof text

备注：pg_ls_dir 返回指定目录里面的除了特殊项“.”和“..”之外所有名称。

示例：

```
postgres=# SELECT pg_ls_dir('./');
      pg_ls_dir
----------------------
 global
 pg_xlog
 pg_clog
 pg_csnlog
 pg_notify
```

```
pg_serial
pg_snapshots
pg_twophase
pg_multixact
base
pg_replslot
pg_tblspc
pg_stat_tmp
pg_llog
pg_errorinfo
undo
pg_logical
pg_location
PG_VERSION
pg_ctl.lock
postgresql.conf.lock
postmaster.pid.lock
mot.conf
gs_gazelle.conf
pg_hba.conf
pg_ident.conf
postgresql.conf.bak
server.crt
server.key
cacert.pem
server.key.cipher
server.key.rand
pg_hba.conf.lock
pg_hba.conf.bak
gaussdb.state
postmaster.opts
gswlm_userinfo.cfg
postgresql.conf
postmaster.pid
(39 rows)
```

- pg_read_file(filename text, offset bigint, length bigint)

  描述：返回一个文本文件的内容。

  返回值类型：text

  备注：pg_read_file 返回一个文本文件的一部分，从 offset 开始，最多返回 length 字节

（如果先达到文件结尾，则小于这个数值）。如果 offset 是负数，则它是相对于文件结尾回退的长度。如果省略了 offset 和 length，则返回整个文件。

示例：

```
postgres=# SELECT pg_read_file('postmaster.pid',0,100);
              pg_read_file
----------------------------------------
7961                                    +
/home/gbase/project/install/data/dn/dn1_1+
1651723153                              +
20008                                   +
/home/gbase/gbase8s/tmp                 +
[local_ip]                              +
 2
(1 row)
```

- pg_read_binary_file(filename text [, offset bigint, length bigint,missing_ok boolean])

  描述：返回一个二进制文件的内容。

  返回值类型：bytea

  备注：pg_read_binary_file 的功能与 pg_read_file 类似，除了结果的返回值为 bytea 类型不一致，相应地不会执行编码检查。与 convert_from 函数结合，这个函数可以用来读取用指定编码的一个文件。

```
SELECT convert_from(pg_read_binary_file('filename'), 'UTF8');
```

- pg_stat_file(filename text)

  描述：返回一个文本文件的状态信息。

  返回值类型：record

  备注：pg_stat_file 返回一条记录，其中包含：文件大小、最后访问时间戳、最后更改时间戳、最后文件状态修改时间戳以及标识传入参数是否为目录的 Boolean 值。典型的用法：

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

示例：

```
postgres=# SELECT convert_from(pg_read_binary_file('postmaster.pid'),
'UTF8');
              convert_from
```

```
----------------------------------------
 7961                                  +
 /home/gbase/project/install/data/dn/dn1_1+
 1651723153                            +
 20008                                 +
 /home/gbase/gbase8s/tmp               +
 [local_ip]                             +
  20008001         1                   +

(1 row)
postgres=# SELECT * FROM pg_stat_file('postmaster.pid');
size |         access         |      modification      |
change         | creation | isdir
------+------------------------+------------------------+---------
--------------+----------+-------
  118 | 2022-05-18 11:59:29+08 | 2022-05-05 11:59:13+08 | 2022-05-
05 11:59:13+08 |          | f
(1 row)
 postgres=# SELECT (pg_stat_file('postmaster.pid')).modification;
      modification
-----------------------
 2022-05-05 11:59:13+08
(1 row)
```

5.26.3 服务器信号函数

服务器信号函数向其他服务器进程发送控制信号。只有系统管理员才能使用这些函数。

- pg_cancel_backend(pid int)

  描述：取消一个后端的当前查询。

  返回值类型：Boolean

  备注：pg_cancel_backend 向由 pid 标识的后端进程发送一个查询取消（SIGINT）信号。一个活动的后端进程的 PID 可以从 pg_stat_activity 视图的 pid 字段找到，或者在服务器上用 ps 列出数据库进程。具有 SYSADMIN 权限的用户，后端进程所连接的数据库的属主，后端进程的属主或者继承了内置角色 gs_role_signal_backend 权限的用户有权使用该函数。

- pg_reload_conf()

  描述：导致所有服务器进程重新装载它们的配置文件（需要系统管理员角色）。

返回值类型：Boolean

备注：pg_reload_conf 给服务器发送一个 SIGHUP 信号，导致所有服务器进程重新装载配置文件。

- pg_rotate_logfile()

描述：滚动服务器的日志文件（需要系统管理员角色）。

返回值类型：Boolean

备注：pg_rotate_logfile 给日志文件管理器发送信号，告诉它立即切换到一个新的输出文件。这个函数只有在 redirect_stderr 用于日志输出的时候才有用，否则根本不存在日志文件管理器子进程。

- pg_terminate_backend(pid int)

描述：终止一个后台线程。

返回值类型：Boolean

备注：如果成功，函数返回 true，否则返回 false。具有 SYSADMIN 权限的用户，后端进程所连接的数据库的属主，后端进程的属主或者继承了内置角色 gs_role_signal_backend 权限的用户有权使用该函数。

示例：

```
postgres=# SELECT pid from pg_stat_activity;
      pid
-----------------
 140573611915008
 140573668599552
 140574052771584
 140573954000640
 140574121588480
 140574004344576
 140573970781952
 140573987563264
(8 rows)

postgres=# SELECT pg_terminate_backend(140573987563264);
 pg_terminate_backend
----------------------
 t
```

```
(1 row)
```

- pg_terminate_session(pid int64, sessionid int64)

  描述：终止一个后台 session。

  返回值类型：Boolean

  备注：如果成功，函数返回 true，否则返回 false。具有 SYSADMIN 权限的用户，会话所连接的数据库的属主，会话的属主或者继承了内置角色 gs_role_signal_backend 权限的用户有权使用该函数。

5.26.4 备份恢复控制函数

5.26.4.1 备份控制函数

备份控制函数可帮助进行在线备份。

- pg_create_restore_point(name text)

  描述：为执行恢复创建一个命名点。（需要管理员角色）

  返回值类型：text

  备注：pg_create_restore_point 创建了一个可以用作恢复目的、有命名的事务日志记录，并返回相应的事务日志位置。在恢复过程中，recovery_target_name 可以通过这个名称定位对应的日志恢复点，并从此处开始执行恢复操作。避免使用相同的名称创建多个恢复点，因为恢复操作将在第一个匹配（恢复目标）的名称上停止。

- pg_current_xlog_location()

  描述：获取当前事务日志的写入位置。

  返回值类型：text

  备注：pg_current_xlog_location 使用与前面那些函数相同的格式显示当前事务日志的写入位置。如果是只读操作，不需要系统管理员权限。

- pg_current_xlog_insert_location()

  描述：获取当前事务日志的插入位置。

  返回值类型：text

  备注：pg_current_xlog_insert_location 显示当前事务日志的插入位置。插入点是事务日

志在某个瞬间的“逻辑终点”，而实际的写入位置则是从服务器内部缓冲区写出时的终点。写入位置是可以从服务器外部检测到的终点，如果要归档部分完成事务日志文件，则该操作即可实现。插入点主要用于服务器调试目的。如果是只读操作，不需要系统管理员权限。

- gs_current_xlog_insert_end_location()

  描述：获取当前事务日志的插入位置。

  返回值类型：text

  备注：gs_current_xlog_insert_end_location 显示当前事务日志的实际插入位置。

- pg_start_backup(label text [, fast boolean])

  描述：开始执行在线备份。（需要管理员角色或复制的角色）

  返回值类型：text

  备注：pg_start_backup 接受一个用户定义的备份标签（通常这是备份转储文件存放地点的名称）。这个函数向 GBase 8s 的数据目录写入一个备份标签文件，然后以文本方式返回备份的事务日志起始位置。

```
 postgres=# SELECT pg_start_backup('label_goes_here');
pg_start_backup
-----------------
 0/20000028
(1 row)
```

- pg_stop_backup()

  描述：完成执行在线备份。（需要管理员角色或复制的角色）

  返回值类型：text

  备注：pg_stop_backup 删除 pg_start_backup 创建的标签文件，并且在事务日志归档区里创建一个备份历史文件。这个历史文件包含给予 pg_start_backup 的标签、备份的事务日志起始与终止位置、备份的起始和终止时间。返回值是备份的事务日志终止位置。计算出中止位置后，当前事务日志的插入点将自动前进到下一个事务日志文件，这样，结束的事务日志文件可以被立即归档从而完成备份。

- pg_switch_xlog()

  描述：切换到一个新的事务日志文件。（需要管理员角色）

  返回值类型：text

备注：pg_switch_xlog 移动到下一个事务日志文件，以允许将当前日志文件归档（假定使用连续归档）。返回值是刚完成的事务日志文件的事务日志结束位置+1。如果从最后一次事务日志切换以来没有活动的事务日志，则 pg_switch_xlog 什么事也不做，直接返回当前事务日志文件的开始位置。

- pg_xlogfile_name(location text)

  描述：将事务日志的位置字符串转换为文件名。

  返回值类型：text

  备注：pg_xlogfile_name 仅抽取事务日志文件名称。如果给定的事务日志位置恰好位于事务日志文件的交界上，这两个函数都返回前一个事务日志文件的名称。这对于管理事务日志归档来说是非常有利的，因为前一个文件是当前最后一个需要归档的文件。

- pg_xlogfile_name_offset(location text)

  描述：将事务日志的位置字符串转换为文件名并返回在文件中的字节偏移量。

  返回值类型：text,integer

  备注：可以使用 pg_xlogfile_name_offset 从前述函数的返回结果中抽取相应的事务日志文件名称和字节偏移量。例如：

```
 postgres=# SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
NOTICE:  WAL archiving is not enabled; you must ensure that all required WAL
segments are copied through other means to complete the backup
        file_name         | file_offset
--------------------------+-------------
 000000010000000000000020 |        3616
(1 row)
```

- pg_xlog_location_diff(location text, location text)

  描述：计算两个事务日志位置之间在字节上的区别。

  返回值类型：numeric

- pg_cbm_tracked_location()

  描述：用于查询 cbm 解析到的 lsn 位置。

  返回值类型：text

- pg_cbm_get_merged_file(startLSNArg text, endLSNArg text)

描述：用于将指定 lsn 范围之内的 cbm 文件合并成一个 cbm 文件，并返回合并完的 cbm 文件名。

返回值类型：text

备注：必须是系统管理员或运维管理员才能获取 cbm 合并文件。

- pg_cbm_get_changed_block(startLSNArg text, endLSNArg text)

  描述：用于将指定 lsn 范围之内的 cbm 文件合并成一个表，并返回表的各行记录。

  返回值类型：records

  备注：pg_cbm_get_changed_block 返回的表字段包含：合并起始的 lsn、合并截止的 lsn、表空间 oid、库 oid、表的 relfilenode、表的 fork number、表是否被删除、表是否被创建、表是否被截断、表被截断后的页面数、有多少页被修改以及被修改的页号的列表。

- pg_cbm_recycle_file(targetLSNArg text)

  描述：删除不再使用的 cbm 文件，并返回删除后的第一条 lsn。

  返回值类型：text

- pg_cbm_force_track(targetLSNArg text,timeOut int)

  描述：强制执行一次 cbm 追踪到指定的 xlog 位置，并返回实际追踪结束点的 xlog 位置。

  返回值类型：text

- pg_enable_delay_ddl_recycle()

  描述：开启延迟 DDL 功能，并返回开启点的 xlog 位置。需要管理员角色或运维管理员角色打开 operate_mode。

  返回值类型：text

- pg_disable_delay_ddl_recycle(barrierLSNArg text, isForce bool)

  描述：关闭延迟 DDL 功能，并返回本次延迟 DDL 生效的 xlog 范围。需要管理员角色或运维管理员角色打开 operate_mode。

  返回值类型：records

- pg_enable_delay_xlog_recycle()

  描述：开启延迟 xlog 回收功能，数据库主节点修复使用。

返回值类型：void

- pg_disable_delay_xlog_recycle()

描述：关闭延迟 xlog 回收功能，数据库主节点修复使用。

返回值类型：void

- pg_cbm_rotate_file(rotate_lsn text)

描述：等待 cbm 解析到 rotate_lsn 之后，强制切换文件，在 build 期间调用。

返回值类型：void。

- gs_roach_stop_backup(backupid text)

描述：停止一个内部备份工具 GaussRoach 开启的备份。与 pg_stop_backup 系统函数类似，但更轻量。

返回值类型：text，内容为当前日志的插入位置。

备注：目前 GBase 8s 不支持。

- gs_roach_enable_delay_ddl_recycle(backupid name)

描述：开启延迟 DDL 功能，并返回开启点的日志位置。与 pg_enable_delay_ddl_recycle 系统函数类似，但更轻量。并且，通过传入不同的 backupid，可以支持并发打开延迟 DDL。

返回值类型：text，内容为返回开启点的日志位置。

备注：目前 GBase 8s 不支持。

- gs_roach_disable_delay_ddl_recycle(backupid text)

描述：关闭延迟 DDL 功能，并返回本次延迟 DDL 生效的日志范围，并删除该范围内被用户删除的列存表物理文件。与 pg_enable_delay_ddl_recycle 系统函数类似，但更轻量。并且，通过传入不同的 backupid，可以支持并发关闭延迟 DDL 功能。

返回值类型：records，内容为本次延迟 DDL 生效的日志范围。

备注：目前 GBase 8s 不支持。

- gs_roach_switch_xlog(request_ckpt bool)

描述：切换当前使用的日志段文件，并且，如果 request_ckpt 为 true，则触发一个全量检查点。

返回值类型：text，内容为切段日志的位置。

备注：目前 GBase 8s 不支持。

5.26.4.2 恢复控制函数

恢复信息函数提供了当前备机状态的信息。这些函数可能在恢复期间或正常运行中执行。

- pg_is_in_recovery()

描述：如果恢复仍然在进行中则返回 true。

返回值类型：bool

- pg_last_xlog_receive_location()

描述：获取最后接收事务日志的位置并通过流复制将其同步到磁盘。当流复制正在进行时，事务日志将持续递增。如果恢复已完成，则最后一次获取的 WAL 记录会被静态保持并在恢复过程中同步到磁盘。如果流复制不可用，或还没有开始，这个函数返回 NULL。

返回值类型：text

- pg_last_xlog_replay_location()

描述：获取最后一个事务日志在恢复时重放的位置。如果恢复仍在进行，事务日志将持续递增。如果已经完成恢复，则将保持在恢复期间最后接收 WAL 记录的值。如果未进行恢复但服务器正常启动时，则这个函数返回 NULL。

返回值类型：text

- pg_last_xact_replay_timestamp()

描述：获取最后一个事务在恢复时重放的时间戳。这是为在主节点上生成事务提交或终止 WAL 记录的时间。如果在恢复时没有事务重放，则这个函数返回 NULL。如果恢复仍在进行，则事务日志将持续递增。如果恢复已经完成，则将保持在恢复期间最后接收 WAL 记录的值。如果服务器无需恢复就已正常启动，则这个函数返回 NULL。

返回值类型：timestamp with time zone

恢复控制函数控制恢复的进程。这些函数可能只在恢复时被执行。

- pg_is_xlog_replay_paused()

描述：如果恢复暂停则返回 true。

返回值类型：bool

- pg_xlog_replay_pause()

  描述：立即暂停恢复。

  返回值类型：void

- pg_xlog_replay_resume()

  描述：如果恢复处于暂停状态，则重新启动。

  返回值类型：void

当恢复暂停时，没有发生数据库更改。如果是在热备里，所有新的查询将看到一致的数据库快照，并且不会有进一步的查询冲突产生，直到恢复继续。

如果不能使用流复制，则暂停状态将无限的延续。当流复制正在进行时，将连续接收 WAL 记录，最终将填满可用磁盘空间，这个进度取决于暂停的持续时间，WAL 生成的速度和可用的磁盘空间。

5.26.5 快照同步函数

快照同步函数是导出当前快照的标识符。

- pg_export_snapshot()

  描述：保存当前的快照并返回它的标识符。

  返回值类型：text

  备注：函数 pg_export_snapshot 保存当前的快照并返回一个文本字符串标识此快照。这个字符串必须传递给想要导入快照的客户端。可用在 set transaction snapshot snapshot_id 时导入 snapshot，但是应用的前提是该事务设置了 SERIALIZABLE 或 REPEATABLE READ 隔离级别。而 GBase 8s 目前是不支持这两种隔离级别的。该函数的输出不可用做 set transaction snapshot 的输入。

- pg_export_snapshot_and_csn()

  描述：保存当前的快照并返回它的标识符。比 pg_export_snapshot()多返回一列 CSN，表示当前快照的 CSN。

  返回值类型：text

5.26.6 数据库对象函数

5.26.6.1 数据库对象尺寸函数

数据库对象尺寸函数计算数据库对象使用的实际磁盘空间。

- pg_column_size(any)

  描述：存储一个指定的数值需要的字节数（可能压缩过）。

  返回值类型：int

  备注：pg_column_size 显示用于存储某个独立数据值的空间。

```
postgres=# SELECT pg_column_size(1);
 pg_column_size
----------------
              4
(1 row)
```

- pg_database_size(oid)

  描述：指定 OID 代表的数据库使用的磁盘空间。

  返回值类型：bigint

- pg_database_size(name)

  描述：指定名称的数据库使用的磁盘空间。

  返回值类型：bigint

  备注：pg_database_size 接受一个数据库的 OID 或者名称，然后返回该对象使用的全部磁盘空间。

  示例：

```
postgres=# SELECT pg_database_size('postgres');
 pg_database_size
------------------
         30840684
(1 row)
```

- pg_relation_size(oid)

  描述：指定 OID 代表的表或者索引所使用的磁盘空间。

返回值类型：bigint

- get_db_source_datasize()

描述：估算当前数据库非压缩态的数据总容量。

返回值类型：bigint

备注：（1）调用该函数前需要做 analyze；（2）通过估算列存的压缩率计算非压缩态的数据总容量。

示例：

```
postgres=# analyze;
ANALYZE
postgres=# select get_db_source_datasize();
 get_db_source_datasize
------------------------
               31029100
(1 row)
```

- pg_relation_size(text)

描述：指定名称的表或者索引使用的磁盘空间。表名称可以用模式名修饰。

返回值类型：bigint

- pg_relation_size(relation regclass, fork text)

描述：指定表或索引的指定分叉树（'main'，'fsm'或'vm'）使用的磁盘空间。

返回值类型：bigint

- pg_relation_size(relation regclass)

描述：pg_relation_size(…, 'main')的简写。

返回值类型：bigint

备注：pg_relation_size 接受一个表、索引、压缩表的 OID 或者名称，然后返回它们的字节大小。

- pg_partition_size(oid,oid)

描述：指定 OID 代表的分区使用的磁盘空间。其中，第一个 oid 为表的 OID，第二个 oid 为分区的 OID。

返回值类型：bigint

- pg_partition_size(text, text)

描述：指定名称的分区使用的磁盘空间。其中，第一个 text 为表名，第二个 text 为分区名。

返回值类型：bigint

- pg_partition_indexes_size(oid,oid)

描述：指定 OID 代表的分区的索引使用的磁盘空间。其中，第一个 oid 为表的 OID，第二个 oid 为分区的 OID。

返回值类型：bigint

- pg_partition_indexes_size(text,text)

描述：指定名称的分区的索引使用的磁盘空间。其中，第一个 text 为表名，第二个 text 为分区名。

返回值类型：bigint

- pg_indexes_size(regclass)

描述：附加到指定表的索引使用的总磁盘空间。

返回值类型：bigint

- pg_size_pretty(bigint)

描述：将以 64 位整数表示的字节值转换为具有单位的易读格式。

返回值类型：text

- pg_size_pretty(numeric)

描述：将以数值表示的字节值转换为具有单位的易读格式。

返回值类型：text

备注：pg_size_pretty 用于把其他函数的结果格式化成一种易读的格式，可以根据情况使用 KB 、MB 、GB 、TB。

- pg_table_size(regclass)

描述：指定的表使用的磁盘空间，不计索引（但是包含 TOAST，自由空间映射和可见

性映射）。

返回值类型：bigint

- pg_tablespace_size(oid)

描述：指定 OID 代表的表空间使用的磁盘空间。

返回值类型：bigint

- pg_tablespace_size(name)

描述：指定名称的表空间使用的磁盘空间。

返回值类型：bigint

备注：pg_tablespace_size 接受一个数据库的 OID 或者名称，然后返回该对象使用的全部磁盘空间。

- pg_total_relation_size(oid)

描述：指定 OID 代表的表使用的磁盘空间，包括索引和压缩数据。

返回值类型：bigint

- pg_total_relation_size(regclass)

描述：指定的表使用的总磁盘空间，包括所有的索引和 TOAST 数据。

返回值类型：bigint

- pg_total_relation_size(text)

描述：指定名称的表所使用的全部磁盘空间，包括索引和压缩数据。表名称可以用模式名修饰。

返回值类型：bigint

备注：pg_total_relation_size 接受一个表或者一个压缩表的 OID 或者名称，然后返回以字节计的数据和所有相关的索引和压缩表的尺寸。

- datalength(any)

描述：计算一个指定的数据需要的字节数（不考虑数据的管理空间和数据压缩、数据类型转换等情况）。

返回值类型：int

备注：datalength 用于计算某个独立数据值的空间。

示例：

```
 postgres=# SELECT datalength(1);
datalength
------------
4
(1 row)
```

目前支持的数据类型及计算方式见下表。

表 5-26 数据类型及计算方式

数据类型			存储空间
值类型	整数类型	TINYINT	1
		SMALLINT	2
		INTEGER	4
		BINARY_INTEGER	4
		BIGINT	8
	任意精度型	DECIMAL	每 4 位十进制数占两个字节，小数点前后数字分别计算
		NUMERIC	每 4 位十进制数占两个字节，小数点前后数字分别计算
		NUMBER	每 4 位十进制数占两个字节，小数点前后数字分别计算
	序列整型	SMALLSERIAL	2
		SERIAL	4
		LARGESERIAL	8

		BIGSERIAL	每 4 位十进制数占两个字节，小数点前后数字分别计算
	浮点类型	FLOAT4	4
		DOUBLE PRECISION	8
		FLOAT8	8
		BINARY_DOUBLE	8
		FLOAT[(p)]	每 4 位十进制数占两个字节，小数点前后数字分别计算
		DEC[(p[,s])]	每 4 位十进制数占两个字节，小数点前后数字分别计算
		INTEGER[(p[,s])]	每 4 位十进制数占两个字节，小数点前后数字分别计算
布尔类型	布尔类型	BOOLEAN	1
字符类型	字符类型	CHAR	n
		CHAR(n)	n
		CHARACTER(n)	n
		NCHAR(n)	n
		VARCHAR(n)	n
		CHARACTER	字符实际字节数
		VARYING(n)	字符实际字节数
		VARCHAR2(n)	字符实际字节数

		NVARCHAR(n)	字符实 际字节数
		NVARCHAR2(n)	字符实际字节数
		TEXT	字符实际字节数
		CLOB	字符实际字节数
时间类型	时间类型	DATE	8
		TIME	8
		TIMEZ	12
		TIMESTAMP	8
		TIMESTAMPZ	8
		SMALLDATETIME	8
		INTERVAL DAY TO SECOND	16
		INTERVAL	16
		RELTIME	4
		ABSTIME	4
		TINTERVAL	12

5.26.6.2 数据库对象位置函数

- pg_relation_filenode(relation regclass)

  描述：指定关系的文件节点数。

  返回值类型：oid

  备注：pg_relation_filenode 接受一个表、索引、序列或压缩表的 OID 或者名称，并且返回当前分配给它的“filenode”数。文件节点是关系使用的文件名称的基本组件。对大多数

表来说，结果和 pg_class.relfilenode 相同，但对确定的系统目录来说，relfilenode 为 0 而且这个函数必须用来获取正确的值。如果传递一个没有存储的关系，比如一个视图，那么这个函数返回 NULL。

- pg_relation_filepath(relation regclass)

  描述：指定关系的文件路径名。

  返回值类型：text

  备注：pg_relation_filepath 类似于 pg_relation_filenode，但是它返回关系的整个文件路径名（相对于 GBase 8s 的数据目录 PGDATA）。

- pg_filenode_relation(tablespace oid, filenode oid)

  描述：获取对应的 tablespace 和 relfilenode 所对应的表名。

  返回类型：regclass

- pg_partition_filenode(partition_oid)

  描述：获取到指定分区表的 oid 锁对应的 filenode。

  返回类型：oid

- pg_partition_filepath(partition_oid)

  描述：指定分区的文件路径名。

  返回值类型：text

5.26.6.3 回收站对象函数

- gs_is_recycle_object(classid, objid, objname)

  描述：判断是否为回收站对象。

  返回值类型：bool

5.26.7 咨询锁函数

咨询锁函数用于管理咨询锁（Advisory Lock）。

- pg_advisory_lock(key bigint)

  描述：获取会话级别的排它咨询锁。

返回值类型：void

备注：pg_advisory_lock 锁定应用程序定义的资源，该资源可以用一个 64 位或两个不重叠的 32 位键值标识。如果已经有另外的会话锁定了该资源，则该函数将阻塞到该资源可用为止。这个锁是排它的。多个锁定请求将会被压入栈中，因此，如果同一个资源被锁定了三次，它必须被解锁三次以将资源释放给其他会话使用。

- pg_advisory_lock(key1 int, key2 int)

  描述：获取会话级别的排它咨询锁。

  返回值类型：void

  备注：只允许 sysadmin 对键值对(65535, 65535)加会话级别的排它咨询锁，普通用户无权限。

- pg_advisory_lock(int4, int4, Name)

  描述：获取指定数据库的排它咨询锁。

  返回值类型：void

- pg_advisory_lock_shared(key bigint)

  描述：获取会话级别的共享咨询锁。

  返回值类型：void

- pg_advisory_lock_shared(key1 int, key2 int)

  描述：获取会话级别的共享咨询锁。

  返回值类型：void

  备注：pg_advisory_lock_shared 类似于 pg_advisory_lock，不同之处仅在于共享锁会话可以和其他请求共享锁的会话共享资源，但排它锁除外。

- pg_advisory_unlock(key bigint)

  描述：释放会话级别的排它咨询锁。

  返回值类型：Boolean

- pg_advisory_unlock(key1 int, key2 int)

  描述：释放会话级别的排它咨询锁。

返回值类型：Boolean

备注：pg_advisory_unlock 释放先前取得的排它咨询锁。如果释放成功则返回 true。如果实际上并未持有指定的锁，将返回 false 并在服务器中产生一条 SQL 警告信息。

- pg_advisory_unlock(int4, int4, Name)

  描述：释放指定数据库上的排它咨询锁。

  返回值类型：Boolean

  备注：如果释放成功则返回 true；如果未持有锁，则返回 false。

- pg_advisory_unlock_shared(key bigint)

  描述：释放会话级别的共享咨询锁。

  返回值类型：Boolean

- pg_advisory_unlock_shared(key1 int, key2 int)

  描述：释放会话级别的共享咨询锁。

  返回值类型：Boolean

  备注：pg_advisory_unlock_shared 类似于 pg_advisory_unlock，不同之处在于该函数释放的是共享咨询锁。

- pg_advisory_unlock_all()

  描述：释放当前会话持有的所有咨询锁。

  返回值类型：void

  备注：pg_advisory_unlock_all 将会释放当前会话持有的所有咨询锁，该函数在会话结束的时候被隐含调用，即使客户端异常地断开连接也是一样。

- pg_advisory_xact_lock(key bigint)

  描述：获取事务级别的排它咨询锁。

  返回值类型：void

- pg_advisory_xact_lock(key1 int, key2 int)

  描述：获取事务级别的排它咨询锁。

  返回值类型：void

备注：pg_advisory_xact_lock 类似于 pg_advisory_lock，不同之处在于锁是自动在当前事务结束时释放，而且不能被显式的释放。只允许 sysadmin 对键值对(65535, 65535)加事务级别的排它咨询锁，普通用户无权限。

- pg_advisory_xact_lock_shared(key bigint)

  描述：获取事务级别的共享咨询锁。

  返回值类型：void

- pg_advisory_xact_lock_shared(key1 int, key2 int)

  描述：获取事务级别的共享咨询锁。

  返回值类型：void

  备注：pg_advisory_xact_lock_shared 类似于 pg_advisory_lock_shared，不同之处在于锁是在当前事务结束时自动释放，而且不能被显式的释放。

- pg_try_advisory_lock(key bigint)

  描述：尝试获取会话级排它咨询锁。

  返回值类型：Boolean

  备注：pg_try_advisory_lock 类似于 pg_advisory_lock，不同之处在于该函数不会阻塞以等待资源的释放。它要么立即获得锁并返回 true，要么返回 false 表示目前不能锁定。

- pg_try_advisory_lock(key1 int, key2 int)

  描述：尝试获取会话级排它咨询锁。

  返回值类型：Boolean

  备注：只允许 sysadmin 对键值对(65535, 65535)加会话级别的排它咨询锁，普通用户无权限。

- pg_try_advisory_lock_shared(key bigint)

  描述：尝试获取会话级共享咨询锁。

  返回值类型：Boolean

- pg_try_advisory_lock_shared(key1 int, key2 int)

  描述：尝试获取会话级共享咨询锁。

返回值类型：Boolean

备注：pg_try_advisory_lock_shared 类似于 pg_try_advisory_lock，不同之处在于该函数尝试获得共享锁而不是排它锁。

- pg_try_advisory_xact_lock(key bigint)

  描述：尝试获取事务级别的排它咨询锁。

  返回值类型：Boolean

- pg_try_advisory_xact_lock(key1 int, key2 int)

  描述：尝试获取事务级别的排它咨询锁。

  返回值类型：Boolean

  备注：pg_try_advisory_xact_lock 类似于 pg_try_advisory_lock，不同之处在于如果得到锁，在当前事务的结束时自动释放，而且不能被显式的释放。只允许 sysadmin 对键值对(65535, 65535)加事务级别的排它咨询锁，普通用户无权限。

- pg_try_advisory_xact_lock_shared(key bigint)

  描述：尝试获取事务级别的共享咨询锁。

  返回值类型：Boolean

- pg_try_advisory_xact_lock_shared(key1 int, key2 int)

  描述：尝试获取事务级别的共享咨询锁。

  返回值类型：Boolean

  备注：pg_try_advisory_xact_lock_shared 类似于 pg_try_advisory_lock_shared，不同之处在于如果得到锁，在当前事务结束时自动释放，而且不能被显式的释放。

- lock_cluster_ddl()

  描述：尝试对 GBase 8s 内所有存活的数据库主节点获取会话级别的排他咨询锁。

  返回值类型：Boolean

  备注：只允许 sysadmin 调用，普通用户无权限。

- unlock_cluster_ddl()

  描述：尝试对数据库主节点会话级别的排他咨询锁。

返回值类型：Boolean

5.26.8 逻辑复制函数

- pg_create_logical_replication_slot('slot_name', 'plugin_name')

  描述：创建逻辑复制槽。

  参数说明：

  - slot_name ：流复制槽名称。

    取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

  - plugin_name

    插件名称。

    取值范围：字符串，当前支持 mppdb_decoding。

  返回值类型：name, text

  备注：第一个返回值表示 slot_name，第二个返回值表示该逻辑复制槽解码的起始 LSN 位置。调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。此函数目前只支持在主机调用。

- pg_create_physical_replication_slot('slot_name', 'isDummyStandby')

  描述：创建新的物理复制槽。

  参数说明：

  - slot_name ：流复制槽名称。

    取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

  - isDummyStandby

    是否是从从备连接主机创建的复制槽。

    类型：bool。

  返回值类型：name, text

  备注：调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。目前默认不支持主备从部署模式。

- pg_drop_replication_slot('slot_name')

  描述：删除流复制槽。

  参数说明：

  - slot_name ：流复制槽名称。

    取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

  返回值类型：void

  备注：调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。此函数目前只支持在主机调用。

- pg_logical_slot_peek_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')

  描述：解码并不推进流复制槽（下次解码可以再次获取本次解出的数据）。

  参数说明：

  - slot_name

    流复制槽名称。

    取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

  - LSN

    日志的 LSN，表示只解码小于等于此 LSN 的日志。

    取值范围：字符串（LSN，格式为 xlogid/xrecoff），如'1/2AAFC60'。为 NULL 时表示不对解码截止的日志位置做限制。

  - upto_nchanges

    解码条数（包含 begin 和 commit）。假设一共有三条事务，分别包含 3、5、7 条记录，如果 upto_nchanges 为 4，那么会解码出前两个事务共 8 条记录。解码完第二条事务时发现解码条数记录大于等于 upto_nchanges，会停止解码。

    取值范围：非负整数。

    说明

    LSN 和 upto_nchanges 中任一参数达到限制，解码都会结束。

options：此项为可选参数，由一系列 options_name 和 options_value 一一对应组成。

- include-xids

  解码出的 data 列是否包含 xid 信息。

  取值范围：0 或 1，默认值为 1。

  0：设为 0 时，解码出的 data 列不包含 xid 信息。

  1：设为 1 时，解码出的 data 列包含 xid 信息。

- skip-empty-xacts

  解码时是否忽略空事务信息。

  取值范围：0 或 1，默认值为 0。

  0：设为 0 时，解码时不忽略空事务信息。

  1：设为 1 时，解码时会忽略空事务信息。

- include-timestamp

  解码信息是否包含 commit 时间戳。

  取值范围：0 或 1，默认值为 0。

  0：设为 0 时，解码信息不包含 commit 时间戳。

  1：设为 1 时，解码信息包含 commit 时间戳。

- only-local

  是否仅解码本地日志。

  取值范围：0 或 1，默认值为 1。

  0：设为 0 时，解码非本地日志和本地日志。

  1：设为 1 时，仅解码本地日志。

- force-binary

  是否以二进制格式输出解码结果。

  取值范围：0，默认值为 0。

  0：设为 0 时，以文本格式输出解码结果。

- white-table-list

  白名单参数，包含需要进行解码的 schema 和表名。

  取值范围：包含白名单中表名的字符串，不同的表以','为分隔符进行隔离；使用'*'来模糊匹配所有情况；schema 名和表名间以'.'分割，不允许存在任意空白符。例：select * from pg_logical_slot_peek_changes('slot1', NULL, 4096, 'white-table-list', 'public.t1,public.t2');

返回值类型：text, xid, text

备注：函数返回解码结果，每一条解码结果包含三列，对应上述返回值类型，分别表示 LSN 位置、xid 和解码内容。

调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。

- pg_logical_slot_get_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')

  描述：解码并推进流复制槽。

  参数说明：与 pg_logical_slot_peek_changes 一致，详细内容请参见• pg_logical_slot_peek_ch•••。

  备注：调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。此函数目前只支持在主机调用。

- pg_logical_slot_peek_binary_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')

  描述：以二进制格解码且不推进流复制槽（下次解码可以再次获取本次解出的数据）。

  参数说明：

  - slot_name

    流复制槽名称。

    取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

  - LSN

    日志的 LSN，表示只解码小于等于此 LSN 的日志。

取值范围：字符串（LSN，格式为 xlogid/xrecoff），如'1/2AAFC60'。为 NULL 时表示不对解码截止的日志位置做限制。

- upto_nchanges

解码条数（包含 begin 和 commit）。假设一共有三条事务，分别包含 3、5、7 条记录，如果 upto_nchanges 为 4，那么会解码出前两个事务共 8 条记录。解码完第二条事务时发现解码条数记录大于等于 upto_nchanges，会停止解码。

取值范围：非负整数。

 说明

LSN 和 upto_nchanges 中任一参数达到限制，解码都会结束。

options：此项为可选参数，由一系列 options_name 和 options_value 一一对应组成。

- include-xids

解码出的 data 列是否包含 xid 信息。

取值范围：0 或 1，默认值为 1。

0：设为 0 时，解码出的 data 列不包含 xid 信息。

1：设为 1 时，解码出的 data 列包含 xid 信息。

- skip-empty-xacts

解码时是否忽略空事务信息。

取值范围：0 或 1，默认值为 0。

0：设为 0 时，解码时不忽略空事务信息。

1：设为 1 时，解码时会忽略空事务信息。

- include-timestamp

解码信息是否包含 commit 时间戳。

取值范围：0 或 1，默认值为 0。

0：设为 0 时，解码信息不包含 commit 时间戳。

1：设为 1 时，解码信息包含 commit 时间戳。

- only-local

是否仅解码本地日志。

取值范围：0 或 1，默认值为 1。

0：设为 0 时，解码非本地日志和本地日志。

1：设为 1 时，仅解码本地日志。

- force-binary

是否以二进制格式输出解码结果。

取值范围：0 或 1，默认值为 0，均以二进制格式输出结果。

- white-table-list

白名单参数，包含需要进行解码的 schema 和表名。

取值范围：包含白名单中表名的字符串，不同的表以','为分隔符进行隔离；使用'*'来模糊匹配所有情况；schema 名和表名间以'.'分割，不允许存在任意空白符。例：select * from pg_logical_slot_peek_binary_changes('slot1', NULL, 4096, 'white-table-list', 'public.t1,public.t2');

返回值类型：text, xid, bytea

备注：函数返回解码结果，每一条解码结果包含三列，对应上述返回值类型，分别表示 LSN 位置、xid 和二进制格式的解码内容。调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。

- pg_logical_slot_get_binary_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')

  描述：以二进制格式解码并推进流复制槽。

  参数说明：

  与 pg_logical_slot_peek_binary_changes 一致。

  备注：调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。

- pg_replication_slot_advance ('slot_name', 'LSN')

  描述：直接推进流复制槽到指定 LSN，不输出解码结果。

  参数说明：

- slot_name

流复制槽名称。

取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

- LSN

推进到的日志 LSN 位置，下次解码时只会输出提交位置比该 LSN 大的事务结果。如果输入的 LSN 比当前流复制槽记录的推进位置还要小，则直接返回；如果输入的 LSN 比当前最新物理日志 LSN 还要大，则推进到当前最新物理日志 LSN。

取值范围：字符串（LSN，格式为 xlogid/xrecoff）。

返回值类型：name, text

备注：返回值分别对应 slot_name 和实际推进至的 LSN。调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。此函数目前只支持在主机调用。

- pg_logical_get_area_changes('LSN_start', 'LSN_end', upto_nchanges, 'decoding_plugin', 'xlog_path', 'options_name', 'options_value')

描述：没有 ddl 的前提下，指定 lsn 区间进行解码，或者指定 xlog 文件进行解码。

约束条件如下：

调用接口时，需要将日志级别参数设置为 logical，即 wal_level=logical 并重启数据库生效。只有该条件期间产生的日志文件才能被解析。如果使用的 xlog 文件为非 logical 级别，则解码内容没有对应的值和类型，无其他影响。

xlog 文件只能被完全同构的 dn 的某个副本解析，确保可以找到数据对应的元信息，且没有 DDL 操作和 VACUUM FULL。

用户可以找到需要解析的 xlog。用户需要注意一次不要读入过多 xlog 文件，推荐一次一个，一个 xlog 文件估测占用内存为 xlog 文件大小的 2~3 倍。无法解码扩容前的 xlog 文件。

参数说明：

- LSN_start

指定开始解码的 lsn。

取值范围：字符串（LSN，格式为 xlogid/xrecoff），如'1/2AAFC60'。为 NULL 时

表示不对解码截止的日志位置做限制。

- LSN_end

指定解码结束的 lsn。

取值范围：字符串（LSN，格式为 xlogid/xrecoff），如'1/2AAFC60'。为 NULL 时表示不对解码截止的日志位置做限制。LSN_start 必须要小于 LSN_end，否则报错。

- upto_nchanges

解码条数（包含 begin 和 commit）。假设一共有三条事务，分别包含 3、5、7 条记录，如果 upto_nchanges 为 4，那么会解码出前两个事务共 8 条记录。解码完第二条事务时发现解码条数记录大于等于 upto_nchanges，会停止解码。

取值范围：非负整数。

说明

LSN 和 upto_nchanges 中任一参数达到限制，解码都会结束。

- decoding_plugin

解码插件，指定解码内容输出格式的 so 插件。

取值范围：提供 mppdb_decoding 和 sql_decoding 两个解码插件。

- xlog_path

解码插件，指定解码文件的 xlog 绝对路径，文件级别

取值范围：NULL 或者 xlog 文件绝对路径的字符串。

options：此项为可选参数，由一系列 options_name 和 options_value 一一对应组成，可以缺省，详见 pg_logical_slot_peek_changes。

示例：

```
postgres=# SELECT pg_current_xlog_location();
pg_current_xlog_location
--------------------------
 0/45D9370
(1 row)
postgres=# create table t1 (a int primary key,b int,c int);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t1_pkey" for table
"t1"
```

```
CREATE TABLE
postgres=# insert into t1 values(1,1,1);
INSERT 0 1
postgres=# insert into t1 values(2,2,2);
INSERT 0 1
postgres=# select data from pg_logical_get_area_changes('0/35D9370',
'0/45D9370',NULL,'mppdb_decoding',NULL);
data
------
(0 rows)
```

- pg_get_replication_slots()

  描述：获取复制槽列表。

  返回值类型：text，text，text，oid，boolean，xid，xid，text，boolean

  示例：

```
 postgres=# select * from pg_get_replication_slots();
slot_name | plugin | slot_type | datoid | active | xmin | catalog
_xmin | restart_lsn | dummy_standby
-----------+--------+-----------+--------+--------+------+--------
------+-------------+---------------
(0 rows)
```

- gs_get_parallel_decode_status()

  描述：监控各个解码线程的读取日志队列和解码结果队列的长度，以便定位并行解码性能瓶颈。

  返回值类型：text, int, text, text

  示例：

```
 postgres=# select * from gs_get_parallel_decode_status();
slot_name | parallel_decode_num | read_change_queue_length | deco
de_change_queue_length
-----------+---------------------+--------------------------+-----
----------------------
(0 rows)
```

  备注：返回值的 slot_name 代表复制槽名，parallel_decode_num 代表该复制槽的并行解码线程数，read_change_queue_length 列出了每个解码线程读取日志队列的当前长度，decode_change_queue_length 列出了每个解码线程解码结果队列的当前长度。

- pg_replication_origin_create (node_name)

  描述：用给定的外部名称创建一个复制源，并且返回分配给它的内部 ID。

  备注：调用该函数的用户需要具有 SYSADMIN 权限。

  参数说明：

  - node_name

  待创建的复制源的名称。

  取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

  返回值类型：oid

- pg_replication_origin_drop (node_name)

  描述：删除一个以前创建的复制源，包括任何相关的重放进度。

  备注：调用该函数的用户需要具有 SYSADMIN 权限。

  参数说明：

  - node_name

  待删除的复制源的名称。

  取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

- pg_replication_origin_oid (node_name)

  描述：根据名称查找复制源并返回内部 ID。如果没有发现这样的复制源，则抛出错误。

  备注：调用该函数的用户需要具有 SYSADMIN 权限。

  参数说明：

  - node_name

  要查找的复制源的名称

  取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

  返回值类型：oid

- pg_replication_origin_session_setup (node_name)

  描述：将当前会话标记为从给定的原点回放，从而允许跟踪回放进度。只能在当前没有

选择原点时使用。使用 pg_replication_origin_session_reset 命令来撤销。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

参数说明：

- node_name

复制源名称。

取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

- pg_replication_origin_session_reset ()

描述：取消 pg_replication_origin_session_setup()的效果。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

- pg_replication_origin_session_is_setup ()

描述：如果在当前会话中选择了复制源则返回真。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

返回值类型：boolean

- pg_replication_origin_session_progress (flush)

描述：返回当前会话中选择的复制源的重放位置。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

参数说明：

- flush

决定对应的本地事务是否被确保已经刷入磁盘。

取值范围：boolean

返回值类型：LSN

- pg_replication_origin_xact_setup (origin_lsn, origin_timestamp)

描述：将当前事务标记为重放在给定 LSN 和时间戳上提交的事务。只能在使用 pg_replication_origin_session_setup 选择复制源时调用。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

参数说明：

- origin_lsn

复制源回放位置。

取值范围：LSN

- origin_timestamp

事务提交时间。

取值范围：timestamp with time zone

- pg_replication_origin_xact_reset ()

描述：取消 pg_replication_origin_xact_setup()的效果。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

- pg_replication_origin_advance (node_name, lsn)

描述：

将给定节点的复制进度设置为给定的位置。这主要用于设置初始位置，或在配置更改或类似的变更后设置新位置。

注意：这个函数的使用不当可能会导致不一致的复制数据。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

参数说明：

- node_name

已有复制源名称。

取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

- lsn

复制源回放位置。

取值范围：LSN

- pg_replication_origin_progress (node_name, flush)

描述：返回给定复制源的重放位置。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

参数说明：

- node_name

复制源名称。

取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

- flush

决定对应的本地事务是否被确保已经刷入磁盘。

取值范围：boolean

- pg_show_replication_origin_status()

描述：获取复制源的复制状态。

备注：调用该函数的用户需要具有 SYSADMIN 权限。

返回值类型：

local_id：oid，复制源 id。

external_id：text，复制源名称。

remote_lsn：LSN，复制源的 lsn 位置。

local_lsn：LSN，本地的 lsn 位置。

- pg_get_publication_tables(pub_name)

描述：根据发布的名称，返回对应发布要发布的表的 relid 列表

参数说明：

- pub_name

已存在的发布名称

取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

返回值类型：relid 列表

- pg_stat_get_subscription(sub_oid oid) → record

描述：

输入订阅的 oid，返回订阅的状态信息。

参数说明：

- subid

订阅的 oid。

取值范围：oid

返回值类型：

relid：oid，表的 oid。

pid：thread_id，后台 apply/sync 线程的 thread id。

received_lsn：pg_lsn，从发布端接收到的最近的 lsn。

last_msg_send_time：timestamp，最近发布端发送消息的时间。

last_msg_receipt_time：timestamp，最新订阅端收到消息的时间。

latest_end_lsn：pg_lsn，最近一次收到保活消息时发布端的 lsn。

latest_end_time：timstamp，最近一次收到保活消息的时间。

5.26.9 段页式存储函数

- local_segment_space_info(tablespacename TEXT, databasename TEXT)

描述：输出为该表空间下所有 ExtentGroup 的使用信息。

返回值类型如下表所示：

表 5-27 返回值类型及描述

参数	描述
extent_size	该 ExtentGroup 的 extent 规格，单位是 block 数。
forknum	Fork 号。
total_blocks	物理文件总 extent 数目。
meta_data_blocks	表空间管理的 metadata 占用的 block 数，只包括 space

	header、map page 等，不包括 segment head。
used_data_blocks	存数据占用的 extent 数目。包括 segment head。
utilization	使用的 block 数占总 block 数的百分比。即 (used_data_blocks+meta_data_block)/total_blocks。
high_water_mark	高水位线，被分配出去的 extent，最大的物理页号。超过高水位线的 block 都没有被使用，可以被直接回收。

例如：

```
postgres=# select * from local_segment_space_info('pg_default', 'postgres');
INFO:  Segment is not initialized in current database
 node_name | extent_size | forknum | total_blocks | meta_data_bloc
ks | used_data_blocks | utilization | high_water_mark
-----------+-------------+---------+--------------+---------------
---+------------------+-------------+-----------------
(0 rows)
```

- pg_stat_segment_extent_usage(int4 tablespace oid, int4 database oid, int4 extent_type, int4 forknum)

描述：每次返回一个 ExtentGroup 中，每个被分配出去的 extent 的使用情况。extent_type 表示 ExtentGroup 的类型，合理取值为[1,5]的 int 值。在此范围外的会报 error。forknum 表示 fork 号，合法取值为[0,4]的 int 值，目前只有三种值有效，数据文件为 0，FSM 文件为 1，visibility map 文件为 2。

返回值类型如下表所示：

表 5-28 返回值类型及描述

名称	描述
start_block	Extent 的起始物理页号。
extent_size	Extent 的大小。
usage_type	Extent 的使用类型，比如 segment head、data extent 等。

ower_location	有指针指向该 extent 的对象的位置。比如 data extent 的 owner 就是它所属的 segment 的 head 位置。
special_data	extent 在其 owner 中的位置。该字段的数据跟使用类型有关。比如 data extent 的 special data，就是其所属 segment 中的 extent id。

其中，usage_type 为枚举类型，每一项的含义为：

- Non-bucket table segment head ：非 hashbucket 表的数据段头。
- Non-bucket table fork head：非段页式表的 fork 段头。
- Data extent：数据块。

例如：

```
postgres=# select * from pg_stat_segment_extent_usage((select oid::int4 from
pg_tablespace where spcname='pg_default'), (select oid::int4 from pg_database
where datname='postgres'), 1, 0);
INFO:  Segment is not initialized in current database
 start_block | extent_size | usage_type | ower_location | special_
data
-------------+-------------+------------+---------------+---------
-----
(0 rows)
```

- local_space_shrink(tablespacename TEXT, databasename TEXT)

描述：当前节点上对指定段页式空间做物理空间收缩。注意，目前只支持对当前连接的 database 做 shrink。

返回值：空

- gs_space_shrink(int4 tablespace, int4 database, int4 extent_type, int4 forknum)

描述：效果跟 local_space_shrink 类似，对指定段页式空间做物理空间收缩,但参数不同，传入的是 tablespace 和 database 的 oid，extent_type 为[2,5]的 int 值。注意：extent_type = 1 表示段页式元数据，目前不支持对元数据所在的物理文件做收缩。该函数仅限工具使用，不建议用户直接使用。

返回值：空

- pg_stat_remain_segment_info()

描述：展示在当前节点上，因为故障等原因而残留的 extent。残留 extent 主要分为两类：分配而未被利用的 segment 和分配出去而未被利用的 extent。两者主要区别在于 segment 会包含多个 extent，回收时，要将 segment 上的 extent 一并全部回收。

返回值类型如下表所示：

表 5-29 返回值类型及描述

名称	描述
space_id	表空间 ID
db_id	数据库 ID
block_id	Extent 的 ID
type	Extent 的类型，当前有三种：ALLOC_SEGMENT、DROP_SEGMENT、SHRINK_EXTENT

其中 type 的三种类型分别表示：

- ALLOC_SEGMENT:用户创建一张段页式表，当 segment 刚被分配，但是建表语句所在事务仍未提交时，节点故障，导致该 segment 被分配后，没有被使用。
- DROP_SEGMENT:用户删除段页式表，当该事务成功提交，但是此表的 segment 页面对应的 bit 位未被重置，就发生掉电等故障，造成该 segment 未被使用，也未被释放。
- SHRINK_EXTENT:用户对段页式表执行 shrink 操作，在未对空置出的 extent 进行释放时，发生掉电等故障，造成该 extent 残留，无法被重新利用。

例如：

```
postgres=# select * from pg_stat_remain_segment_info();
 space_id | db_id | block_id | type
----------+-------+----------+------
(0 rows)
```

- pg_free_remain_segment(int4 spaceId, int4 dbId, int4 segmentId)

描述：释放指定的残留 extent。参数取值必须为从函数 pg_stat_remain_segment_info 中查询获取。函数会对传入值校验，如果指定 extent 不在记录的残留 extent 中，将返回错误信息。指定的 extent 如果为单个 extent，则只将其独自释放；如果为一个 segment，则会将此 segment 以及此 segment 上记录的所有 extent 释放。

返回值：空

5.26.10其它函数

- plan_seed()

  描述：获取前一次查询语句的 seed 值（内部使用）。

  返回值类型：int

- pg_stat_get_env()

  描述：获取当前节点的环境变量信息，仅 sysadmin 和 monitor admin 可以访问。

  返回值类型：record

  示例：

```
postgres=# select pg_stat_get_env();
                                                                pg_stat_get_env
------------------------------------------------------------------------------------------------------------------------

(dn1,"localhost,[local_ip]",7961,20008,/home/gbase/gbase8s/app,/home/gbase/project/install/data/dn/dn1_1,/home/gbase/gbase8s/log/pg_log/dn1_1)
(1 row)
```

- pg_catalog.plancache_clean()

  描述：清理节点上无人使用的全局计划缓存。

  返回值类型：bool

- pg_catalog.plancache_status()

  描述：显示节点上全局计划缓存的信息，函数返回信息和 GLOBAL_PLANCACHE_STATUS 一致。

  返回值类型：record

- textlen(text)

  描述：提供查询 text 的逻辑长度的方法。

  返回值类型：int

- threadpool_status()

  描述：显示线程池中工作线程及会话的状态信息。

  返回值类型：record

- get_local_active_session()

  描述：提供当前节点保存在内存中的历史活跃 session 状态的采样记录。

  返回值类型：record

- pg_stat_get_thread()

  描述：提供当前节点下所有线程的状态信息，sysadmin 和 monitor admin 用户可以查看所有线程信息，普通用户查看本用户的线程信息。

  返回值类型：record

- pg_stat_get_sql_count()

  描述：提供当前节点中用户执行的 SELECT/UPDATE/INSERT/DELETE/MERGE INTO 语句的计数结果，sysadmin 和 monitor admin 用户可以查看所有用户的信息，普通用户查看本用户的统计信息。

  返回值类型：record

- pg_stat_get_data_senders()

  描述：提供当前活跃的数据复制发送线程的详细信息。

  返回值类型：record

- get_wait_event_info()

  描述：提供 wait event 事件的具体信息。

  返回值类型：record

- generate_wdr_report(begin_snap_id bigint, end_snap_id bigint, report_type cstring,

report_scope cstring, node_name cstring)

描述：基于两个 snapshot 生成系统诊断报告。需要在 postgres 库下执行，默认初始化用户或 monadmin 用户可以访问，V8.8.500R001C20SPC002 及其之前的版本初始化用户或 sysadmin 用户可以访问。只可在系统库中查询到结果，用户库中无法查询。

返回值类型：record

表 5-30 参数类型及描述

参数	说明	取值范围
begin_snap_id	生成某段时间内性能诊断报告的开始 snapshotid。	——
end_snap_id	结束 snapshot 的 id，默认 end_snap_id 大于 begin_snap_id。	——
report_type	指定生成 report 的类型。	summary detail all，即同时包含 summary 和 detail。
report_scope	指定生成 report 的范围。	cluster：数据库级别的信息 node：节点级别的信息
node_name	在 report_scope 指定为 node 时，需要把该参数指定为对应节点的名称。（节点名称可以执行 select * from pg_node_env;查询）。 在 report_scope 为 cluster 时，该值可以省略或者指定为空或 NULL。	cluster：省略/空/NULL node：数据库节点名称

- create_wdr_snapshot()

  描述：手工生成系统诊断快照，该函数需要 sysadmin 权限。

  返回值类型：text

- kill_snapshot()

描述：kill 后台的 WDR snapshot 线程，调用该函数的用户需要具有 SYSADMIN 权限或具有 REPLICATION 权限或继承了内置角色 gs_role_replication 的权限。

返回值类型：void

- capture_view_to_json(text,integer)

描述：将视图的结果存入 GUC：perf_directory 所指定的目录，如果 is_crossdb 为 1，则表示对于所有的 database 都会访问一次 view；如果 is_crossdb 为 0，则表示仅对当前 database 进行一次视图访问。该函数只有 sysadmin 和 monitor admin 用户可以执行。

返回值类型：int

- reset_unique_sql

描述：用来清理数据库节点内存中的 Unique SQL（需要 sysadmin 权限）。

返回值类型：bool

表 5-31 参数类型及描述

参数	类型	描述
scope	text	清理范围类型： GLOBAL：清理所有的节点，如果是'GLOBAL'，则只可以为主节点执行此函数。 ● LOCAL：清理本节点。
clean_type	text	BY_USERID：按用户 ID 来进行清理 Unique SQL。 BY_CNID：按主节点的 ID 来进行清理 Unique SQL。 ● ALL：全部清理。
clean_value	int8	具体清理 type 对应的清理值。

须知

- scope 的取值划分为 GLOBAL 和 LOCAL 两种类型，在单机场景中二者意义相同，均表示清理本节点。
- clean_type 参数的 BY_CNID 值对于单机场景无效。

- wdr_xdb_query(db_name_str text, query text)

  描述：提供本地跨数据库执行 query 的能力。例如：在连接到 postgres 库时，访问 test 库下的表。

```
select col1 from wdr_xdb_query('dbname=test','select col1 from t1') as dd(col1 int);
```

  返回值类型：record

- pg_wlm_jump_queue(pid int)

  描述：调整任务到数据库主节点队列的最前端。

  返回值类型：boolean

  返回值：

  - true：成功。
  - false：失败。

- gs_wlm_switch_cgroup(pid int, cgroup text)

  描述：调整作业的优先级到新控制组。

  返回值类型：boolean

  返回值：

  - true：成功。
  - false：失败。

- pv_session_memctx_detail(threadid tid, MemoryContextName text)

  描述：将线程 tid 的 MemoryContextName 内存上下文信息记录到“$GAUSSLOG/pg_log/${node_name}/dumpmem”目录下的“threadid_timestamp.log”文件中。其中 threadid 可通过视图 GS_SESSION_MEMORY_DETAIL 中的 sessid 后获得。在正式发布的版本中仅接受 MemoryContextName 为空串（两个单引号表示输入为空串，即”）的输入，此时会记录所有的内存上下文信息，否则不会有任何操作。对供内部开发人员和测试人员调试用的 DEBUG 版本，可以指定需要统计的 MemoryContextName，此时会将该 Context 所有的内存使用情况记录到指定文件。该函数需要管理员权限的用户才能执行。

  返回值类型：boolean

返回值：

- true：成功。
- false：失败。

- pg_shared_memctx_detail(MemoryContextName text)

描述：将 MemoryContextName 内存上下文信息记录到“$GAUSSLOG/pg_log/${node_name}/dumpmem”目录下的“threadid_timestamp.log”文件中。该函数功能仅在 DEBUG 版本中供内部开发人员和测试人员调试使用，在正式发布版本中调用该函数不会有任何操作。该函数需要管理员权限的用户才能执行。

返回值类型：boolean

返回值：

- true：成功。
- false：失败。

- local_bgwriter_stat()

描述：显示本实例的 bgwriter 线程刷页信息，候选 buffer 链中页面个数，buffer 淘汰信息。

返回值类型：record

- local_candidate_stat()

描述：显示本实例的候选 buffer 链中页面个数，buffer 淘汰信息，包含 normal buffer pool 和 segment buffer pool。

返回值类型：record

- local_ckpt_stat()

描述：显示本实例的检查点信息和各类日志刷页情况。

返回值类型：record

- local_double_write_stat()

描述：显示本实例的双写文件的情况。

返回值类型：record

表 5-32 参数类型及描述

参数	类型	描述
node_name	text	实例名称。
curr_dwn	int8	当前双写文件的序列号。
curr_start_page	int8	当前双写文件恢复起始页面。
file_trunc_num	int8	当前双写文件复用的次数。
file_reset_num	int8	当前双写文件写满后发生重置的次数。
total_writes	int8	当前双写文件总的 I/O 次数。
low_threshold_writes	int8	低效率写双写文件的 I/O 次数(一次 I/O 刷页数量少于 16 页面)。
high_threshold_writes	int8	高效率写双写文件的 I/O 次数(一次 I/O 刷页数量多于一批，421 个页面)。
total_pages	int8	当前刷页到双写文件区的总的页面个数。
low_threshold_pages	int8	低效率刷页的页面个数。
high_threshold_pages	int8	高效率刷页的页面个数。
file_id	int8	当前双写文件的 id 号。

- local_single_flush_dw_stat()

  描述：显示本实例的单页面淘汰双写文件的情况。

  返回值类型：record

- local_pagewriter_stat()

描述：显示本实例的刷页信息和检查点信息。

返回值类型：record

- local_redo_stat()

描述：显示本实例的备机的当前回放状态。

返回值类型：record

备注：返回的回放状态主要包括当前回放位置、回放最小恢复点位置等信息。

- local_recovery_status()

描述：显示本实例的主机和备机的日志流控信息。

返回值类型：record

- gs_wlm_node_recover(boolean isForce)

描述：获取当前内存中记录的 TopSQL 查询语句级别相关统计信息，当传入的参数不为 0 时，会将这部分信息从内存中清理掉。

返回值类型：record

- gs_wlm_node_clean(cstring nodename)

描述：动态负载管理节点故障后做数据清理操作。该函数只有管理员用户可以执行，属于数据库实例管理模块调用的，不建议用户直接调用。该视图在集中式和单机环境上不支持。

返回值类型：bool

- gs_cgroup_map_ng_conf(group name)

描述：读取指定逻辑数据库的 cgroup 配置文件。

返回值类型：record

- gs_wlm_switch_cgroup(sess_id int8, cgroup name)

描述：切换指定会话的控制组。

返回值类型：record

- hdfs_fdw_handler()

描述：用于外表重写功能，定义外表时需要定义的函数。

返回值类型：record

- hdfs_fdw_validator(text[], oid)

  描述：用于外表重写功能，定义外表时需要定义的函数。

  返回值类型：record

- comm_client_info()

  描述：用于查询单个节点活跃的客户端连接信息。

  返回值类型：setof record

- pg_sync_cstore_delta(text)

  描述：同步指定列存表的 delta 表表结构，使其与列存表主表一致。

  返回值类型：bigint

- pg_sync_cstore_delta()

  描述：同步所有列存表的 delta 表表结构，使其与列存表主表一致。

  返回值类型：bigint

- pg_get_flush_lsn()

  描述：返回当前节点 flush 的 xlog 位置。

  返回值类型：text

- pg_get_sync_flush_lsn()

  描述：返回当前节点多数派 flush 的 xlog 位置。

  返回值类型：text

- gs_create_log_tables()

  描述：用于创建运行日志和性能日志的外表和视图。

  返回值类型：void

  示例：

```
postgres=# select gs_create_log_tables();
gs_create_log_tables
----------------------
```

```
(1 row)
```

- dbe_perf.get_global_full_sql_by_timestamp(start_timestamp timestamp with time zone, end_timestamp timestamp with time zone)

  描述：获取数据库级的全量 SQL(Full SQL)信息。只可在系统库中查询到结果，用户库中无法查询。

  返回值类型：record

表 5-33 参数类型及描述

参数	类型	描述
start_timestamp	timestamp with time zone	SQL 启动时间范围的开始时间点。
end_timestamp	timestamp with time zone	SQL 启动时间范围的结束时间点。

- dbe_perf.get_global_slow_sql_by_timestamp(start_timestamp timestamp with time zone, end_timestamp timestamp with time zone)

  描述：获取数据库级的慢 SQL 信息。只可在系统库中查询到结果，用户库中无法查询。

  返回值类型：record

表 5-34 参数类型及描述

参数	类型	描述
start_timestamp	timestamp with time zone	SQL 启动时间范围的开始时间点。
end_timestamp	timestamp with time zone	SQL 启动时间范围的结束时间点。

- statement_detail_decode(detail text, format text, pretty boolean)

  描述：解析全量/慢 SQL 语句中的 details 字段的信息。只可在系统库中查询到结果，用

户库中无法查询。

返回值类型：text

表 5-35 参数类型及描述

参数	类型	描述
detail	text	SQL 语句产生的事件的集合（不可读）。
format	text	解析输出格式，取值为 plaintext。
pretty	boolean	当 format 为 plaintext 时，是否为标准格式： ● true 表示通过“\n”分隔事件。 ● false 表示通过“，”分隔事件。

- get_prepared_pending_xid

  描述：当恢复完成时，返回 nextxid。

  参数：nan

  返回值类型：text

- pg_clean_region_info

  描述：清理 regionmap。

  参数：nan

  返回值类型：character varying

- pg_get_delta_info

  描述：从单个 dn 获取 delta info。

  参数：rel text、 schema_name text

  返回值类型：part_name text、 live_tuple bigint、 data_size bigint、 blocknum bigint

- pg_get_replication_slot_name

  描述：获取 slot name。

参数：nan

返回值类型：text

- pg_get_running_xacts

描述：获取运行中的 xact。

参数：nan

返回值类型：handle integer、 gxid xid、 state tinyint、 node text、 xmin xid、 vacuum boolean、 timeline bigint、 prepare_xid xid、 pid bigint、 next_xid xid

- pg_get_variable_info

描述：获取共享内存变量 cache。

参数：nan

返回值类型：node_name text、 nextOid oid、 nextXid xid、 oldestXid xid、 xidVacLimit xid、 oldestXidDB oid、 lastExtendCSNLogpage xid、 startExtendCSNLogpage xid、 nextCommitSeqNo xid、 latestCompletedXid xid、 startupMaxXid xid

- pg_get_xidlimit

描述：从共享内存获取事物 id 信息。

参数：nan

返回值类型：nextXid xid、 oldestXid xid、 xidVacLimit xid、 xidWarnLimit xid、 xidStopLimit xid、 xidWrapLimit xid、 oldestXidDB oid

- get_global_user_transaction()

描述：返回所有节点上各用户的事务相关信息。

返回值类型：node_name name、usename name、commit_counter bigint、rollback_counter bigint、 resp_min bigint、 resp_max bigint、 resp_avg bigint、 resp_total bigint、 bg_commit_counter bigint、 bg_rollback_counter bigint、 bg_resp_min bigint、 bg_resp_max bigint、 bg_resp_avg bigint、 bg_resp_total bigint

- pg_collation_for

描述：返回入参字符串对应的排序规则。

参数：any（如果是常量必须进行显式类型转换）

返回值类型：text

- pgxc_unlock_for_sp_database(name Name)

  描述：释放指定数据库锁。

  参数：数据库名

  返回值类型：布尔

- pgxc_lock_for_sp_database(name Name)

  描述：对指定的数据库加锁。

  参数：数据库名

  返回值类型：布尔

- copy_error_log_create()

  描述：创建 COPY FROM 容错机制所需要的错误表（public.pgxc_copy_error_log）。

  返回值类型：Boolean

  说明

  此函数会尝试创建 public.pgxc_copy_error_log 表，表的详细信息请参见表 6。

  在 relname 列上创建 B-tree 索引，并 REVOKE ALL on public.pgxc_copy_error_log FROM public 对错误表进行权限控制（与 COPY 语句权限一致）。

  由于尝试创建的 public.pgxc_copy_error_log 定义是一张行存表，因此数据库实例上必须支持行存表的创建才能够正常运行此函数，并使用后续的 COPY 容错功能。需要特别注意的是，enable_hadoop_env 这个 GUC 参数开启后会禁止在数据库实例内创建行存表（GaussDB Kernel 默认为 off）。

  此函数权限与错误表、COPY 权限一致，需为 Sysadmin 及以上。

  若创建前，public.pgxc_copy_error_log 表已存在或者 copy_error_log_relname_idx 索引已存在，则此函数会报错回滚。

表 5-36 参数类型及描述

列名称	类型	描述

relname	character varying	表名称。以模式名.表名形式显示。
begintime	timestamp with time zone	出现数据格式错误的时间。
filename	character varying	出现数据格式错误的数据源文件名。
lineno	bigint	在数据源文件中，出现数据格式错误的行号。
rawrecord	text	在数据源文件中，出现数据格式错误的原始记录。
detail	text	详细错误信息。

- dynamic_func_control(scope text, function_name text, action text, “{params}” text[])

  描述：动态开启内置的功能，当前仅支持动态开启全量 SQL。

  返回值类型：record

表 5-37 参数类型及描述

参数	类型	描述
scope	text	动态开启功能的范围，当前仅支持 LOCAL。
function_name	text	功能的名称，当前仅支持 STMT。
action	text	当 function_name 为'STMT'时，action 仅支持 TRACK/UNTRACK/LIST/CLEAN： TRACK：开始记录归一化 SQL 的全量 SQL 信息。 UNTRACK：取消记录归一化 SQL 的全量 SQL 信息。 LIST：列取当前 TRACK 的归一化 SQL 的

		信息。 CLEAN：清理记录当前归一化 SQL 的信息。
params	text[]	当 function_name 为'STMT'时，对应不同的 action 时，对应的 params 设置如下： TRACK：'{“归一化 SQLID”, “L0/L1/L2”}' ● UNTRACK：'{“归一化 SQLID”}' ● LIST：'{}' ● CLEAN：'{}'

- gs_parse_page_bypath(path text, blocknum bigint, relation_type text, read_memory boolean)

  描述：用于解析指定表页面，并返回存放解析内容的路径。

  返回值类型：text

  备注：必须是系统管理员或运维管理员才能执行此函数。

表 5-38 参数类型及描述

参数	类型	描述
path	text	对于普通表或段页式表，相对路径为：tablespace name/database oid/表的 relfilenode(物理文件名)。例如：base/16603/16394。 表文件的相对路径可以通过 pg_relation_filepath(table_name text)查找。 合法的 path 格式列举： ● global/relNode ● base/dbNode/relNode ● pg_tblspc/spcNode/version_dir/dbNode/relNode
blocknum	bigint	● -1 所有 block 的信息 ● 0- MaxBlockNumber 对应 block 的信息
relation_type	text	● heap(astore 表)

		● uheap(ustore 表) ● btree_index(BTree 索引) ● ubtree_index(UBTree 索引) ● segment(段页式)
read_memory	boolean	● false，从磁盘文件解析。 true，首先尝试从共享缓冲区中解析该页面；如果共享缓冲区中不存在，则从磁盘文件解析。

- gs_xlogdump_lsn(start_lsn text, end_lsn text)

描述：用于解析指定 lsn 范围之内的 XLOG 日志，并返回存放解析内容的路径。可以通过 pg_current_xlog_location()获取当前 XLOG 位置。

返回值类型：text

参数：LSN 起始位置，LSN 结束位置

备注：必须是系统管理员或运维管理员才能执行此函数。

- gs_xlogdump_xid(c_xid xid)

描述：用于解析指定 xid 的 XLOG 日志，并返回存放解析内容的路径。可以通过 txid_current()获取当前事务 ID。

参数：事务 ID

返回值类型：text

备注：必须是系统管理员或运维管理员才能执行此函数。

- gs_xlogdump_tablepath(path text, blocknum bigint, relation_type text)

描述：用于解析指定表页面对应的日志，并返回存放解析内容的路径。

返回值类型：text

备注：必须是系统管理员或运维管理员才能执行此函数。

表 5-39 参数类型及描述

参数	类型	描述

path	text	对于普通表或段页式表，相对路径为：tablespace name/database oid/表的 relfilenode(物理文件名)。例如：base/16603/16394。 表文件的相对路径可以通过 pg_relation_filepath(table_name text)查找。 合法的 path 格式列举： ● global/relNode ● base/dbNode/relNode ● pg_tblspc/spcNode/version_dir/dbNode/relNode
blocknum	bigint	● -1 所有 block 的信息 ● 0- MaxBlockNumber 对应 block 的信息
relation_type	text	● heap(astore 表) ● uheap(ustore 表) ● btree_index(BTree 索引) ● ubtree_index(UBTree 索引) ● segment(段页式)

- gs_xlogdump_parsepage_tablepath(path text, blocknum bigint, relation_type text, read_memory boolean)

描述：用于解析指定表页面和表页面对应的日志，并返回存放解析内容的路径。可以看做一次执行 gs_parse_page_bypath 和 gs_xlogdump_tablepath。该函数执行的前置条件是表文件存在。如果想查看已删除的表的相关日志，请直接调用 gs_xlogdump_tablepath。

返回值类型：text

备注：必须是系统管理员或运维管理员才能执行此函数。

表 5-40 参数类型及描述

参数	类型	描述
path	text	对于普通表或段页式表，相对路径为：tablespace name/database oid/表的 relfilenode(物理文件名)；例

		如：base/16603/16394 表文件的相对路径可以通过pg_relation_filepath(table_name text)查找。 合法的 path 格式列举： ● global/relNode ● base/dbNode/relNode ● pg_tblspc/spcNode/version_dir/dbNode/relNode
blocknum	bigint	● -1 所有 block 的信息 ● 0- MaxBlockNumber 对应 block 的信息
relation_type	text	● heap(astore 表) ● uheap(ustore 表) ● btree_index(BTree 索引) ● ubtree_index(UBTree 索引) ● segment(段页式)

- gs_index_verify(Oid oid, uint32:wq blkno)

  描述：用于校验 UBtree 索引页面或者索引树上 key 的顺序是否正确。

  返回值类型：record

表 5-41 参数类型及描述

参数	类型	描述
oid	Oid	索引文件 relfilenode,可以通过 select relfilenode from pg_class where relname='name'查询，其中 name 表示对应的索引文件名字。
blkno	uint32	● 0 ：表示检验整个索引树上所有页面。 ● >0：表示校验页面编码等于 blkno 的索引页面。

- gs_index_recycle_queue(Oid oid, int type, uint32 blkno)

  描述：用于解析 UBtree 索引回收队列信息。

返回值类型：record

表 5-42 参数类型及描述

参数	类型	描述
oid	oid	索引文件 relfilenode，可以通过 select relfilenode from pg_class where relname='name'查询，其中 name 表示对应的索引文件名字。
type	int	● 0，表示解析整个待回收队列。 ● 1，表示解析整个空页队列。 ● 2，表示解析单个页面。
blkno	uint32	回收队列页面编号，该参数只有在 type=2 的时候有效，blkno 有效取值范围为 1~4294967294。

- gs_stat_wal_entrytable(int64 idx)

  描述：用于输出 xlog 中预写日志插入状态表的内容。

  返回值类型：recordgs_walwriter_flush_position()

  描述：输出预写日志的刷新位置。

  返回值类型：record

表 5-43 参数类型及描述

参数类型	参数名	类型	描述
输入参数	idx	int64	● -1：查询数组所有元素。 ● 0-最大值：具体某个数组元素内容。
输出参数	idx	uint64	记录对应数组中的下标。
输出参数	endlsn	uint64	记录的 LSN 标签。
输出参数	lrc	int32	记录对应的 LRC。

输出参数	status	uint32	标识当前 entry 对应的 xlog 是否已经完全拷贝到 wal buffer 中： ● 0：非 COPIED ● 1：COPIED

- gs_walwriter_flush_position()

  描述：输出预写日志的刷新位置。

  返回值类型：record

表 5-44 参数类型及描述

参数类型	参数名	类型	描述
输出参数	last_flush_status_entry	int32	Xlog flush 上一个刷盘的 tblEntry 下标索引。
输出参数	last_scanned_lrc	int32	Xlog flush 上一次扫描到的最后一个 tblEntry 记录的 LRC。
输出参数	curr_lrc	int32	WALInsertStatusEntry 状态表中 LRC 最新的使用情况，该 LRC 表示下一个 Xlog 记录写入时在 WALInsertStatusEntry 对应的 LRC 值。
输出参数	curr_byte_pos	uint64	Xlog 记录写入 WAL 文件，最新分配的位置，下一个 xlog 记录插入点。
输出参数	prev_byte_size	uint32	上一个 xlog 记录的长度。
输出参数	flush_result	uint64	当前全局 xlog 刷盘的位置。
输出参数	send_result	uint64	当前主机上 xlog 发送位置。
输出参数	shm_rqst_write_pos	uint64	共享内存中记录的 XLogCtl 中

			LogwrtRqst 请求的 write 位置。
输出参数	shm_rqst_flush_pos	uint64	共享内存中记录的 XLogCtl 中 LogwrtRqst 请求的 flush 位置。
输出参数	shm_result_write_pos	uint64	共享内存中记录的 XLogCtl 中 LogwrtResult 的 write 位置。
输出参数	shm_result_flush_pos	uint64	共享内存中记录的 XLogCtl 中 LogwrtResult 的 flush 位置。
输出参数	curr_time	text	当前时间。

- gs_walwriter_flush_stat(int operation)

  描述：用于统计预写日志 write 与 sync 的次数频率与数据量，以及 xlog 文件的信息。

  返回值类型：recordgs_comm_proxy_thread_status()

  描述：用于在数据库实例配置用户态网络的场景下，代理通信库 comm_proxy 收发数据包统计。

  参数：nan

  返回值类型：record

  说明

  此函数的查询仅在集中式环境开始部署用户态网络，且 comm_proxy_attr 参数中 enable_dfx 配置为 true 的条件下显示具体信息。其他场景报错不支持查询。

表 5-45 参数类型及描述

参数类型	参数名	类型	描述
输入参数	operation	int	● -1：关闭统计开关(默认状态为关闭)。 ● 0：打开统计开关。 ● 1：查询统计信息。 ● 2：重置统计信息。

输出参数	write_times	uint64	Xlog 调用 write 接口的次数。
输出参数	sync_times	uint64	Xlog 调用 sync 接口次数。
输出参数	total_xlog_sync_bytes	uint64	Backend 线程请求写入 xlog 总量统计值。
输出参数	total_actual_xlog_sync_bytes	uint64	调用 sync 接口实际刷盘的 xlog 总量统计值。
输出参数	avg_write_bytes	uint32	每次调用 XLogWrite 接口请求写的 xlog 量。
输出参数	avg_actual_write_bytes	uint32	实际每次调用 write 接口写的 xlog 量。
输出参数	avg_sync_bytes	uint32	平均每次请求 sync 的 xlog 量。
输出参数	avg_actual_sync_bytes	uint32	实际每次调用 sync 刷盘 xlog 量。
输出参数	total_write_time	uint64	调用 write 操作总时间统计(单位：us)。
输出参数	total_sync_time	uint64	调用 sync 操作总时间统计(单位：us)。
输出参数	avg_write_time	uint32	每次调用 write 接口平均时间(单位：us)。
输出参数	avg_sync_time	uint32	每次调用 sync 接口平均时间(单位：us)。
输出参	curr_init_xlog	uint64	当前最新创建的 xlog 段文件编号。

数	_segno		
输出参数	curr_open_xlog_segno	uint64	当前正在写的 xlog 段文件编号。
输出参数	last_reset_time	text	上一次重置统计信息的时间。
输出参数	curr_time	text	当前时间。

- gs_comm_proxy_thread_status()

描述：用于在数据库实例配置用户态网络的场景下，代理通信库 comm_proxy 收发数据包统计。

参数：nan

返回值类型：record

说明

仅在集中式环境开始部署用户态网络，且 comm_proxy_attr 参数中 enable_dfx 配置为 true 的条件下，此函数的查询才显示具体信息。其他场景报错不支持查询。

5.26.11 Undo 系统函数

- gs_undo_meta(type, zoneId, location)

描述：Undo 各模块元信息。

参数说明：

- type(元信息类型)
  - 0 表示 Undo Zone(Record) 对应的元信息。
  - 1 表示 Undo Zone(Transaction Slot) 对应的元信息。
  - 2 表示 Undo Space(Record) 对应的元信息。
  - 3 表示 Undo Space(Transaction Slot) 对应的元信息。

- zoneId(undo zone 编号)
  - -1 表示所有 undo zone 的元信息。
  - 0-1024*1024 表示对应 zoneid 的元信息。
- location(读取位置)
  - 0 表示从当前内存中读取。
  - 1 表示从物理文件中读取。

返回值类型：record

- gs_undo_translot(location, zoneId)

  描述：Undo 事务槽信息。

  参数说明:

  - location(读取位置)
    - 0 表示从当前内存中读取。
    - 1 表示从物理文件中读取。
  - zoneId(undo zone 编号)
    - -1 表示所有 undo zone 的元信息。
    - 0-1024*1024 表示对应 zoneId 的元信息。

  返回值类型：record

- gs_stat_undo()

  描述：Undo 统计信息。

  返回值类型：record

表 5-46 参数类型及描述

参数类型	参数名	类型	描述

输出参数	curr_used_zone_count	uint32	当前使用的 Undo zone 数量。
输出参数	top_used_zones	text	前三个使用量最大的 Undo zone 信息，格式输出为： (zoneId1:使用大小，zoneId2:使用大小，zoneId3:使用大小)。
输出参数	curr_used_undo_size	uint32	当前使用的 Undo 总空间大小，单位为 MB。
输出参数	undo_threshold	uint32	为 guc 参数 undo_space_limit_size * 80%计算的结果,单位为 MB。
输出参数	oldest_xid_in_undo	uint64	当前 Undo 空间回收到的事务 xid(小于该 xid 事务产生的 Undo 记录都已经被回收)。
输出参数	oldest_xmin	uint64	最老的活跃事务。
输出参数	total_undo_chain_len	int64	所有访问过的 Undo 链总长度。
输出参数	max_undo_chain_len	int64	最大访问过的 Undo 链长度。
输出参数	create_undo_file_count	uint32	创建的 Undo 文件数量统计。
输出参数	discard_undo_file_count	uint32	删除的 Undo 文件数量统计。

- gs_undo_record(undoptr)

  描述：Undo 记录解析。

  参数说明：

- undoptr(undo 记录指针)

  返回值类型：record

5.26.12行存压缩系统函数

compress_buffer_stat_info()

描述：查看 pca buffer 统计信息。

返回值类型：record

表 5-47 compress_buffer_stat_info 参数说明

参数类型	参数名	类型	描述
输出参数	ctrl_cnt	bigint	pca_ page_ctrl_t 结构体。
输出参数	main_cnt	bigint	各个分区所有的 main lru 链的总数。
输出参数	free_cnt	bigint	各分区上 free lru 链的总数。
输出参数	recycle_times	bigint	buffer 进行 lru 的淘汰次数。

- compress_ ratio_info(file_ path text)

  描述：查看文件压缩率信息。

  返回值类型：record

表 5-48 compress_ ratio_info 参数说明

参数类型	参数名	类型	描述
输入参数	file_ path	text	相对文件路径。

输出参数	path	text	文件相对路径。
输出参数	is_compress	boolean	是否为压缩文件。
输出参数	file_count	bigint	包含段文件个数。
输出参数	logic_size	bigint	逻辑大小，单位是 byte。
输出参数	physic_size	bigint	实际物理大小，单位 byte。
输出参数	compress_ ratio	text	文件的压缩率。

- compress_statistic_info(file_ path text ， step smallint)

  描述：统计文件压缩后的离散度信息。

  返回值类型：record

表 5-49 compress_statistic_info 参数说明

参数类型	参数名	类型	描述
输入参数	file_ path	text	文件相对路径。
输入参数	step	smallint	采样统计步长。
输出参数	path	text	文件相对路径。
输出参数	extent_count	bigint	extent 的数量。
输出参数	dispersion_count	bigint	含有离散 chunk 的页面个数。
输出参数	void_count	bigint	还有 chunk 空洞的页面个数。

- compress_address_header(oid regclass, seg_id bigint)

  描述：查看文件压缩页面的管理信息。

  返回值类型：record

表 5-50 compress_address_header 参数说明

参数类型	参数名	类型	描述
输入参数	oid	regclass	文件所属表的 reloid。
输入参数	seg_id	bigint	segment 文件的序号。
输出参数	extent	bigint	extent 的编号。
输出参数	nblocks	bigint	extent 里的 page 个数。
输出参数	alocated_chunks	integer	ext 中分配了配多少个 chunk。
输出参数	chunk_size	integer	chunksize 大小，单位是 byte。
输出参数	algorithm	bigint	使用的压缩算法。

- compress_address_details(oid regclass, seg_id bigint)

  描述：页面 chunk 使用的详细信息。

  返回值类型：record

表 5-51 compress_address_details 参数说明

参数类型	参数名	类型	描述
输入参数	oid	regclass	文件所属表的 reloid。
输入参数	seg_id	bigint	segment 文件的序号。
输出参数	extent	bigint	extent 的编号。
输出参数	extent_block_number	bigint	该 extent 内的页面编号，0~127。
输出参数	block_ number	bigint	整体页面编号。
输出参数	alocated_chunks	integer	该页面用了多少个 chunk。
输出参数	nchunks	integer	该页面实际用了多少个 chunk，

			不大于 alocated_chunks。
输出参数	chunknos	integer	用的 chunks 的编号，从 1 开始。

5.27 统计信息函数

统计信息函数根据访问对象分为两种类型：针对某个数据库进行访问的函数，以数据库中每个表或索引的 OID 作为参数，标识需要报告的数据库；针对某个服务器进行访问的函数，以一个服务器进程号为参数，其范围从 1 到当前活跃服务器的数目。

- pg_stat_get_db_conflict_tablespace(oid)

  描述：由于恢复与数据库中删除的表空间发生冲突而取消的查询数。

  返回值类型：bigint

- pg_control_group_config

  描述：在当前节点上打印 cgroup 配置。

  返回值类型：record

- pg_stat_get_db_stat_reset_time(oid)

  描述：上次重置数据库统计信息的时间。首次连接到每个数据库期间初始化为系统时间。当在数据库上调用 pg_stat_reset，以及对任何表或索引执行 pg_stat_reset_single_table_counters 时，重置时间都会更新。

  返回值类型：timestamptz

- pg_stat_get_function_total_time(oid)

  描述：该函数花费的总挂钟时间，以微秒为单位。包括花费在此函数调用其它函数上的时间。

  返回值类型：bigint

- pg_stat_get_xact_tuples_returned(oid)

  描述：当前事务中参数为表时通过顺序扫描读取的行数，或参数为索引时返回的索引条目数。

返回值类型：bigint

- pg_lock_status()

描述：查询打开事务所持有的锁信息，所有用户均可执行该函数。

返回值类型：返回字段可参考 PG_LOCKS 视图返回字段，该视图是通过查询本函数得到的结果。

- pg_stat_get_xact_numscans(oid)

描述：当前事务中参数为表时执行的顺序扫描次数，或参数为索引时执行的索引扫描次数。

返回值类型：bigint

- pg_stat_get_xact_blocks_fetched(oid)

描述：当前事务中对表或索引的磁盘块获取请求数。

返回值类型：bigint

- pg_stat_get_xact_blocks_hit(oid)

描述：当前事务中对缓存中找到的表或索引的磁盘块获取请求数。

返回值类型：bigint

- pg_stat_get_xact_function_calls(oid)

描述：在当前事务中调用该函数的次数。

返回值类型：bigint

- pg_stat_get_xact_function_self_time(oid)

描述：在当前事务中仅花费在此函数上的时间，不包括花费在此函数内部调用其它函数上的时间。

返回值类型：bigint

- pg_stat_get_xact_function_total_time(oid)

描述：当前事务中该函数所花费的总挂钟时间（以微秒为单位），包括花费在此函数内部调用其它函数上的时间。

返回值类型：bigint

- pg_stat_get_wal_senders()

  描述：在主机端查询 walsender 信息。

  返回值类型：setofrecord

  返回字段说明：

表 5-52 返回字段说明

字段名称	字段类型	字段说明
pid	bigint	walsender 的线程号。
sender_pid	integer	walsender 的 pid 相对的轻量级线程号。
local_role	text	主节点类型。
peer_role	text	备节点类型。
peer_state	text	备节点状态。
state	text	walsender 状态。
catchup_start	timestamp with time zone	catchup 启动时间。
catchup_end	timestamp with time zone	catchup 结束时间。
sender_sent_location	text	主节点发送位置。
sender_write_location	text	主节点落盘位置。
sender_flush_location	text	主节点 flush 磁盘位置。
sender_replay_location	text	主节点 redo 位置。
receiver_received_location	text	备节点接收位置。
receiver_write_location	text	备节点落盘位置。

receiver_flush_location	text	备节点 flush 磁盘位置。
receiver_replay_location	text	备节点 redo 磁盘位置。
sync_percent	text	同步百分比。
sync_state	text	同步状态。
sync_priority	text	同步复制的优先级。
sync_most_available	text	最大可用模式设置。
channel	text	walsender 信道信息。

- get_paxos_replication_info()

  描述：查询 Paxos 模式下主机或备机的复制状态。

  返回值类型：setofrecord

  返回字段说明：

表 5-53 返回字段说明

字段名称	字段类型	字段说明
paxos_write_location	text	已经写入 DCF 的 XLog 位置。
paxos_commit_location	text	已经在 DCF 中达成一致的 XLog 位置。
local_write_location	text	节点的落盘位置。
local_flush_location	text	节点的 flush 磁盘位置。
local_replay_location	text	节点的 redo 磁盘位置。
dcf_replication_info	text	节点的 DCF 模块信息。

- pg_stat_get_stream_replications()

  描述：查询主备复制状态。

返回值类型：setofrecord

返回值说明如下。

表 5-54 返回字段说明

返回参数	返回参数类型	返回参数说明
local_role	text	本地角色。
static_connections	integer	连接统计。
db_state	text	数据库状态。
detail_information	text	详细信息。

- pg_stat_get_db_numbackends(oid)

  描述：处理该数据库活跃的服务器进程数目。

  返回值类型：integer

- pg_stat_get_db_xact_commit(oid)

  描述：数据库中已提交事务的数量。

  返回值类型：bigint

- pg_stat_get_db_xact_rollback(oid)

  描述：数据库中回滚事务的数量。

  返回值类型：bigint

- pg_stat_get_db_blocks_fetched(oid)

  描述：数据库中磁盘块抓取请求的总数。

  返回值类型：bigint

- pg_stat_get_db_blocks_hit(oid)

  描述：数据库在缓冲区中找到的磁盘块抓取请求的总数。

  返回值类型：bigint

- pg_stat_get_db_tuples_returned(oid)

  描述：为数据库返回的 Tuple 数。

  返回值类型：bigint

- pg_stat_get_db_tuples_fetched(oid)

  描述：为数据库中获取的 Tuple 数。

  返回值类型：bigint

- pg_stat_get_db_tuples_inserted(oid)

  描述：在数据库中插入 Tuple 数。

  返回值类型：bigint

- pg_stat_get_db_tuples_updated(oid)

  描述：在数据库中更新的 Tuple 数。

  返回值类型：bigint

- pg_stat_get_db_tuples_deleted(oid)

  描述：数据库中删除 Tuple 数。

  返回值类型：bigint

- pg_stat_get_db_conflict_lock(oid)

  描述：数据库中锁冲突的数量。

  返回值类型：bigint

- pg_stat_get_db_deadlocks(oid)

  描述：数据库中死锁的数量。

  返回值类型：bigint

- pg_stat_get_numscans(oid)

  描述：如果参数是一个表，则顺序扫描读取的行数目。如果参数是一个索引，则返回索引行的数目。

  返回值类型：bigint

- pg_stat_get_role_name(oid)

  描述：根据用户 oid 获取用户名。仅 sysadmin 和 monitor admin 用户可以访问。

  返回值类型：text

  示例：

```
 postgres=# select pg_stat_get_role_name(10);
pg_stat_get_role_name
----------------------
 gbase
(1 row)
```

- pg_stat_get_tuples_returned(oid)

  描述：如果参数是一个表，则顺序扫描读取的行数目。如果参数是一个索引，则返回的索引行的数目。

  返回值类型：bigint

- pg_stat_get_tuples_fetched(oid)

  描述：如果参数是一个表，则位图扫描抓取的行数目。如果参数是一个索引，则用简单索引扫描抓取的行数目。

  返回值类型：bigint

- pg_stat_get_tuples_inserted(oid)

  描述：插入表中行的数量。

  返回值类型：bigint

- pg_stat_get_tuples_updated(oid)

  描述：在表中已更新行的数量。

  返回值类型：bigint

- pg_stat_get_tuples_deleted(oid)

  描述：从表中删除行的数量。

  返回值类型：bigint

- pg_stat_get_tuples_changed(oid)

描述：该表上一次 analyze 或 autoanalyze 之后插入、更新、删除行的总数量。

返回值类型：bigint

- pg_stat_get_tuples_hot_updated(oid)

  描述：表热更新的行数。

  返回值类型：bigint

- pg_stat_get_live_tuples(oid)

  描述：表活行数。

  返回值类型：bigint

- pg_stat_get_dead_tuples(oid)

  描述：表死行数。

  返回值类型：bigint

- pg_stat_get_blocks_fetched(oid)

  描述：表或者索引的磁盘块抓取请求的数量。

  返回值类型：bigint

- pg_stat_get_blocks_hit(oid)

  描述：在缓冲区中找到的表或者索引的磁盘块请求数目。

  返回值类型：bigint

- pg_stat_get_partition_tuples_inserted(oid)

  描述：插入相应表分区中行的数量。

  返回值类型：bigint

- pg_stat_get_partition_tuples_updated(oid)

  描述：在相应表分区中已更新行的数量。

  返回值类型：bigint

- pg_stat_get_partition_tuples_deleted(oid)

  描述：从相应表分区中删除行的数量。

返回值类型：bigint

- pg_stat_get_partition_tuples_changed(oid)

  描述：该表分区上一次 analyze 或 autoanalyze 之后插入、更新、删除行的总数量。

  返回值类型：bigint

- pg_stat_get_partition_live_tuples(oid)

  描述：分区表活行数。

  返回值类型：bigint

- pg_stat_get_partition_dead_tuples(oid)

  描述：分区表死行数。

  返回值类型：bigint

- pg_stat_get_xact_tuples_fetched(oid)

  描述：事务中扫描的元组（tuple）行数。

  返回值类型：bigint

- pg_stat_get_xact_tuples_inserted(oid)

  描述：表相关的活跃子事务中插入的元组（tuple）数。

  返回值类型：bigint

- pg_stat_get_xact_tuples_deleted(oid)

  描述：表相关的活跃子事务中删除的元组数。

  返回值类型：bigint

- pg_stat_get_xact_tuples_hot_updated(oid)

  描述：表相关的活跃子事务中热更新的元组数。

  返回值类型：bigint

- pg_stat_get_xact_tuples_updated(oid)

  描述：表相关的活跃子事务中更新的元组数。

  返回值类型：bigint

- pg_stat_get_xact_partition_tuples_inserted(oid)

  描述：表分区相关的活跃子事务中插入的元组数。

  返回值类型：bigint

- pg_stat_get_xact_partition_tuples_deleted(oid)

  描述：表分区相关的活跃子事务中删除的元组数。

  返回值类型：bigint

- pg_stat_get_xact_partition_tuples_hot_updated(oid)

  描述：表分区相关的活跃子事务中热更新的元组数。

  返回值类型：bigint

- pg_stat_get_xact_partition_tuples_updated(oid)

  描述：表分区相关的活跃子事务中更新的元组数。

  返回值类型：bigint

- pg_stat_get_last_vacuum_time(oid)

  描述：用户在该表上最后一次手动启动清理或者 autovacuum 线程启动清理的时间。

  返回值类型：timestamptz

- pg_stat_get_last_autovacuum_time(oid)

  描述：autovacuum 守护进程在该表上最后一次启动清理的时间。

  返回值类型：timestamptz

- pg_stat_get_vacuum_count(oid)

  描述：用户在该表上启动清理的次数。

  返回值类型：bigint

- pg_stat_get_autovacuum_count(oid)

  描述：autovacuum 守护进程在该表上启动清理的次数。

  返回值类型：bigint

- pg_stat_get_last_analyze_time(oid)

描述：用户在该表上最后一次手动启动分析或者 autovacuum 线程启动分析的时间。

返回值类型：timestamptz

- pg_stat_get_last_autoanalyze_time(oid)

  描述：autovacuum 守护进程在该表上最后一次启动分析的时间。

  返回值类型：timestamptz

- pg_stat_get_analyze_count(oid)

  描述：用户在该表上启动分析的次数。

  返回值类型：bigint

- pg_stat_get_autoanalyze_count(oid)

  描述：autovacuum 守护进程在该表上启动分析的次数。

  返回值类型：bigint

- pg_total_autovac_tuples(bool,bool)

描述：返回 total autovac 相关的元组记录，如 nodename,nspname,relname 以及各类元组的 IUD 信息，入参分别为：是否查询 relation 信息，是否查询 local 信息。

返回值类型：setofrecord

返回参数说明如下。

表 5-55 返回参数说明

返回参数	返回参数类型	返回参数说明
nodename	name	节点名称。
nspname	name	名称空间名称。
relname	name	表、索引、视图等对象名称。
partname	name	分区名称。
n_dead_tuples	bigint	表分区内的死行数。

n_live_tuples	bigint	表分区内的活行数。
changes_since_analyze	bigint	analyze 产生改变的数量。

- pg_autovac_status(oid)

描述：返回和 autovac 状态相关的参数信息，如 nodename,nspname,relname,analyze,vacuum 设置，analyze/vacuum 阈值, analyze/vacuum tuple 数等。仅 sysadmin 可以使用该函数。

返回值类型：setofrecord

返回值参数说明如下。

表 5-56 返回值参数说明

返回参数	返回参数类型	返回参数说明
nspname	text	名称空间名称。
relname	text	表、索引、视图等对象名称。
nodename	text	节点名称。
doanalyze	Boolean	是否执行 analyze。
anltuples	bigint	analyze tuple 数量。
anlthresh	bigint	analyze 阈值。
dovacuum	Boolean	是否执行 vacuum。
vactuples	bigint	vacuum tuple 数量。
vacthresh	bigint	vacuum 阈值。

- pg_autovac_timeout(oid)

描述：返回某个表做 autovac 连续超时的次数，表信息非法或 node 信息异常返回 NULL。

返回值类型：bigint

- pg_stat_get_last_data_changed_time(oid)

描述：insert/update/delete, exchange/truncate/drop partition 在该表上最后一次操作的时间，PG_STAT_ALL_TABLES 视图 last_data_changed 列的数据是通过该函数求值，在表数量很大的场景中，通过视图获取表数据最后修改时间的性能较差，建议直接使用该函数获取表数据的最后修改时间。

返回值类型：timestamptz

- pg_stat_set_last_data_changed_time(oid)

描述：手动设置该表上最后一次 insert/update/delete, exchange/truncate/drop partition 操作的时间。

返回值类型：void

- pg_backend_pid()

描述：当前会话的服务器线程的线程 ID。

返回值类型：integer

- pg_stat_get_activity(integer)

描述：返回一个关于带有特殊 PID 的后台进程的记录信息，当参数为 NULL 时，则返回每个活动的后台进程的记录。返回结果不包含 connection_info 列。初始用户、系统管理员和 monadmin 可以查看所有的数据，普通用户只能查询自己的结果。

示例：

```
 postgres=# select * from pg_stat_get_activity(139881386280704);
datid | pid | sessionid | usesysid | application_name | state | query | waiting
| xact_start
| query_start | backend_start | state_change | client_addr | client_hostname |
client_port | e
nqueue | query_id | srespool | global_sessionid | unique_sql_id | trace_id
------+-----+-----------+----------+------------------+-------+-------+-----
----+------------
+------------+--------------+-------------+------------+----------------
+-------------+--
------+----------+----------+------------------+--------------+----------
(0 rows)
```

返回值类型：setofrecord

返回参数说明如下。

表 5-57 返回参数说明

返回参数	返回参数类型	返回参数说明
datid	oid	用户会话在后台连接到的数据库 OID。
pid	bigint	后台线程 ID。
sessionid	bigint	会话 ID。
usesysid	oid	登录该后台的用户 OID。
application_name	text	连接到该后台的应用名。
state	text	该后台当前总体状态。
query	text	该后台的最新查询。如果 state 状态是 active(活跃的），此字段显示当前正在执行的查询。所有其他情况表示上一个查询。
waiting	Boolean	如果后台当前正等待锁则为 true。
xact_start	timestamp with time zone	启动当前事务的时间，如果没有事务是活跃的，则为 null。 如果当前查询是首个事务，则这列等同于 query_start 列。
query_start	timestamp with time zone	开始当前活跃查询的时间，如果 state 的值不是 active，则这个值是上一个查询的开始时间。
backend_start	timestamp with time zone	该过程开始的时间，即当客户端连接服务器时。
state_change	timestamp with time zone	上次状态改变的时间。

client_addr	inet	连接到该后台的客户端的 IP 地址。如果此字段是 null，它表明通过服务器机器上 UNIX 套接字连接客户端或者这是内部进程，如 autovacuum。
client_hostname	text	客户端的主机名，这个字段是通过 client_addr 的反向 DNS 查找得到。这个字段只有在启动 log_hostname 且使用 IP 连接时才非空。
client_port	integer	客户端用于与后台通讯的 TCP 端口号，如果使用 Unix 套接字，则为-1。
enqueue	text	该字段暂不支持。
query_id	bigint	查询语句的 ID。
srespool	name	资源池名字。
global_sessionid	text	全局会话 id。
unique_sql_id	bigint	语句的 unique sql id。
trace_id	text	驱动传入的 trace id，与应用的一次请求相关联。

- pg_stat_get_activity_with_conninfo(integer)

描述：返回一个关于带有特殊 PID 的后台进程的记录信息，当参数为 NULL 时，则返回每个活动的后台进程的记录。初始用户、系统管理员和 monadmin 可以查看所有的数据，普通用户只能查询自己的结果。

返回值类型：setofrecord

返回值说明如下。

表 5-58 返回值说明

返回值	返回值类型	返回值说明

datid	oid	用户会话在后台连接到的数据库 OID。
pid	bigint	后台线程 ID。
sessionid	bigint	会话 ID。
usesysid	oid	登录该后台的用户 OID。
application_name	text	连接到该后台的应用名。
state	text	改后台当前总体状态
query	text	该后台的最新查询。如果 state 状态是 active(活跃的），此字段显示当前正在执行的查询。所有其他情况表示上一个查询。
waiting	Boolean	如果后台当前正等待锁则为 true
xact_start	timestamp with time zone	启动当前事务的时间，如果没有事务是活跃的，则为 null。如果当前查询是首个事务，则这列等同于 query_start 列。
query_start	timestamp with time zone	开始当前活跃查询的时间，如果 state 的值不是 active，则这个值是上一个查询的开始时间。
backend_start	timestamp with time zone	该过程开始的时间，即当客户端连接服务器时。
state_change	timestamp with time zone	上次状态改变的时间。
client_addr	inet	连接到该后台的客户端的 IP 地址。如果此字段是 null，它表明通过服务器机器上 UNIX 套接字连接客户端或者这是内部进程，如 autovacuum
client_hostname	text	客户端的主机名，这个字段是通过 client_addr 的反向 DNS 查找得到。这个字段只有在启动

		log_hostname 且使用 IP 连接时才非空。
client_port	integer	客户端用于与后台通讯的 TCP 端口号，如果使用 Unix 套接字，则为-1。
enqueue	text	该字段暂不支持。
query_id	bigint	查询语句的 ID。
connection_info	text	json 格式字符串，记录当前连接数据库的驱动类型、驱动版本号、当前驱动的部署路径、进程属主用户等信息。
srespool	name	资源池名字。
global_sessionid	text	全局会话 ID。
unique_sql_id	bigint	语句的 unique sql id。
trace_id	text	驱动传入的 trace id，与应用的一次请求相关联。

- pg_user_iostat(text)

  描述：显示和当前用户执行作业正在运行时的 IO 负载管理相关信息。

  返回值类型：record

  函数返回字段说明如下：

表 5-59 返回值说明

名称	类型	描述
userid	oid	用户 id。
min_curr_iops	int4	当前该用户 io 在数据库节点中的最小值。对于行存，以万次/s 为单位；对于列存，以次/s 为单位。
max_curr_iops	int4	当前该用户 io 在数据库节点中的最大值。对于行

		存，以万次/s 为单位；对于列存，以次/s 为单位。
min_peak_iops	int4	该用户 io 峰值中，数据库节点的最小值。对于行存，以万次/s 为单位；对于列存，以次/s 为单位。
max_peak_iops	int4	该用户 io 峰值中，数据库节点的最大值。对于行存，以万次/s 为单位；对于列存，以次/s 为单位。
io_limits	int4	用户指定的资源池所设置的 io_limits。对于行存，以万次/s 为单位；对于列存，以次/s 为单位。
io_priority	text	该用户所设 io_priority。对于行存，以万次/s 为单位；对于列存，以次/s 为单位。
curr_io_limits	int4	使用 io_priority 管控 io 时的实时 io_limits 值。

- pg_stat_get_function_calls(oid)

  描述：函数已被调用次数。

  返回值类型：bigint

- pg_stat_get_function_self_time(oid)

  描述：只有在此函数上所花费的时间。此函数调用其它函数上花费的时间被排除在外。

  返回值类型：bigint

- pg_stat_get_backend_idset()

  描述：设置当前活动的服务器进程数（从 1 到活动服务器进程的数量）。

  返回值类型：setofinteger

- pg_stat_get_backend_pid(integer)

  描述：给定的服务器线程的线程 ID。

  返回值类型：bigint

- pg_stat_get_backend_dbid(integer)

  描述：给定服务器进程的数据库 ID。

返回值类型：oid

- pg_stat_get_backend_userid(integer)

描述：给定服务器进程的用户 ID。

返回值类型：oid

- pg_stat_get_backend_activity(integer)

描述：给定服务器进程的当前活动查询，仅在调用者是系统管理员或被查询会话的用户，并且打开 track_activities 的时候才能获得结果。

返回值类型：text

- pg_stat_get_backend_waiting(integer)

描述：如果给定服务器进程在等待某个锁，并且调用者是系统管理员或被查询会话的用户，并且打开 track_activities 的时候才返回真。

返回值类型：Boolean

- pg_stat_get_backend_activity_start(integer)

描述：给定服务器进程当前正在执行的查询的起始时间，仅在调用者是系统管理员或被查询会话的用户，并且打开 track_activities 的时候才能获得结果。

返回值类型：timestampwithtimezone

- pg_stat_get_backend_xact_start(integer)

描述：给定服务器进程当前正在执行的事务的开始时间，但只有当前用户是系统管理员或被查询会话的用户，并且打开 track_activities 的时候才能获得结果。

返回值类型：timestampwithtimezone

- pg_stat_get_backend_start(integer)

描述：给定服务器进程启动的时间，如果当前用户不是系统管理员或被查询的后端的用户，则返回 NULL。

返回值类型：timestampwithtimezone

- pg_stat_get_backend_client_addr(integer)

描述：连接到给定客户端后端的 IP 地址。如果是通过 Unix 域套接字连接的则返回 NULL；

如果当前用户不是系统管理员或被查询会话的用户，也返回 NULL。

返回值类型：inet

- pg_stat_get_backend_client_port(integer)

描述：连接到给定客户端后端的 TCP 端口。如果是通过 Unix 域套接字连接的则返回-1；如果当前用户不是系统管理员或被查询会话的用户，也返回 NULL。

返回值类型：integer

- pg_stat_get_bgwriter_timed_checkpoints()

描述：后台写进程开启定时检查点的时间（因为 checkpoint_timeout 时间已经过期了）。

返回值类型：bigint

- pg_stat_get_bgwriter_requested_checkpoints()

描述：后台写进程开启基于后端请求的检查点的时间，因为已经超过了 checkpoint_segments 或因为已经执行了 CHECKPOINT。

返回值类型：bigint

- pg_stat_get_bgwriter_buf_written_checkpoints()

描述：在检查点期间后台写进程写入的缓冲区数目。

返回值类型：bigint

- pg_stat_get_bgwriter_buf_written_clean()

描述：为日常清理脏块，后台写进程写入的缓冲区数目。

返回值类型：bigint

- pg_stat_get_bgwriter_maxwritten_clean()

描述：后台写进程停止清理扫描的时间，因为已经写入了更多的缓冲区（相比 bgwriter_lru_maxpages 参数声明的缓冲区数）。

返回值类型：bigint

- pg_stat_get_buf_written_backend()

描述：后端进程写入的缓冲区数，因为它们需要分配一个新的缓冲区。

返回值类型：bigint

- pg_stat_get_buf_alloc()

  描述：分配的总缓冲区数。

  返回值类型：bigint

- pg_stat_clear_snapshot()

  描述：清理当前的统计快照。

  返回值类型：void

- pg_stat_reset()

  描述：为当前数据库重置统计计数器为 0（需要系统管理员权限）。

  返回值类型：void

- pg_stat_reset_shared(text)

  描述：重置 shared cluster 每个节点当前数据统计计数器为 0（需要系统管理员权限）。

  返回值类型：void

- pg_stat_reset_single_table_counters(oid)

  描述：为当前数据库中的一个表或索引重置统计为 0（需要系统管理员权限）。

  返回值类型：void

- pg_stat_reset_single_function_counters(oid)

  描述：为当前数据库中的一个函数重置统计为 0（需要系统管理员权限）。

  返回值类型：void

- pg_stat_session_cu(int, int, int)

  描述：获取当前节点所运行 session 的 CU 命中统计信息。

  返回值类型：record

- pg_stat_get_cu_mem_hit(oid)

  描述：获取当前节点当前数据库中一个列存表的 CU 内存命中次数。

  返回值类型：bigint

- pg_stat_get_cu_hdd_sync(oid)

  描述：获取当前节点当前数据库中一个列存表从磁盘同步读取 CU 次数。

  返回值类型：bigint

- pg_stat_get_cu_hdd_asyn(oid)

  描述：获取当前节点当前数据库中一个列存表从磁盘异步读取 CU 次数。

  返回值类型：bigint

- pg_stat_get_db_cu_mem_hit(oid)

  描述：获取当前节点一个数据库 CU 内存命中次数。

  返回值类型：bigint

- pg_stat_get_db_cu_hdd_sync(oid)

  描述：获取当前节点一个数据库从磁盘同步读取 CU 次数。

  返回值类型：bigint

- fenced_udf_process(integer)

  描述：查看本地 UDF Master 和 Work 进程数。入参为 1 时查看 master 进程数，入参为 2 时查看 worker 进程数，入参为 3 时杀死所有 worker 进程。

  返回值类型：text

- total_cpu()

  描述：获取当前节点使用的 CPU 时间，单位是 jiffies。

  返回值类型：bigint

- mot_global_memory_detail()

  描述：检查 MOT 全局内存的大小，主要包括数据和索引。

  返回值类型：record

- mot_local_memory_detail()

  描述：检查 MOT 局部内存的大小，主要包括数据和索引。

  返回值类型：record

- mot_session_memory_detail()

  描述：检查所有会话对 MOT 内存的使用情况。

  返回值类型：record

- total_memory()

  描述：获取当前节点使用的虚拟内存大小，单位 KB。

  返回值类型：bigint

- pg_stat_get_db_cu_hdd_asyn(oid)

  描述：获取当前节点一个数据库从磁盘异步读取 CU 次数。

  返回值类型：bigint

- pg_stat_bad_block(text, int, int, int, int, int, timestamp with time zone, timestamp with time zone)

  描述：获取当前节点自启动后，读取出现 Page/CU 的损坏信息。**示例**：

```
postgres=# select * from pg_stat_bad_block();
```

  返回值类型：record

- pg_stat_bad_block_clear()

  描述：清理节点记录的读取出现的 Page/CU 损坏信息（需要系统管理员权限）。

  返回值类型：void

- gs_respool_exception_info(pool text)

  描述：查看某个资源池关联的查询规则信息。

  返回值类型：record

- gs_control_group_info(pool text)

  描述：查看资源池关联的控制组信息

  返回值类型：record

  返回信息如下：

表 5-60 返回值说明

属性	属性值	描述
name	class_a:workload_a1	class 和 workload 名称
class	class_a	Class 控制组名称
workload	workload_a1	Workload 控制组名称
type	DEFWD	控制组类型（Top、CLASS、BAKWD、DEFWD、TSWD）
gid	87	控制组 id
shares	30	占父节点 CPU 资源的百分比
limits	0	占父节点 CPU 核数的百分比
rate	0	Timeshare 中的分配比例
cpucores	0-3	CPU 核心数

- gs_all_control_group_info()

  描述：查看数据库内所有的控制组信息。

  返回值类型：record

- gs_get_control_group_info()

  描述：查看所有的控制组信息。

  返回值类型：record

- get_instr_workload_info(integer)

  描述：获取数据库主节点上事务量信息，事务时间信息。

  返回值类型：record

表 5-61 返回值说明

属性	属性值	描述

resourcepool_oid	10	资源池的 oid(逻辑同负载等价)
commit_counter	4	前端事务 commit 数量
rollback_counter	1	前端事务 rollback 数量
resp_min	949	前端事务最小响应时间（单位：微秒）
resp_max	201891	前端事务最大响应时间（单位：微秒）
resp_avg	43564	前端事务平均响应时间(单位：微秒)
resp_total	217822	前端事务总响应时间（单位：微秒）
bg_commit_counter	910	后端事务 commit 数量
bg_rollback_counter	0	后端事务 rollback 数量
bg_resp_min	97	后端事务最小响应时间（单位：微秒）
bg_resp_max	678080687	后端事务最大响应时间（单位：微秒）
bg_resp_avg	327847884	后端事务平均响应时间（单位：微秒）
bg_resp_total	298341575300	后端事务总响应时间（单位：微秒）

- pv_instance_time()

  描述：获取当前节点上各个关键阶段的时间消耗。

  返回值类型：record

表 5-62 返回值说明

Stat_name 属性	属性值	描述
DB_TIME	1062385	所有线程端到端的墙上时间（WALL TIME）消耗总和(单位：微秒)
CPU_TIME	311777	所有线程 CPU 时间消耗总和(单位：

		微秒)
EXECUTION_TIME	380037	消耗在执行器上的时间总和(单位:微秒)
PARSE_TIME	6033	消耗在 SQL 解析上的时间总和(单位：微秒)
PLAN_TIME	173356	消耗在执行计划生成上的时间总和(单位：微秒)
REWRITE_TIME	2274	消耗在查询重写上的时间总和(单位:微秒)
PL_EXECUTION_TIME	0	消耗在PL/SQL执行上的时间总和(单位：微秒)
PL_COMPILATION_TIME	557	消耗在 SQL 编译上的时间总和(单位：微秒)
NET_SEND_TIME	1673	消耗在网络发送上的时间总和(单位:微秒)
DATA_IO_TIME	426622	消耗在数据读写上的时间总和(单位:微秒)

- DBE_PERF.get_global_instance_time()

  描述：提供 GBase 8s 各个关键阶段的时间消耗，查询该函数必须具有 sysadmin 权限。

  返回值类型：record

- get_instr_unique_sql()

  描述：获取当前节点的执行语句（归一化 SQL）信息，查询该函数必须具有 sysadmin 权限。

  返回值类型：record

- reset_unique_sql(text, text, bigint)

描述：重置系统执行语句（归一化 SQL）信息，执行该函数必须具有 sysadmin 权限。第一个参数取值范围“global/local”，global 表示清理所有节点上的信息，local 表示只清理当前节点；第二参数取值范围“ALL/BY_USERID/BY_CNID”，ALL 表示清理所有信息，BY_USERID 表示通过指定 USERID 清理只属于该用户的 sql 信息，BY_CNID 表示清理系统中涉及到该数据库主节点的 sql 信息；第三个参数表示具体的 CNID 和 USERID，如果第二个参数为 ALL，第三个参数不起作用，可以取任意值。

返回值类型：boolean

说明

在 GBase 8s 中 global 与 local 功能一致，取值范围不支持 BY_CNID。

- get_instr_wait_event(NULL)

  描述：获取当前节点 event 等待的统计信息。

  返回值类型：record

- get_instr_user_login()

  描述：获取当前节点的用户登入登出次数信息，查询该函数必须具有 sysadmin 或者 monitor admin 权限。

  返回值类型：record

- get_instr_rt_percentile(integer)

  描述：获取数据库 SQL 响应时间 P80,P95 分布信息。

  返回值类型：record

- get_node_stat_reset_time()

  描述：获取当前节点的统计信息重置（重启，主备倒换，数据库删除）时间。

  返回值类型：record

- DBE_PERF.get_global_os_runtime()

  描述：显示当前操作系统运行的状态信息，查询该函数必须具有 sysadmin 权限。

  返回值类型：record

- DBE_PERF.get_global_os_threads()

描述：提供 GBase 8s 中所有正常节点下的线程状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_workload_sql_count()

描述：提供 GBase 8s 中不同负载 SELECT，UPDATE，INSERT，DELETE，DDL，DML，DCL 计数信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_workload_sql_elapse_time()

描述：提供 GBase 8s 中不同负载 SELECT，UPDATE，INSERT，DELETE，响应时间信息（TOTAL,AVG, MIN, MAX），查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_workload_transaction()

描述：获取 GBase 8s 内所有节点上的事务量信息，事务时间信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_session_stat()

描述：获取 GBase 8s 节点上的会话状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

说明

状态信息有 17 项：commit、rollback、sql、table_scan、blocks_fetched、physical_read_operation、shared_blocks_dirtied、local_blocks_dirtied、shared_blocks_read、local_blocks_read、blocks_read_time、blocks_write_time、sort_imemory、sort_idisk、cu_mem_hit、cu_hdd_sync_read、cu_hdd_asyread。

- DBE_PERF.get_global_session_time()

描述：提供 GBase 8s 各节点各个关键阶段的时间消耗，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_session_memory()

描述：汇聚各节点的 Session 级别的内存使用情况，包含执行作业在数据节点上 Postgres 线程和 Stream 线程分配的所有内存，单位为 MB，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_session_memory_detail()

描述：汇聚各节点的线程的内存使用情况，以 MemoryContext 节点来统计，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- create_wlm_session_info(int flag)

描述：将当前内存中记录的 TopSQL 查询语句级别相关统计信息清理。该函数只有管理员用户可以执行。

返回值类型：int

- pg_stat_get_wlm_session_info(int flag)

描述：获取当前内存中记录的 TopSQL 查询语句级别相关统计信息，当传入的参数不为 0 时，会将这部分信息从内存中清理掉。该函数只有 system admin 和 monitor admin 用户可以执行。

返回值类型：record

- gs_paxos_stat_replication()

描述：在主机端查询备机信息。目前只支持集中式 DCF 模式。

返回值类型：setofrecord

返回字段说明如下：

表 5-63 返回值类型及说明

字段名称	字段类型	字段说明
local_role	text	发送日志节点的角色。
peer_role	text	接收日志节点的角色。

local_dcf_role	text	发送日志节点的 DCF 角色。
peer_dcf_role	text	接收日志节点的 DCF 角色。
peer_state	text	接收日志节点的状态。
sender_write_location	text	发送日志节点写到 xlog buffer 的位置。
sender_commit_location	text	发送日志节点 DCF 日志达成一致性点。
sender_flush_location	text	发送日志节点写到 xlog disk 的位置。
sender_replay_location	text	发送日志节点 replay 的位置。
receiver_write_location	text	接收日志节点写到 xlog buffer 的位置。
receiver_commit_location	text	接收日志节点 DCF 日志达成一致性点。
receiver_flush_location	text	接收日志节点写到 xlog disk 的位置。
receiver_replay_location	text	接收日志节点重演 xlog 的位置。
sync_percent	text	同步百分比。
dcf_run_mode	int4	DCF 同步模式。
channel	text	信道信息。

- gs_wlm_get_resource_pool_info(int)

  描述：获取所有用户的资源使用统计信息，入参为 int 类型可以为可以为任意 int 值或 NULL。

  返回值类型：record

- gs_wlm_get_all_user_resource_info()

  描述：获取所有用户的资源使用统计信息。

  返回值类型：record

- gs_wlm_get_user_info(int)

描述：获取所有用户的相关信息，入参为 int 类型可以为可以为任意 int 值或 NULL。该函数只有 sysadmin 权限的用户可以执行。

返回值类型：record

- gs_wlm_get_workload_records()

描述：获取动态负载管理下的所有作业信息，该函数只在动态负载管理开的情况下有效。

返回值类型：record

- gs_wlm_readjust_user_space()

描述：修正所有用户的存储空间使用情况。该函数只有管理员用户可以执行。

返回值类型：record

- gs_wlm_readjust_user_space_through_username(text name)

描述：修正指定用户的存储空间使用情况。该函数普通用户只能修正自己的使用情况，只有管理员用户可以修正所有用户的使用情况。当 name 指定位“0000”，表示需要修正所有用户的使用情况。

返回值类型：record

- gs_wlm_readjust_user_space_with_reset_flag(text name, boolean isfirst)

描述：修正指定用户的存储空间使用情况。入参 isfirst 为 true 表示从 0 开始统计，否则从上一次结果继续统计。该函数普通用户只能修正自己的使用情况，只有管理员用户可以修正所有用户的使用情况。当 name 指定位“0000”，表示需要修正所有用户的使用情况。

返回值类型：record

- gs_wlm_session_respool(bigint)

描述：获取当前所有后台线程的 session resouce pool 相关信息，入参为 bigint 类型可以为可以为任意 bigint 值或 NULL。

返回值类型：record

- gs_wlm_get_session_info()

描述：目前该接口已废弃，暂不可用。

- gs_wlm_get_user_session_info()

描述：目前该接口已废弃，暂不可用。

- gs_io_wait_status()

描述：目前该接口不支持单机和集中式，暂不可用。

- global_stat_get_hotkeys_info()

描述：获取整个数据库实例中热点 key 的统计情况。目前该接口不支持单机和集中式，暂不可用。

- global_stat_clean_hotkeys()

描述：清理整个数据库实例中热点 key 的统计信息。目前该接口不支持单机和集中式，暂不可用。

- DBE_PERF.get_global_session_stat_activity()

描述：汇聚 GBase 8s 内各节点上正在运行的线程相关的信息，查询该函数必须具有 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_thread_wait_status()

描述：汇聚所有节点上工作线程（backend thread）以及辅助线程（auxiliary thread）的阻塞等待情况，查询该函数必须具有 sysadmin 和 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_operator_history_table()

描述：汇聚当前用户数据库主节点上执行作业结束后的算子相关记录（持久化），查询该函数必须具有 sysadmin 和 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_operator_history()

描述：汇聚当前用户数据库主节点上执行作业结束后的算子相关记录，查询该函数必须具有 sysadmin 和 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_operator_runtime()

描述：汇聚当前用户数据库主节点上执行作业实时的算子相关记录，查询该函数必须具有 sysadmin 和 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_statement_complex_history()

描述：汇聚当前用户数据库主节点上复杂查询的历史记录，查询该函数必须具有 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_statement_complex_history_table()

描述：汇聚当前用户数据库主节点上复杂查询的历史记录（持久化），查询该函数必须具有 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_statement_complex_runtime()

描述：汇聚当前用户数据库主节点上复杂查询的实时信息，查询该函数必须具有 sysadmin 和 monadmin 权限。

返回值类型：record

- DBE_PERF.get_global_memory_node_detail()

描述：汇聚所有节点某个数据库节点内存使用情况，查询该函数必须具有 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_shared_memory_detail()

描述：汇聚所有节点已产生的共享内存上下文的使用信息，查询该函数必须具有 monitoradmin 权限。

返回值类型：record

- DBE_PERF.get_global_statio_all_indexes()

描述：汇聚所有节点当前数据库中的每个索引行，显示特定索引的 I/O 的统计，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_stat_all_tables()

描述：显示汇聚各节点数据中每个表（包括 TOAST 表）的一行的统计信息

返回值类型：record

- DBE_PERF.get_global_stat_all_tables()

描述：显示各节点数据中每个表（包括 TOAST 表）的一行的统计信息。

返回值类型：record

- DBE_PERF.get_local_toastname_and_toastindexname()

描述：提供本地 toast 表的 name 和 index 和其关联表的对应关系，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_statio_all_indexes()

描述：统计所有节点当前数据库中的每个索引行，显示特定索引的 I/O 的统计，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_statio_all_sequences()

描述：提供命名空间中所有 sequences 的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_statio_all_tables()

描述：汇聚各节点的数据库中每个表 I/O 的统计，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_statio_all_tables()

描述：统计 GBase 8s 内数据库中每个表 I/O 的统计，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_local_toast_relation()

描述：提供本地 toast 表的 name 和其关联表的对应关系，查询该函数必须具有 sysadmin 权限

返回值类型：record

- DBE_PERF.get_global_statio_sys_indexes()

描述：汇聚各节点的命名空间中所有系统表索引的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_statio_sys_indexes()

描述：统计各节点的命名空间中所有系统表索引的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_statio_sys_sequences()

描述：提供命名空间中所有系统表为 sequences 的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_statio_sys_tables()

描述：提供各节点的命名空间中所有系统表的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_statio_sys_tables()

描述：GBase 8s 内汇聚命名空间中所有系统表的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_statio_user_indexes()

描述：各节点的命名空间中所有用户关系表索引的 IO 状态信息，查询该函数必须具有

sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_statio_user_indexes()

描述：GBase 8s 内汇聚命名空间中所有用户关系表索引的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_statio_user_sequences()

描述：显示各节点的命名空间中所有用户的 sequences 的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_statio_user_tables()

描述：显示各节点的命名空间中所有用户关系表的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_statio_user_tables()

描述：GBase 8s 数据库内汇聚命名空间中所有用户关系表的 IO 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_stat_db_cu()

描述：视图查询 GBase 8s 各个节点，每个数据库的 CU 命中情况，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_all_indexes()

描述：汇聚所有节点数据库中每个索引的统计信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_stat_all_indexes()

描述：统计所有节点数据库中每个索引的统计信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_sys_tables()

描述：汇聚各节点 pg_catalog、information_schema 模式的所有命名空间中系统表的统计信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_stat_sys_tables()

描述：统计各节点 pg_catalog、information_schema 模式的所有命名空间中系统表的统计信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_sys_indexes()

描述：汇聚各节点 pg_catalog、information_schema 模式中所有系统表的索引状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_stat_sys_indexes()

描述：统计各节点 pg_catalog、information_schema 模式中所有系统表的索引状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_user_tables()

描述：汇聚所有命名空间中用户自定义普通表的状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_stat_user_tables()

描述：统计所有命名空间中用户自定义普通表的状态信息，查询该函数必须具有

sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_user_indexes()

描述：汇聚所有数据库中用户自定义普通表的索引状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_stat_user_indexes()

描述：统计所有数据库中用户自定义普通表的索引状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_database()

描述：汇聚所有节点数据库统计信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_database_conflicts()

描述：统计所有节点数据库统计信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_xact_all_tables()

描述：汇聚命名空间中所有普通表和 toast 表的事务状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_stat_xact_all_tables()

描述：统计命名空间中所有普通表和 toast 表的事务状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_xact_sys_tables()

描述：汇聚所有节点命名空间中系统表的事务状态信息，查询该函数必须具有 sysadmin

权限。

返回值类型：record

- DBE_PERF.get_summary_stat_xact_sys_tables()

描述：统计所有节点命名空间中系统表的事务状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_xact_user_tables()

描述：汇聚所有节点命名空间中用户表的事务状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_stat_xact_user_tables()

描述：统计所有节点命名空间中用户表的事务状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_user_functions()

描述：汇聚所有节点命名空间中用户定义函数的事务状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_xact_user_functions()

描述：统计所有节点命名空间中用户定义函数的事务状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_stat_bad_block()

描述：汇聚所有节点表、索引等文件的读取失败信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_file_redo_iostat()

  描述：统计所有节点表、索引等文件的读取失败信息，查询该函数必须具有 sysadmin 权限。

  返回值类型：record

- DBE_PERF.get_global_file_iostat()

  描述：汇聚所有节点数据文件 IO 的统计，查询该函数必须具有 sysadmin 权限。

  返回值类型：record

- DBE_PERF.get_global_locks()

  描述：汇聚所有节点的锁信息，查询该函数必须具有 sysadmin 和 monadmin 权限。

  返回值类型：record

- DBE_PERF.get_global_replication_slots()

  描述:汇聚所有节点上逻辑复制信息,查询该函数必须具有 sysadmin 和 monadmin 权限。

  返回值类型：record

- DBE_PERF.get_global_bgwriter_stat()

  描述：汇聚所有节点后端写进程活动的统计信息，查询该函数必须具有 sysadmin 权限。

  返回值类型：record

- DBE_PERF.get_global_replication_stat()

  描述：汇聚各节点日志同步状态信息，如发起端发送日志位置，收端接收日志位置等，查询该函数必须具有 sysadmin 和 monadmin 权限。

  返回值类型：record

- DBE_PERF.get_global_transactions_running_xacts()

  描述：汇聚各节点运行事务的信息，查询该函数必须具有 sysadmin 权限。

  返回值类型：record

- DBE_PERF.get_summary_transactions_running_xacts()

  描述：统计各节点运行事务的信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_transactions_prepared_xacts()

描述：汇聚各节点当前准备好进行两阶段提交的事务的信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_transactions_prepared_xacts()

描述：统计各节点当前准备好进行两阶段提交的事务的信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_statement()

描述：汇聚各节点历史执行语句状态信息，查询该函数必须具有 monitor admin 和 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_statement_count()

描述：汇聚各节点 SELECT，UPDATE，INSERT，DELETE，响应时间信息（TOTAL,AVG, MIN, MAX），查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_config_settings()

描述：汇聚各节点 GUC 参数配置信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_wait_events()

描述：汇聚各节点 wait events 状态信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_statement_responsetime_percentile()

描述：获取 GBase 8s 的 SQL 响应时间 P80，P95 分布信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_summary_user_login()

描述：统计 GBase 8s 各节点用户登入登出次数信息，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.get_global_record_reset_time()

描述：汇聚数据库统计信息重置（重启，主备倒换，数据库删除）时间，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.track_memory_context(context_list text)

描述：设置需要统计内存申请详细信息的内存上下文。入参为内存上下文的名称，使用“，”分隔，如“ThreadTopMemoryContext, SessionCacheMemoryContext”，注意该内存上下文名称是上下文敏感的。此外，单个内存上下文的长度为 63，超过的部分会被截断。而且一次能够统计的内存上下文上限为 16 个，设置超过 16 个内存上下文会设置失败。每一次调用该函数都会将上次统计的结果清空，当入参指定为“”时，表示取消该统计功能。只有初始用户（super user）或者具有 monadmin 权限的用户可以执行该函数。

返回值类型：boolean

- DBE_PERF.track_memory_context_detail()

描述：获取 DBE_PERF.track_memory_context 函数指定的内存上下文的内存申请详细信息。返回值的定义见视图 DBE_PERF.track_memory_context_detail。只有初始用户（super user）或者具有 monadmin 权限的用户可以执行该函数。

返回值类型：record

- pg_stat_get_mem_mbytes_reserved(tid)

描述：统计资源管理相关变量值，仅用于定位问题使用。

参数：线程 id。

返回值类型：text

- gs_wlm_user_resource_info(name text)

描述：查询具体某个用户的资源限额和资源使用情况。

返回值类型：record

- pg_stat_get_file_stat()

描述：通过对数据文件 IO 的统计，反映数据的 IO 性能，用以发现 IO 操作异常等性能问题。

返回值类型：record

- pg_stat_get_redo_stat()

描述：用于统计会话线程日志回放情况。

返回值类型：record

- pg_stat_get_status(int8)

描述：可以检测当前实例中工作线程（backend thread）以及辅助线程（auxiliary thread）的阻塞等待情况。

返回值类型：record

- get_local_rel_iostat()

描述：查询当前节点的数据文件 IO 状态累计值。

返回值类型：record

- DBE_PERF.get_global_rel_iostat()

描述：汇聚所有节点数据文件 IO 的统计，查询该函数必须具有 sysadmin 权限。

返回值类型：record

- DBE_PERF.global_threadpool_status()

描述：显示在所有节点上的线程池中工作线程及会话的状态信息。函数返回信息具体字段 GLOBAL_THREADPOOL_STATUS 字段。

返回值类型：record

- remote_bgwriter_stat()

描述：显示数据库所有实例的 bgwriter 线程刷页信息，候选 buffer 链中页面个数，buffer 淘汰信息（本节点除外、DN 上不可使用）。

返回值类型：record

- pv_os_run_info

描述：显示当前操作系统运行的状态信息，具体字段信息参考 GS_OS_RUN_INFO。

参数：nan

返回值类型：setof record

- pv_session_stat

描述：以会话线程或 AutoVacuum 线程为单位，统计会话状态信息，具体字段信息参考 GS_SESSION_STAT。

参数：nan

返回值类型：setof record

- pv_session_time

描述：用于统计会话线程的运行时间信息，及各执行阶段所消耗时间，具体字段信息参考 GS_SESSION_TIME。

参数：nan

返回值类型：setof record

- pg_stat_get_db_temp_bytes

描述：用于统计通过数据库查询写入临时文件的数据总量。计算所有临时文件，不论为什么创建临时文件，而且不管 log_temp_files 设置。

参数：oid

返回值类型：bigint

- pg_stat_get_db_temp_files

描述：通过数据库查询创建的临时文件数量。计算所有临时文件，不论为什么创建临时文件（比如排序或者哈希），而且不管 log_temp_files 设置。

参数：oid

返回值类型：bigint

- remote_candidate_stat()

描述：用于显示数据库所有实例的检查点信息和各类日志刷页情况（本节点除外），集中式不支持。

返回值类型：record

- dbe_perf.gs_stat_activity_timeout(int)

描述：获取当前节点上执行时间超过超时阈值的查询作业信息。需要 GUC 参数 track_activities 设置为 on 才能正确返回结果。超时阈值的取值范围是 0~2147483。

返回值类型：setof record

表 5-64 返回值类型及说明

名称	类型	描述
database	name	用户会话连接的数据库名称。
pid	bigint	后台线程 ID。
sessionid	bigint	会话 ID。
usesysid	oid	登录该后台的用户 OID。
application_name	text	连接到该后台的应用名。
query	text	该后台正在执行的查询。
xact_start	timestamptz	启动当前事务的时间。
query_start	timestamptz	开始当前查询的时间。
query_id	bigint	查询语句 ID。

- gs_wlm_user_resource_info(name text)

描述：查询具体某个用户的资源限额和资源使用情况。普通用户只能查询到自己相关的信息，管理员权限的用户可以查看全部用户的信息。

返回值类型：record

- create_wlm_instance_statistics_info

描述：将当前实例的历史监控数据进行持久化保存。

参数：nan

返回值类型：integer

- gs_session_memory

描述：统计 Session 级别的内存使用情况，包含执行作业在数据节点上 Postgres 线程和 Stream 线程分配的所有内存。

说明

若 GUC 参数 enable_memory_limit=off，该函数不能使用。

返回值类型：record

表 5-65 返回值类型及说明

名称	类型	描述
sessid	text	线程启动时间+线程标识。
init_mem	integer	当前正在执行作业进入执行器前已分配的内存，单位 MB。
used_mem	integer	当前正在执行作业已分配的内存，单位 MB。
peak_mem	integer	当前正在执行作业已分配的内存峰值，单位 MB。

- gs_wlm_persistent_user_resource_info()

描述：将当前所有的用户资源使用统计信息归档到 gs_wlm_user_resource_history 系统表中，只有 sysadmin 有权限查询。

返回值类型：record

- create_wlm_operator_info(int flag)

描述：将当前内存中记录的 TopSQL 算子级别相关统计信息清理，当传入的参数大于 0 时，会将这部分信息归档到 gs_wlm_operator_info 和 gs_wlm_ec_operator_info 中，否则不会

归档。该函数只有 sysadmin 权限的用户可以执行。

返回值类型：int

- GS_ALL_NODEGROUP_CONTROL_GROUP_INFO(text)

描述：提供了所有逻辑数据库实例的控制组信息。该函数在调用的时候需要指定要查询逻辑数据库实例的名称。例如要查询'installation'逻辑数据库实例的控制组信息：

```
SELECT * FROM GS_ALL_NODEGROUP_CONTROL_GROUP_INFO('installation')
```

返回值类型：record

函数返回字段如下：

表 5-66 返回值类型及说明

名称	类型	描述
name	text	控制组的名称。
type	text	控制组的类型。
gid	bigint	控制组 ID。
classgid	bigint	Workload 所属 Class 的控制组 ID。
class	text	Class 控制组。
workload	text	Workload 控制组。
shares	bigint	控制组分配的 CPU 资源配额。

- gs_total_nodegroup_memory_detail

描述：返回当前数据库逻辑数据库使用内存的信息，单位为 MB 得到一个逻辑数据库。

返回值类型：setof record

- local_redo_time_count()

描述：返回本节点各个回放线程的各个流程的耗时统计（仅在备机上有有效数据）。

返回值如下：

local_redo_time_count 返回参数说明。

表 5-67 返回值类型及说明

字段名	描述
thread_name	线程名字
step1_total	step1 的总时间，每个线程对应的流程如下： ● 极致 RTO： ■ batch redo：从队列中获取一条日志 ■ redo manager：从队列中获取一条日志 ■ redo worker：从队列中获取一条日志 ■ trxn manager：从队列中读取一条日志 ■ trxn worker：从队列中读取一条日志 ■ read worker：从文件中读取一次 xlog page（整体） ■ read page worker：从队列中获取一个日志 ■ startup：从队列中获取一个日志 ● 并行回放： ■ page redo：从队列中获取一条日志 ■ startup：读取一条日志
step1_count	step1 的统计次数
step2_total	step2 的总时间，每个线程对应的流程如下： ● 极致 RTO： ■ batch redo：处理日志（整体） ■ redo manager：处理日志（整体） ■ redo worker：处理日志（整体） ■ trxn manager：处理日志（整体） ■ trxn worker：处理日志（整体） ■ redo worker：读取 xlog page 耗时 ■ read page worker：生成和发送 lsn forwarder

	■ startup：check stop(是否回放到指定位置) ● 并行回放： ■ page redo：处理日志（整体） ■ startup：check stop(是否回放到指定位置)
step2_count	step2 的统计次数
step3_total	step3 的总时间，每个线程对应的流程如下： ● 极致 RTO： ■ batch redo：更新 standbystate ■ redo manager：数据日志处理 ■ redo worker：回放 page 也日志（整体） ■ trxn manager：更新 flush lsn ■ trxn worker：回放日志处理 ■ redo worker：推进 xlog segment ■ read page worker：获取一个新的 item ■ startup：redo delay（延迟回放特性等待时间） ● 并行回放： ■ page redo：更新 standbystate ■ startup：redo delay（延迟回放特性等待时间）
step3_count	step3 的统计次数
step4_total	step4 的总时间，每个线程对应的流程如下： ● 极致 RTO： ■ batch redo：解析 xlog ■ redo manager：DDL 处理 ■ redo worker：读取数据 page 页 ■ trxn manager：同步等待时间 ■ trxn worker：更新本线程 lsn ■ read page worker：将日志放入分发线程 ■ startup：分发（整体）

	● 并行回放： ■ page redo：undo 日志回放 ■ startup：分发（整体）
step4_count	step4 的统计次数
step5_total	step5 的总时间，每个线程对应的流程如下： ● 极致 RTO： ■ batch redo：分发给 redo manager ■ redo manager：分发给 redo worker ■ redo worker：回放数据 page 页的日志 ■ trxn manager：分发给 trxn worker ■ trxn worker：强同步 wait 时间 ■ read page worker：更新本线程 lsn ■ startup：日志 decode ● 并行回放： ■ page redo：sharetrxn 日志回放 ■ startup：日志回放
step5_count	step5 的统计次数
step6_total	step6 的总时间，每个线程对应的流程如下： ● 极致 RTO： ■ redo worker：回放非数据页 page 日志 ■ trxn manager：全局 lsn 更新 ■ read page worker：日志 crc 校验 ● 并行回放： ■ page redo：synctrxn 日志回放 ■ startup：强同步等待
step6_count	step6 的统计次数
step7_total	step7 的总时间，每个线程对应的流程如下：

	● 极致 RTO： ■ redo worker：fsm 更新 ● 并行回放： ■ page redo：single 日志回放
step7_count	step7 的统计次数
step8_total	step8 的总时间，每个线程对应的流程如下： ● 极致 RTO： ■ redo worker：强同步等待 ● 并行回放： ■ page redo：all workers do 日志回放
step8_count	step8 的统计次数
step9_total	step9 的总时间，每个线程对应的流程如下： ● 极致 RTO： ■ 无 ● 并行回放： ■ page redo：muliti workers do 日志回放
step9_count	step9 的统计次数

- local_xlog_redo_statics()

  描述：返回本节点已经回放的各个类型类型的日志统计信息（仅在备机上有有效数据）。

  返回值如下：

表 5-68 local_xlog_redo_statics 返回参数说明

字段名	描述
xlog_type	日志类型
rmid	resource manager id

info	xlog operation
num	日志个数
extra	针对 page 回放日志和 xact 日志有有效值。page 页回放日志标识从磁盘读取 page 的个数。xact 日志表示删除文件的个数。

- gs_get_shared_memctx_detail(text)

描述：返回指定内存上下文上的内存申请的详细信息，包含每一处内存申请所在的文件、行号和大小（同一文件同一行大小会做累加）。只支持查询通过 pg_shared_memory_detail 视图查询出来的内存上下文，入参为内存上下文名称（即 pg_shared_memory_detail 返回结果的 contextname 列）。查询该函数必须具有 sysadmin 权限或者 monitor admin 权限。

返回值类型：setof record

表 5-69 参数类型及说明

名称	类型	描述
file	text	申请内存所在文件的文件名。
line	int8	申请内存所在文件的代码行号。
size	int8	申请的内存大小，同一文件同一行多次申请会做累加。

说明

该视图不支持 release 版本小型化场景。

- gs_get_session_memctx_detail(text)

描述：返回指定内存上下文上的内存申请的详细信息，包含每一处内存申请所在的文件、行号和大小（同一文件同一行大小会做累加）。仅在线程池模式下生效。只支持查询通过 pv_session_memory_context 视图查询出来的内存上下文，入参为内存上下文名称（即 pv_session_memory_context 返回结果的 contextname 列）。查询该函数必须具有 sysadmin 权限或者 monitor admin 权限。

返回值类型：setof record

表 5-70 参数类型及说明

名称	类型	描述
file	text	申请内存所在文件的文件名。
line	int8	申请内存所在文件的代码行号。
size	int8	申请的内存大小，同一文件同一行多次申请会做累加。

说明

该视图仅在线程池模式下生效，且该视图不支持 release 版本小型化场景。

- gs_get_thread_memctx_detail(tid,text)

描述：返回指定内存上下文上的内存申请的详细信息，包含每一处内存申请所在的文件，行号和大小（同一文件同一行大小会做累加）。只支持查询通过 pv_thread_memory_context 视图查询出来的内存上下文，第一个入参为线程 id（即 pv_thread_memory_context 返回数据的 tid 列），第二个参数为内存上下文名称（即 pv_thread_memory_context 返回数据的 contextname 列）。查询该函数必须具有 sysadmin 权限或者 monitor admin 权限。

返回值类型：setof record

表 5-71 参数类型及说明

名称	类型	描述
file	text	申请内存所在文件的文件名。
line	int8	申请内存所在文件的代码行号。
size	int8	申请的内存大小，同一文件同一行多次申请会做累加。

说明

该视图不支持 release 版本小型化场景。

5.28 触发器函数

- pg_get_triggerdef(oid)

  描述：获取触发器的定义信息。

  参数：待查触发器的 OID。

  返回值类型：text

  示例:

```
postgres=# select pg_get_triggerdef(oid) from pg_trigger;

pg_get_triggerdef
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------
 CREATE TRIGGER tg1 BEFORE INSERT ON gtest26 FOR EACH STATEMENT EXECUTE PROCEDURE
gtest_trigger_func()
 CREATE TRIGGER tg03 AFTER INSERT ON gtest26 FOR EACH ROW WHEN ((new.a IS NOT
NULL))EXECUTE PROCEDURE gtest_trigger_func()
(2 rows)
```

- pg_get_triggerdef(oid, boolean)

  描述：获取触发器的定义信息。

  参数：待查触发器的 OID 及是否以 pretty 方式展示。

  说明

  仅在创建 trigger 时指定 WHEN 条件的情况下，布尔类型参数才生效。

  返回值类型：text

  示例:

```
postgres=# select pg_get_triggerdef(oid,true) from pg_trigger;

pg_get_triggerdef
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------
 CREATE TRIGGER tg1 BEFORE INSERT ON gtest26 FOR EACH STATEMENT EXECUTE PROCEDURE
gtest_trigger_func()
```

```
 CREATE TRIGGER tg03 AFTER INSERT ON gtest26 FOR EACH ROW WHEN (new.a IS NOT NULL)
EXECUTE PROCEDURE gtest_trigger_func()
(2 rows)
postgres=# select pg_get_triggerdef(oid,false) from pg_trigger;
pg_get_triggerdef
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------
 CREATE TRIGGER tg1 BEFORE INSERT ON gtest26 FOR EACH STATEMENT EXECUTE PROCEDURE
gtest_trigger_func()
 CREATE TRIGGER tg03 AFTER INSERT ON gtest26 FOR EACH ROW WHEN ((new.a IS NOT NULL))
EXECUTE PROCEDURE gtest_trigger_func()
(2 rows)
```

5.29 事件触发器函数

- pg_event_trigger_ddl_commands

描述：在 ddl_command_end 事件触发器中，该函数用于报告运行中的 DDL 命令。 参数：空

说明：

该函数仅能被事件触发器函数使用。

返回值类型：oid,oid,int4,text,text,text,text,bool,pg_ddl_command。

示例：

```
postgres=# CREATE OR REPLACE FUNCTION ddl_command_test()
RETURNS event_trigger
AS $$
DECLARE
obj record;
BEGIN
FOR obj IN SELECT * FROM pg_event_trigger_ddl_commands()
LOOP
RAISE NOTICE 'command: %',
obj.command_tag;

RAISE NOTICE 'triggered';
END LOOP;
END; $$ LANGUAGE plpgsql;
```

- pg_event_trigger_dropped_objects

  描述：在 sql_drop 事件触发器中，让被删除的对象列表对用户可见。

  参数：空

  说明：

  该函数仅能被事件触发器函数使用。

  返回值类型： oid,oid,int4,bool,bool,booloid,text,text,text,text,TEXTARRAY,TEXTARRAY

  示例：

```
postgres=# CREATE OR REPLACE FUNCTION test_evtrig_dropped_objects() RETURNS
event_trigger
LANGUAGE plpgsql AS $$
DECLARE
obj record;
BEGIN
FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
LOOP
IF obj.object_type = 'table' THEN
EXECUTE format('DROP TABLE IF EXISTS audit_tbls.%I',    format('%s_%s',
obj.schema_name, obj.object_name));
END IF;
INSERT INTO dropped_objects
(type, schema, object) VALUES
(obj.object_type, obj.schema_name, obj.object_identity);
END LOOP;
END
$$;
```

- pg_event_trigger_table_ rewrite_oid

  描述：在 table_ rewrite 事件触发器中，让被重写的对象 oid 对用户可见。

  参数：空

  说明：该函数仅能被事件触发器函数使用。

  返回值类型：oid

  示例：

```
postgres=# CREATE OR REPLACE FUNCTION test_evtrig_no_rewrite() RETURNS
event_trigger
```

```
LANGUAGE plpgsql AS $$
BEGIN
RAISE NOTICE 'Table ''%'' is being rewritten (reason = %)',
pg_event_trigger_table_rewrite_oid()::regclass,
pg_event_trigger_table_rewrite_reason();
END;
$$;
```

- pg_event_trigger_table_ rewrite_ reason

描述：在 table_ rewrite 事件触发器中，让被重写的对象的重写原因对用户可见。 参数：空

说明：该函数仅能被事件触发器函数使用。

返回值类型：int4

示例：

```
postgres=# CREATE OR REPLACE FUNCTION test_evtrig_no_rewrite() RETURNS
event_trigger
LANGUAGE plpgsql AS $$
BEGIN
RAISE NOTICE 'Table ''%'' is being rewritten (reason = %)',
pg_event_trigger_table_rewrite_oid()::regclass,
pg_event_trigger_table_rewrite_reason();
END;
$$;
```

5.30 HashFunc 函数

- bucketabstime(value ， flag)

描述：对 abstime 格式的数值 value 计算 hash 值并找到对应的 hashbucket 桶。

参数：value 为需要转换的数值，类型为 abstime ， flag 为 int 类型表示数据分布方式，0 表示 hash 分布。

返回值类型：int32

示例：

```
postgres=# select bucketabstime('2011-10-01 10:10:10.112',1);
bucketabstime
---------------
13954
```

```
(1 row)
```

- bucketbool(value ， flag)

  描述：对 bool 格式的数值 value 计算 hash 值并找到对应的 hashbucket 桶。

  参数：value 为需要转换的数值，类型为 bool ， flag 为 int 类型表示数据分布方式，0 表示 hash 分布。

  返回值类型：int32

  示例：

```
postgres=# select bucketbool(true,1);
bucketbool
------------
1
(1 row)
postgres=#  select bucketbool(false,1);
bucketbool
------------
0
(1 row)
```

- bucketbpchar(value, flag)

  描述：对 bpchar 格式的数值 value 计算 hash 值并找到对应的 hashbucket 桶。

  参数：value 为需要转换的数值，类型为 bpchar ， flag 为 int 类型表示数据分布方式，0 表示 hash 分布。

  返回值类型：int32

  示例：

```
postgres=# select bucketbpchar('test',1);
bucketbpchar
--------------
9761
(1 row)
```

- bucketbytea(value ， flag)

  描述：对 bytea 格式的数值 value 计算 hash 值并找到对应的 hashbucket 桶。

  参数：value 为需要转换的数值，类型为 bytea ， flag 为 int 类型表示数据分布方式，0 表示 hash 分布。

返回值类型：int32

示例：

```
postgres=# select bucketbytea('test',1);
bucketbytea
-------------
9761
(1 row)
```

- bucketcash(value ， flag)

描述：对 money 格式的数值 value 计算 hash 值并找到对应的 hashbucket 桶。

参数：value 为需要转换的数值，类型为 money ， flag 为 int 类型表示数据分布方式，0 表示 hash 分布。

返回值类型：int32

示例：

```
postgres=# select bucketcash(10::money,1);
bucketcash
------------
8468
(1 row)
```

- getbucket(value ， flag)

描述：从分布列获取 hashbucket 桶。

value 为需要输入的数值，类型：

“char”、abstime、 bigint、 boolean、 bytea、character varying、character、date、double precision、int2vector、integer、interval、money、name、numeric、 nvarchar、nvarchar2、oid、oidvector、raw、real、record、reltime、smalldatetime、 smallint、text、time with time zone、time without time zone、timestamp with time zone、timestamp without time zone、tinyint、uuid

flag 表示数据分布方式，类型：integer

返回值类型：integer

示例：

```
postgres=# select getbucket(10,'H');
getbucket
```

```
-----------
14535
(1 row)
postgres=# select getbucket(11,'H');
getbucket
-----------
13449
(1 row)
postgres=# select getbucket(11,'R');
getbucket
-----------
13449
(1 row)
postgres=# select getbucket(12,'R');
getbucket
-----------
9412
(1 row)
```

- hash_array(anyarray)

  描述：数组哈希，将数组的元素通过哈希函数得到结果，并返回合并结果。

  参数：数据类型为 anyarray。

  返回值类型：integer

  示例：

```
postgres=# select hash_array(ARRAY[[1,2,3],[1,2,3]]);
hash_array
------------
-382888479
(1 row)
```

- hash_group(key)

  描述：流引擎中，该函数可将 Group Clause 中的各列计算为一个 hash 值。 参数： key 为 Group Clause 中各列的值。

  返回值类型：32 位 hash 值

  示例：

  按照步骤依次执行。

```
postgres=# CREATE TABLE tt(a int, b int,c int,d int);
CREATE TABLE
postgres=# select * from tt;
a | b | c | d
---+---+---+---
(0 rows)

postgres=# insert into tt values(1,2,3,4);
INSERT 0 1
postgres=# select * from tt;
a | b | c | d
---+---+---+---
1 | 2 | 3 | 4
(1 row)

postgres=# insert into tt values(5,6,7,8);
INSERT 0 1
postgres=# select * from tt;
a | b | c | d
---+---+---+---
1 | 2 | 3 | 4
5 | 6 | 7 | 8
(2 rows)

postgres=# select hash_group(a,b) from tt where a=1 and b=2;
hash_group
------------
990882385
(1 row)
```

- hash_ numeric(numeric)

  描述：计算 Numeric 类型的数据的 hash 值。

  参数：Numeric 类型的数据。

  返回值类型：integer

  示例：

```
postgres=# select hash_numeric(30);
hash_numeric
--------------
-282860963
```

```
(1 row)
```

- hash_ range(anyrange)

  描述：计算 range 的哈希值。

  参数：anyrange 类型的数据。

  返回值类型：integer

  示例：

```
postgres=# select hash_range(numrange(1.1,2.2));
hash_range
------------
683508754
(1 row)
```

- hashbpchar(character)

  描述：计算 bpchar 的哈希值。

  参数：character 类型的数据。

  返回值类型：integer

  示例：

```
postgres=# select hashbpchar('hello');
hashbpchar
-------------
-1870292951
(1 row)
```

- hashchar(char)

  描述：char 和布尔数据转换为哈希值

  参数：char 类型的数据或者 bool 类型的数据。

  返回值类型：integer

  示例：

```
postgres=# select hashbpchar('hello');
hashbpchar
-------------
-1870292951
(1 row)
```

```
postgres=# select hashchar('true');
hashchar
------------
1686226652
(1 row)
```

- hashenum(anyenum)

  描述：枚举类型转哈希值。

  参数：anyenum 类型的数据。

  返回值类型：integer

  示例：

```
postgres=# CREATE TYPE b1 AS ENUM('good', 'bad', 'ugly');
CREATE TYPE
postgres=# call hashenum('good'::b1);
hashenum
------------
1821213359
(1 row)
```

- hashfloat4(real)

  描述：float4 转哈希值。

  参数：real 类型的数据。

  返回值类型：integer

  示例：

```
postgres=# select hashfloat4(12.1234);
hashfloat4
------------
1398514061
(1 row)
```

- hashfloat8(double precision)

  描述：float8 转哈希值。

  参数：double precision 类型的数据。

  返回值类型：integer

  示例：

```
postgres=# select hashfloat8(123456.1234);
hashfloat8
------------
1673665593
(1 row)
```

- hashinet(inet)

  描述：支持 inet / cidr 上的哈希索引的功能。返回传入 inet 的 hash 值。

  参数：inet 类型的数据。

  返回值类型：integer

  示例：

```
postgres=# select hashinet('127.0.0.1'::inet);
hashinet
-------------
-1435793109
(1 row)
```

- hashint1(tinyint)

  描述：INT1 转哈希值。

  参数：tinyint 类型的数据。

  返回值类型：uint32

  示例：

```
postgres=# select hashint1(20);
hashint1
-------------
-2014641093
(1 row)
```

- hashint2(smallint)

  描述：INT2 转哈希值。

  参数：smallint 类型的数据。

  返回值类型：uint32

  示例：

```
postgres=# select hashint2(20000);
```

```
hashint2
------------
-863179081
(1 row)
```

- bucketchar

  描述：计算入参的哈希值。

  参数：char, integer

  返回值类型：integer

- bucketdate

  描述：计算入参的哈希值。

  参数：date, integer

  返回值类型：integer

- bucketfloat4

  描述：计算入参的哈希值。

  参数：real, integer

  返回值类型：integer

- bucketfloat8

  描述：计算入参的哈希值。

  参数：double precision, integer

  返回值类型：integer

- bucketint1

  描述：计算入参的哈希值。

  参数：tinyint, integer

  返回值类型：integer

- bucketint2

  描述：计算入参的哈希值。

  参数：smallint, integer

返回值类型：integer

- bucketint2vector

  描述：计算入参的哈希值。

  参数：int2vector, integer

  返回值类型：integer

- bucketint4

  描述：计算入参的哈希值。

  参数：integer, integer

  返回值类型：integer

- bucketint8

  描述：计算入参的哈希值。

  参数：bigint, integer

  返回值类型：integer

- bucketinterval

  描述：计算入参的哈希值。

  参数：interval, integer

  返回值类型：integer

- bucketname

  描述：计算入参的哈希值。

  参数：name, integer

  返回值类型：integer

- bucketnumeric

  描述：计算入参的哈希值。

  参数：numeric, integer

  返回值类型：integer

- bucketnvarchar2

描述：计算入参的哈希值。

参数：nvarchar/nvarchar2, integer

返回值类型：integer

- bucketoid

  描述：计算入参的哈希值。

  参数：oid, integer

  返回值类型：integer

- bucketoidvector

  描述：计算入参的哈希值。

  参数：oidvector, integer

  返回值类型：integer

- bucketraw

  描述：计算入参的哈希值。

  参数：raw, integer

  返回值类型：integer

- bucketreltime

  描述：计算入参的哈希值。

  参数：reltime, integer

  返回值类型：integer

- bucketsmalldatetime

  描述：计算入参的哈希值。

  参数：smalldatetime, integer

  返回值类型：integer

- buckettext

  描述：计算入参的哈希值。

  参数：text, integer

返回值类型：integer

- buckettime

  描述：计算入参的哈希值。

  参数：time without time zone, integer

  返回值类型：integer

- buckettimestamp

  描述：计算入参的哈希值。

  参数：timestamp without time zone, integer

  返回值类型：integer

- buckettimestamptz

  描述：计算入参的哈希值。

  参数：timestamp with time zone, integer

  返回值类型：integer

- buckettimetz

  描述：计算入参的哈希值。

  参数：time with time zone, integer

  返回值类型：integer

- bucketuuid

  描述：计算入参的哈希值。

  参数：uuid, integer

  返回值类型：integer

- bucketvarchar

  描述：计算入参的哈希值。

  参数：character varying, integer

  返回值类型：integer

5.31 提示信息函数

- report_application_error

  描述：PL 执行过程中，可以使用此函数来抛 ERROR。

  返回值类型：void

表 5-72 report_application_error 参数说明

参数	类型	说明	是否必选
log	text	error 消息的内容。	是
code	int4	error 消息对应的 error code，范围为：-20999 ~ -20000。	否

5.32 全局临时表函数

- pg_get_gtt_relstats(relOid)

  描述：显示当前会话指定的全局临时表的基本信息。

  参数：全局临时表的 OID。

  返回值类型：record

  示例:

```
postgres=# select * from pg_get_gtt_relstats(74069);
relfilenode | relpages | reltuples | relallvisible | relfrozenxid | relminmxid
-------------+----------+-----------+---------------+--------------+------------
---
       74069 |       58 |     13000 |             0 |        11151 |          0
(1 row)
```

- pg_get_gtt_statistics(relOid, attnum, “::text)

  描述：显示当前会话指定的全局临时表的单列统计信息。

  参数：全局临时表的 OID 和属性 attnum。

  返回值类型：record

示例:

```
postgres=# select * from pg_get_gtt_statistics(74069,1,''::text);
 starelid | starelkind | staattnum | stainherit | stanullfrac | stawidth |
stadistinct | stakind1 | stakind2 | stakind3 | stakind4 | stakind5 | staop1 | staop2
| staop3 | staop4 | staop5 | stanumbers1 | stanumbers2 | stanumbers3 | stanu
mbers4 | stanumbers5 |
                                        stavalues1
                                                                                |
stavalues2 | stavalues3 | stavalues4 | stavalues5 | stadndistinct | staextinfo
---------+-----------+----------+-----------+------------+---------+---
---------+---------+---------+---------+---------+---------+--------+--
------+--------+--------+--------+-------------+-------------+-------------+-
-----
-------+------------+-----------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
-----
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
-----
-----------------------------------------------------------------------------+----
--------+-----------+-----------+-----------+-------------+-----------
    74069 | c         |         1 | f         |           0 |         4 |    -1
|         2 |         3 |         0 |         0 |         0 |      97 |      97 | 0 |
0 |       0 |             | {1}         |             |
       |             |
{1, 130, 260, 390, 520, 650, 780, 910, 1040, 1170, 1300, 1430, 1560, 1690, 1820, 1950, 2080, 2
210, 2340, 2470, 2600, 2730, 2860, 2990, 3120, 3250, 3380, 3510, 3640, 3770, 3900, 4030, 416
0, 4290, 4420, 4550, 4680, 4810, 4940, 5070, 5200, 5330, 5460, 5590, 57
20, 5850, 5980, 6110, 6240, 6370, 6500, 6630, 6760, 6890, 7020, 7150, 7280, 7410, 7540, 7670,
7800, 7930, 8060, 8190, 8320, 8450, 8580, 8710, 8840, 8970, 9100, 9230, 9360, 9490, 9620, 97
50, 9880, 10010, 10140, 10270, 10400, 10530, 10660, 10790, 10920, 11050, 11180, 11310, 114
40, 1
1570, 11700, 11830, 11960, 12090, 12220, 12350, 12480, 12610, 12740, 12870, 13000} |    |
|             |             |             0 |
(1 row)
```

- pg_gtt_attached_pid(relOid)

  描述：显示正在使用指定全局临时表的所有线程 pid。

参数：全局临时表的 OID。

返回值类型：record

示例:

```
postgres=# select * from pg_gtt_attached_pid(74069);
relid |       pid
-------+-----------------
74069 | 139648170456832
74069 | 139648123270912
(2 rows)
```

- dbe_perf.get_global_full_sql_by_timestamp(start_timestamp timestamp, end_timestamp timestamp)

  描述：获取实例级的全量 SQL(Full SQL)信息。

  返回值类型：record

表 5-73 dbe_perf.get_global_full_sql_by_timestamp 参数说明

参数	类型	描述
start_timestamp	timestamp	SQL 启动时间范围的开始时间点。
end_timestamp	timestamp	SQL 启动时间范围的结束时间点。

- dbe_perf.get_global_slow_sql_by_timestamp(start_timestamp timestamp, end_timestamp timestamp)

  描述：获取实例级的慢 SQL(Slow SQL)信息。

  返回值类型：record

表 5-74 dbe_perf.get_global_slow_sql_by_timestamp 参数说明

参数	类型	描述
start_timestamp	timestamp	SQL 启动时间范围的开始时间点。

end_timestamp	timestamp	SQL 启动时间范围的结束时间点。

- statement_detail_decode(detail text, format text, pretty bool)

  解析全量/慢 SQL 语句中的 details 字段的信息。

表 5-75 statement_detail_decode 参数说明

参数	类型	描述
detail	text	SQL 语句产生的事件的集合（不可读）。
format	text	解析输出格式，取值为 plaintext 或 json。
pretty	bool	当 format 为 plaintext 时，是否以优雅的格式展示： ● true 表示通过“\n”分隔事件。 ● false 表示通过“，”分隔事件。

- pg_list_gtt_relfrozenxids()

  描述：显示各会话的冻结事务 xid。

  pid=0 的行，显示所有会话中最老的冻结事务 xid。

  参数：无。

  返回值类型：record

  示例:

```
postgres=# select * from pg_list_gtt_relfrozenxids();
      pid       | relfrozenxid
----------------+--------------
 139648123270912 |        11151
 139648170456832 |        11155
               0 |        11151
(3 rows)
```

5.33 故障注入系统函数

- gs_fault_inject(int64, text, text, text, text, text)

描述：该函数不能调用，调用时会报 WARNING 信息："unsupported fault injection"，并不会对数据库产生任何影响和改变。

参数：int64 注入故障类型（0：CLOG 扩展页面，1：读取 CLOG 页面，2：强制死锁）。

  - text 第二个入参在第一入参为 2 的模式下若为"1"则死锁，其余不死锁；

第二个入参在第一入参为 0，1 时，表示 CLOG 开始扩展或读取的起始页面号。

  - text 第三个入参在第一入参为 0，1 时，表示扩展或读取的页面个数。
  - text 第四到六入参为预留参数。

返回值类型：int64

5.34 AI 特性函数

- gs_index_advise(text)

描述：针对单条查询语句推荐索引。

参数：SQL 语句字符串

返回值类型：record

示例：

```
postgres=# select "table", "column" from gs_index_advise('SELECT c_discount from bmsql_customer where c_w_id = 10');
     table      |  column
----------------+----------
 bmsql_customer | c_w_id
(1 row)
```

上述结果表明：应当在 bmsql_customer 的 c_w_id 列上创建索引。例如可以通过下述 SQL 语句创建索引：

```
postgres=# CREATE INDEX idx on bmsql_customer(c_w_id);
```

某些 SQL 语句，也可能被推荐创建联合索引，例如：

```
postgres=# select "table", "column" from gs_index_advise('select name, age, sex from t1 where age >= 18 and age < 35 and sex = ''f'';');
 table | column
```

```
-------+-----------
 t1    | age, sex
(1 row)
```

则上述语句表明应该在表 t1 上创建一个联合索引（age, sex)。可以通过下述命令创建：

```
postgres=# CREATE INDEX idx1 on t1(age, sex);
```

针对分区表可推荐具体索引类型，例如：

```
postgres=# select "table", "column", "indextype" from gs_index_advise('select
name, age, sex from range_table where age = 20;');
 table | column | indextype
-------+--------+-----------
 t1    | age    | global
(1 row)
```

- hypopg_create_index(text)

  描述：创建虚拟索引。

  参数：创建索引语句的字符串

  返回值类型：record

  示例：

```
postgres=# select * from hypopg_create_index('create index on
bmsql_customer(c_w_id)');
 indexrelid |              indexname
------------+-------------------------------------
     329726 | <329726>btree_bmsql_customer_c_w_id
(1 row)
```

- hypopg_display_index()

  描述：显示所有创建的虚拟索引信息。

  参数：无

  返回值类型：record

  示例：

```
postgres=# select * from hypopg_display_index();
                indexname                  | indexrelid |     table      |
column
-------------------------------------------+------------+----------------+--
---------------
```

```
 <329726>btree_bmsql_customer_c_w_id         |     329726 | bmsql_customer |
(c_w_id)
 <329729>btree_bmsql_customer_c_d_id_c_w_id |     329729 | bmsql_customer |
(c_d_id, c_w_id)
(2 rows)
```

- hypopg_drop_index(oid)

  描述：删除指定的虚拟索引。

  参数：索引的 oid

  返回值类型：bool

  示例：

```
postgres=# select * from hypopg_drop_index(329726);
 hypopg_drop_index
-------------------
 t
(1 row)
```

- hypopg_reset_index()

  描述：清除所有虚拟索引。

  参数：无

  返回值类型：无

  示例：

```
postgres=# select * from hypopg_reset_index();
 hypopg_reset_index
--------------------


(1 row)
```

- hypopg_estimate_size(oid)

  描述：估计指定索引创建所需的空间大小。

  参数：索引的 oid

  返回值类型：int8

  示例：

```
postgres=# select * from hypopg_estimate_size(329730);
 hypopg_estimate_size
----------------------
             15687680
(1 row)
```

- check_engine_status(ip text, port text)

  描述：测试给定的 ip 和 port 上是否有 predictor engine 提供服务。

  参数：predictor engine 的 ip 地址和端口号。

  返回值类型：text

- encode_plan_node(optname text, orientation text, strategy text, options text, dop int8, quals text, projection text)

  描述：对入参的计划算子信息进行编码。

  参数：计划算子信息。

  返回值类型：text。

  说明

  该函数为内部功能调用函数，不建议用户直接使用。

- model_train_opt(template text, model text)

  描述：训练给定的查询性能预测模型。

  参数：性能预测模型的模板名和模型名。

  返回值类型：tartup_time_accuracy FLOAT8、total_time_accuracy FLOAT8、rows_accuracy FLOAT8、peak_memory_accuracy FLOAT8

- track_model_train_opt(ip text, port text)

  描述：返回给定 ip 和 port predictor engine 的训练日志地址。

  参数：predictor engine 的 ip 地址和端口号。

  返回值类型：text

- encode_feature_perf_hist(datname text)

  描述：将目标数据库已收集的历史计划算子进行编码。

参数：数据库名。

返回值类型：queryid bigint、 plan_node_id int、 parent_node_id int、 left_child_id int、right_child_id int, encode text 、 startup_time bigint 、 total_time bigint 、 rows bigint 、peak_memory int

- gather_encoding_info(datname text)

描述：调用 encode_feature_perf_hist，将编码好的数据进行持久化保存。

参数：数据库名。

返回值类型：int

- db4ai_predict_by_bool (text, VARIADIC “any”)

描述：获取返回值为布尔型的模型进行模型推断任务。此函数为内部调用函数，建议直接使用语法 PREDICT BY 进行推断任务。

参数：模型名称和推断任务的输入列。

返回值类型：bool

- db4ai_predict_by_float4(text, VARIADIC “any”)

描述：获取返回值为 float4 的模型进行模型推断任务。此函数为内部调用函数，建议直接使用语法 PREDICT BY 进行推断任务。

参数：模型名称和推断任务的输入列。

返回值类型：float

- db4ai_predict_by_float8(text, VARIADIC “any”)

描述：获取返回值为 float8 的模型进行模型推断任务。此函数为内部调用函数，建议直接使用语法 PREDICT BY 进行推断任务。

参数：模型名称和推断任务的输入列。

返回值类型：float

- db4ai_predict_by_int32(text, VARIADIC “any”)

描述：获取返回值为 int32 的模型进行模型推断任务。此函数为内部调用函数，建议直接使用语法 PREDICT BY 进行推断任务。

参数：模型名称和推断任务的输入列。

返回值类型：int

- db4ai_predict_by_int64(text, VARIADIC “any”)

描述：获取返回值为 int64 的模型进行模型推断任务。此函数为内部调用函数，建议直接使用语法 PREDICT BY 进行推断任务。

参数：模型名称和推断任务的输入列。

返回值类型：int

- db4ai_predict_by_numeric(text, VARIADIC “any”)

描述：获取返回值为 numeric 的模型进行模型推断任务。此函数为内部调用函数，建议直接使用语法 PREDICT BY 进行推断任务。

参数：模型名称和推断任务的输入列。

返回值类型：numeric

- db4ai_predict_by_text(text, VARIADIC “any”)

描述：获取返回值为字符型的模型进行模型推断任务。此函数为内部调用函数，建议直接使用语法 PREDICT BY 进行推断任务。

参数：模型名称和推断任务的输入列。

返回值类型：text

- db4ai_predict_by_float8_array(text, VARIADIC “any”)

描述：获取返回值为字符型的模型进行模型推断任务。此函数为内部调用函数，建议直接使用语法 PREDICT BY 进行推断任务。

参数：模型名称和推断任务的输入列。

返回值类型：text

- gs_explain_model(text)

描述：获取返回值为字符型的模型进行模型解析文本化任务。

参数：模型名称。

返回值类型：text

5.35 动态数据脱敏函数

说明

该函数为内部功能调用函数。

- creditcardmasking(col text, letter char default 'x')

  描述：将 col 字符串后四位之前的数字使用 letter 替换。

  参数：待替换的字符串、替换字符。

  返回值类型：text

- basicmailmasking(col text, letter char default 'x')

  描述：将 col 字符串中第一个'@'之前的字符使用 letter 替换。

  参数：待替换的字符串、替换字符。

  返回值类型：text

- fullmailmasking(col text, letter char default 'x')

  描述：将 col 字符串中出现最后一个'.'之前的字符(除'@'外)使用 letter 替换。

  参数：待替换的字符串、替换字符。

  返回值类型：text

- alldigitsmasking(col text, letter char default '0')

  描述：将 col 字符串中出现的数字使用 letter 替换。

  参数：待替换的字符串、替换字符。

  返回值类型：text

- shufflemasking(col text)

  描述：将 col 字符串中的字符乱序排列。

  参数：待替换的字符串、替换字符。

  返回值类型：text

- randgbaseasking(col text)

描述：将 col 字符串中的字符随机化。

参数：待替换的字符串、替换字符。

返回值类型：text

- regexpmasking

描述：脱敏策略的内部函数，对字符进行正则表达式替换。

参数：col text, reg text, replace_text text, pos INTEGER default 0, reg_len INTEGER default -1

返回值类型：text

5.36 其他系统函数

GBase 8s 的内建函数和操作符继承自开源 PG，下述函数不作赘述，详情请参见 PG 官方文档。

表 5-76 其他系统函数

_pg_char_max_length	_pg_char_octet_length	_pg_datetime_precision	_pg_expandarray	_pg_index_position	_pg_interval_type	_pg_numeric_precision
_pg_numeric_precision_radix	_pg_numeric_scale	_pg_truetypid	_pg_truetypmod	abbrev	abs	abstime
abstimeeq	abstimege	abstimegt	abstimein	abstimele	abstimelt	abstimene
abstimeout	abstimerecv	abstimesend	aclcontains	acldefault	aclexplode	aclinsert
aclitemeq	aclitemin	aclitemout	aclremove	acos	age	akeys
any_in	any_out	anyarray_in	anyarray_out	anyarray_recv	anyarray_send	anyelement_in

anyelement_out	anyenum_in	anyenum_out	anynonarray_in	anynonarray_out	anyrange_in	anyrange_out
anytextcat	area	areajoinsel	areasel	array_agg	array_agg_finalfn	array_agg_transfn
array_append	array_cat	array_dims	array_eq	array_fill	array_ge	array_gt
array_in	array_larger	array_le	array_length	array_lower	array_lt	array_ndims
array_ne	array_out	array_prepend	array_recv	array_send	array_smaller	array_to_json
array_to_string	array_typanalyze	array_upper	arraycontained	arraycontains	arraycontjoinsel	arraycontsel
arrayoverlap	ascii	asin	atan	atan2	avals	avg
big5_to_euc_tw	big5_to_mic	big5_to_utf8	bit	bit_and	bit_in	bit_length
bit_or	bit_out	bit_recv	bit_send	bitand	bitcat	bitcmp
biteq	bitge	bitgt	bitle	bitlt	bitne	bitnot
bitor	bitshiftleft	bitshiftright	bittypmodin	bittypmodout	bitxor	bool
bool_and	bool_or	booland_statefunc	booleq	boolge	boolgt	boolin

boolle	boollt	boolne	boolor_statefunc	boolout	boolrecv	boolsend
box	box_above	box_above_eq	box_add	box_below	box_below_eq	box_center
box_contain	box_contain_pt	box_contained	box_distance	box_div	box_eq	box_ge
box_gt	box_in	box_intersect	box_le	box_left	box_lt	box_mul
box_out	box_overabove	box_overbelow	box_overlap	box_overleft	box_overright	box_recv
box_right	box_same	box_send	box_sub	bpchar	bpchar_larger	bpchar_pattern_ge
bpchar_pattern_gt	bpchar_pattern_le	bpchar_pattern_lt	bpchar_smaller	bpchar_sortsupport	bpcharcmp	bpchareq
bpcharge	bpchargt	bpcharliclike	bpcharicnlike	bpcharicregexeq	bpcharicregexne	bpcharin
bpcharle	bpcharlike	bpcharlt	bpcharne	bpcharnlike	bpcharout	bpcharrecv
bpcharregexeq	bpcharregexne	bpcharsend	bpchartypmodin	bpchartypmodout	broadcast	btabstimecmp
btarraycmp	btbeginscan	btboolcmp	btbpchar_pattern_cmp	btbuild	btbuildempty	btbulkdelete

btcanreturn	btcharcmp	btcostestimate	btendscan	btfloat48smp	btfloat4cmp	btfloat4sortsupport
btfloat84cmp	btfloat8smp	btfloat8sortsupport	btgetbitmap	btgettuple	btinsert	btint24cmp
btint28smp	btint2cmp	btint2sortsupport	btint42cmp	btint48smp	btint4cmp	btint4sortsupport
btint82cmp	btint84cmp	btint8smp	btint8sortsupport	btmarkpos	btnamecmp	btnamesortsupport
btoidcmp	btoidsortsupport	btoidvectorcmp	btoptions	btrecordcmp	btreltimecmp	btrescan
btrestrpos	btrim	bttext_pattern_cmp	bttextcmp	bttextsortsupport	bttidcmp	bttintervalcmp
btvacuumcleanup	bytea_sortsupport	bytea_string_agg_finalfn	bytea_string_agg_transfn	byteacat	byteacmp	byteaeq
byteage	byteagt	byteain	byteale	bytealike	bytealt	byteane
byteanlike	byteaout	bytearecv	byteasend	cash_cmp	cash_div_cash	cash_div_flt4
cash_div_flt8	cash_div_int2	cash_div_int4	cash_div_int8	cash_eq	cash_ge	cash_gt
cash_in	cash_le	cash_lt	cash_mi	cash_mul_flt4	cash_mul_flt8	cash_mul_int2

cash_mul_int4	cash_mul_int8	cash_ne	cash_out	cash_pl	cash_recv	cash_send
cashlarger	cashsmaller	cbrt	ceil	ceiling	center	char
char_length	character_length	chareq	charge	chargt	charin	charle
charlt	charne	charout	charrecv	charsend	chr	cideq
cidin	cidout	cidr	cidr_in	cidr_out	cidr_recv	cidr_send
cidrecv	cidsend	circle	circle_above	circle_add_pt	circle_below	circle_center
circle_contain	circle_contain_pt	circle_contained	circle_distance	circle_div_pt	circle_eq	circle_ge
circle_gt	circle_in	circle_le	circle_left	circle_lt	circle_mul_pt	circle_ne
circle_out	circle_overabove	circle_overbelow	circle_overlap	circle_overleft	circle_overright	circle_recv
circle_right	circle_same	circle_send	circle_sub_pt	clock_timestamp	close_lb	close_ls
close_lseg	close_pb	close_pl	close_ps	close_sb	close_sl	col_Description
concat	concat_ws	contjoinsel	contsel	convert	convert_from	convert_to
corr	cos	cot	count	covar_po	covar_sam	cstring_in

				p	p	
cstring_out	cstring_recv	cstring_send	cume_dist	current_database	current_query	current_schema
xpath_exists	current_setting	current_user	currtid	currtid2	currval	cursor_to_xml
cursor_to_xmlschema	database_to_xml	database_to_xml_and_xmlschema	database_to_xmlschema	date	date_cmp	date_cmp_timestamp
date_cmp_timestamptz	date_eq	date_eq_timestamp	date_eq_timestamptz	date_ge	date_ge_timestamp	date_ge_timestamptz
date_gt	date_gt_timestamp	date_gt_timestamptz	date_in	date_larger	date_le	date_le_timestamp
date_le_timestamptz	date_lt	date_lt_timestamp	date_lt_timestamptz	date_mi	date_mi_interval	date_mii
date_ne	date_ne_timestamp	date_ne_timestamptz	date_out	date_pl_interval	date_pli	date_recv
date_send	date_smaller	date_sortsupport	daterange_canonical	daterange_subdiff	datetime_pl	datetimetz_pl
dcbrt	decode	defined	degrees	delete	dense_rank	dexp
diagonal	diameter	dispell_init	dispell_lexize	dist_cpoly	dist_lb	dist_pb

dist_pc	dist_pl	dist_ppath	dist_ps	dist_sb	dist_sl	div
dlog1	dlog10	domain_in	domain_recv	dpow	dround	dsimple_init
dsimple_lexize	dsnowball_init	dsnowball_lexize	dsqrt	dsynonym_init	dsynonym_lexize	dtrunc
each	enum_ne	enum_out	enum_range	enum_recv	enum_send	enum_smaller
eqjoinsel	eqsel	euc_cn_to_mic	euc_cn_to_utf8	euc_jis_2004_to_shift_jis_2004	euc_jis_2004_to_utf8	euc_jp_to_mic
euc_jp_to_sjis	euc_jp_to_utf8	euc_kr_to_mic	euc_kr_to_utf8	euc_tw_to_big5	euc_tw_to_mic	euc_tw_to_utf8
every	exist	exists_all	exists_any	exp	factorial	family
fdw_handler_in	fdw_handler_out	fetchval	first_value	float4	float4_accum	float48div
float48eq	float48ge	float48gt	float48le	float48lt	float48mi	float48mul
float48ne	float48pl	float4abs	float4div	float4eq	float4ge	float4gt
float4in	float4larger	float4le	float4lt	float4mi	float4mul	float4ne
float4out	float4pl	float4recv	float4send	float4smaller	float4um	float4up
float8	float8_accum	float8_avg	float8_coll	float8_cor	float8_cova	float8_cova

			ect	r	r_pop	r_samp
float8_regr_accum	float8_regr_avgx	float8_regr_avgy	float8_regr_collect	float8_regr_intercept	float8_regr_r2	float8_regr_slope
float8_regr_sxx	float8_regr_sxy	float8_regr_syy	float8_stddev_pop	float8_stddev_samp	float8_var_pop	float8_var_samp
float84div	float84eq	float84ge	float84gt	float84le	float84lt	float84mi
float84mul	float84ne	float84pl	float8abs	float8div	float8eq	float8ge
float8gt	float8in	float8larger	float8le	float8lt	float8mi	float8mul
float8ne	float8out	float8pl	float8recv	float8send	float8smaller	float8um
float8up	floor	flt4_mul_cash	flt8_mul_cash	fmgr_c_validator	fmgr_internal_validator	fmgr_sql_validator
format	format_type	gb18030_to_utf8	gbk_to_utf8	generate_series	generate_subscripts	get_bit
get_byte	get_current_ts_config	-	-	gin_clean_pending_list	gin_cmp_prefix	gin_cmp_tslexeme
gin_extract_tsquery	gin_extract_tsvector	gin_tsquery_consistent	gin_tsquery_triconsistent	ginarrayconsistent	ginarrayextract	ginarraytriconsistent
ginbeginscan	ginbuild	ginbuildempty	ginbulkdelete	gincostestimate	ginendscan	gingetbitmap

gininsert	ginmarkpos	ginoptions	ginqueryarrayextract	ginrescan	ginrestrpos	ginvacuumcleanup
gist_box_compress	gist_box_consistent	gist_box_decompress	gist_box_penalty	gist_box_picksplit	gist_box_same	gist_box_union
gist_circle_compress	gist_circle_consistent	gist_point_compress	gist_point_consistent	gist_point_distance	gist_poly_compress	gist_poly_consistent
gistbeginscan	gistbuild	gistbuildempty	gistbulkdelete	gistcostestimate	gistendscan	gistgetbitmap
gistgettuple	gistinsert	gistmarkpos	gistoptions	gistrescan	gistrestrpos	gistvacuumcleanup
gtsquery_compress	gtsquery_consistent	gtsquery_decompress	gtsquery_penalty	gtsquery_picksplit	gtsquery_same	gtsquery_union
gtsvector_compress	gtsvector_consistent	gtsvector_decompress	gtsvector_penalty	gtsvector_picksplit	gtsvector_same	gtsvector_union
gtsvectorin	gtsvectorout	has_tablespace_privilege	has_type_privilege	hash_aclitem	hashbeginscan	hashbuild
hashbuildempty	hashbulkdelete	hashcostestimate	hashendscan	hashgetbitmap	hashgettuple	hashinsert
hashint2vector	hashint4	hashint8	hashmacaddr	hashmarkpos	hashname	hashoid
hashoidve	hashoptions	hashrescan	hashrestrp	hashtext	hashvacuu	hashvarlena

ctor			os		mcleanup	
host	hostmask	iclikejoinsel	iclikesel	icnlikejoinsel	icnlikesel	icregexeqjoinsel
icregexeqsel	icregexnejoinsel	icregexnesel	inet_client_addr	inet_client_port	inet_in	inet_out
inet_recv	inet_send	inet_server_addr	inet_server_port	inetand	inetmi	inetmi_int8
inetnot	inetor	inetpl	initcap	int2_accum	int2_avg_accum	int2_mul_cash
int2_sum	int24div	int24eq	int24ge	int24gt	int24le	int24lt
int24mi	int24mul	int24ne	int24pl	int28div	int28eq	int28ge
int28gt	int28le	int28lt	int28mi	int28mul	int28ne	int28pl
int2abs	int2and	int2div	int2eq	int2ge	int2gt	int2in
int2larger	int2le	int2lt	int2mi	int2mod	int2mul	int2ne
int2not	int2or	int2out	int2pl	int2recv	int2send	int2shl
int2shr	int2smaller	int2um	int2up	int2vectoreq	int2vectorin	int2vectorout
int2vectorrecv	int2vectorsend	int2xor	int4_accum	int4_avg_accum	int4_mul_cash	int4_sum
int42div	int42eq	int42ge	int42gt	int42le	int42lt	int42mi

int42mul	int42ne	int42pl	int48div	int48eq	int48ge	int48gt
int48le	int48lt	int48mi	int48mul	int48ne	int48pl	int4abs
int4and	int4div	int4eq	int4ge	int4gt	int4in	int4inc
int4larger	int4le	int4lt	int4mi	int4mod	int4mul	int4ne
int4not	int4or	int4out	int4pl	int4range	int4range_canonical	int4range_subdiff
int4recv	int4send	int4shl	int4shr	int4smaller	int4um	int4up
int4xor	int8	int8_avg	int8_avg_accum	int8_avg_collect	int8_mul_cash	int8_sum
int8_sum_to_int8	int8_accum	int82div	int82eq	int82ge	int82gt	int82le
int82lt	int82mi	int82mul	int82ne	int82pl	int84div	int84eq
int84ge	int84gt	int84le	int84lt	int84mi	int84mul	int84ne
int84pl	int8abs	int8and	int8div	int8eq	int8ge	int8gt
int8in	int8inc	int8inc_any	int8inc_float8_float8	int8larger	int8le	int8lt
int8mi	int8mod	int8mul	int8ne	int8not	int8or	int8out
int8pl	int8pl_inet	int8range	int8range_canonical	int8range_subdiff	int8recv	int8send

int8shl	int8shr	int8smaller	int8um	int8up	int8xor	integer_pl_date
inter_lb	inter_sb	inter_sl	internal_in	internal_out	interval	interval_accum
interval_avg	interval_cmp	interval_collect	interval_div	interval_eq	interval_ge	interval_gt
interval_hash	interval_in	interval_larger	interval_le	interval_lt	interval_mi	interval_mul
interval_ne	interval_out	interval_pl	interval_pl_date	interval_pl_time	interval_pl_timestamp	interval_pl_timestamptz
interval_pl_timetz	interval_recv	interval_send	interval_smaller	interval_transform	interval_um	intervaltypmodin
intervaltypmodout	intinterval	isexists	ishorizontal	iso_to_koi8r	iso_to_mic	iso_to_win1251
iso_to_win866	iso8859_1_to_utf8	iso8859_to_utf8	isparallel	isperp	isvertical	johab_to_utf8
jsonb_in	jsonb_out	jsonb_recv	jsonb_send	-	-	-
json_in	json_out	json_recv	json_send	justify_days	justify_hours	justify_interval
koi8r_to_iso	koi8r_to_mic	koi8r_to_utf8	koi8r_to_win1251	koi8r_to_win866	koi8u_to_utf8	language_handler_in

language_handler_out	latin1_to_mic	latin2_to_mic	latin2_to_win1250	latin3_to_mic	latin4_to_mic	like_escape
likejoinsel	likesel	line	line_distance	line_eq	line_horizontal	line_in
line_interpt	line_intersect	line_out	line_parallel	line_perp	line_recv	line_send
line_vertical	ln	lo_close	lo_creat	lo_create	lo_export	lo_import
lo_lseek	lo_open	lo_tell	lo_truncate	lo_unlink	log	loread
lower	lower_inc	lower_inf	lowrite	lpad	lseg	lseg_center
lseg_distance	lseg_eq	lseg_ge	lseg_gt	lseg_horizontal	lseg_in	lseg_interpt
lseg_intersect	lseg_le	lseg_length	lseg_lt	lseg_ne	lseg_out	lseg_parallel
lseg_perp	lseg_recv	lseg_send	lseg_vertical	ltrim	macaddr_and	macaddr_cmp
macaddr_eq	macaddr_ge	macaddr_gt	macaddr_in	macaddr_le	macaddr_lt	macaddr_ne
macaddr_not	macaddr_or	macaddr_out	macaddr_recv	macaddr_send	makeaclitem	masklen

max	md5	mic_to_big5	mic_to_euc_cn	mic_to_euc_jp	mic_to_euc_kr	mic_to_euc_tw
mic_to_iso	mic_to_koi8r	mic_to_latin1	mic_to_latin2	mic_to_latin3	mic_to_latin4	mic_to_sjis
mic_to_win1250	mic_to_win1251	mic_to_win866	min	mktinterval	money	mul_d_interval
name	nameeq	namege	namegt	nameiclike	nameicnlike	nameicregexeq
nameicregexne	namein	namele	namelike	namelt	namene	namenlike
nameout	namerecv	nameregexeq	nameregexne	namesend	neqjoinsel	neqsel
network_cmp	network_eq	network_ge	network_gt	network_le	network_lt	network_ne
network_sub	network_subeq	network_sup	network_supeq	nlikejoinsel	nlikesel	numeric
numeric_abs	numeric_accum	numeric_add	numeric_avg	numeric_avg_accum	numeric_avg_collect	numeric_cmp
numeric_collect	numeric_div	numeric_div_trunc	numeric_eq	numeric_exp	numeric_fac	numeric_ge
numeric_gt	numeric_in	numeric_inc	numeric_larger	numeric_le	numeric_ln	numeric_log

numeric_lt	numeric_mod	numeric_mul	numeric_ne	numeric_out	numeric_power	numeric_recv
numeric_send	numeric_smaller	numeric_sortsupport	numeric_sqrt	numeric_stddev_pop	numeric_stddev_samp	numeric_sub
numeric_transform	numeric_uminus	numeric_uplus	numeric_var_pop	numeric_var_samp	numerictypmodin	numerictypmodout
numrange_subdiff	oid	oideq	oidge	oidgt	oidin	oidlarger
oidle	oidlt	oidne	oidout	oidrecv	oidsend	oidsmaller
oidvectoreq	oidvectorge	oidvectorgt	oidvectorin	oidvectorle	oidvectorlt	oidvectorne
oidvectorout	oidvectorrecv	oidvectorsend	oidvectortypes	on_pb	on_pl	on_ppath
on_ps	on_sb	on_sl	opaque_in	opaque_out	ordered_set_transition	overlaps
overlay	path	path_add	path_add_pt	path_center	path_contain_pt	path_distance
path_div_pt	path_in	path_inter	path_length	path_mul_pt	path_n_eq	path_n_ge
path_n_gt	path_n_le	path_n_lt	path_npoints	path_out	path_recv	path_send

path_sub_pt	percentile_cont	percentile_cont_float8_final	percentile_cont_interval_final	pg_char_to_encoding	pg_cursor	pg_encoding_max_length
pg_encoding_to_char	pg_Extension_config_dump	-	-	pg_node_tree_in	pg_node_tree_out	pg_node_tree_recv
pg_node_tree_send	pg_prepared_statement	pg_prepared_xact	-	-	pg_show_all_settings	pg_stat_get_bgwriter_stat_reset_time
pg_stat_get_buf_fsync_backend	pg_stat_get_checkpoint_sync_time	pg_stat_get_checkpoint_write_time	pg_stat_get_db_blk_read_time	pg_stat_get_db_blk_write_time	pg_stat_get_db_conflict_all	pg_stat_get_db_conflict_bufferpin
pg_stat_get_db_conflict_snapshot	pg_stat_get_db_conflict_startup_deadlock	pg_switch_xlog	xpath	pg_timezone_abbrevs	pg_timezone_names	pg_stat_get_wal_receiver
plpgsql_call_handler	plpgsql_inline_handler	plpgsql_validator	point_above	point_add	point_below	point_distance
point_div	point_eq	point_horiz	point_in	point_left	point_mul	point_ne
point_out	point_recv	point_right	point_send	point_sub	point_vert	poly_above
poly_below	poly_center	poly_contain	poly_contain_pt	poly_contained	poly_distance	poly_in

poly_left	poly_npoints	poly_out	poly_over above	poly_over below	poly_overlap	poly_overleft
poly_overright	poly_recv	poly_right	poly_same	poly_send	polygon	position
positionjoinsel	positionsel	postgresql_fdw_validator	pow	power	prsd_end	prsd_headline
prsd_lexytype	prsd_nexttoken	prsd_start	pt_contained_circle	pt_contained_poly	query_to_xml	query_to_xml_and_xmlschema
query_to_xmlschema	quote_ident	quote_literal	quote_nullable	radians	radius	random
range_adjacent	range_after	range_before	range_cmp	range_contained_by	range_contains	range_contains_elem
range_eq	range_ge	range_gist_compress	range_gist_consistent	range_gist_decompress	range_gist_penalty	range_gist_picksplit
range_gist_same	range_gist_union	range_gt	range_in	range_intersect	range_le	range_lt
range_minus	range_ne	range_out	range_overlaps	range_overleft	range_overright	range_recv
range_send	range_typanalyze	range_union	rank	record_eq	record_ge	record_gt

record_in	record_le	record_lt	record_ne	record_out	record_recv	record_send
regclass	regclassin	regclassout	regclassrecv	regclasssend	regconfigin	regconfigout
regconfigrecv	regconfigsend	regdictionaryin	regdictionaryout	regdictionaryrecv	regdictionarysend	regexeqjoinsel
regexeqsel	regexnejoinsel	regexnesel	regexp_matches	regexp_replace	regexp_split_to_array	regexp_split_to_table
regoperatorin	regoperatorout	regoperatorrecv	regoperatorsend	regoperin	regoperout	regoperrecv
regopersend	regprocedurein	regprocedureout	regprocedurerecv	regproceduresend	regprocin	regprocout
regprocrecv	regprocsend	regr_avgx	regr_avgy	regr_count	regr_intercept	regr_r2
regr_slope	regr_sxx	regr_sxy	regr_syy	regtypein	regtypeout	regtyperecv
regtypesend	reltime	reltimeeq	reltimege	reltimegt	reltimein	reltimele
reltimelt	reltimene	reltimeout	reltimerecv	reltimesend	repeat	replace
reverse	RI_FKey_cascade_del	RI_FKey_cascade_upd	RI_FKey_check_ins	RI_FKey_check_upd	RI_FKey_noaction_del	RI_FKey_noaction_upd

RI_FKey_restrict_del	RI_FKey_restrict_upd	RI_FKey_setdefault_del	RI_FKey_setdefault_upd	RI_FKey_setnull_del	RI_FKey_setnull_upd	right
round	row_number	row_to_json	rpad	rtrim	scalargtjoinsel	scalargtsel
scalarltjoinsel	scalarltsel	schema_to_xml	schema_to_xml_and_xmlschema	schema_to_xmlschema	session_user	set_bit
set_byte	set_config	set_maskleln	shift_jis_2004_to_euc_jis_2004	shift_jis_2004_to_utf8	sjis_to_euc_jp	sjis_to_mic
sjis_to_utf8	smgrin	smgrout	spg_kd_choose	spg_kd_config	spg_kd_inner_consistent	spg_kd_picksplit
spg_quad_choose	spg_quad_config	spg_quad_inner_consistent	spg_quad_leaf_consistent	spg_quad_picksplit	spg_text_choose	spg_text_config
spg_text_inner_consistent	spg_text_leaf_consistent	spg_text_picksplit	spgbeginscan	spgbuild	spgbuildempty	spgbulkdelete
spgcanreturn	spgcostestimate	spgendscan	spggetbitmap	spggettuple	spginsert	spgmarkpos
spgoptions	spgrescan	spgrestrpos	spgvacuu	stddev	stddev_pop	stddev_sam

			mcleanup			p
string_agg	string_agg_finalfn	string_agg_transfn	strip	sum	suppress_redundant_updates_trigger	table_to_xml
table_to_xml_and_xmlschema	table_to_xmlschema	tan	text	text_ge	text_gt	text_larger
text_le	text_lt	text_pattern_ge	text_pattern_gt	text_pattern_le	text_pattern_lt	text_smaller
textanycat	textcat	texteq	texticlike	texticnlike	texticregexeq	texticregexne
textin	textlike	textne	textnlike	textout	textrecv	textregexeq
textregexne	textsend	thesaurus_init	thesaurus_lexize	tideq	tidge	tidgt
tidin	tidlarger	tidle	tidlt	tidne	tidout	tidrecv
tidsend	tidsmaller	time	time_cmp	time_eq	time_ge	time_gt
time_hash	time_in	time_larger	time_le	time_lt	time_mi_interval	time_mi_time
time_ne	time_out	time_pl_interval	time_recv	time_send	time_smaller	time_transform
timedate_pl	timemi	timepl	timestamp	timestamp_cmp	timestamp_cmp_date	timestamp_cmp_timest

						amptz
timestamp_eq	timestamp_eq_date	timestamp_eq_timestamptz	timestamp_ge	timestamp_ge_date	timestamp_ge_timestamptz	timestamp_gt
timestamp_gt_date	timestamp_gt_timestamptz	timestamp_hash	timestamp_in	timestamp_larger	timestamp_le	timestamp_le_date
timestamp_le_timestamptz	timestamp_lt	timestamp_lt_date	timestamp_lt_timestamptz	timestamp_mi	timestamp_mi_interval	timestamp_ne
timestamp_ne_date	timestamp_ne_timestamptz	timestamp_out	timestamp_pl_interval	timestamp_recv	timestamp_send	timestamp_smaller
timestamp_sortsupport	timestamp_transform	timestamptypmodin	timestamptypmodout	timestamptz	timestamptz_cmp	timestamptz_cmp_date
timestamptz_cmp_timestamp	timestamptz_eq	timestamptz_eq_date	timestamptz_eq_timestamp	timestamptz_ge	timestamptz_ge_date	timestamptz_ge_timestamp
timestamptz_gt	timestamptz_gt_date	timestamptz_gt_timestamp	timestamptz_in	timestamptz_larger	timestamptz_le	timestamptz_le_date
timestamptz_le_timestamp	timestamptz_lt	timestamptz_lt_date	timestamptz_lt_timestamp	timestamptz_mi	timestamptz_mi_interval	timestamptz_ne
timestampt	timestamptz_	timestampt	timestamp	timestamp	timestampt	timestamptz

z_ne_date	ne_timestamp	z_out	tz_pl_interval	tz_recv	z_send	_smaller
timestamptztypmodin	timestamptztypmodout	timetypmodin	timetypmodout	timetz	timetz_cmp	timetz_eq
timetz_ge	timetz_gt	timetz_hash	timetz_in	timetz_larger	timetz_le	timetz_lt
timetz_mi_interval	timetz_ne	timetz_out	timetz_pl_interval	timetz_recv	timetz_send	timetz_smaller
timetzdate_pl	timetztypmodin	timetztypmodout	timezone（2069）	timezone（1159）	timezone（2037）	timezone (2070)
timezone (1026)	timezone (2038)	tintervalct	tintervaleq	tintervalge	tintervalgt	tintervalin
tintervalle	tintervalleneq	tintervallenge	tintervallengt	tintervallenle	tintervallenlt	tintervallenne
tintervallt	tintervalne	tintervalout	tintervalov	tintervalrecv	tintervalsame	tintervalsend
tintervalstart	to_ascii（1845）	to_ascii（1847）	to_ascii（1846）	trigger_in	trigger_out	ts_match_qv
ts_match_tq	ts_match_tt	ts_match_vq	ts_rank	ts_rank_cd	ts_rewrite	ts_stat
ts_token_type	ts_typanalyze	tsmatchjoinsel	tsmatchsel	tsq_mcontained	tsq_mcontains	tsquery_and
tsquery_c	tsquery_eq	tsquery_ge	tsquery_gt	tsquery_le	tsquery_lt	tsquery_ne

mp						
tsquery_not	tsquery_or	tsqueryin	tsqueryout	tsqueryrecv	tsquerysend	tsrange
tsrange_subdiff	tstzrange	tstzrange_subdiff	tsvector_cmp	tsvector_concat	tsvector_eq	tsvector_ge
tsvector_gt	tsvector_le	tsvector_lt	tsvector_ne	tsvector_update_trigger	tsvector_update_trigger_column	tsvectorin
tsvectorout	tsvectorrecv	tsvectorsend	txid_current	txid_current_snapshot	txid_snapshot_in	txid_snapshot_out
txid_snapshot_recv	txid_snapshot_send	txid_snapshot_xip	txid_snapshot_xmax	txid_snapshot_xmin	txid_visible_in_snapshot	uhc_to_utf8
unique_key_recheck	unknownin	unknownout	unknownrecv	unknownsend	unnest	utf8_to_big5
utf8_to_euc_cn	utf8_to_euc_jis_2004	utf8_to_euc_jp	utf8_to_euc_kr	utf8_to_euc_tw	utf8_to_gb18030	utf8_to_gbk
utf8_to_iso8859	utf8_to_iso8859_1	utf8_to_johab	utf8_to_koi8r	utf8_to_koi8u	utf8_to_shift_jis_2004	utf8_to_sjis
utf8_to_uhc	utf8_to_win	uuid_cmp	uuid_eq	uuid_ge	uuid_gt	uuid_hash
uuid_in	uuid_le	uuid_lt	uuid_ne	uuid_out	uuid_recv	uuid_send

var_pop	var_samp	varbit	varbit_in	varbit_out	varbit_recv	varbit_send
varbit_transform	varbitcmp	varbiteq	varbitge	varbitgt	varbitle	varbitlt
varbitne	varbittypmodin	varbittypmodout	varchar	varchar_transform	varcharin	varcharout
varcharrecv	varcharsend	varchartypmodin	varchartypmodout	variance	void_in	void_out
void_recv	void_send	win_to_utf8	win1250_to_latin2	win1250_to_mic	win1251_to_iso	win1251_to_koi8r
win1251_to_mic	win1251_to_win866	win866_to_iso	win866_to_koi8r	win866_to_mic	win866_to_win1251	xideq
xideqint4	xidin	xidout	xidrecv	xidsend	xml	xml_in
xml_is_well_formed	xml_is_well_formed_content	xml_is_well_formed_document	xml_out	xml_recv	xml_send	xmlagg
xmlcomment	xmlconcat2	xmlexists	xmlvalidate	pg_notify	-	-

下述列表为 GBase 8s 实现系统内部功能所使用的函数，不推荐使用，若需使用，请联系 GBase 8s 技术支持工程师。

- pv_compute_ pool_workload()

描述：提供云上加速集群当前负载信息。

返回值类型： record

- locktag_decode(locktag text)

描述：从 locktag 中解析锁的具体信息。

返回值类型： text

- smgreq(a smgr, b smgr)

描述：比较两个 smgr 是否一样。

参数： smgr、 smgr

返回值类型： boolean

- smgrne(a smgr, b smgr)

描述：判断两个 smgr 是否不一样。

参数： smgr、 smgr

返回值类型： boolean

xidin4

描述：输入 4 字节的 xid。

参数： cstring

返回值类型： xid32

- set_hashbucket_info

描述：设置哈希桶信息。

参数： text

返回值类型： boolean

- hs_concat

描述：拼接两个 hstore 类型数据。

参数： hstore、 hstore

返回值类型： hstore

- hs_contained

描述：判断两个 hstore 类型数据是否包含，返回值布尔类型。

参数： hstore、 hstore

返回值类型： boolean

- hs_contains

描述：判断两个 hstore 类型数据是否包含，返回值布尔类型。

参数： hstore、 hstore

返回值类型： boolean

- hstore

描述：将参数转为 hstore 类型。

参数： text、 text

返回值类型： hstore

hstore_in

描述：以 string 格式接收 hstore 数据。

参数： cstring

返回值类型： hstore

- hstore_out

描述：以 string 格式发送 hstore 数据。

参数： hstore

返回值类型： cstring

- hstore_send

描述：以 bytea 格式发送 hstore 数据。

参数： hstore

返回值类型： bytea

- hstore_to_array

描述：以 text 数组格式发送 hstore 数据。

参数： hstore

返回值类型： text[]

- hstore_to_ matrix

描述：以 text 数组格式发送 hstore 数据。

参数： hstore

返回值类型： text[]

- hstore_version_diag

描述：以 integer 数组格式发送 hstore 数据。

参数： hstore

返回值类型： integer

- int1send

描述：将无符号一字节整数打包放入内部数据缓冲流。

参数： tinyint

返回值类型： bytea

- isdefined

描述：判断指定 key 是否存在。

参数： hstore、 text

返回值类型： boolean

- listagg

描述： list 类型 agg 聚集函数。

参数： smallint、 text

返回值类型： text

- log_fdw_validator

描述：验证函数。

参数： text[]、 oid

返回值类型： void

- nvarchar2typmodin

描述：获取 varchar 的 typmod 信息。

参数： cstring[]

返回值类型： integer

- nvarchar2typmodout

描述：获取 varchar 的 typmod 信息， 并构造字符串返回。

参数： integer

返回值类型： cstring

- read_disable_conn_file

描述：读取禁止的连接文件。

参数： nan

返回值类型：disconn_ mode text、 disconn_host text、 disconn_ port text、local_host text、 local_ port text、 redo_finished text

- regex_like_ m

描述：正则匹配，判断字符串是否符合给定的正则表达式。

参数： text、 text

返回值类型： boolean

- update_ pgjob

描述：更新 job。

参数：bigint、 "char"、 bigint、 timestamp without time zone、 timestamp without time zone 、 timestamp without time zone 、 timestamp without time zone 、 timestamp without time zone、 smallint

返回值类型： void

- enum_cmp

描述：枚举类比较函数，用于判断两个枚举类是否相等，以及相对大小。

参数： anyenum、 anyenum

返回值类型： integer

- enum_eq

描述：枚举类比较函数，用于实现=符号。

参数： anyenum、 anyenum

enum_first

描述：返回枚举类中的第一个元素。

参数： anyenum

返回值类型： anyenum

- enum_ge

描述：枚举类比较函数，用于实现>=符号。

参数： anyenum、 anyenum

返回值类型： boolean

- enum_gt

描述：枚举类比较函数，用于实现>符号。

参数： anyenum、 anyenum

返回值类型： boolean

- enum_in

描述：枚举类比较函数，用于判断元素是否在枚举类中。

参数： cstring、 oid

返回值类型： anyenum

- enum_larger

描述：枚举类比较函数，用于实现>符号。

参数： anyenum、 anyenum

返回值类型： anyenum

- enum_last

描述：返回枚举类中的最后一个元素。

参数： anyenum

返回值类型： anyenum

enum_le

描述：枚举类比较函数，用于实现<=符号。

参数： anyenum、 anyenum

返回值类型： boolean

- enum_lt

描述：枚举类比较函数，用于实现<符号。

参数： anyenum、 anyenum

返回值类型： boolean

- enum_smaller

描述：枚举类比较函数，用于实现<符号。

参数： anyenum、 anyenum

返回值类型： boolean

- node_oid_ name

描述：不支持。

参数： oid

返回值类型： cstring

- pg_buffercache_ pages

描述：读取共享缓冲区的状态数据。

参数： nan

返回值类型： setof record

返回字段说明如下：

名称	类型	描述
bufferid	integer	缓冲区的内部 ID。
relfilenode	oid	缓冲区中页面所属关系的 OID。
bucketid	integer	缓冲区所处的哈希桶号。
storage_type	bigint	缓冲区中数据的存储类型。
reltablespace	oid	缓冲区中数据所处的表空间的 OID。
reldatabase	oid	缓冲区中数据所处的数据库的 OID。
relforknumber	integer	缓冲区中数据所在的关系文件分支类型。
relblocknumber	integer	缓冲区中数据在其所属关系文件中的文件块号。
isdirty	boolean	缓冲区是否为脏。

isvalid	boolean	缓冲区是否有效。
usage_count	smallint	缓冲区的使用计数。
pinning_backends	integer	正在使用缓冲区的后端数。

- pg_check_xidlimit

  描述：判断 nextxid 是否>= xidwarnlimit。

  参数： nan

  返回值类型： boolean

- pg_comm_delay

  描述：展示单个 DN 的通信库时延状态。

  参数： nan

  返回值类型： text、 text、 integer、 integer、 integer、 integer

- pg_comm_ recv_stream

  描述：展示单个 DN 上所有的通信库接收流状态。

  参数： nan

  返回值类型： text、 bigint、 text、 bigint、 integer、 integer、 integer、 text、 bigint、 integer、 integer、 integer、 bigint、 bigint、 bigint、 bigint、 bigint

- pg_comm_send_stream

  描述：展示单个 DN 上所有的通信库发送流状态。

  参数： nan

  返回值类型： text、 bigint、 text、 bigint、 integer、 integer、 integer、 text、 bigint、 integer、 integer、 integer、 bigint、 bigint、 bigint、 bigint、 bigint

- pg_comm_status

  描述：展示单个 DN 的通信状态。

  参数： nan

  返回值类型： text、 integer、 integer、 bigint、 bigint、 bigint、 bigint、 bigint、 integer、 integer、 integer、 integer、 integer

- pg_log_comm_status

  描述：在 DN 上打印一些 log。

  参数： nan

  返回值类型： boolean

- pg_ parse_clog

  描述：解析 clog 获取 xid 的 status。

  参数： nan

  返回值类型： xid xid、 status text

- pg_ pool_ ping

  描述：设置 PoolerPing。

  参数： boolean

  返回值类型： SETOF boolean

- pg_ resume_bkp_flag

  描述：用于备份恢复获取 delay xlong 标志。

  参数： slot_ name name

  返回值类型： start_backup_flag boolean、 to_delay boolean、ddl_delay_ recycle_ ptr text、rewind_time text

- pgfadvise_DONTNEED

  描述：这个函数为当前关系设置 DONTNEED 标记。这意味着如果需要释放一些内存，操作系统会首先卸载该文件的页。主要思想是卸载不再使用的文件(而不是可能会被使用的页面)。

  参数：表名称或者索引名称，支持分区表和二级分区表，不支持列存表和段页式表。

- pgfadvise_WILLNEED

描述：这个函数为当前关系设置 WILLNEED 标记。这意味着操作系统将尝试加载该关系尽可能多的页面。主要思想是在服务器启动时预加载文件，预加载文件时可能使用缓存命中率/失误率以及最可能被使用的关系/索引等信息。

参数：表名称或者索引名称，支持分区表和二级分区表，不支持列存表和段页式表

- pgfadvise_NORMAL

描述：这个函数为当前关系设置 NORMAL 标记。

- pgfadvise_SEQUENTIAL

描述：这个函数为当前关系设置 SEQUENTIAL 标记。

- pgfadvise_RANDOM

描述：这个函数为当前关系设置 RANDOM 标记。

- pgfadvise_loader

描述：这个函数允许直接与页面缓存交互。它可以用于根据表示要加载/卸载的页面映射 的 varbit 从内存加载和/或卸载页面。

参数：

第一个参数：表名称或者索引名称，支持分区表、二级分区表，不支持列存表和段页 式表。

第二个参数： forkname，每个关系的数据都存储在一个所谓的 fork 中：通常情况下，

forkname 的默认值是 main，该参数可以省略。

第三个参数：关系的类型，需要传入一个 char 字符，如果是普通的关系，该参数 为'r'；如果是分区表，该参数是'p'；如果是二级分区表，该参数是's'。

第四个参数：如果是分区表，该参数为分区的名称；如果是二级分区表，该参数是二

级分区的名称；如果是普通表，填写 NULL，如果传入其他值，对于普通表的查询没 有影响。

第五个参数：段号。

第六个参数：布尔值，是否进行 load 操作。

第七个参数：布尔值，是否进行 unload 操作。

第八个参数： databit，该参数一般通过 pgfincore()获得。

示例：该示例使用的关系名称为 pgbench_accounts，段号为 0 以及任意的 varbit 映射。

- pgfincore

描述：这个函数提供关于文件系统缓存(页面缓存)的信息。

参数：

第一个参数：表名称或者索引名称，支持分区表和二级分区表，不支持列存表和段页 式表。

第二个参数： forkname，每个关系的数据都存储在一个所谓的 fork 中：通常情况下，forkname 的默认值是 main，该参数可以省略。

第三个参数：布尔值，true 即需要返回 databit, false 不需要返回 databit；该参数可以省略，省略不返回 databit；对于整形或者浮点数，零相当于 false，非零相当于 true. (布尔值不建议传入整形或者浮点型)。

对于一个指定的关系，该函数返回包括以下字段：

- relpath ：该关系的路径。
- segment ：被分析的段号。
- os_page_size ：一个页面的大小。
- rel_os_ pages ：该关系的总页面数。
- pages_ mem ：关系在页面缓存中的页面总数。 (不是来自 PostgreSQL 的共享缓冲区，而是操作系统缓存)。
- group_ mem ：相邻 pages_ mem 的组数。
- os_ page_free ：操作系统页面缓存中空闲的页面数。
- databit ：该文件的 varbit 映射，因为该字段的大小关系，若需要输出该字段，需要使用 pgfincore('pgbench_accounts' ，true)来激活它。如果对于一个没有插入数据的表，使用 true 激活后，该字段不回显数值。
- pages_dirty ：如果定义了 HAVE_FINCORE 常量，平台将提供相关的信息，和 pages_ mem 类似，只不过是对于脏页面的。
- group_dirty ：如果定义了 HAVE_FINCORE 常量，平台将提供相关的信息，和

group_ mem 类似，只不过是对于脏页面的。

- pgsysconf

描述：这个函数输出操作系统块的大小，操作系统页面缓冲区中空闲页面的数量。 示例:

```
cedric=# select * from pgsysconf();
os_page_size | os_pages_free | os_total_pages
-------------+--------------+---------------
4096 |         80431 |        4094174
```

- pgsysconf_ pretty

描述：该函数的功能同上，不同之处在于该函数进行了单位转换，便于阅读。

示例：

```
cedric=# select * from pgsysconf_pretty();
os_page_size | os_pages_free | os_total_pages
-------------+--------------+---------------
4096 bytes   | 314 MB        | 16 GB
```

- pgfincore_drawer

描述：一个非常简单的渲染器。这个函数需要一个 varbit 类型的参数。通常，这个参数的值来源于 pgfincore 函数的 databit 返回字段。 databit 字段的值由 0 和 1 组成，如果是 0，意味着该页不在操作系统页面缓存中，如果是 1，则意味着该页在操作系统页面缓存中。

示例:

```
cedric=# select * from pgfincore_drawer(B'000111');
drawer
--------------

...
cedric=# select * from pgfincore_drawer(B'111000');
drawer
--------------
...
```

- psortoptions

描述：返回 psort 属性。

参数： text[]、 boolean

返回值类型： bytea

- xideq4

  描述：对比两个 xid 类型的值是否相等。

  参数： xid32、 xid32

  返回值类型： boolean

- xideqint8

  描述：对比 xid 类型和 int8 类型的值是否相等。

  参数： xid、 bigint

  返回值类型： boolean

- xidlt

  描述：返回 xid1 < xid2 是否成立。

  参数： xid、 xid

  返回值类型： boolean

- xidlt4

  描述：返回 xid1 < xid2 是否成立。

  参数： xid32、 xid32

  返回值类型： boolean

5.37 内部函数

GBase 8s 中还支持以下使用内部数据类型的函数，用户无法直接调用。

5.37.1 选择率计算函数

areajoinsel	areasel	arraycontjoinsel	arraycontsel	contjoinsel	contsel	eqjoinsel
eqsel	iclikejoinsel	iclikesel	icnlikejoinsel	icnlikesel	icregexeqjoinsel	icregexeqsel
icregexne	icregexn	likejoinse	likesel	neqjoins	neqsel	nlikejoi

joinsel	esel	l		el		nsel
nlikesel	positionjoinsel	positionsel	regexeqjoinsel	regexeqsel	regexnejoinsel	regexnesel
scalargtjoinsel	scalargtsel	scalarltjoinsel	scalarltsel	tsmatchjoinsel	tsmatchsel	——

5.37.2 统计信息收集函数

array_typanalyze

range_typanalyze

ts_typanalyze

local_rto_stat

5.37.3 排序内部功能函数

bpchar_sortsupport

bytea_sortsupport

date_sortsupport

numeric_sortsupport

timestamp_sortsupport

5.37.4 全文检索内部功能函数

dispell_init	dispell_lexize	dsimple_init	dsimple_lexize	dsnowball_init	dsnowball_lexize	dsynonym_init
dsynonym_lexize	gtsquery_compress	gtsquery_consistent	gtsquery_decompress	gtsquery_penalty	gtsquery_picksplit	gtsquery_same
gtsquery	ngram_e	ngram_le	ngram_star	pound_e	pound_le	pound

_union	nd	xtype	t	nd	xtype	_start
prsd_end	prsd_headline	prsd_lextype	prsd_start	thesaurus_init	thesaurus_lexize	zhprs_end
zhprs_getlexeme	zhprs_lextype	zhprs_start	-	-	-	-

5.37.5 内部类型处理函数

abstimerecv	euc_jis_2004_to_utf8	int2recv	line_recv	oidvectorrecv_extend	tidrecv	utf8_to_koi8u
anyarray_recv	euc_jp_to_mic	int2vectorrecv	lseg_recv	path_recv	time_recv	utf8_to_shift_jis_2004
array_recv	euc_jp_to_sjis	int4recv	macaddr_recv	pg_node_tree_recv	time_transform	utf8_to_sjis
ascii_to_mic	euc_jp_to_utf8	int8recv	mic_to_ascii	point_recv	timestamp_recv	utf8_to_uhc
ascii_to_utf8	euc_kr_to_mic	internal_out	mic_to_big5	poly_recv	timestamp_transform	utf8_to_win
big5_to_euc_tw	euc_kr_to_utf8	interval_recv	mic_to_euc_cn	pound_nexttoken	timestamptz_recv	uuid_recv
big5_to_mic	euc_tw	interval	mic_t	prsd_nextto	timetz_r	varbit_r

	_to_big5	_transform	o_euc_jp	ken	ecv	ecv
big5_to_utf8	euc_tw_to_mic	iso_to_koi8r	mic_to_euc_kr	range_recv	tintervalrecv	varbit_transform
bit_recv	euc_tw_to_utf8	iso_to_mic	mic_to_euc_tw	rawrecv	tsqueryrecv	varchar_transform
boolrecv	float4recv	iso_to_win1251	mic_to_iso	record_recv	tsvectorrecv	varcharrecv
box_recv	float8recv	iso_to_win866	mic_to_koi8r	regclassrecv	txid_snapshot_recv	void_recv
bpcharrecv	gb18030_to_utf8	iso8859_1_to_utf8	mic_to_latin1	regconfigrecv	uhc_to_utf8	win_to_utf8
btoidsortsupport	gbk_to_utf8	iso8859_to_utf8	mic_to_latin2	regdictionaryrecv	unknownrecv	win1250_to_latin2
bytearecv	gin_extract_tsvector	johab_to_utf8	mic_to_latin3	regoperatorrecv	utf8_to_ascii	win1250_to_mic
byteawithoutorderwithequalcolrecv	gtsvector_compress	json_recv	mic_to_latin4	regoperrecv	utf8_to_big5	win1251_to_iso
cash_recv	gtsvector_cons	koi8r_t	mic_t	regprocedur	utf8_to_	win1251_to_ko

	istent	o_iso	o_sjis	erecv	euc_cn	i8r
charrecv	gtsvector_decompress	koi8r_to_mic	mic_to_win1250	regprocrecv	utf8_to_euc_jis_2004	win1251_to_mic
cidr_recv	gtsvector_penalty	koi8r_to_utf8	mic_to_win1251	regtyperecv	utf8_to_euc_jp	win1251_to_win866
cidrecv	gtsvector_picksplit	koi8r_to_win1251	mic_to_win866	reltimerecv	utf8_to_euc_kr	win866_to_iso
circle_recv	gtsvector_same	koi8r_to_win866	namerecv	shift_jis_2004_to_euc_jis_2004	utf8_to_euc_tw	win866_to_koi8r
cstring_recv	gtsvector_union	koi8u_to_utf8	ngram_nexttoken	shift_jis_2004_to_utf8	utf8_to_gb18030	win866_to_mic
date_recv	hll_recv	latin1_to_mic	numeric_recv	sjis_to_euc_jp	utf8_to_gbk	win866_to_win1251
domain_recv	hll_trans_recv	latin2_to_mic	numeric_transform	sjis_to_mic	utf8_to_iso8859	xidrecv
euc_cn_to_mic	hstore_recv	latin2_to_win1250	nvarchar2recv	sjis_to_utf8	utf8_to_iso8859_1	xidrecv4
euc_cn_to_ut	inet_re	latin3_t	oidrec	smalldateti	utf8_to_j	xml_rec

f8	cv	o_mic	v	me_recv	ohab	v
euc_jis_2004_to_shift_jis_2004	int1recv	latin4_to_mic	oidvectorrecv	textrecv	utf8_to_koi8r	cstore_tid_out
i16toi1	int16	int16_bool	int16eq	int16div	int16ge	int16gt
int16in	int16le	int16lt	int16mi	int16mul	int16ne	int16out
int16pl	int16recv	int16send	numeric_bool	int2vectorin_extend	int2vectorout_extend	int2vectorrecv_extend
int2vectorsend_extend	jsonb_in	jsonb_out	jsonb_recv	jsonb_send	complex_array_in	bool_int1
bool_int2	bool_int8	bpchar_float4	bpchar_float8	bpchar_int4	bpchar_int8	bpchar_numeric
bpchar_timestamp	f4toi1	f8toi1	float4_bpchar	float4_text	float4_varchar	float8_bpchar
float8_text	float8_varchar	i1tof4	i1tof8	i1toi2	i1toi4	i1toi8
i2toi1	i4toi1	i8toi1	int1_avg_accum	int1_bool	int1_bpchar	int1_numeric
int1_nvarchar2	int1_text	int1_varchar	int1abs	int1and	int1cmp	int1div

int1eq	int1ge	int1gt	int1in	int1inc	int1larger	int1le
int1lt	int1mi	int1mod	int1mul	int1ne	int1not	int1or
int1out	int1pl	int1shl	int1shr	int1smaller	int1um	int1up
int1xor	int2_bool	int2_bpchar	int2_text	int2_varchar	int4_bpchar	int4_text
int4_varchar	int8_bool	int8_bpchar	int8_text	int8_varchar	job_submit	numeric_bpchar
numeric_int1	numeric_text	numeric_varchar	nvarchar2in	nvarchar2out	nvarchar2send	oidvectorin_extend
oidvectorout_extend	oidvectorsend_extend	rawcmp	raweq	rawge	rawgt	rawin
rawle	rawlike	rawlt	rawne	rawnlike	rawout	rawsend
text_float4	text_float8	text_int1	text_int2	text_int4	text_int8	text_numeric
timestamp_text	timestamp_varchar	varchar_float4	varchar_float8	varchar_int4	varchar_int8	varchar_numeric
xidout4	xidsend4	calculate_quantile_o	calculate_value_at	large_seq_rollback_ntree	large_seq_upgrade_ntree	-

		f				

5.37.6 聚合操作内部函数

rray_agg_finalfn	array_agg_transfn	bytea_string_agg_finalfn	bytea_string_agg_transfn	date_list_agg_noarg2_transfn	date_list_agg_transfn	float4_list_agg_noarg2_transfn
float4_list_agg_transfn	float8_list_agg_noarg2_transfn	float8_list_agg_transfn	int2_list_agg_noarg2_transfn	int2_list_agg_transfn	int4_list_agg_noarg2_transfn	int4_list_agg_transfn
int8_list_agg_noarg2_transfn	int8_list_agg_transfn	interval_list_agg_noarg2_transfn	interval_list_agg_transfn	list_agg_finalfn	list_agg_noarg2_transfn	list_agg_transfn
median_float8_finalfn	median_interval_finalfn	median_transfn	mode_final	numeric_list_agg_noarg2_transfn	numeric_list_agg_transfn	ordered_set_transition
percentile_cont_float8_final	percentile_cont_interval_final	string_agg_finalfn	string_agg_transfn	timestamp_list_agg_noarg2_transfn	timestamp_list_agg_transfn	timestamptz_list_agg_noarg2_transfn
timestamptz_list_agg_transfn	checksumtext_agg_transfn	-	-	-	-	-
json_agg_finalf	json_agg	json_object_agg_	json_object_ag	-	-	-

n	_transfn	finalfn	g_transfn			

5.37.7 哈希内部功能函数

hashbeginscan	hashbuild	hashbuildempty	hashbulkdelete	hashcostestimate	hashendscan	hashgetbitmap
hashgettuple	hashinsert	hashmarkpos	hashmerge	hashrescan	hashrestrpos	hashvacuumcleanup
hashvarlena	-	-	-	-	-	-

5.37.8 Btree 索引内部功能函数

cbtreebuild	cbtreecanreturn	cbtreecostestimate	cbtreegetbitmap	cbtreegettuple	btbeginscan	btbuild
btbuildempty	btbulkdelete	btcanreturn	btcostestimate	btendscan	btfloat4sortsupport	btfloat8sortsupport
btgetbitmap	btgettuple	btinsert	btint2sortsupport	btint4sortsupport	btint8sortsupport	btmarkpos
btmerge	btnamesortsupport	btrescan	btrestrpos	bttextsortsupport	btvacuumcleanup	cbtreeoptions

5.37.9 GiST 索引内部功能函数

gist_box_compress	gist_box_consistent	gist_box_decompress	gist_box_penalty	gist_box_picksplit	gist_box_same	gist_box_union
gist_circle	gist_circl	gist_poi	gist_point	gist_point	gist_po	gist_pol

_compress	e_consistent	nt_compress	_consistent	_distance	ly_compress	y_consistent
gistbeginscan	gistbuild	gistbuildempty	gistbulkdelete	gistcostestimate	gistendscan	gistgetbitmap
gistinsert	gistmarkpos	gistmerge	gistrescan	gistrestrpos	gistvacuumcleanup	range_gist_compress
range_gist_decompress	range_gist_penalty	range_gist_picksplit	range_gist_same	range_gist_union	spg_kd_choose	spg_kd_config
spg_kd_picksplit	spg_quad_choose	spg_quad_config	spg_quad_inner_consistent	spg_quad_leaf_consistent	spg_quad_picksplit	spg_text_choose
spg_text_inner_consistent	spg_text_leaf_consistent	spg_text_picksplit	spgbeginscan	spgbuild	spgbuildempty	spgbulkdelete
spgcostestimate	spgendscan	spggetbitmap	spggettuple	spginsert	spgmarkpos	spgmerge
spgrestrpos	spgvacuumcleanup	-	-	-	-	-

5.37.10Gin 索引内部功能函数

in_cmp_prefix	in_cmp_prefix	in_cmp_prefix	in_cmp_prefix	in_cmp_prefix	in_cmp_prefix	in_cmp_prefix
gin_extract_tsquery	gin_extract_tsquery	gin_extract_tsquery	gin_extract_tsquery	gin_extract_tsquery	gin_extract_tsquery	gin_extract_tsquery

gin_tsquery_consistent	gin_tsquery_consistent	gin_tsquery_consistent	gin_tsquery_consistent	gin_tsquery_consistent	gin_tsquery_consistent	gin_tsquery_consistent
gin_tsquery_triconsistent	gin_tsquery_triconsistent	gin_tsquery_triconsistent	gin_tsquery_triconsistent	gin_tsquery_triconsistent	gin_tsquery_triconsistent	gin_tsquery_triconsistent
ginarrayconsistent	ginarrayconsistent	ginarrayconsistent	ginarrayconsistent	ginarrayconsistent	ginarrayconsistent	ginarrayconsistent

5.37.11 Psort 索引内部函数

psortbuild

psortcanreturn

psortcostestimate

psortgetbitmap

psortgettuple

5.37.12 Ubtree 索引内部函数

btbeginscan	btbeginscan	btbeginscan	btbeginscan	btbeginscan
ubtbuild	ubtbuild	ubtbuild	ubtbuild	ubtbuild
ubtbuildempty	ubtbuildempty	ubtbuildempty	ubtbuildempty	ubtbuildempty
ubtbulkdelete	ubtbulkdelete	ubtbulkdelete	ubtbulkdelete	ubtbulkdelete

5.37.13 plpgsql 内部函数

plpgsql_inline_handler

5.37.14集合相关内部函数

array_index by_delete	array_index by_length	array_integer _deleteidx	array_integ er_exists	array_inte ger_first	array_inte ger_last
array_integ er_next	array_integ er_prior	array_varcha r_deleteidx	array_varc har_exists	array_var char_first	array_var char_last
array_varch ar_next	array_varch ar_prior	-	-	-	-

5.37.15外表相关内部函数

dist_fdw_handler

roach_handler

streaming_fdw_handler

dist_fdw_validator

file_fdw_handler

file_fdw_validator

log_fdw_handler

5.37.16主数据库节点远程读取备数据库节点数据页辅助函数

gs_read_block_from_remote 用于读取非段页式表文件的页面。默认只有初始化用户可以查看，其余用户需要赋权后才可以使用。

pg_read_binary_file_blocks 用于读取数据，非压缩表返回实际数据，压缩表返回压缩后的数据。默认只有初始化用户可以查看，其余用户需要赋权后才可以使用。

gs_read_segment_block_from_remote 用于读取段页式表文件的页面。默认只有初始化用户可以查看，其余用户需要赋权后才可以使用。

5.37.17主数据库节点远程读取备数据库节点数据文件辅助函数

gs_read_file_from_remote 用于读取指定的文件。gs_repair_file 利用 gs_read_file_size_from_remote 函数获取文件大小后，依赖这个函数将远端文件逐段读取。默

认只有初始化用户可以查看，其余用户需要赋权后才可以使用。

gs_read_file_size_from_remote 用于读取指定文件的大小。用于读取指定文件的大小，gs_repair_file 函数修复文件时，要先获取远端关于这个文件的大小，用于校验本地文件缺失的文件信息，然后将缺失的文件逐个修复。默认只有初始化用户可以查看，其余用户需要赋权后才可以使用。

5.37.18账本数据库函数

get_dn_hist_relhash

5.37.19AI 特性函数

create_snapshot

create_snapshot_internal

prepare_snapshot_internal

prepare_snapshot

manage_snapshot_internal

archive_snapshot

publish_snapshot

purge_snapshot_internal

purge_snapshot

sample_snapshot

5.37.20PKG_SERVICE 函数

isubmit_on_nodes

submit_on_nodes

5.37.21其他函数

o_tsvector_for_batch	value_of_perc	session_conte	bind_variable	job_update	job_cancel	job_finish

	entile	xt				
similar_escape	table_skewness （不可用）	timetz_text	time_text	reltime_text	abstime_text	_pg_keysequal
analyze_query (不可用)	analyze_workload （不可用）	ssign_table_type	gs_comm_proxy_thread_status	gs_txid_oldestxmin	pg_cancel_session	pg_stat_segment_space_info
remote_segment_space_info	set_cost_params	set_weight_params	start_collect_workload	tdigest_in	tdigest_merge	tdigest_merge_to_one
tdigest_mergep	tdigest_out	pg_get_delta_info	disable_conn	-	-	-

5.38 Global SysCache 特性函数

- gs_gsc_table_detail(database_id dafault NULL, rel_id default NULL)

描述：查看数据库里全局系统缓存的表元数据。调用该函数的用户需要具有 SYSADMIN 权限。

参数：指定需要查看全局系统缓存的数据库和表，database_id 默认值 NULL 或者-1 表示所有的数据库，0 表示共享表，其他数字表示指定数据库及共享表，rel_id 表示指定表的 oid，默认值 NULL 或者-1 表示所有的表，其他值表示指定的表，表不存在会报错。

返回值类型：Tuple

示例:

```
postgres=# select * from gs_gsc_table_detail(-1) limit 1;
database_oid | database_name | reloid |         relname         | relnamespace
| reltype | reloftype | relowner | relam | relfilenode | reltablespace |
```

```
relhasindex | relisshared | relkind | relnatts | relhasoids | relhaspkey | parttype
| tdhasuids | attnames  | extinfo
-------------+-------------+---------+----------+------------+------------+----------
---+---------+----------+----------+-------+------------+--------------+--
----------+------------+---------+----------+-----------+-----------+----
------+----------+----------+---------
0 |               |   2676 | pg_authid_rolname_index |           11 |       0 |0
|       10 |   403 |           0 |          1664 | f           | t           |
i       |        1 | f           | f           | n         | f           | 'rolname'
|
(1 row)
```

- gs_gsc_catalog_detail(database_id dafault NULL, rel_id default NULL)

描述：查看数据库里全局系统缓存的系统表行信息。调用该函数的用户需要具有 SYSADMIN 权限。

参数：指定需要查看全局系统缓存的数据库和表，database_id 默认值 NULL 或者-1 表示所有的数据库，0 表示共享表，其他数字表示指定数据库及共享表， rel_id 表示指定表的 id，仅包含所有有系统缓存的系统表，默认值 NULL 或者-1 表示所有的表，其他值表示指定的表，表不存在会报错。

返回值类型：Tuple

示例：

```
postgres=# select * from gs_gsc_catalog_detail(16574, 1260);
database_id | database_name | rel_id | rel_name  | cache_id |  self  |  ctid  |
infomask | infomask2 | hash_value | refcount
------------+---------------+--------+-----------+----------+--------+------
--+----------+-----------+------------+----------
           0 |               |   1260 | pg_authid |       10 | (0, 9) | (0, 9)
|    10507 |        26 |  531311568 |        10
           0 |               |   1260 | pg_authid |       11 | (0, 4) | (0, 4)
|     2313 |        26 |  365368336 |         1
           0 |               |   1260 | pg_authid |       11 | (0, 9) | (0, 9)
|    10507 |        26 | 3911517328 |        10
           0 |               |   1260 | pg_authid |       11 | (0, 7) | (0, 7)
|     2313 |        26 | 1317799983 |         1
           0 |               |   1260 | pg_authid |       11 | (0, 5) | (0, 5)
|     2313 |        26 | 3664347448 |         1
           0 |               |   1260 | pg_authid |       11 | (0, 1) | (0, 1)
|     2313 |        26 |  276477273 |         1
```

```
          0 |                |   1260 | pg_authid |       11 | (0, 3) | (0, 3)
|     2313 |        26 | 2465837659 |         1
          0 |                |   1260 | pg_authid |       11 | (0, 8) | (0, 8)
|     2313 |        26 | 3205288035 |         1
          0 |                |   1260 | pg_authid |       11 | (0, 6) | (0, 6)
|     2313 |        26 |  131811687 |         1
          0 |                |   1260 | pg_authid |       11 | (0, 2) | (0, 2)
|     2313 |        26 | 1226484587 |         1
(10 rows)
```

- gs_gsc_clean(database_id dafault NULL)

描述：清理 global syscache 的缓存，需要注意，正在使用中的数据不会被清理。调用该函数的用户需要具有 SYSADMIN 权限。

参数：指定需要清理全局系统缓存的数据库，默认值 NULL 或者-1 表示清理所有的数据库全局系统缓存，0 表示只清理共享表的全局系统缓存，其他数字表示清理指定数据库以及共享表的全局系统缓存，表不存在会报错。

返回值类型：bool

示例：

```
 postgres=# select * from gs_gsc_clean();
gs_gsc_clean
--------------
t
(1 row)
```

- gs_gsc_dbstat_info(database_id default NULL)

描述：获取本地节点的 GSC 的内存统计信息，包括 tuple、relation、partition 的缓存查询，命中，加载、失效、占用空间信息，DB 级别的淘汰信息，线程引用信息，内存占用信息。可以用于定位性能问题，例如当发现 hits/searches 数组远小于 1 时，可能是 global_syscache_threshold 设置太小，导致查询命中率下降。调用该函数的用户需要具有 SYSADMIN 权限。

参数：指定需要查看的数据库全局系统缓存统计信息，NULL 或者-1 表示查看所有的数据库，0 表示只查看共享表信息，其他数字表示查看指定的数据库和共享表的信息。不合法的输入值，表不存在会报错。

返回值类型：Tuple

示例：

```
 postgres=# select * from gs_gsc_dbstat_info();
database_id | database_name | tup_searches | tup_hits | tup_miss | tup_count |
tup_dead | tup_memory | rel_searches | rel_hits | rel_mis
s | rel_count | rel_dead | rel_memory | part_searches | part_hits | part_miss |
part_count | part_dead | part_memory | total_memory | swa
pout_count | refcount
-------------+---------------+--------------+----------+----------+-----------
-+----------+------------+--------------+----------+--------
--+-----------+----------+------------+---------------+-----------+-----------
-+------------+-----------+-------------+--------------+----
-----------+----------
           0 |               |          300 |      235 |       31 |        22 |
2 |       9752 |          598 |      108 |       1
8 |        18 |        0 |      77720 |             0 |         0 |         0 |
0 |         0 |          0 |       752912 |
         0 |        0
       16574 | postgres      |         3368 |     2289 |      329 |       273 |
0 |      92593 |         1113 |      524 |       4
8 |        48 |        0 |     340456 |             0 |         0 |         0 |
0 |         0 |          0 |      4124792 |
         0 |       10
(2 rows)
```

5.39 数据损坏检测修复函数

- gs_verify_data_file(verify_segment bool)

描述：校验当前实例当前库是否存在文件丢失的情况。校验只包括数据表主文件是否有中间段丢失的情况。默认参数是 false，表示不校验段页式表数据文件。参数设置为 true 时仅校验段页式表文件。默认只有初始化用户、具有 sysadmin 属性的用户以及在运维模式下具有运维管理员属性的用户可以查看，其余用户需要赋权后才可以使用。

返回的结果：

- 非段页式表：rel_oid 和 rel_name 是对应文件的表 oid 和表名，miss_file_path 表示丢失文件的相对路径。
- 段页式表：因所有表存放在相同文件中，所以 rel_oid 和 rel_name 无法显示具体表的信息。对于段页式表，如果第一个文件损坏，不会检查出后面的.1 .2 等文件。例如 3、3.1、3.2 损坏，只能检查出 3 损坏。当段页式文件不足 5 个时，使用函数检

测时，未生成的文件也会校验出来，例如只有 1 和 2 文件，校验段页式时，也会检测出 3，4，5 文件。以下示例，第一个是校验非段页式表的示例，第二是校验段页式表的示例。

参数说明：

verify_segment

指定文件校验的范围。false 校验非段页式表；true 校验段页式表。

取值范围：true 和 false，默认是 false。

返回值类型：record

示例：

校验非段页式表

```
postgres=# select * from gs_verify_data_file();
node_name          | rel_oid |  rel_name    |  miss_file_path
-------------------+---------+--------------+-----------------
dn_6001_6002_6003 |   16554 |     test     | base/16552/24745
```

校验段页式表

```
postgres=# select * from gs_verify_data_file(true);
    node_name      | rel_oid | rel_name | miss_file_path
-------------------+---------+----------+---------------
 dn_6001_6002_6003 |       0 | none     | base/16573/2
```

- gs_repair_file(tableoid Oid，path text, timeout int)

描述：根据传入的参数修复文件，仅支持有正常主备连接的主 DN 使用。参数依据 gs_verify_data_file 函数返回的 oid 和路径填写。段页式表 tableoid 赋值为 0 到 4,294,967,295 的任意值（内部校验根据文件路径判断是否是段页式表文件，段页式表文件则不使用 tableoid）。修复成功返回值为 true，修复失败会显示具体失败原因。默认只有在主 DN 节点上，使用初始化用户、具有 sysadmin 属性的用户以及在运维模式下具有运维管理员属性的用户可以查看，其余用户需要赋权后才可以使用。

注意

- 当 DN 实例上存在文件损坏时，进行升主会校验出错，报 PANIC 退出无法升主，为正常现象。可在其他 DN 升主后，通过备 DN 自动修复进行修复。
- 当文件存在但是大小为 0 时，此时不会去修复该文件，若想要修复该文件，需要将

为 0 的文件删除后再修复。

- 删除文件需要等文件 fd 自动关闭后再修复，人工操作可以执行重启进程、主备切换命令。

参数说明：

- tableoid

要修复的文件对应的表 oid，依据 gs_verify_data_file 函数返回的列表中 rel_oid 一列填写。

取值范围：Oid，0 - 4294967295。注意：输入负值等都会被强制转成非负整数类型。

- path

需要修复的文件路径，依据 gs_verify_data_file 函数返回的列表中 miss_file_path 一列填写。

取值范围：字符串。

- timeout

等待备 DN 回放的时长，修复文件需要等待备 DN 回放到当前主 DN 对应的位置，根据备 DN 回放所需时长设定。

取值范围：60s - 3600s。

返回值类型：bool

示例：

```
postgres=# select * from gs_repair_file(16554,'base/16552/24745',360);
gs_repair_file
----------------
t
```

- local_bad_block_info()

描述：显示本实例页面损坏的情况。从磁盘读取页面，发现页面 CRC 校验失败时进行记录。默认只有初始化用户、具有 sysadmin 属性的用户、具有监控管理员属性的用户以及在运维模式下具有运维管理员属性的用户、以及监控用户可以查看，其余用户需要赋权后才可以使用。

显示信息：file_path 是损坏文件的相对路径，如果是段页式表，则显示的是逻辑信息，

不是实际的物理文件信息。block_num 是该文件损坏的具体页面号，页面号从 0 开始。check_time 表示发现页面损坏的时间。repair_time 表示修复页面的时间。

返回值类型：record

示例：

```
 postgres=# select * from local_bad_block_info();
node_name     | spc_node | db_node | rel_node| bucket_node | fork_num | block_num
|    file_path      |  check_time             |   repair_time
----------------+-------+--------+--------+-------------+----------+-------
----+----------------+-------------------------+---------------------------
----
dn_6001_6002_6003|  1663 |  16552 |  24745 |         -1     |     0     | 0
| base/16552/24745 | 2022-01-13 20:19:08.385004+08 | 2022-01-13
20:19:08.407314+08
(1 rows)
```

- local_clear_bad_block_info()

描述：清理 local_bad_block_info 中已修复页面的数据，也就是 repair_time 不为空的信息。默认只有初始化用户、具有 sysadmin 属性的用户以及在运维模式下具有运维管理员属性的用户、以及监控用户可以查看，其余用户需要赋权后才可以使用。

返回值类型：bool

示例：

```
 postgres=# select * from local_clear_bad_block_info();
result
--------
t
(1 rows)
```

- gs_verify_and_tryrepair_page (path text, blocknum oid, verify_mem bool, is_segment bool)

描述：校验本实例指定页面的情况。默认只有在主 DN 节点上，使用初始化用户、具有 sysadmin 属性的用户以及在运维模式下具有运维管理员属性的用户可以查看，其余用户需要赋权后才可以使用。

返回的结果信息：disk_page_res 表示磁盘上页面的校验结果；mem_page_res 表示内存中页面的校验结果；is_repair 表示在校验的过程中是否触发修复功能，t 表示已修复，f 表示未修复。

注意

- 当 DN 实例上存在页面损坏时，进行升主会校验出错，报 PANIC 退出无法升主，为正常现象。可在其他 DN 升主后，通过备 DN 自动修复进行修复。

参数说明：

- path

损坏文件的路径。依据 local_bad_block_info 中 file_path 一列填写。

取值范围：字符串。

- blocknum

损坏文件的页号。依据 local_bad_block_info 中 block_num 一列填写。

取值范围：Oid，0 - 4294967295。注意：输入负值等都会被强制转成非负整数类型。

- verify_mem

指定是否校验内存中的指定页面。设定为 false 时，只校验磁盘上的页面。设置为 true 时，校验内存中的页面和磁盘上的页面。如果发现磁盘上页面损坏，会将内存中的页面做一个基本信息校验刷盘，修复磁盘上页面。如果校验内存页面时发现页面不在内存中，会经内存接口读取磁盘上的页面。此过程中如果磁盘页面有问题，则会触发远程读自动修复功能。

取值范围：bool，true 和 false。

- is_segment

是否是段页式表。根据 local_bad_block_info 中的 bucket_node 列值决定。如果 bucket_node 为-1 时，表示不是段页式表，将 is_segment 设置为 false；非-1 的情况将 is_segment 设置为 true。

取值范围：bool，true 和 false。

返回值类型：record

示例：

```
postgres=# select * from
gs_verify_and_tryrepair_page('base/16552/24745',0,false,false);
node_name         |       path       | blocknum |        disk_page_res
| mem_page_res | is_repair
-----------------+------------------+----------+-------------------------
---+--------------+----------
```

```
dn_6001_6002_6003 | base/16552/24745 |      0       | page verification
succeeded.|               | f
```

- gs_repair_page(path text, blocknum oid, is_segment bool, timeout int)

描述：修复本实例指定页面，仅支持有正常主备连接的主 DN 使用。页面修复成功返回 true，修复过程中出错会有报错信息提示。默认只有在主 DN 节点上，使用初始化用户、具有 sysadmin 属性的用户以及在运维模式下具有运维管理员属性的用户可以查看，其余用户需要赋权后才可以使用。

注意

- 当 DN 实例上存在页面损坏时，进行升主会校验出错，报 PANIC 退出无法升主，为正常现象。可在其他 DN 升主后，通过备 DN 自动修复进行修复。

参数说明：

- path

损坏页面的路径。根据 local_bad_block_info 中 file_path 一列设置，或者是 gs_verify_and_tryrepair_page 函数中 path 一列设置。

取值范围：字符串。

- blocknum

损坏页面的页面号。根据 local_bad_block_info 中 block_num 一列设置，或者是 gs_verify_and_tryrepair_page 函数中 blocknum 一列设置。

取值范围：Oid，0 - 4294967295。注意：输入负值等都会被强制转成非负整数类型。

- is_segment

是否是段页式表。根据 local_bad_block_info 中的 bucket_node 列值决定，如果 bucket_node 为-1 时，表示不是段页式表，将 is_segment 设置为 false；非-1 的情况将 is_segment 设置为 true。

取值范围：bool，true 或者 false。

- timeout

等待备 DN 回放的时长。修复页面需要等待备 DN 回放到当前主 DN 对应的位置，根据备 DN 回放所需时长设定。

取值范围：60s - 3600s。

返回值类型：bool

示例：

```
 postgres=# select * from gs_repair_page('base/16552/24745',0,false,60);
result
--------
t
(1 row)
```

5.40 XML 类型函数

以下函数均继承自开源 PG9.2。

- xmlparse ({ DOCUMENT | CONTENT 目 value)

  描述：使用函数 xmlparse，来从字符数据产生 xml 类型的值。

  参数：数据类型为 text

  返回值类型： xml

  示例：

```
postgres=# SELECT XMLPARSE (DOCUMENT '<?xml version="1.0"?><book>
<title>Manual</title><chapter>...</chapter></book>');
xmlparse
-----------------------------------------------------
<book><title>Manual</title><chapter>...</chapter></book> (1 row)
postgres=# SELECT XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>');
xmlparse
------------------------------
abc<foo>bar</foo><bar>foo</bar>
(1 row)
```

- xmlserialize({ DOCUMENT | CONTENT 目 value AS type)

  描述：使用函数 xmlserialize，来从 xml 产生一个字符串。

  参数：类型可以是 character ， character varying 或 text 或其中某个的变种。

  返回值类型：xml

  示例：

```
postgres=# SELECT XMLSERIALIZE(CONTENT 'good' AS CHAR(10));
xmlserialize
```

```
-------------
good
(1 row)
postgres=# SELECT xmlserialize(DOCUMENT '<head>bad</head>' as text);
xmlserialize
-----------------
<head>bad</head>
(1 row)
```

说明：

当一个字符串值在没有通过 XMLPARSE 或 XMLSERIALIZE 的情况下，与 xml 类型

进行转换时，分别的选择 DOCUMENT 与 CONTENT 由"XML option" 会话配置参 数决定，这个配置参数可以由标准命令来设置：

SET XML OPTION { DOCUMENT | CONTENT };

或使用类似的语法来设置：

SET xmloption TO { DOCUMENT | CONTENT };

- xmlcomment(text)

描述：创建一个 XML 值，并且它包含一个用指定文本作为内容的 XML 注释。该文本不包 含“--”字符且不存在是“-”字符的结尾、 符合 XML 注释的格式要求。且当参数为空时、结果 也为空。

参数：数据类型为 text。

返回值类型： xml

示例：

```
postgres=# SELECT xmlcomment('hello');
xmlcomment
--------------
<!--hello-->
```

- xmlconcat(xml[, ...])

描述：将由单个 XML 值组成的列表串接成一个单独的值，该值包含一个 XML 的内容片 断。其中空值会被忽略，并且只有当所有参数都为空时结果才为空。

参数：数据类型为 xml。

返回值类型： xml

示例：

```
postgres=# SELECT xmlconcat('<abc/>', '<bar>foo</bar>');
xmlconcat
----------------------
<abc/><bar>foo</bar>
```

- xmlelement(name name [, xmlattributes(value [AS attname] [, ...])] [, content, ...])

  描述：使用给定名称、属性和内容产生一个 XML 元素。

  返回值类型： xml

  示例：

```
postgres=# SELECT xmlelement(name foo);
xmlelement
------------
<foo/>
```

- xmlforest(content [AS name] [, ...])

  描述：使用给定名称和内容产生一个元素的 XML 序列。

  返回值类型： xml

  示例：

```
postgres=# SELECT xmlforest('abc' AS foo, 123 AS bar);
xmlforest
------------------------------
<foo>abc</foo><bar>123</bar>
```

- xmlpi(name target [, content])

  描述：创建一个 XML 处理指令。若内容不为空、则内容不能包含字符序列。

  返回值类型： xml

  示例：

```
postgres=# SELECT xmlpi(name php, 'echo "hello world";');
xmlpi
-----------------------------
<?php echo "hello world";?>
```

- xmlroot(xml, version text | no value [, standalone yes|no|no value])

  描述：修改一个 XML 值的根结点的属性。如果指定了一个版本，它会替换根节点的版

本

声明中的值、如果指定了一个独立设置、它会替换根节点的独立声明中的值。 示例：

```
postgres=# SELECT xmlroot('<?xml version="1.1"?>
<content>abc</content>',version '1.0', standalone yes);
xmlroot
----------------------------------------
<?xml version="1.0" standalone="yes"?>
<content>abc</content>
```

- xmlagg(xml)

描述：该函数是一个聚集函数、它将聚集函数调用的输入值串接起来，且支持跨行串接。

参数： xml

返回值类型： xml

示例：

```
postgres=# CREATE TABLE xmltest (
id int,
data xml
);
postgres=# INSERT INTO xmltest VALUES (1, '<value>one</value>');
INSERT 0 1
postgres=# INSERT INTO xmltest VALUES (2, '<value>two</value>');
INSERT 0 1
postgres=# SELECT xmlagg(data) FROM xmltest;
xmlagg
----------------------------------
<value>one</value><value>two</value>
(1 row)
```

- xmlexists(text passing [BY REF] xml [BY REF])

描述：评价一个 XPath 1.0 表达式(第一个参数)，以传递的 XML 值作为其上下文项。 如果 评价的结果产生一个空节点集，该函数返回 false，如果产生任何其他值，则返回 true。

如果任何参数为空，则函数返回 null。 作为上下文项传递的非空值必须是一个 XML 文 档，而不是内容片段或任何非 XML 值。

参数： xml

返回值类型： bool

示例：

```
postgres=# SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY REF
'<towns><town>Toronto</town><town>Ottawa</town></towns>');
xmlexists
------------
t
(1 row)
```

- xml_is_well_formed(text)

  描述：检查 text 是不是正确的 XML 类型格式、返回值为布尔类型。

  参数： text

  返回值类型： bool

  示例：

```
postgres=# SELECT xml_is_well_formed('<>');
xml_is_well_formed
--------------------
f
(1 row)
```

- xml_is_well_formed_document(text)

  描述：检查 text 是不是正确的 XML 类型格式、返回值为布尔类型。

  参数： text

  返回值类型： bool

  示例：

```
postgres=# SELECT xml_is_well_formed_document('<pg:foo
xmlns:pg="http://postgresql.org/stuff">bar</pg:foo>');
xml_is_well_formed_document
----------------------------
t
(1 row)
```

- xml_is_well_formed_content(text)

  描述：检查 text 是不是正确的 XML 类型格式、返回值为布尔类型。

  参数： text

  返回值类型： bool

示例：

```
postgres=# select xml_is_well_formed_content('k');
xml_is_well_formed_content
----------------------------
t
(1 row)
```

- xpath(xpath, xml [, nsarray])

描述：在 XML 值 xml 上计算 XPath 1.0 表达式 xpath (a text value)。它返回一个 XML 值的数组，该数组对应于该 XPath 表达式产生的结点集合。如果该 XPath 表达式返回一个标量值而不是一个结点集合，将会返回一个单一元素的数组。

第二个参数必须是一个良构的 XML 文档。特殊地，它必须有一个单一根结点元素。

该函数可选的第三个参数是一个名字空间映射的数组。这个数组应该是一个二维 text 数组，其第二轴长度等于 2 (即它应该是一个数组的数组，其中每一个都由刚好 2 个元素组 成)。每个数组项的第一个元素是名字空间的名称(别名)，第二个元素是名字空间的 URI。并不要求在这个数组中提供的别名和在 XML 文档本身中使用的那些名字空间相同(换句话说，在 XML 文档中和在 xpath 函数环境中，别名都是本地的)。

返回值类型： xml

示例：

```
postgres=# SELECT xpath('/my:a/text()', '<my:a
xmlns:my="http://example.com">test</my:a>',ARRAY[ARRAY['my',
'http://example.com']]);
xpath
--------
{test}
(1 row)
```

- xpath_exists(xpath, xml [, nsarray])

描述：该函数是 xpath 函数的一种特殊形式。这个函数不是返回满足 XPath 1.0 表达式的单一 XML 值，它返回一个布尔值表示查询是否被满足(具体来说，它是否产生了空节点集以外的任何值)。这个函数等价于标准的 XMLEXISTS 谓词，不过它还提供了对一个名字空间映射参数的支持。

返回值类型： bool

示例：

```
postgres=# SELECT xpath_exists('/my:a/text()', '<my:a
xmlns:my="http://example.com">test</my:a>',ARRAY[ARRAY['my',
'http://example.com']]);
xpath_exists
--------------
t
(1 row)
```

- query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)

  描述：该函数将关系表的内容映射成 XML 值，可理解为 XML 的导出功能。

  返回值类型： xml

- query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

  描述：返回 XML 模式文档，这些文档描述上述对应函数所执行的映射。

- query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

  描述：产生 XML 数据映射和对应的 XML 模式，并把产生的结果链接在一起放在一个文档 中。

- cursor_to_xml(cursor refcursor, count int, nulls boolean,tableforest boolean, targetns text)

  描述：该函数将关系表的内容映射成 XML 值，可理解为 XML 的导出功能。

  返回值类型： xml

- cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)

  描述：返回 XML 模式文档，这些文档描述上述对应函数所执行的映射。 返回值类型：xml

- schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)

  描述：用于产生相似的整个模式或整个当前数据库的映射。

  返回值类型： xml

- schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

  描述：用于产生相似的整个模式或整个当前数据库的映射。

- schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

  描述：用于产生相似的整个模式或整个当前数据库的映射。

- database_to_xml(nulls boolean, tableforest boolean, targetns text)

  描述：用于产生相似的整个模式或整个当前数据库的映射。

  返回值类型： xml

- database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)

  描述：用于产生相似的整个模式或整个当前数据库的映射。

- database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)

  描述：用于产生相似的整个模式或整个当前数据库的映射。

- table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)

  描述：该函数将关系表的内容映射成 XML 值，可理解为 XML 的导出功能。 返回值类型： xml

- table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)

  描述：返回 XML 模式文档，这些文档描述上述对应函数所执行的映射。

  返回值类型： xml

- table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)

  描述：产生 XML 数据映射和对应的 XML 模式，并把产生的结果链接在一起放在一个文档中。

  说明：

  xpath 相关函数仅支持 xpath() 和 xpath_exists()、由于其使用 xpath 语言查询 XML 文档，而这些函数都依赖于 libxml2 库，且这个库仅在 Xpath1.0 提供、所以 对 XPath 的限制为 1.0。

  不支持 xquery、xml extension、xslt。

5.41 废弃函数

GBase 8s 中下列函数在最新版本中已废弃：

gs_wlm_get_session_info	gs_wlm_get_user_session_info	pgxc_get_csn

pgxc_lock_for_backup	pgxc_lock_for_sp_database	pgxc_lock_for_tran
pgxc_ pool_connection_status	pgxc_ pool_ reload	pgxc_ prepared_xa
pgxc_version	array_extend	prepare_statement
pg_stat_get_ pooler_status	pg_stat_get_wlm_ node_ resource_info	pg_stat_get_wlm_s
gs_stat_ustore	table_skewness(text)	table_skewness(te

6 类型转换

6.1 概述

背景信息

在 SQL 语言中，每个数据都与一个决定其行为和用法的数据类型相关。GBase 8s 提供一个可扩展的数据类型系统，该系统比其它 SQL 实现更具通用性和灵活性。因而，GBase 8s 中大多数类型转换是由通用规则来管理的，这种做法允许使用混合类型的表达式。

GBase 8s 扫描/分析器只将词法元素分解成五个基本种类：整数、浮点数、字符串、标识符和关键字。大多数非数字类型首先表现为字符串。SQL 语言的定义允许将常量字符串声明为具体的类型。例，下面查询：

```
postgres=# SELECT text 'Origin' AS "label", point '(0,0)' AS "value"; label | value
--------+-------
Origin | (0,0)
(1 row)
```

示例中有两个文本常量，类型分别为 text 和 point。如果没有为字符串文本声明类型，则该文本首先被定义成一个 unknown 类型。

在 GBase 8s 分析器里，有四种基本的 SQL 结构需要独立的类型转换规则：

- 函数调用

多数 SQL 类型系统是建筑在一套丰富的函数上的。函数调用可以有一个或多个参数。因为 SQL 允许函数重载，所以不能通过函数名直接找到要调用的函数，分析器必须根据函数提供的参数类型选择正确的函数。

- 操作符

SQL 允许在表达式上使用前缀或后缀（单目）操作符，也允许表达式内部使用双目操作符（两个参数）。像函数一样，操作符也可以被重载，因此操作符的选择也和函数一样取决于参数类型。

- 值存储

INSERT 和 UPDATE 语句将表达式结果存入表中。语句中的表达式类型必须和目标字段的类型一致或者可以转换为一致。

- UNION，CASE 和相关构造

因为联合 SELECT 语句中的所有查询结果必须在一列里显示出来，所以每个 SELECT 子句中的元素类型必须相互匹配并转换成一个统一类型。类似地，一个 CASE 构造的结果表达式必须转换成统一的类型，这样整个 case 表达式会有一个统一的输出类型。同样的要求也存在于 ARRAY 构造以及 GREATEST 和 LEAST 函数中。

系统表 pg_cast 存储了有关数据类型之间的转换关系以及如何执行这些转换的信息。详细信息请参见《GBase 8s V8.8.5 5.0.0_数据库参考手册》中系统表 PG_CAST。

语义分析阶段会决定表达式的返回值类型并选择适当的转换行为。数据类型的基本类型分类，包括 ：Boolean，numeric，string，bitstring，datetime，timespan，geometric 和 network。每种类型都有一种或多种首选类型用于解决类型选择的问题。根据首选类型和可用的隐含转换，就可能保证有歧义的表达式（那些有多个候选解析方案的）得到有效的方式解决。

所有类型转换规则都是建立在下面几个基本原则上的：

- 隐含转换决不能有奇怪的或不可预见的输出。
- 如果一个查询不需要隐含的类型转换，分析器和执行器不应该进行更多的额外操作。这就是说，任何一个类型匹配、格式清晰的查询不应该在分析器里耗费更多的时间，也不应该向查询中引入任何不必要的隐含类型转换调用。
- 另外，如果一个查询在调用某个函数时需要进行隐式转换，当用户定义了一个有正确参数的函数后，解释器应该选择使用新函数。

6.2 操作符

操作符类型解析

从系统表 pg_operator 中选出要考虑的操作符。如果可以找到一个参数类型以及参 数个数都一致的操作符，那么这个操作符就是最终使用的操作符。如果找到了多个备选的操作符，我们将从中选择一个最合适的。

寻找最优匹配。

抛弃那些输入类型不匹配并且也不能隐式转换成匹配的候选操作符。unknown 文本在这种情况下可以转换成任何东西。如果只剩下一个候选项， 则用之，否则继续下一步。

遍历所有候选操作符，保留那些输入类型匹配最准确的。此时，域被看作和他们的基本类型相同。如果没有一个操作符能被保留，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

遍历所有候选操作符，保留那些需要类型转换时接受(属于输入数据类型的类型范畴的)首选类型位置最多的操作符。如果没有接受首选类型的操作符，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

如果有任何输入参数是 unknown 类型，检查剩余的候选操作符对应参数位置 的类型范畴。在每一个能够接受字符串类型范畴的位置使用 string 类型（这种对字符串的偏爱是合适的，因为 unknown 文本确实像字符串）。另外，如果 所有剩下的候选操作符都接受相同的类型范畴，则选择该类型范畴，否则抛出一个错误（因为在没有更多线索的条件下无法作出正确的选择）。现在抛弃不接受选定的类型范畴的候选操作符，然后，如果任意候选操作符在某个给定的参数位置接受一个首选类型，则抛弃那些在该参数位置接受非首选类型的候选操作符。如果没有一个操作符能被保留，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

如果同时有 unknown 和已知类型的参数，并且所有已知类型的参数都是相同 的类型，那么假设 unknown 参数也是那种类型，并检查哪个候选操作符在 unknown 参数位置接受那个类型。如果只有一个操作符符合，那么使用它。 否则，产生一个错误。

示例

示例 1：阶乘操作符类型解析。在系统表中里只有一个阶乘操作符（后缀!），它以 bigint 作为参数。扫描器给下面查询表达式的参数赋予 bigint 的初始类型：

```
postgres=# SELECT 40 ! AS "40 factorial";

40 factorial
----------------------------------------------
815915283247897734345611269596115894272000000000
(1 row)
```

分析器对参数做类型转换，查询等效于：

```
postgres=# SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

示例 2：字符串连接操作符类型分析。一种字符串风格的语法既可以用于字符串也可以用于复杂的扩展类型。未声明类型的字符串将被所有可能的候选操作符匹配。有一个未声明的参数的例子：

```
postgres=# SELECT text 'abc' || 'def' AS "text and unknown"; text and unknown
------------------
abcdef (1 row)
```

本例中分析器寻找两个参数都是 text 的操作符。确实有这样的操作符，两个参数都是 text

类型。

下面是连接两个未声明类型的值：

```
postgres=# SELECT 'abc' || 'def' AS "unspecified"; unspecified
-------------
abcdef (1 row)
```

说明

- 因为查询中没有声明任何类型，所以本例中对类型没有任何初始提示。因此，分析器查找所有候选操作符，发现既存在接受字符串类型范畴的操作符也存在接受位串类型范畴的操作符。因为字符串类型范畴是首选，所以选择字符串类型范畴的首选类型 text 作为解析未知类型文本的声明类型。

示例 3：绝对值和取反操作符类型分析。GBase 8s 操作符表里面有几条记录对应于前缀操作符@，它们都用于为各种数值类型实现绝对值操作。其中之一用于 float8 类型， 它是数值类型范畴中的首选类型。因此，在面对 unknown 输入的时候， GBase 8s 会使用该类型：

```
postgres=# SELECT @ '-4.5' AS "abs"; abs
----- 4.5
(1 row)
```

此处，系统在应用选定的操作符之前隐式的转换 unknown 类型的文字为 float8 类型。

示例 4：数组包含操作符类型分析。这里是解决一个操作符带有一个已知和一个未知类型输入的例子：

```
postgres=# SELECT array[1,2] <@ '{1,2,3}' as "is subset"; is subset
-----------
t
(1 row)
```

说明

- GBase 8s 操作符表有几条记录对应于中缀操作符<@，但是只有两个可以在左侧接受一个整数数组的操作符是数组包含(anyarray <@ anyarray) 和范围包含(anyelement <@ anyrange)的。因为没有多态的伪类型(参阅 16.3.16 伪类型)是首选的，所以解析器不能解决这个基础上的歧义。然而，最后一个解析规则告诉用户，假设未知类型的文字是和另外一个输入相同的类型，也就是，整数数组。现在只有两个操作符中的一个可以匹配，所以选择数组包含。 (如果用户选择了范围包含，

用户将得到一个错误，因为字符串没有正确的格式成为范围的文字。）

6.3 函数

函数类型解析

1. 从系统表 pg_proc 中选择所有可能被选到的函数。如果使用了一个不带模式修饰的函数名称，那么认为该函数是那些在当前搜索路径中的函数。如果给出一个带修饰的函数名，那么只考虑指定模式中的函数。

   如果搜索路径中找到了多个不同参数类型的函数。将从中选择一个合适的函数。

2. 查找和输入参数类型完全匹配的函数。如果找到一个，则用之。如果输入的实参类型都是 unknown 类型，则不会找到匹配的函数。

3. 如果未找到完全匹配，请查看该函数是否为一个特殊的类型转换函数。

4. 寻找最优匹配。

   (1) 抛弃那些输入类型不匹配并且也不能隐式转换成匹配的候选函数。unknown 文本在这种情况下可以转换成任何东西。如果只剩下一个候选项，则用之， 否则继续下一步。

   (2) 遍历所有候选函数，保留那些输入类型匹配最准确的。此时，域被看作和它们的基本类型相同。如果没有一个函数能准确匹配，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

   (3) 遍历所有候选函数，保留那些需要类型转换时接受首选类型位置最多的函 数。如果没有接受首选类型的函数，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

   (4) 如果有任何输入参数是 unknown 类型，检查剩余的候选函数对应参数位置的 类型范畴。在每一个能够接受字符串类型范畴的位置使用 string 类型（这种对字符串的偏爱是合适的，因为 unknown 文本确实像字符串）。另外，如果所 有剩下的候选函数都接受相同的类型范畴，则选择该类型范畴，否则抛出一个错误（因为在没有更多线索的条件下无法作出正确的选择）。现在抛弃不接受选定的类型范畴的候选

函数，然后，如果任意候选函数在那个范畴接受一个首选类型，则抛弃那些在该参数位置接受非首选类型的候选函数。如果没有一个候选符合这些测试则保留所有候选。如果只有一个候选函数符合， 则使用它；否则，继续下一步。

(5) 如果同时有 unknown 和已知类型的参数，并且所有已知类型的参数有相同的 类型，假设 unknown 参数也是这种类型，检查哪个候选函数可以在 unknown 参数位置接受这种类型。如果正好一个候选符合，那么使用它。否则，产生一个错误。

示例

示例 1：圆整函数参数类型解析。只有一个 round 函数有两个参数（第一个是 numeric，第二个是 integer）。所以下面的查询自动把第一个类型为 integer 的参数转换成 numeric 类型。

```
postgres=# SELECT round(4, 4); round
--------
 4.0000
(1 row)
```

实际上它被分析器转换成：

```
postgres=# SELECT round(CAST (4 AS numeric), 4);
```

因为带小数点的数值常量初始时被赋予 numeric 类型，因此下面的查询将不需要类型转换，并且可能会略微高效一些：

```
postgres=# SELECT round(4.0, 4);
```

示例 2：子字符串函数类型解析。有好几个 substr 函数，其中一个接受 text 和 integer 类型。如果用一个未声明类型的字符串常量调用它，系统将选择接受 string 类型范畴的首选类型（也就是 text 类型）的候选函数。

```
postgres=# SELECT substr('1234', 3); substr
--------
 34
(1 row)
```

如果该字符串声明为 varchar 类型，就像从表中取出来的数据一样，分析器将试着将其转换成 text 类型：

```
postgres=# SELECT substr(varchar '1234', 3); substr
--------
 34
```

```
(1 row)
```

被分析器转换后实际上变成：

```
postgres=# SELECT substr(CAST (varchar '1234' AS text), 3);
```

说明

- 分析器从 pg_cast 表中了解到 text 和 varchar 是二进制兼容的，意思是说一个可以传递给接受另一 个的函数而不需要做任何物理转换。因此，在这种情况下，实际上没有做任何类型转换。

而且，如果以 integer 为参数调用函数，分析器将试图将其转换成 text 类型：

```
postgres=# SELECT substr(1234, 3); substr
--------
 34
(1 row)
```

被分析器转换后实际上变成：

```
postgres=# SELECT substr(CAST (1234 AS text), 3); substr
--------
 34
(1 row)
```

6.4 值存储

值存储数据类型解析

1. 查找与目标字段准确的匹配。
2. 试着将表达式直接转换成目标类型。如果已知这两种类型之间存在一个已注册的转换函数，那么直接调用该转换函数即可。如果表达式是一个未知类型文本，该文本字符串的内容将交给目标类型的输入转换过程。
3. 检查一下看目标类型是否有长度转换。长度转换是一个从某类型到自身的转换。如果在 pg_cast 表里面找到一个，那么在存储到目标字段之前先在表达式上应用。这样的转换函数总是接受一个额外的类型为 integer 的参数，它接收目标字段的 atttypmod 值（实际上是其声明长度，atttypmod 的解释随不同的数据类型而不 同），并且它可能接受一个 Boolean 类型的第三个参数，表示转换是显式的还是隐式的。转换函数负责施加那些

长度相关的语义，比如长度检查或者截断。

示例

character 存储类型转换。对一个目标列定义为 character(20)的语句，下面的语句显示存储值的长度正确：

```
postgres=# CREATE TABLE tpcds.value_storage_t1 ( VS_COL1 CHARACTER(20));
postgres=# INSERT INTO tpcds.value_storage_t1 VALUES('abcdef');
postgres=# SELECT VS_COL1, octet_length(VS_COL1) FROM tpcds.value_storage_t1;
vs_col1     | octet_length
--------------------+--------------
abcdef
(1 row)

postgres=# DROP TABLE tpcds.value_storage_t1;
```

说明

- 这里真正发生的事情是两个 unknown 文本缺省解析成 text，这样就允许||操作符解析成 text 连接。然后操作符的 text 结果转换成 bpchar("空白填充的字符型"，character 类型内部名称)以匹配目标字段类型。不过，从 text 到 bpchar 的转换是二进制兼容的，这样的转换是隐含的并且实际 上不做任何函数调用。最后，在系统表里找到长度转换函数 bpchar(bpchar, integer, Boolean) 并且应用于该操作符的结果和存储的字段长。这个类型相关的函数执行所需的长度检查和额外的空白填充。

6.5 UNION，CASE 和相关构造

SQL UNION 构造必须把那些可能不太相似的类型匹配起来成为一个结果集。解析算法分别应用于联合查询的每个输出字段。INTERSECT 和 EXCEPT 构造对不相同的类型使用和 UNION 相同的算法进行解析。CASE、ARRAY、VALUES、GREATEST 和 LEAST 构造也使用同样的算法匹配它的部件表达式并且选择一个结果数据类型。

UNION，CASE 和相关构造解析

- 如果所有输入都是相同的类型，并且不是 unknown 类型，那么解析成这种类型。
- 如果所有输入都是 unknown 类型则解析成 text 类型（字符串类型范畴的首选类 型）。否则，忽略 unknown 输入。

- 如果输入不属于同一个类型范畴，失败。（unknown 类型除外）
- 如果输入类型是同一个类型范畴，则选择该类型范畴的首选类型。（例外：union 操作会选择第一个分支的类型作为所选类型。）

说明

系统表 pg_type 中 typcategory 表示数据类型范畴，typispreferred 表示是否是 typcategory 分类中的首选类型。

- 把所有输入转换为所选的类型（对于字符串保持原有长度）。如果从给定的输入到所选的类型没有隐式转换则失败。
- 若输入中含 json、txid_snapshot、sys_refcursor 或几何类型，则不能进行 union。

对于 case 和 coalesce，在 TD 兼容模式下的处理

- 如果所有输入都是相同的类型，并且不是 unknown 类型，那么解析成这种类型。
- 如果所有输入都是 unknown 类型则解析成 text 类型。
- 如果输入字符串（包括 unknown，unknown 当 text 来处理）和数字类型，那么解析成字符串类型，如果是其他不同的类型范畴，则报错。
- 如果输入类型是同一个类型范畴，则选择该类型的优先级较高的类型。
- 把所有输入转换为所选的类型。如果从给定的输入到所选的类型没有隐式转换则失败。

对于 case，在 ORA 兼容模式下的处理

- decode(expr, search1, result1, search2, result2, ..., defresult)，也即 case expr when search1 then result1 when search2 then result2 else defresult end; 在 ORA 兼容模式下的处理，将整个表达式最终的返回值类型定为 result1 的数据类型，或者与 result1 同 类型范畴的更高精度的数据类型。（例如，numeric 与 int 同属数值类型范畴，但 numeric 比 int 精度要高，具有更高优先级）
- 将 result1 的数据类型置为最终的返回值类型 preferType，其所属类型范畴为 preferCategory。
- 依次考虑 result2、result3 直至 defresult 的数据类型。如果其类型范畴也是 preferCategory，

即与 result1 具有相同的类型范畴，则判断其精度（优先级）是 否高于 preferType，如果高于，则将 preferType 更新为更高精度的数据类型；如果其类型范畴不是 preferCategory，则判断其数据类型是否可以隐式转换为 preferType，不可以则报错。

- 将最终 preferType 记录的数据类型作为整个表达式最终的返回值类型；表达式的 结果向此类型进行隐式转换。

注 1：

为了兼容一种特殊情况，即表示了超大数字的字符类型向数值类型转换的情况，例如 select decode(1, 2, 2, "53465465676465454657567678676")，大数超过了 bigint、double 等的表示范围。所以，当 result1 的类型范畴为数值类型时，将返回值的类型直 接置为 numeric，以兼容此种特殊情况。

注 2：

数值类型的优先级排序：numeric>float8>float4>int8>int4>int2>int1 字符类型的优先级排序：text>varchar=nvarchar2>bpchar>char

日期类型的优先级排序：

timestamptz>timestamp>smalldatetime>date>abstime>timetz>time

日期跨度类型的优先级排序：interval>tinterval>reltime 注 3：

ORA 兼容模式，开启 set sql_beta_feature = 'a_style_coerce'; 参数的情况下，所支持的隐式类型转换见下图，\代表不需要转换，yes 表示支持，空白表示不支持：

示例

示例 1：Union 中的待定类型解析。这里，unknown 类型文本'b'将被解析成 text 类型。

```
postgres=# SELECT text 'a' AS "text" UNION
SELECT 'b'; text
------
a b
(2 rows)
```

示例 2：简单 Union 中的类型解析。文本 1.2 的类型为 numeric，而且 integer 类型的 1 可以隐含地转换为 numeric，因此使用这个类型。

```
postgres=# SELECT 1.2 AS "numeric" UNION SELECT 1; numeric
---------
1
```

```
1.2
(2 rows)
```

示例 3：转置 Union 中的类型解析。这里，因为类型 real 不能被隐含转换成 integer，但是 integer 可以隐含转换成 real，那么联合的结果类型将是 real。

```
postgres=# SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
real
------ 1
2.2
(2 rows)
```

示例 4：TD 模式下，coalesce 参数输入 int 和 varchar 类型，那么解析成 varchar 类型。A 模式下会报错。

```
--在 A 模式下，创建 A 兼容模式的数据库 a_1。
postgres=# CREATE DATABASE a_1 dbcompatibility = 'A';

--切换数据库为 a_1。postgres=# \c a_1

--创建表 t1。
a_1=# CREATE TABLE t1(a int, b varchar(10));

--查看 coalesce 参数输入 int 和 varchar 类型的查询语句的执行计划。a_1=# EXPLAIN
SELECT coalesce(a, b) FROM t1;
ERROR: COALESCE types integer and character varying cannot be matched LINE 1:
EXPLAIN SELECT coalesce(a, b) FROM t1;
^
CONTEXT: referenced column: coalesce

--删除表。
a_1=# DROP TABLE t1;

--切换数据库为 gbase。a_1=# \c gbase

--在 TD 模式下，创建 TD 兼容模式的数据库 td_1。postgres=# CREATE DATABASE td_1
dbcompatibility = 'C';

--切换数据库为 td_1。postgres=# \c td_1

--创建表 t2。
td_1=# CREATE TABLE t2(a int, b varchar(10));
```

```
--查看 coalesce 参数输入 int 和 varchar 类型的查询语句的执行计划。td_1=# EXPLAIN
VERBOSE select coalesce(a, b) from t2;
QUERY PLAN
------------------------------------------------------------------------------
----------
Data Node Scan (cost=0.00..0.00 rows=0 width=0) Output:
(COALESCE((t2.a)::character varying, t2.b)) Node/s: All dbnodes
Remote query: SELECT COALESCE(a::character varying, b) AS "coalesce" FROM
public.t2 (4 rows)
--删除表。
td_1=# DROP TABLE t2;

--切换数据库为 gbase。td_1=# \c gbase

--删除 A 和 TD 模式的数据库。postgres=# DROP DATABASE a_1; postgres=# DROP DATABASE
td_1;
```

示例 5：ORA 模式下，将整个表达式最终的返回值类型定为 result1 的数据类型，或者与 result1 同类型范畴的更高精度的数据类型。

```
--在 ORA 模式下，创建 ORA 兼容模式的数据库 ora_1。
postgres=# CREATE DATABASE ora_1 dbcompatibility = 'A';

--切换数据库为 ora_1。postgres=# \c ora_1

--开启 Decode 兼容性参数。
set sql_beta_feature='a_style_coerce';

--创建表 t1。
ora_1=# CREATE TABLE t1(c_int int, c_float8 float8, c_char char(10), c_text text,
c_date date);

--插入数据。
ora_1=# INSERT INTO t1 VALUES(1, 2, '3', '4', date '12-10-2010');

--result1 类型为 char，defresult 类型为 text，text 精度更高，返回值的类型由 char
更新为 text。ora_1=# SELECT decode(1, 2, c_char, c_text) AS result,
pg_typeof(result) FROM t1;
result | pg_typeof
--------+-----------
4  | text (1 row)
```

```
--result1 类型为 int，属于数值类型范畴，返回值的类型置为 numeric。
ora_1=# SELECT decode(1, 2, c_int, c_float8) AS result, pg_typeof(result) FROM
t1; result | pg_typeof
--------+-----------
2 | numeric
(1 row)

--不存在 defresult 数据类型向 result1 数据类型之间的隐式转换，报错处理。ora_1=#
SELECT decode(1, 2, c_int, c_date) FROM t1;
ERROR: CASE types integer and timestamp without time zone cannot be matched LINE
1: SELECT decode(1, 2, c_int, c_date) FROM t1;
^
CONTEXT: referenced column: c_date

--关闭 Decode 兼容性参数。set sql_beta_feature='none';

--删除表。
ora_1=# DROP TABLE t1;
DROP TABLE

--切换数据库为 postgres。ora_1=# \c postgres

--删除 ORA 模式的数据库。
postgres=# DROP DATABASE ora_1; DROP DATABASE
```

7 别名

SQL 可以重命名一张表或者一个字段的名称，这个名称为该表或该字段的别名。创建 别名是为了让表名或列名的可读性更强。 SQL 中使用 AS 来创建别名。

7.1 语法格式

列别名语法

```
SELECT
{ * | [column [ AS ] output_name, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ];
```

表别名语法

```
SELECT column1, column2....
```

7.2 参数说明

- output_name

通过使用子句 AS output_name 可以为输出字段取个别名，这个别名通常用于输出 字段的显示。支持关键字 name 、value 和 type 作为列别名。

7.3 示例

用 C 表示表 customer_t1 的别名，查询表内数据。例如：

```
postgres=# SELECT c.c-first-name,c.amount FROM customer_t1 AS c;
c-first-name| amount
-------------+--------
    Grace     |  1000
    Joes      |
    James     |
    Lily      |  2200
    Local     |  5000
......
```

8 锁

如果需要保持数据库数据的一致性，可以使用 LOCK TABLE 来阻止其他用户修改表。

例如，一个应用需要保证表中的数据在事务的运行过程中不被修改。为实现这个目的，则可以对表使用进行锁定。这样将防止数据不被并发修改。

LOCK TABLE 只在一个事务块的内部有用，在事务结束时就会被释放。

8.1 语法格式

```
LOCK [ TABLE ] name  IN lock_mode MODE
```

8.2 参数说明

- name

  要锁定的表的名称。

- lock_mode

  锁的模式。基本模式包括：

  - ACCESS EXCLUSIVE

  这个模式保证其所有者（事务）是可以访问该表的唯一事务。也是缺省锁模式。

  - ACCESS SHARE

  只读取表而不修改的锁模式。

8.3 示例

在执行删除操作时对一个表进行 ACCESS EXCLUSIVE 锁。

```
--创建示例表格。
postgres=# CREATE TABLE graderecord
            (
            number INTEGER,
            name CHAR(20),
            class CHAR(20),
            grade INTEGER
            );
--插入数据。
postgres=# insert into graderecord values('210101','Alan','21.01',92);
```

```
insert into graderecord values('210102','Ben','21.01',62);
insert into graderecord values('210103','Brain','21.01',26);
insert into graderecord values('210204','Carl','21.02',77);
insert into graderecord values('210205','David','21.02',47);
insert into graderecord values('210206','Eric','21.02',97);
insert into graderecord values('210307','Frank','21.03',90);
insert into graderecord values('210308','Gavin','21.03',100);
insert into graderecord values('210309','Henry','21.03',67);
insert into graderecord values('210410','Jack','21.04',75);
insert into graderecord values('210311','Jerry','21.04',60);

--启动进程。
postgres=# START TRANSACTION;

--给示例表格。
postgres=# LOCK TABLE graderecord IN ACCESS EXCLUSIVE MODE;

--删除示例表格。
postgres=# DELETE FROM graderecord WHERE name ='Alan';

postgres=# COMMIT;
```

9 事务

9.1 管理事务

事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位。GBase 8s 数据库支持的事务控制命令有启动、设置、提交、回滚事务。支持的事务隔离级别有读已提交和可重复读。

9.1.1 事务控制

以下是数据库支持的事务命令：

- 启动事务

  用户可以使用 START TRANSACTION 和 BEGIN 语法启动事务。

- 设置事务

  用户可以使用 SET TRANSACTION 或者 SET LOCAL TRANSACTION 语法设置事务特性，详细操作请参考 SET TRANSACTION 章节。

- 提交事务

  用户可以使用 COMMIT 或者 END 完成提交事务的功能，即提交事务的所有操作， 详细操作请参考 COMMIT | END 章节。

- 回滚事务

  回滚是在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤销，详细操作请参考 ROLLBACK 章节。

9.1.2 事务隔离级别

事务隔离级别，它决定多个事务并发操作同一个对象时的处理方式。

说明

在事务中第一个数据修改语句(SELECT, INSERT, DELETE, UPDATE, FETCH, COPY)执行之后，事务隔离级别就不能再次设置。

- READ COMMITTED：读已提交隔离级别，事务只能读到已提交的数据而不会读到未提交的数据，这是缺省值。

实际上，SELECT 查询会查看到在查询开始运行的瞬间该数据库的一个快照。不 过，SELECT 能查看到其自身所在事务中先前更新的执行结果。即使先前更新尚未提交。请注意，在同一个事务里两个相邻的 SELECT 命令可能会查看到不同的快 照，因为其它事务会在第一个 SELECT 执行期间提交。

因为在读已提交模式里，每个新的命令都是从一个新的快照开始的，而这个快照包含所有到该时刻为止已提交的事务，因此同一事务中后面的命令将看到任何已提交的其它事务的效果。这里关心的问题是在单个命令里是否看到数据库里绝对一致的视图。

读已提交模式提供的部分事务隔离对于许多应用而言是足够的，并且这个模式速度快，使用简单。不过，对于做复杂查询和更新的应用，可能需要保证数据库有比读已提交模式更加严格的一致性视图。

- REPEATABLE READ：事务可重复读隔离级别，事务只能读到事务开始之前已提交的数据，不能读到未提交的数据以及事务执行期间其它并发事务提交的修改（但是，查询能查看到自身所在事务中先前更新的执行结果，即使先前更新尚未提交）。这个级别和读已提交是不一样的，因为可重复读事务中的查询看到的是事务开始时的快照，不是该事务内部当前查询开始时的快照，就是说，单个事务内部的 select 命令总是查看到同样的数据，查看不到自身事务开始之后其他并发事务修改后提交的数据。使用该级别的应用必须准备好重试事务，因为可能会发生串行化失败。

9.2 事务控制

事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位。

9.2.1 启动事务

GBase 8s 通过 START TRANSACTION 和 BEGIN 语法启动事务，请参考 START TRANSACTION 和 BEGIN。

9.2.2 设置事务

GBase 8s 通过 SET TRANSACTION 或者 SET LOCAL TRANSACTION 语法设置事务，请参考 SET TRANSACTION。

9.2.3 提交事务

GBase 8s 通过 COMMIT 或者 END 可完成提交事务的功能，即提交事务的所有操作，

请参考 COMMIT | END。

9.2.4 回滚事务

回滚是在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤销。请参考 ROLLBACK。

说明：

数据库中收到的一次执行请求（不在事务块中），如果含有多条语句，将会被打包成一个事务，如果其中有一个语句失败，那么整个请求都将会被回滚。

10 自治事务

自治事务（Autonomous Transaction），在主事务执行过程中新启的独立的事务。自治事务的提交和回滚不会影响主事务已提交的数据，同时自治事务也不受主事务影响。

自治事务在存储过程、函数和匿名块中定义，用 PRAGMA AUTONOMOUS_TRANSACTION 关键字来声明。

10.1 存储过程支持自治事务

自治事务可以在存储过程中定义，标识符为 PRAGMA AUTONOMOUS_TRANSACTION，其余语法与创建存储过程语法相同，示例如下。

```
--建表
create table t2(a int, b int);
insert into t2 values(1,2);
select * from t2;

--创建包含自治事务的存储过程
CREATE OR REPLACE PROCEDURE autonomous_4(a int, b int)  AS
DECLARE
   num3 int := a;
   num4 int := b;
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   insert into t2 values(num3, num4);

END;
/
--创建调用自治事务存储过程的普通存储过程
CREATE OR REPLACE PROCEDURE autonomous_5(a int, b int)  AS
DECLARE
BEGIN

   insert into t2 values(666, 666);
   autonomous_4(a,b);
   rollback;
END;
/
--调用普通存储过程
select autonomous_5(11,22);
```

```
--查看表结果
select * from t2 order by a;
```

上述例子，最后在回滚的事务块中执行包含自治事务的存储过程，直接说明了自治事务的特性，即主事务的回滚，不会影响自治事务已经提交的内容。

10.2 匿名块支持自治事务

自治事务可以在匿名块中定义，标识符为 PRAGMA AUTONOMOUS_TRANSACTION，其余语法与创建匿名块语法相同，示例如下。

```
create table t1(a int ,b text);

START TRANSACTION;
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN

   insert into t1 values(1,'you are so cute,will commit!');
END;
/
insert into t1 values(1,'you will rollback!');
rollback;

select * from t1;
```

上述例子，最后在回滚的事务块前执行包含自治事务的匿名块，也能直接说明了自治事务的特性，即主事务的回滚，不会影响自治事务已经提交的内容。

10.3 用户自定义函数支持自治事务

自治事务可以在函数中定义，标识符为 PRAGMA AUTONOMOUS_TRANSACTION，执行的函数块中使用包含 start transaction 和 commit/rollback 的 sql，其余语法与 CREATE FUNCTION 创建函数语法类似，一个简单的用例如下：

```
--创建表。
CREATE TABLE test1 (a int, b text);
--创建包含自治事务的函数。
CREATE OR REPLACE FUNCTION autonomous_easy_2(i int) RETURNS integer
LANGUAGE plpgsql
```

```
AS $$
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    START TRANSACTION;
    INSERT INTO test1 VALUES (2, 'a');
    IF i % 2 = 0 THEN
        COMMIT;
    ELSE
        ROLLBACK;
    END IF;
  RETURN i % 2 = 0;
END;
$$;
--执行命令。
select autonomous_easy_2(1);
--执行结果。
 autonomous_easy_2
-------------------
                 0
(1 row)
--执行命令，查询表数据。
select * from test1;
--执行结果。
 a | b
---+---
(0 rows)
--执行命令。
select autonomous_easy_2(2);
--执行结果。
 autonomous_easy_2
-------------------
                 1
(1 row)
--执行命令，查询表数据。
select * from test1;
--执行结果
 a | b
---+---
 2 | a
(1 row)
--清空表数据。
```

```
truncate table test1;
--在回滚的事务块中执行包含自治事务的函数。
begin;
insert into test1 values(1,'b');
select autonomous_easy_2(2);
rollback;
--检查表数据。
select * from test1;
--执行结果如下。
 a | b
---+---
 2 | a
(1 row)
```

上述例子，最后在回滚的事务块中执行包含自治事务的函数，直接说明了自治事务的特性，即主事务的回滚，不会影响自治事务已经提交的内容。

10.4 规格约束

注意

- 自治事务执行时，将会在后台启动自治事务 session，我们可以通过 max_concurrent_autonomous_transactions 设置自治事务执行的最大并行数量，该参数取值范围为 0~1024，默认值为 10。
- 当 max_concurrent_autonomous_transactions 参数设置为 0 时，自治事务将无法执行。
- 自治事务新启 session 后，将使用默认 session 参数，不共享主 session 下对象（包括 session 级别变量，本地临时变量，全局临时表的数据等）。
- 触发器函数不支持自治事务。

```
CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);

CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
$$
DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2, NEW.id3);
RETURN NEW;
END
$$ LANGUAGE PLPGSQL;
```

- 自治事务不支持非顶层匿名块调用（仅支持顶层自治事务,包括存储过程、函数、匿名块）。
- 自治事务不支持 ref_cursor 参数传递。

```
create table sections(section_ID int);
insert into sections values(1);
insert into sections values(1);
insert into sections values(1);
insert into sections values(1);

CREATE OR REPLACE function proc_sys_ref()
return SYS_REFCURSOR
IS
declare
  PRAGMA AUTONOMOUS_TRANSACTION;
        C1 SYS_REFCURSOR;
BEGIN
 OPEN C1 FOR SELECT section_ID FROM sections ORDER BY section_ID;
 return C1;
END;
/

CREATE OR REPLACE PROCEDURE proc_sys_call() AS
DECLARE
  C1 SYS_REFCURSOR;
  TEMP NUMBER(4);
BEGIN
  c1 = proc_sys_ref();
  if c1%isopen then
    raise notice '%','ok';
 end if;

  LOOP
    FETCH C1 INTO TEMP;
    raise notice '%',C1%ROWCOUNT;
    EXIT WHEN C1%NOTFOUND;
  END LOOP;
END;
/

select proc_sys_call();
```

```
CREATE OR REPLACE function proc_sys_ref(OUT C2 SYS_REFCURSOR, OUT a int)
return SYS_REFCURSOR
IS
declare
  PRAGMA AUTONOMOUS_TRANSACTION;
        C1 SYS_REFCURSOR;
BEGIN
 OPEN C1 FOR SELECT section_ID FROM sections ORDER BY section_ID;
 return C1;
END;
/

CREATE OR REPLACE PROCEDURE proc_sys_call() AS
DECLARE
  C1 SYS_REFCURSOR;
  TEMP NUMBER(4);
  a int;
BEGIN
  OPEN C1 FOR SELECT section_ID FROM sections ORDER BY section_ID;
  c1 = proc_sys_ref(C1,a);
  if c1%isopen then
    raise notice '%','ok';
 end if;

  LOOP
    FETCH C1 INTO TEMP;
    raise notice '%',C1%ROWCOUNT;
    EXIT WHEN C1%NOTFOUND;
  END LOOP;
END;
/

select proc_sys_call();
```

- 自治事务函数不支持返回非 out 形式的 record 类型。
- 不支持修改自治事务的隔离级别。
- 不支持自治事务返回集合类型（setof）。

```
create table test_in (id int,a date);
create table test_main (id int,a date);
```

```
insert into test_main values (1111,'2021-01-01'),(2222,'2021-02-02');
truncate test_in,test_main;
CREATE OR REPLACE FUNCTION autonomous_f_022(num1  int) RETURNS SETOF test_in
LANGUAGE plpgsql AS $$
DECLARE
count int :=3;
test_row test_in%ROWTYPE;
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    while true
    loop
    if count=3 then
    null;
    else
    if count=2 then
        insert into test_main values (count,'2021-03-03');
        goto pos1;
    end if;
    end if;
    count=count-1;
    end loop;
    insert into test_main values (1000,'2021-04-04');
     <<pos1>>
     for test_row in select * from test_main
     loop
        return next test_row;
       end loop;
      return;
END;
$$
;
```

11 普通表

在当前数据库中创建一个新的空白表，该表由命令执行者所有。在不同的数据库中可以存放相同的表。您可以使用 CREATE TABLE 语句创建表。

11.1 语法格式

```
CREATE TABLE table_name
    (column_name data_type [, ... ]);
```

11.2 参数说明

- table_name

  要创建的表名。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

11.3 示例

执行如下命令创建一个表，表名为 customer_t1，表字段为 c_customer_sk 、c_customer_id、c_first_name 和 c_last_name，每个表字段对应的数据类型为 integer、char(5)、char(6)和 char(8)。

```
postgres=# CREATE TABLE customer_t1
(
    c_customer_sk             integer,
    c_customer_id             char(5),
    c_first_name              char(6),
    c_last_name               char(8),
    Amount                    integer
);
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

12 分区表

一张表内的数据过多时，就会严重影响到数据的查询和操作效率。GBase 8s 支持把一张表从逻辑上分成多个小的分片，从而避免一次处理大量数据，提高处理效率。

GBase 8s 数据库支持这些划分类型：

- 范围分区表：指定一个或多个列划分为多个范围，每个范围创建一个分区，用来存储相应的数据。例如可以采用日期划分范围，将销售数据按照月份进行分区。
- 列表分区表：直接按照一个列上的值来划分出分区。例如可以采用销售门店划分销售数据。
- 间隔分区表：是一种特殊的范围分区，新增了间隔值定义。当插入记录找不到匹配的分区时可以根据间隔值自动创建分区。
- 哈希分区表：根据表的一列，为每个分区指定模数和余数，将要插入表的记录划分到对应的分区中。

分区表的操作除了创建之外还有：

- 查询分区表：按照分区名或者分区中的值查询数据。
- 导入数据：直接导入数据或从现有表格中导入。
- 修改分区表：包括增加分区、删除分区、切割分区、合并分区，以及修改分区名称等。
- 删除分区表：与删除普通表格相同。

12.1 范围分区表的分类

范围分区表按照划分范围的方式，分为以下类别：

- VALUES LESS THAN：通过给出每个分区的上限来确定分区范围。上个分区的上限<=分区的范围<本分区的上限。
- START END：通过以下方式划分：
  - 分区的起点和终点；
  - 仅给出分区起点；
  - 仅给出分区终点；

给出分区起点和终点后，再给出该范围内的间隔值。

以上这些方式的综合应用。

12.2 创建 VALUES LESS THAN 范围分区表

语法格式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
  [, ... ]
)
     PARTITION BY RANGE (partition_key)
         (
         PARTITION partition_name VALUES LESS THAN (partition_value | MAXVALUE})
         [, ... ]
         );
```

参数说明

- partition_table_name

  分区表的名称。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

- partition_key

  partition_key 为分区键的名称。

  对于从句是 VALUE LESS THAN 的语法格式，范围分区策略的分区键最多支持 4 列。

- partition_name

  partition_name 为范围分区的名称。

- VALUES LESS THAN

  分区中的数值必须小于上边界值。

- partition_value

  partition_value 为范围分区的上边界，取值依赖于 partition_key 的类型。

- MAXVALUE

MAXVALUE 表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

示例

示例 1：创建范围分区表 sales_table，含有 4 个分区，分区键为 DATE 类型。分区的范围分别为：sales_date<2021-04-01，2021-04-01<= sales_date<2021-07-01，2021-07-01<=sales_date< 2021-10-01，2021-10-01 <= sales_date< MAXVALUE。

```
--创建分区表 sales_table。
postgres=# CREATE TABLE sales_table
(
    order_no              INTEGER          NOT NULL,
    goods_name            CHAR(20)         NOT NULL,
    sales_date            DATE             NOT NULL,
    sales_volume          INTEGER,
    sales_store           CHAR(20)
)
PARTITION BY RANGE(sales_date)
(
        PARTITION season1 VALUES LESS THAN('2021-04-01 00:00:00'),
        PARTITION season2 VALUES LESS THAN('2021-07-01 00:00:00'),
        PARTITION season3 VALUES LESS THAN('2021-10-01 00:00:00'),
        PARTITION season4 VALUES LESS THAN(MAXVALUE)
);
-- 数据插入分区 season1
postgres=# INSERT INTO sales_table VALUES(1, 'jacket', '2021-01-10 00:00:00',
3,'Alaska');

-- 数据插入分区 season2
postgres=# INSERT INTO sales_table VALUES(2, 'hat', '2021-05-06 00:00:00',
5,'Clolorado');

-- 数据插入分区 season3
postgres=# INSERT INTO sales_table VALUES(3, 'shirt', '2021-09-17 00:00:00',
7,'Florida');

-- 数据插入分区 season4
postgres=# INSERT INTO sales_table VALUES(4, 'coat', '2021-10-21 00:00:00',
9,'Hawaii');
```

12.3 查询分区表

语法格式

```
SELECT * FROM partition_table_name PARTITION { ( partition_name ) | FOR
(  partition_value [, ...] ) }
```

参数说明

- partition_table_name

  分区表的名称。

- partition_name

  partition_name 为分区的名称。

- partition_value

  用于指定分区的值。PARTITION FOR 子句指定的值所在的分区，就是进行查询的分区。

语法示例

示例 2：查询示例 1 中建立的分区表 sales_table。

```
--查询 sales_table 的数据。
postgres=# SELECT * FROM sales_table;
 order_no |      goods_name      |     sales_date      | sales_volume |
sale
s_store
---------+----------------------+---------------------+--------------+------
---
-------------
        1 | jacket               | 2021-01-10 00:00:00 |            3 | Alaska

        2 | hat                  | 2021-05-06 00:00:00 |            5 |
Clolorado

        3 | shirt                | 2021-09-17 00:00:00 |            7 | Florida

        4 | coat                 | 2021-10-21 00:00:00 |            9 | Hawaii

(4 rows)

--查询 sales_table 的 4 季度数据。这里采用“sales_table PARTITION (season4);”来
引用第 4 季度数据所在分区。
```

```
postgres=# SELECT * FROM sales_table PARTITION (season4);
 order_no |      goods_name      |     sales_date      | sales_volume |
sales_store
----------+----------------------+---------------------+--------------+------
---------------
        4 | coat                 | 2021-10-21 00:00:00 |            9 | Hawaii
(1 row)

--查询 sales_table 的 1 季度数据。这里采用“sales_table PARTITION FOR ('2021-3-21
00:00:00')”来引用第 1 季度数据所在分区。其中的'2021-3-21 00:00:00'处于第 1 季度
所在分区。
postgres=# SELECT * FROM sales_table PARTITION FOR ('2021-3-21 00:00:00');
 order_no |      goods_name      |     sales_date      | sales_volume |
sales_store
----------+----------------------+---------------------+--------------+------
---------------
        1 | jacket               | 2021-01-10 00:00:00 |            3 | Alaska
(1 row)
```

12.4 创建 START END 范围分区表

语法格式

START END 范围分区表有多种表达方式，而且这些方式可以在一个分区表内组合使用。

方式一：START(partition_value) END (partition_value | MAXVALUE)方式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
  [, ... ]
)
     PARTITION BY RANGE (partition_key)
         (
         PARTITION partition_name START(partition_value) END (partition_value |
MAXVALUE)
         [, ... ]
         );
```

方式二：START(partition_value)方式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
    [, ... ]
] )
```

```
    PARTITION BY RANGE (partition_key)
        (
        PARTITION partition_name START(partition_value)
        [, ... ]
        );
```

方式三：END(partition_value | MAXVALUE)方式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
    [, ... ]
] )
    PARTITION BY RANGE (partition_key)
        (
        PARTITION partition_name END(partition_value | MAXVALUE)
        [, ... ]
        );
```

方式四：START(partition_value) END (partition_value) EVERY (interval_value)方式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
    [, ... ]
] )
    PARTITION BY RANGE (partition_key)
        (
        PARTITION partition_name START(partition_value) END (partition_value)
EVERY (interval_value)
        [, ... ]
        );
```

参数说明

- partition_table_name

  分区表的名称。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

- partition_key

partition_key 为分区键的名称。

对于从句是 START END 的语法格式，范围分区策略的分区键仅支持 1 列。

- partition_name

  partition_name 为范围分区的名称或者范围分区的名称前缀。

  若该定义是“START(partition_value) END (partition_value) EVERY (interval_value)”从句，假定其中的 partition_name 是 p1，则分区的名称依次为 p1_1, p1_2, …。

  例如对于定义“PARTITION p1 START(1) END(4) EVERY(1)”，则生成的分区是：[1, 2), [2, 3) 和 [3, 4)，名称依次为 p1_1, p1_2 和 p1_3，即此处的 p1 是名称前缀。

  若该定义是第一个分区定义，且该定义有 START 值，则范围（MINVALUE, START）将自动作为第一个实际分区，其名称为 p1_0，然后该定义语义描述的分区名称依次为 p1_1, p1_2, …。

  例如对于完整定义“PARTITION p1 START(1), PARTITION p2 START(2)”，生成的分区是：(MINVALUE, 1), [1, 2) 和 [2, MAXVALUE)，其名称依次为 p1_0, p1_1 和 p2，即此处 p1 是名称前缀，p2 是分区名称。这里 MINVALUE 表示最小值。

  其余的情况都是范围分区名称。

- VALUES LESS THAN

  分区中的数值必须小于上边界值。

- partition_value

  partition_value 为范围分区的端点值（起始或终点），取值依赖于 partition_key 的类型。

- interval_value：

  对[START，END）表示的范围进行切分，interval_value 是指定切分后每个分区的宽度。如果（END-START）值不能整除以 EVERY 值，则仅最后一个分区的宽度小于 EVERY 值。

- MAXVALUE

  MAXVALUE 表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

示例

示例 3：以“START(partition_value) END (partition_value | MAXVALUE)”方式创建 START END 范围分区表 graderecord。含有 3 个分区，分区键为 INTEGER 类型。分区的范围分别为：0<= grade<60，60<= grade<90，90<=grade< MAXVALUE。

```
--创建分区表 graderecord。
postgres=# CREATE TABLE graderecord
  (
  number INTEGER,
  name CHAR(20),
  class CHAR(20),
  grade INTEGER
  )
  PARTITION BY RANGE(grade)
  (
  PARTITION pass START(60) END(90),
  PARTITION excellent START(90) END(MAXVALUE)
  );

-- 数据插入分区。
postgres=# insert into graderecord values('210101','Alan','21.01',92);
postgres=# insert into graderecord values('210102','Ben','21.01',62);
postgres=# insert into graderecord values('210103','Brain','21.01',26);
postgres=# insert into graderecord values('210204','Carl','21.02',77);
postgres=# insert into graderecord values('210205','David','21.02',47);
postgres=# insert into graderecord values('210206','Eric','21.02',97);
postgres=# insert into graderecord values('210307','Frank','21.03',90);
postgres=# insert into graderecord values('210308','Gavin','21.03',100);
postgres=# insert into graderecord values('210309','Henry','21.03',67);
postgres=# insert into graderecord values('210410','Jack','21.04',75);
postgres=# insert into graderecord values('210311','Jerry','21.04',60);

--查询 graderecord 的数据。
postgres=# SELECT * FROM graderecord;
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210103 | Brain                | 21.01                |    26
 210205 | David                | 21.02                |    47
 210102 | Ben                  | 21.01                |    62
 210204 | Carl                 | 21.02                |    77
 210309 | Henry                | 21.03                |    67
 210410 | Jack                 | 21.04                |    75
 210311 | Jerry                | 21.04                |    60
 210101 | Alan                 | 21.01                |    92
 210206 | Eric                 | 21.02                |    97
 210307 | Frank                | 21.03                |    90
 210308 | Gavin                | 21.03                |   100
```

```
(11 rows)

--查询 graderecord 的 pass 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (pass);
ERROR:  partition "pass" of relation "graderecord" does not exist
查询失败。
原因是“PARTITION pass START(60) END(90),”是第一个分区定义，且该定义有 START
值，则范围（MINVALUE, 60）将自动作为第一个实际分区，其名称为“pass_0”。
而该定义语义描述的“60<= grade<90”分区的名称为“pass_1”。

--查询 graderecord 的 pass_0 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (pass_0);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210103 | Brain                | 21.01                |    26
 210205 | David                | 21.02                |    47
(2 rows)

--查询 graderecord 的 pass_1 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (pass_1);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210102 | Ben                  | 21.01                |    62
 210204 | Carl                 | 21.02                |    77
 210309 | Henry                | 21.03                |    67
 210410 | Jack                 | 21.04                |    75
 210311 | Jerry                | 21.04                |    60
(5 rows)

--查询 graderecord 的 execllent 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (excellent);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210101 | Alan                 | 21.01                |    92
 210206 | Eric                 | 21.02                |    97
 210307 | Frank                | 21.03                |    90
 210308 | Gavin                | 21.03                |   100
(4 rows)
```

示例 4：以“START(partition_value) END (partition_value) EVERY (interval_value)”方式创建 START END 范围分区表 metro_ride_record。含有 7 个分区，分区键为 INTEGER 类型。总范围是 ride_stations_number<21， 每 3 站为一个分区。

```
--创建分区表 metro_ride_record。记录乘车人、上下车站点、乘坐站点数量。并按照乘坐站点数量，每 3 站建立一个分区。
postgres=# CREATE TABLE metro_ride_record
  (
  record_number INTEGER,
  name CHAR(20),
  enter_station CHAR(20),
  leave_station CHAR(20),
  ride_stations_number INTEGER
  )
  PARTITION BY RANGE(ride_stations_number)
  (
  PARTITION cost START(3) END(21)  EVERY (3)
   );

-- 数据插入分区。
postgres=# insert into metro_ride_record values('120101','Brain','Tung Chung','Tsing Yi',2);
postgres=# insert into metro_ride_record values('120102','David','Po Lam','Yau Tong',4);
postgres=# insert into metro_ride_record values('120103','Ben','Yau Ma Tei','Wong Tai Sin',6);
postgres=# insert into metro_ride_record values('120104','Carl','Tai Wo Hau','Prince Edward',8);
postgres=# insert into metro_ride_record values('120105','Henry','Admiralty','Lai King',10);
postgres=# insert into metro_ride_record values('120106','Jack','Chai Wan','Central',12);
postgres=# insert into metro_ride_record values('120107','Jerry','Central','Tai Wo Hau',14);
postgres=# insert into metro_ride_record values('120108','Alan','Diamond Hill','Kwai Hing',16);
postgres=# insert into metro_ride_record values('120109','Eric','Jordan','Shek Kip Mei',18);
postgres=# insert into metro_ride_record values('120110','Frank','Lok Fu','Sunny Bay',20);

--查询 metro_ride_record 的数据。
postgres=# SELECT * FROM metro_ride_record;
 record_number |         name         |    enter_station     |
leave_station     | ride_stations_number
---------------+----------------------+----------------------+--------------
```

```
-------+--------------------
       120101 | Brain               | Tung Chung          | Tsing Yi
|                    2
       120102 | David               | Po Lam              | Yau Tong
|                    4
       120103 | Ben                 | Yau Ma Tei          | Wong Tai Sin
|                    6
       120104 | Carl                | Tai Wo Hau          | Prince Edward
|                    8
       120105 | Henry               | Admiralty           | Lai King
|                   10
       120106 | Jack                | Chai Wan            | Central
|                   12
       120107 | Jerry               | Central             | Tai Wo Hau
|                   14
       120108 | Alan                | Diamond Hill        | Kwai Hing
|                   16
       120109 | Eric                | Jordan              | Shek Kip Mei
|                   18
       120110 | Frank               | Lok Fu              | Sunny Bay
|                   20
(10 rows)
```

“PARTITION cost START(3) END(21) EVERY (3)”是第一个分区定义，且该定义有 START 值，则范围 (MINVALUE, 3) 将自动作为第一个实际分区，其名称为“cost_0”。
其余分区依次为“cost_1”、...、“cost_6”.

```
--查询 metro_ride_record 的 cost_0 分区数据。
postgres=# SELECT * FROM metro_ride_record PARTITION (cost_0);
 record_number |         name         |    enter_station    |
leave_station     | ride_stations_number
---------------+----------------------+---------------------+---------------
-------+----------------------
       120101 | Brain               | Tung Chung          | Tsing Yi
|                    2
(1 row)

--查询 metro_ride_record 的 cost_1 分区数据。
postgres=# SELECT * FROM metro_ride_record PARTITION (cost_1);
 record_number |         name         |    enter_station    |
leave_station     | ride_stations_number
---------------+----------------------+---------------------+---------------
```

```
------+--------------------
       120102 | David               | Po Lam              | Yau Tong
|                   4
(1 row)

--查询 metro_ride_record 的 cost_6 分区数据。
postgres=# SELECT * FROM metro_ride_record PARTITION (cost_6);
 record_number |         name         |    enter_station     |
leave_station     | ride_stations_number
---------------+----------------------+----------------------+--------------
------+---------------------
       120109 | Eric                | Jordan              | Shek Kip Mei
|                  18
       120110 | Frank               | Lok Fu              | Sunny Bay
|                  20
(2 rows)
```

示例 5：以“START(partition_value) ”方式创建 START END 范围分区表 graderecord。含有 3 个分区，分区键为 INTEGER 类型。分区的范围分别为：0<= grade<60，60<= grade<90，90<=grade< MAXVALUE。

```
--创建分区表 graderecord。
postgres=# CREATE TABLE graderecord
  (
  number INTEGER,
  name CHAR(20),
  class CHAR(20),
  grade INTEGER
  )
  PARTITION BY RANGE(grade)
  (
  PARTITION pass START(60),
  PARTITION excellent START(90)
  );

-- 数据插入分区。
postgres=# insert into graderecord values('210101','Alan','21.01',92);
postgres=# insert into graderecord values('210102','Ben','21.01',62);
postgres=# insert into graderecord values('210103','Brain','21.01',26);
postgres=# insert into graderecord values('210204','Carl','21.02',77);
postgres=# insert into graderecord values('210205','David','21.02',47);
postgres=# insert into graderecord values('210206','Eric','21.02',97);
```

```
postgres=# insert into graderecord values('210307','Frank','21.03',90);
postgres=# insert into graderecord values('210308','Gavin','21.03',100);
postgres=# insert into graderecord values('210309','Henry','21.03',67);
postgres=# insert into graderecord values('210410','Jack','21.04',75);
postgres=# insert into graderecord values('210311','Jerry','21.04',60);


--查询 graderecord 的数据。
postgres=# SELECT * FROM graderecord;
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210103 | Brain                | 21.01                |    26
 210205 | David                | 21.02                |    47
 210102 | Ben                  | 21.01                |    62
 210204 | Carl                 | 21.02                |    77
 210309 | Henry                | 21.03                |    67
 210410 | Jack                 | 21.04                |    75
 210311 | Jerry                | 21.04                |    60
 210101 | Alan                 | 21.01                |    92
 210206 | Eric                 | 21.02                |    97
 210307 | Frank                | 21.03                |    90
 210308 | Gavin                | 21.03                |   100
(11 rows)

--查询 graderecord 的 pass 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (pass);
ERROR:  partition "pass" of relation "graderecord" does not exist
查询失败。
原因是“PARTITION pass START(60),”是第一个分区定义，且该定义有 START 值，则范围 (MINVALUE, 60) 将自动作为第一个实际分区，其名称为“pass_0”。
而该定义语义描述的“60<= grade<90”分区的名称为“pass_1”。

--查询 graderecord 的 pass_0 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (pass_0);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210103 | Brain                | 21.01                |    26
 210205 | David                | 21.02                |    47
(2 rows)

--查询 graderecord 的 pass_1 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (pass_1);
```

```
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210102 | Ben                  | 21.01                |    62
 210204 | Carl                 | 21.02                |    77
 210309 | Henry                | 21.03                |    67
 210410 | Jack                 | 21.04                |    75
 210311 | Jerry                | 21.04                |    60
(5 rows)

--查询 graderecord 的 execllent 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (excellent);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210101 | Alan                 | 21.01                |    92
 210206 | Eric                 | 21.02                |    97
 210307 | Frank                | 21.03                |    90
 210308 | Gavin                | 21.03                |   100
(4 rows)
```

示例 6：以“END(partition_value | MAXVALUE) ”方式创建 START END 范围分区表 graderecord。含有 3 个分区，分区键为 INTEGER 类型。分区的范围分别为：0<= grade<60，60<= grade<90，90<=grade< MAXVALUE。。

```
--创建分区表 graderecord。
postgres=# CREATE TABLE graderecord
  (
  number INTEGER,
  name CHAR(20),
  class CHAR(20),
  grade INTEGER
  )
  PARTITION BY RANGE(grade)
  (
  PARTITION no_pass END(60),
  PARTITION pass END(90),
  PARTITION excellent END(MAXVALUE)
  );

-- 数据插入分区。
postgres=# insert into graderecord values('210101','Alan','21.01',92);
postgres=# insert into graderecord values('210102','Ben','21.01',62);
postgres=# insert into graderecord values('210103','Brain','21.01',26);
```

```
postgres=# insert into graderecord values('210204','Carl','21.02',77);
postgres=# insert into graderecord values('210205','David','21.02',47);
postgres=# insert into graderecord values('210206','Eric','21.02',97);
postgres=# insert into graderecord values('210307','Frank','21.03',90);
postgres=# insert into graderecord values('210308','Gavin','21.03',100);
postgres=# insert into graderecord values('210309','Henry','21.03',67);
postgres=# insert into graderecord values('210410','Jack','21.04',75);
postgres=# insert into graderecord values('210311','Jerry','21.04',60);


--查询 graderecord 的数据。
postgres=# SELECT * FROM graderecord;
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210103 | Brain                | 21.01                |    26
 210205 | David                | 21.02                |    47
 210102 | Ben                  | 21.01                |    62
 210204 | Carl                 | 21.02                |    77
 210309 | Henry                | 21.03                |    67
 210410 | Jack                 | 21.04                |    75
 210311 | Jerry                | 21.04                |    60
 210101 | Alan                 | 21.01                |    92
 210206 | Eric                 | 21.02                |    97
 210307 | Frank                | 21.03                |    90
 210308 | Gavin                | 21.03                |   100
(11 rows)

--查询 graderecord 的 no_pass 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (no_pass);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210103 | Brain                | 21.01                |    26
 210205 | David                | 21.02                |    47
(2 rows)

--查询 graderecord 的 pass 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (pass);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210102 | Ben                  | 21.01                |    62
 210204 | Carl                 | 21.02                |    77
 210309 | Henry                | 21.03                |    67
```

```
 210410 | Jack                 | 21.04                |     75
 210311 | Jerry                | 21.04                |     60
(5 rows)

--查询 graderecord 的 execllent 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (excellent);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210101 | Alan                 | 21.01                |     92
 210206 | Eric                 | 21.02                |     97
 210307 | Frank                | 21.03                |     90
 210308 | Gavin                | 21.03                |    100
(4 rows)
```

12.5 创建列表分区表

语法格式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
  [, ... ]
)
     PARTITION BY LIST (partition_key)
         (
         PARTITION partition_name VALUES (list_values_clause)
         [, ... ]
         );
```

参数说明

- partition_table_name

  分区表的名称。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

- partition_key

  partition_key 为分区键的名称。

列表分区策略的分区键仅支持 1 列。

- partition_name

  partition_name 为范围分区的名称。

- list_values_clause

  对应分区存在的一个或者多个键值。多个键值之间以逗号分隔。

- VALUES (DEFAULT)

  加入的数据如有“list_values_clause”中未列出的键值，存放在 VALUES (DEFAULT)对应的分区。

- MAXVALUE

  MAXVALUE 表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

示例

示例 7：创建列表分区表 graderecord。含有 4 个分区，分区键为 CHAR 类型。分区的范围分别为：21.01，21.02，21.03，21.04。

```
--创建分区表 graderecord。
postgres=# CREATE TABLE graderecord
  (
  number INTEGER,
  name CHAR(20),
  class CHAR(20),
  grade INTEGER
  )
  PARTITION BY LIST(class)
  (
  PARTITION class_01 VALUES ('21.01'),
  PARTITION class_02 VALUES ('21.02'),
  PARTITION class_03 VALUES ('21.03'),
  PARTITION class_04 VALUES ('21.04')
  );

-- 数据插入分区。
postgres=# insert into graderecord values('210101','Alan','21.01',92);
postgres=# insert into graderecord values('210102','Ben','21.01',62);
postgres=# insert into graderecord values('210103','Brain','21.01',26);
postgres=# insert into graderecord values('210204','Carl','21.02',77);
```

```
postgres=# insert into graderecord values('210205','David','21.02',47);
postgres=# insert into graderecord values('210206','Eric','21.02',97);
postgres=# insert into graderecord values('210307','Frank','21.03',90);
postgres=# insert into graderecord values('210308','Gavin','21.03',100);
postgres=# insert into graderecord values('210309','Henry','21.03',67);
postgres=# insert into graderecord values('210410','Jack','21.04',75);
postgres=# insert into graderecord values('210311','Jerry','21.04',60);


--查询 graderecord 的数据。
postgres=# SELECT * FROM graderecord;
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210410 | Jack                 | 21.04                |    75
 210311 | Jerry                | 21.04                |    60
 210307 | Frank                | 21.03                |    90
 210308 | Gavin                | 21.03                |   100
 210309 | Henry                | 21.03                |    67
 210204 | Carl                 | 21.02                |    77
 210205 | David                | 21.02                |    47
 210206 | Eric                 | 21.02                |    97
 210101 | Alan                 | 21.01                |    92
 210102 | Ben                  | 21.01                |    62
 210103 | Brain                | 21.01                |    26
(11 rows)

--查询 graderecord 的 class_01 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (class_01);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210101 | Alan                 | 21.01                |    92
 210102 | Ben                  | 21.01                |    62
 210103 | Brain                | 21.01                |    26
(3 rows)

--查询 graderecord 的 class_04 分区数据。
postgres=# SELECT * FROM graderecord PARTITION (class_04);
 number |         name         |        class         | grade
--------+----------------------+----------------------+-------
 210410 | Jack                 | 21.04                |    75
 210311 | Jerry                | 21.04                |    60
(2 rows)
```

12.6 创建间隔分区表

语法格式

间隔分区是在范围分区的基础上，增加了间隔值“PARTITION BY RANGE (partition_key)”的定义。

VALUES LESS THAN 间隔分区语法格式：

```
CREATE TABLE partition_table_name
( [column_name data_type ]
  [, ... ]
)
     PARTITION BY RANGE (partition_key)
         (
         INTERVAL ('interval_expr')
         PARTITION partition_name VALUES LESS THAN (partition_value | MAXVALUE})
         [, ... ]
         );
```

START END 间隔分区表语法格式：

方式一：START(partition_value) END (partition_value | MAXVALUE)方式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
  [, ... ]
)
     PARTITION BY RANGE (partition_key)
         (
         INTERVAL ('interval_expr')
         PARTITION partition_name START(partition_value) END (partition_value |
MAXVALUE)
         [, ... ]
         );
```

方式二：START(partition_value) END (partition_value) EVERY (interval_value)方式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
    [, ... ]
] )
     PARTITION BY RANGE (partition_key)
         (
         PARTITION partition_name START(partition_value) END (partition_value)
```

```
EVERY (interval_value)
        [, ... ]
        );
```

方式三：START(partition_value)方式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
    [, ... ]
] )
     PARTITION BY RANGE (partition_key)
         (
         INTERVAL ('interval_expr')
         PARTITION partition_name START(partition_value)
         [, ... ]
         );
```

方式四：END(partition_value | MAXVALUE)方式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
    [, ... ]
] )
     PARTITION BY RANGE (partition_key)
         INTERVAL ('interval_expr')
         (
         PARTITION partition_name END(partition_value | MAXVALUE)
         [, ... ]
         );
```

参数说明

- INTERVAL (‘interval_expr’)

间隔分区定义信息。只支持 TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE 数据类型。

interval_expr 自动创建分区的间隔，例如：

自动创建分区的间隔，例如：1 day、1 month。

- partition_name

partition_name 为范围分区的名称。

系统自动建立的分区按照建立的先后顺序，依次命名为：sys_p1、sys_p2、sys_p3…

示例

示例 8：间隔分区表 sales_table。

```
--创建分区表 sales_table。
postgres=# CREATE TABLE sales_table
(
    order_no              INTEGER          NOT NULL,
    goods_name            CHAR(20)         NOT NULL,
    sales_date            DATE             NOT NULL,
    sales_volume          INTEGER,
    sales_store           CHAR(20)
)
PARTITION BY RANGE(sales_date)
        INTERVAL ('1 month')
        (
        PARTITION start VALUES LESS THAN('2021-01-01 00:00:00'),
        PARTITION later VALUES LESS THAN('2021-01-10 00:00:00')
        );
-- 数据插入分区 later
postgres=# INSERT INTO sales_table VALUES(1, 'jacket', '2021-01-8 00:00:00',
3,'Alaska');

-- 不在已有分区的数据插入，系统会新建分区 sys_p1。
postgres=# INSERT INTO sales_table VALUES(2, 'hat', '2021-04-06 00:00:00',
255,'Clolorado');

-- 不在已有分区的数据插入，系统会新建分区 sys_p2。
postgres=# INSERT INTO sales_table VALUES(3, 'shirt', '2021-11-17 00:00:00',
7000,'Florida');

-- 数据插入分区 start
postgres=# INSERT INTO sales_table VALUES(4, 'coat', '2020-10-21 00:00:00',
9000,'Hawaii');

--查询 sales_table 的数据。
postgres=# SELECT * FROM sales_table;
 order_no |      goods_name      |     sales_date      | sales_volume |
sales_store
----------+----------------------+---------------------+--------------+------
---------------
        4 | coat                 | 2020-10-21 00:00:00 |         9000 | Hawaii
```

```
        1 | jacket                | 2021-01-08 00:00:00 |             3 | Alaska
        2 | hat                   | 2021-04-06 00:00:00 |           255 |
Clolorado
        3 | shirt                 | 2021-11-17 00:00:00 |          7000 | Florida
(4 rows)

--查询 sales_table 的 start 分区数据。这里采用“sales_table PARTITION (start);”
来引用分区。
postgres=# SELECT * FROM sales_table PARTITION (start);
 order_no |      goods_name       |     sales_date      | sales_volume |
sales_store
----------+-----------------------+---------------------+--------------+------
----------------
        4 | coat                  | 2020-10-21 00:00:00 |          9000 | Hawaii
(1 row)

--查询 sales_table 的 later 分区数据。这里采用“sales_table PARTITION (later);”
来引用分区。
postgres=# SELECT * FROM sales_table PARTITION (later);
 order_no |      goods_name       |     sales_date      | sales_volume |
sales_store
----------+-----------------------+---------------------+--------------+------
----------------
        1 | jacket                | 2021-01-08 00:00:00 |             3 | Alaska
(1 row)

--查询 sales_table 的 sys_p1 分区数据。这里采用“sales_table PARTITION (sys_p1);”
来引用分区。
postgres=# SELECT * FROM sales_table PARTITION (sys_p1);
 order_no |      goods_name       |     sales_date      | sales_volume |
sales_store
----------+-----------------------+---------------------+--------------+------
----------------
        2 | hat                   | 2021-04-06 00:00:00 |           255 |
Clolorado
(1 row)

--查询 sales_table 的 sys_p2 分区数据。这里采用“sales_table PARTITION (sys_p2);”
来引用分区。
postgres=# SELECT * FROM sales_table PARTITION (sys_p2);
 order_no |      goods_name       |     sales_date      | sales_volume |
sales_store
```

```
---------+--------------------+--------------------+-------------+-----
---------------
       3 | shirt              | 2021-11-17 00:00:00 |        7000 | Florida
(1 row)
```

12.7 创建哈希分区表

语法格式

```
CREATE TABLE partition_table_name
( [column_name data_type ]
  [, ... ]
)
    PARTITION BY HASH (partition_key)
       (PARTITION partition_name )
       [, ... ]
   );
```

参数说明

- partition_table_name

  分区表的名称。

- column_name

  新表中要创建的字段名。

- data_type

  字段的数据类型。

- partition_key

  partition_key 为分区键的名称。哈希分区策略的分区键仅支持 1 列。

- partition_name

  partition_name 为哈希分区的名称。希望创建几个哈希分区就给出几个分区名。

示例

示例 9：哈希分区表 hash_partition_table。

```
--创建哈希分区表 hash_partition_table
postgres=# create table hash_partition_table (
col1 int,
col2 int)
```

```
partition by hash(col1)
(
partition p1,
partition p2
);

-- 数据插入
postgres=# INSERT INTO hash_partition_table VALUES(1, 1);
INSERT 0 1
postgres=# INSERT INTO hash_partition_table VALUES(2, 2);
INSERT 0 1
postgres=# INSERT INTO hash_partition_table VALUES(3, 3);
INSERT 0 1
postgres=# INSERT INTO hash_partition_table VALUES(4, 4);
INSERT 0 1

-- 查看数据
postgres=# select * from hash_partition_table partition (p1);
 col1 | col2
------+------
    3 |    3
    4 |    4
(2 rows)

postgres=# select * from hash_partition_table partition (p2);
 col1 | col2
------+------
    1 |    1
    2 |    2
(2 rows)
```

12.8 导入数据

语法格式

导入单行数据：

```
INSERT INTO partition_table_name [ ( column_name [, ...] ) ]  VALUES
[ ( value )[, ...]  ];
```

导入结构相同的现有表格数据：

```
INSERT INTO partition_table_name SELECT * FROM  source_table_name
```

参数说明

- partition_table_name

  分区表的名称。

- column_name

  分区表中的字段名。可省略。

- value

  字段对应的值：

  - 提供了 column_name 值时： value 子句提供的值从左到右关联到对应列。
  - 没提供 column_name 值时：value 子句提供的值从左到右关联到 partition_table_name 对应列。

示例

示例 10：

```
--创建分区表 employees_table。
postgres=# CREATE TABLE employees_table
(
    employee_id             INTEGER          NOT NULL,
    employee_name           CHAR(20)         NOT NULL,
    onboarding_date         DATE             NOT NULL,
    position                CHAR(20)
)
PARTITION BY RANGE(onboarding_date)
(
        PARTITION founders VALUES LESS THAN('2000-01-01 00:00:00'),
        PARTITION senate VALUES LESS THAN('2010-01-01 00:00:00'),
        PARTITION seniors VALUES LESS THAN('2020-01-01 00:00:00'),
        PARTITION newcomer VALUES LESS THAN(MAXVALUE)
);

-- 数据插入分区 founders
postgres=# INSERT INTO  employees_table VALUES(1, 'SMITH', '1997-01-10
00:00:00','Manager');

-- 查看 founders 分区数据
postgres=# select * from employees_table partition (founders);
```

```
-- 创建表格 employees_data_table
postgres=# CREATE TABLE employees_data_table
(
    employee_id             INTEGER          NOT NULL,
    employee_name           CHAR(20)         NOT NULL,
    onboarding_date         DATE             NOT NULL,
    position                CHAR(20)
);
-- 插入数据
postgres=# insert into employees_data_table (employee_id, employee_name,
onboarding_date, position) VALUES
(2, 'JONES', '2001-05-06 00:00:00', 'Supervisor'),
(3, 'WILLIAMS', '2011-09-17 00:00:00', 'Engineer'),
(4, 'TAYLOR', '2021-10-21 00:00:00', 'Clerk');

-- 查看表格数据
postgres=# select * from employees_data_table;

--数据导入 employees_table
postgres=# INSERT INTO employees_table SELECT * FROM  employees_data_table;

-- 查看 senate 分区数据
postgres=# select * from employees_table partition (senate);
 employee_id |    employee_name     |   onboarding_date   |       position
-------------+----------------------+---------------------+------------------
----
           2 | JONES                | 2001-05-06 00:00:00 | Supervisor
(1 row)

-- 查看 seniors 分区数据
postgres=# select * from employees_table partition (seniors);
 employee_id |    employee_name     |   onboarding_date   |       position
-------------+----------------------+---------------------+------------------
----
           3 | WILLIAMS             | 2011-09-17 00:00:00 | Engineer
(1 row)

-- 查看 newcomer 分区数据
postgres=# select * from employees_table partition (newcomer);
 employee_id |    employee_name     |   onboarding_date   |       position
-------------+----------------------+---------------------+------------------
----
```

```
          4 | TAYLOR               | 2021-10-21 00:00:00 | Clerk
(1 row)
```

12.9 修改分区表

语法格式

删除分区：

```
ALTER TABLE partition_table_name DROP PARTITION partition_name;
```

增加分区：

```
ALTER TABLE partition_table_name ADD {partition_less_than_item |
partition_start_end_item| partition_list_item };
```

重命名分区：

```
ALTER TABLE partition_table_name RENAME PARTITION partition_name TO
partition_new_name;
```

分裂分区（指定切割点 split_partition_value 的语法）：

```
ALTER TABLE partition_table_name SPLIT PARTITION partition_name AT
( split_partition_value ) INTO ( PARTITION partition_new_name1, PARTITION
partition_new_name2);
```

分裂分区（指定分区范围的语法）：

```
ALTER TABLE partition_table_name SPLIT PARTITION partition_name INTO
{ ( partition_less_than_item [, ...] ) | ( partition_start_end_item [, ...] ) };
```

合并分区：

```
ALTER TABLE partition_table_name MERGE PARTITIONS { partition_name } [, ...] INTO
PARTITION partition_name;
```

参数说明

- partition_table_name

  分区表的名称。

- partition_name

  partition_name 为分区的名称。

- split_partition_value

  切割点。

- PARTITION partition_new_name1, PARTITION partition_new_name2

  按照切割点分裂出的两个分区。

- partition_less_than_item

  分区项的描述语句，语法为：

```
PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE }
[, ...] )
```

  用法与创建 VALUES LESS THAN 范围分区表语法格式中相同。

- partition_start_end_item

  分区项的描述语句，语法为：

```
PARTITION partition_name {
        {START(partition_value) END (partition_value) EVERY (interval_value)} |
        {START(partition_value) END ({partition_value | MAXVALUE})} |
        {START(partition_value)} |
        {END({partition_value | MAXVALUE})}
```

  用法与创建 START END 范围分区表语法格式中相同。

- partition_list_item

  分区项的描述语句，语法为：

```
PARTITION partition_name VALUES (list_values_clause)
```

  用法与创建列表分区表中相同。

- split_point_clause

  分裂分区时，指定的切割点。

- partition_value

  分区键值。

示例

  示例 11：

```
--创建分区表 employees_table。
postgres=# CREATE TABLE employees_table
(
    employee_id             INTEGER         NOT NULL,
    employee_name           CHAR(20)        NOT NULL,
```

```
    onboarding_date          DATE             NOT NULL,
    position                 CHAR(20)
)
PARTITION BY RANGE(onboarding_date)
(
        PARTITION founders VALUES LESS THAN('2000-01-01 00:00:00'),
        PARTITION senate VALUES LESS THAN('2010-01-01 00:00:00'),
        PARTITION seniors VALUES LESS THAN('2020-01-01 00:00:00'),
        PARTITION newcomer VALUES LESS THAN(MAXVALUE)
);

-- 插入数据
postgres=# INSERT INTO  employees_table VALUES
(1, 'SMITH', '1997-01-10 00:00:00','Manager'),
(2, 'JONES', '2001-05-06 00:00:00', 'Supervisor'),
(3, 'WILLIAMS', '2011-09-17 00:00:00', 'Engineer'),
(4, 'TAYLOR', '2021-10-21 00:00:00', 'Clerk');

查看 newcomer 分区
postgres=# SELECT * FROM employees_table PARTITION (newcomer);
 employee_id |    employee_name     |   onboarding_date   |       position
-------------+----------------------+---------------------+-----------------
----
           4 | TAYLOR               | 2021-10-21 00:00:00 | Clerk
(1 row)

--删除 newcomer 分区。
postgres=# ALTER TABLE employees_table DROP PARTITION newcomer;
ALTER TABLE

-- 查看 newcomer 分区数据
postgres=# select * from employees_table partition (newcomer);
ERROR:  partition "newcomer" of relation "employees_table" does not exist

--增加 fresh 分区。
postgres=# ALTER TABLE employees_table ADD PARTITION fresh VALUES LESS THAN
('2040-01-01 00:00:00');
ALTER TABLE

--以 2030-01-01 00:00:00 为分割点，分裂 fresh 分区为 current、future 两个分区
postgres=# ALTER TABLE employees_table SPLIT PARTITION fresh AT ('2030-01-01
00:00:00') INTO (PARTITION current, PARTITION future);
```

```
ALTER TABLE

--将分区 current 改名为 now
postgres=# ALTER TABLE employees_table RENAME PARTITION current TO now;
ALTER TABLE

--将 founders, senate 合并为一个分区 original。
postgres=# ALTER TABLE  employees_table MERGE PARTITIONS founders, senate INTO
PARTITION original;
```

12.10 删除分区表

语法格式

```
DROP TABLE partition_table_name;
```

参数说明

- partition_table_name

  分区表的名称。

示例

示例 12:

```
--删除分区表 employees_table。
postgres=# DROP TABLE employees_table;
DROP TABLE
```

13 索引

索引是一个指向表中数据的指针。一个数据库中的索引与一本书的索引目录是非常相似的。

索引可以用来提高数据库查询性能，但是不恰当的使用将导致数据库性能下降。建议仅在匹配如下某条原则时创建索引：

- 经常执行查询的字段。
- 在连接条件上创建索引，对于存在多字段连接的查询，建议在这些字段上建立组合索引。例如，select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b，可以在 t1 表上的 a，b 字段上建立组合索引。
- WHERE 子句的过滤条件字段上（尤其是范围条件）。
- 经常出现在 ORDER BY、GROUP BY 和 DISTINCT 后的字段。

13.1 语法格式

单列索引

单列索引是一个只基于表的一个列上创建的索引。

```
CREATE INDEX [ [schema_name.]index_name ] ON table_name  (column_name);
```

组合索引

组合索引是基于表的多列上创建的索引。

```
CREATE INDEX [ [schema_name.]index_name ] ON table_name
(column1_name,column2_name,...);
```

唯一索引

指定唯一索引的字段不允许重复值插入。

```
CREATE  UNIQUE INDEX [ [schema_name.]index_name ] ON table_name  (column_name);
```

局部索引

在表的子集上构建索引，子集由一个条件表达式定义。

```
CREATE INDEX [ [schema_name.]index_name ] ON table_name  (expression);
```

部分索引

部分索引是一个只包含表的一部分记录的索引，通常是该表中比其他部分数据更有用的

部分。

```
CREATE INDEX [ [schema_name.]index_name ] ON table_name  (column_name)
   [ WHERE predicate ]
```

删除索引

```
DROP INDEX index_name;
```

13.2 参数说明

- UNIQUE

创建唯一性索引，每次添加数据时检测表中是否有重复值。如果插入或更新的值会引起重复的记录时，将导致一个错误。

目前只有 B-tree 索引支持唯一索引。

- schema_name

模式的名称。

取值范围：已存在模式名。

- index_name

要创建的索引名，索引的模式与表相同。

取值范围：字符串，要符合标识符的命名规范。

- table_name

需要为其创建索引的表的名称，可以用模式修饰。

取值范围：已存在的表名。

- column_name

表中需要创建索引的列的名称（字段名）。

如果索引方式支持多字段索引，可以声明多个字段。全局索引最多可以声明 31 个字段，其他索引最多可以声明 32 个字段。

- expression

创建一个基于该表的一个或多个字段的表达式索引，通常必须写在圆括弧中。如果表达式有函数调用的形式，圆括弧可以省略。

表达式索引可用于获取对基本数据的某种变形的快速访问。比如，一个在 upper(col)上

的函数索引将允许 WHERE upper(col) = ’JIM’子句使用索引。

在创建表达式索引时，如果表达式中包含 IS NULL 子句，则这种索引是无效的。此时，建议用户尝试创建一个部分索引。

- WHERE predicate

创建一个部分索引。部分索引是一个只包含表的一部分记录的索引，通常是该表中比其他部分数据更有用的部分。例如，有一个表，表里包含已记账和未记账的定单，未记账的定单只占表的一小部分而且这部分是最常用的部分，此时就可以通过只在未记账部分创建一个索引来改善性能。另外一个可能的用途是使用带有 UNIQUE 的 WHERE 强制一个表的某个子集的唯一性。

取值范围：predicate 表达式只能引用表的字段，它可以使用所有字段，而不仅是被索引的字段。目前，子查询和聚集表达式不能出现在 WHERE 子句里。

13.3 示例

创建表 tpcds.ship_mode_t1。

```
postgres=# CREATE SCHEMA tpcds;
postgres=# CREATE TABLE tpcds.ship_mode_t1
(
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
) ;
```

在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_ID 字段上创建单列索引。

```
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index0 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_ID);
```

在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建普通的唯一索引。

```
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
```

在表 tpcds.ship_mode_t1 上 SM_CODE 字段上创建表达式索引。

```
postgres=# CREATE INDEX ds_ship_mode_t1_index2 ON
tpcds.ship_mode_t1(SUBSTR(SM_CODE,1 ,4));
```

在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建 SM_SHIP_MODE_SK 大于 10 的部分索引。

```
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index3 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK) WHERE SM_SHIP_MODE_SK>10;
```

删除已创建索引。

```
postgres=# DROP INDEX tpcds.ds_ship_mode_t1_index2;
```

14 约束

约束子句用于声明约束，新行或者更新的行必须满足这些约束才能成功插入或更新。如果存在违反约束的数据行为，行为会被约束终止。

约束可以在创建表时规定（通过 CREATE TABLE 语句），或者在表创建之后规定（通过 ALTER TABLE 语句）。

约束可以是列级或表级。列级约束仅适用于列，表级约束被应用到整个表。

GBase 8s 中常用的约束如下：

- NOT NULL：指示某列不能存储 NULL 值。
- UNIQUE：确保某列的值都是唯一的。
- PRIMARY KEY：NOT NULL 和 UNIQUE 的结合。确保某列（或两个列多个列的结合）有唯一标识，有助于更容易更快速地找到表中的一个特定的记录。
- FOREIGN KEY：保证一个表中的数据匹配另一个表中的值的参照完整性。
- CHECK：保证列中的值符合指定的条件。

14.1 NOT NULL 约束

创建表时，如果不指定约束，默认值为 NULL，即允许列插入空值。如果您不想某列存在 NULL 值，那么需要在该列上定义 NOT NULL 约束，指定在该列上的值不允许存在 NULL 值。插入数据时，如果该列存在 NULL 值，则会报错，插入失败。

NULL 与没有数据是不一样的，它代表着未知的数据。

例如，创建表 staff，共有 5 个字段，其中 NAME，ID 设置不接受空值。

```
postgres=# CREATE TABLE staff(
   ID              INT       NOT NULL,
   NAME            char(8)     NOT NULL,
   AGE             INT      ,
   ADDRESS         CHAR(50),
   SALARY          REAL
);
```

给表 staff 插入数据。当 ID 字段插入空值时，数据库返回报错。

```
postgres=# INSERT INTO staff  VALUES (1,'lily',28);
INSERT 0 1
```

```
postgres=# INSERT INTO staff (NAME,AGE) VALUES ('JUCE',28);
ERROR:  null value in column "id" violates not-null constraint
DETAIL:  Failing row contains (null, JUCE    , 28, null, null).
```

14.2 UNIQUE 约束

UNIQUE 约束表示表里的一个字段或多个字段的组合必须在全表范围内唯一。

对于唯一约束，NULL 被认为是互不相等的。

例如，创建表 staff1，表包含 5 个字段，其中 AGE 设置为 UNIQUE，因此不能添加两条有相同年龄的记录。

```
postgres=# CREATE TABLE staff1(
   ID              INT       NOT NULL,
   NAME            char(8)     NOT NULL,
   AGE             INT    NOT NULL  UNIQUE  ,
   ADDRESS         CHAR(50),
   SALARY          REAL
);
```

给表 staff1 表插入数据。当字段 AGE 插入两条一样的数据时，数据库返回报错。

```
postgres=# INSERT INTO staff1  VALUES (1,'lily',28);
INSERT 0 1
postgres=# INSERT INTO staff1 VALUES (2, 'JUCE',28);
ERROR:  duplicate key value violates unique constraint "staff1_age_key"
DETAIL:  Key (age)=(28) already exists.
```

14.3 PRIMARY KEY

PRIMARY KEY 为主键，是数据表中每一条记录的唯一标识。主键约束声明表中的一个或者多个字段只能包含唯一的非 NULL 值。

主键是非空约束和唯一约束的组合。一个表只能声明一个主键。

例如，创建表 staff2，其中 ID 为主键。

```
postgres=# CREATE TABLE staff2(
   ID              INT      PRIMARY KEY     ,
   NAME            TEXT     NOT NULL,
   AGE             INT      NOT NULL,
   ADDRESS         CHAR(50),
   SALARY          REAL
);
```

```
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "staff2_pkey" for
table "staff2"
CREATE TABLE
```

14.4 FOREIGN KEY

FOREIGN KEY 即外键约束，指定列(或一组列)中的值必须匹配另一个表的某一行中出现的值。通常一个表中的FOREIGN KEY 指向另一个表中的 UNIQUE KEY(唯一约束的键)，即维护了两个相关表之间的引用完整性。

例如，创建表 staff3，包含 5 个字段。

```
postgres=# CREATE TABLE staff3(
   ID              INT    PRIMARY KEY  NOT NULL,
   NAME            TEXT    NOT NULL,
   AGE             INT     NOT NULL,
   ADDRESS         CHAR(50),
   SALARY          REAL
);
```

创建一张 DEPARTMENT 表，并添加 3 个字段，其中 EMP_ID 为外键，参照 staff3 的 ID 字段：

```
postgres=# CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT            CHAR(50) NOT NULL,
   EMP_ID          INT      references staff3(ID)
);
```

14.5 CHECK 约束

CHECK 约束声明一个布尔表达式，每次要插入的新行或者要更新的行的新值必须使表达式结果为真或未知才能成功，否则会抛出一个异常并且不会修改数据库。

声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以引用多个字段。expression 表达式中，如果存在“<>NULL”或“！=NULL”，这种写法是无效的，需要写成“is NOT NULL”。

例如，创建表 staff4，对字段 SALARY 新增 CHECK 约束，确保插入此列数值大于 0。

```
postgres=# CREATE TABLE staff4(
   ID INT PRIMARY KEY     NOT NULL,
   NAME            TEXT    NOT NULL,
   AGE             INT     NOT NULL,
```

```
   ADDRESS        CHAR(50),
   SALARY         REAL     CHECK(SALARY > 0)
);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "staff4_pkey" for
table "staff4"
CREATE TABLE
```

给表 staff4 插入数据。当字段 SALARY 插入数据不大于 0 时，数据库返回报错。

```
postgres=# INSERT INTO staff4(ID,NAME,AGE,SALARY) VALUES (2, 'JUCE',16,0);
ERROR:  new row for relation "staff4" violates check constraint
"staff4_salary_check"
DETAIL:  N/A
```

15 物化视图

物化视图是一种特殊的物理表，物化视图是相对普通视图而言的。普通视图是虚拟表，应用的局限性较大，任何对视图的查询实际上都是转换为对 SQL 语句的查询，性能并没有实际上提高。物化视图实际上就是存储 SQL 执行语句的结果，起到缓存的效果。

目前 Ustore 引擎不支持创建、使用物化视图。

15.1 全量物化视图

15.1.1 概述

全量物化视图仅支持对已创建的物化视图进行全量更新，而不支持进行增量更新。创建全量物化视图语法和 CREATE TABLE AS 语法类似。

15.1.2 使用

语法格式

创建全量物化视图

```
CREATE MATERIALIZED VIEW [ view_name ] AS { query_block };
```

全量刷新物化视图

```
REFRESH MATERIALIZED VIEW [ view_name ];
```

删除物化视图

```
DROP MATERIALIZED VIEW [ view_name ];
```

查询物化视图

```
SELECT * FROM [ view_name ];
```

示例

```
--准备数据。
postgres=# CREATE TABLE t1(c1 int, c2 int);
postgres=# INSERT INTO t1 VALUES(1, 1);
postgres=# INSERT INTO t1 VALUES(2, 2);

--创建全量物化视图。
postgres=# CREATE MATERIALIZED VIEW mv AS select count(*) from t1;
CREATE MATERIALIZED VIEW
```

```
--查询物化视图结果。
postgres=# SELECT * FROM mv;
 count
-------
     2
(1 row)

--向物化视图中基表插入数据。
postgres=# INSERT INTO t1 VALUES(3, 3);

--对全量物化视图做全量刷新。
postgres=# REFRESH MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

--查询物化视图结果。
postgres=# SELECT * FROM mv;
 count
-------
     3
(1 row)

--删除物化视图。
postgres=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
```

15.1.3 支持和约束

支持场景

通常全量物化视图所支持的查询范围与 CREATE TABLE AS 语句一致。

全量物化视图上支持创建索引。

支持 analyze、explain。

不支持场景

物化视图不支持增删改操作，只支持查询语句。

约束

全量物化视图的刷新、删除过程中会给基表加高级别锁，若物化视图的定义涉及多张表，需要注意业务逻辑，避免死锁产生。

15.2 增量物化视图

15.2.1 概述

增量物化视图可以对物化视图增量刷新，需要用户手动执行语句完成对物化视图在一段时间内的增量数据刷新。与全量创建物化视图的不同在于目前增量物化视图所支持场景较小。目前物化视图创建语句仅支持基表扫描语句或者 UNION ALL 语句。

15.2.2 使用

语法格式

创建增量物化视图

```
CREATE INCREMENTAL MATERIALIZED VIEW [ view_name ] AS { query_block };
```

全量刷新物化视图

```
REFRESH MATERIALIZED VIEW [ view_name ];
```

增量刷新物化视图

```
REFRESH INCREMENTAL MATERIALIZED VIEW [ view_name ];
```

删除物化视图

```
DROP MATERIALIZED VIEW [ view_name ];
```

查询物化视图

```
SELECT * FROM [ view_name ];
```

示例

```
--准备数据。
postgres=# CREATE TABLE t1(c1 int, c2 int);
postgres=# INSERT INTO t1 VALUES(1, 1);
postgres=# INSERT INTO t1 VALUES(2, 2);

--创建增量物化视图。
postgres=# CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT * FROM t1;
CREATE MATERIALIZED VIEW

--插入数据。
postgres=# INSERT INTO t1 VALUES(3, 3);
INSERT 0 1
```

```
--增量刷新物化视图。
postgres=# REFRESH INCREMENTAL MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

--查询物化视图结果。
postgres=# SELECT * FROM mv;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
(3 rows)

--插入数据。
postgres=# INSERT INTO t1 VALUES(4, 4);
INSERT 0 1

--全量刷新物化视图。
postgres=# REFRESH MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

--查询物化视图结果。
postgres=# select * from mv;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)

--删除物化视图。
postgres=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
```

15.2.3 支持和约束

支持场景

单表查询语句。

多个单表查询的 UNION ALL。

物化视图上支持创建索引。

物化视图支持 Analyze 操作。

不支持场景

物化视图中不支持多表 Join 连接计划以及 subquery 计划。

除少部分 ALTER 操作外，不支持对物化视图中基表执行绝大多数 DDL 操作。

物化视图不支持增删改操作，只支持查询语句。

不支持用临时表/hashbucket/unlog/分区表创建物化视图。

不支持物化视图嵌套创建（即物化视图上创建物化视图）。

仅支持行存表，不支持列存表。

不支持 UNLOGGED 类型的物化视图，不支持 WITH 语法。

约束

物化视图定义如果为 UNION ALL，则其中每个子查询需使用不同的基表。

增量物化视图的创建、全量刷新、删除过程中会给基表加高级别锁，若物化视图的定义为 UNION ALL，需要注意业务逻辑，避免死锁产生。

16 游标

为了处理 SQL 语句，存储过程进程分配一段内存区域来保存上下文联系。游标是指向上下文区域的句柄或指针。借助游标，存储过程可以控制上下文区域的变化。

16.1 语法

定义游标

```
CURSOR cursor_name
    [ BINARY ]  [ NO SCROLL ]  [ { WITH | WITHOUT } HOLD ]
    FOR query ;
```

通过已经创建的游标检索数据

```
FETCH [ direction { FROM | IN } ] cursor_name;
```

其中 direction 子句为可选参数。

```
NEXT
    | PRIOR
    | FIRST
    | LAST
    | ABSOLUTE count
    | RELATIVE count
    | count
    | ALL
    | FORWARD
    | FORWARD count
    | FORWARD ALL
    | BACKWARD
    | BACKWARD count
    | BACKWARD ALL
```

在不检索数据的情况下重新定位一个游标

MOVE 的作用类似于 FETCH 命令，但只是重定位游标而不返回行。

```
MOVE [ direction [ FROM | IN ] ] cursor_name;
```

其中 direction 子句为可选参数。

```
NEXT
    | PRIOR
    | FIRST
    | LAST
```

```
| ABSOLUTE count
| RELATIVE count
| count
| ALL
| FORWARD
| FORWARD count
| FORWARD ALL
| BACKWARD
| BACKWARD count
| BACKWARD ALL
```

关闭游标，释放和一个游标关联的所有资源

```
CLOSE { cursor_name | ALL } ;
```

16.2 参数说明

- cursor_name

  将要创建、关闭的游标名。

- BINARY

  指明游标以二进制而不是文本格式返回数据。

- NO SCROLL

  声明游标检索数据行的方式。

  - NO SCROLL：声明该游标不能用于以倒序的方式检索数据行。
  - 未声明：根据执行计划的不同，自动判断该游标是否可以用于以倒序的方式检索数据行。

- WITH HOLD | WITHOUT HOLD

  声明当创建游标的事务结束后，游标是否能继续使用。

  - WITH HOLD：声明该游标在创建它的事务结束后仍可继续使用。
  - WITHOUT HOLD：声明该游标在创建它的事务之外不能再继续使用，此游标将在事务结束时被自动关闭。
  - 如果不指定 WITH HOLD 或 WITHOUT HOLD，默认行为是 WITHOUT HOLD。
  - 跨节点事务不支持 WITH HOLD(例如在多 DBnode 部署 GBase 8s 中所创建的含有 DDL 的事务属于跨节点事务）。

- query

  使用 SELECT 或 VALUES 子句指定游标返回的行。

  取值范围：SELECT 或 VALUES 子句。

- direction_clause

  定义抓取数据的方向。

  取值范围：

  - NEXT（缺省值）

  从当前关联位置开始，抓取下一行。

  - PRIOR

  从当前关联位置开始，抓取上一行。

  - FIRST

  抓取查询的第一行（和 ABSOLUTE 1 相同）。

  - LAST

  抓取查询的最后一行（和 ABSOLUTE -1 相同）。

  - ABSOLUTE count

  抓取查询中第 count 行。

  ABSOLUTE 抓取不会比用相对位移移动到需要的数据行更快，因为下层的实现必须遍历所有中间的行。

  count 取值范围：有符号的整数

  - count 为正数，就从查询结果的第一行开始，抓取第 count 行。
  - count 为负数，就从查询结果末尾抓取第 abs(count)行。
  - count 为 0 时，定位在第一行之前。

- RELATIVE count

  从当前关联位置开始，抓取随后或前面的第 count 行。

  取值范围：有符号的整数

  - count 为正数就抓取当前关联位置之后的第 count 行。

- count 为负数就抓取当前关联位置之前的第 abs(count)行。
- 如果当前行没有数据的话，RELATIVE 0 返回空。

- count

  抓取随后的 count 行（和 FORWARD count 一样）。

- ALL

  从当前关联位置开始，抓取所有剩余的行（和 FORWARD ALL 一样）。

- FORWARD

  抓取下一行（和 NEXT 一样）。

- FORWARD count

  从当前关联位置开始，抓取随后或前面的 count 行。

- FORWARD ALL

  从当前关联位置开始，抓取所有剩余行。

- BACKWARD

  从当前关联位置开始，抓取前面一行(和 PRIOR 一样) 。

- BACKWARD count

  从当前关联位置开始，抓取前面的 count 行（向后扫描）。

  取值范围：有符号的整数

  - count 为正数就抓取当前关联位置之前的 count 行。
  - count 为负数就抓取当前关联位置之后的 abs（count）行。
  - 如果有数据的话，BACKWARD 0 重新抓取当前行。

- BACKWARD ALL

  从当前关联位置开始，抓取所有前面的行（向后扫描） 。

- { FROM | IN } cursor_name

  使用关键字 FROM 或 IN 指定游标名称。

  取值范围：已创建的游标的名称。

- ALL

关闭所有已打开的游标。

16.3 示例

假设存在表 customer_t1 ，数据内容如下：

```
postgres=# SELECT * FROM customer_t1;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          3769 |               | Grace        |             |
          3769 | hello         |              |             |
          6885 | maps          | Joes         |             |   2200
          4321 | tpcds         | Lily         |             |   3000
          9527 | world         | James        |             |   5000
(5 rows)
```

用一个游标读取一个表。

```
--开始一个事务。
postgres=# START TRANSACTION;
START TRANSACTION

--建立一个名为 cursor1 的游标。
postgres=# CURSOR cursor1 FOR SELECT * FROM customer_t1;
DECLARE CURSOR

--抓取前 3 行到游标 cursor1 里。
postgres=# FETCH FORWARD 3 FROM cursor1;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          3769 |               | Grace        |             |
          3769 | hello         |              |             |
          6885 | maps          | Joes         |             |   2200
(3 rows)

--关闭游标并提交事务。
postgres=# CLOSE cursor1;
CLOSE CURSOR

--结束一个事务。
postgres=# END;
COMMIT
```

用一个游标读取 VALUES 子句中的内容。

```
--开始一个事务。
postgres=# START TRANSACTION;
START TRANSACTION

--建立一个名为 cursor2 的游标。
postgres=# CURSOR cursor2 FOR VALUES(1,2),(0,3) ORDER BY 1;
DECLARE CURSOR

--抓取前 2 行到游标 cursor2 里。
postgres=# FETCH FORWARD 2 FROM cursor2;
 column1 | column2
---------+---------
       0 |       3
       1 |       2
(2 rows)

--关闭游标并提交事务。
postgres=# CLOSE cursor2;
CLOSE CURSOR

--结束一个事务。
postgres=# END;
COMMIT
```

WITH HOLD 游标的使用。

```
--开启事务。
postgres=# START TRANSACTION;

--创建一个 with hold 游标。
postgres=# DECLARE cursor1 CURSOR WITH HOLD FOR SELECT * FROM customer_t1;

--抓取头 2 行到游标 cursor1 里。
postgres=# FETCH FORWARD 2 FROM cursor1;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          3769 |               | Grace        |             |
          3769 | hello         |              |             |
(2 rows)

--结束事务。
postgres=# END;
COMMIT
```

```
--抓取下一行到游标 cursor1 里。
postgres=# FETCH FORWARD 1 FROM cursor1;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          6885 | maps          | Joes         |             |   2200
(1 row)

--关闭游标。
postgres=# CLOSE cursor1;
CLOSE CURSOR
```

MOVE 语句的使用。

```
--开始一个事务。
postgres=# START TRANSACTION;
START TRANSACTION

--定义一个名为 cursor1 的游标。
postgres=# CURSOR cursor1 FOR SELECT * FROM customer_t1;
DECLARE CURSOR

--忽略游标 cursor1 的前 3 行。
postgres=# MOVE FORWARD 1 FROM cursor1;
MOVE 1

--抓取游标 cursor1 的前 2 行。
postgres=# FETCH 2 FROM cursor1;
 c_customer_sk | c_customer_id | c_first_name | c_last_name | amount
---------------+---------------+--------------+-------------+--------
          3769 | hello         |              |             |
          6885 | maps          | Joes         |             |   2200
(2 rows)


--关闭游标。
postgres=# CLOSE cursor1;
CLOSE CURSOR

--结束一个事务。
postgres=# END;
COMMIT
```

17 匿名块

匿名块（Anonymous Block）是存储过程的字块之一，没有名称。一般用于不频繁执行的脚本或不重复进行的活动。

17.1 语法

匿名块的语法参见下图。

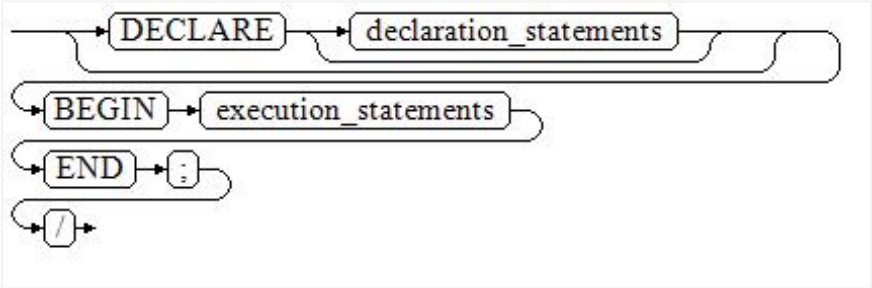

图 17-1 anonymous_block::=

对以上语法图的解释如下：

匿名块程序实施部分，以 BEGIN 语句开始，以 END 语句停顿，以一个分号结束。输入“/”按回车执行它。

须知

- 最后的结束符“/”必须独占一行，不能直接跟在 END 后面。

声明部分包括变量定义、类型、游标定义等。

最简单的匿名块不执行任何命令。但一定要在任意实施块里至少有一个语句，甚至是一个 NULL 语句。

17.2 参数说明

- DECLARE

用于开始 DECLARE 语句的可选关键字，此关键字可用于声明数据类型、变量或游标。此关键字的使用取决于此块所在的上下文。

- declaration_statements

指定作用域限定于块的数据类型、变量、游标、异常或过程声明。每个声明都必须以分号终止。

- BEGIN

用于引入可执行节的必需关键字，该节可以包含一个或多个 SQL 或 PL/SQL 语句。BEGIN-END 块可以包含嵌套的 BEGIN-END 块。

- execution_statements

指定 PL/SQL 或 SQL 语句。每个语句都必须以分号终止。

- END

用于结束块的必需关键字。

17.3 示例

```
--空语句块
postgres=# BEGIN
     NULL;
END;
/

--创建演示表格:
postgres=# CREATE TABLE table1(id1 INT, id2 INT, id3 INT);
CREATE TABLE

--使用匿名块插入数据:
postgres=# BEGIN
             insert into table1 values(1,2,3);
            END;
            /
ANONYMOUS BLOCK EXECUTE

--查询插入数据:
postgres=# select * from table1;
 id1 | id2 | id3
-----+-----+-----
   1 |   2 |   3
(1 rows)
```

18 存储过程

18.1 存储过程概述

商业规则和业务逻辑可以通过程序存储在 GBase 8s 中，这个程序就是存储过程。

存储过程是 SQL 和 PL/SQL 的组合。存储过程使执行商业规则的代码可以从应用程序中移动到数据库。从而，代码存储一次能够被多个程序使用。

存储过程的创建及调用办法请参考 CREATE PROCEDURE。

PL/pgSQL 语言函数节所提到的 PL/pgSQL 语言创建的函数与存储过程的应用方法相通。下面各节中，除非特别声明，否则内容通用于存储过程和 PL/pgSQL 语言函数。

18.2 数据类型

数据类型是一组值的集合以及定义在这个值集上的一组操作。GBase 8s 数据库是由表的集合组成的，而各表中的列定义了该表，每一列都属于一种数据类型，GBase 8s 根据数据类型有相应函数对其内容进行操作，例如 GBase 8s 可对数值型数据进行加、减、乘、除操作。

18.3 数据类型转换

数据库中允许有些数据类型进行隐式类型转换（赋值、函数调用的参数等），有些数据类型间不允许进行隐式数据类型转换，可尝试使用 GBase 8s 提供的类型转换函数，例如 CAST 进行数据类型强转。

GBase 8s 数据库常见的隐式类型转换，请参见表 18-1。

须知

- GBase 8s 支持的 DATE 的效限范围是：公元前 4713 年到公元 294276 年。

表 18-1 隐式类型转换表

原始数据类型	目标数据类型	备注
CHAR	VARCHAR2	-
CHAR	NUMBER	原数据必须由数字组成。

CHAR	DATE	原数据不能超出合法日期范围。
CHAR	RAW	-
CHAR	CLOB	-
VARCHAR2	CHAR	-
VARCHAR2	NUMBER	原数据必须由数字组成。
VARCHAR2	DATE	原数据不能超出合法日期范围。
VARCHAR2	CLOB	-
NUMBER	CHAR	-
NUMBER	VARCHAR2	-
DATE	CHAR	-
DATE	VARCHAR2	-
RAW	CHAR	-
RAW	VARCHAR2	-
CLOB	CHAR	-
CLOB	VARCHAR2	-
CLOB	NUMBER	原数据必须由数字组成。
INT4	CHAR	-
INT4	BOOLEAN	-
BOOLEAN	INT4	-

18.4 数组和 record

18.4.1 数组

数组类型的使用

在使用数组之前，需要自定义一个数组类型。

在存储过程中紧跟 AS 关键字后面定义数组类型。定义方法如下。

TYPE array_type IS VARRAY(size) OF data_type;

其中：

- array_type：要定义的数组类型名。
- VARRAY：表示要定义的数组类型。
- size：取值为正整数，表示可以容纳的成员的最大数量。
- data_type：要创建的数组中成员的类型。

说明

➢ 在 GBase 8s 中，数组会自动增长，访问越界会返回一个 NULL，不会报错。

在存储过程中定义的数组类型，其作用域仅在该存储过程中。

建议选择上述定义方法的一种来自定义数组类型，当同时使用两种方法定义同名的数组类型时，GBase 8s 会优先选择存储过程中定义的数组类型来声明数组变量。

data_type 也可以为存储过程中定义的 record 类型（匿名块不支持），但不可以为存储过程中定义的数组或集合类型。

GBase 8s 支持使用圆括号来访问数组元素，且还支持一些特有的函数，如 extend、count、first、last、prior、exists、 trim、next、delete 来访问数组的内容。

说明

➢ 存储过程中如果有 DML 语句（SELECT、UPDATE、INSERT、DELETE），DML 语句推荐使用中括号来访问数组元素，使用小括号默认识别为数组访问，若数组不存在，则识别为函数表达式。

➢ 如果 clob 类型大于 1GB，则存储过程中的 table of 类型、record 类型、clob 作为出入参、游标、raise info 等功能不支持。

18.4.2 集合

集合类型的使用

在使用集合之前，需要自定义一个集合类型。

在存储过程中紧跟 AS 关键字后面定义集合类型。定义方法如下。

其中：

- table_type：要定义的集合类型名。
- TABLE：表示要定义集合类型。
- data_type：要创建的集合中成员的类型。
- indexby_type：创建集合索引的类型。

说明

➢ 在 GBase 8s 中，集合会自动增长，访问越界会返回一个 NULL，不会报错。

在存储过程中定义的集合类型，其作用域仅在该存储过程中。

索引的类型仅支持 integer 和 varchar 类型，其中 varchar 的长度暂不约束。

NOT NULL 只支持语法不支持功能。

data_type 可以为存储过程内定义的 record 类型，集合类型（匿名块不支持），不可以为数组类型。

不支持跨 package 的嵌套集合类型变量使用。

不支持 record 嵌套 table of index by 类型的变量作为存储过程的出入参。

不支持 table of index by 类型的变量作为函数的出入参。

不支持通过 raise info 打印整个嵌套 table of 变量。

不支持跨自治事务传递 table of 变量。

不支持存储过程的出入参定义为嵌套 table of 类型。

GBase 8s 支持使用圆括号来访问集合元素，且还支持一些特有的函数，如 extend，count，first，last, prior, next, delete 来访问集合的内容。

集合函数支持 multiset union/intersect/except all/distinct 函数。

说明

- 同一个表达式里不支持两个以上 table of index by 类型变量的函数调用。

18.4.3 record

record 类型的变量

创建一个 record 变量的方式：

定义一个 record 类型，然后使用该类型来声明一个变量。

语法

record 类型的语法参见下图。

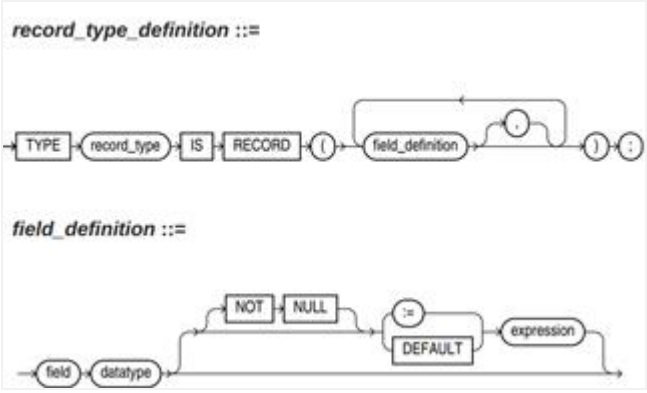

图 18-1 record 类型的语法

对以上语法格式的解释如下：

- record_type：声明的类型名称。
- field：record 类型中的成员名称。
- datatype：record 类型中成员的类型。
- expression：设置默认值的表达式。

说明

在 GBase 8s 中，record 类型变量的赋值支持：

- 在函数或存储过程的声明阶段，声明一个 record 类型，并且可以在该类型中定义成员变量。
- 一个 record 变量到另一个 record 变量的赋值。
- SELECT INTO 和 FETCH 向一个 record 类型的变量中赋值。
- 将一个 NULL 值赋值给一个 record 变量。

- 不支持 INSERT 和 UPDATE 语句使用 record 变量进行插入数据和更新数据。
- 如果成员有复合类型，在声明阶段不支持指定默认值，该行为同声明阶段的变量一样。
- datatype 可以为存储过程中定义 record 类型、数组类型和集合类型(匿名块不支持)。

示例

下面示例中用到的表定义如下：

```
postgres=# \d emp_rec
Table "public.emp_rec"
Column  |  Type     | Modifiers
---------+-----------------------------+-----------
empno     | numeric(4,0)     | not null ename      | character varying(10)
   |
job     | character varying(9)   |
mgr         | numeric(4,0)       | hiredate | timestamp(0) without time zone | sal
   | numeric(7,2)   |
comm    | numeric(7,2)   |
deptno    | numeric(2,0)     |
--演示在函数中对数组进行操作。
postgres=# CREATE OR REPLACE FUNCTION regress_record(p_w VARCHAR2) RETURNS;
VARCHAR2 AS $$ DECLARE
--声明一个 record 类型.
type rec_type is record (name varchar2(100), epno int); employer rec_type;
--使用%type 声明 record 类型
type rec_type1 is record (name emp_rec.ename%type, epno int not null :=10);
employer1 rec_type1;
--声明带有默认值的 record 类型 type rec_type2 is record (
name varchar2 not null := 'SCOTT', epno int not null :=10);
employer2 rec_type2;
CURSOR C1 IS select ename,empno from emp_rec order by 1 limit 1;
BEGIN
--对一个 record 类型的变量的成员赋值。employer.name     :=      'WARD';
employer.epno = 18;
raise info 'employer name: % , epno:%', employer.name, employer.epno;
--将一个 record 类型的变量赋值给另一个变量。employer1 := employer;
raise info 'employer1 name: % , epno: %',employer1.name, employer1.epno;
--将一个 record 类型变量赋值为 NULL。employer1 := NULL;
raise info 'employer1 name: % , epno: %',employer1.name, employer1.epno;
--获取 record 变量的默认值。
```

```
raise info 'employer2 name: % ,epno: %', employer2.name, employer2.epno;
--在 for 循环中使用 record 变量
for employer in select ename,empno from emp_rec order by 1 limit 1 loop
raise info 'employer name: % , epno: %', employer.name, employer.epno; end loop;
--在 select into 中使用 record 变量。
select ename,empno  into employer2 from emp_rec order by 1 limit 1;  raise info
'employer name: % , epno: %', employer2.name, employer2.epno;
--在 cursor 中使用 record 变量。OPEN C1;
FETCH C1 INTO employer2;
raise info 'employer name: % , epno: %', employer2.name, employer2.epno; CLOSE
C1;
RETURN employer.name; END;
$$
LANGUAGE plpgsql;
--调用该函数。
postgres=# CALL regress_record('abc');
--删除函数。
postgres=# DROP FUNCTION regress_record;
```

18.5 声明语法

18.5.1 基本结构

18.5.1.1 结构

PL/SQL 块中可以包含子块，子块可以位于 PL/SQL 中任何部分。PL/SQL 块的结构如下：

声明部分：声明 PL/SQL 用到的变量、类型、游标、局部的存储过程和函数。

```
DECLARE
```

说明

- 不涉及变量声明时声明部分可以没有。
- 对匿名块来说，没有变量声明部分时，可以省去 DECLARE 关键字。
- 对存储过程来说，没有 DECLARE， AS 相当于 DECLARE。即便没有变量声明的部分，关键字 AS 也必须保留。

执行部分：过程及 SQL 语句，程序的主要部分。必选。

```
BEGIN
```

执行异常部分：错误处理。可选。

```
EXCEPTION
```

结束

```
END;
/
```

须知

- 禁止在 PL/SQL 块中使用连续的 Tab，连续的 Tab 可能会造成在使用 gsql 工具带“-r”参数执行 PL/SQL 块时出现异常。

18.5.1.2 分类

PL/SQL 块可以分为以下几类：

匿名块：动态构造，只能执行一次。语法请参考图 17-2。

子程序：存储在数据库中的存储过程、函数、操作符和高级包等。当在数据库上建立好后，可以在其他程序中调用它们。

18.5.2 匿名块

匿名块（Anonymous Block）一般用于不频繁执行的脚本或不重复进行的活动。它们在一个会话中执行，并不被存储。

语法

匿名块的语法参见下图。

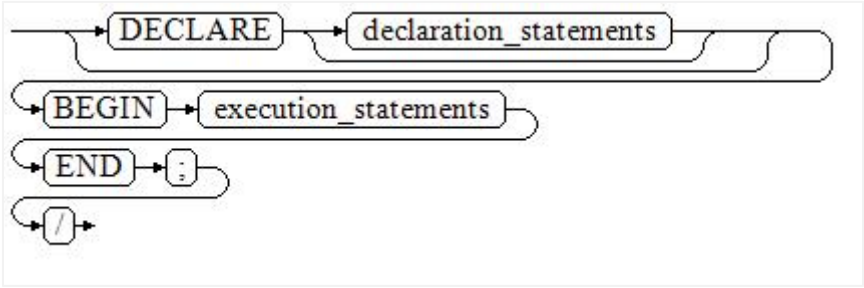

图 18-2 anonymous_block::=语法

对以上语法图的解释如下：

匿名块程序实施部分，以 BEGIN 语句开始，以 END 语句停顿，以一个分号结束。输入“/”按回车执行它。

须知

- 最后的结束符“/”必须独占一行，不能直接跟在 END 后面。

声明部分包括变量定义、类型、游标定义等。

最简单的匿名块不执行任何命令。但一定要在任意实施块里至少有一个语句，甚至是一个 NULL 语句。

18.5.3 子程序

存储在数据库中的存储过程、函数、操作符和高级包等。当在数据库上建立好后，可以在其他程序中调用它们。

18.6 基本语句

在编写 PL/SQL 过程中，会定义一些变量，给变量赋值，调用其他存储过程等。介绍 PL/SQL 中的基本语句，包括定义变量、赋值语句、调用语句以及返回语句。

说明

- 尽量不要在存储过程中调用包含密码的 SQL 语句，因为存储在数据库中的存储过程文本可能被其他有权限的用户看到导致密码信息被泄漏。如果存储过程中包含其他敏感信息也需要配置存储过程的访问权限，保证敏感信息不会泄漏。

18.6.1 定义变量

介绍 PL/SQL 中变量的声明，以及该变量在代码中的作用域。

18.6.1.1 变量声明

变量声明语法请参见下图。

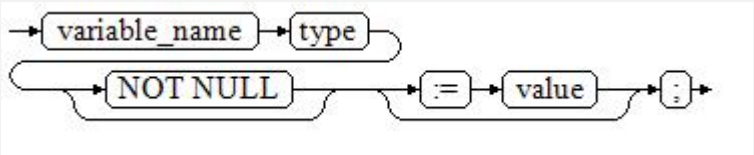

图 18-3 declare_variable::=

对以上语法格式的解释如下：

- variable_name：变量名。

- type：变量类型。
- value：该变量的初始值（如果不给定初始值，则初始为 NULL）。value 也可以是表达式。

示例

```
postgres=# DECLARE
    emp_id  INTEGER := 7788;  --定义变量并赋值
BEGIN
    emp_id := 5*7784;  --变量赋值
END;
/
```

变量类型除了支持基本类型，还可以是使用%TYPE 和%ROWTYPE 去声明一些与其他表字段或表结构本身相关的变量。

%TYPE 属性

%TYPE 主要用于声明某个与其他变量类型（例如，表中某列的类型）相同的变量。假如我们想定义一个 my_name 变量，它的变量类型与 employee 的 firstname 类型相同，我们可以通过如下定义：

```
my_name employee.firstname%TYPE
```

这样定义可以带来两个好处，首先，我们不用预先知道 employee 表的 firstname 类型具体是什么。其次，即使之后 firstname 类型有了变化，我们也不需要再次修改 my_name 的类型。

```
TYPE employee_record is record (id INTEGER, firstname VARCHAR2(20));
my_employee employee_record;
my_id my_employee.id%TYPE;
my_id_copy my_id%TYPE;
```

%ROWTYPE 属性

%ROWTYPE 属性主要用于对一组数据的类型声明，用于存储表中的一行数据或从游标匹配的结果。假如，我们需要一组数据，该组数据的字段名称与字段类型都与 employee 表相同。我们可以通过如下定义：

```
my_employee employee%ROWTYPE
```

同样可以使用在 cursor 上面，该组数据的字段名称与字段类型都与 employee 表相同（对于 PACKAGE 中的 cursor，可以省略%ROWTYPE）。%TYPE 也可以引用 cursor 中某一列的类型，我们可以通过如下定义：

```
cursor cur is select * from employee;
my_employee cur%ROWTYPE
my_name cur.firstname%TYPE
my_employee2 cur -- 对于 PACKAGE 中定义的 cursor，可以省略%ROWTYPE 字段
```

须知

- TYPE 不支持引用复合类型或 RECORD 类型变量的类型、RECORD 类型的某列类型、跨 PACKAGE 复合类型变量的某列类型、跨 PACKAGE cursor 变量的某列类型等。
- ROWTYPE 不支持引用复合类型或 RECORD 类型变量的类型、跨 PACKAGE cursor 的类型。

18.6.1.2 变量作用域

变量的作用域表示变量在代码块中的可访问性和可用性。只有在它的作用域内，变量才有效。

变量必须在 declare 部分声明，即必须建立 BEGIN-END 块。块结构也强制变量必须先声明后使用，即变量在过程内有不同作用域、不同的生存期。

同一变量可以在不同的作用域内定义多次，内层的定义会覆盖外层的定义。

在外部块定义的变量，可以在嵌套块中使用。但外部块不能访问嵌套块中的变量。

18.6.2 赋值语句

18.6.2.1 变量赋值

语法

给变量赋值的语法请参见下图。

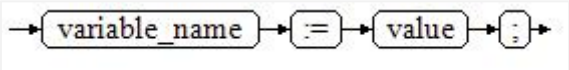

图 18-4 assignment_value::=

对以上语法格式的解释如下：

- variable_name：变量名。
- value：可以是值或表达式。值 value 的类型需要和变量 variable_name 的类型兼容才能正确赋值。

示例：

```
postgres=# DECLARE
    emp_id  INTEGER := 7788;--赋值
BEGIN
    emp_id := 5;--赋值
    emp_id := 5*7784;
END;
/
```

18.6.2.2 嵌套赋值

给变量嵌套赋值的语法请参见下图。

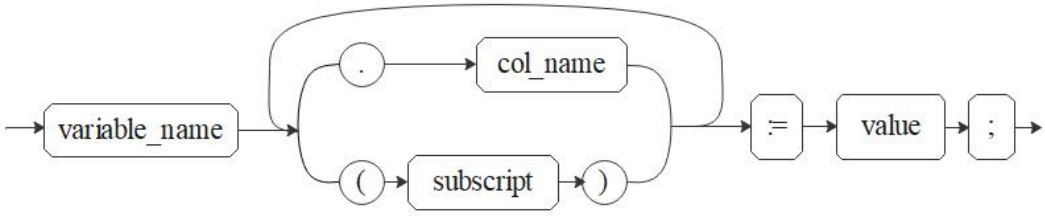

图 18-5 nested_assignment_value::=

对以上语法格式的解释如下：

- variable_name：变量名。
- col_name：列名。
- subscript：下标，针对数组变量使用，可以是值或表达式，类型必须为 int。
- value：可以是值或表达式。值 value 的类型需要和变量 variable_name 的类型兼容才能正确赋值。

示例

```
postgres=# CREATE TYPE o1 as (a int, b int);
postgres=# DECLARE
    TYPE r1 is VARRAY(10) of o1;
    emp_id  r1;
BEGIN
    emp_id(1).a := 5;--赋值
    emp_id(1).b := 5*7784;
END;
/
```

须知

- INTO 方式赋值仅支持对第一层列赋值，且不支持二维及以上数组；
- 引用嵌套的列值时，若存在数组下标，目前仅支持在前三层列中只存在一个小括号情况，建议使用方括号[ ]引用下标；

18.6.2.3 INTO/BULK COLLECT INTO

将存储过程内语句返回的值存储到变量内，BULK COLLECT INTO 允许将部分或全部返回值暂存到数组内部。

示例：

```
postgres=# DECLARE
    my_id integer;
BEGIN
    select id into my_id from customers limit 1; -- 赋值
END;
/
postgres=# DECLARE
    type id_list is varray(6) of customers.id%type;
    id_arr id_list;
BEGIN
    select id bulk collect into id_arr from customers order by id DESC limit 20;
-- 批量赋值
END;
/
```

须知

- BULK COLLECT INTO 只支持批量赋值给数组。合理使用 LIMIT 字段避免操作过量数据导致性能下降。

18.6.3 调用语句

语法

调用一个语句的语法请参见下图。

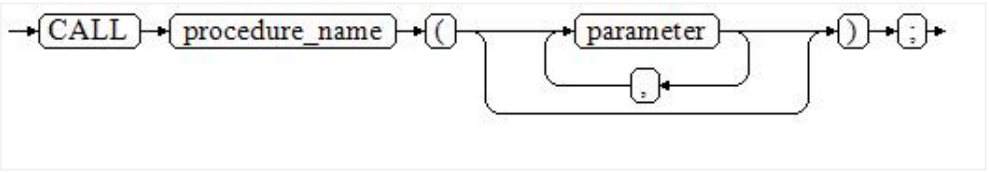

图 18-6 call_clause::=

对以上语法格式的解释如下：

- procedure_name：存储过程名。
- parameter：存储过程的参数，可以没有或者有多个参数。

示例

```
--创建存储过程 proc_staffs
postgres=# CREATE OR REPLACE PROCEDURE proc_staffs
(
section     NUMBER(6),
salary_sum out NUMBER(8,2),
staffs_count out INTEGER
)
IS
BEGIN
SELECT sum(salary), count(*) INTO salary_sum, staffs_count FROM hr.staffs where
section_id = section;
END;
/
--调用存储过程 proc_return.
postgres=# CALL proc_staffs(2,8,6);
--清除存储过程
postgres=# DROP PROCEDURE proc_staffs;
```

18.7 动态语句

18.7.1 执行动态查询语句

介绍执行动态查询语句。GBase 8s 提供两种方式：使用 EXECUTE IMMEDIATE、OPEN FOR 实现动态查询。前者通过动态执行 SELECT 语句，后者结合了游标的使用。当需要将查询的结果保存在一个数据集用于提取时，可使用 OPEN FOR 实现动态查询。

18.7.1.1 EXECUTE IMMEDIATE

语法图请参见下图。

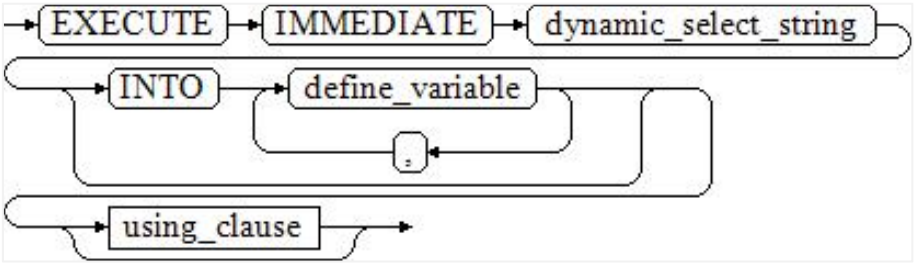

图 18-7 EXECUTE IMMEDIATE dynamic_select_clause::=

using_clause 子句的语法图参见下图。

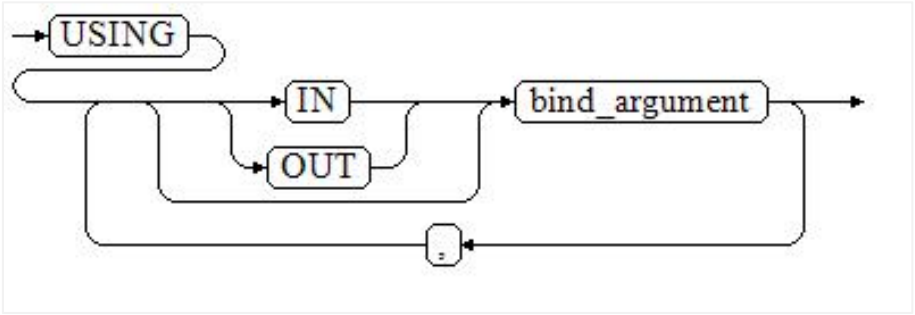

图 18-8 using_clause::=

对以上语法格式的解释如下：

- define_variable：用于指定存放单行查询结果的变量。
- USING IN bind_argument：用于指定存放传递给动态 SQL 值的变量，即在 dynamic_select_string 中存在占位符时使用。
- USING OUT bind_argument：用于指定存放动态 SQL 返回值的变量。

  须知

  - 查询语句中，into 和 out 不能同时存在；
  - 占位符命名以“:”开始，后面可跟数字、字符或字符串，与 USING 子句的 bind_argument 一一对应；
  - bind_argument 只能是值、变量或表达式，不能是表名、列名、数据类型等数 据库对象，即不支持使用 bind_argument 为动态 SQL 语句传递模式对象。如果存储过程需要通过声明参数传递数据库对象来构造动态 SQL 语句（常见于执行 DDL 语句时），建议采用连接运算符“||”拼接 dynamic_select_clause；
  - 动态 PL/SQL 块允许出现重复的占位符，即相同占位符只能与 USING 子句的一个

bind_argument 按位置对应。

18.7.1.2 OPEN FOR

动态查询语句还可以使用 OPEN FOR 打开动态游标来执行。语法参见下图。

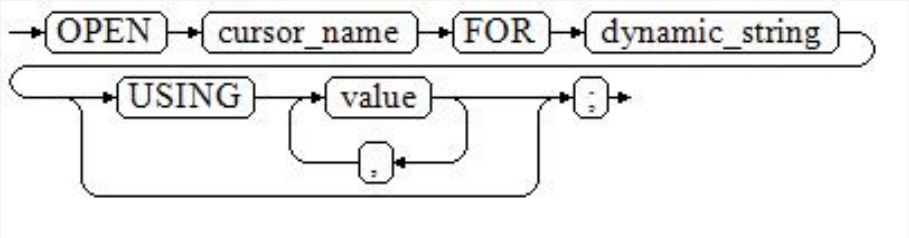

图 18-9 open_for::=

参数说明：

- cursor_name：要打开的游标名。
- dynamic_string：动态查询语句。
- USING value：在 dynamic_string 中存在占位符时使用。游标的使用请参考游标。

18.7.2 执行动态非查询语句

语法

语法请参见下图。

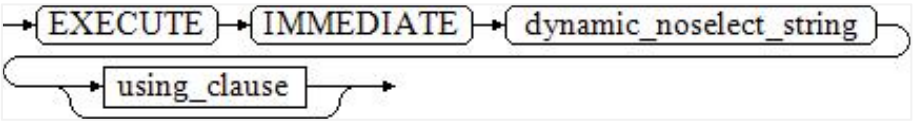

图 18-10 noselect::=

using_clause 子句的语法参见下图。

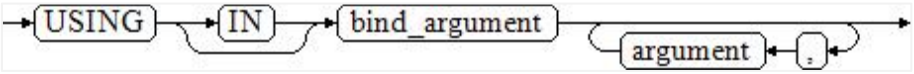

图 18-11 using_clause::=

对以上语法格式的解释如下：

USING IN bind_argument 用于指定存放传递给动态 SQL 值的变量，在 dynamic_noselect_string 中存在占位符时使用，即动态 SQL 语句执行时，bind_argument 将替

换相对应的占位符。要注意的是，bind_argument 只能是值、变 量或表达式，不能是表名、列名、数据类型等数据库对象。如果存储过程需要通过声明参数传递数据库对象来构造动态 SQL 语句（常见于执行 DDL 语句时），建议采用连接运算符“||”拼接 dynamic_select_clause。另外，动态语句允许出现重复的占位符，相同占位符只能与唯一一个 bind_argument 按位置一一对应。

示例

```
--创建表
postgres=# CREATE TABLE sections_t1 (
section    NUMBER(4), section_name VARCHAR2(30), manager_id NUMBER(6), place_id
   NUMBER(4)
);
--声明变量 postgres=# DECLARE
section    NUMBER(4) := 280;
section_name VARCHAR2(30) := 'Info support'; manager_id NUMBER(6) := 103;
place_id   NUMBER(4) := 1400; new_colname VARCHAR2(10) := 'sec_name';
BEGIN
--执行查询
EXECUTE IMMEDIATE 'insert into sections_t1 values(:1, :2, :3, :4)' USING section,
section_name, manager_id,place_id;
--执行查询（重复占位符）
EXECUTE IMMEDIATE 'insert into sections_t1 values(:1, :2, :3, :1)' USING section,
section_name, manager_id;
--执行 ALTER 语句（建议采用“||”拼接数据库对象构造 DDL 语句）
EXECUTE IMMEDIATE 'alter table sections_t1 rename section_name to ' || new_colname;
END;
/
--查询数据
postgres=# SELECT * FROM sections_t1;
--删除表
postgres=# DROP TABLE sections_t1;
```

18.7.3 动态调用存储过程

动态调用存储过程必须使用匿名的语句块将存储过程或语句块包在里面，使用 EXECUTE IMMEDIATE…USING 语句后面带 IN、OUT 来输入、输出参数。

18.7.3.1 语法

语法请参见下图。

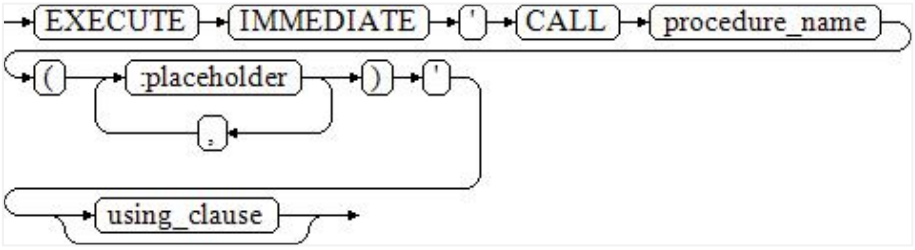

图 18-12 call_procedure::=

using_clause 子句的语法参见下图。

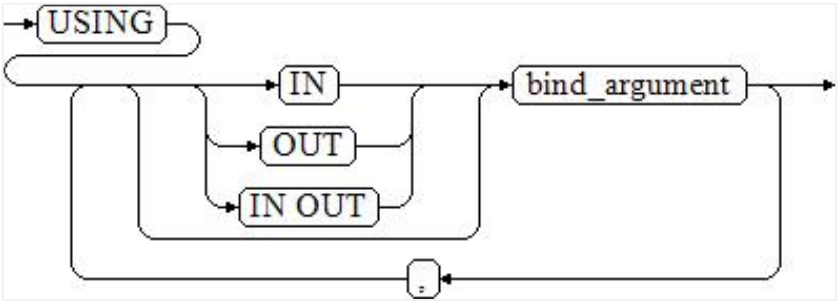

图 18-13 using_clause::=

对以上语法格式的解释如下：

- CALL procedure_name：调用存储过程。
- [:placeholder1，:placeholder2，…]：　存储过程参数占位符列表。占位符个数与参数个数相同。
- USING [IN|OUT|IN OUT] bind_argument：用于指定存放传递给存储过程参数值的变量。bind_argument 前的修饰符与对应参数的修饰符一致。

18.7.4 动态调用匿名块

动态调用匿名块是指在动态语句中执行匿名块，使用 EXECUTE IMMEDIATE…USING 语句后面带 IN、OUT 来输入、输出参数。

语法

语法请参见下图。

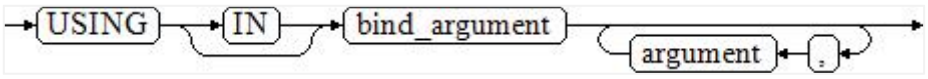

图 18-14 call_anonymous_block::=

using_clause 子句的语法参见下图。

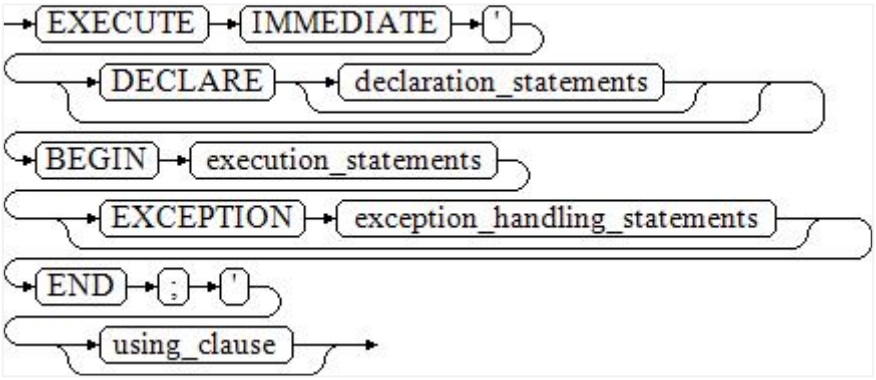

图 18-15 using_clause::=

对以上语法格式的解释如下：

匿名块程序实施部分，以 BEGIN 语句开始，以 END 语句停顿，以一个分号结束。

USING [IN|OUT|IN OUT] bind_argument，用于指定存放传递给存储过程参数值的变量。bind_argument 前的修饰符与对应参数的修饰符一致。

匿名块中间的输入输出参数使用占位符来指明，要求占位符个数与参数个数相同，并且占位符所对应参数的顺序和 USING 中参数的顺序一致。

目前 GBase 8s 在动态语句调用匿名块时，EXCEPTION 语句中暂不支持使用占位 符进行输入输出参数的传递。

18.8 控制语句

18.8.1 返回语句

GBase 8s 提供两种方式返回数据：RETURN 或 RETURN NEXT 及 RETURN QUERY。其中，RETURN NEXT 和 RETURN QUERY 只适用于函数，不适用存储过程。

18.8.1.1 RETURN

语法

返回语句的语法请参见下图。

图 18-16 return_clause::=

对以上语法的解释如下：

用于将控制从存储过程或函数返回给调用者。

示例

请参见调用语句的示例。

18.8.1.2 RETURN NEXT 及 RETURN QUERY

语法

创建函数时需要指定返回值 SETOF datatype。

```
return_next_clause::=
return_query_clause::=
```

对以上语法的解释如下：

当需要函数返回一个集合时，使用 RETURN NEXT 或者 RETURN QUERY 向结果集追加结果，然后继续执行函数的下一条语句。随着后续的RETURN NEXT或RETURN QUERY 命令的执行，结果集中会有多个结果。函数执行完成后会一起返回所有结果。

RETURN NEXT 可用于标量和复合数据类型。

RETURN QUERY 有一种变体 RETURN QUERY EXECUTE，后面还可以增加动态查询， 通过 USING 向查询插入参数。

示例

```
postgres=# CREATE TABLE t1(a int); postgres=# INSERT INTO t1 VALUES(1),(10);
--RETURN NEXT
postgres=# CREATE OR REPLACE FUNCTION fun_for_return_next() RETURNS SETOF t1 AS
$$ DECLARE
r t1%ROWTYPE;
BEGIN
FOR r IN select * from t1 LOOP
RETURN NEXT r; END LOOP; RETURN;
END;
$$ LANGUAGE PLPGSQL;
postgres=# call fun_for_return_next(); a
--- 1
10
(2 rows)
-- RETURN QUERY
```

```
postgres=# CREATE OR REPLACE FUNCTION fun_for_return_query() RETURNS SETOF t1 AS
$$ DECLARE
r t1%ROWTYPE;
BEGIN
RETURN QUERY select * from t1; END;
$$
language plpgsql;
postgres=# call fun_for_return_query(); a
---
1
10
(2 rows)
```

18.8.2 条件语句

条件语句的主要作用是判断参数或者语句是否满足已给定的条件，根据判定结果执行相应的操作。

GBase 8s 有五种形式的 IF：

- IF_THEN

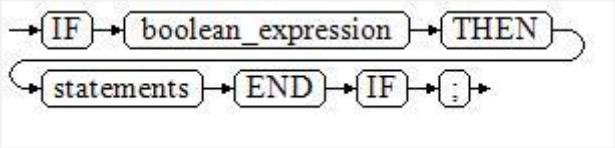

图 18-17 IF_THEN::=

IF_THEN 语句是 IF 的最简单形式。如果条件为真，statements 将被执行。否则，将忽略它们的结果使该 IF_THEN 语句执行结束。

示例

```
postgres=# IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

- IF_THEN_ELSE

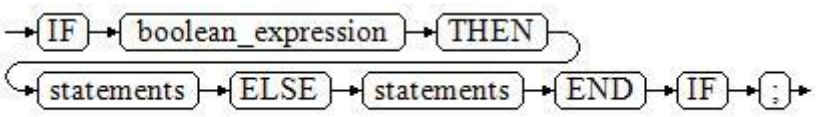

图 18-18 IF_THEN_ELSE::=

IF_THEN_ELSE 语句增加了 ELSE 的分支，可以声明在条件为假的时候执行的语句。

示例

```
postgres=# IF parentid IS NULL OR parentid = ''
THEN
    RETURN;
ELSE
    hp_true_filename(parentid);--表示调用存储过程
END IF;
IF_THEN_ELSE IF
```

IF 语句可以嵌套，嵌套方式如下：

```
postgres=# IF sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
```

这种形式实际上就是在一个 IF 语句的 ELSE 部分嵌套了另一个 IF 语句。因此需要一个 END IF 语句给每个嵌套的 IF，另外还需要一个 END IF 语句结束父 IF-ELSE。如果有多个选项，可使用下面的形式。

- IF_THEN_ELSIF_ELSE

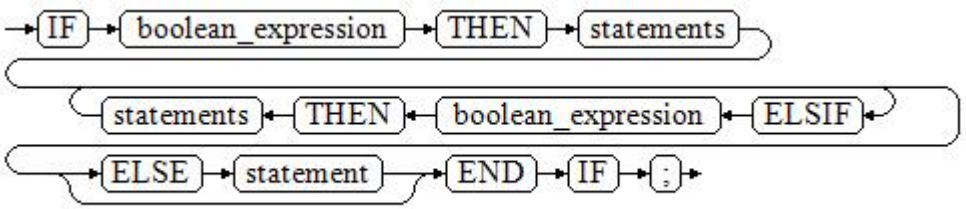

图 18-19 IF_THEN_ELSIF_ELSE::=

示例

```
IF number_tmp = 0 THEN
    result := 'zero';
ELSIF number_tmp > 0 THEN
    result := 'positive';
ELSIF number_tmp < 0 THEN
```

```
    result := 'negative';
ELSE
    result := 'NULL';
END IF;
IF_THEN_ELSEIF_ELSE
ELSEIF 是 ELSIF 的别名。
```

示例

```
CREATE OR REPLACE PROCEDURE proc_control_structure(i in integer)
AS
    BEGIN
        IF i > 0 THEN
            raise info 'i:% is greater than 0. ',i;
        ELSIF i < 0 THEN
            raise info 'i:% is smaller than 0. ',i;
        ELSE
            raise info 'i:% is equal to 0. ',i;
        END IF;
        RETURN;
    END;
/

CALL proc_control_structure(3);

--删除存储过程
DROP PROCEDURE proc_control_structure;
```

18.8.3 循环语句

18.8.3.1 简单 LOOP 语句

语法图

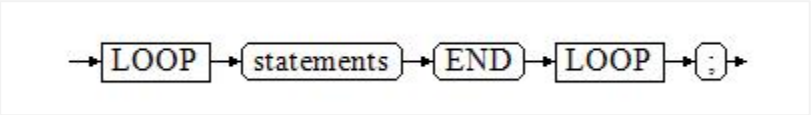

图 18-20 LOOP 语法图

示例

```
CREATE OR REPLACE PROCEDURE proc_loop(i in integer, count out integer)
AS
```

```
    BEGIN
        count:=0;
        LOOP
        IF count > i THEN
            raise info 'count is %. ', count;
            EXIT;
        ELSE
            count:=count+1;
        END IF;
        END LOOP;
    END;
/

CALL proc_loop(10,5);
```

须知

- 该循环必须要结合 EXIT 使用，否则将陷入死循环。

18.8.3.2 WHILE_LOOP 语句

语法图

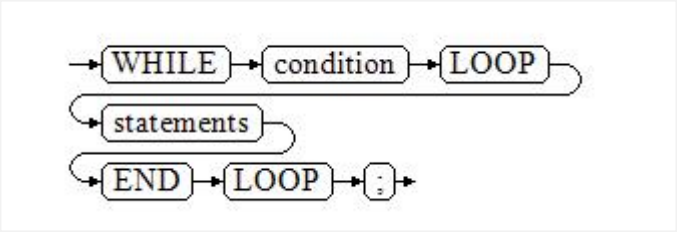

图 18-21 WHILE_LOOP 语法图

只要条件表达式为真，WHILE 语句就会不停的在一系列语句上进行循环，在每次进入循环体的时候进行条件判断。

示例

```
CREATE TABLE integertable(c1 integer) ;
CREATE OR REPLACE PROCEDURE proc_while_loop(maxval in integer)
AS
    DECLARE
    i int :=1;
    BEGIN
        WHILE i < maxval LOOP
```

```
            INSERT INTO integertable VALUES(i);
            i:=i+1;
        END LOOP;
    END;
/

--调用函数
CALL proc_while_loop(10);

--删除存储过程和表
DROP PROCEDURE proc_while_loop;
DROP TABLE integertable;
```

18.8.3.3 FOR_LOOP（integer 变量）语句

语法图

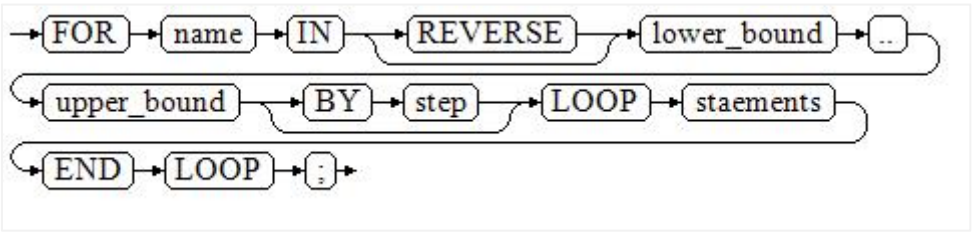

图 18-22 FOR_LOOP 语法图

说明

- 变量 name 会自动定义为 integer 类型并且只在此循环里存在。变量 name 介于 lower_bound 和 upper_bound 之间。
- 当使用 REVERSE 关键字时，lower_bound 必须大于等于 upper_bound，否则循环体不会被执行。

18.8.3.4 FOR_LOOP 查询语句

语法图

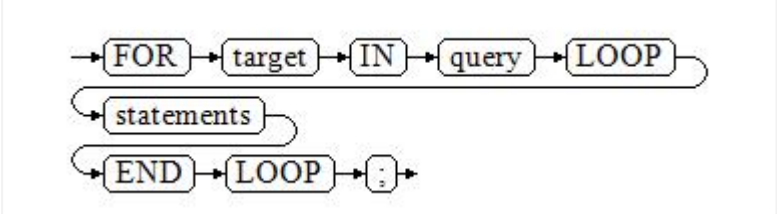

图 18-23 FOR_LOOP 查询语法图

说明：变量 target 会自动定义，类型和 query 的查询结果的类型一致，并且只在此循环中有效。target 的取值就是 query 的查询结果。

18.8.3.5 FORALL 批量查询语句

语法图

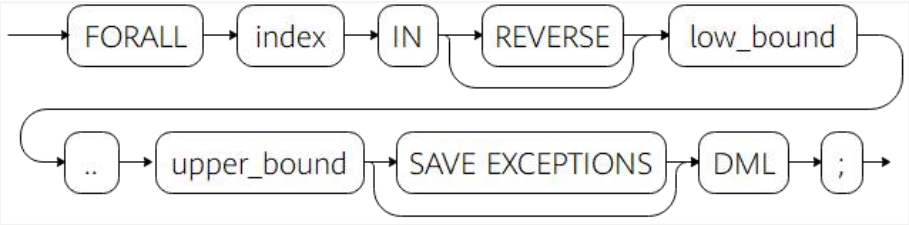

图 18-24 FORALL 批量查询语法图

说明

- 变量 index 会自动定义为 integer 类型并且只在此循环里存在。index 的取值介于 low_bound 和 upper_bound 之间。
- 如果声明了 SAVE EXCEPTIONS，则会将循环体 DML 执行过程中每次遇到的异常保存在 SQL&BULK_EXCEPTIONS 中，并在执行结束后统一抛出一个异常，循环过程中没有异常的执行的结果在当前子事务内不会回滚。

示例

```
CREATE TABLE hdfs_t1 (
  title NUMBER(6),
  did VARCHAR2(20),
  data_period VARCHAR2(25),
  kind VARCHAR2(25),
  interval VARCHAR2(20),
  time DATE,
  isModified VARCHAR2(10)
);

INSERT INTO hdfs_t1 VALUES( 8, 'Donald', 'OConnell', 'DOCONNEL', '650.507.9833',
to_date('21-06-1999', 'dd-mm-yyyy'), 'SH_CLERK' );

CREATE OR REPLACE PROCEDURE proc_forall()
```

```
AS
BEGIN
    FORALL i IN 100..120
        update hdfs_t1 set title = title + 100*i;
END;
/

--调用函数
CALL proc_forall();

--查询存储过程调用结果
SELECT * FROM hdfs_t1 WHERE title BETWEEN 100 AND 120;

--删除存储过程和表
DROP PROCEDURE proc_forall;
DROP TABLE hdfs_t1;
```

18.8.4 分支语句

语法图

分支语句的语法请参见下图。

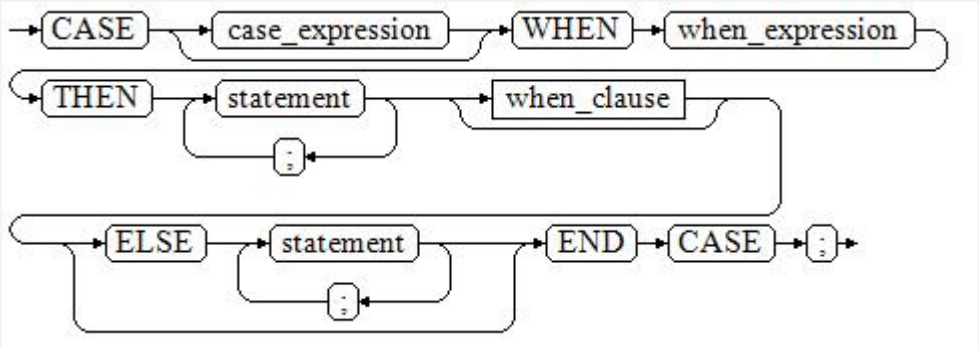

图 18-25 分支语句语法图

when_clause 子句的语法图参见下图。

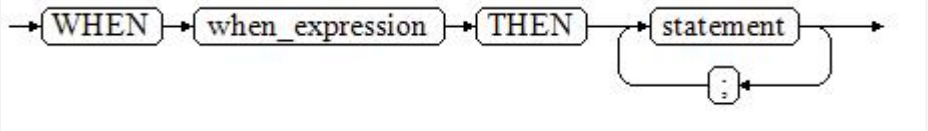

图 18-26 when_clause 子句语法图

参数说明：

- case_expression：变量或表达式。
- when_expression：常量或者条件表达式。
- statement：执行语句。

示例

```
CREATE OR REPLACE PROCEDURE proc_case_branch(pi_result in integer, pi_return out
integer)
AS
    BEGIN
        CASE pi_result
            WHEN 1 THEN
                pi_return := 111;
            WHEN 2 THEN
                pi_return := 222;
            WHEN 3 THEN
                pi_return := 333;
            WHEN 6 THEN
                pi_return := 444;
            WHEN 7 THEN
                pi_return := 555;
            WHEN 8 THEN
                pi_return := 666;
            WHEN 9 THEN
                pi_return := 777;
            WHEN 10 THEN
                pi_return := 888;
            ELSE
                pi_return := 999;
        END CASE;
        raise info 'pi_return : %',pi_return ;
END;
/

CALL proc_case_branch(3,0);

--删除存储过程
DROP PROCEDURE proc_case_branch;
```

18.8.5 空语句

在 PL/SQL 程序中，可以用 NULL 语句来说明“不用做任何事情”，相当于一个占位符，可以使某些语句变得有意义，提高程序的可读性。

语法

空语句的用法如下：

```
DECLARE
    ...
BEGIN
    ...
    IF v_num IS NULL THEN
        NULL; -- 不需要处理任何数据。
    END IF;
END;
/
```

18.8.6 错误捕获语句

缺省时，当 PL/SQL 函数执行过程中发生错误时退出函数执行，并且周围的事务也会回滚。可以用一个带有 EXCEPTION 子句的 BEGIN 块捕获错误并且从中恢复。其语法是正常的 BEGIN 块语法的一个扩展：

```
[<<label>>]
[DECLARE
    declarations]
BEGIN
    statements
EXCEPTION
    WHEN condition [OR condition ...] THEN
        handler_statements
    [WHEN condition [OR condition ...] THEN
        handler_statements
    ...]
END;
```

如果没有发生错误，这种形式的块儿只是简单地执行所有语句，然后转到 END 之后的下一个语句。但是如果在执行的语句内部发生了一个错误，则这个语句将会回滚，然后转到 EXCEPTION 列表。寻找匹配错误的第一个条件。若找到匹配，则执行对应的 handler_statements，然后转到 END 之后的下一个语句。如果没有找到匹配，则会向事务的

外层报告错误，和没有 EXCEPTION 子句一样。错误码可以捕获同一类的其他错误码。

也就是说该错误可以被一个包围块用 EXCEPTION 捕获，如果没有包围块，则进行退出函数处理。

condition 的名称可以是 SQL 标准错误码编号说明的任意值。特殊的条件名 OTHERS 匹配除了 QUERY_CANCELED 之外的所有错误类型。

如果在选中的 handler_statements 里发生了新错误，则不能被这个 EXCEPTION 子句捕获，而是向事务的外层报告错误。一个外层的 EXCEPTION 子句可以捕获它。

如果一个错误被 EXCEPTION 捕获，PL/SQL 函数的局部变量保持错误发生时的原值，但是所有该块中想写入数据库中的状态都回滚。

示例：

```
CREATE TABLE mytab(id INT,firstname VARCHAR(20),lastname VARCHAR(20)) ;

INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');

CREATE FUNCTION fun_exp() RETURNS INT
AS $$
DECLARE
    x INT :=0;
    y INT;
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;$$
LANGUAGE plpgsql;

call fun_exp();
NOTICE:  caught division_by_zero
 fun_exp
---------
       1
(1 row)
```

```
select * from mytab;
 id | firstname | lastname
----+-----------+----------
    | Tom       | Jones
(1 row)

DROP FUNCTION fun_exp();
DROP TABLE mytab;
```

当控制到达给 y 赋值的地方时，会有一个 division_by_zero 错误失败。这个错误将被 EXCEPTION 子句捕获。而在 RETURN 语句里返回的数值将是 x 的增量值。

说明

- 进入和退出一个包含 EXCEPTION 子句的块要比不包含的块开销大的多。因此，不必要的时候不要使用 EXCEPTION。

在下列场景中，无法捕获处理异常，整个存储过程回滚：节点故障、网络故障引起的存储过程参与节点线程退出以及 COPY FROM 操作中源数据与目标表的表结构不一致造成的异常。

示例：UPDATE/INSERT 异常

这个例子根据使用异常处理器执行恰当的 UPDATE 或 INSERT 。

```
CREATE TABLE db (a INT, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP

--第一次尝试更新 key
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
--不存在，所以尝试插入 key，如果其他人同时插入相同的 key，我们可能得到唯一 key 失败。
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
        --什么也不做，并且循环尝试再次更新。
```

```
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');

--删除 FUNCTION 和 TABLE
DROP FUNCTION merge_db;
DROP TABLE db;
```

18.8.7 GOTO 语句

GOTO 语句可以实现从 GOTO 位置到目标语句的无条件跳转。GOTO 语句会改变原本的执行逻辑，因此应该慎重使用，或者也可以使用 EXCEPTION 处理特殊场景。当执行 GOTO 语句时，目标 Label 必须是唯一的。

语法

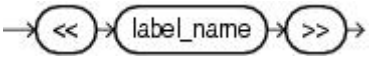

图 18-27 label declaration ::=

图 18-28 goto statement ::=

示例

```
postgres=# CREATE OR REPLACE PROCEDURE GOTO_test()
AS
DECLARE
    v1  int;
BEGIN
    v1  := 0;
        LOOP
        EXIT WHEN v1 > 100;
                v1 := v1 + 2;
                if v1 > 25 THEN
                        GOTO pos1;
```

```
                END IF;
        END LOOP;
<<pos1>>
v1 := v1 + 10;
raise info 'v1 is %. ', v1;
END;
/

call GOTO_test();
```

限制场景

GOTO 使用有以下限制场景：

不支持有多个相同的 GOTO labels 目标场景，无论是否在同一个 block 中。

```
BEGIN
  GOTO pos1;
  <<pos1>>
  SELECT * FROM ...
  <<pos1>>
  UPDATE t1 SET ...
END;
```

不支持 GOTO 跳转到 IF 语句、CASE 语句、LOOP 语句中。

```
BEGIN
   GOTO pos1;
   IF valid THEN
     <<pos1>>
     SELECT * FROM ...
   END IF;
 END;
```

不支持 GOTO 语句从一个 IF 子句跳转到另一个 IF 子句，或从一个 CASE 语句的 WHEN 子句跳转到另一个 WHEN 子句。

```
BEGIN
   IF valid THEN
     GOTO pos1;
     SELECT * FROM ...
   ELSE
     <<pos1>>
     UPDATE t1 SET ...
```

```
  END IF;
END;
```

不支持从外部块跳转到内部的 BEGIN-END 块。

```
BEGIN
  GOTO pos1;
  BEGIN
    <<pos1>>
    UPDATE t1 SET ...
  END;
END;
```

不支持从异常处理部分跳转到当前的 BEGIN-END 块。但可以跳转到上层 BEGIN-END 块。

```
BEGIN
  <<pos1>>
  UPDATE t1 SET ...
  EXCEPTION
    WHEN condition THEN
       GOTO pos1;
END;
```

如果从 GOTO 到一个不包含执行语句的位置，需要添加 NULL 语句。

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
     IF done THEN
        GOTO end_loop;
     END IF;
     <<end_loop>>  -- not allowed unless an executable statement follows
     NULL; -- add NULL statement to avoid error
  END LOOP;  -- raises an error without the previous NULL
END;
/
```

18.9 事务管理

存储过程本身就处于一个事务中，开始调用最外围存储过程时会自动开启一个事务，在调用结束时自动提交或者发生异常时回滚。除了系统自动的事务控制外，也可以使用 COMMIT/ROLLBACK 来控制存储过程中的事务。在存储过程中调用 COMMIT/ROLLBACK

命令，将提交/回滚当前事务并自动开启一个新的事务，后续的所有操作都会在此新事务中运行。

保存点 SAVEPOINT 是事务中的一个特殊记号，它允许将那些在它建立后执行的命令全部回滚，把事务的状态恢复到保存点所在的时刻。存储过程中允许使用保存点来进行事务管理，当前支持保存点的创建、回滚和释放操作。存储过程中使用回滚保存点只是回退当前事务的修改，而不会改变存储过程的执行流程，也不会回退存储过程中的局部变量值等。

18.9.1 语法格式

定义保存点

```
SAVEPOINT savepoint_name;
```

回滚保存点

```
ROLLBACK TO [SAVEPOINT] savepoint_name;
```

释放保存点

```
RELEASE [SAVEPOINT] savepoint_name;
```

18.9.2 使用场景

支持调用的上下文环境：

支持在 PLSQL 的存储过程内使用 COMMIT/ROLLBACK/SAVEPOINT。

支持含有 EXCEPTION 的存储过程使用 COMMIT/ROLLBACK/SAVEPOINT。

支持在存储过程的 EXCEPTION 语句内使用 COMMIT/ROLLBACK/SAVEPOINT。

支持在事务块里调用含有 COMMIT/ROLLBACK/SAVEPOINT 的存储过程，即通过/BEGIN/START/END 等开启控制的外部事务。

支持在子事务中调用含有 SAVEPOINT 的存储过程，即存储过程中使用外部定义的 SAVEPOINT，回退事务状态到存储过程外定义的 SAVEPOINT 位置。

支持存储过程外部对存储过程内定义的 SAVEPOINT 可见，即存储过程外可以将事务修改回滚到存储过程中定义 SAVEPOINT 的位置。

支持多数 PLSQL 的上下文和语句内调用 COMMIT/ROLLBACK/SAVEPOINT，包括常用的 IF/FOR/CURSOR LOOP/WHILE。

支持存储过程返回值与简单表达式计算中调用含有

COMMIT/ROLLBACK/SAVEPOINT 的存储过程或者函数。

支持提交/回滚的内容：

支持 DDL 在 COMMIT/ROLLBACK 后的提交/回滚。

支持 DML 的 COMMIT/ROLLBACK 后的提交。

支持存储过程内 GUC 参数的回滚提交。

18.9.3 使用限制

不支持调用的上下文环境：

不支持除 PLSQL 的其他存储过程中调用 COMMIT/ROLLBACK/SAVEPOINT，例如 PLJAVA、PLPYTHON 等。

不支持函数中调用 COMMIT/ROLLBACK/SAVEPOINT，包括函数调用含有 COMMIT/ROLLBACK/SAVEPOINT 的存储过程。

不支持事务块中调用了 SAVEPOINT 后，调用含有 COMMIT/ROLLBACK 的存储过程。

不支持 TRIGGER 中调用含有 COMMIT/ROLLBACK/SAVEPOINT 语句的存储过程。

不支持 EXECUTE 语句中调用 COMMIT/ROLLBACK/SAVEPOINT 语句。

不支持在 CURSOR 语句中打开一个含有 COMMIT/ROLLBACK/SAVEPOINT 的存储过程。

不支持带有 IMMUTABLE 以及 SHIPPABLE 的存储过程调用 COMMIT/ROLLBACK/SAVEPOINT，或调用带有 COMMIT/ROLLBACK/SAVEPOINT 语句的存储过程。

不支持 SQL 中调用含有 COMMIT/ROLLBACK/SAVEPOINT 语句的存储过程，除了 SELECT PROC 以及 CALL PROC。

存储过程头带有 GUC 参数设置的不允许调用 COMMIT/ROLLBACK/SAVEPOINT 语句。

不支持 CURSOR/EXECUTE 语句，以及各类表达式内调用 COMMIT/ROLLBACK/SAVEPOINT。

自治事务和存储过程事务是两个独立的事务，不能互相使用对方事务中定义的保存点。

不支持提交回滚的内容：

不支持存储过程内声明变量以及传入变量的提交/回滚。

不支持存储过程内必须重启生效的 GUC 参数的提交/回滚。

18.9.4 示例

示例 1：支持在 PLSQL 的存储过程内使用 COMMIT/ROLLBACK。

```
CREATE TABLE EXAMPLE1(COL1 INT);

CREATE OR REPLACE PROCEDURE TRANSACTION_EXAMPLE()
AS
BEGIN
    FOR i IN 0..20 LOOP
        INSERT INTO EXAMPLE1(COL1) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END;
/
```

示例 2：

支持含有 EXCEPTION 的存储过程使用 COMMIT/ROLLBACK。

支持在存储过程的 EXCEPTION 语句内使用 COMMIT/ROLLBACK。

支持 DDL 在 COMMIT/ROLLBACK 后的提交/回滚。

```
CREATE OR REPLACE PROCEDURE TEST_COMMIT_INSERT_EXCEPTION_ROLLBACK()
AS
BEGIN
    DROP TABLE IF EXISTS TEST_COMMIT;
    CREATE TABLE TEST_COMMIT(A INT, B INT);
    INSERT INTO TEST_COMMIT SELECT 1, 1;
    COMMIT;
        CREATE TABLE TEST_ROLLBACK(A INT, B INT);
    RAISE EXCEPTION 'RAISE EXCEPTION AFTER COMMIT';
EXCEPTION
    WHEN OTHERS THEN
    INSERT INTO TEST_COMMIT SELECT 2, 2;
    ROLLBACK;
END;
```

```
/
```

示例 3：支持在事务块里调用含有 COMMIT/ROLLBACK 的存储过程，即通过/BEGIN/START/END 等开启控制的外部事务。

```
BEGIN;
    CALL TEST_COMMIT_INSERT_EXCEPTION_ROLLBACK();
END;
```

示例 4：支持多数 PLSQL 的上下文和语句内调用 COMMIT/ROLLBACK，包括常用的 IF/FOR/CURSOR LOOP/WHILE。

```
CREATE OR REPLACE PROCEDURE TEST_COMMIT2()
IS
BEGIN
    DROP TABLE IF EXISTS TEST_COMMIT;
    CREATE TABLE TEST_COMMIT(A INT);
    FOR I IN REVERSE 3..0 LOOP
    INSERT INTO TEST_COMMIT SELECT I;
    COMMIT;
    END LOOP;
    FOR I IN REVERSE 2..4 LOOP
    UPDATE TEST_COMMIT SET A=I;
    COMMIT;
    END LOOP;
EXCEPTION
WHEN OTHERS THEN
    INSERT INTO TEST_COMMIT SELECT 4;
    COMMIT;
END;
/
```

示例 5：支持存储过程返回值与简单表达式计算。

```
CREATE OR REPLACE PROCEDURE exec_func3(RET_NUM OUT INT)
AS
BEGIN
  RET_NUM := 1+1;
COMMIT;
END;
/
CREATE OR REPLACE PROCEDURE exec_func4(ADD_NUM IN INT)
AS
SUM_NUM INT;
```

```
BEGIN
SUM_NUM := ADD_NUM + exec_func3();
COMMIT;
END;
/
```

示例 6：支持存储过程内 GUC 参数的回滚提交。

```
SHOW explain_perf_mode;
SHOW enable_force_vector_engine;

CREATE OR REPLACE PROCEDURE GUC_ROLLBACK()
AS
BEGIN
    SET enable_force_vector_engine = on;
    COMMIT;
    SET explain_perf_mode TO pretty;
    ROLLBACK;
END;
/

call GUC_ROLLBACK();
SHOW explain_perf_mode;
SHOW enable_force_vector_engine;
SET enable_force_vector_engine = off;
```

示例 7：函数（Function）中不允许调用 commit/rollback 语句，同时不允许函数调用含有 commit/rollback 的存储过程。

```
CREATE OR REPLACE FUNCTION FUNCTION_EXAMPLE1() RETURN INT
AS
EXP INT;
BEGIN
    FOR i IN 0..20 LOOP
        INSERT INTO EXAMPLE1(col1) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
    SELECT COUNT(*) FROM EXAMPLE1 INTO EXP;
    RETURN EXP;
```

```
END;
/
```

示例 8：函数（Fucntion）中不允许调用带有 commit/rollback 语句的存储过程。

```
CREATE OR REPLACE FUNCTION FUNCTION_EXAMPLE2() RETURN INT
AS
EXP INT;
BEGIN
    --transaction_example 为存储过程，带有 commit/rollback 语句
    CALL transaction_example();
    SELECT COUNT(*) FROM EXAMPLE1 INTO EXP;
    RETURN EXP;
END;
/
```

示例 9：不允许 Trigger 的存储过程包含 commit/rollback 语句，或调用带有 commit/rollback 语句的存储过程。

```
CREATE OR REPLACE FUNCTION FUNCTION_TRI_EXAMPLE2() RETURN TRIGGER
AS
EXP INT;
BEGIN
    FOR i IN 0..20 LOOP
        INSERT INTO EXAMPLE1(col1) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
    SELECT COUNT(*) FROM EXAMPLE1 INTO EXP;
END;
/

CREATE TRIGGER TRIGGER_EXAMPLE AFTER DELETE ON EXAMPLE1
FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRI_EXAMPLE2();

DELETE FROM EXAMPLE1;
```

示例 10：不支持带有 IMMUABLE 以及 SHIPPABLE 的存储过程调用 commit/rollback，或调用带有 commit/rollback 语句的存储过程。

```
CREATE OR REPLACE PROCEDURE TRANSACTION_EXAMPLE1()
```

```
IMMUTABLE
AS
BEGIN
    FOR i IN 0..20 LOOP
        INSERT INTO EXAMPLE1 (col1) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END;
/
```

示例 11：不支持出现在 SQL 中的调用（除了 Select Procedure）。

```
CREATE OR REPLACE PROCEDURE TRANSACTION_EXAMPLE3()
AS
BEGIN
    FOR i IN 0..20 LOOP
        INSERT INTO EXAMPLE1 (col1) VALUES (i);
        IF i % 2 = 0 THEN
            EXECUTE IMMEDIATE 'COMMIT';
        ELSE
            EXECUTE IMMEDIATE 'ROLLBACK';
        END IF;
    END LOOP;
END;
/
```

示例 12：存储过程头带有 GUC 参数设置的不允许调用 commit/rollback 语句。

```
CREATE OR REPLACE PROCEDURE TRANSACTION_EXAMPLE4()
SET ARRAY_NULLS TO "ON"
AS
BEGIN
    FOR i IN 0..20 LOOP
        INSERT INTO EXAMPLE1 (col1) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
```

```
END;
/
```

示例 13：游标 open 的对象不允许为带有 commit/rollback 语句的存储过程。

```
CREATE OR REPLACE PROCEDURE TRANSACTION_EXAMPLE5(INTIN IN INT, INTOUT OUT INT)
AS
BEGIN
INTOUT := INTIN + 1;
COMMIT;
END;
/

CREATE OR REPLACE PROCEDURE TRANSACTION_EXAMPLE6()
AS
CURSOR CURSOR1(EXPIN INT)
IS SELECT TRANSACTION_EXAMPLE5(EXPIN);
INTEXP INT;
BEGIN
    FOR i IN 0..20 LOOP
        OPEN CURSOR1(i);
        FETCH CURSOR1 INTO INTEXP;
        INSERT INTO EXAMPLE1(COL1) VALUES (INTEXP);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
        CLOSE CURSOR1;
    END LOOP;
END;
/
```

示例 14：不支持 CURSOR/EXECUTE 语句，以及各类表达式内调用 COMMIT/ROLLBACK。

```
CREATE OR REPLACE PROCEDURE exec_func1()
AS
BEGIN
    CREATE TABLE TEST_exec(A INT);
COMMIT;
END;
/
```

```
CREATE OR REPLACE PROCEDURE exec_func2()
AS
BEGIN
EXECUTE exec_func1();
COMMIT;
END;
/
```

示例 15：存储过程使用保存点回退事务部分修改。

```
CREATE OR REPLACE PROCEDURE STP_SAVEPOINT_EXAMPLE1()
AS
BEGIN
    INSERT INTO EXAMPLE1 VALUES(1);
    SAVEPOINT s1;
    INSERT INTO EXAMPLE1 VALUES(2);
    ROLLBACK TO s1;  -- 回退插入记录 2
    INSERT INTO EXAMPLE1 VALUES(3);
END;
/
```

示例 16：存储过程中使用保存点回退到存储过程外部定义的保存点。

```
CREATE OR REPLACE PROCEDURE STP_SAVEPOINT_EXAMPLE2()
AS
BEGIN
    INSERT INTO EXAMPLE1 VALUES(2);
    ROLLBACK TO s1;  -- 回退插入记录 2
    INSERT INTO EXAMPLE1 VALUES(3);
END;
/

BEGIN;
INSERT INTO EXAMPLE1 VALUES(1);
SAVEPOINT s1;
CALL STP_SAVEPOINT_EXAMPLE2();
SELECT * FROM EXAMPLE1;
COMMIT;
```

示例 17：存储过程外部回退到存储过程中定义的保存点。

```
CREATE OR REPLACE PROCEDURE STP_SAVEPOINT_EXAMPLE3()
AS
BEGIN
    INSERT INTO EXAMPLE1 VALUES(1);
```

```
    SAVEPOINT s1;
    INSERT INTO EXAMPLE1 VALUES(2);
END;
/

BEGIN;
INSERT INTO EXAMPLE1 VALUES(3);
CALL STP_SAVEPOINT_EXAMPLE3();
ROLLBACK TO SAVEPOINT s1; --回退存储过程中插入记录 2
SELECT * FROM EXAMPLE1;
COMMIT;
```

18.10 其他语句

18.10.1锁操作

GBase 8s 提供了多种锁模式用于控制对表中数据的并发访问。这些模式可以用在 MVCC（多版本并发控制）无法给出期望行为的场合。同样，大多数 GBase 8s 命令自动施加恰当的锁，以保证被引用的表在命令的执行过程中不会以一种不兼容的方式被删除或者修改。比如，在存在其他并发操作的时候，ALTER TABLE 是不能在同一个表上执行的。

18.10.2游标操作

GBase 8s 中游标（cursor）是系统为用户开设的一个数据缓冲区，存放着 SQL 语句的执行结果。每个游标区都有一个名称。用户可以用 SQL 语句逐一从游标中获取记录，并赋给主变量，交由主语言进一步处理。

游标的操作主要有游标的定义、打开、获取和关闭。

完整的游标操作示例可参考显式游标。

18.11 游标

18.11.1游标概述

为了处理 SQL 语句，存储过程进程分配一段内存区域来保存上下文联系。游标是指向上下文区域的句柄或指针。借助游标，存储过程可以控制上下文区域的变化。

须知

- 当游标作为存储过程的返回值时，如果使用 JDBC 调用该存储过程，返回的游标将不可用。

游标的使用分为显式游标和隐式游标。对于不同的 SQL 语句，游标的使用情况不同，详细信息请参见下表。

表 18-2 游标使用情况

SQL 语句	游标
非查询语句	隐式的
结果是单行的查询语句	隐式的或显式的
结果是多行的查询语句	显式的

18.11.2 显式游标

显式游标主要用于对查询语句的处理，尤其是在查询结果为多条记录的情况下。

18.11.2.1 处理步骤

显式游标处理需六个 PL/SQL 步骤：

(1) 定义静态游标：就是定义一个游标名，以及与其相对应的 SELECT 语句。

定义静态游标的语法图，请参见下图。

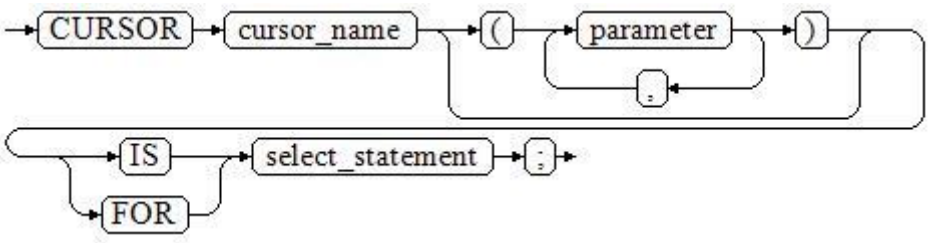

图 18-29 static_cursor_define::=

参数说明：

- cursor_name：定义的游标名。

- parameter：游标参数，只能为输入参数，其格式为：
- parameter_name datatype
- select_statement：查询语句。

  说明：

  ➢ 根据执行计划的不同，系统会自动判断该游标是否可以用于以倒序的方式检索数据行。

(2) 定义动态游标：指 ref 游标，可以通过一组静态的 SQL 语句动态的打开游标。首先定义 ref 游标类型，然后定义该游标类型的游标变量，在打开游标时通过 OPEN FOR 动态绑定 SELECT 语句。

定义动态游标的语法图，请参见图 18-30 和图 18-31。

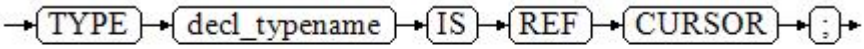

图 18-30 cursor_typename::=

图 18-31 dynamic_cursor_define::=

(3) 打开静态游标：就是执行游标所对应的 SELECT 语句，将其查询结果放入工作区，并且指针指向工作区的首部，标识游标结果集合。如果游标查询语句中带有 FOR UPDATE 选项，OPEN 语句还将锁定数据库表中游标结果集合对应的数据行。

打开静态游标的语法图，请参见下图。

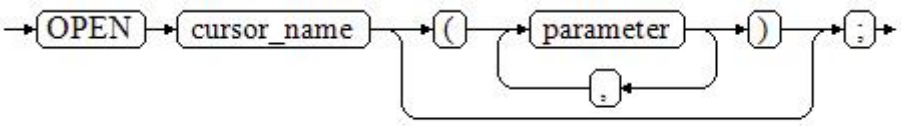

图 18-32 open_static_cursor::=

(4) 打开动态游标：可以通过 OPEN FOR 语句打开动态游标，动态绑定 SQL 语句。

打开动态游标的语法图，请参见下图。

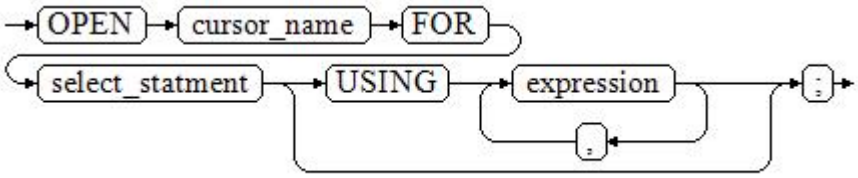

图 18-33 open_dynamic_cursor::=

PL/SQL 程序不能用 OPEN 语句重复打开一个游标。

(5) 提取游标数据：检索结果集合中的数据行，放入指定的输出变量中。

提取游标数据的语法图，请参见下图。

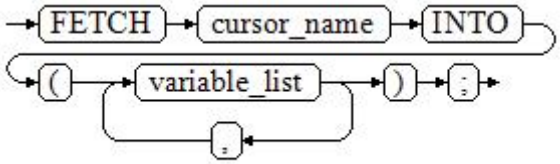

图 18-34 fetch_cursor::=

(6) 对该记录进行处理。

(7) 继续处理，直到活动集合中没有记录。

(8) 关闭游标：当提取和处理完游标结果集合数据后，应及时关闭游标，以释放该游标所占用的系统资源，并使该游标的工作区变成无效，不能再使用 FETCH 语句获取其中数据。关闭后的游标可以使用 OPEN 语句重新打开。

关闭游标的语法图，请参见下图。

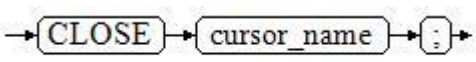

图 18-35 close_cursor::=

18.11.2.2 属性

游标的属性用于控制程序流程或者了解程序的状态。当运行 DML 语句时，PL/SQL 打开一个内建游标并处理结果，游标是维护查询结果的内存中的一个区域，游标在运行 DML 语句时打开，完成后关闭。显式游标的属性为：

- %FOUND 布尔型属性：当最近一次读记录时成功返回，则值为 TRUE。

- %NOTFOUND 布尔型属性：与%FOUND 相反。
- %ISOPEN 布尔型属性：当游标已打开时返回 TRUE。
- %ROWCOUNT 数值型属性：返回已从游标中读取的记录数。

18.11.3 隐式游标

对于非查询语句，如修改、删除操作，则由系统自动地为这些操作设置游标并创建其工作区，这些由系统隐含创建的游标称为隐式游标，隐式游标的名称为 SQL，这是由系统定义的。

18.11.3.1 简介

对于隐式游标的操作，如定义、打开、取值及关闭操作，都由系统自动地完成，无需用户进行处理。用户只能通过隐式游标的相关属性，来完成相应的操作。在隐式游标的工作区中，所存放的数据是最新处理的一条 SQL 语句所包含的数据，与用户自定义的显式游标无关。

格式调用为：SQL%

说明

- INSERT、UPDATE、DELETE、SELECT 语句中不必明确定义游标。
- 兼容 O 模式下，GUC 参数 behavior_compat_options 为 compat_cursor 时，隐式游标跨存储过程有效。

18.11.3.2 属性

隐式游标属性为：

- SQL%FOUND 布尔型属性：当最近一次读记录时成功返回，则值为 TRUE。
- SQL%NOTFOUND 布尔型属性：与%FOUND 相反。
- SQL%ROWCOUNT 数值型属性：返回已从游标中读取得记录数。
- SQL%ISOPEN 布尔型属性：取值总是 FALSE。SQL 语句执行完毕立即关闭隐式游标。

18.11.3.3 示例

```
--删除员工表 hr.staffs 表中某部门的所有员工，如果该部门中已没有员工，则在部门表 hr.sections 中删除该部门。
```

```
CREATE OR REPLACE PROCEDURE proc_cursor3()
AS
    DECLARE
    V_DEPTNO NUMBER(4) := 100;
    BEGIN
        DELETE FROM hr.staffs WHERE section_ID = V_DEPTNO;
        --根据游标状态做进一步处理
        IF SQL%NOTFOUND THEN
        DELETE FROM hr.sections WHERE section_ID = V_DEPTNO;
        END IF;
    END;
/

CALL proc_cursor3();

--删除存储过程和临时表
DROP PROCEDURE proc_cursor3;
```

18.11.4 游标循环

游标在 WHILE 语句、LOOP 语句中的使用称为游标循环，一般这种循环都需要使用 OPEN、FETCH 和 CLOSE 语句。下面要介绍的一种循环不需要这些操作，可以简化游标循环的操作，这种循环方式适用于静态游标的循环，不用执行静态游标的四个步骤。

18.11.4.1 语法

FOR AS 循环的语法请参见下图。

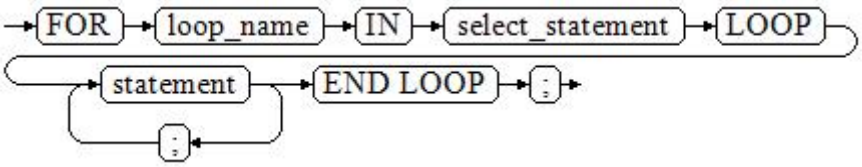

图 18-36 FOR_AS_loop::=

18.11.4.2 注意事项

不能在该循环语句中对查询的表进行更新操作。

变量 loop_name 会自动定义且只在此循环中有效，类型和 select_statement 的查询结果类型一致。loop_name 的取值就是 select_statement 的查询结果。

游标的属性中%FOUND、%NOTFOUND、%ROWCOUNT 在 GBase 8s 数据库中都是访问同一个内部变量，事务和匿名块不支持多个游标同时访问。

18.12 高级包

本章介绍高级包的基础接口。

18.12.1基础接口

18.12.1.1 PKG_SERVICE

PKG_SERVICE 支持的所有接口请参见下表。

表 18-3 PKG_SERVICE

接口名称	描述
PKG_SERVICE.SQL_IS_CONTEXT_ACTIVE	确认该 CONTEXT 是否已注册。
PKG_SERVICE.SQL_CLEAN_ALL_CONTEXTS	取消所有注册的 CONTEXT。
PKG_SERVICE.SQL_REGISTER_CONTEXT	注册一个 CONTEXT。
PKG_SERVICE.SQL_UNREGISTER_CONTEXT	取消注册该 CONTEXT。
PKG_SERVICE.SQL_SET_SQL	向 CONTEXT 设置一条 SQL 语句，目前只支持 SELECT。
PKG_SERVICE.SQL_RUN	在一个 CONTEXT 上执行设置的 SQL 语句。
PKG_SERVICE.SQL_NEXT_ROW	读取该 CONTEXT 中的下一行数据。

接口名称	描述
PKG_SERVICE.SQL_GET_VALUE	读取该 CONTEXT 中动态定义的列值
PKG_SERVICE.SQL_SET_RESULT_TYPE	根据类型 OID 动态定义该 CONTEXT 的一个列。
PKG_SERVICE.JOB_CANCEL	通过任务 ID 来删除定时任务。
PKG_SERVICE.JOB_FINISH	禁用或者启用定时任务。
PKG_SERVICE.JOB_SUBMIT	提交一个定时任务。作业号由系统自动生成或由用户指定。
PKG_SERVICE.JOB_UPDATE	修改定时任务的属性，包括任务内容、下次执行时间、执行间隔。
PKG_SERVICE.SUBMIT_ON_NODES	提交一个任务到所有节点，作业号由系统自动生成。
PKG_SERVICE.ISUBMIT_ON_NODES	提交一个任务到所有节点，作业号由用户指定。
PKG_SERVICE.SQL_GET_ARRAY_RESULT	获取该 CONTEXT 中返回的数组值。
PKG_SERVICE.SQL_GET_VARIABLE_RESULT	获取该 CONTEXT 中返回的列值。

- PKG_SERVICE.SQL_IS_CONTEXT_ACTIVE

该函数用来确认一个 CONTEXT 是否已注册。该函数传入想查找的 CONTEXT ID，如果该 CONTEXT 存在返回 TRUE，反之返回 FALSE。

PKG_SERVICE.SQL_IS_CONTEXT_ACTIVE 函数原型为：

```
PKG_SERVICE.SQL_IS_CONTEXT_ACTIVE(
 context_id     IN INTEGER
)
RETURN BOOLEAN;
```

表 18-4 PKG_SERVICE.SQL_IS_CONTEXT_ACTIVE 接口说明

参数名称	描述
context_id	想查找的 CONTEXT ID 号

- PKG_SERVICE.SQL_CLEAN_ALL_CONTEXTS

  该函数用来取消所有 CONTEXT

  PKG_SERVICE.SQL_CLEAN_ALL_CONTEXTS 函数原型为：

```
PKG_SERVICE.SQL_CLEAN_ALL_CONTEXTS(
)
RETURN VOID;
```

- PKG_SERVICE.SQL_REGISTER_CONTEXT

  该函数用来打开一个 CONTEXT，是后续对该 CONTEXT 进行各项操作的前提。该函数不传入任何参数，内部自动递增生成 CONTEXT ID，并作为返回值返回给 integer 定义的变量。

  PKG_SERVICE.SQL_REGISTER_CONTEXT 函数原型为：

```
DBE_SQL.REGISTER_CONTEXT(
)
RETURN INTEGER;
```

- PKG_SERVICE.SQL_UNREGISTER_CONTEXT

  该函数用来关闭一个 CONTEXT，是该 CONTEXT 中各项操作的结束。如果在存储过程结束时没有调用该函数，则该 CONTEXT 占用的内存仍然会保存，因此关闭 CONTEXT 非常重要。由于异常情况的发生会中途退出存储过程，导致 CONTEXT 未能关闭，因此建议存储过程中有异常处理，将该接口包含在内。

  PKG_SERVICE.SQL_UNREGISTER_CONTEXT 函数原型为：

```
PKG_SERVICE.SQL_UNREGISTER_CONTEXT(
 context_id     IN INTEGER
)
RETURN INTEGER;
```

表 18-5 PKG_SERVICE.SQL_UNREGISTER_CONTEXT 接口说明

参数名称	描述
context_id	打算关闭的 CONTEXT ID 号

- PKG_SERVICE.SQL_SET_SQL

该函数用来解析给定游标的查询语句，被传入的查询语句会立即执行。目前仅支持 SELECT 查询语句的解析，且语句参数仅可通过 text 类型传递，长度不大于 1G。

PKG_SERVICE.SQL_SET_SQL 函数的原型为：

```
PKG_SERVICE.SQL_SET_SQL(
context_id     IN INTEGER,
query_string   IN TEXT,
language_flag  IN INTEGER
)
RETURN BOOLEAN;
```

表 18-6 PKG_SERVICE.SQL_SET_SQL 接口说明

参数名称	描述
context_id	执行查询语句解析的 CONTEXT ID
query_string	执行的查询语句
language_flag	版本语言号，目前只支持 1

- PKG_SERVICE.SQL_RUN

该函数用来执行一个给定的 CONTEXT。该函数接收一个 CONTEXT ID，运行后获得

的数据用于后续操作。目前仅支持 SELECT 查询语句的执行。

PKG_SERVICE.SQL_RUN 函数的原型为：

```
PKG_SERVICE.SQL_RUN(
context_id      IN INTEGER,
)
RETURN INTEGER;
```

表 18-7 PKG_SERVICE.SQL_RUN 接口说明

参数名称	描述
context_id	执行查询语句解析的 CONTEXT ID

- PKG_SERVICE.SQL_NEXT_ROW

该函数返回符合查询条件的数据行数，每一次运行该接口都会获取到新的行数的集合，直到数据读取完毕获取不到新行为止。

PKG_SERVICE.SQL_NEXT_ROW 函数的原型为：

```
PKG_SERVICE.SQL_NEXT_ROW(
context_id      IN INTEGER,
)
RETURN INTEGER;
```

表 18-8 PKG_SERVICE.SQL_NEXT_ROW 接口说明

参数名称	描述
context_id	执行的 CONTEXT ID

- PKG_SERVICE.SQL_GET_VALUE

该函数用来返回给定 CONTEXT 中给定位置的 CONTEXT 元素值，该接口访问的是 PKG_SERVICE.SQL_NEXT_ROW 获取的数据。

PKG_SERVICE.SQL_GET_VALUE 函数的原型为：

```
PKG_SERVICE.SQL_GET_VALUE(
```

```
context_id          IN    INTEGER,
pos                 IN    INTEGER,
col_type            IN    ANYELEMENT
)
RETURN ANYELEMENT;
```

表 18-9 PKG_SERVICE.SQL_GET_VALUE 接口说明

参数名称	描述
context_id	执行的 CONTEXT ID
pos	动态定义列在查询中的位置
col_type	任意类型变量，定义列的返回值类型

- PKG_SERVICE.SQL_SET_RESULT_TYPE

该函数用来定义从给定 CONTEXT 返回的列，该接口只能应用于 SELECT 定义的 CONTEXT 中。定义的列通过查询列表的相对位置来标识，PKG_SERVICE.SQL_SET_RESULT_TYPE 函数的原型为：

```
PKG_SERVICE.SQL_SET_RESULT_TYPE(
context_id      IN INTEGER,
pos             IN INTEGER,
coltype_oid     IN ANYELEMENT,
maxsize         IN INTEGER
)
RETURN INTEGER;
```

表 18-10 PKG_SERVICE.SQL_SET_RESULT_TYPE 接口说明

参数名称	描述
context_id	执行的 CONTEXT ID。

参数名称	描述
pos	动态定义列在查询中的位置。
coltype_oid	任意类型的变量，可根据变量类型得到对应类型 OID。
maxsize	定义的列的长度。

- PKG_SERVICE.JOB_CANCEL

  存储过程 CANCEL 删除指定的定时任务。

  PKG_SERVICE.JOB_CANCEL 函数原型为：

```
PKG_SERVICE.JOB_CANCEL(
job   IN   INTEGER);
```

表 18-11 PKG_SERVICE.JOB_CANCEL 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
id	integer	IN	否	指定的作业号。

示例：

```
CALL PKG_SERVICE.JOB_CANCEL(101);
```

- PKG_SERVICE.JOB_FINISH

  存储过程 FINISH 禁用或者启用定时任务。

  PKG_SERVICE.JOB_FINISH 函数原型为：

```
PKG_SERVICE.JOB_FINISH(
id           IN    INTEGER,
broken        IN    BOOLEAN,
next_time     IN    TIMESTAMP   DEFAULT   sysdate);
```

表 18-12 PKG_SERVICE.JOB_FINISH 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
id	integer	IN	否	指定的作业号。
broken	Boolean	IN	否	状态标志位，true 代表禁用，false 代表启用。根据 true 或 false 值更新当前 job；如果为空值，则不改变原有 job 的状态。
next_time	timestamp	IN	是	下次运行时间，默认为当前系统时间。如果参数 broken 状态为 true，则更新该参数为'4000-1-1'；如果参数 broken 状态为 false，且如果参数 next_time 不为空值，则更新指定 job 的 next_time 值，如果 next_time 为空值，则不更新 next_time 值。该参数可以省略，为默认值。

- PKG_SERVICE.JOB_SUBMIT

  存储过程 JOB_SUBMIT 提交一个系统提供的定时任务。

  PKG_SERVICE.JOB_SUBMIT 函数原型为：

```
PKG_SERVICE.JOB_SUBMIT(
id            IN   BIGINT DEFAULT,
content       IN   TEXT,
next_date     IN   TIMESTAMP DEFAULT sysdate,
interval_time IN   TEXT  DEFAULT 'null',
job           OUT  INTEGER);
```

  说明

  当创建一个定时任务（JOB）时，系统默认将当前数据库和用户名与当前创建的定时任务绑定起来。该接口函数可以通过 call 或 select 调用，如果通过 select 调用，可以不填写出参。如果在存储过程中，则需要通过 perform 调用该接口函数。如果提交的 sql 语句任务使用到非 public 的 schema，应该指定表或者函数的 schema，或者在 sql 语句前添加 set

current_schema = xxx;语句。

表 18-13 PKG_SERVICE.JOB_SUBMIT 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
id	bigint	IN	否	作业号。如果传入 id 为 NULL，则内部会生成作业 ID。
context	text	IN	否	要执行的 SQL 语句。支持一个或多个‘DML’，‘匿名块’，‘调用存储过程的语句’或 3 种混合的场景。
next_time	timestamp	IN	否	下次作业运行时间。默认值为当前系统时间（sysdate）。如果是过去时间，在提交作业时表示立即执行。
interval_time	text	IN	是	用来计算下次作业运行时间的时间表达式，可以是 interval 表达式，也可以是 sysdate 加上一个 numeric 值（例如：sysdate+1.0/24）。如果为空值或字符串”null”表示只执行一次，执行后 JOB 状态 STATUS 变成'd' 不再执行。
job	integer	OUT	否	作业号。范围为 1￣32767。当使用 select 调用 pkg_service.job_submit 时，该参数可以省略。

示例：

```
SELECT PKG_SERVICE.JOB_SUBMIT(NULL, 'call pro_xxx();',
to_date('20180101','yyyymmdd'),'sysdate+1');

SELECT PKG_SERVICE.JOB_SUBMIT(NULL, 'call pro_xxx();',
to_date('20180101','yyyymmdd'),'sysdate+1.0/24');

CALL PKG_SERVICE.JOB_SUBMIT(NULL,'INSERT INTO T_JOB  VALUES(1);  call pro_1();
call pro_2();', add_months(to_date('201701','yyyymm'),1),
'date_trunc(''day'',SYSDATE) + 1 +(8*60+30.0)/(24*60)' ,:jobid);

SELECT PKG_SERVICE.JOB_SUBMIT (101, 'insert_msg_statistic1;', sysdate,
'sysdate+3.0/24');
```

- PKG_SERVICE.JOB_UPDATE

  存储过程 UPDATE 修改定时任务的属性，包括任务内容、下次执行时间、执行间隔。

  PKG_SERVICE.JOB_UPDATE 函数原型为：

```
PKG_SERVICE.JOB_UPDATE(
id              IN   BIGINT,
next_time       IN   TIMESTAMP,
interval_time   IN   TEXT,
content         IN   TEXT);
```

表 18-14 PKG_SERVICE.JOB_UPDATE 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
id	integer	IN	否	指定的作业号。
next_time	timestamp	IN	是	下次运行时间。如果该参数为空值，则不更新指定 job 的 next_time 值，否则更新指定 job 的 next_time 值。

参数	类型	入参/出参	是否可以为空	描述
interval_time	text	IN	是	用来计算下次作业运行时间的时间表达式。如果该参数为空值，则不更新指定 job 的 interval_time 值；如果该参数不为空值，会校验 interval_time 是否为有效的时间类型或 interval 类型，则更新指定 job 的 interval_time 值。如果为字符串”null”表示只执行一次，执行后 JOB 状态 STATUS 变成'd' 不再执行。
content	text	IN	是	执行的存储过程名或者 sql 语句块。如果该参数为空值，则不更新指定 job 的 content 值，否则更新指定 job 的 content 值。

示例：

```
CALL PKG_SERVICE.JOB_UPDATE(101, 'call userproc();', sysdate, 'sysdate +
1.0/1440');
CALL PKG_SERVICE.JOB_UPDATE(101, 'insert into tbl_a values(sysdate);', sysdate,
'sysdate + 1.0/1440');
```

- PKG_SERVICE.SUBMIT_ON_NODES

存储过程 SUBMIT_ON_NODES 创建一个结点上的定时任务，仅 sysadmin/monitor admin 有此权限。

PKG_SERVICE.SUBMIT_ON_NODES 函数原型为：

```
PKG_SERVICE.SUBMIT_ON_NODES(
node_name     IN    TEXT,
database      IN    TEXT
what          IN    TEXT,
next_date     IN    TIMESTAMP DEFAULT sysdate,
```

```
job_interval IN   TEXT  DEFAULT 'null',
job          OUT  INTEGER);
```

表 18-15 PKG_SERVICE.SUBMIT_ON_NODES 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
node_name	text	IN	否	指定作业的执行节点，当前仅支持值为'ALL_NODE'(在所有节点执行)与'CCN'(注：CCN 在集中式/小型化环境下无意义）。
database	text	IN	否	数据库实例作业所使用的 database，节点类型为'ALL_NODE'时仅支持值为'postgres'。
what	text	IN	否	要执行的 SQL 语句。支持一个或多个‘DML’，‘匿名块’，‘调用存储过程的语句’或 3 种混合的场景。
nextdate	timestamp	IN	否	下次作业运行时间。默认值为当前系统时间（sysdate）。如果是过去时间，在提交作业时表示立即执行。
job_interval	text	IN	否	用来计算下次作业运行时间的时间表达式，可以是 interval 表达式，也可以是 sysdate 加上一个 numeric 值（例如：sysdate+1.0/24）。如果为空值或字符串”null”表示只执行一次，执行后 JOB 状态 STATUS 变成'd'不再执行。

参数	类型	入参/出参	是否可以为空	描述
job	integer	OUT	否	作业号。范围为 1—32767。当使用 select 调用 dbms.submit_on_nodes 时，该参数可以省略。

示例：

```
select pkg_service.submit_on_nodes('ALL_NODE', 'postgres', 'select
capture_view_to_json(''dbe_perf.statement'', 0);', sysdate, 'interval ''60
second''');
select pkg_service.submit_on_nodes('CCN', 'postgres', 'select
capture_view_to_json(''dbe_perf.statement'', 0);', sysdate, 'interval ''60
second''');
```

- PKG_SERVICE.ISUBMIT_ON_NODES

ISUBMIT_ON_NODES 与 SUBMIT_ON_NODES 语法功能相同，但其第一个参数是入参，即指定的作业号，SUBMIT 最后一个参数是出参，表示系统自动生成的作业号。仅 sysadmin/monitor admin 有此权限。

- PKG_SERVICE.SQL_GET_ARRAY_RESULT

该函数用来返回绑定的数组类型的 OUT 参数的值，可以用来获取存储过程中的 OUT 参数。

PKG_SERVICE.SQL_GET_ARRAY_RESULT 函数原型为：

```
PKG_SERVICE.SQL_GET_ARRAY_RESULT(
    context_id in int,
    pos in VARCHAR2,
    column_value inout anyarray,
    result_type in anyelement
);
```

表 18-16 PKG_SERVICE.SQL_GET_ARRAY_RESULT 接口说明

参数名称	描述
context_id	想查找的 CONTEXT ID 号。
pos	绑定的参数名。
column_value	返回值。
result_type	返回值类型。

- PKG_SERVICE.SQL_GET_VARIABLE_RESULT

该函数用来返回绑定的非数组类型的 OUT 参数的值，可以用来获取存储过程中的 OUT 参数。

PKG_SERVICE.SQL_GET_VARIABLE_RESULT 函数原型为：

```
PKG_SERVICE.SQL_GET_VARIABLE_RESULT(
    context_id in int,
    pos in VARCHAR2,
    result_type in anyelement
)
RETURNS anyelement;
```

表 18-17 PKG_SERVICE.SQL_GET_VARIABLE_RESULT 接口说明

参数名称	描述
context_id	想查找的 CONTEXT ID 号。
pos	绑定的参数名。
result_type	返回值类型。

18.12.1.2 PKG_UTIL

PKG_UTIL 支持的所有接口请参见下表。

表 18-18 PKG_UTIL

接口名称	描述
PKG_UTIL.LOB_GET_LENGTH	获取 lob 的长度。
PKG_UTIL.LOB_READ	读取 lob 对象的一部分。
PKG_UTIL.LOB_WRITE	将源对象按照指定格式写入到目标对象。
PKG_UTIL.LOB_APPEND	将 lob 源对象指定个数的字符追加到目标 lob 对象。
PKG_UTIL.LOB_COMPARE	根据指定长度比较两个 lob 对象。
PKG_UTIL.LOB_MATCH	返回一个字符串在 LOB 中第 N 次出现的位置。
PKG_UTIL.LOB_RESET	将 lob 的指定位置重置为指定字符。
PKG_UTIL.IO_PRINT	将字符串打印输出。
PKG_UTIL.RAW_GET_LENGTH	获取 raw 的长度。
PKG_UTIL.RAW_CAST_FROM_VARCHAR2	将 varchar2 转化为 raw。

接口名称	描述
PKG_UTIL.RAW_CAST_FROM_BINARY_INTEGER	将 binary integer 转化为 raw。
PKG_UTIL.RAW_CAST_TO_BINARY_INTEGER	将 raw 转化为 binary integer。
PKG_UTIL.SET_RANDOM_SEED	设置随机种子。
PKG_UTIL.RANDOM_GET_VALUE	返回随机值。
PKG_UTIL.FILE_SET_DIRNAME	设置当前操作的目录。
PKG_UTIL.FILE_OPEN	根据指定文件名和设置的目录打开一个文件。
PKG_UTIL.FILE_SET_MAX_LINE_SIZE	设置写入文件一行的最大长度。
PKG_UTIL.FILE_IS_CLOSE	检测一个文件句柄是否关闭。
PKG_UTIL.FILE_READ	从一个打开的文件句柄中读取指定长度的数据。
PKG_UTIL.FILE_READLINE	从一个打开的文件句柄中读取一行数据。
PKG_UTIL.FILE_WRITE	将 BUFFER 中的数据写入到文件中。
PKG_UTIL.FILE_WRITELINE	将 buffer 写入文件，并追加换行符。

接口名称	描述
PKG_UTIL.FILE_NEWLINE	新起一行。
PKG_UTIL.FILE_READ_RAW	从一个打开的文件句柄中读取指定长度的二进制数据。
PKG_UTIL.FILE_WRITE_RAW	将二进制数据写入到文件中。
PKG_UTIL.FILE_FLUSH	将一个文件句柄中的数据写入到物理文件中。
PKG_UTIL.FILE_CLOSE	关闭一个打开的文件句柄。
PKG_UTIL.FILE_REMOVE	删除一个物理文件，操作需要有对应权限。
PKG_UTIL.FILE_RENAME	对于磁盘上的文件进行重命名，类似 Unix 的 mv。
PKG_UTIL.FILE_SIZE	返回文件大小。
PKG_UTIL.FILE_BLOCK_SIZE	返回文件含有的块数量。
PKG_UTIL.FILE_EXISTS	判断文件是否存在。
PKG_UTIL.FILE_GETPOS	返回文件的偏移量，单位字节。
PKG_UTIL.FILE_SEEK	设置文件位置为指定偏移。

接口名称	描述
PKG_UTIL.FILE_CLOSE_ALL	关闭一个会话中打开的所有文件句柄。
PKG_UTIL.EXCEPTION_REPORT_ERROR	抛出一个异常。
PKG_UTIL.RANDOM_SET_SEED	设置一个随机数种子。
pkg_util.app_read_client_info	读取 client_info 信息。
pkg_util.app_set_client_info	设置 client_info 信息。
pkg_util.lob_converttoblob	clob 类型转换成 blob 类型。
pkg_util.lob_converttoclob	blob 类型转换成 clob 类型。
pkg_util.lob_rawtotext	raw 类型转成 text 类型。
pkg_util.lob_reset	清空一个 lob 类型的数据。
pkg_util.lob_texttoraw	text 类型转成 raw 类型。
pkg_util.lob_write	将数据写入 lob 类型。
pkg_util.match_edit_distance_similarity	计算两个字符串的差距。
pkg_util.raw_cast_to_varchar2	raw 类型转成 varchar2 类型。

接口名称	描述
pkg_util.session_clear_context	清空 session_context 中的属性值。
pkg_util.session_search_context	查找一个属性值。
pkg_util.session_set_context	设置一个属性值。
pkg_util.utility_format_call_stack	查看存储过程的调用堆栈。
pkg_util.utility_format_error_backtrace	查看存储过程的错误堆栈。
pkg_util.utility_format_error_stack	查看存储过程的报错信息。
pkg_util.utility_get_time	查看系统 unix 时间戳。

- PKG_UTIL.LOB_GET_LENGTH

  该函数 LOB_GET_LENGTH 获取输入数据的长度。

  PKG_UTIL.LOB_GET_LENGTH 函数原型为：

```
PKG_UTIL.LOB_GET_LENGTH(
lob        IN   anyelement
)
RETURN INTEGER;
```

表 18-19 PKG_UTIL.LOB_GET_LENGTH 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
lob	clob/blob	IN	否	待获取长度的对象。

- PKG_UTIL.LOB_READ

该函数 LOB_READ 读取一个对象，并返回指定部分。

PKG_UTIL.LOB_READ 函数原型为：

```
PKG_UTIL.LOB_READ(
lob        IN    anyelement,
len        IN    int,
start      IN    int,
mode       IN    int
)
RETURN ANYELEMENT
```

表 18-20 PKG_UTIL.LOB_READ 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
lob	clob/blob	IN	否	clob 或者 blob 类型数据。
len	int	IN	否	返回结果长度。
start	int	IN	否	相较于第一个字符的偏移量。
mode	int	IN	否	判断读取操作的类型， 0 ： read; 1 ： trim; 2 ： substr。

- PKG_UTIL.LOB_WRITE

  该函数 LOB_WRITE 将源对象按照指定的参数写入目标对象，并返回目标对象。

  PKG_UTIL.LOB_WRITE 函数原型为：

```
PKG_UTIL.LOB_WRITE(
dest_lob     INOUT    anyelement,
src_lob      IN       varchar2
len          IN       int,
start        IN       int
)
RETURN ANYELEMENT;
```

表 18-21 PKG_UTIL.LOB_WRITE 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
dest_lob	clob/blob	INOUT	否	写入的目标对象。
src_lob	clob/blob	IN	否	被写入的源对象。
len	int	IN	否	源对象的写入长度。
start	int	IN	否	目标对象的写入起始位置。

- PKG_UTIL.LOB_APPEND

  该函数 LOB_APPEND 将源 blob/clob 对象追加到目标 blob/clob 对象，并返回目标对象。

  PKG_UTIL.LOB_APPEND 函数原型为：

```
PKG_UTIL.LOB_APPEND(
dest_lob     INOUT    blob,
src_lob      IN       blob,
len          IN       int default NULL
)
RETURN BLOB;

PKG_UTIL.LOB_APPEND(
dest_lob     INOUT    clob,
src_lob      IN       clob,
len          IN       int default NULL
)
RETURN CLOB;
```

表 18-22 PKG_UTIL.LOB_APPEND 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
dest_lob	blob/clob	INOUT	否	写入的目标 blob/clob 对象。
src_lob	blob/clob	IN	否	被写入的源 blob/clob 对象。
len	int	IN	是	写入源对象的长度，为 NULL 则默认写入源对象全部。

- PKG_UTIL.LOB_COMPARE

该函数 LOB_COMPARE 按照指定的起始位置、个数比较对象是否相同，lob1 大则返回 1，lob2 大返回-1，相等返回 0。

PKG_UTIL.LOB_COMPARE 函数原型为：

```
PKG_UTIL.LOB_COMPARE(
lob1        IN   anyelement,
lob2        IN   anyelement,
len         IN   int,
start1      IN   int,
start2      IN   int
)
RETURN INTEGER;
```

表 18-23 PKG_UTIL.LOB_COMPARE 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
lob1	clob/blob	IN	否	待比较的字符串。
lob2	clob/blob	IN	否	待比较的字符串。

参数	类型	入参/出参	是否可以为空	描述
len	int	IN	否	比较的长度。
start1	int	IN	否	lob1 起始偏移量。
start2	int	IN	否	lob2 起始偏移量。

- PKG_UTIL.LOB_MATCH

  该函数 LOB_MATCH 返回 pattern 出现在 lob 对象中第 match_nth 次的位置。

  PKG_UTIL.LOB_MATCH 函数原型为：

```
PKG_UTIL.LOB_MATCH(
lob           IN   anyelement,
pattern       IN   anyelement,
start         IN   int,
match_nth     IN   int
)
RETURN INTEGER;
```

表 18-24 PKG_UTIL.LOB_MATCH 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
lob	clob/blob	IN	否	待比较的字符串。
pattern	clob/blob	IN	否	待匹配的 pattern。
start	int	IN	否	lob 的起始比较位置。

参数	类型	入参/出参	是否可以为空	描述
match_nth	int	IN	否	第几次匹配到。

- PKG_UTIL.LOB_RESET

  该函数 LOB_RESET 清除一段数据为字符 value。

  PKG_UTIL.LOB_RESET 函数原型为：

```
PKG_UTIL.LOB_RESET(
lob           INOUT   bytea,
len           INOUT   int,
start         IN   int DEFAUTL 1,
value         IN   char default 0
)
RETURN record;
```

表 18-25 PKG_UTIL.LOB_RESET 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
lob	bytea	IN	否	待重置的字符串。
len	int	IN	否	重置的长度。
start	int	IN	否	重置的起始位置。
value	int	IN	是	设置的字符。默认值‘0’。

- PKG_UTIL.IO_PRINT

  该函数 IO_PRINT 将一段字符串打印输出。

PKG_UTIL.IO_PRINT 函数原型为：

```
PKG_UTIL.IO_PRINT(
format       IN   text,
is_one_line  IN   boolean
)
RETURN void;
```

表 18-26 PKG_UTIL.IO_PRINT 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
format	text	IN	否	待打印输出的字符串。
is_one_line	boolean	IN	否	是否输出为一行。

- PKG_UTIL.RAW_GET_LENGTH

  该函数 RAW_GET_LENGTH 获取 raw 的长度。

```
PKG_UTIL.RAW_GET_LENGTH 函数原型为：
PKG_UTIL.RAW_GET_LENGTH(
value       IN   raw
)
RETURN integer;
```

表 18-27 PKG_UTIL.RAW_GET_LENGTH 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
raw	raw	IN	否	待获取长度的对象。

- PKG_UTIL.RAW_CAST_FROM_VARCHAR2

  该函数 RAW_CAST_FROM_VARCHAR2 将 varchar2 转化为 raw。

PKG_UTIL.RAW_CAST_FROM_VARCHAR2 函数原型为：

```
PKG_UTIL.RAW_CAST_FROM_VARCHAR2(
str        IN    varchar2
)
RETURN raw;
```

表 18-28 PKG_UTIL.RAW_CAST_FROM_VARCHAR2 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
str	varchar2	IN	否	需要转化的源数据。

- PKG_UTIL.CAST_FROM_BINARY_INTEGER

  该函数 CAST_FROM_BINARY_INTEGER 将 binary integer 数据转化为 raw 。

  PKG_UTIL.CAST_FROM_BINARY_INTEGER 函数原型为：

```
PKG_UTIL.CAST_FROM_BINARY_INTEGER(
value        IN    integer,
endianess    IN    integer
)
RETURN raw;
```

表 18-29 PKG_UTIL.CAST_FROM_BINARY_INTEGER 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
value	integer	IN	否	需要转化为 raw 的源数据。
endianess	integer	IN	否	表示字节序的整型值 1 或 2（1 代表 BIG_ENDIAN，2 代表 LITTLE-ENDIAN）。

- PKG_UTIL.CAST_TO_BINARY_INTEGER

  该函数 CAST_TO_BINARY_INTEGER 将 raw 数据转化为 binary integer 。

  PKG_UTIL.CAST_TO_BINARY_INTEGER 函数原型为：

```
PKG_UTIL.CAST_TO_BINARY_INTEGER(
value       IN   raw,
endianess   IN   integer
)
RETURN integer;
```

表 18-30 PKG_UTIL.CAST_TO_BINARY_INTEGER 接口参数说明

参数	类型	入参/出参	是否可以为空	描述
value	raw	IN	否	需要转化为 binary integer 的 raw 类型源数据。
endianess	integer	IN	否	表示字节序的整型值 1 或 2（1 代表 BIG_ENDIAN，2 代表 LITTLE-ENDIAN）。

- PKG_UTIL.SET_RANDOM_SEED

  该函数 SET_RANDOM_SEED 设置随机数种子。

  PKG_UTIL.SET_RANDOM_SEED 函数原型为：

```
PKG_UTIL.RANDOM_SET_SEED(
seed         IN   int
)
RETURN integer;
```

表 18-31 PKG_UTIL.SET_RANDOM_SEED 接口参数说明

参数	类型	入参/出参	是否可以为空	描述

参数	类型	入参/出参	是否可以为空	描述
seed	int	IN	否	随机数种子。

- PKG_UTIL.RANDOM_GET_VALUE

该函数 RANDOM_GET_VALUE 返回 0~1 区间的一个随机数。

PKG_UTIL.RANDOM_GET_VALUE 函数原型为：

```
PKG_UTIL.RANDOM_GET_VALUE(
)
RETURN numeric;
```

- PKG_UTIL.FILE_SET_DIRNAME

设置当前操作的目录，基本上所有涉及单个目录的操作，都需要调用此方法先设置操作的目录。

PKG_UTIL.FILE_SET_DIRNAME 函数原型为：

```
PKG_UTIL.FILE_SET_DIRNAME(
dir  IN  text
)
RETURN bool
```

表 18-32 PKG_UTIL.FILE_SET_DIRNAME 接口参数说明

参数	描述
dirname	文件的目录位置，这个字符串是一个目录对象名。 说明： 文件目录的位置，需要添加到系统表 PG_DIRECTORY 中，如果传入的路径和 PG_DIRECTORY 中的路径不匹配，会报路径不存在的错误，下面的涉及 location 作为参数的函数也是同样的情况。

- PKG_UTIL.FILE_OPEN

该函数用来打开一个文件，最多可以同时打开 50 个文件。并且该函数返回 INTEGER 类型的一个句柄。

PKG_UTIL.FILE_OPEN 函数原型为：

```
PKG_UTIL.FILE_OPEN(
file_name     IN  text,
open_mode     IN  integer)
```

表 18-33 PKG_UTIL.FILE_OPEN 接口参数说明

参数	描述
file_name	文件名，包含扩展（文件类型），不包括路径名。如果文件名中包含路径，在 OPEN 中会被忽略，在 Unix 系统中，文件名不能以/.结尾。
open_mode	指定文件的打开模式，包含 r：read text，w：write text 和 a：append text。 说明： 对于写操作，会检测文件类型，如果写入 elf 文件，将会报错并退出。

- PKG_UTIL.FILE_SET_MAX_LINE_SIZE

  设置写入文件一行的最大长度。

  PKG_UTIL.FILE_SET_MAX_LINE_SIZE 函数原型为：

```
PKG_UTIL.FILE_SET_MAX_LINE_SIZE(
max_line_size in integer)
RETURN BOOL
```

表 18-34 PKG_UTIL.FILE_SET_MAX_LINE_SIZE 接口参数说明

参数	描述
max_line_size	每行最大字符数，包含换行符（最小值是 1，最大值是 32767）。如果没有指定，会指定一个默认值 1024。

- PKG_UTIL.FILE_IS_CLOSE

检测一个文件句柄是否关闭。

PKG_UTIL.FILE_IS_CLOSE 函数原型为：

```
PKG_UTIL.FILE_IS_CLOSE(
file in integer
)
RETURN BOOL
```

表 18-35 PKG_UTIL.FILE_IS_CLOSE 接口参数说明

参数	描述
file	一个打开的文件句柄。

- PKG_UTIL.FILE_READ

  根据指定的长度从一个打开的文件句柄中读取出数据。

  PKG_UTIL.FILE_READ 函数原型为：

```
PKG_UTIL.FILE_READ(
file      IN   integer,
buffer    OUT  text,
len       IN   integer)
```

表 18-36 PKG_UTIL.FILE_READ 接口参数说明

参数	描述
file	通过调用 OPEN 打开的文件句柄，文件必须以读的模式打开，否则会抛出 INVALID_OPERATION 的异常。
buffer	用于接收数据的 BUFFER。
len	从文件中读取的字节数。

- PKG_UTIL.FILE_READLINE

根据指定的长度从一个打开的文件句柄中读取出一行数据。

PKG_UTIL.FILE_READLINE 函数原型为：

```
PKG_UTIL.FILE_READLINE(
file    IN  integer,
buffer  OUT text,
len     IN  integer default 1024)
```

表 18-37 PKG_UTIL.FILE_READLINE 接口参数说明

参数	描述
file	通过调用 OPEN 打开的文件句柄，文件必须以读的模式打开，否则会抛出 INVALID_OPERATION 的异常。
buffer	用于接收数据的 BUFFER。
len	从文件中读取的字节数，默认是 NULL。如果是默认 NULL，会使用 max_line_size 来指定大小。

- PKG_UTIL.FILE_WRITE

  将 BUFFER 中指定的数据写入到文件中。

  PKG_UTIL.FILE_WRITE 函数原型为：

```
PKG_UTIL.FILE_WRITE(
file in integer,
buffer in text
)
RETURN BOOL
```

表 18-38 PKG_UTIL.FILE_WRITE 接口参数说明

参数	描述
file	一个打开的文件句柄。

参数	描述
buffer	要写入文件的文本数据，BUFFER 的最大值是 32767 个字节。如果没有指定值，默认是 1024 个字节，没有刷新到文件之前，一系列的 PUT 操作的 BUFFER 总和不能超过 32767 个字节。 说明： 对于写操作，会检测文件类型，如果写入 elf 文件，将会报错并退出。

- PKG_UTIL.FILE_NEWLINE

  向一个打开的文件中写入一个行终结符。行终结符和平台相关。

  PKG_UTIL.FILE_NEWLINE 函数原型为：

```
PKG_UTIL.FILE_NEWLINE(
file in integer
)
RETURN BOOL
```

表 18-39 PKG_UTIL.FILE_NEWLINE 接口参数说明

参数	描述
file	一个打开的文件句柄。

- PKG_UTIL.FILE_WRITELINE

  向一个打开的文件中写入一行。

  PKG_UTIL.FILE_WRITELINE 函数原型为：

```
PKG_UTIL.FILE_WRITELINE(
file in integer,
buffer in text,
flush in bool default false
)
RETURN BOOL
```

表 18-40 PKG_UTIL.FILE_WRITELINE 接口参数说明

参数	描述
file	一个打开的文件句柄。
buffer	要写入的内容。
flush	是否落盘。

- PKG_UTIL.FILE_READ_RAW

从一个打开的文件句柄中读取指定长度的二进制数据，返回读取的二进制数据，返回类型为 raw。

PKG_UTIL.FILE_READ_RAW 函数原型为：

```
PKG_UTIL.FILE_READ_RAW(
file       in integer,
length     in integer default NULL
)
RETURN raw
```

表 18-41 PKG_UTIL.FILE_READ_RAW 接口参数说明

参数	描述
file	一个打开的文件句柄。
length	要读取的长度，默认为 NULL。默认情况下读取文件中所有数据，最大为 1G。

- PKG_UTIL.FILE_WRITE_RAW

向一个打开的文件中写入传入二进制对象 RAW。插入成功返回 true。

PKG_UTIL.FILE_WRITE_RAW 函数原型为：

```
PKG_UTIL.FILE_WRITE_RAW(
file in integer,
```

```
r    in raw
)
RETURN BOOL
```

表 18-42 PKG_UTIL.FILE_NEWLINE 接口参数说明

参数	描述
file	一个打开的文件句柄。
r	准备传入文件的数据 说明：对于写操作，会检测文件类型，如果写入 elf 文件，将会报错并退出。

- PKG_UTIL.FILE_FLUSH

一个文件句柄中的数据要写入到物理文件中，缓冲区中的数据必须要有一个行终结符。当文件必须在打开时读取，刷新非常有用。例如，调试信息可以刷新到文件中，以便立即读取。

PKG_UTIL.FILE_FLUSH 函数原型为：

```
PKG_UTIL.FILE_FLUSH (
file in integer
)
RETURN VOID
```

表 18-43 PKG_UTIL.FILE_FLUSH 接口参数说明

参数	描述
file	一个打开的文件句柄。

- PKG_UTIL.FILE_CLOSE

关闭一个打开的文件句柄。

PKG_UTIL.FILE_CLOSE 函数原型为：

```
PKG_UTIL.FILE_CLOSE (
```

```
file in integer
)
RETURN BOOL
```

表 18-44 PKG_UTIL.FILE_CLOSE 接口参数说明

参数	描述
file	一个打开的文件句柄。

- PKG_UTIL.FILE_REMOVE

  删除一个磁盘文件，操作的时候需要有充分的权限。

  PKG_UTIL.FILE_REMOVE 函数原型为：

```
PKG_UTIL.FILE_REMOVE(
file_name in text
)
RETURN VOID
```

表 18-45 PKG_UTIL.FILE_REMOVE 接口参数说明

参数	描述
filen_ame	要删除的文件名

- PKG_UTIL.FILE_RENAME

  对于磁盘上的文件进行重命名，类似 Unix 的 mv。

  PKG_UTIL.FILE_RENAME 函数原型为：

```
PKG_UTIL.FILE_RENAME(
text src_dir in text,
text src_file_name in text,
text dest_dir in text,
text dest_file_name in text,
 overwrite boolean default false)
```

表 18-46 PKG_UTIL.FILE_RENAME 接口参数说明

参数	描述
src_dir	源文件目录（大小写敏感）。
src_file_name	源文件名。
dest_dir	目标文件目录（大小写敏感）。
dest_file_name	目标文件名。
overwrite	默认是 false，如果目的目录下存在一个同名的文件，不会进行重写。

- PKG_UTIL.FILE_SIZE

  返回指定的文件大小。

  PKG_UTIL.FILE_SIZE 函数原型为：

```
bigint PKG_UTIL.FILE_SIZE(
file_name in text
)
```

表 18-47 PKG_UTIL.FILE_SIZE 接口参数说明

参数	描述
file_name	文件名

- PKG_UTIL.FILE_BLOCK_SIZE

  返回指定的文件含有的块数量。

  PKG_UTIL.FILE_BLOCK_SIZE 函数原型为：

```
bigint PKG_UTIL.FILE_BLOCK_SIZE(
file_name in text
)
```

表 18-48 PKG_UTIL.FILE_BLOCK_SIZE 接口参数说明

参数	描述
file_name	文件名

- PKG_UTIL.FILE_EXISTS

  判断指定的文件是否存在。

  PKG_UTIL.FILE_EXISTS 函数原型为：

```
PKG_UTIL.FILE_EXISTS(
file_name in text
)
RETURN BOOL
```

表 18-49 PKG_UTIL.FILE_EXISTS 接口参数说明

参数	描述
file_name	文件名

- PKG_UTIL.FILE_GETPOS

  返回文件的偏移量，单位字节。

  PKG_UTIL.FILE_GETPOS 函数原型为：

```
PKG_UTIL.FILE_GETPOS(
file in integer
)
RETURN BIGINT
```

表 18-50 PKG_UTIL.FILE_GETPOS 接口参数说明

参数	描述
file	一个打开的文件句柄。

- PKG_UTIL.FILE_SEEK

  根据用户指定的字节数向前或者向后调整文件指针的位置。

  PKG_UTIL.FILE_SEEK 函数原型为：

```
void PKG_UTIL.FILE_SEEK(
file in integer,
start in bigint default null
)
RETURN VOID
```

表 18-51 PKG_UTIL.FILE_SEEK 接口参数说明

参数	描述
file	一个打开的文件句柄。
start	文件偏移，字节。

- PKG_UTIL.FILE_CLOSE_ALL

  关闭一个会话中打开的所有的文件句柄。

  PKG_UTIL.FILE_CLOSE_ALL 函数原型为：

```
PKG_UTIL.FILE_CLOSE_ALL(
)
RETURN VOID↵
PKG_UTIL.EXCEPTION_REPORT_ERROR
```

  抛出一个异常。

  PKG_UTIL.EXCEPTION_REPORT_ERROR 函数原型为：

```
PKG_UTIL.EXCEPTION_REPORT_ERROR(
code integer,
log text,
flag boolean DEFAULT false
)
RETURN INTEGER
```

表 18-52 PKG_UTIL.EXCEPTION_REPORT_ERROR 接口参数说明

参数	描述
code	抛异常所打印的错误码。
log	抛异常所打印的日志提示信息。
flag	保留字段，默认为 false。

- PKG_UTIL.app_read_client_info

  读取 client_info 信息。

  PKG_UTIL.app_read_client_info 函数原型为：

```
PKG_UTIL.app_read_client_info(
OUT buffer text
)
```

表 18-53 PKG_UTIL.app_read_client_info 接口参数说明

参数	描述
buffer	返回的 client_info 信息。

- PKG_UTIL.app_set_client_info

  设置 client_info 信息。

  PKG_UTIL.app_set_client_info 函数原型为：

```
PKG_UTIL.app_set_client_info(
str text
)
RETURN INTEGER
```

表 18-54 PKG_UTIL.app_set_client_info 接口参数说明

参数	描述

参数	描述
str	要设置的 client_info 信息。

- PKG_UTIL.lob_converttoblob

  将 clob 转成 blob，amout 为要转换的长度。

  PKG_UTIL.lob_converttoblob 函数原型为：

```
PKG_UTIL.lob_converttoblob(
dest_lob blob,
src_clob clob,
amount integer,
dest_offset integer,
src_offset integer
)
```

表 18-55 PKG_UTIL.lob_converttoblob 接口参数说明

参数	描述
dest_lob	目标 lob。
src_clob	要转换的 clob。
amount	转换的长度。
dest_offset	目标 lob 的起始位置。
src_offset	源 clob 的起始位置。

- PKG_UTIL.lob_converttoclob

  将 blob 转成 clob，amout 为要转换的长度。

  PKG_UTIL.lob_converttoclob 函数原型为：

```
PKG_UTIL.lob_converttoclob(
dest_lob clob,
src_blob blob,
amount integer,
dest_offset integer,
src_offset integer
)
```

表 18-56 PKG_UTIL.lob_converttoclob 接口参数说明

参数	描述
dest_lob	目标 lob。
src_blob	要转换的 blob。
amount	转换的长度。
dest_offset	目标 lob 的起始位置。
src_offset	源 clob 的起始位置。

- PKG_UTIL.lob_texttoraw

  将 text 转成 raw。

  PKG_UTIL.lob_texttoraw 函数原型为：

```
PKG_UTIL.lob_texttoraw(
src_lob clob
)
RETURN raw
```

表 18-57 PKG_UTIL.lob_texttoraw 接口参数说明

参数	描述

参数	描述
src_lob	要转换的 lob 数据。

- PKG_UTIL.match_edit_distance_similarity

  计算两个字符串的差别。

  PKG_UTIL.match_edit_distance_similarity 函数原型为：

```
PKG_UTIL.match_edit_distance_similarity(
str1 text,
str2 text
)
RETURN INTEGER
```

表 18-58 PKG_UTIL.match_edit_distance_similarity 接口参数说明

参数	描述
str1	第一个字符串。
str2	第二个字符串。

- PKG_UTIL.raw_cast_to_varchar2

  raw 类型转成 varchar2。

  PKG_UTIL.raw_cast_to_varchar2 函数原型为：

```
PKG_UTIL.raw_cast_to_varchar2(
str1 text,
str2 text
)
RETURN INTEGER
```

表 18-59 PKG_UTIL.raw_cast_to_varchar2 接口参数说明

参数	描述

参数	描述
str1	第一个字符串。
str2	第二个字符串。

- PKG_UTIL.session_clear_context

  清除 session_context 信息。

  PKG_UTIL.session_clear_context 函数原型为：

```
PKG_UTIL.session_clear_context(
namespace text,
client_identifier text,
attribute text
)
RETURN INTEGER
```

表 18-60 PKG_UTIL.session_clear_context 接口参数说明

参数	描述
namespace	属性的命名空间。
client_identifier	client_identifier，一般与 namespace 即可，当为 null 时，默认修改所有 namesapce。
attribute	要清除的属性值。

- PKG_UTIL.session_search_context

  查找属性值。

  PKG_UTIL.session_clear_context 函数原型为：

```
PKG_UTIL.session_clear_context(
namespace text,
```

```
attribute text
)
RETURN INTEGER
```

表 18-61 PKG_UTIL.session_clear_context 接口参数说明

参数	描述
namespace	属性的命名空间。
attribute	要清除的属性值。

- PKG_UTIL.session_set_context

  设置属性值。

  PKG_UTIL.session_set_context 函数原型为：

```
PKG_UTIL.session_set_context(
namespace text,
attribute text,
value text
)
RETURN INTEGER
```

表 18-62 PKG_UTIL.session_set_context 接口参数说明

参数	描述
namespace	属性的命名空间
attribute	要设置的属性
value	属性对应的值

- PKG_UTIL.utility_get_time

  打印 unix 时间戳。

PKG_UTIL.utility_get_time 函数原型为：

```
PKG_UTIL.utility_get_time()
RETURN text
```

- PKG_UTIL.utility_format_error_backtrace

  查看存储过程调用堆栈。

  PKG_UTIL.utility_format_error_backtrace 函数原型为：

```
PKG_UTIL.utility_format_error_backtrace()
RETURN text
```

- PKG_UTIL.utility_format_error_stack

  查看存储过程错误信息。

  PKG_UTIL.utility_format_error_stack 函数原型为：

```
PKG_UTIL.utility_format_error_stack()
RETURN text
```

- PKG_UTIL.utility_format_call_stack

  查看存储过程调用堆栈。

  PKG_UTIL.utility_format_call_stack 函数原型为：

```
PKG_UTIL.utility_format_call_stack()
RETURN text
```

18.13 Retry 管理

Retry 是数据库在 SQL 或存储过程（包含匿名块）执行失败时，在数据库内部进行重新执行的过程，以提高执行成功率和用户体验。数据库内部通过检查发生错误时的错误码及 Retry 相关配置，决定是否进行重试。

失败时回滚之前执行的语句，并重新执行存储过程进行 Retry。

示例：

```
postgres=# CREATE OR REPLACE PROCEDURE retry_basic ( IN  x INT)
AS
BEGIN
      INSERT INTO t1 (a) VALUES (x);
      INSERT INTO t1 (a) VALUES (x+1);
END;
```

```
/

postgres=# CALL retry_basic(1);
```

18.14 调试

18.14.1语法

18.14.1.1 RAISE 语法

有以下五种语法格式：

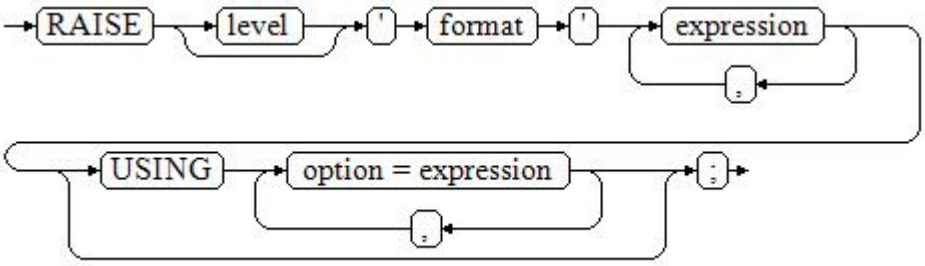

图 18-37 raise_format::=

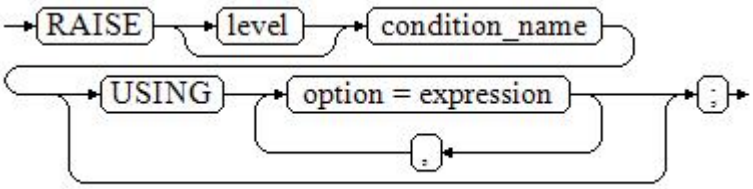

图 18-38 raise_condition::=

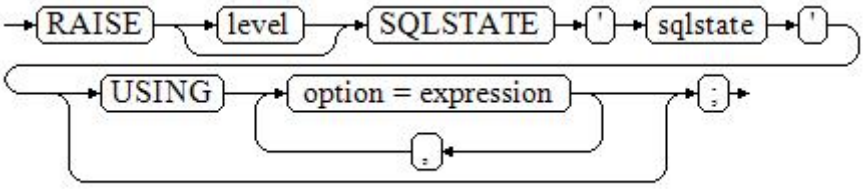

图 18-39 raise_sqlstate::=

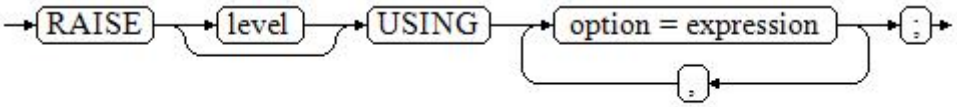

图 18-40raise_option::=

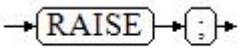

图 18-41 raise::=

参数说明：

level 选项用于指定错误级别，有 DEBUG，LOG，INFO，NOTICE，WARNING 以及 EXCEPTION（默认值）。EXCEPTION 抛出一个正常终止当前事务的异常，其他的仅产生不同异常级别的信息。特殊级别的错误信息是否报告到客户端、写到服务器日志由 log_min_messages 和 client_min_messages 这两个配置参数控制。

format：格式字符串，指定要报告的错误消息文本。格式字符串后可跟表达式，用于向消息文本中插入。在格式字符串中，%由 format 后面跟着的参数的值替换，%%用于打印出%。例如：

--v_job_id 将替换字符串中的 %：

```
RAISE NOTICE 'Calling cs_create_job(%)',v_job_id;
```

option = expression：向错误报告中添加另外的信息。关键字 option 可以是 MESSAGE、DETAIL、HINT 以及 ERRCODE，并且每一个 expression 可以是任意的字符串。

MESSAGE，指定错误消息文本，这个选项不能用于在 USING 前包含一个格式字符串的 RAISE 语句中。

DETAIL，说明错误的详细信息。

HINT，用于打印出提示信息。

ERRCODE，向报告中指定错误码（SQLSTATE）。可以使用条件名称或者直接用五位字符的 SQLSTATE 错误码。

condition_name：错误码对应的条件名。

sqlstate：错误码。

如果在 RAISE EXCEPTION 命令中既没有指定条件名也没有指定 SQLSTATE，默认用

RAISE EXCEPTION (P0001)。如果没有指定消息文本，默认用条件名或者 SQLSTATE 作为消息文本。

须知

- 当由 SQLSTATE 指定了错误码，则不局限于已定义的错误码，可以选择任意包含五个数字或者大写的 ASCII 字母的错误码，而不是 00000。建议避免使用以三个 0 结尾的错误码，因为这种错误码是类别码，会被整个种类捕获。
- 兼容 O 模式下，SQLCODE 等于 SQLSTATE。

说明

- 图 18-41 所示的语法不接任何参数。这种形式仅用于一个 BEGIN 块中的 EXCEPTION 语句，它使得错误重新被处理。

18.14.1.2 EXCEPTION_INIT 语法

兼容 O 模式下，支持使用 EXCEPTION_INIT 语法自定义错误码 SQLCODE。语法格式如下：

图 18-42 exception_init::=

参数说明：

exception_name 为用户申明的异常名，EXCEPTION_INIT 语法必须出现在与申明异常相同部分，位于申明异常之后。

sqlcode 为自定义的 SQLCODE，必须为负整数，取值范围-2147483647~-1。

须知

- 使用 EXCEPTION_INIT 语法自定义错误码 SQLCODE 时，SQLSTATE 与 SQLCODE 相同，SQLERRM 格式为” xxx：non-GaussDB Exception”。比如自定义 SQLCODE=-1，则 SQLSTATE=“-1”，SQLERRM=” 1： non-GaussDB Exception”。

18.14.2示例

终止事务时，给出错误和提示信息：

```
CREATE OR REPLACE PROCEDURE proc_raise1(user_id in integer)
AS
BEGIN
RAISE EXCEPTION 'Noexistence ID --> %',user_id USING HINT = 'Please check your
user ID';
END;
/

call proc_raise1(300011);

--执行结果
ERROR:  Noexistence ID --> 300011
HINT:  Please check your user ID
```

两种设置 SQLSTATE 的方式，其中一种方式为：

```
CREATE OR REPLACE PROCEDURE proc_raise2(user_id in integer)
AS
BEGIN
RAISE 'Duplicate user ID: %',user_id USING ERRCODE = 'unique_violation';
END;
/

\set VERBOSITY verbose
call proc_raise2(300011);

--执行结果
ERROR:  Duplicate user ID: 300011
SQLSTATE: 23505
```

如果主要的参数是条件名或者是 SQLSTATE，可以使用：

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

例如：

```
CREATE OR REPLACE PROCEDURE division(div in integer, dividend in integer)
AS
DECLARE
res int;
    BEGIN
    IF dividend=0 THEN
        RAISE division_by_zero;
        RETURN;
```

```
    ELSE
        res := div/dividend;
        RAISE INFO 'division result: %', res;
        RETURN;
    END IF;
    END;
/
call division(3,0);

--执行结果
ERROR:  division_by_zero
```

或者另一种方式:

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

兼容 O 模式下，支持使用语法 EXCEPTION_INIT 自定义错误码 SQLCODE:

```
declare
    deadlock_detected exception;
    pragma exception_init(deadlock_detected, -1);
begin
    if 1 > 0 then
        raise deadlock_detected;
    end if;
exception
    when deadlock_detected then
        raise notice
'sqlcode:%,sqlstate:%,sqlerrm:%',sqlcode,sqlstate,sqlerrm;
end;
/
--执行结果
NOTICE:  sqlcode:-1,sqlstate:-1,sqlerrm: 1: non-GaussDB Exception
```

19 PL/pgSQL 语言函数

PL/pgSQL 是一种可载入的过程语言。

用 PL/pgSQL 创建的函数可以被用在任何可以使用内建函数的地方。例如，可以创建复杂条件的计算函数并且后面用它们来定义操作符或把它们用于索引表达式。

SQL 被大多数数据库用作查询语言。它是可移植的并且容易学习。但是每一个 SQL 语句必须由数据库服务器单独执行。

这意味着客户端应用必须发送每一个查询到数据库服务器、等待它被处理、接收并处理结果、做一些计算，然后发送更多查询给服务器。如果客户端和数据库服务器不在同一台机器上，则会引起进程间通信并且将带来网络负担。

通过 PL/pgSQL，可以将一整块计算和一系列查询分组在数据库服务器内部，这样就有了一种过程语言的能力并且使 SQL 更易用，同时能节省客户端/服务器通信开销。

客户端和服务器之间的额外往返通信被消除。

客户端不需要的中间结果不必被整理或者在服务器和客户端之间传送。

多轮的查询解析可以被避免。

PL/pgSQL 可以使用 SQL 中所有的数据类型、操作符和函数。一些常见函数，例如 gs_extend_library。

应用 PL/pgSQL 创建函数的语法为 CREATE FUNCTION。PL/pgSQL 是一种可载入的过程语言。其应用方法与 18 存储过程相似，只是存储过程无返回值，函数有返回值。

20 触发器

触发器会在指定的数据库事件发生时自动执行函数。

20.1 语法格式

创建触发器

```
CREATE TRIGGER trigger_name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments );
```

修改触发器

```
ALTER TRIGGER trigger_name ON table_name RENAME TO new_trigger_name;
```

删除触发器

```
DROP TRIGGER trigger_name ON table_name [ CASCADE | RESTRICT ];
```

20.2 参数说明

- trigger_name

  触发器名称。

- BEFORE

  触发器函数是在触发事件发生前执行。

- AFTER

  触发器函数是在触发事件发生后执行。

- INSTEAD OF

  触发器函数直接替代触发事件。

- event

  启动触发器的事件，取值范围包括：INSERT、UPDATE、DELETE 或 TRUNCATE，也可以通过 OR 同时指定多个触发事件。

- table_name

  触发器对应的表名称。

- FOR EACH ROW | FOR EACH STATEMENT

  触发器的触发频率。

  FOR EACH ROW 是指该触发器是受触发事件影响的每一行触发一次。

  FOR EACH STATEMENT 是指该触发器是每个 SQL 语句只触发一次。

  未指定时默认值为 FOR EACH STATEMENT。约束触发器只能指定为 FOR EACH ROW。

- function_name

  用户定义的函数，必须声明为不带参数并返回类型为触发器，在触发器触发时执行。

- arguments

  执行触发器时要提供给函数的可选的以逗号分隔的参数列表。

- new_trigger_name

  修改后的新触发器名称。

20.3 示例

```
--创建源表及触发表
postgres=# CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);
postgres=# CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);

--创建触发器函数
postgres=# CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
           $$
           DECLARE
           BEGIN
                   INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2,
NEW.id3);
                   RETURN NEW;
           END
           $$ LANGUAGE PLPGSQL;

postgres=# CREATE OR REPLACE FUNCTION tri_update_func() RETURNS TRIGGER AS
           $$
           DECLARE
           BEGIN
                   UPDATE test_trigger_des_tbl SET id3 = NEW.id3 WHERE
id1=OLD.id1;
```

```
                RETURN OLD;
        END
        $$ LANGUAGE PLPGSQL;

postgres=# CREATE OR REPLACE FUNCTION TRI_DELETE_FUNC() RETURNS TRIGGER AS
        $$
        DECLARE
        BEGIN
                DELETE FROM test_trigger_des_tbl WHERE id1=OLD.id1;
                RETURN OLD;
        END
        $$ LANGUAGE PLPGSQL;

--创建 INSERT 触发器
postgres=# CREATE TRIGGER insert_trigger
        BEFORE INSERT ON test_trigger_src_tbl
        FOR EACH ROW
        EXECUTE PROCEDURE tri_insert_func();

--创建 UPDATE 触发器
postgres=# CREATE TRIGGER update_trigger
        AFTER UPDATE ON test_trigger_src_tbl
        FOR EACH ROW
        EXECUTE PROCEDURE tri_update_func();

--创建 DELETE 触发器
postgres=# CREATE TRIGGER delete_trigger
        BEFORE DELETE ON test_trigger_src_tbl
        FOR EACH ROW
        EXECUTE PROCEDURE tri_delete_func();

--执行 INSERT 触发事件并检查触发结果
postgres=# INSERT INTO test_trigger_src_tbl VALUES(100,200,300);
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效。

--执行 UPDATE 触发事件并检查触发结果
postgres=# UPDATE test_trigger_src_tbl SET id3=400 WHERE id1=100;
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效

--执行 DELETE 触发事件并检查触发结果
```

```
postgres=# DELETE FROM test_trigger_src_tbl WHERE id1=100;
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效

--修改触发器
postgres=# ALTER TRIGGER delete_trigger ON test_trigger_src_tbl RENAME TO
delete_trigger_renamed;

--删除触发器
postgres=# DROP TRIGGER insert_trigger ON test_trigger_src_tbl;
postgres=# DROP TRIGGER update_trigger ON test_trigger_src_tbl;
postgres=# DROP TRIGGER delete_trigger_renamed ON test_trigger_src_tbl;
```

21 全文检索

21.1 介绍

21.1.1 全文检索概述

文本搜索操作符在数据库中已存在多年。GBase 8s 为文本数据类型提供~、~*、LIKE 和 ILIKE 操作符；但它们缺乏现代信息系统所要求的许多必要属性。这些缺憾可以通过使用索引及词典进行解决。

文本检索缺乏信息系统所要求的必要属性：

没有语义支持，即使是英语。

由于要识别派生词并不是那么容易，因此正则表达式也不能满足要求。如， satisfies 和 satisfy，当使用正则表达式寻找 satisfy 时，并不会查询到包含 satisfies 的文档。用户可以使用 OR 搜索多种派生形式，但过程非常繁琐。并且有些词会有上千的派生词，因此容易出错。

没有对搜索结果的分类（排序）。当搜索出成千的文档时，查找效率很低。

由于没有索引的支持，每一次的搜索需要遍历所有的文档，整体搜索比较缓慢。

使用全文索引可以对文档进行预处理，并且可以使后续的搜索更快速。预处理过程包括：

- 将文档解析成 token。

为每个文档标记不同类别的 token 是非常有必要的，例如：数字、文字、复合词、电子邮件地址，这样就可以做不同的处理。原则上 token 的类别依赖于具体的应 用，但对于大多数的应用来说，可以使用一组预定义的 token 类。

- 将 token 转换为词素。

词素像 token 一样是一个字符串，但它已经标准化处理，这样同一个词的不同形式是一样的。例如，标准化通常包括：将大写字母折成小写字母、删除后缀（如英语中的 s 或者 es）。这将允许通过搜索找到同一个词的不同形式，不需要繁琐地输入所有可能的变形样式。同时，这一步通常会删除停用词。这些停用词通常因为太常见而对搜索无用。（总之，token 是文档文本的原片段，而词素被认为是有用的索引和搜索词。）GBase 8s 使用词典执行这一步，且提供了各种标准的词典。

- 保存搜索优化后的预处理文档。

比如，每个文档可以呈现为标准化词素的有序组合。伴随词素，通常还需要存储词素位置信息以用于邻近排序。因此文档包含的查询词越密集其排序越高。

词典能够对 token 如何标准化做到细粒度控制。使用合适的词典，可以定义不被索引的停用词。

数据类型 tsvector 用于存储预处理文档，tsquery 用于存储查询条件，详细请参见 16.3.10 文本搜索类型。为这些数据类型提供的函数和操作符请参见 16.5.12 文本检索函数和操作符。其中最重要的是匹配运算符@@，将在 16.8.1.3 基本文本匹配中介绍。

21.1.2 文档概念

文档是全文搜索系统的搜索单元，例如：杂志上的一篇文章或电子邮件消息。文本搜索引擎必须能够解析文档，而且可以存储父文档的关联词素（关键词）。后续，这些关联词素用来搜索包含查询词的文档。

在 GBase 8s 中，文档通常是一个数据库表中一行的文本字段，或者这些字段的可能组合（级联）。文档可能存储在多个表中或者需动态获取。换句话说，一个文档由被索引化的不同部分构成，因此无法存储为一个整体。比如：

```
postgres=# SELECT d_dow||'-'||d_dom||'-'||d_fy_week_seq AS identify_serials
FROM tpcds.date_dim WHERE d_fy_week_seq = 1;
identify_serials
----------------- 5-6-1
0-8-1
2-3-1
3-4-1
4-5-1
1-2-1
6-7-1
(7 rows)
```

注意：

实际上，在这些示例查询中，应该使用 coalesce 防止一个独立的 NULL 属性导致整个文档的 NULL 结果。支持 json、jsonb 格式。

另外一种可能是：文档在文件系统中作为简单的文本文件存储。在这种情况下，数据库可以用于存储全文索引并且执行搜索，同时可以使用一些唯一标识从文件系统中检索文档。然而，从数据库外部检索文件需要拥有系统管理员权限或者特殊函数支持。因此，还是将所有数据保存在数据库中比较方便。同时，将所有数据保存在数据库中可以方便地访问文档元

数据以便于索引和显示。

为了实现文本搜索目的，必须将每个文档减少至预处理后的 tsvector 格式。搜索和相关性排序都是在 tsvector 形式的文档上执行的。原始文档只有在被选中要呈现给用户时才会被检索。因此，我们常将 tsvector 说成文档，但是很显然其实它只是完整文档的一种紧凑表示。

21.1.3 基本文本匹配

GBase 8s 的全文检索基于匹配算子@@，当一个 tsvector(document)匹配到一个 tsquery(query)时，则返回 true。其中，tsvector(document)和 tsquery(query)两种数据类型可以任意排序。

```
postgres=# SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat
& rat'::tsquery AS RESULT; result
----------
t
(1 row)
postgres=# SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat
rat'::tsvector AS RESULT; result
----------
f
(1 row)
```

正如上面例子表明，tsquery 不仅是文本，且比 tsvector 包含的要多。tsquery 包含已经标注化为词条的搜索词，同时可能是使用 AND、OR、或 NOT 操作符连接的多个术语。详细请参见文本搜索类型。函数 to_tsquery 和 plainto_tsquery 对于将用户书写文本转换成适合的 tsquery 是非常有用的，比如将文本中的词标准化。类似地，to_tsvector 用于解析和标准化文档字符串。因此，实际中文本搜索匹配看起来更像这样：

```
postgres=# SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat')
AS RESULT; result
----------
t
(1 row)
```

需要注意的是，下面这种方式是不可行的：

```
postgres=# SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat')AS
RESULT; result
----------
f
(1 row)
```

由于 tsvector 没有对 rats 进行标准化，所以 rats 不匹配 rat。

@@操作符也支持 text 输入，允许一个文本字符串的显示转换为 tsvector 或者在简单情况下忽略 tsquery。可用形式是：

```
tsvector @@
tsquery tsquery
@@ tsvector text
@@ tsquery
text @@ text
```

我们已经看到了前面两种，形式 text @@ tsquery 等价于 to_tsvector(text) @@ tsquery，而 text @@ text 等价于 to_tsvector(text) @@ plainto_tsquery(text)。

21.1.4 分词器

全文检索功能还可以做更多事情：忽略索引某个词（停用词），处理同义词和使用复杂解析，例如：不仅基于空格的解析。这些功能通过文本搜索分词器控制。GBase 8s 支持多语言的预定义的分词器，并且可以创建分词器（gsql 的\dF 命令显示 了所有可用分词器）。

在安装期间选择一个合适的分词器，并且在 postgresql.conf 中相应的设置 default_text_search_config。如果为了 GBase 8s 使用同一个文本搜索分词器可以使 用 postgresql.conf 中的值。如果需要在 GBase 8s 中使用不同分词器，可以使用 ALTER DATABASE ... SET 在任一数据库进行配置。用户也可以在每个会话中设置 default_text_search_config。

每个依赖于分词器的文本搜索函数有一个可选的配置参数，用以明确声明所使用的分词器。仅当忽略这个参数的时候，才使用 default_text_search_config。

为了更方便的建立自定义文本搜索分词器，可以通过简单的数据库对象建立分词器。

GBase 8s 文本搜索功能提供了四种类型与分词器相关的数据库对象：

- 文本搜索解析器将文档分解为 token，并且分类每个 token（例如：词和数字）。
- 文本搜索词典将 token 转换成规范格式并且丢弃停用词。
- 文本搜索模板提供潜在的词典功能：一个词典指定一个模板，并且为模板设置参数。
- 文本搜索分词器选择一个解析器，并且使用一系列词典规范化语法分析器产生的 token。

21.2 表和索引

21.2.1 搜索表

在不使用索引的情况下也可以进行全文检索。

- 一个简单查询：将 body 字段中包含 america 的每一行打印出来。

```
postgres=# DROP SCHEMA IF EXISTS tsearch CASCADE;
postgres=# CREATE SCHEMA tsearch;
postgres=# CREATE TABLE tsearch.pgweb(id int, body text, title text,
last_mod_date date);
postgres=# INSERT INTO tsearch.pgweb VALUES(1, 'China, officially the People''s
Republic of China (PRC), located in Asia, is the world''s most populous state.',
'China', '2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(2, 'America is a rock band, formed
in England in 1970 by multi-instrumentalists Dewey Bunnell, Dan Peek, and Gerry
Beckley.', 'America', '2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(3, 'England is a country that is part
of the United Kingdom. It shares land borders with Scotland to the north and Wales
to the west.', 'England',
'2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(4, 'Australia, officially the
commonwealth of Australia, is a country comprising the mainland of the Australian
continent, the island of Tasmania, and numerous smaller islands.', 'Australia',
'2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(5, 'Russia, also officially known as
the Russian Federation, is a sovereign state in northern Eurasia.', 'Russia',
'2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(6, 'Japan is an island country in East
Asia.', 'Japan', '2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(7, 'Germany, officially the Federal
Republic of Germany, is a sovereign state and federal parliamentary republic in
central-western Europe.', 'Germany', '2010-1-1');
```

```
postgres=# INSERT INTO tsearch.pgweb VALUES(8, 'France, is a sovereign state
comprising territory in western Europe and several overseas regions and
territories.', 'France', '2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(9, 'Italy officially the Italian
Republic, is a unitary parliamentary republic in Europe.', 'Italy', '2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(10, 'India, officially the Republic
of India, is a country in South Asia.', 'India', '2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(11, 'Brazil, officially the
Federative Republic of Brazil, is the largest country in both South America and
Latin America.', 'Brazil', '2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(12, 'Canada is a country in the
northern half of North America.', 'Canada', '2010-1-1');

postgres=# INSERT INTO tsearch.pgweb VALUES(13, 'Mexico, officially the United
Mexican States, is a federal republic in the southern part of North America.',
'Mexico', '2010-1-1');

postgres=# SELECT id, body, title FROM tsearch.pgweb WHERE to_tsvector('english',
body) @@ to_tsquery('english', 'america');
id |    body     | title
----
+-------------------------------------------------------------------------------
---------------------------------------
----+---------
2 | America is a rock band, formed in England in 1970 by multi-instrumentalists
Dewey Bunnell, Dan Peek, and Gerry Beckley. | America
12| Canada is a country in the northern half of North America. | Canada
13| Mexico, officially the United Mexican States, is a federal republic in the
southern part of North America.      | Mexico
11 | Brazil, officially the Federative Republic of Brazil, is the largest country
in both South America and Latin America. | Brazil
(4 rows)
```

像 America 这样的相关词也会被找到，因为这些词都被处理成了相同标准的词条。

上面的查询指定 english 配置来解析和规范化字符串。当然也可以省略此配置，通过 default_text_search_config 进行配置设置：

```
postgres=# SHOW default_text_search_config; default_text_search_config
```

```
--------------------------
pg_catalog.english (1 row)

postgres=# SELECT id, body, title FROM tsearch.pgweb WHERE to_tsvector(body) @@
to_tsquery('america');
id |              body                          | title        |
+-------------------------------------------------------------------------------
-------------------------------------
----+---------
11 | Brazil, officially the Federative Republic of Brazil, is the largest country
in both South America and Latin America. | Brazil
2 | America is a rock band, formed in England in 1970 by multi-instrumentalists
Dewey Bunnell, Dan Peek, and Gerry Beckley. | America
12| Canada is a country in the northern half of North America. | Canada
13| Mexico, officially the United Mexican States, is a federal republic in the
southern part of North America.     | Mexico
(4 rows)
```

- 一个复杂查询：检索出在 title 或者 body 字段中包含 north 和 america 的最近 10 篇文档：

```
postgres=# SELECT title FROM tsearch.pgweb WHERE to_tsvector(title || ' ' || body)
@@ to_tsquery('north & america') ORDER BY last_mod_date DESC LIMIT 10;
title
--------
Mexico Canada (2 rows)
```

为了清晰，举例中没有调用 coalesce 函数在两个字段中查找包含 NULL 的行。

以上例子均在没有索引的情况下进行查询。对于大多数应用程序来说，这个方法很慢。因此除了偶尔的特定搜索，文本搜索在实际使用中通常需要创建索引。

21.2.2 创建索引

为了加速文本搜索，可以创建 GIN 索引。

```
postgres=# CREATE INDEX pgweb_idx_1 ON tsearch.pgweb USING
gin(to_tsvector('english',  body));
```

to_tsvector()函数有两个版本。只输一个参数的版本和输两个参数的版本。只输一个 参数时，系统默认采用 default_text_search_config 所指定的分词器。

请注意：创建索引时必须使用 to_tsvector 的两参数版本。只有指定了分词器名称的全文检索函数才可以在索引表达式中使用。这是因为索引的内容必须不受 default_text_search_config 的影响，否则索引内容可能不一致。由于 default_text_search_config

的值可以随时调整，从而导致不同条目生成的 tsvector 采用 了不同的分词器，并且没有办法区分究竟使用了哪个分词器。正确地转储和恢复这样的索引也是不可能的。

因为在上述创建索引中 to_tsvector 使用了两个参数，只有当查询时也使用了两个参数，且参数值与索引中相同时，才会使用该索引。也就是说，WHERE

to_tsvector('english', body) @@ 'a & b' 可以使用索引，但 WHERE to_tsvector(body) @@ 'a & b'不能使用索引。这确保只使用这样的索引——索引各条目是使用相同的分词器创建的。

索引中的分词器名称由另一列指定时可以建立更复杂的表达式索引。例如：

```
postgres=# CREATE INDEX pgweb_idx_2 ON tsearch.pgweb USING
gin(to_tsvector('ngram', body));
```

其中 body 是 pgweb 表中的一列。当对索引的各条目使用了哪个分词器进行记录时，允许在同一索引中存在混合分词器。在某些场景下这将是有用的。例如，文档集合中包含不同语言的文档时。再次强调，打算使用索引的查询必须措辞匹配，例如，WHERE to_tsvector(config_name, body) @@ 'a & b'与索引中的 to_tsvector 措辞匹配。

索引甚至可以连接列：

```
postgres=# CREATE INDEX pgweb_idx_3 ON tsearch.pgweb USING
gin(to_tsvector('english', title || ' ' || body));
```

另一个方法是创建一个单独的 tsvector 列控制 to_tsvector 的输出。下面的例子是 title 和 body 的连接， 当其它是 NULL 的时候，使用 coalesce 确保一个字段仍然会被索引：

```
postgres=# ALTER TABLE tsearch.pgweb ADD COLUMN textsearchable_index_col
tsvector;
postgres=# UPDATE tsearch.pgweb SET textsearchable_index_col =
to_tsvector('english', coalesce(title,'')
|| ' ' || coalesce(body,''));
```

然后为加速搜索创建一个 GIN 索引：

```
postgres=# CREATE INDEX textsearch_idx_4 ON tsearch.pgweb USING
gin(textsearchable_index_col);
```

现在，就可以执行一个快速全文搜索了：

```
postgres=# SELECT title FROM tsearch.pgweb
WHERE textsearchable_index_col @@ to_tsquery('north & america') ORDER BY
last_mod_date DESC
LIMIT 10;
```

```
title
--------
Canada Mexico (2 rows)
```

相比于一个表达式索引，单独列方法的一个优势是：它没有必要在查询时明确指定分词器以便能使用索引。正如上面例子所示，查询可以依赖于 default_text_search_config。另一个优势是搜索比较快速，因为它没有必要重新利用 to_tsvector 调用来验证索引匹配。表达式索引方法更容易建立，且它需要较少的磁盘 空间，因为 tsvector 形式没有明确存储。

21.2.3 索引使用约束

下面是一个使用索引的例子：

```
postgres=# create table table1 (c_int int,c_bigint bigint,c_varchar
varchar,c_text text) with(orientation=row);

postgres=# create text search configuration ts_conf_1(parser=POUND);
postgres=# create text search configuration ts_conf_2(parser=POUND)
with(split_flag='%');

postgres=# set default_text_search_config='ts_conf_1';
postgres=# create index idx1 on table1 using gin(to_tsvector(c_text));

postgres=# set default_text_search_config='ts_conf_2';
postgres=# create index idx2 on table1 using gin(to_tsvector(c_text));

postgres=# select c_varchar,to_tsvector(c_varchar) from table1 where
to_tsvector(c_text) @@ plainto_tsquery('￥#@……&**') and to_tsvector(c_text)
@@ plainto_tsquery('某公司 ') and c_varchar is not null order by 1 desc limit 3;
```

该例子的关键点是表 table1 的同一个列 c_text 上建立了两个 gin 索引：idx1 和 idx2，但这两个索引是在不同 default_text_search_config 的设置下建立的。该例子和同一张表 的同一个列上建立普通索引的不同之处在于：

gin 索引使用了不同的 parser（即分隔符不同），那么 idx1 和 idx2 的索引数据是不同的；

在同一张表的同一个列上建立的多个普通索引的索引数据是相同的。因此当执行同一个查询时，使用 idx1 和 idx2 查询出的结果是不同的。

使用约束

通过上面的例子，索引使用满足如下条件时：

- 在同一个表的同一个列上建立了多个 gin 索引；

- 这些 gin 索引使用了不同的 parser（即分隔符不同）；
- 在查询中使用了该列，且执行计划中使用索引进行扫描；
- 为了避免使用不同 gin 索引导致查询结果不同的问题，需要保证在物理表的一列上只有一个 gin 索引可用。

21.3 控制文本搜索

21.3.1 解析文档

GBase 8s 中提供了 to_tsvector 函数把文档处理成 tsvector 数据类型。

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

to_tsvector 将文本文档解析为 token，再将 token 简化到词素，并返回一个 tsvector。其 中 tsvector 中列出了词素及它们在文档中的位置。文档是根据指定的或默认的文本搜索分词器进行处理的。这里有一个简单的例子：

```
postgres=# SELECT to_tsvector('english', 'a fat cat sat on a mat - it ate a fat
rats'); to_tsvector
-----------------------------------------------
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

通过以上例子可发现结果 tsvector 不包含词 a、on 或者 it，rats 变成 rat，并且忽略标点符号-。

to_tsvector 函数内部调用一个解析器，将文档的文本分解成 token 并给每个 token 指定一个类型。对于每个 token，有一系列词典可供查询。词典系列因 token 类型的不同而 不同。识别 token 的第一本词典将发出一个或多个标准词素来表示 token。例如：

rats 变成 rat 因为词典认为词 rats 是 rat 的复数形式。

有些词被作为停用词（请参考 16.8.6.2 停用词），这样它们就会被忽略，因为它们出现得太过频繁以致于搜索中没有用处。比如例子中的 a、on 和 it。

如果没有词典识别 token，那么它也被忽略。在这个例子中，符号“-”被忽略， 因为词典没有给它分配 token 类型（空间符号），即空间记号永远不会被索引。

语法解析器、词典和要索引的 token 类型由选定的文本搜索分词器决定。可以在同一个数据库中有多种不同的分词器，以及提供各种语言的预定义分词器。在以上例子中， 使用缺省分词器 english。

函数 setweight 可以给 tsvector 的记录加权重，权重是字母 A、B、C、D 之一。这通常

用于标记来自文档不同部分的记录，比如标题、正文。之后，这些信息可以用于排序搜索结果。

因为 to_tsvector(NULL)会返回空，当字段可能是空的时候，建议使用 coalesce。以下是推荐的为结构化文档创建 tsvector 的方法：

```
postgres=# CREATE TABLE tsearch.tt (id int, title text, keyword text, abstract
text, body text, ti tsvector);

postgres=# INSERT INTO tsearch.tt(id, title, keyword, abstract, body) VALUES (1,
'China', 'Beijing', 'China','China, officially the People''s Republic of China
(PRC), located in Asia, is the world''s most populous state.');

postgres=# UPDATE tsearch.tt SET ti = setweight(to_tsvector(coalesce(title,'')),
'A') || setweight(to_tsvector(coalesce(keyword,'')), 'B') ||
setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
setweight(to_tsvector(coalesce(body,'')), 'D'); postgres=# DROP TABLE
tsearch.tt;
```

上例使用 setweight 标记已完成的 tsvector 中的每个词的来源，并且使用 tsvector 连接操作符||合并标记过的 tsvector 值，16.8.4.1 处理 tsvector 一节详细介绍了这些操作。

21.3.2 解析查询

GBase 8s 提供了函数 to_tsquery 和 plainto_tsquery 将查询转换为 tsquery 数据类型，to_tsquery 提供比 plainto_tsquery 更多的功能，但对其输入要求更严格。

to_tsquery([config regconfig,] querytext text) returns tsquery

to_tsquery 从 querytext 中创建一个 tsquery，querytext 必须由布尔运算符&(AND)，|（OR）和! （NOT）分割的单个 token 组成。这些运算符可以用圆括弧分组。换句话说，to_tsquery 输入必须遵循 tsquery 输入的通用规则，具体请参见 16.3.10 文本搜索类型。不同的是基本 tsquery 以 token 表面值作为输入，而 to_tsquery 使用指定或默认分词器将每个 token 标准化成词素，并依据分词器丢弃属于停用词的 token。例如：

```
postgres=# SELECT to_tsquery('english', 'The & Fat & Rats'); to_tsquery
---------------
'fat' & 'rat' (1 row)
```

像在基本 tsquery 中的输入一样，weight(s)可以附加到每个词素来限制它只匹配那些有相同 weight(s)的 tsvector 词素。比如：

```
postgres=# SELECT to_tsquery('english', 'Fat | Rats:AB'); to_tsquery
```

```
-----------------
'fat' | 'rat':AB (1 row)
```

同时，*也可以附加到词素来指定前缀匹配：

```
postgres=# SELECT to_tsquery('supern:*A & star:A*B'); to_tsquery
--------------------------
'supern':*A & 'star':*AB (1 row)
```

这样的词素将匹配 tsquery 中指定字符串和权重的项。

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

plainto_tsquery 将未格式化的文本 querytext 变换为 tsquery。类似于 to_tsvector，文本 被解析并且标准化，然后在存在的词之间插入&（AND）布尔算子。

比如：

```
postgres=# SELECT plainto_tsquery('english', 'The Fat Rats'); plainto_tsquery
-----------------
'fat' & 'rat' (1 row)
```

请注意，plainto_tsquery 无法识别布尔运算符、权重标签，或在其输入中的前缀匹配标签：

```
postgres=# SELECT plainto_tsquery('english', 'The Fat & Rats:C');
plainto_tsquery
---------------------
'fat' & 'rat' & 'c' (1 row)
```

在这里，所有输入的标点符号作为空格符号丢弃。

21.3.3 排序查询结果

排序试图针对特定查询衡量文档的相关度，从而将众多的匹配文档中相关度最高的文档排在最前。GBase 8s 提供了两个预置的排序函数。函数考虑了词法，距离，和结构信息；也就是，他们考虑查询词在文档中出现的频率、紧密程度、以及他们出现的地方在文档中的重要性。然而，相关性的概念是模糊的，并且是跟应用强相关的。不同的应用程序可能需要额外的信息来排序，比如，文档的修改时间，内置的排序函数等。也可以开发自己的排序函数或者采用附加因素组合这些排序函数的结果来满足特定需求。

两个预置的排序函数：

```
ts_rank([ weights float4[], ] vector tsvector, query tsquery [, normalization
integer ]) returns float4
```

基于词素匹配率对 vector 进行排序：

```
ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4
```

该函数需要位置信息的输入。因此它不能在“剥离”tsvector 值的情况下运行—它将总是返回零。

对于这两个函数，可选的 weights 参数提供给词加权重的能力，词的权重大小取决于所加的权值。权重阵列指定在排序时为每类词汇加多大的权重。

```
{D-weight, C-weight, B-weight, A-weight}
```

如果没有提供 weights，则使用缺省值：{0.1, 0.2, 0.4, 1.0}。

通常的权重是用来标记文档特殊领域的词，如标题或最初的摘要，所以相对于文章主体中的词它们有着更高或更低的重要性。

由于较长的文档有更多的机会包含查询词，因此有必要考虑文档的大小。例如，包含有 5 个搜索词的一百字文档比包含有 5 个搜索词的一千字文档相关性更高。两个预置的排序函数都采用了一个整型的标准化选项来定义文档长度是否影响排序及如何影响。这个整型选项控制多个行为，所以它是一个屏蔽字：可以使用|指定一个或多个行为（例如，2|4）。

- 0（缺省）表示：跟长度大小没有关系
- 1 表示：排名（rank）除以（文档长度的对数+1）
- 2 表示：排名除以文档的长度
- 4 表示：排名除以两个扩展词间的调和平均距离。只能使用 ts_rank_cd 实现
- 8 表示：排名除以文档中单独词的数量
- 16 表示：排名除以单独词数量的对数+1
- 32 表示：排名除以排名本身+1

当指定多个标志位时，会按照所列的顺序依次进行转换。

需要特别注意的是，排序函数不使用任何全局信息，所以不可能产生一个某些情况下需要的 1%或 100%的理想标准值。标准化选项 32 (rank/(rank+1))可用于所有规模的从零到一之间的排序，当然，这只是一个表面变化；它不会影响搜索结果的排序。

下面是一个例子，仅选择排名前十的匹配：

```
postgres=# SELECT id, title, ts_rank_cd(to_tsvector(body), query) AS rank FROM tsearch.pgweb, to_tsquery('america') query
```

```
WHERE query @@ to_tsvector(body) ORDER BY rank DESC
LIMIT 10;
id | title | rank
----+---------+------
11 | Brazil | .2
2 | America | .1
| Canada   |   .1
|  Mexico   |  .1
(4 rows)
```

这是使用标准化排序的相同例子：

```
postgres=# SELECT id, title, ts_rank_cd(to_tsvector(body), query, 32 /*
rank/(rank+1) */ ) AS rank FROM tsearch.pgweb, to_tsquery('america') query
WHERE query @@ to_tsvector(body) ORDER BY rank DESC
LIMIT 10;
id | title | rank
----+---------+----------
11 | Brazil  |  .166667
2 | America | .0909091
12 | Canada  | .0909091
13 | Mexico   | .0909091
(4 rows)
```

下面是使用中文分词法排序查询的例子：

```
postgres=# CREATE TABLE tsearch.ts_ngram(id int, body text); postgres=# INSERT
INTO tsearch.ts_ngram VALUES(1, ' 中 文 '); postgres=# INSERT INTO
tsearch.ts_ngram VALUES(2, '中文检索'); postgres=# INSERT INTO tsearch.ts_ngram
VALUES(3, '检索中文');
--精确匹配
postgres=# SELECT id, body, ts_rank_cd(to_tsvector('ngram',body), query) AS rank
FROM tsearch.ts_ngram, to_tsquery('中文') query WHERE query @@
to_tsvector(body);
id | body | rank
----+------+------
1 | 中文 | .1
(1 row)

--模糊匹配
postgres=# SELECT id, body, ts_rank_cd(to_tsvector('ngram',body), query) AS rank
FROM tsearch.ts_ngram, to_tsquery('中文') query WHERE query @@
to_tsvector('ngram',body);
id | body | rank
```

```
----+---------+------
3 | 检索中文 | .1
1 | 中文     | .1
2 | 中文检索 | .1
(3 rows)
```

排序要遍历每个匹配的 tsvector，因此资源消耗多，可能会因为 I/O 限制导致排序慢。可是这是很难避免的，因为实际查询中通常会有大量的匹配。

21.3.4 高亮搜索结果

搜索结果的理想显示是：列出每篇文档中与搜索相关的部分，并标识为什么与查询相关。搜索引擎能够显示标识了搜索词的文档片段。GBase 8s 提供了函数 ts_headline 支持这部分功能。

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ])
returns text
```

ts_headline 的输入是带有查询条件的文档，其返回文档中的摘录，在摘录中查询词是高亮显示的。用来解析文档的分词器由 config 参数指定。如果省略 config，则使用 default_text_search_config 的值所指定的分词器。

指定 options 字符串时，需由一个或多个 option=value 对组成，且必须用逗号分隔。

options 可以是下面的选项：

- StartSel，StopSel：分隔文档中出现的查询词，以区别于其他摘录词。当包含有空格或逗号时，必须用双引号将字符串引起来。
- MaxWords，MinWords：定义摘录的最长和最短值。
- ShortWord：在摘录的开始和结束会丢弃此长度或更短的词。默认值 3 会消除常见的英语冠词。
- HighlightAll：布尔标志。如果为真，整个文档将作为摘录。忽略前面三个参数的值。
- MaxFragments：要显示的文本摘录或片段的最大数量。默认值 0 表示选择非片段的摘录生成方法。大于 0 的值表示选择基于片段的摘录生成。此方法查找带有尽可能多查询词的文本片段，并显示查询词周围的上下文片段。因此，查询词临近每个片段的中间，且查询词两边都有词。每个片段至多有 MaxWords，并且长度 为 ShortWord 或更短的词在

每一个片段开始和结束被丢弃。如果在文档中没有找 到所有的查询词，则文档中开头将显示 MinWords 单片段。

- FragmentDelimiter：当有一个以上的片段时，通过该字符串分隔这些片段。不声明选项时，采用下面的缺省值：

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE, MaxFragments=0,
FragmentDelimiter=" ... "
```

例如：

```
postgres=# SELECT ts_headline('english', 'The most common type of search
is to find all documents containing given query terms and return them in order
of their similarity to the query.',
to_tsquery('english', 'query & similarity')); ts_headline
------------------------------------------------------------

containing given <b>query</b> terms
and return them in order of their <b>similarity</b> to the
<b>query</b>. (1 row)

postgres=# SELECT ts_headline('english', 'The most common type of search
is to find all documents containing given query terms and return them in order
of their similarity to the query.',
to_tsquery('english', 'query & similarity'), 'StartSel = <, StopSel = >');
ts_headline
------------------------------------------------

containing given <query> terms
and return them in order of their <similarity> to the
<query>. (1 row)
```

ts_headline 使用原始文档，而不是 tsvector 摘录，因此使用起来会慢，应慎重使用。

21.4 附加功能

21.4.1 处理 tsvector

GBase 8s 提供了用来操作 tsvector 类型的函数和操作符。

tsvector || tsvector

tsvector 连接操作符返回一个新的 tsvector 类型，它综合了两个 tsvector 中词素和位置信息，并保留词素的位置信息和权重标签。右侧的 tsvector 的起始位置位于左侧 tsvector 的最

后位置，因此，新生成的 tsvector 几乎等同于将两个原始文档字串连接后进行 to_tsvector 操作。（这个等价是不准确的，因为任何从左边 tsvector 中删除的停用词都不会影响结果，但是，在使用文本连接时，则会影响词素在右侧 tsvector 中的位置。）

相较于对文本进行连接后再执行 to_tsvector 操作，使用 tsvector 类型进行连接操作 的优势在于，可以对文档的不同部分使用不同配置进行解析。因为 setweight 函数 会对给定的 tsvector 中的语素进行统一设置，如果想要对文档的不同部分设置不同的权重，需要在连接之前对文本进行解析和权重设置。

setweight(vector tsvector, weight "char") returns tsvector

setweight 返回一个输入 tsvector 的副本，其中每一个位置都使用给定的权重做了标记。权值可以为 A、B、C 或 D（D 是 tsvector 副本的默认权重，并且不在输出中 呈现）。当对 tsvector 进行连接操作时，这些权重标签将会被保留，文档不同部分以不同的权重进行排序。

注意

权重标签作用于位置，而不是词素。如果传入的 tsvector 已经被剥离了位置信息， 那么 setweight 函数将什么都不做。

- length(vector tsvector) returns integer

  返回 vector 中的词素的数量。

- strip(vector tsvector) returns tsvector

返回一个 tsvector 类型，其中包含输入的 tsvector 的同义词，但不包含任何位置和 权重信息。虽然在相关性排序中，这里返回的 tsvector 要比未拆分的 tsvector 的作 用小很多，但它通常都比未拆分的 tsvector 小的多。

21.4.2 处理查询

GBase 8s 提供了函数和操作符用来操作 tsquery 类型的查询。

- tsquery && tsquery

  返回两个给定查询 tsquery 的与结果。

- tsquery || tsquery

  返回两个给定查询 tsquery 的或结果。

- !! tsquery

返回给定查询 tsquery 的非结果。

- numnode(query tsquery) returns integer

返回 tsquery 中的节点数目（词素加操作符），这个函数在检查查询是否有效（返回值大于 0），或者只包含停用词（返回值等于 0）时，是有用的。例如：

```
postgres=# SELECT numnode(plainto_tsquery('the any'));
NOTICE: text-search query contains only stop words or doesn't contain lexemes,
ignored CONTEXT: referenced column: numnode
numnode
---------
0

postgres=# SELECT numnode('foo & bar'::tsquery); numnode
---------
3
```

- querytree(query tsquery) returns text

返回可用于索引搜索的 tsquery 部分，该函数对于检测非索引查询是有用的（例如只包含停用词或否定项）。例如:

```
postgres=# SELECT querytree(to_tsquery('!defined')); querytree
-----------
 T
(1 row)
```

21.4.2.1 查询重写

ts_rewrite 函数族可以从 tsquery 中搜索一个特定的目标子查询，并在该子查询每次出现的地方都替换为另一个子查询。 实际上这只是通过字串替换而得到的一个特定 tsquery 版本。目标子查询和替换查询组合起来可以被认为是一个重写规则。一组类似的重写规则可以为搜索提供强大的帮助。例如，可以使用同义词扩大搜索范围（例如，new york, big apple, nyc, gotham）或限制搜索范围在用户直接感兴趣的热点话题上。

- ts_rewrite (query tsquery, target tsquery, substitute tsquery) returns tsquery

ts_rewrite 的这种形式只适用于一个单一的重写规则：任何出现目标子查询的地方都被无条件替换。例如：

```
postgres=# SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
ts_rewrite
------------
```

```
'b' & 'c'
```

- ts_rewrite (query tsquery, select text) returns tsquery

ts_rewrite 的这种形式接受一个起始查询和 SQL 查询命令。 这里的查询命令是文本字串形式，必须产生两个 tsquery 列。查询结果的每一行，第一个字段的值（目标子查询） 都会被第二个字段（替代子查询）替换。

说明

当多个规则需要重写时，重写顺序非常重要； 因此在实践中需要使用 ORDER BY 将源查询按照某些字段进行排序。

例如：举一个现实生活中天文学上的例子。我们将使用表驱动的重写规则扩大 supernovae 的查询范围：

```
postgres=# CREATE TABLE tsearch.aliases (id int, t tsquery, s tsquery);

postgres=# INSERT INTO tsearch.aliases VALUES(1, to_tsquery('supernovae'),
to_tsquery('supernovae|sn'));
postgres=# SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM
tsearch.aliases'); ts_rewrite
--------------------------------
'crab' & ( 'supernova' | 'sn' )
```

可以通过更新表修改重写规则：

```
postgres=# UPDATE tsearch.aliases
SET s = to_tsquery('supernovae|sn & !nebulae') WHERE t =
to_tsquery('supernovae');
postgres=# SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM
tsearch.aliases'); ts_rewrite
-------------------------------------------
'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

需要重写的规则越多，重写操作就越慢。因为它要检查每一个可能匹配的规则。为了过滤明显的非候选规则，可以使用 tsquery 类型的操作符来实现。在下面的例子中， 我们只选择那些可能与原始查询匹配的规则：

```
postgres=# SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM tsearch.aliases
WHERE ''a & b''::tsquery @> t');

ts_rewrite
------------
'b' & 'a' (1 row)
```

```
postgres=# DROP TABLE tsearch.aliases;
```

21.4.2.2 收集文献统计

函数 ts_stat 可用于检查配置和查找候选停用词。

```
ts_stat(sqlquery text, [ weights text, ] OUT word text, OUT ndoc integer,
OUT nentry integer) returns setof record
```

sqlquery 是一个包含 SQL 查询语句的文本，该 SQL 查询将返回一个 tsvector。ts_stat 执行 SQL 查询语句并返回一个包含 tsvector 中每一个不同的语素（词）的统计信息。返回信息包括：

- word text：词素。
- ndoc integer：词素在文档（tsvector）中的编号。
- nentry integer：词素出现的频率。

如果设置了权重条件，只有标记了对应权重的词素才会统计频率。例如，在一个文档集中检索使用频率最高的十个单词：

```
postgres=# SELECT * FROM ts_stat('SELECT to_tsvector(''english'', sr_reason_sk)
FROM tpcds.store_returns WHERE sr_customer_sk < 10') ORDER BY nentry DESC, ndoc
DESC, word LIMIT 10;
word | ndoc | nentry
------+------+--------
32     |   2 | 2
33     |   2 | 2
1          1 | 1
10     |   1 | 1
13     |   1 | 1
14     |   1 | 1
15     |   1 | 1
17     |   1 | 1
20     |   1 | 1
22
(10        | 1 |
ows)   1
```

同样的情况，但是只计算权重为 A 或者 B 的单词使用频率：

```
postgres=# SELECT * FROM ts_stat('SELECT to_tsvector(''english'', sr_reason_sk)
FROM tpcds.store_returns WHERE sr_customer_sk < 10', 'a') ORDER BY nentry DESC,
ndoc DESC, word LIMIT 10; word | ndoc | nentry
```

```
------+------+--------
(0 rows)
```

21.4.3 解析器

文本搜索解析器负责将原文档文本分解为多个 token，并标识每个 token 的类型。这里 的类型集由解析器本身定义。注意，解析器并不修改文本，它只是确定合理的单词边界。由于这一限制，人们更需要定制词典，而不是为每个应用程序定制解析器。

目前 GBase 8s 提供了三个内置的解析器，分别为 pg_catalog.default/ pg_catalog.ngram/pg_catalog.pound，其中 pg_catalog.default 适用于英文分词场景，pg_catalog.ngram/pg_catalog.pound 是为了支持中文全文检索功能新增的两种解 析器，适用于中文及中英混合分词场景。

内置解析器 pg_catalog.default，它能识别 23 种 token 类型，显示在下表中。

表 21-1 默认解析器类型

别名	描述	示例
asciiword	Word, all ASCII letters	elephant
word	Word, all letters	mañana
numword	Word, letters and digits	beta1
asciihword	Hyphenated word, all ASCII	up-to-date
hword	Hyphenated word, all letters	lógico-matemática
numhword	Hyphenated word, letters and digits	openGauss-beta1
hword_asciipart	Hyphenated word part, all ASCII	openGauss in the context openGauss-beta1
hword_part	Hyphenated word part, all letters	lógico or matemática in the context lógico-matemática

hword_numpart	Hyphenated word part, letters and digits	beta1 in the context openGauss-beta1
email	Email address	foo@example.com
protocol	Protocol head	http://
url	URL	example.com/stuff/index.html
host	Host	example.com
url_path	URL path	/stuff/index.html, in the context of a URL
file	File or path name	/usr/local/foo.txt, if not within a URL
sfloat	Scientific notation	-1.23E+56
float	Decimal notation	-1.234
int	Signed integer	-1234
uint	Unsigned integer	1234
version	Version number	8.3.0
tag	XML tag	<a href="dictionaries.html">
entity	XML entity	&
blank	Space symbols	(any whitespace or punctuation not otherwise recognized)

注意：对于解析器来说，一个“字母”的概念是由数据库的语言区域设置，即 lc_ctype 设置决定的。只包含基本 ASCII 字母的词被报告为一个单独的 token 类型，因为这类词 有时需要被区分出来。大多数欧洲语言中，对 token 类型 word 和 asciiword 的处理方法是类似的。

email 不支持某些由 RFC 5322 定义的有效电子邮件字符。具体来说，可用于 email 用户

名的非字母数字字符仅包含句号、破折号和下划线。

解析器可能对同一内容进行重叠 token。例如，包含连字符的单词将作为一个整体进行报告，其组件也会分别被报告：

```
postgres=# SELECT alias, description, token FROM
ts_debug('english','foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+---------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

这种行为是有必要的，因为它支持搜索整个复合词和各组件。这里是另一个例子：

```
postgres=# SELECT alias, description, token FROM
ts_debug('english','http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

N-gram 是一种机械分词方法，适用于无语义中文分词场景。N-gram 分词法可以保证分词的完备性，但是为了照顾所有可能，把很多不必要的词也加入到索引中，导致索引项增加。N-gram 支持中文编码包括 GBK、UTF-8。内置 6 种 token 类型，如下表所示。

表 21-2 token 类型

别名	描述
zh_words	chinese words
en_word	english word
numeric	numeric data

alnum	alnum string
grapsymbol	graphic symbol
multisymbol	multiple symbol

Pound 是一种固定格式分词方法，适用于无语意但待解析文本以固定分隔符分割开来的中英文分词场景。支持中文编码包括 GBK、UTF8，支持英文编码包括 ASCII。内置 6 种 token 类型，如表 8-3 所示；支持 5 种分隔符，如表 8-4 所示，在用户不进行自定义设置的情况下分隔符默认为“#”。Pound 限制单个 token 长度不能超过 256 个字符。

表 21-3 token 类型

别名	描述
zh_words	chinese words
en_word	english word
numeric	numeric data
alnum	alnum string
grapsymbol	graphic symbol
multisymbol	multiple symbol

表 21-4 分隔符类型

分隔符	描述
@	Special character
#	Special character
$	Special character
%	Special character

/	Special character

21.4.4 词典

21.4.4.1 词典概述

词典用于定义停用词（stop words），即全文检索时不搜索哪些词。

词典还可以用于对同一词的不同形式进行规范化，这样同一个词的不同派生形式都可以进行匹配。规范化后的词称为词位（lexeme）。

除了提高检索质量外，词的规范化和删除停用词可以减少文档 tsvector 格式的大小， 从而提高性能。词的规范化和删除停用词并不总是具有语言学意义，用户可以根据应用环境在词典定义文件中自定义规范化和删除规则。

一个词典是一个程序，接收标记（token）作为输入，并返回：

- 如果 token 在词典中已知，返回对应 lexeme 数组（注意，一个标记可能对应多个 lexeme）。
- 一个 lexeme。一个新 token 会代替输入 token 被传递给后继词典（当前词典可被称为过滤词典）。
- 如果 token 在词典中已知，但它是一个停用词，返回空数组。
- 如果词典不能识别输入的 token，返回 NULL。

GBase 8s 提供了多种语言的预定义字典，同时提供了五种预定义的词典模板，分别是 Simple，Synonym，Thesaurus，Ispell，和 Snowball，可用于创建自定义参数的新词典。

在使用全文检索时，建议用户：

- 可以在文本搜索配置中定义一个解析器，以及一组用于处理该解析器的输出标记词典。对于解析器返回的每个标记类型，可以在配置中指定不同的词典列表进行处理。当解析器输出一种类型的标记后，在对应列表的每个字典中会查阅该标 记，直到某个词典识别它。如果它被识别为一个停用词， 或者没有任何词典识别，该 token 将被丢弃，即不被索引或检索到。通常情况下，第一个返回非空结果的词典决定了最终结果，后继词典将不会继续处理。但是一个过滤类型的词典可以依据规则替换输入 token，然后将替换后的 token 传递给后继词典进行处理。

- 配置字典列表的一般规则是，第一个位置放置一个应用范围最小的、最具体化定义的词典，其次是更一般化定义的词典， 最后是一个普适定义的词典，比如 Snowball 词干词典或 Simple 词典。在下面例子中，对于一个针对天文学的文本搜 索配置 astro_en，可以定义标记类型 asciiword（ASCII 词）对应的词典列表为：天文术语的 Synonym 同义词词典，Ispell 英语词典和 Snowball 英语词干词典。

```
postgres=# ALTER TEXT SEARCH CONFIGURATION astro_en
ADD MAPPING FOR asciiword WITH astro_syn, english_ispell, english_stem;
```

过滤类型的词典可以放置在词典列表中除去末尾的任何地方，放置在末尾时是无效的。使用这些词典对标记进行部分规范化，可以有效简化后继词典的处理。

21.4.4.2 停用词

停用词是很常见的词，几乎出现在每一个文档中，并且没有区分值。因此，在全文搜索的语境下可忽视它们。停用词处理逻辑和词典类型相关。例如，Ispell 词典会先对标记进行规范化，然后再查看停用词表，而 Snowball 词典会最先检查输入标记是否为停 用词。

例如，每个英文文本包含像 a 和 the 的单词，因此没必要将它们存储在索引中。然而，停用词影响 tsvector 中的位置，同时位置也会影响相关度：

```
postgres=# SELECT to_tsvector('english','in the list of stop words'); to_tsvector
----------------------------
'list':3 'stop':5 'word':6
```

位置 1、2、4 是停用词，所以不显示。为包含和不包含停用词的文档计算出的排序是完全不同的：

```
postgres=# SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'),
to_tsquery('list & stop')); ts_rank_cd
------------
.05

postgres=# SELECT ts_rank_cd (to_tsvector('english','list stop words'),
to_tsquery('list & stop')); ts_rank_cd
------------
.1
```

21.4.4.3 Simple 词典

Simple 词典首先将输入标记转换为小写字母，然后检查停用词表。如果识别为停用词

则返回空数组，即表示该标记会被丢弃。否则，输入标记的小写形式作为规范化后的 lexeme 返回。此外，Simple 词典可通过设置参数 Accept 为 false（默认值 true），将非 停用词报告为未识别，传递给后继词典继续处理。

注意事项

大多数词典的功能依赖于词典定义文件，词典定义文件名仅支持小写字母、数字、下划线组合。

临时模式 pg_temp 下不允许创建词典。

词典定义文件的字符集编码必须为 UTF-8 格式。实际应用时，如果与数据库的字符编码格式不一致，在读入词典定义文件时会进行编码转换。

通常情况下，每个 session 仅读取词典定义文件一次，当且仅当在第一次使用该词典时。需要修改词典文件时，可通过 ALTER TEXT SEARCH DICTIONARY 命令进行词典定义文件的更新和重新加载。

操作步骤

步骤 1 创建 Simple 词典。

其中，停用词表文件全名为 english.stop。关于创建 simple 词典的语法和更多参数，请参见 CREATE TEXT SEARCH DICTIONARY。

```
postgres=# CREATE TEXT SEARCH DICTIONARY public.simple_dict ( TEMPLATE =
pg_catalog.simple,
STOPWORDS = english);
```

步骤 2 使用 Simple 词典。

```
postgres=# SELECT ts_lexize('public.simple_dict','YeS'); ts_lexize
-----------
{yes}
(1 row)

postgres=# SELECT ts_lexize('public.simple_dict','The'); ts_lexize
-----------
{}
(1 row)
```

步骤 3 设置参数 ACCEPT=false，使 Simple 词典返回 NULL，而不是返回非停用词的小写形式。

```
postgres=# ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );
ALTER TEXT SEARCH DICTIONARY
postgres=# SELECT ts_lexize('public.simple_dict','YeS'); ts_lexize
-----------


(1 row)

postgres=# SELECT ts_lexize('public.simple_dict','The'); ts_lexize
-----------
{}
(1 row)
```

21.4.4.4 Synonym 词典

Synonym 词典用于定义、识别 token 的同义词并转化，不支持词组（词组形式的同义词可用 Thesaurus 词典定义，详细请参见 Thesaurus 词典）。

示例

Synonym 词典可用于解决语言学相关问题，例如，为避免使单词"Paris"变成"pari"，可在 Synonym 词典文件中定义一行"Paris　paris"，并将该词典放置在预定义的 english_stem 词典之前。

```
postgres=# SELECT * FROM ts_debug('english', 'Paris');
alias | description | token | dictionaries | dictionary | lexemes
-----------+-----------------+-------+---------------+-------------+-------
--
asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari} (1
row)

postgres=# CREATE TEXT SEARCH DICTIONARY my_synonym ( TEMPLATE = synonym,
SYNONYMS = my_synonyms, FILEPATH = 'file:///home/dicts/'
);

postgres=# ALTER TEXT SEARCH CONFIGURATION english ALTER MAPPING FOR asciiword
WITH my_synonym, english_stem;

postgres=# SELECT * FROM ts_debug('english', 'Paris');
alias     |    description    | token |  dictionaries    | dictionary | lexemes
-----------+-----------------+-------+--------------------------+-----------
-+---------
```

```
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym |
{paris} (1 row)

postgres=# SELECT * FROM ts_debug('english', 'paris');
alias     |    description    | token |  dictionaries   | dictionary | lexemes
----------+-------------------+-------+-----------------------+----------
-+---------
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym |
{paris} (1 row)
postgres=# ALTER TEXT SEARCH DICTIONARY my_synonym ( CASESENSITIVE=true);
postgres=# SELECT * FROM ts_debug('english', 'Paris');
alias     |    description    | token |  dictionaries   | dictionary | lexemes
----------+-------------------+-------+-----------------------+----------
-+---------
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym |
{paris} (1 row)

postgres=# SELECT * FROM ts_debug('english', 'paris');
alias     |    description    | token |  dictionaries   | dictionary | lexemes
----------+-------------------+-------+-----------------------+----------
-+---------
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym |
{pari} (1 row)
```

其中，同义词词典文件全名为 my_synonyms.syn，所在目录为当前连接数据库主 节点的/home/dicts/下。关于创建词典的语法和更多参数，请参见 16.14.37 ALTER TEXT SEARCH DICTIONARY。

星号（*）可用于词典文件中的同义词结尾，表示该同义词是一个前缀。在 to_tsvector()中该星号将被忽略，但在 to_tsquery()中会匹配该前缀并对应输出结果（参照 16.8.4.2 处理查询一节）。

假设词典文件 synonym_sample.syn 内容如下：

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

创建并使用词典：

```
postgres=# CREATE TEXT SEARCH DICTIONARY syn (
```

```
    TEMPLATE = synonym,
    SYNONYMS = synonym_sample
);

postgres=# SELECT ts_lexize('syn','indices');
 ts_lexize
-----------
 {index}
(1 row)

postgres=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);

postgres=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH
syn;

postgres=# SELECT to_tsvector('tst','indices');
 to_tsvector
-------------
 'index':1
(1 row)

postgres=# SELECT to_tsquery('tst','indices');
 to_tsquery
------------
 'index':*
(1 row)

postgres=# SELECT 'indexes are very useful'::tsvector;
            tsvector
--------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

postgres=# SELECT 'indexes are very useful'::tsvector @@
to_tsquery('tst','indices');
 ?column?
----------
 t
(1 row)
```

21.4.4.5 Thesaurus 词典

Thesaurus 词典，也叫做分类词典（缩写为 TZ），是一组定义了词以及词组间关系的集合，包括广义词（BT）、狭义词（NT）、首选词、非首选词、相关词等。根据词典文件中的定义，TZ 词典用一个指定的短语替换对应匹配的所有短语，并且可选择保留原始短语进行索引。TZ 词典实际上是 Synonym 词典的一个扩展，增加了短语支持。

注意事项

由于 TZ 词典需要识别短语，所以在处理过程中必须保存当前状态并与解析器进行交互，以决定是否处理下一个 token 或是结束当前识别。此外，TZ 词典配置时需谨慎，如果设置 TZ 词典仅处理 asciiword 类型的 token，则类似 one 7 的分类词典定义将不会生效，因为 uint 类型的 token 不会传给 TZ 词典处理。

在索引期间要用到分类词典，因此分类词典参数中的任何变化都要求重新索引。对于其他大多数类型的词典来说，类似添加或删除停用词这种修改并不需要强制重新索引。

操作步骤

步骤 1 创建一个名为 thesaurus_astro 的 TZ 词典。

以一个简单的天文学词典 thesaurus_astro 为例，其中定义了两组天文短语及其同义词 如下：

```
supernovae stars : sn
crab nebulae : crab
```

执行如下语句创建 TZ 词典：

```
postgres=# CREATE TEXT SEARCH DICTIONARY thesaurus_astro ( TEMPLATE = thesaurus,
DictFile = thesaurus_astro,
Dictionary = pg_catalog.english_stem, FILEPATH = 'file:///home/dicts/'
);
```

其中，词典定义文件全名为 thesaurus_astro.ths，所在目录为当前连接数据库主节点的 /home/dicts/下 。子词典 pg_catalog.english_stem 是预定义的 Snowball 类型的英语词干词典，用于规范化输入词，子词典自身相关配置（例如停用词等）不在此处显示。关于创建词典的语法和更多参数，请参见 16.14.92 CREATE TEXT SEARCH DICTIONARY。

步骤 2 创建词典后，将其绑定到对应文本搜索配置中需要处理的 token 类型上：

```
postgres=# ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

步骤 3 使用 TZ 词典。

测试 TZ 词典。

ts_lexize 函数对于测试 TZ 词典作用不大，因为该函数是按照单个 token 处理输入。 可以使用 plainto_tsquery、to_tsvector、to_tsquery 函数测试 TZ 词典，这些函数 能够将输入分解成多个 token（to_tsquery 函数需要将输入加上引号）。

```
postgres=# SELECT plainto_tsquery('russian','supernova star'); plainto_tsquery
-----------------
'sn'
(1 row)

postgres=# SELECT to_tsvector('russian','supernova star'); to_tsvector
-------------
'sn':1 (1 row)

postgres=# SELECT to_tsquery('russian',''supernova star''); to_tsquery
------------
'sn'
(1 row)
```

其中，supernova star 匹配了词典 thesaurus_astro 定义中的 supernovae stars，这是因为在 thesaurus_astro 词典定义中指定了 Snowball 类型的子词典 english_stem，该词典移除了 e 和 s。

如果同时需要索引原始短语，只要将其同时放置在词典定义文件中对应定义的右侧即可，如下：

```
supernovae stars : sn supernovae stars

postgres=# ALTER TEXT SEARCH DICTIONARY thesaurus_astro ( DictFile =
thesaurus_astro,
FILEPATH = 'file:///home/dicts/');

postgres=# SELECT plainto_tsquery('russian','supernova star'); plainto_tsquery
----------------------------
'sn' & 'supernova' & 'star' (1 row)
```

21.4.4.6 Ispell 词典

Ispell 词典模板支持词法词典，它可以把一个词的各种语言学形式规范化成相同的词位。比如，一个 Ispell 英语词典可以匹配搜索词 bank 的词尾变化和词形变化，如 banking、banked、banks、banks'和 bank's 等。

GBase 8s 不提供任何预定义的 Ispell 类型词典或词典文件。dict 文件和 affix 文件支持多种开源词典格式，包括 Ispell、MySpell 和 Hunspell 等。

操作步骤

步骤 1 获取词典定义文件和词缀文件。

用户可以使用开源词典，直接获取的开源词典后缀名可能为.aff 和.dic，此时需要将扩展名改为.affix 和.dict。此外，对于某些词典文件，还需要使用下面的命令把字符转换成 UTF-8 编码，比如挪威语词典：

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

步骤 2 创建 Ispell 词典。

```
postgres=# CREATE TEXT SEARCH DICTIONARY norwegian_ispell (
    TEMPLATE = ispell,
    DictFile = nn_no,
    AffFile = nn_no,
    FilePath = 'file:///home/dicts'
);
```

其中，词典文件全名为 nn_no.dict 和 nn_no.affix，所在目录为当前连接数据库主节点的/home/dicts/下 。关于创建词典的语法和更多参数，请参见 CREATE TEXT SEARCH DICTIONARY。

步骤 3 使用 Ispell 词典进行复合词拆分。

```
postgres=# SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
      ts_lexize
--------------------
 {sjokolade,fabrikk}
(1 row)
```

MySpell 不支持复合词，Hunspell 对复合词有较好的支持。GBase 8s 仅支持 Hunspell 中基本的复合词操作。通常情况下，Ispell 词典能够识别的词是一个有限集合，其后应该配置一个更广义的词典，例如一个可以识别所有词的 Snowball 词典。

21.4.4.7 Snowball 词典

Snowball 词典模板支持词干分析词典，基于 Martin Porter 的 Snowball 项目，内置有许多语言的词干分析算法。GBase 8s 中预定义有多种语言的 Snowball 词典，可通过《GBase 8s

V8.8.5 5.0.0_数据库参考手册》中系统表 PG_TS_DICT 查看预定义的词干分析词典以及支持的语言词干分析算法。

无论是否可以简化，Snowball 词典将标示所有输入为已识别，因此它应当被放置在词 典列表的最后。把 Snowball 词典放在任何其他词典前面会导致后继词典失效，因为输入 token 不会通过 Snowball 词典进入到下一个词典。

关于 Snowball 词典的语法，请参见 CREATE TEXT SEARCH DICTIONARY。

21.4.5 配置示例

文本搜索配置（Text Search Configuration），指定了将文档转换成 tsvector 过程中所必需的组件：

- 解析器，用于把文本分解成标记 token；
- 词典列表，用于将每个 token 转换成词位 lexeme。

每次 to_tsvector 或 to_tsquery 函数调用时，都需要指定一个文本搜索配置来指定具体的处理过程。GUC 参数 default_text_search_config 指定了默认的文本搜索配置，当文本搜索函数中没有显式指定文本搜索配置参数时，将会使用该默认值进行处理。

GBase 8s 中预定义有一些可用的文本搜索配置，用户也可创建自定义的文本搜索配置。此外，为了便于管理文本搜索对象，还提供有多个 gsql 元命令，可以显示有关文本搜索对象的信息。详细请参见《GBase 8s V8.8.5 5.0.0_工具参考手册》中“客户端工具 >元命令参考”章节。

操作步骤

(1) 创建一个文本搜索配置 ts_conf，复制预定义的文本搜索配置 english。

```
postgres=# CREATE TEXT SEARCH CONFIGURATION ts_conf ( COPY =
pg_catalog.english );
CREATE TEXT SEARCH CONFIGURATION
```

(2) 创建 Synonym 词典。

假设同义词词典定义文件 pg_dict.syn 内容如下：

```
postgres    pg
pgsql       pg
postgresql  pg
```

执行如下语句创建 Synonym 词典：

```
postgres=# CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict,
    FILEPATH = 'file:///home/dicts'
);
```

(3) 创建一个 Ispell 词典 english_ispell（词典定义文件来自开源词典）。

```
postgres=# CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english,
    FILEPATH = 'file:///home/dicts'
);
```

(4) 设置文本搜索配置 ts_conf，修改某些类型的 token 对应的词典列表。关于 token 类型的详细信息，请参见解析器。

```
postgres=# ALTER TEXT SEARCH CONFIGURATION ts_conf
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

(5) 在文本搜索配置中，选择设置不索引或搜索某些 token 类型。

```
postgres=# ALTER TEXT SEARCH CONFIGURATION ts_conf
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

(6) 使用文本检索调测函数 ts_debug()对所创建的词典配置 ts_conf 进行测试。

```
postgres=# SELECT * FROM ts_debug('ts_conf', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

(7) 可以设置当前 session 使用 ts_conf 作为默认的文本搜索配置。此设置仅在当前 session 有效。

```
postgres=# \dF+ ts_conf
      Text search configuration "public.ts_conf"
Parser: "pg_catalog.default"
      Token      |            Dictionaries
-----------------+------------------------------------
 asciihword      | pg_dict,english_ispell,english_stem
```

```
 asciiword        | pg_dict,english_ispell,english_stem
 file             | simple
 host             | simple
 hword            | pg_dict,english_ispell,english_stem
 hword_asciipart  | pg_dict,english_ispell,english_stem
 hword_numpart    | simple
 hword_part       | pg_dict,english_ispell,english_stem
 int              | simple
 numhword         | simple
 numword          | simple
 uint             | simple
 version          | simple
 word             | pg_dict,english_ispell,english_stem

postgres=# SET default_text_search_config = 'public.ts_conf';
SET
postgres=# SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.ts_conf
(1 row)
```

21.4.6 测试和调试文本搜索

21.4.6.1 分词器测试

函数 ts_debug 允许简单测试文本搜索分词器。

```
ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
         returns setof record
```

ts_debug 显示 document 的每个 token 信息，token 是由解析器生成，由指定的词典进行 处理。如果忽略对应参数，则使用 config 指定的分词器或者 default_text_search_config 指定的分词器。

ts_debug 为文本解析器标识的每个 token 返回一行记录。记录中的列分别是：

- alias：text 类型，token 的别名。
- description：text 类型，token 的描述。
- token：text 类型，token 的文本内容。
- dictionaries：regdictionary 数组类型，是分词器为 token 选定的词典。
- dictionary：regdictionary 类型，用来识别 token 的词典。如果为空，则不做识 别。
- lexemes：text 数组类型，词典识别 token 时生成的词素。如果为空，则不生成词素。空数组（{}）意味着 token 将被识别成停用词。

一个简单的例子：

```
postgres=# SELECT * FROM ts_debug('english','a fat  cat sat on a mat - it ate a
fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+-------
--
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

```
(24 rows)
```

21.4.6.2 解析器测试

函数 ts_parse 可以直接测试文本搜索解析器。

```
ts_parse(parser_name text, document text,
        OUT tokid integer, OUT token text) returns setof record
```

ts_parse 解析指定的 document 并返回一系列的记录，一条记录代表一个解析生成的 token。每条记录包括标识 token 类型的 tokid，及 token 文本。例如：

```
postgres=# SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
(6 rows)
```

函数 ts_token_type 返回指定解析器的 token 类型及其描述信息。

```
ts_token_type(parser_name text, OUT tokid integer,
             OUT alias text, OUT description text) returns setof record
```

ts_token_type 返回一个表，这个表描述了指定解析器可以识别的每种 token 类型。对于每个 token 类型，表中给出了整数类型的 tokid--用于解析器标记对应的 token 类型； alias一一命名分词器命令中的 token 类型；及简单描述。比如：

```
postgres=# SELECT * FROM ts_token_type('default');
 tokid |      alias      |                description
-------+-----------------+-----------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
```

```
   11 | hword_asciipart | Hyphenated word part, all ASCII
   12 | blank           | Space symbols
   13 | tag             | XML tag
   14 | protocol        | Protocol head
   15 | numhword        | Hyphenated word, letters and digits
   16 | asciihword      | Hyphenated word, all ASCII
   17 | hword           | Hyphenated word, all letters
   18 | url_path        | URL path
   19 | file            | File or path name
   20 | float           | Decimal notation
   21 | int             | Signed integer
   22 | uint            | Unsigned integer
   23 | entity          | XML entity
(23 rows)
```

21.4.6.3 词典测试

函数 ts_lexize 用于进行词典测试。

ts_lexize(dict regdictionary, token text) returns text[]如果输入的 token 可以被词典识别，那么 ts_lexize 返回词素的数组；如果 token 可以被词典识别到它是一个停用词，则 返回空数组；如果是一个不可识别的词则返回 NULL。

比如：

```
postgres=# SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

postgres=# SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

注意

ts_lexize 函数支持单一 token，不支持文本。

21.4.7 限制约束

GBase 8s 的全文检索功能当前限制约束是：

- 每个分词长度必须小于 2K 字节。

- tsvector 结构（分词+位置）的长度必须小于 1 兆字节。
- tsvector 的位置值必须大于 0，且小于等于 16,383。
- 每个分词在文档中位置数必须小于 256，若超过将舍弃后面的位置信息。
- tsquery 中的关键字及对应运算符最大支持到 32768。

22 扩展函数

下表列举了 GBase 8s 中支持的扩展函数，不作为商用特性交付，仅供参考。

分类	函数名称	描述
访问权限查询函数	has_sequence_privilege(user, sequence, privilege)	指定用户是否有访问序列的权限
	has_sequence_privilege(sequence, privilege)	当前用户是否有访问序列的权限
触发器函数	pg_get_triggerdef(oid)	为触发器获取 CREATE [CONSTRAINT] TRIGGER 命令
	pg_get_triggerdef(oid, boolean)	为触发器获取 CREATE [CONSTRAINT] TRIGGER 命令

23 扩展语法

GBase 8s 提供的扩展语法如下。

类别	语法关键字	描述
创建表 CREATE TABLE	INHERITS (parent_table [, …])	支持继承表。
	column_constraint :	支持用 REFERENCES

类别	语法关键字	描述
	REFERENCES reftable [(refcolumn)] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE][ON DELETE action] [ON UPDATE action]	reftable[(refcolumn)] [MATCH FULL |MATCH PARTIAL | MATCH SIMPLE] [ON DELETE action] [ON UPDATE action] 为表创建外键约束。
加载模块	CREATE EXTENSION	把一个新的模块（例如 DBLINK）加载进当前数据库中。
	DROP EXTENSION	删除已加载的模块。
聚集函数	CREATE AGGREGATE	定义一个新的聚集函数。
	ALTER AGGREGATE	修改一个聚集函数的定义。
	DROP AGGREGATE	删除一个现存的聚集函数。

24 类型基础值

INSERT 语句右值引用基础值

在兼容 B 模式下，INSERT 语句支持右值引用，当被引用列有 NOT NULL 约束且没有默认值时，将使用其基础值参与计算。若无基础值则继续使用 NULL 值参与计算(数组类型、用户自定义类型等)。

支持的各类型的基础值如下表所示。

表 24-1 类型基础值

类型	基础值	备注
int	0	
tinyint	0	
smallint	0	
integer	0	
binary_integer	0	
bigint	0	
boolean	f	
numeric	0	
decimal	0	
dec	0	
double precision	0	
float8	0	
float	0	

类型	基础值	备注
char(n)	""	注意：当字符串参与运算时，会根据内置规则进行值类型转换，而定长的字符串存储后的值长度与指定长度一致，会填 充空白字符(不同存储方式 可能会不同)
varchar(n)	""	
varchar2(n)	""	
nchar(n)	""	注意：当字符串参与运算时，会根据内置规则进行值类型转换，而定长的字符串存储后的值长度与指定长度一致，会填充空白字符(不同存储方式 可能会不同)
nvarchar2(n)	""	
nvarchar(n)	""	
date	01-01-1970	
time	00:00:00	
timestamp	当前时间戳	
smalldatetime	Thu Jan 01 00:00:00 1970	
interval year	@ 0	
interval month	@ 0	

类型	基础值	备注
interval day	@ 0	
interval hour	@ 0	
interval minute	@ 0	
interval second	@ 0	
interval day to second	@ 0	
interval day to hour	@ 0	
interval day to minute	@ 0	
interval hour to minute	@ 0	
interval hour to second	@ 0	
interval minute to second	@ 0	
reltime	@ 0	
abstime	Wed Dec 31 16:00:00 1969 PST	

类型	基础值	备注
money	$0.00	
int4range	empty	
blob		数据内容为空的对象
raw		数据内容为空的对象
bytea	\x	
point	(0,0)	
lseg	[(0,0) ,(0,0)]	
box	(0,0) ,(0,0)	
path	((0,0))	
polygon	((0,0))	
circle	<(0,0) ,0>	
cidr	0.0.0.0/32	
inet	0.0.0.0	
macaddr	00:00:00:00:00:00	
BIT		数据内容为空的对象
BIT VARYING		数据内容为空的对象
UUID	00000000-0000-0000-0000-000000000000	
json	null	数据内容为 null

类型	基础值	备注
jsonb	null	数据内容为 null
int8range	empty	
numrange	empty	
tsrange	empty	
tstzrange	empty	
daterange	empty	
hll	\x	
SET	""	
tsvector		数据内容为空的对象
tsquery		数据内容为空的对象
HASH16	0000000000000000	
HASH32	00000000000000000000000000000000	
enum	第一项	

25 GIN 索引

25.1 介绍

GIN（Generalized Inverted Index）通用倒排索引。设计为处理索引项为组合值的情况，查询时需要通过索引搜索出出现在组合值中的特定元素值。例如，文档是由多个单词组成，需要查询出文档中包含的特定单词。

使用 item 表示索引的组合值，key 表示一个元素值。GIN 用来存储和搜索 key，而不是 item。

GIN 索引存储一系列（key、posting list）键值对，这里的 posting list 是一组出现 key 的行 ID。由于每个 item 都可能包含多个 key，同一个行 ID 可能会出现在多个 posting list 中，而每个 key 值只被存储一次，所以在相同的 key 在 item 中出现多次的情况下，GIN 索引是非常简洁的。

因为 GIN 索引的访问方式不需要了解他的运行方式，所以 GIN 索引是通用的。GIN 索引使用为特殊数据类型定义的策略。策略定义了如何从索引选项和查询条件中抽出 key，以及如何确定在查询中包含某些 key 值的行是否实际满足查询条件。

25.2 实现

在内部，GIN 索引包含一个在键上构造的 B-tree 索引，每个键是一个或多个被索引项的一个元素（比如，一个数组的一个成员）。并且页面上每个元组包含了堆指针的 B-tree 的一个指针（一个 posting tree），当列表小到足以和键值一起存储到一个索引元组中时，则是堆指针的一个简单列表（一个 posting list）。

多列 GIN 索引通过在组合值（列号、键值）上建立一个单个的 B-tree 实现。不同列的键值可以有不同的类型。

GIN 快速更新技术

由于倒排索引的本身特性影响，更新一个 GIN 索引可能会比较慢。插入或更新一个堆行可能导致许多往索引的插入。当对表执行 VACUUM 后，或者如果待处理实体的列表太大了（大于 work_mem），这些实体被使用和初始索引创建时用到的相同的 bulk 插入方法，移动到主要的 GIN 数据结构。即使把额外的 VACUUM 开销算进去，这也大大提升了 GIN 索引更新的速度。而且，这种额外开销的工作可以通过后台进程而不是前端查询来处理。

这种方法的主要缺点在于搜索时除了常规的索引还必须要扫描待处理实体的列表。因此，

大的待处理实体的列表会显著的拖慢搜索。另一个缺点是，虽然大多数更新很快，但是一个导致待处理列表（pending list）变得“太大”的更新将引发一个立即清理，并因此比起其它更新会非常慢。恰当的使用 autovacuum 可以弱化这两个问题。

如果一致的响应时间（清理实体速度和更新速度的响应时间）比更新速度更重要，可以通过把 GIN 索引的存储参数 FASTUPDATE 设置为 off 而不使用待处理实体。详细请参考 CREATE INDEX。

部分匹配算法

GIN 可以支持“部分匹配”查询。即：查询并不决定单个或多个键的一个精确的匹配，而是，可能的匹配落在一个合理的狭窄键值范围内（根据 compare 支持函数决定的键值排序顺序）。此时，extractQuery 方法并不返回一个用于精确匹配的键值，取而代之的是，返回一个要被搜索的键值范围的下边界，并且设置 pmatch 为 true。然后，使用 comparePartial 方式扫描这个键值范围。comparePartial 必须为一个相匹配的索引键返回 0，如果不匹配但依然在被搜索范围内时返回小于 0 的值，对超过可以匹配的范围的索引键则返回大于 0 的值。

25.3 扩展性

GIN 索引的接口实现了一个高层次的抽象，要求访问用户仅需要实现被访问数据类型的语义。GIN 层自身可以处理并发操作、记录日志、搜索树结构的任务。

定义 GIN 索引的访问方式所要做的事情就是实现多个用户定义的方法，这些方法定义了键在树中的行为、键与键之间的关系、需要索引的 item、能够使用索引的查询。简而言之，GIN 索引将扩展性与普遍性、代码重用、清晰的接口结合在了一起。

实现 GIN 索引的操作符类有如下四个方法：

- int compare(Datum a, Datum b)

比较两个 key（不是索引的 item）然后返回一个小于零、零或大于零的值，分别表示第一个 key 小于、等于或大于第二个 key。NULL 不会被传入这个函数。

Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)

给定一个要被索引的 item，返回一个对应 key 的数组。返回 key 的数目必须存储在*nkeys 中。如果任何 key 都可能为 NULL，还要分配包含*nkeys 个布尔元素的数组，将地址存储到*nullFlags，并且根据需要设置 NULL 值。 如果所有 key 都是非 NULL，可以让*nullFlags 保持为 NULL（他的初始值）。如果 item 不包含任何 key，返回值可以为 NULL。

- Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch,

Pointer **extra_data, bool **nullFlags, int32 *searchMode)

给定一个被查询的值，返回一个对应的 key 的数组。也就是说，query 是可索引操作符右侧的值，而该操作符左侧是被索引的字段。 n 是操作符类中操作符的策略号。通常，extractQuery 需要参考 n 来决定 query 的数据类型以及抽取键值的方法。返回 key 的个数必须存放在*nkeys 中。如果任何 key 都可能为 NULL，还要分配包含*nkeys 个布尔元素的数组，将地址存储到*nullFlags，并且根据需要设置 NULL 值。 如果所有 key 都是非 NULL 的，可以让*nullFlags 保持为 NULL（他的初始值）。如果 query 不包含任何 key，返回值可以为 NULL。

searchMode 是一个输出参数，他允许 extractQuery 指定一些关于如何执行搜索的细节。如果设置*searchMode 为 GIN_SEARCH_MODE_DEFAULT（这也是调用函数前此参数的初始化值），只有那些至少返回一个 key 的 item 才能被考虑作为候选匹配项。如果设置*searchMode 为 GIN_SEARCH_MODE_INCLUDE_EMPTY，除了包含至少一个匹配 key 的 item 之外，根本不包含任何 key 的 item 也被考虑作为候选匹配项。（这个模式对于实现像“是否是子集”这样的操作是有用的）如果设置*searchMode 为 GIN_SEARCH_MODE_ALL，索引中所有非 NULL 的 item 都被考虑作为候选匹配项，不管他们是否匹配返回 key 中的任何一个。

pmatch 是一个允许支持部分匹配的输出参数。如果使用此参数，extractQuery 必须分配有*nkeys 个布尔元素的数组，并把数组地址保存到*pmatch。如果需要部分匹配相应的 key，则数组的每个元素应该设置为 TRUE；如果不需要匹配，则设置为 FALSE。如果设置*pmatch 为 NULL，则假设 GIN 不需要部分匹配。在函数调用前这个值被初始化为 NULL，因此，对于不支持部分匹配的操作符类，可以忽略这个参数。

extra_data 是一个允许 extractQuery 以 consistent 和 comparePartial 的方式传递额外数据的输出参数。如果使用他，extractQuery 必须分配一个包含*nkeys 个 Pointer 元素的数组，并把数组地址保存到*extra_data，然后把他想附加的东西存储到各个独立的指针中。在函数调用前这个值初始化为 NULL，因此，对于不需要附加数据的操作符类，可以忽略这个参数。如果设置了*extra_data，那么以 consistent 方式传递整个数组，使用 comparePartial 方式传递适当的元素。

- bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])

如果被索引项满足 StrategyNumber 为 n 的查询操作符则返回 TRUE。这个函数并不直接访问被索引项的值，因为 GIN 并没有精确的把项目保存下来，但是需要知道从查询中提取

的哪些键值出现在给定的被索引项中。 check 数组的长度是 nkeys，这个与 query 调用 extractQuery 函数返回的键值的数目相同。如果索引项包含了相应的查询键，check 数组中对应的元素值就是 TRUE。比如，如果(check[i] == TRUE)，那么意味着 extractQuery 的结果数组的第 i 个键出现在索引项中。考虑可能会用到 consistent 方式，原始的 query 也被作为参数传入进来。与此相同的还有 extractQuery 函数返回的 queryKeys[]和 nullFlags[]。 extra_data 是 extractQuery 函数返回的额外数据数组，如果没有的话就是 NULL。

当 extractQuery 在 queryKeys[]中返回一个 NULL 的键值，如果被索引项包含 NULL 键值，相应的 check[]中的元素是 TRUE。也就是说，check[]的语义很像 IS NOT DISTINCT FROM。如果需要知道是通常值匹配还是 NULL 匹配，consistent 函数可以检查相应的 nullFlags[]元素。

成功执行后，如果堆元组需要针对查询运算符进行重新检查，*recheck 需要设置为 TRUE，如果索引测试已经是精确的了，则设为 FALSE。也就是说，FALSE 的返回值确保堆元组不匹配这个查询；设置*recheck 为 FALSE 的 TRUE 的返回值确保堆元组匹配这个查询；设置*recheck 为 TRUE 的 TRUE 的返回值意味着堆元组可能匹配这个查询，因此需要通过直接对照原始索引项对查询运算符进行获取和重新检查。

GIN 操作符类可以可选地提供第五个函数。

- int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)

比较一个部分匹配查询键和一个索引键。返回一个整型值，它的符号代表了不同的含义：小于 0 意味着索引键不匹配查询，但是索引扫描应该继续； 0 意味着索引键匹配查询；大于 0 指示应该终止索引扫描，因为不可能再有更多的匹配。在需要确定何时结束扫描的语义的情况下，这里提供了生成部分一致查询的操作符的策略号 n。同样的， extra_data 是 extractQuery 生成的额外数据数组中的相应元素，如果没有对应的元素，则为 NULL。 NULL 的键永远不会被传入这个函数。

为了支持”部分匹配”查询，一个操作符类必须提供 comparePartial 方法，并且当遇到部分匹配查询时，他的 extractQuery 方法必须设置 pmatch 参数。详细信息请参考部分匹配算法。

上面的各种 Datum 值的实际数据类型根据操作符类的不同而不同。传入到 extractValue 中的项目值总是操作符类的输入类型，所有的键值类型必须是这个类的 STORAGE 类型。传入到 extractQuery 和 consistent 的 query 参数的类型是由策略号识别的类成员操作符的右操作数的输入类型。他不需要和项目类型相同，只要可以从中抽取出正确类型的键值。

25.4 GIN 提示与技巧

创建 vs 插入

由于可能要为每个项目插入很多键，所以 GIN 索引的插入可能比较慢。对于向表中大量插入的操作，我们建议先删除 GIN 索引，在完成插入之后再重建索引。与 GIN 索引创建、查询性能相关的 GUC 参数如下：

- maintenance_work_mem

  GIN 索引的构建时间对 maintenance_work_mem 的设置非常敏感。

- work_mem

  在向启用了 FASTUPDATE 的 GIN 索引执行插入操作的期间，只要待处理实体列表的大小超过了 work_mem，系统就会清理这个列表。为了避免可观察到的响应时间的大起大落，让待处理实体列表在后台被清理是比较合适的（比如通过 autovacuum）。前端清理操作可以通过增加 work_mem 或者执行 autovacuum 来避免。然而，扩大 work_mem 意味着如果发生了前端清理，那么他的执行时间将更长。

- gin_fuzzy_search_limit

  开发 GIN 索引的主要目的是为了让 GBase 8s 支持高度可伸缩的全文索引，并且常常会遇见全文索引返回海量结果的情形。而且，这经常发生在查询高频词的时候，因而这样的结果集没什么用处。因为从磁盘读取大量记录并对其进行排序会消耗大量资源，这在产品环境下是不能接受的。为了控制这种情况，GIN 索引有一个可配置的返回结果行数的软上限的配置参数 gin_fuzzy_search_limit。缺省值 0 表示没有限制。如果设置了非零值，那么返回结果就是从完整结果集中随机选择的一部分。“软上限”的意思是返回结果的实际数量可能与指定的限制有偏差，这取决于查询和系统随机数生成器的质量系统操作。

26 Schema

GBase 8s 的 Schema 如下表所示。

Schema 名称	描述
blockchain	用于存储账本数据库特性中创建防篡改表时自动创建的用户历史表。

Schema 名称	描述
cstore	该模式用于储存列存表相关的辅助表如 cudesc 或者 delta 表。
db4ai	用于管理 AI 训练中不同版本的数据信息。
dbe_perf	DBE_PERF Schema 内视图主要用来诊断性能问题，也是 WDR Snapshot 的数据来源。数据库安装后，默认只有初始用户和监控管理员具有模式 dbe_perf 的权限，有权查看该模式下的视图和函数。
dbe_pldebugger	用于调试 plpgsql 函数及存储过程。
snapshot	用于管理 WDR snapshot 的相关的数据信息，默认初始化用户或监控管理员用户可以访问。
sqladvisor	用于分布列推荐，GBase 8s 不可用。
sys	用于提供系统信息视图接口。
pg_catalog	用于维护系统的 catalog 信息，包含系统表和所有内置数据类型、函数、操作符。
pg_toast	用于存储大对象（系统内部使用）。
public	公共模式，缺省时，创建的表（以及其它对象）自动放入该模式。
pkg_service	用于管理 package 服务相关信息。
pkg_util	用于管理 package 工具相关信息。

26.1 Information Schema

信息模式本身是一个名为 information_schema 的模式。这个模式自动存在于所有数据库中。信息模式由一组视图构成，它们包含定义在当前数据库中对象的信息。这个模式的拥有者是初始数据库用户，并且该用户自然地拥有这个模式上的所有特权，包括删除它的能力。

信息模式继承自开源 PGXC/PG，相关细节描述请参见 PGXC/PG 官方文档，链接如下：

http://postgres-xc.sourceforge.net/docs/1_1/information-schema.html

https://www.postgresql.org/docs/9.2/information-schema.html

下面章节只显示未在上述链接内的视图信息。

26.1.1 _PG_FOREIGN_DATA_WRAPPERS

显示外部数据封装器的信息。该视图只有 sysadmin 权限可以查看。

名称	类型	描述
oid	oid	外部数据封装器的 oid。
fdwowner	oid	外部数据封装器的所有者的 oid。
fdwoptions	text[]	外部数据封装器指定选项，使用“keyword=value”格式的字符串。
foreign_data_wrapper_catalog	information_schema.sql_identifier	外部封装器所在的数据库名称（永远为当前数据库）。
foreign_data_wrapper_name	information_schema.sql_identifier	外部数据封装器名

		称。
authorization_identifier	information_schema.sql_identifier	外部数据封装器所有者的角色名称。
foreign_data_wrapper_language	information_schema.character_data	外部数据封装器的实现语言。

26.1.2 _PG_FOREIGN_SERVERS

显示外部服务器的信息。该视图只有 sysadmin 权限可以查看。

名称	类型	描述
oid	oid	外部服务器的 oid。
srvoptions	text[]	外部服务器指定选项，使用“keyword=value”格式的字符串。
foreign_server_catalog	information_schema.sql_identifier	外部服务器所在 database 名称（永远为当前数据库）。
foreign_server_name	information_schema.sql_identifier	外部服务器名称。
foreign_data_wrapper_catalog	information_schema.sql_identifier	外部数据封装器所在 database 名称（永远为当前数据库）。
foreign_data_wrapper_name	information_schema.sql_identifier	外部数据封装器名称。

foreign_server_type	information_schema.character_data	外部服务器的类型。
foreign_server_version	information_schema.character_data	外部服务器的版本。
authorization_identifier	information_schema.sql_identifier	外部服务器的所有者的角色名称。

26.1.3 _PG_FOREIGN_TABLE_COLUMNS

显示外部表的列信息。该视图只有 sysadmin 权限可以查看。

名称	类型	描述
nspname	name	schema 名称。
relname	name	表名称。
attname	name	列名称。
attfdwoptions	text[]	外部数据封装器的属性选项，使用“keyword=value”格式的字符串。

26.1.4 _PG_FOREIGN_TABLES

存储所有的定义在本数据库的外部表信息。只显示当前用户有权访问的外部表信息。该视图只有 sysadmin 权限可以查看。

名称	类型	描述
foreign_table_catalog	information_schema.sql_identifier	外部表所在的数据库名称（永远是当前数据库）。

foreign_table_schema	name	外部表的 schema 名称。
foreign_table_name	name	外部表的名称。
ftoptions	text[]	外部表的可选项。
foreign_server_catalog	information_schema.sql_identifier	外部服务器所在的数据库名称（永远是当前数据库）。
foreign_server_name	information_schema.sql_identifier	外部服务器的名称。
authorization_identifier	information_schema.sql_identifier	所有者的角色名称。

26.1.5 _PG_USER_MAPPINGS

存储从本地用户到远程的映射。该视图只有 sysadmin 权限可以查看。

名称	类型	描述
oid	oid	从本地用户到远程的映射的 oid。
umoptions	text[]	用户映射指定选项，使用“keyword=value”格式的字符串。
umuser	oid	被映射的本地用户的 OID，如果用户映射是公共的则为 0。
authorization_identifier	information_schema.sql_identifier	本地用户角色名称。

foreign_server_catalog	information_schema.sql_identifier	外部服务器定义所在的 database 名称。
foreign_server_name	information_schema.sql_identifier	外部服务器名称。
srvowner	information_schema.sql_identifier	外部服务器所有者。

26.1.6 INFORMATION_SCHEMA_CATALOG_NAME

用来显示当前所在的 database 的名称。

名称	类型	描述
catalog_name	information_schema.sql_identifier	当前 database 的名称。

26.2 DBE_PERF Schema

DBE_PERF Schema 内视图主要用来诊断性能问题，也是 WDR Snapshot 的数据来源。数据库安装后，默认只有初始用户具有模式 dbe_perf 的权限。若是由旧版本升级而来，为保持权限的前向兼容，模式 dbe_perf 的权限与旧版本保持一致。从 OS、Instance、Memory 等多个维度划分组织视图，并且符合如下命名规范：

GLOBAL_开头的视图，代表从数据库节点请求数据，并将数据追加对外返回，不会处理数据。

SUMMARY_开头的视图，代表是将 GBase 8s 内的数据概述，多数情况下是返回数据库节点（有时只有数据库主节点的）的数据，会对数据进行加工和汇聚。

非这两者开头的视图，一般代表本地视图，不会向其它数据库节点请求数据。

26.2.1 OS

26.2.1.1 OS_RUNTIME

显示当前操作系统运行的状态信息。

名称	类型	描述
id	integer	编号。
name	text	操作系统运行状态名称。
value	numeric	操作系统运行状态值。
comments	text	操作系统运行状态注释。
cumulative	boolean	操作系统运行状态的值是否为累加值。

26.2.1.2 GLOBAL_OS_RUNTIME

提供 GBase 8s 中所有正常节点下的操作系统运行状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
id	integer	编号。
name	text	操作系统运行状态名称。
value	numeric	操作系统运行状态值。
comments	text	操作系统运行状态注释。
cumulative	boolean	操作系统运行状态的值是否为累加值。

26.2.1.3 OS_THREADS

提供当前节点下所有线程的状态信息。

名称	类型	描述
node_name	text	数据库进程名称。
pid	bigint	数据库进程中正在运行的线程号。
lwpid	integer	与 pid 对应的轻量级线程号。
thread_name	text	与 pid 对应的线程名称。
creation_time	timestamp with time zone	与 pid 对应的线程创建的时间。

26.2.1.4 GLOBAL_OS_THREADS

提供 GBase 8s 中所有正常节点下的线程状态信息。

名称	类型	描述
node_name	text	数据库进程名称。
pid	bigint	当前节点进程中正在运行的线程号。
lwpid	integer	与 pid 对应的轻量级线程号。
thread_name	text	与 pid 对应的线程名称。
creation_time	timestamp with time zone	与 pid 对应的线程创建的时间。

26.2.2 Instance

26.2.2.1 INSTANCE_TIME

提供当前节点下的各种时间消耗信息，主要分为以下类型：

DB_TIME：作业在多核下的有效时间花销。

CPU_TIME：CPU 的时间花销。

EXECUTION_TIME：执行器内的时间花销。

PARSE_TIME：SQL 解析的时间花销。

PLAN_TIME：生成 Plan 的时间花销。

REWRITE_TIME：SQL 重写的时间花销。

PL_EXECUTION_TIME：plpgsql（存储过程）执行的时间花销。

PL_COMPILATION_TIME：plpgsql（存储过程）编译的时间花销。

NET_SEND_TIME：网络上的时间花销。

DATA_IO_TIME：IO 上的时间花销。

名称	类型	描述
stat_id	integer	统计编号。
stat_name	text	类型名称。
value	bigint	时间值（单位：微秒）。

26.2.2.2 GLOBAL_INSTANCE_TIME

提供 GBase 8s 中所有正常节点下的各种时间消耗信息(时间类型见 instance_time 视图)。

名称	类型	描述

名称	类型	描述
node_name	name	数据库进程的名称。
stat_id	integer	统计编号。
stat_name	text	类型名称。
value	bigint	时间值（单位：微秒）。

26.2.3 Memory

26.2.3.1 MEMORY_NODE_DETAIL

显示某个数据库节点内存使用情况。

名称	类型	描述
nodename	text	节点名称。
memorytype	text	内存的名称。 max_process_memory：GBase 8s 实例所占用的内存大小。 process_used_memory：进程所使用的内存大小。 max_dynamic_memory：最大动态内存。 dynamic_used_memory：已使用的动态内存。 dynamic_peak_memory：内存的动态峰值。 dynamic_used_shrctx：最大动态共享内存上下文。 dynamic_peak_shrctx：共享内存上下文的动态峰值。 max_shared_memory：最大共享内存。

名称	类型	描述
		shared_used_memory：已使用的共享内存。 max_cstore_memory：列存所允许使用的最大内存。 cstore_used_memory：列存已使用的内存大小。 max_sctpcomm_memory：sctp 通信所允许使用的最大内存。 sctpcomm_used_memory：sctp 通信已使用的内存大小。 sctpcomm_peak_memory：sctp 通信的内存峰值。 other_used_memory：其他已使用的内存大小。 gpu_max_dynamic_memory：GPU 最大动态内存。 gpu_dynamic_used_memory：GPU 已使用的动态内存。 gpu_dynamic_peak_memory：GPU 内存的动态峰值。 pooler_conn_memory：链接池申请内存计数。 pooler_freeconn_memory：链接池空闲连接的内存计数。 storage_compress_memory：存储模块压缩使用的内存大小。 udf_reserved_memory：UDF 预留的内存大小。
memorymbytes	integer	内存使用的大小，单位为 MB。

26.2.3.2 GLOBAL_MEMORY_NODE_DETAIL

显示当前 GBase 8s 中所有正常节点下的内存使用情况。

名称	类型	描述
nodename	text	数据库进程名称。

名称	类型	描述
memorytype	text	内存使用的名称。 max_process_memory：GBase 8s 实例所占用的内存大小。 process_used_memory：进程所使用的内存大小。 max_dynamic_memory：最大动态内存。 dynamic_used_memory：已使用的动态内存。 dynamic_peak_memory：内存的动态峰值。 dynamic_used_shrctx：最大动态共享内存上下文。 dynamic_peak_shrctx：共享内存上下文的动态峰值。 max_shared_memory：最大共享内存。 shared_used_memory：已使用的共享内存。 max_cstore_memory：列存所允许使用的最大内存。 cstore_used_memory：列存已使用的内存大小。 max_sctpcomm_memory：sctp 通信所允许使用的最大内存。 sctpcomm_used_memory：sctp 通信已使用的内存大小。 sctpcomm_peak_memory：sctp 通信的内存峰值。 other_used_memory：其他已使用的内存大小。 gpu_max_dynamic_memory：GPU 最大动态内存。 gpu_dynamic_used_memory：GPU 已使用的动态内存。 gpu_dynamic_peak_memory：GPU 内存的动态峰值。 pooler_conn_memory：链接池申请内存计数。 pooler_freeconn_memory：链接池空闲连接的内存计数。 storage_compress_memory：存储模块压缩使用的内存大小。 udf_reserved_memory：UDF 预留的内存大小。

名称	类型	描述
memorymbytes	integer	内存使用的大小，单位为 MB。

26.2.3.3 SHARED_MEMORY_DETAIL

查询当前节点所有已产生的共享内存上下文的使用信息。

名称	类型	描述
contextname	text	内存上下文的名称。
level	smallint	内存上下文的级别。
parent	text	上级内存上下文。
totalsize	bigint	共享内存总大小（单位：字节）。
freesize	bigint	共享内存剩余大小（单位：字节）。
usedsize	bigint	共享内存使用大小（单位：字节）。

26.2.3.4 GLOBAL_SHARED_MEMORY_DETAIL

查询 GBase 8s 中所有正常节点下的共享内存上下文的使用信息。

名称	类型	描述
node_name	name	数据库进程名称。
contextname	text	内存上下文的名称。

名称	类型	描述
level	smallint	内存上下文的级别。
parent	text	上级内存上下文。
totalsize	bigint	共享内存总大小（单位：字节）。
freesize	bigint	共享内存剩余大小（单位：字节）。
usedsize	bigint	共享内存使用大小（单位：字节）。

26.2.4 File

26.2.4.1 FILE_IOSTAT

通过对数据文件 IO 的统计，反映数据的 IO 性能，用以发现 IO 操作异常等性能问题。

名称	类型	描述
filenum	oid	文件标识。
dbid	oid	数据库标识。
spcid	oid	表空间标识。
phyrds	bigint	读物理文件的数目。
phywrts	bigint	写物理文件的数目。
phyblkrd	bigint	读物理文件块的数目。

名称	类型	描述
phyblkwrt	bigint	写物理文件块的数目。
readtim	bigint	读文件的总时长（单位：微秒）。
writetim	bigint	写文件的总时长（单位：微秒）。
avgiotim	bigint	读写文件的平均时长（单位：微秒）。
lstiotim	bigint	最后一次读文件时长（单位：微秒）。
miniotim	bigint	读写文件的最小时长（单位：微秒）。
maxiowtm	bigint	读写文件的最大时长（单位：微秒）。

26.2.4.2 SUMMARY_FILE_IOSTAT

通过 GBase 8s 内对数据文件汇聚 IO 的统计，反映数据的 IO 性能，用以发现 IO 操作异常等性能问题。

名称	类型	描述
filenum	oid	文件标识。
dbid	oid	数据库标识。
spcid	oid	表空间标识。
phyrds	numeric	读物理文件的数目。

名称	类型	描述
phywrts	numeric	写物理文件的数目。
phyblkrd	numeric	读物理文件块的数目。
phyblkwrt	numeric	写物理文件块的数目。
readtim	numeric	读文件的总时长（单位：微秒）。
writetim	numeric	写文件的总时长（单位：微秒）。
avgiotim	bigint	读写文件的平均时长（单位：微秒）。
lstiotim	bigint	最后一次读文件时长（单位：微秒）。
miniotim	bigint	读写文件的最小时长（单位：微秒）。
maxiowtm	bigint	读写文件的最大时长（单位：微秒）。

26.2.4.3 GLOBAL_FILE_IOSTAT

得到所有节点上的数据文件 IO 统计信息。

名称	类型	描述
node_name	name	数据库进程名称。
filenum	oid	文件标识。

名称	类型	描述
dbid	oid	数据库标识。
spcid	oid	表空间标识。
phyrds	bigint	读物理文件的数目。
phywrts	bigint	写物理文件的数目。
phyblkrd	bigint	读物理文件块的数目。
phyblkwrt	bigint	写物理文件块的数目。
readtim	bigint	读文件的总时长（单位：微秒）。
writetim	bigint	写文件的总时长（单位：微秒）。
avgiotim	bigint	读写文件的平均时长（单位：微秒）。
lstiotim	bigint	最后一次读文件时长（单位：微秒）。
miniotim	bigint	读写文件的最小时长（单位：微秒）。
maxiowtm	bigint	读写文件的最大时长（单位：微秒）。

26.2.4.4 FILE_REDO_IOSTAT

本节点 Redo（WAL）相关的统计信息。

名称	类型	描述
phywrts	bigint	向 wal buffer 中写的次数。
phyblkwrt	bigint	向 wal buffer 中写的 block 的块数。
writetim	bigint	向 xlog 文件中写操作的时间（单位：微秒）。
avgiotim	bigint	平均写 xlog 的时间（writetim/phywrts）（单位：微秒）。
lstiotim	bigint	最后一次写 xlog 的时间（单位：微秒）。
miniotim	bigint	最小的写 xlog 时间（单位：微秒）。
maxiowtm	bigint	最大的写 xlog 时间（单位：微秒）。

26.2.4.5 SUMMARY_FILE_REDO_IOSTAT

GBase 8s 内汇聚所有的 Redo（WAL）相关的统计信息。

名称	类型	描述
phywrts	numeric	向 wal buffer 中写的次数。
phyblkwrt	numeric	向 wal buffer 中写的 block 的块数。
writetim	numeric	向 xlog 文件中写操作的时间（单位：微秒）。

名称	类型	描述
avgiotim	bigint	平均写 xlog 的时间（writetim/phywrts）（单位：微秒）。
lstiotim	bigint	最后一次写 xlog 的时间（单位：微秒）。
miniotim	bigint	最小的写 xlog 时间（单位：微秒）。
maxiowtm	bigint	最大的写 xlog 时间（单位：微秒）。

26.2.4.6 GLOBAL_FILE_REDO_IOSTAT

得到 GBase 8s 内各节点的 Redo（WAL）相关统计信息。

名称	类型	描述
node_name	name	数据库进程名称。
phywrts	bigint	向 wal buffer 中写的次数。
phyblkwrt	bigint	向 wal buffer 中写的 block 的块数。
writetim	bigint	向 xlog 文件中写操作的时间（单位：微秒）。
avgiotim	bigint	平均写 xlog 的时间（writetim/phywrts）（单位：微秒）。
lstiotim	bigint	最后一次写 xlog 的时间（单位：微秒）。

名称	类型	描述
miniotim	bigint	最小的写 xlog 时间（单位：微秒）。
maxiowtm	bigint	最大的写 xlog 时间（单位：微秒）。

26.2.4.7 LOCAL_REL_IOSTAT

获取当前节点中数据文件 IO 状态的累计值，显示为所有数据文件 IO 状态的总和。

名称	类型	描述
phyrds	bigint	读物理文件的数目。
phywrts	bigint	写物理文件的数目。
phyblkrd	bigint	读物理文件的块的数目。
phyblkwrt	bigint	写物理文件的块的数目。

26.2.4.8 GLOBAL_REL_IOSTAT

获取所有节点上的数据文件 IO 统计信息。

名称	类型	描述
node_name	name	数据库进程名称。
phyrds	bigint	读物理文件的数目。
phywrts	bigint	写物理文件的数目。

名称	类型	描述
phyblkrd	bigint	读物理文件块的数目。
phyblkwrt	bigint	写物理文件块的数目。

26.2.4.9 SUMMARY_REL_IOSTAT

获取所有节点上的数据文件 IO 统计信息。

名称	类型	描述
phyrds	numeric	读物理文件的数目。
phywrts	numeric	写物理文件的数目。
phyblkrd	numeric	读物理文件的块的数目。
phyblkwrt	numeric	写物理文件的块的数目。

26.2.5 Object

26.2.5.1 STAT_USER_TABLES

显示当前节点所有命名空间中用户自定义普通表的状态信息。

名称	类型	描述
relid	oid	表的 OID。
schemaname	name	该表的模式名。

名称	类型	描述
relname	name	表名。
seq_scan	bigint	该表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。
idx_scan	bigint	该表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（即没有更新所需的单独索引）。
n_live_tup	bigint	估计活跃行数。
n_dead_tup	bigint	估计死行数。
last_vacuum	timestamp with time zone	最后一次该表是手动清理的（不计算 VACUUM FULL）时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。

名称	类型	描述
last_analyze	timestamp with time zone	上次手动分析该表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析的时间。
vacuum_count	bigint	该表被手动清理的次数(不计算 VACUUM FULL）。
autovacuum_count	bigint	该表被 autovacuum 清理的次数。
analyze_count	bigint	该表被手动分析的次数。
autoanalyze_count	bigint	该表被 autovacuum 守护进程分析的次数。

26.2.5.2 SUMMARY_STAT_USER_TABLES

GBase 8s 内汇聚所有命名空间中用户自定义普通表的状态信息。

名称	类型	描述
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	numeric	此表发起的顺序扫描数。
seq_tup_read	numeric	顺序扫描抓取的活跃行数。
idx_scan	numeric	此表发起的索引扫描数。

名称	类型	描述
idx_tup_fetch	numeric	索引扫描抓取的活跃行数。
n_tup_ins	numeric	插入行数。
n_tup_upd	numeric	更新行数。
n_tup_del	numeric	删除行数。
n_tup_hot_upd	numeric	HOT 更新行数（即没有更新所需的单独索引）。
n_live_tup	numeric	估计活跃行数。
n_dead_tup	numeric	估计死行数。
last_vacuum	timestamp with time zone	最后一次此表是手动清理的（不计算 VACUUM FULL）时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。
last_analyze	timestamp with time zone	上次手动分析这个表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析的时间。
vacuum_count	numeric	这个表被手动清理的次数（不计算 VACUUM FULL）。

名称	类型	描述
autovacuum_count	numeric	这个表被 autovacuum 清理的次数。
analyze_count	numeric	这个表被手动分析的次数。
autoanalyze_count	numeric	这个表被 autovacuum 守护进程分析的次数。

26.2.5.3 GLOBAL_STAT_USER_TABLES

得到各节点所有命名空间中用户自定义普通表的状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	表的 OID。
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	bigint	此表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。
idx_scan	bigint	此表发起的索引扫描数。

名称	类型	描述
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（即没有更新所需的单独索引）。
n_live_tup	bigint	估计活跃行数。
n_dead_tup	bigint	估计死行数。
last_vacuum	timestamp with time zone	最后一次此表是手动清理的（不计算 VACUUM FULL）时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。
last_analyze	timestamp with time zone	上次手动分析这个表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析的时间。
vacuum_count	bigint	这个表被手动清理的次数（不计算 VACUUM FULL）。

名称	类型	描述
autovacuum_count	bigint	这个表被 autovacuum 清理的次数。
analyze_count	bigint	这个表被手动分析的次数。
autoanalyze_count	bigint	这个表被 autovacuum 守护进程分析的次数。

26.2.5.4 STAT_USER_INDEXES

显示数据库中用户自定义普通表的索引状态信息。

名称	类型	描述
relid	oid	此索引的表的 OID。
indexrelid	oid	索引的 OID。
schemaname	name	索引的模式名。
relname	name	索引的表名。
indexrelname	name	索引名。
idx_scan	bigint	索引上开始的索引扫描数。
idx_tup_read	bigint	通过索引上扫描返回的索引项数。
idx_tup_fetch	bigint	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.5 SUMMARY_STAT_USER_INDEXES

GBase 8s 内汇聚所有数据库中用户自定义普通表的索引状态信息。

名称	类型	描述
schemaname	name	索引中模式名。
relname	name	索引的表名。
indexrelname	name	索引名。
idx_scan	numeric	索引上开始的索引扫描数。
idx_tup_read	numeric	通过索引上扫描返回的索引项数。
idx_tup_fetch	numeric	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.6 GLOBAL_STAT_USER_INDEXES

得到各节点数据库中用户自定义普通表的索引状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	这个索引的表的 OID。
indexrelid	oid	索引的 OID。
schemaname	name	索引中模式名。

名称	类型	描述
relname	name	索引的表名。
indexrelname	name	索引名。
idx_scan	bigint	索引上开始的索引扫描数。
idx_tup_read	bigint	通过索引上扫描返回的索引项数。
idx_tup_fetch	bigint	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.7 STAT_SYS_TABLES

显示单节点内 pg_catalog、information_schema 以及 pg_toast 模式下的所有系统表的统计信息。

名称	类型	描述
relid	oid	表的 OID。
schemaname	name	该表的模式名。
relname	name	表名。
seq_scan	bigint	该表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。
idx_scan	bigint	该表发起的索引扫描数。

名称	类型	描述
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。
n_live_tup	bigint	估计活跃行数。
n_dead_tup	bigint	估计死行数。
last_vacuum	timestamp with time zone	最后一次该表是手动清理的（不计算 VACUUM FULL）时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。
last_analyze	timestamp with time zone	上次手动分析该表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析的时间。
vacuum_count	bigint	这个表被手动清理的次数（不计算 VACUUM FULL）。

名称	类型	描述
autovacuum_count	bigint	该表被 autovacuum 清理的次数。
analyze_count	bigint	该表被手动分析的次数。
autoanalyze_count	bigint	该表被 autovacuum 守护进程分析的次数。

26.2.5.8 SUMMARY_STAT_SYS_TABLES

GBase 8s 内汇聚 pg_catalog、information_schema 以及 pg_toast 模式下的所有系统表的统计信息。

名称	类型	描述
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	numeric	此表发起的顺序扫描数。
seq_tup_read	numeric	顺序扫描抓取的活跃行数。
idx_scan	numeric	此表发起的索引扫描数。
idx_tup_fetch	numeric	索引扫描抓取的活跃行数。
n_tup_ins	numeric	插入行数。
n_tup_upd	numeric	更新行数。

名称	类型	描述
n_tup_del	numeric	删除行数。
n_tup_hot_upd	numeric	HOT 更新行数（比如没有更新所需的单独索引）。
n_live_tup	numeric	估计活跃行数。
n_dead_tup	numeric	估计死行数。
last_vacuum	timestamp with time zone	最后一次此表是手动清理的（不计算 VACUUM FULL）时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。
last_analyze	timestamp with time zone	上次手动分析这个表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析的时间。
vacuum_count	numeric	这个表被手动清理的次数（不计算 VACUUM FULL）。
autovacuum_count	numeric	这个表被 autovacuum 清理的次数。
analyze_count	numeric	这个表被手动分析的次数。

名称	类型	描述
autoanalyze_count	numeric	这个表被 autovacuum 守护进程分析的次数。

26.2.5.9 GLOBAL_STAT_SYS_TABLES

得到各节点 pg_catalog、information_schema 以及 pg_toast 模式下的所有系统表的统计信息。

名称	类型	描述
node_name	name	节点名称。
relid	oid	表的 OID。
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	bigint	此表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。
idx_scan	bigint	此表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。

名称	类型	描述
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。
n_live_tup	bigint	估计活跃行数。
n_dead_tup	bigint	估计死行数。
last_vacuum	timestamp with time zone	最后一次此表是手动清理的（不计算 VACUUM FULL）时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。
last_analyze	timestamp with time zone	上次手动分析这个表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析的时间。
vacuum_count	bigint	这个表被手动清理的次数（不计算 VACUUM FULL）。
autovacuum_count	bigint	这个表被 autovacuum 清理的次数。
analyze_count	bigint	这个表被手动分析的次数。

名称	类型	描述
autoanalyze_count	bigint	这个表被 autovacuum 守护进程分析的次数。

26.2.5.10 STAT_SYS_INDEXES

显示 pg_catalog、information_schema 以及 pg_toast 模式中所有系统表的索引状态信息。

名称	类型	描述
relid	oid	此索引的表的 OID。
indexrelid	oid	索引的 OID。
schemaname	name	索引的模式名。
relname	name	索引的表名。
indexrelname	name	索引名。
idx_scan	bigint	索引上开始的索引扫描数。
idx_tup_read	bigint	通过索引上扫描返回的索引项数。
idx_tup_fetch	bigint	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.11 SUMMARY_STAT_SYS_INDEXES

GBase 8s 内汇聚 pg_catalog、information_schema 以及 pg_toast 模式中所有系统表的索引状态信息。

名称	类型	描述
schemaname	name	索引中模式名。
relname	name	索引的表名。
indexrelname	name	索引名。
idx_scan	numeric	索引上开始的索引扫描数。
idx_tup_read	numeric	通过索引上扫描返回的索引项数。
idx_tup_fetch	numeric	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.12 GLOBAL_STAT_SYS_INDEXES

得到各节点 pg_catalog、information_schema 以及 pg_toast 模式中所有系统表的索引状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	这个索引的表的 OID。
indexrelid	oid	索引的 OID。
schemaname	name	索引中模式名。
relname	name	索引的表名。

名称	类型	描述
indexrelname	name	索引名。
idx_scan	bigint	索引上开始的索引扫描数。
idx_tup_read	bigint	通过索引上扫描返回的索引项数。
idx_tup_fetch	bigint	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.13 STAT_ALL_TABLES

本节点内数据库中每个表（包括 TOAST 表）的一行的统计信息。

名称	类型	描述
relid	oid	表的 OID。
schemaname	name	该表的模式名。
relname	name	表名。
seq_scan	bigint	该表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。
idx_scan	bigint	该表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。

名称	类型	描述
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。
n_live_tup	bigint	估计活跃行数。
n_dead_tup	bigint	估计死行数。
last_vacuum	timestamp with time zone	最后一次该表是手动清理的（不计算 VACUUM FULL）的时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。
last_analyze	timestamp with time zone	上次手动分析该表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析时间。
vacuum_count	bigint	该表被手动清理的次数（不计算 VACUUM FULL）。

名称	类型	描述
autovacuum_count	bigint	该表被 autovacuum 清理的次数。
analyze_count	bigint	该表被手动分析的次数。
autoanalyze_count	bigint	该表被 autovacuum 守护进程分析的次数。

26.2.5.14 SUMMARY_STAT_ALL_TABLES

GBase 8s 内汇聚数据库中每个表的一行（包括 TOAST 表）的统计信息。

名称	类型	描述
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	numeric	此表发起的顺序扫描数。
seq_tup_read	numeric	顺序扫描抓取的活跃行数。
idx_scan	numeric	此表发起的索引扫描数。
idx_tup_fetch	numeric	索引扫描抓取的活跃行数。
n_tup_ins	numeric	插入行数。
n_tup_upd	numeric	更新行数。

名称	类型	描述
n_tup_del	numeric	删除行数。
n_tup_hot_upd	numeric	HOT 更新行数(比如没有更新所需的单独索引)。
n_live_tup	numeric	估计活跃行数。
n_dead_tup	numeric	估计死行数。
last_vacuum	timestamp with time zone	最后一次此表是手动清理的(不计算 VACUUM FULL)的时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。
last_analyze	timestamp with time zone	上次手动分析这个表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析时间。
vacuum_count	numeric	这个表被手动清理的次数(不计算 VACUUM FULL)。
autovacuum_count	numeric	这个表被 autovacuum 清理的次数。
analyze_count	numeric	这个表被手动分析的次数。

名称	类型	描述
autoanalyze_count	numeric	这个表被 autovacuum 守护进程分析的次数。

26.2.5.15 GLOBAL_STAT_ALL_TABLES

得到各节点数据中每个表的一行（包括 TOAST 表）的统计信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	表的 OID。
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	bigint	此表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。
idx_scan	bigint	此表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。

名称	类型	描述
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。
n_live_tup	bigint	估计活跃行数。
n_dead_tup	bigint	估计死行数。
last_vacuum	timestamp with time zone	最后一次此表是手动清理的（不计算 VACUUM FULL）的时间。
last_autovacuum	timestamp with time zone	上次被 autovacuum 守护进程清理的时间。
last_analyze	timestamp with time zone	上次手动分析这个表的时间。
last_autoanalyze	timestamp with time zone	上次被 autovacuum 守护进程分析时间。
vacuum_count	bigint	这个表被手动清理的次数（不计算 VACUUM FULL）。
autovacuum_count	bigint	这个表被 autovacuum 清理的次数。
analyze_count	bigint	这个表被手动分析的次数。

名称	类型	描述
autoanalyze_count	bigint	这个表被 autovacuum 守护进程分析的次数。

26.2.5.16 STAT_ALL_INDEXES

将包含本节点数据库中的每个索引行，显示访问特定索引的统计。

名称	类型	描述
relid	oid	这个索引的表的 OID。
indexrelid	oid	索引的 OID。
schemaname	name	索引中模式名。
relname	name	索引的表名。
indexrelname	name	索引名。
idx_scan	bigint	索引上开始的索引扫描数。
idx_tup_read	bigint	通过索引上扫描返回的索引项数。
idx_tup_fetch	bigint	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.17 SUMMARY_STAT_ALL_INDEXES

将包含 GBase 8s 内汇聚数据库中的每个索引行，显示访问特定索引的统计。

名称	类型	描述

名称	类型	描述
schemaname	name	索引中模式名。
relname	name	索引的表名。
indexrelname	name	索引名。
idx_scan	numeric	索引上开始的索引扫描数。
idx_tup_read	numeric	通过索引上扫描返回的索引项数。
idx_tup_fetch	numeric	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.18 GLOBAL_STAT_ALL_INDEXES

将包含各节点数据库中的每个索引行，显示访问特定索引的统计。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	这个索引的表的 OID。
indexrelid	oid	索引的 OID。
schemaname	name	索引中模式名。
relname	name	索引的表名。

名称	类型	描述
indexrelname	name	索引名。
idx_scan	bigint	索引上开始的索引扫描数。
idx_tup_read	bigint	通过索引上扫描返回的索引项数。
idx_tup_fetch	bigint	通过使用索引的简单索引扫描抓取的活表行数。

26.2.5.19 STAT_DATABASE

视图将包含本节点中每个数据库的统计信息。

名称	类型	描述
datid	oid	数据库的 OID。
datname	name	此数据库的名称。
numbackends	integer	当前连接到该数据库的后端数。这是在返回一个反映目前状态值的视图中唯一的列；自上次重置所有其他列返回累积值。
xact_commit	bigint	此数据库中已经提交的事务数。
xact_rollback	bigint	此数据库中已经回滚的事务数。
blks_read	bigint	在这个数据库中读取的磁盘块的数量。

名称	类型	描述
blks_hit	bigint	高速缓存中已经发现的磁盘块的次数，这样读取是不必要的（这只包括 PostgreSQL 缓冲区高速缓存，没有操作系统的文件系统缓存）。
tup_returned	bigint	通过数据库查询返回的行数。
tup_fetched	bigint	通过数据库查询抓取的行数。
tup_inserted	bigint	通过数据库查询插入的行数。
tup_updated	bigint	通过数据库查询更新的行数。
tup_deleted	bigint	通过数据库查询删除的行数。
conflicts	bigint	由于数据库恢复冲突取消的查询数量（只在备用服务器发生的冲突）。请参见 STAT_DATABASE_CONFLICTS 获取更多信息。
temp_files	bigint	通过数据库查询创建的临时文件数量。计算所有临时文件，不论为什么创建临时文件（比如排序或者哈希），而且不管 log_temp_files 设置。
temp_bytes	bigint	通过数据库查询写入临时文件的数据总量。计算所有临时文件，不论为什么创建临时文件，而且不管 log_temp_files 设置。
deadlocks	bigint	在该数据库中检索的死锁数。

名称	类型	描述
blk_read_time	double precision	通过数据库后端读取数据文件块花费的时间，以毫秒计算。
blk_write_time	double precision	通过数据库后端写入数据文件块花费的时间，以毫秒计算。
stats_reset	timestamp with time zone	重置当前状态统计的时间。

26.2.5.20 SUMMARY_STAT_DATABASE

视图将包含数据库内汇聚的每个数据库的每一行，显示数据库统计。

名称	类型	描述
datname	name	这个数据库的名称。
numbackends	bigint	当前连接到该数据库的后端数。这是在返回一个反映目前状态值的视图中唯一的列；自上次重置所有其他列返回累积值。
xact_commit	numeric	此数据库中已经提交的事务数。
xact_rollback	numeric	此数据库中已经回滚的事务数。
blks_read	numeric	在这个数据库中读取的磁盘块的数量。
blks_hit	numeric	高速缓存中已经发现的磁盘块的次数，这样读取是不必要的（这只包括 GBase 8s 缓冲区高速

名称	类型	描述
		缓存，没有操作系统的文件系统缓存）。
tup_returned	numeric	通过数据库查询返回的行数。
tup_fetched	numeric	通过数据库查询抓取的行数。
tup_inserted	bigint	通过数据库查询插入的行数。
tup_updated	bigint	通过数据库查询更新的行数。
tup_deleted	bigint	通过数据库查询删除的行数。
conflicts	bigint	由于数据库恢复冲突取消的查询数量（只在备用服务器发生的冲突）。请参见 STAT_DATABASE_CONFLICTS 获取更多信息。
temp_files	numeric	通过数据库查询创建的临时文件数量。计算所有临时文件，不论为什么创建临时文件（比如排序或者哈希），而且不管 log_temp_files 设置。
temp_bytes	numeric	通过数据库查询写入临时文件的数据总量。计算所有临时文件，不论为什么创建临时文件，而且不管 log_temp_files 设置。
deadlocks	bigint	在该数据库中检索的死锁数。
blk_read_time	double precision	通过数据库后端读取数据文件块花费的时间，

名称	类型	描述
		以毫秒计算。
blk_write_time	double precision	通过数据库后端写入数据文件块花费的时间，以毫秒计算。
stats_reset	timestamp with time zone	重置当前状态统计的时间。

26.2.5.21 GLOBAL_STAT_DATABASE

视图将包含 GBase 8s 中各节点的每个数据库的每一行，显示数据库统计。

名称	类型	描述
node_name	name	数据库进程名称。
datid	oid	数据库的 OID。
datname	name	这个数据库的名称。
numbackends	integer	当前连接到该数据库的后端数。这是在返回一个反映目前状态值的视图中唯一的列；自上次重置所有其他列返回累积值。
xact_commit	bigint	此数据库中已经提交的事务数。
xact_rollback	bigint	此数据库中已经回滚的事务数。
blks_read	bigint	在这个数据库中读取的磁盘块的数量。

名称	类型	描述
blks_hit	bigint	高速缓存中已经发现的磁盘块的次数，这样读取是不必要的（这只包括数据库内核缓冲区高速缓存，没有操作系统的文件系统缓存）。
tup_returned	bigint	通过数据库查询返回的行数。
tup_fetched	bigint	通过数据库查询抓取的行数。
tup_inserted	bigint	通过数据库查询插入的行数。
tup_updated	bigint	通过数据库查询更新的行数。
tup_deleted	bigint	通过数据库查询删除的行数。
conflicts	bigint	由于数据库恢复冲突取消的查询数量（只在备用服务器发生的冲突）。请参见 STAT_DATABASE_CONFLICTS 获取更多信息。
temp_files	bigint	通过数据库查询创建的临时文件数量。计算所有临时文件，不论为什么创建临时文件（比如排序或者哈希），而且不管 log_temp_files 设置。
temp_bytes	bigint	通过数据库查询写入临时文件的数据总量。计算所有临时文件，不论为什么创建临时文件，而且不管 log_temp_files 设置。
deadlocks	bigint	在该数据库中检索的死锁数。

名称	类型	描述
blk_read_time	double precision	通过数据库后端读取数据文件块花费的时间，以毫秒计算。
blk_write_time	double precision	通过数据库后端写入数据文件块花费的时间，以毫秒计算。
stats_reset	timestamp with time zone	重置当前状态统计的时间。

26.2.5.22 STAT_DATABASE_CONFLICTS

显示当前节点数据库冲突状态的统计信息。

名称	类型	描述
datid	oid	数据库标识。
datname	name	数据库名称。
confl_tablespace	bigint	冲突的表空间的数目。
confl_lock	bigint	冲突的锁数目。
confl_snapshot	bigint	冲突的快照数目。
confl_bufferpin	bigint	冲突的缓冲区数目。

名称	类型	描述
confl_deadlock	bigint	冲突的死锁数目。

26.2.5.23 SUMMARY_STAT_DATABASE_CONFLICTS

显示 GBase 8s 内汇聚的数据库冲突状态的统计信息。

名称	类型	描述
datname	name	数据库名称。
confl_tablespace	bigint	冲突的表空间的数目。
confl_lock	bigint	冲突的锁数目。
confl_snapshot	bigint	冲突的快照数目。
confl_bufferpin	bigint	冲突的缓冲区数目。
confl_deadlock	bigint	冲突的死锁数目。

26.2.5.24 GLOBAL_STAT_DATABASE_CONFLICTS

显示每个节点的数据库冲突状态的统计信息。

名称	类型	描述
node_name	name	数据库进程名称。
datid	oid	数据库标识。

名称	类型	描述
datname	name	数据库名称。
confl_tablespace	bigint	冲突的表空间的数目。
confl_lock	bigint	冲突的锁数目。
confl_snapshot	bigint	冲突的快照数目。
confl_bufferpin	bigint	冲突的缓冲区数目。
confl_deadlock	bigint	冲突的死锁数目。

26.2.5.25 STAT_XACT_ALL_TABLES

显示命名空间中所有普通表和 toast 表的事务状态信息。

名称	类型	描述
relid	oid	表的 OID。
schemaname	name	该表的模式名。
relname	name	表名。
seq_scan	bigint	该表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。

名称	类型	描述
idx_scan	bigint	该表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.26 SUMMARY_STAT_XACT_ALL_TABLES

显示 GBase 8s 内汇聚的命名空间中所有普通表和 toast 表的事务状态信息。

名称	类型	描述
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	numeric	此表发起的顺序扫描数。
seq_tup_read	numeric	顺序扫描抓取的活跃行数。
idx_scan	numeric	此表发起的索引扫描数。

名称	类型	描述
idx_tup_fetch	numeric	索引扫描抓取的活跃行数。
n_tup_ins	numeric	插入行数。
n_tup_upd	numeric	更新行数。
n_tup_del	numeric	删除行数。
n_tup_hot_upd	numeric	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.27 GLOBAL_STAT_XACT_ALL_TABLES

显示各节点的命名空间中所有普通表和 toast 表的事务状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	表的 OID。
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	bigint	此表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。

名称	类型	描述
idx_scan	bigint	此表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.28 STAT_XACT_SYS_TABLES

显示当前节点命名空间中系统表的事务状态信息。

名称	类型	描述
relid	oid	表的 OID。
schemaname	name	该表的模式名。
relname	name	表名。
seq_scan	bigint	该表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。

名称	类型	描述
idx_scan	bigint	该表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.29 SUMMARY_STAT_XACT_SYS_TABLES

显示 GBase 8s 内汇聚的命名空间中系统表的事务状态信息。

名称	类型	描述
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	numeric	此表发起的顺序扫描数。
seq_tup_read	numeric	顺序扫描抓取的活跃行数。
idx_scan	numeric	此表发起的索引扫描数。

名称	类型	描述
idx_tup_fetch	numeric	索引扫描抓取的活跃行数。
n_tup_ins	numeric	插入行数。
n_tup_upd	numeric	更新行数。
n_tup_del	numeric	删除行数。
n_tup_hot_upd	numeric	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.30 GLOBAL_STAT_XACT_SYS_TABLES

显示各节点命名空间中系统表的事务状态信息。

名称	类型	描述
node_name	name	节点名称。
relid	oid	表的 OID。
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	bigint	此表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。

名称	类型	描述
idx_scan	bigint	此表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.31 STAT_XACT_USER_TABLES

显示当前节点命名空间中用户表的事务状态信息。

名称	类型	描述
relid	oid	表的 OID。
schemaname	name	该表的模式名。
relname	name	表名。
seq_scan	bigint	该表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。

名称	类型	描述
idx_scan	bigint	该表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.32 SUMMARY_STAT_XACT_USER_TABLES

显示数据库内汇聚的命名空间中用户表的事务状态信息。

名称	类型	描述
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	numeric	此表发起的顺序扫描数。
seq_tup_read	numeric	顺序扫描抓取的活跃行数。
idx_scan	numeric	此表发起的索引扫描数。

名称	类型	描述
idx_tup_fetch	numeric	索引扫描抓取的活跃行数。
n_tup_ins	numeric	插入行数。
n_tup_upd	numeric	更新行数。
n_tup_del	numeric	删除行数。
n_tup_hot_upd	numeric	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.33 GLOBAL_STAT_XACT_USER_TABLES

显示各节点命名空间中用户表的事务状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	表的 OID。
schemaname	name	此表的模式名。
relname	name	表名。
seq_scan	bigint	此表发起的顺序扫描数。
seq_tup_read	bigint	顺序扫描抓取的活跃行数。

名称	类型	描述
idx_scan	bigint	此表发起的索引扫描数。
idx_tup_fetch	bigint	索引扫描抓取的活跃行数。
n_tup_ins	bigint	插入行数。
n_tup_upd	bigint	更新行数。
n_tup_del	bigint	删除行数。
n_tup_hot_upd	bigint	HOT 更新行数（比如没有更新所需的单独索引）。

26.2.5.34 STAT_XACT_USER_FUNCTIONS

视图包含当前节点本事务内函数执行的统计信息。

名称	类型	描述
funcid	oid	函数标识。
schemaname	name	模式的名称。
funcname	name	函数名称。
calls	bigint	函数被调用的次数。
total_time	double precision	函数的总执行时长。

名称	类型	描述
self_time	double precision	当前线程调用函数的总的时长。

26.2.5.35 SUMMARY_STAT_XACT_USER_FUNCTIONS

视图包含 GBase 8s 内汇聚的本事务内函数执行的统计信息。

名称	类型	描述
schemaname	name	模式的名称。
funcname	name	函数名称。
calls	numeric	函数被调用的次数。
total_time	double precision	函数的总执行时长。
self_time	double precision	当前线程调用函数的总的时长。

26.2.5.36 GLOBAL_STAT_XACT_USER_FUNCTIONS

视图包含各节点本事务内函数执行的统计信息。

名称	类型	描述
node_name	name	节点名称。
funcid	oid	函数标识。
schemaname	name	模式的名称。

名称	类型	描述
funcname	name	函数名称。
calls	bigint	函数被调用的次数。
total_time	double precision	此函数及其调用的所有其他函数所花费的总时间。
self_time	double precision	在此函数本身中花费的总时间（不包括它调用的其他函数）。

26.2.5.37 STAT_BAD_BLOCK

获得当前节点表、索引等文件的读取失败信息。

名称	类型	描述
nodename	text	数据库进程名称。
databaseid	integer	database 的 oid。
tablespaceid	integer	tablespace 的 oid。
relfilenode	integer	relation 的 file node。
bucketid	smallint	一致性 hash bucket ID。
forknum	integer	fork 编号。
error_count	integer	error 的数量。

名称	类型	描述
first_time	timestamp with time zone	坏块第一次出现的时间。
last_time	timestamp with time zone	坏块最后出现的时间。

26.2.5.38 SUMMARY_STAT_BAD_BLOCK

获得 GBase 8s 内汇聚的表、索引等文件的读取失败信息。

名称	类型	描述
databaseid	integer	database 的 oid。
tablespaceid	integer	tablespace 的 oid。
relfilenode	integer	relation 的 file node。
forknum	bigint	fork 编号。
error_count	bigint	error 的数量。
first_time	timestamp with time zone	坏块第一次出现的时间。
last_time	timestamp with time zone	坏块最后出现的时间。

26.2.5.39 GLOBAL_STAT_BAD_BLOCK

获得各节点的表、索引等文件的读取失败信息。

名称	类型	描述
node_name	text	数据库进程名称。
databaseid	integer	database 的 oid。
tablespaceid	integer	tablespace 的 oid。
relfilenode	integer	relation 的 file node。
forknum	integer	fork 编号。
error_count	integer	error 的数量。
first_time	timestamp with time zone	坏块第一次出现的时间。
last_time	timestamp with time zone	坏块最后出现的时间。

26.2.5.40 STAT_USER_FUNCTIONS

STAT_USER_FUNCTIONS 视图显示命名空间中用户自定义函数（函数语言为非内部语言）的状态信息。

名称	类型	描述

名称	类型	描述
funcid	oid	函数标识。
schemaname	name	schema 的名称。
funcname	name	用户自定义函数的名称。
calls	bigint	函数被调用的次数。
total_time	double precision	调用此函数花费的总时间，包含调用其它函数的时间（单位：毫秒）。
self_time	double precision	调用此函数自己花费的时间，不包含调用其它函数的时间（单位：毫秒）。

26.2.5.41 SUMMARY_STAT_USER_FUNCTIONS

SUMMARY_STAT_USER_FUNCTIONS 用来统计所数据库节点用户自定义函数的相关统计信息。

名称	类型	描述
schemaname	name	schema 的名称。
funcname	name	用户 function 的名称。
calls	numeric	总调用次数。
total_time	double precision	调用此 function 的总时间花费，包含调用其它 function 的时间（单位：毫秒）。

名称	类型	描述
self_time	double precision	调用此 function 自己时间的花费，不包含调用其它 function 的时间（单位：毫秒）。

26.2.5.42 GLOBAL_STAT_USER_FUNCTIONS

提供 GBase 8s 中各个节点的用户所创建的函数的状态的统计信息。

名称	类型	描述
node_name	name	数据库进程名称。
funcid	oid	函数的 id。
schemaname	name	此函数所在模式的名称。
funcname	name	函数名称。
calls	bigint	该函数被调用的次数。
total_time	double precision	此函数及其调用的所有其他函数所花费的总时间(以毫秒为单位）。
self_time	double precision	在此函数本身中花费的总时间（不包括它调用的其他函数），以毫秒为单位。

26.2.6 Workload

26.2.6.1 WORKLOAD_SQL_COUNT

显示当前节点 workload 上的 SQL 数量分布。普通用户只可以看到自己在 workload 上的 SQL 分布；初始用户可以看到总的 workload 的负载情况。

名称	类型	描述
workload	name	负载名称。
select_count	bigint	select 数量。
update_count	bigint	update 数量。
insert_count	bigint	insert 数量。
delete_count	bigint	delete 数量。
ddl_count	bigint	ddl 数量。
dml_count	bigint	dml 数量。
dcl_count	bigint	dcl 数量。

26.2.6.2 SUMMARY_WORKLOAD_SQL_COUNT

显示 GBase 8s 内各数据库主节点的 workload 上的 SQL 数量分布。

名称	类型	描述
node_name	name	数据库进程名称。
workload	name	负载名称。
select_count	bigint	select 数量。

名称	类型	描述
update_count	bigint	update 数量。
insert_count	bigint	insert 数量。
delete_count	bigint	delete 数量。
ddl_count	bigint	ddl 数量。
dml_count	bigint	dml 数量。
dcl_count	bigint	dcl 数量。

26.2.6.3 WORKLOAD_TRANSACTION

当前节点上负载的事务信息。

名称	类型	描述
workload	name	负载的名称。
commit_counter	bigint	用户事务 commit 数量。
rollback_counter	bigint	用户事务 rollback 数量。
resp_min	bigint	用户事务最小响应时间（单位：微秒）。
resp_max	bigint	用户事务最大响应时间（单位：微秒）。

名称	类型	描述
resp_avg	bigint	用户事务平均响应时间（单位：微秒）。
resp_total	bigint	用户事务总响应时间（单位：微秒）。
bg_commit_counter	bigint	后台事务 commit 数量。
bg_rollback_counter	bigint	后台事务 rollback 数量。
bg_resp_min	bigint	后台事务最小响应时间（单位：微秒）。
bg_resp_max	bigint	后台事务最大响应时间（单位：微秒）。
bg_resp_avg	bigint	后台事务平均响应时间（单位：微秒）。
bg_resp_total	bigint	后台事务总响应时间（单位：微秒）。

26.2.6.4 SUMMARY_WORKLOAD_TRANSACTION

显示 GBase 8s 内汇聚的负载事务信息。

名称	类型	描述
workload	name	负载的名称。
commit_counter	numeric	用户事务 commit 数量。
rollback_counter	numeric	用户事务 rollback 数量。

名称	类型	描述
resp_min	bigint	用户事务最小响应时间（单位：微秒）。
resp_max	bigint	用户事务最大响应时间（单位：微秒）。
resp_avg	bigint	用户事务平均响应时间（单位：微秒）。
resp_total	numeric	用户事务总响应时间（单位：微秒）。
bg_commit_counter	numeric	后台事务 commit 数量。
bg_rollback_counter	numeric	后台事务 rollback 数量。
bg_resp_min	bigint	后台事务最小响应时间（单位：微秒）。
bg_resp_max	bigint	后台事务最大响应时间（单位：微秒）。
bg_resp_avg	bigint	后台事务平均响应时间（单位：微秒）。
bg_resp_total	numeric	后台事务总响应时间（单位：微秒）。

26.2.6.5 GLOBAL_WORKLOAD_TRANSACTION

显示各节点上的 workload 的负载信息。

名称	类型	描述
node_name	name	数据库进程名称。

名称	类型	描述
workload	name	负载的名称。
commit_counter	bigint	用户事务 commit 数量。
rollback_counter	bigint	用户事务 rollback 数量。
resp_min	bigint	用户事务最小响应时间（单位：微秒）。
resp_max	bigint	用户事务最大响应时间（单位：微秒）。
resp_avg	bigint	用户事务平均响应时间（单位：微秒）。
resp_total	bigint	用户事务总响应时间（单位：微秒）。
bg_commit_counter	bigint	后台事务 commit 数量。
bg_rollback_counter	bigint	后台事务 rollback 数量。
bg_resp_min	bigint	后台事务最小响应时间（单位：微秒）。
bg_resp_max	bigint	后台事务最大响应时间（单位：微秒）。
bg_resp_avg	bigint	后台事务平均响应时间（单位：微秒）。
bg_resp_total	bigint	后台事务总响应时间（单位：微秒）。

26.2.6.6 WORKLOAD_SQL_ELAPSE_TIME

WORKLOAD_SQL_ELAPSE_TIME 用来统计 workload（业务负载）上的 SUID 信息。

名称	类型	描述
workload	name	workload（业务负载）名称。
total_select_elapse	bigint	总 select 的时间花费（单位：微秒）。
max_select_elapse	bigint	最大 select 的时间花费（单位：微秒）。
min_select_elapse	bigint	最小 select 的时间花费（单位：微秒）。
avg_select_elapse	bigint	平均 select 的时间花费（单位：微秒）。
total_update_elapse	bigint	总 update 的时间花费（单位：微秒）。
max_update_elapse	bigint	最大 update 的时间花费（单位：微秒）。
min_update_elapse	bigint	最小 update 的时间花费（单位：微秒）。
avg_update_elapse	bigint	平均 update 的时间花费（单位：微秒）。
total_insert_elapse	bigint	总 insert 的时间花费（单位：微秒）。
max_insert_elapse	bigint	最大 insert 的时间花费（单位：微秒）。
min_insert_elapse	bigint	最小 insert 的时间花费（单位：微秒）。

名称	类型	描述
avg_insert_elapse	bigint	平均 insert 的时间花费（单位：微秒）。
total_delete_elapse	bigint	总 delete 的时间花费（单位：微秒）。
max_delete_elapse	bigint	最大 delete 的时间花费（单位：微秒）。
min_delete_elapse	bigint	最小 delete 的时间花费（单位：微秒）。
avg_delete_elapse	bigint	平均 delete 的时间花费（单位：微秒）。

26.2.6.7 SUMMARY_WORKLOAD_SQL_ELAPSE_TIME

SUMMARY_WORKLOAD_SQL_ELAPSE_TIME 用来统计数据库主节点上 workload(业务）负载的 SUID 信息。

名称	类型	描述
node_name	name	数据库进程名称。
workload	name	workload（业务负载）名称。
total_select_elapse	bigint	总 select 的时间花费（单位：微秒）。
max_select_elapse	bigint	最大 select 的时间花费（单位：微秒）。
min_select_elapse	bigint	最小 select 的时间花费（单位：微秒）。

名称	类型	描述
avg_select_elapse	bigint	平均 select 的时间花费（单位：微秒）。
total_update_elapse	bigint	总 update 的时间花费（单位：微秒）。
max_update_elapse	bigint	最大 update 的时间花费（单位：微秒）。
min_update_elapse	bigint	最小 update 的时间花费（单位：微秒）。
avg_update_elapse	bigint	平均 update 的时间花费（单位：微秒）。
total_insert_elapse	bigint	总 insert 的时间花费（单位：微秒）。
max_insert_elapse	bigint	最大 insert 的时间花费（单位：微秒）。
min_insert_elapse	bigint	最小 insert 的时间花费（单位：微秒）。
avg_insert_elapse	bigint	平均 insert 的时间花费（单位：微秒）。
total_delete_elapse	bigint	总 delete 的时间花费（单位：微秒）。
max_delete_elapse	bigint	最大 delete 的时间花费（单位：微秒）。
min_delete_elapse	bigint	最小 delete 的时间花费（单位：微秒）。
avg_delete_elapse	bigint	平均 delete 的时间花费（单位：微秒）。

26.2.6.8 USER_TRANSACTION

USER_TRANSACTION 用来统计用户执行的事务信息。monadmin 用户能看到所有用户执行事务的信息，普通用户只能查询到自己执行的事务信息。

名称	类型	描述
username	name	用户的名称。
commit_counter	bigint	用户事务 commit 数量。
rollback_counter	bigint	用户事务 rollback 数量。
resp_min	bigint	用户事务最小响应时间（单位：微秒）。
resp_max	bigint	用户事务最大响应时间（单位：微秒）。
resp_avg	bigint	用户事务平均响应时间（单位：微秒）。
resp_total	bigint	用户事务总响应时间（单位：微秒）。
bg_commit_counter	bigint	后台事务 commit 数量。
bg_rollback_counter	bigint	后台事务 rollback 数量。
bg_resp_min	bigint	后台事务最小响应时间（单位：微秒）。
bg_resp_max	bigint	后台事务最大响应时间（单位：微秒）。
bg_resp_avg	bigint	后台事务平均响应时间（单位：微秒）。

名称	类型	描述
bg_resp_total	bigint	后台事务总响应时间（单位：微秒）。

26.2.6.9 GLOBAL_USER_TRANSACTION

GLOBAL_USER_TRANSACTION 用来统计全局用户执行的事务信息。

名称	类型	描述
node_name	name	节点名称。
username	name	用户的名称。
commit_counter	bigint	用户事务 commit 数量。
rollback_counter	bigint	用户事务 rollback 数量。
resp_min	bigint	用户事务最小响应时间（单位：微秒）。
resp_max	bigint	用户事务最大响应时间（单位：微秒）。
resp_avg	bigint	用户事务平均响应时间（单位：微秒）。
resp_total	bigint	用户事务总响应时间（单位：微秒）。
bg_commit_counter	bigint	后台事务 commit 数量。
bg_rollback_counter	bigint	后台事务 rollback 数量。

名称	类型	描述
bg_resp_min	bigint	后台事务最小响应时间（单位：微秒）。
bg_resp_max	bigint	后台事务最大响应时间（单位：微秒）。
bg_resp_avg	bigint	后台事务平均响应时间（单位：微秒）。
bg_resp_total	bigint	后台事务总响应时间（单位：微秒）。

26.2.7 Session/Thread

26.2.7.1 SESSION_STAT

当前节点以会话线程或 AutoVacuum 线程为单位，统计会话状态信息。

名称	类型	描述
sessid	text	线程启动时间+线程标识。
statid	integer	统计编号。
statname	text	统计会话名称。
statunit	text	统计会话单位。
value	bigint	统计会话值。

26.2.7.2 GLOBAL_SESSION_STAT

各节点上以会话线程或 AutoVacuum 线程为单位，统计会话状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
sessid	text	线程启动时间+线程标识。
statid	integer	统计编号。
statname	text	统计会话名称。
statunit	text	统计会话单位。
value	bigint	统计会话值。

26.2.7.3 SESSION_TIME

用于统计当前节点会话线程的运行时间信息，及各执行阶段所消耗时间。

名称	类型	描述
sessid	text	线程启动时间+线程标识。
stat_id	integer	统计编号。
stat_name	text	会话类型名称。
value	bigint	会话值。

26.2.7.4 GLOBAL_SESSION_TIME

用于统计各节点会话线程的运行时间信息，及各执行阶段所消耗时间。

名称	类型	描述
node_name	name	数据库进程名称。
sessid	text	线程启动时间+线程标识。
stat_id	integer	统计编号。
stat_name	text	会话类型名称。
value	bigint	会话值。

26.2.7.5 SESSION_MEMORY

统计 Session 级别的内存使用情况，包含执行作业在数据节点上 GBase 8s 线程和 Stream 线程分配的所有内存，单位为 MB。

名称	类型	描述
sessid	text	线程启动时间+线程标识。
init_mem	integer	当前正在执行作业进入执行器前已分配的内存。
used_mem	integer	当前正在执行作业已分配的内存。
peak_mem	integer	当前正在执行作业已分配的内存峰值。

26.2.7.6 GLOBAL_SESSION_MEMORY

统计各节点的 Session 级别的内存使用情况，包含执行作业在数据节点上 GBase 8s 线程和 Stream 线程分配的所有内存，单位为 MB。

名称	类型	描述
node_name	name	数据库进程名称。
sessid	text	线程启动时间+线程标识。
init_mem	integer	当前正在执行作业进入执行器前已分配的内存。
used_mem	integer	当前正在执行作业已分配的内存。
peak_mem	integer	当前正在执行作业已分配的内存峰值。

26.2.7.7 SESSION_MEMORY_DETAIL

统计线程的内存使用情况，以 MemoryContext 节点来统计。

名称	类型	描述
sessid	text	线程启动时间+线程标识。
sesstype	text	线程名称。
contextname	text	内存上下文名称。
level	smallint	内存上下文的重要级别。
parent	text	父级内存上下文名称。
totalsize	bigint	总申请内存大小（单位：字节）。

名称	类型	描述
freesize	bigint	空闲内存大小（单位：字节）。
usedsize	bigint	使用内存大小（单位：字节）。

26.2.7.8 GLOBAL_SESSION_MEMORY_DETAIL

统计各节点的线程的内存使用情况，以 MemoryContext 节点来统计。

名称	类型	描述
node_name	name	数据库进程名称。
sessid	text	线程启动时间+线程标识。
sesstype	text	线程名称。
contextname	text	内存上下文名称。
level	smallint	内存上下文的重要级别。
parent	text	父级内存上下文名称。
totalsize	bigint	总申请内存大小（单位：字节）。
freesize	bigint	空闲内存大小（单位：字节）。
usedsize	bigint	使用内存大小（单位：字节）。

26.2.7.9 SESSION_STAT_ACTIVITY

显示当前节点上正在运行的线程相关的信息。

名称	类型	描述
datid	oid	用户会话在后台连接到的数据库 OID。
datname	name	用户会话在后台连接到的数据库名称。
pid	bigint	后台线程 ID。
usesysid	oid	登录该后台的用户 OID。
usename	name	登录该后台的用户名。
application_name	text	连接到该后台的应用名。
client_addr	inet	连接到该后台的客户端的 IP 地址。如果此字段是 null，它表明通过服务器机器上 UNIX 套接字连接客户端或者这是内部进程，如 autovacuum。
client_hostname	text	客户端的主机名，这个字段是通过 client_addr 的反向 DNS 查找得到。这个字段只有在启动 log_hostname 且使用 IP 连接时才非空。
client_port	integer	客户端用于与后台通讯的 TCP 端口号，如果使用 Unix 套接字，则为-1。
backend_start	timestampwit	该过程开始的时间，即当客户端连接服务器时间。

名称	类型	描述
	h time zone	
xact_start	timestampwit h time zone	启动当前事务的时间，如果没有事务是活跃的，则为 null。如果当前查询是首个事务，则这列等同于 query_start 列。
query_start	timestampwit h time zone	开始当前活跃查询的时间，如果 state 的值不是 active，则这个值是上一个查询的开始时间。
state_change	timestampwit h time zone	上次状态改变的时间。
waiting	boolean	如果后台当前正等待锁则为 true。
enqueue	text	该字段不支持。
state	text	该后台当前总体状态。可能值是： active：后台正在执行一个查询。 idle：后台正在等待一个新的客户端命令。 idle in transaction：后台在事务中，但是目前无法执行查询。 idle in transaction (aborted)：这个状态除说明事务中有某个语句导致了错误外，类似于 idle in transaction fastpath function call：后台正在执行一个 fast-path 函数。 disabled：如果后台禁用 track_activities，则报告这个状态。 说明：

名称	类型	描述
		普通用户只能查看到自己帐户所对应的会话状态。即其他帐户的 state 信息为空。例如以 judy 用户连接数据库后，在 pg_stat_activity 中查看到的普通用户 joe 及初始用户 gbase 的 stat 信息为空。 postgres=# SELECT datname, usename, usesysid,state,pid FROM pg_stat_activity; datname | usename | usesysid | state | pid ———-+———+———-+———+——— postgres | gbase | 10 | |139968752121616 postgres | gbase | 10 | |139968903116560 db_tpcds | judy | 16398 | active |139968391403280 postgres | gbase | 10 | |139968643069712 postgres | gbase | 10 | |139968680818448 postgres | joe | 16390 | |139968563377936 (6 rows)
resource_pool	name	用户使用的资源池。

名称	类型	描述
query_id	bigint	查询语句的 ID。
query	text	该后台的最新查询。如果 state 状态是 active（活跃的），此字段显示当前正在执行的查询。所有其他情况表示上一个查询。
unique_sql_id	bigint	语句的 unique sql id。
trace_id	text	驱动传入的 trace id，与应用的一次请求相关联。

26.2.7.10 GLOBAL_SESSION_STAT_ACTIVITY

显示 GBase 8s 内各节点上正在运行的线程相关的信息。

名称	类型	描述
coorname	text	数据库进程名称。
datid	oid	用户会话在后台连接到的数据库 OID。
datname	text	用户会话在后台连接到的数据库名称。
pid	bigint	后台线程 ID。
usesysid	oid	登录该后台的用户 OID。
usename	text	登录该后台的用户名。

名称	类型	描述
application_name	text	连接到该后台的应用名。
client_addr	inet	连接到该后台的客户端的 IP 地址。如果此字段是 null，它表明通过服务器机器上 UNIX 套接字连接客户端或者这是内部进程，如 autovacuum。
client_hostname	text	客户端的主机名，这个字段是通过 client_addr 的反向 DNS 查找得到。这个字段只有在启动 log_hostname 且使用 IP 连接时才非空。
client_port	integer	客户端用于与后台通讯的 TCP 端口号，如果使用 Unix 套接字，则为-1。
backend_start	timestampwith time zone	该过程开始的时间，即当客户端连接服务器时间。
xact_start	timestampwith time zone	启动当前事务的时间，如果没有事务是活跃的，则为 null。如果当前查询是首个事务，则这列等同于 query_start 列。
query_start	timestampwith time zone	开始当前活跃查询的时间，如果 state 的值不是 active，则这个值是上一个查询的开始时间。
state_change	timestampwith time zone	上次状态改变的时间。
waiting	boolean	如果后台当前正等待锁则为 true。

名称	类型	描述
enqueue	text	该字段不支持 。
state	text	该后台当前总体状态。可能值是： active：后台正在执行一个查询。 idle：后台正在等待一个新的客户端命令。 idle in transaction：后台在事务中，但是目前无法执行查询。 idle in transaction (aborted)：这个状态除说明事务中有某个语句导致了错误外，类似于 idle in transaction fastpath function call：后台正在执行一个 fast-path 函数。 disabled：如果后台禁用 track_activities，则报告这个状态。 说明： 普通用户只能查看到自己帐户所对应的会话状态。即其他帐户的 state 信息为空。例如以 judy 用户连接数据库后，在 pg_stat_activity 中查看到的普通用户 joe 及初始用户 gbase 的 state 信息为空。 postgres=# SELECT datname, usename, usesysid,state,pid FROM pg_stat_activity; datname | usename | usesysid | state | pid ————-+————+————-+————+————————postgre

名称	类型	描述
		s | gbase | 10 | |139968752121616 postgres | gbase | 10 | |139968903116560 db_tpcds | judy | 16398 | active |139968391403280 postgres | gbase | 10 | |139968643069712 postgres | gbase | 10 | |139968680818448 postgres | joe | 16390 | |139968563377936 (6 rows)
resource_pool	name	用户使用的资源池。
query_id	bigint	查询语句的 ID。
query	text	该后台的最新查询。如果 state 状态是 active（活跃的），此字段显示当前正在执行的查询。所有其他情况表示上一个查询。
unique_sql_id	bigint	语句的 unique sql id。
trace_id	text	驱动传入的 trace id，与应用的一次请求相关联。

26.2.7.11 THREAD_WAIT_STATUS

通过该视图可以检测当前实例中工作线程（backend thread）以及辅助线程（auxiliary thread）的阻塞等待情况，具体事件信息请参见表 15-15-2、PG_THREAD_WAIT_STATUS 中的等待状态列表、轻量级锁等待事件列表、IO 等待事件列表和事务锁等待事件列表。

名称	类型	描述
node_name	text	数据库进程名称。
db_name	text	数据库名称。
thread_name	text	线程名称。
query_id	bigint	查询 ID，对应 debug_query_id。
tid	bigint	当前线程的线程号。
sessionid	bigint	session 的 ID。
lwtid	integer	当前线程的轻量级线程号。
psessionid	bigint	streaming 线程的父线程。
tlevel	integer	streaming 线程的层级。
smpid	integer	并行线程的 ID。
wait_status	text	当前线程的等待状态。等待状态的详细信息请参见 PG_THREAD_WAIT_STATUS 中等待状态列表。

名称	类型	描述
wait_event	text	如果 wait_status 是 acquire lock、acquire lwlock、wait io 三种类型，此列描述具体的锁、轻量级锁、IO 的信息。否则为空。

26.2.7.12 GLOBAL_THREAD_WAIT_STATUS

通过该视图可以检测所有节点上工作线程（backend thread）以及辅助线程（auxiliary thread）的阻塞等待情况。具体事件信息请参见 PG_THREAD_WAIT_STATUS 中的等待状态列表、轻量级锁等待事件列表、IO 等待事件列表和事务锁等待事件列表

通过 GLOBAL_THREAD_WAIT_STATUS 视图，可以查看 GBase 8s 全局各个节点上所有 SQL 语句产生的线程之间的调用层次关系，以及各个线程的阻塞等待状态，从而更容易定位 hang 以及类似现象的原因。

GLOBAL_THREAD_WAIT_STATUS 视图和 THREAD_WAIT_STATUS 视图列定义完全相同，这是由于 GLOBAL_THREAD_WAIT_STATUS 视图本质是到 GBase 8s 中各个节点上查询 THREAD_WAIT_STATUS 视图汇总的结果。

名称	类型	描述
node_name	text	数据库进程名称。
db_name	text	数据库名称。
thread_name	text	线程名称。
query_id	bigint	查询 ID，对应 debug_query_id。
tid	bigint	当前线程的线程号。
sessionid	bigint	session 的 ID。

名称	类型	描述
lwtid	integer	当前线程的轻量级线程号。
psessionid	bigint	streaming 线程的父线程。
tlevel	integer	streaming 线程的层级。
smpid	integer	并行线程的 ID。
wait_status	text	当前线程的等待状态。
wait_event	text	如果 wait_status 是 acquire lock、acquire lwlock、wait io 三种类型，此列描述具体的锁、轻量级锁、IO 的信息。否则是空。

26.2.7.13 LOCAL_THREADPOOL_STATUS

LOCAL_THREADPOOL_STATUS 视图显示线程池下工作线程及会话的状态信息。该视图仅在线程池开启（enable_thread_pool = on）时生效。

名称	类型	描述
node_name	text	数据库进程名称。
group_id	integer	线程池组 ID。
bind_numa_id	integer	该线程池组绑定的 NUMA ID。
bind_cpu_number	integer	该线程池组绑定的 CPU 信息。如果未绑定 CPU，该值为 NULL。

名称	类型	描述
listener	integer	该线程池组的 Listener 线程数量。
worker_info	text	线程池中线程相关信息，包括以下信息： default：该线程池组中的初始线程数量。 new：该线程池组中新增线程的数量。 expect：该线程池组中预期线程的数量。 actual：该线程池组中实际线程的数量。 idle：该线程池组中空闲线程的数量。 pending：该线程池组中等待线程的数量。
session_info	text	线程池中会话相关信息，包括以下信息： total：该线程池组中所有的会话数量。 waiting：该线程池组中等待调度的会话数量。 running：该线程池中正在执行的会话数量。 idle：该线程池组中空闲的会话数量。

26.2.7.14 GLOBAL_THREADPOOL_STATUS

GLOBAL_THREADPOOL_STATUS 视图显示在所有节点上的线程池中工作线程及会话的状态信息。具体的字段表 LOCAL_THREADPOOL_STATUS。

26.2.7.15 SESSION_CPU_RUNTIME

SESSION_CPU_RUNTIME 视图显示和当前用户执行复杂作业（正在运行）时的负载管理 CPU 使用的信息。

名称	类型	描述

名称	类型	描述
datid	oid	连接后端的数据库 OID。
usename	name	登录到该后端的用户名。
pid	bigint	后端线程 ID。
start_time	timestamp with time zone	语句执行的开始时间。
min_cpu_time	bigint	语句在数据库节点上的最小 CPU 时间，单位为 ms。
max_cpu_time	bigint	语句在数据库节点上的最大 CPU 时间，单位为 ms。
total_cpu_time	bigint	语句在数据库节点上的 CPU 总时间，单位为 ms。
query	text	正在执行的语句。
node_group	text	语句所属用户对应的逻辑 GBase 8s。
top_cpu_dn	text	cpu 使用量 topN 信息。

26.2.7.16 SESSION_MEMORY_RUNTIME

名称	类型	描述

datid	oid	连接后端的数据库 OID。
usename	name	登录到该后端的用户名。
pid	bigint	后端线程 ID。
start_time	timestamp with time zone	语句执行的开始时间。
min_peak_memory	integer	语句在数据库节点上的最小内存峰值大小，单位 MB。
max_peak_memory	integer	语句在数据库节点上的最大内存峰值大小，单位 MB。
spill_info	text	语句在数据库节点上的下盘信息： None：数据库节点均未下盘。 All：数据库节点均下盘。 [a:b]：数量为 b 个数据库节点中有 a 个数据库节点下盘。
query	text	正在执行的语句。
node_group	text	语句所属用户对应的逻辑 GBase 8s。
top_mem_dn	text	mem 使用量 topN 信息。

SESSION_MEMORY_RUNTIME 视图显示和当前用户执行复杂作业正在运行时的负载管理内存使用的信息。

名称	类型	描述
datid	oid	连接后端的数据库 OID。
usename	name	登录到该后端的用户名。
pid	bigint	后端线程 ID。
start_time	timestamp with time zone	语句执行的开始时间。
min_peak_memory	integer	语句在数据库节点上的最小内存峰值大小，单位 MB。
max_peak_memory	integer	语句在数据库节点上的最大内存峰值大小，单位 MB。
spill_info	text	语句在数据库节点上的下盘信息： None：数据库节点均未下盘。 All：数据库节点均下盘。 [a:b]：数量为 b 个数据库节点中有 a 个数据库节点下盘。
query	text	正在执行的语句。
node_group	text	语句所属用户对应的逻辑 GBase 8s。
top_mem_dn	text	mem 使用量 topN 信息。

26.2.7.17 STATEMENT_IOSTAT_COMPLEX_RUNTIME

STATEMENT_IOSTAT_COMPLEX_RUNTIME 视图显示当前用户执行作业正在运行时的 IO 负载管理相关信息。以下涉及到 iops，对于行存，均以万次/s 为单位，对于列存，均以次/s 为单位。

名称	类型	描述
query_id	bigint	作业 id。
mincurriops	integer	该作业当前 io 在各数据库节点中的最小值。
maxcurriops	integer	该作业当前 io 在各数据库节点中的最大值。
minpeakiops	integer	在作业运行时，作业 io 峰值中，各数据库节点的最小值。
maxpeakiops	integer	在作业运行时，作业 io 峰值中，各数据库节点的最大值。
io_limits	integer	该作业所设 GUC 参数 io_limits。
io_priority	text	该作业所设 GUC 参数 io_priority。
query	text	作业。
node_group	text	作业所属用户对应的逻辑 GBase 8s。

26.2.7.18 LOCAL_ACTIVE_SESSION

LOCAL_ACTIVE_SESSION 视图显示本节点上的 ACTIVE SESSION PROFILE 内存中的样本。

名称	类型	描述

名称	类型	描述
sampleid	bigint	采样 ID。
sample_time	timestamp with time zone	采样的时间。
need_flush_sample	boolean	该样本是否需要刷新的磁盘。
databaseid	oid	数据库 ID。
thread_id	bigint	线程的 ID。
sessionid	bigint	会话的 ID。
start_time	timestamp with time zone	会话的启动时间。
event	text	具体的事件名称。
lwtid	integer	当前线程的轻量级线程号。
psessionid	bigint	streaming 线程的父线程。
tlevel	integer	streaming 线程的层级。与执行计划的层级（ID）相对应。
smpid	integer	smp 执行模式下并行线程的并行编号。
userid	oid	session 用户的 ID。

名称	类型	描述
application_name	text	应用的名称。
client_addr	inet	client 端的地址。
client_hostname	text	client 端的名称。
client_port	integer	客户端用于与后端通讯的 TCP 端口号。
query_id	bigint	debug query id。
unique_query_id	bigint	unique query id。
user_id	oid	unique query 的 key 中的 user_id。
cn_id	integer	cn id，在 DN 上表示下发该 unique sql 的节点 id，unique query 的 key 中的 cn_id。
unique_query	text	规范化后的 UniqueSQL 文本串。
locktag	text	会话等待锁信息，可通过 locktag_decode 解析。
lockmode	text	会话等待锁模式。
block_sessionid	bigint	如果会话正在等待锁，阻塞该会话获取锁的会话标识。

名称	类型	描述
final_block_sessionid	bigint	表示源头阻塞会话 ID。
wait_status	text	描述 event 列的更多详细信息。
global_sessionid	text	全局会话 ID。

26.2.8 Transaction

26.2.8.1 TRANSACTIONS_PREPARED_XACTS

显示当前准备好进行两阶段提交的事务的信息。

名称	类型	描述
transaction	xid	预备事务的数字事务标识。
gid	text	赋予该事务的全局事务标识。
prepared	timestamp with time zone	事务准备好提交的时间。
owner	name	执行该事务的用户的名称。
database	name	执行该事务所在的数据库名。

26.2.8.2 SUMMARY_TRANSACTIONS_PREPARED_XACTS

显示 GBase 8s 中数据库主节点当前准备好进行两阶段提交的事务的信息。

名称	类型	描述
transaction	xid	预备事务的数字事务标识。
gid	text	赋予该事务的全局事务标识。
prepared	timestamp with time zone	事务准备好提交的时间。
owner	name	执行该事务的用户的名称。
database	name	执行该事务所在的数据库名。

26.2.8.3 GLOBAL_TRANSACTIONS_PREPARED_XACTS

显示各节点当前准备好进行两阶段提交的事务的信息。

名称	类型	描述
transaction	xid	预备事务的数字事务标识。
gid	text	赋予该事务的全局事务标识。
prepared	timestamp with time zone	事务准备好提交的时间。
owner	name	执行该事务的用户的名称。
database	name	执行该事务所在的数据库名。

26.2.8.4 TRANSACTIONS_RUNNING_XACTS

显示当前节点运行事务的信息。

名称	类型	描述
handle	integer	事务对应的事务管理器中的槽位句柄，该值恒为-1。
gxid	xid	事务 id 号。
state	tinyint	事务状态（3：prepared 或者 0：starting）。
node	text	节点名称。
xmin	xid	节点上当前数据涉及的最小事务号 xmin。
vacuum	boolean	标志当前事务是否是 lazy vacuum 事务。
timeline	bigint	标志数据库重启次数。
prepare_xid	xid	处于 prepared 状态的事务的 id 号，若不在 prepared 状态，值为 0。
pid	bigint	事务对应的线程 id。
next_xid	xid	其余节点发送给当前节点的事务 id，该值恒为 0。

26.2.8.5 SUMMARY_TRANSACTIONS_RUNNING_XACTS

显示集群中各个节点运行事务的信息，字段内容和 transactions_running_xacts 一致。

名称	类型	描述

名称	类型	描述
handle	integer	事务对应的事务管理器中的槽位句柄，该值恒为-1。
gxid	xid	事务 id 号。
state	tinyint	事务状态（3：prepared 或者 0：starting）。
node	text	节点名称。
xmin	xid	节点上当前数据涉及的最小事务号 xmin。
vacuum	boolean	标志当前事务是否是 lazy vacuum 事务。
timeline	bigint	标志数据库重启次数。
prepare_xid	xid	处于 prepared 状态的事务的 id 号，若不在 prepared 状态，值为 0。
pid	bigint	事务对应的线程 id。
next_xid	xid	其余节点发送给当前节点的事务 id，该值恒为 0。

26.2.8.6 GLOBAL_TRANSACTIONS_RUNNING_XACTS

显示集群中各个节点运行事务的信息。

名称	类型	描述
handle	integer	事务对应的事务管理器中的槽位句柄，该值恒为-1

名称	类型	描述
gxid	xid	事务 id 号。
state	tinyint	事务状态（3：prepared 或者 0：starting）。
node	text	节点名称。
xmin	xid	节点上当前数据涉及的最小事务号 xmin。
vacuum	boolean	标志当前事务是否是 lazy vacuum 事务。
timeline	bigint	标志数据库重启次数。
prepare_xid	xid	处于 prepared 状态的事务的 id 号，若不在 prepared 状态，值为 0。
pid	bigint	事务对应的线程 id。
next_xid	xid	其余节点发送给当前节点的事务 id，该值恒为 0。

26.2.9 Query

26.2.9.1 STATEMENT

获得当前节点的执行语句（归一化 SQL）的信息。查询视图必须具有 sysadmin 权限或者 monitor admin 权限。数据库主节点上可以看到此数据库主节点接收到的归一化的 SQL 的全量统计信息（包含数据库节点）；数据库节点上仅可看到归一化的 SQL 的此节点执行的统计信息。

名称	类型	描述

名称	类型	描述
node_name	name	数据库进程名称。
node_id	integer	节点的 ID。
user_name	name	用户名称。
user_id	oid	用户 OID。
unique_sql_id	bigint	归一化的 SQL ID。
query	text	归一化的 SQL。
n_calls	bigint	调用次数。
min_elapse_time	bigint	SQL 在内核内的最小运行时间（单位：微秒）。
max_elapse_time	bigint	SQL 在内核内的最大运行时间（单位：微秒）。
total_elapse_time	bigint	SQL 在内核内的总运行时间（单位：微秒）。
n_returned_rows	bigint	SELECT 返回的结果集行数。
n_tuples_fetched	bigint	随机扫描行。
n_tuples_returned	bigint	顺序扫描行。
n_tuples_inserted	bigint	插入行。

名称	类型	描述
n_tuples_updated	bigint	更新行。
n_tuples_deleted	bigint	删除行。
n_blocks_fetched	bigint	buffer 的块访问次数。
n_blocks_hit	bigint	buffer 的块命中次数。
n_soft_parse	bigint	软解析次数，n_soft_parse + n_hard_parse 可能大于 n_calls，因为子查询未计入 n_calls。
n_hard_parse	bigint	硬解析次数，n_soft_parse + n_hard_parse 可能大于 n_calls，因为子查询未计入 n_calls。
db_time	bigint	有效的 DB 时间花费，多线程将累加（单位：微秒）。
cpu_time	bigint	CPU 时间（单位：微秒）。
execution_time	bigint	执行器内执行时间（单位：微秒）。
parse_time	bigint	SQL 解析时间（单位：微秒）。
plan_time	bigint	SQL 生成计划时间（单位：微秒）。
rewrite_time	bigint	SQL 重写时间（单位：微秒）。
pl_execution_time	bigint	plpgsql 上的执行时间（单位：微秒）。

名称	类型	描述
pl_compilation_time	bigint	plpgsql 上的编译时间（单位：微秒）。
net_send_time	bigint	网络上的时间花费（单位：微秒）。
data_io_time	bigint	IO 上的时间花费（单位：微秒）。
sort_count	bigint	排序执行的次数。
sort_time	bigint	排序执行的时间（单位：微秒）。
sort_mem_used	bigint	排序过程中使用的 work memory 大小(单位:KB)。
sort_spill_count	bigint	排序过程中，若发生落盘，写文件的次数。
sort_spill_size	bigint	排序过程中，若发生落盘，使用的文件大小(单位:KB）。
hash_count	bigint	hash 执行的次数。
hash_time	bigint	hash 执行的时间（单位：微秒）。
hash_mem_used	bigint	hash 过程中使用的 work memory 大小(单位:KB)。
hash_spill_count	bigint	hash 过程中，若发生落盘，写文件的次数。
hash_spill_size	bigint	hash 过程中，若发生落盘，使用的文件大小(单位:KB）。

名称	类型	描述
last_updated	timestamp with time zone	最后一次更新该语句的时间。

26.2.9.2 SUMMARY_STATEMENT

获得各数据库主节点的执行语句（归一化 SQL）的全量信息（包含数据库节点）。

名称	类型	描述
node_name	name	数据库进程名称。
node_id	integer	节点的 ID。
user_name	name	用户名称。
user_id	oid	用户 OID。
unique_sql_id	bigint	归一化的 SQL ID。
query	text	归一化的 SQL。
n_calls	bigint	调用次数。
min_elapse_time	bigint	SQL 在内核内的最小运行时间（单位：微秒）。
max_elapse_time	bigint	SQL 在内核内的最大运行时间（单位：微秒）。

名称	类型	描述
total_elapse_time	bigint	SQL 在内核内的总运行时间（单位：微秒）。
n_returned_rows	bigint	SELECT 返回的结果集行数。
n_tuples_fetched	bigint	随机扫描行。
n_tuples_returned	bigint	顺序扫描行。
n_tuples_inserted	bigint	插入行。
n_tuples_updated	bigint	更新行。
n_tuples_deleted	bigint	删除行。
n_blocks_fetched	bigint	buffer 的块访问次数。
n_blocks_hit	bigint	buffer 的块命中次数。
n_soft_parse	bigint	软解析次数。
n_hard_parse	bigint	硬解析次数。
db_time	bigint	有效的 DB 时间花费，多线程将累加（单位：微秒）。
cpu_time	bigint	CPU 时间（单位：微秒）。

名称	类型	描述
execution_time	bigint	执行器内执行时间（单位：微秒）。
parse_time	bigint	SQL 解析时间（单位：微秒）。
plan_time	bigint	SQL 生成计划时间（单位：微秒）。
rewrite_time	bigint	SQL 重写时间（单位：微秒）。
pl_execution_time	bigint	plpgsql 上的执行时间（单位：微秒）。
pl_compilation_time	bigint	plpgsql 上的编译时间（单位：微秒）。
net_send_time	bigint	网络上的时间花费（单位：微秒）。
data_io_time	bigint	IO 上的时间花费（单位：微秒）。
last_updated	timestamp with time zone	最后一次更新该语句的时间。
sort_count	bigint	排序执行的次数。
sort_time	bigint	排序执行的时间（单位：微秒）。
sort_mem_used	bigint	排序过程中使用的 work memory 大小(单位:KB)。
sort_spill_count	bigint	排序过程中，若发生落盘，写文件的次数。

名称	类型	描述
sort_spill_size	bigint	排序过程中，若发生落盘，使用的文件大小（单位：KB）。
hash_count	bigint	hash 执行的次数。
hash_time	bigint	hash 执行的时间（单位：微秒）。
hash_mem_used	bigint	hash 过程中使用的 work memory 大小(单位:KB)。
hash_spill_count	bigint	hash 过程中，若发生落盘，写文件的次数。
hash_spill_size	bigint	hash 过程中，若发生落盘，使用的文件大小（单位：KB）。

26.2.9.3 STATEMENT_COUNT

显示数据库当前节点当前时刻执行的五类语句(SELECT、INSERT、UPDATE、DELETE、MERGE INTO）和（DDL、DML、DCL）统计信息。

说明:

普通用户查询 STATEMENT_COUNT 视图仅能看到该用户当前节点的统计信息；管理员权限用户查询 STATEMENT_COUNT 视图则能看到所有用户当前节点的统计信息。当 GBase 8s 或该节点重启时，计数将清零，并重新开始计数。计数以节点收到的查询数为准，GBase 8s 内部进行的查询。例如，数据库主节点收到一条查询，若下发多条查询数据库节点，那将在数据库节点上进行相应次数的计数。

名称	类型	描述
node_name	text	数据库进程名称。

名称	类型	描述
user_name	text	用户名。
select_count	bigint	select 语句统计结果。
update_count	bigint	update 语句统计结果。
insert_count	bigint	insert 语句统计结果。
delete_count	bigint	delete 语句统计结果。
mergeinto_count	bigint	merge into 语句统计结果。
ddl_count	bigint	DDL 语句的数量。
dml_count	bigint	DML 语句的数量。
dcl_count	bigint	DCL 语句的数量。
total_select_elapse	bigint	总 select 的时间花费（单位：微秒）。
avg_select_elapse	bigint	平均 select 的时间花费（单位：微秒）。
max_select_elapse	bigint	最大 select 的时间花费(单位：微秒)。
min_select_elapse	bigint	最小 select 的时间花费（单位：微秒）。
total_update_elapse	bigint	总 update 的时间花费（单位：微秒）。

名称	类型	描述
avg_update_elapse	bigint	平均 update 的时间花费(单位：微秒)。
max_update_elapse	bigint	最大 update 的时间花费（单位：微秒）。
min_update_elapse	bigint	最小 update 的时间花费（单位：微秒）。
total_insert_elapse	bigint	总 insert 的时间花费（单位：微秒）。
avg_insert_elapse	bigint	平均 insert 的时间花费（单位：微秒）。
max_insert_elapse	bigint	最大 insert 的时间花费（单位：微秒）。
min_insert_elapse	bigint	最小 insert 的时间花费（单位：微秒）。
total_delete_elapse	bigint	总 delete 的时间花费（单位：微秒）。
avg_delete_elapse	bigint	平均 delete 的时间花费（单位：微秒）。
max_delete_elapse	bigint	最大 delete 的时间花费（单位：微秒）。
min_delete_elapse	bigint	最小 delete 的时间花费（单位：微秒）。

26.2.9.4 GLOBAL_STATEMENT_COUNT

显示数据库各节点当前时刻执行的五类语句（SELECT、INSERT、UPDATE、DELETE、MERGE INTO）和（DDL、DML、DCL）统计信息。

名称	类型	描述

名称	类型	描述
node_name	text	数据库进程名称。
user_name	text	用户名。
select_count	bigint	select 语句统计结果。
update_count	bigint	update 语句统计结果。
insert_count	bigint	insert 语句统计结果。
delete_count	bigint	delete 语句统计结果。
mergeinto_count	bigint	merge into 语句统计结果。
ddl_count	bigint	DDL 语句的数量。
dml_count	bigint	DML 语句的数量。
dcl_count	bigint	DCL 语句的数量。
total_select_elapse	bigint	总 select 的时间花费（单位：微秒）。
avg_select_elapse	bigint	平均 select 的时间花费（单位：微秒）。
max_select_elapse	bigint	最大 select 的时间花费（单位：微秒）。
min_select_elapse	bigint	最小 select 的时间花费（单位：微秒）。

名称	类型	描述
total_update_elapse	bigint	总 update 的时间花费（单位：微秒）。
avg_update_elapse	bigint	平均 update 的时间花费（单位：微秒）。
max_update_elapse	bigint	最大 update 的时间花费（单位：微秒）。
min_update_elapse	bigint	最小 update 的时间花费（单位：微秒）。
total_insert_elapse	bigint	总 insert 的时间花费（单位：微秒）。
avg_insert_elapse	bigint	平均 insert 的时间花费（单位：微秒）。
max_insert_elapse	bigint	最大 insert 的时间花费（单位：微秒）。
min_insert_elapse	bigint	最小 insert 的时间花费（单位：微秒）。
total_delete_elapse	bigint	总 delete 的时间花费（单位：微秒）。
avg_delete_elapse	bigint	平均 delete 的时间花费（单位：微秒）。
max_delete_elapse	bigint	最大 delete 的时间花费（单位：微秒）。
min_delete_elapse	bigint	最小 delete 的时间花费（单位：微秒）。

26.2.9.5 SUMMARY_STATEMENT_COUNT

显示数据库汇聚各节点（数据库节点）当前时刻执行的五类语句（SELECT、INSERT、UPDATE、DELETE、MERGE INTO）和（DDL、DML、DCL）统计信息。

名称	类型	描述
user_name	text	用户名。
select_count	numeric	select 语句统计结果。
update_count	numeric	update 语句统计结果。
insert_count	numeric	insert 语句统计结果。
delete_count	numeric	delete 语句统计结果。
mergeinto_count	numeric	merge into 语句统计结果。
ddl_count	numeric	DDL 语句的数量。
dml_count	numeric	DML 语句的数量。
dcl_count	numeric	DCL 语句的数量。
total_select_elapse	numeric	总 select 的时间花费（单位：微秒）。
avg_select_elapse	bigint	平均 select 的时间花费（单位：微秒）。
max_select_elapse	bigint	最大 select 的时间花费（单位：微秒）。
min_select_elapse	bigint	最小 select 的时间花费（单位：微秒）。
total_update_elapse	numeric	总 update 的时间花费（单位：微秒）。

名称	类型	描述
avg_update_elapse	bigint	平均 update 的时间花费（单位：微秒）。
max_update_elapse	bigint	最大 update 的时间花费（单位：微秒）。
min_update_elapse	bigint	最小\ update 的时间花费（单位：微秒）。
total_insert_elapse	numeric	总 insert 的时间花费（单位：微秒）。
avg_insert_elapse	bigint	平均 insert 的时间花费（单位：微秒）。
max_insert_elapse	bigint	最大 insert 的时间花费（单位：微秒）。
min_insert_elapse	bigint	最小\ insert 的时间花费（单位：微秒）。
total_delete_elapse	numeric	总 delete 的时间花费（单位：微秒）。
avg_delete_elapse	bigint	平均 delete 的时间花费（单位：微秒）。
max_delete_elapse	bigint	最大 delete 的时间花费（单位：微秒）。
min_delete_elapse	bigint	最小\ delete 的时间花费（单位：微秒）。

26.2.9.6 GLOBAL_STATEMENT_COMPLEX_HISTORY

显示各个节点执行作业结束后的负载管理记录。

名称	类型	描述

名称	类型	描述
datid	oid	连接后端的数据库 OID。
dbname	text	连接后端的数据库名称。
schemaname	text	模式的名称。
nodename	text	数据库进程名称。
username	text	连接到后端的用户名。
application_name	text	连接到后端的应用名。
client_addr	inet	连接到后端的客户端的 IP 地址。如果此字段是 null，它表明通过服务器机器上 UNIX 套接字连接客户端或者这是内部进程，如 autovacuum。
client_hostname	text	客户端的主机名，这个字段是通过 client_addr 的反向 DNS 查找得到。这个字段只有在启动 log_hostname 且使用 IP 连接时才非空。
client_port	integer	客户端用于与后端通讯的 TCP 端口号，如果使用 Unix 套接字，则为-1。
query_band	text	用于标示作业类型，可通过 GUC 参数 query_band 进行设置，默认为空字符串。

名称	类型	描述
block_time	bigint	语句执行前的阻塞时间，包含语句解析和优化时间，单位 ms。
start_time	timestamp with time zone	语句执行的开始时间。
finish_time	timestamp with time zone	语句执行的结束时间。
duration	bigint	语句实际执行的时间，单位 ms。
estimate_total_time	bigint	语句预估执行时间，单位 ms。
status	text	语句执行结束状态：正常为 finished，异常为 aborted。
abort_info	text	语句执行结束状态为 aborted 时显示异常信息。
resource_pool	text	用户使用的资源池。
control_group	text	语句所使用的 Cgroup。
estimate_memory	integer	语句预估使用内存。
min_peak_memory	integer	语句在数据库节点上的最小内存峰值，单位 MB。

名称	类型	描述
max_peak_memory	integer	语句在数据库节点上的最大内存峰值，单位MB。
average_peak_memory	integer	语句执行过程中的内存使用平均值，单位MB。
memory_skew_percent	integer	语句数据库节点间的内存使用倾斜率。
spill_info	text	语句在数据库节点上的下盘信息： None：数据库节点均未下盘。 All：数据库节点均下盘。 [a:b]：数量为 b 个数据库节点中有 a 个数据库节点下盘。
min_spill_size	integer	若发生下盘，数据库节点上下盘的最小数据量，单位 MB，默认为 0。
max_spill_size	integer	若发生下盘，数据库节点上下盘的最大数据量，单位 MB，默认为 0。
average_spill_size	integer	若发生下盘，数据库节点上下盘的平均数据量，单位 MB，默认为 0。
spill_skew_percent	integer	若发生下盘，数据库节点间下盘倾斜率。
min_dn_time	bigint	语句在数据库节点上的最小执行时间，单位ms。

名称	类型	描述
max_dn_time	bigint	语句在数据库节点上的最大执行时间，单位 ms。
average_dn_time	bigint	语句在数据库节点上的平均执行时间，单位 ms。
dntime_skew_percent	integer	语句在数据库节点的执行时间倾斜率。
min_cpu_time	bigint	语句在数据库节点上的最小 CPU 时间，单位 ms。
max_cpu_time	bigint	语句在数据库节点上的最大 CPU 时间，单位 ms。
total_cpu_time	bigint	语句在数据库节点上的 CPU 总时间，单位 ms。
cpu_skew_percent	integer	语句在数据库节点间的 CPU 时间倾斜率。
min_peak_iops	integer	语句在数据库节点上的每秒最小 IO 峰值（列存单位是次/s，行存单位是万次/s）。
max_peak_iops	integer	语句在数据库节点上的每秒最大 IO 峰值（列存单位是次/s，行存单位是万次/s）。
average_peak_iops	integer	语句在数据库节点上的每秒平均 IO 峰值（列存单位是次/s，行存单位是万次/s）。

名称	类型	描述
iops_skew_percent	integer	语句在数据库节点间的 IO 倾斜率。
warning	text	主要显示如下几类告警信息： Spill file size large than 256MB。 Broadcast size large than 100MB。 Early spill。 Spill times is greater than 3。 Spill on memory adaptive。 Hash table conflict。
queryid	bigint	语句执行使用的内部 query id。
query	text	执行的语句。
query_plan	text	语句的执行计划。
node_group	text	语句所属用户对应的逻辑 GBase 8s。
cpu_top1_node_name	text	cpu 使用率第 1 的节点名称。
cpu_top2_node_name	text	cpu 使用率第 2 的节点名称。
cpu_top3_node_name	text	cpu 使用率第 3 的节点名称。
cpu_top4_node_name	text	cpu 使用率第 4 的节点名称。

名称	类型	描述
cpu_top5_node_name	text	cpu 使用率第 5 的节点名称。
mem_top1_node_name	text	内存使用量第 1 的节点名称。
mem_top2_node_name	text	内存使用量第 2 的节点名称。
mem_top3_node_name	text	内存使用量第 3 的节点名称。
mem_top4_node_name	text	内存使用量第 4 的节点名称。
mem_top5_node_name	text	内存使用量第 5 的节点名称。
cpu_top1_value	bigint	cpu 使用率。
cpu_top2_value	bigint	cpu 使用率。
cpu_top3_value	bigint	cpu 使用率。
cpu_top4_value	bigint	cpu 使用率。
cpu_top5_value	bigint	cpu 使用率。
mem_top1_value	bigint	内存使用量。
mem_top2_value	bigint	内存使用量。
mem_top3_value	bigint	内存使用量。

名称	类型	描述
mem_top4_value	bigint	内存使用量。
mem_top5_value	bigint	内存使用量。
top_mem_dn	text	内存使用量 topN 信息。
top_cpu_dn	text	cpu 使用量 topN 信息。

26.2.9.7 GLOBAL_STATEMENT_COMPLEX_HISTORY_TABLE

显示各个节点执行作业结束后的负载管理记录。此数据是从内核中转储到系统表中的数据。具体的字段请参考 GLOBAL_STATEMENT_COMPLEX_HISTORY 中的字段。

26.2.9.8 GLOBAL_STATEMENT_COMPLEX_RUNTIME

显示当前用户在各个节点上正在执行的作业的负载管理记录。

名称	类型	描述
datid	oid	连接后端的数据 OID。
dbname	name	连接后端的数据库名称。
schemaname	text	模式的名称。
nodename	text	数据库进程名称
username	name	连接到后端的用户名。

名称	类型	描述
application_name	text	连接到后端的应用名。
client_addr	inet	连接到后端的客户端的 IP 地址。如果此字段是 null，它表明通过服务器机器上 UNIX 套接字连接客户端或者这是内部进程，如 autovacuum。
client_hostname	text	客户端的主机名，这个字段是通过 client_addr 的反向 DNS 查找得到。这个字段只有在启动 log_hostname 且使用 IP 连接时才非空。
client_port	integer	客户端用于与后端通讯的 TCP 端口号，如果使用 Unix 套接字，则为-1。
query_band	text	用于标示作业类型，可通过 GUC 参数 query_band 进行设置，默认为空字符串。
pid	bigint	后端线程 ID。
block_time	bigint	语句执行前的阻塞时间，单位 ms。
start_time	timestamp with time zone	语句执行的开始时间。
duration	bigint	语句已经执行的时间，单位 ms。
estimate_total_time	bigint	语句执行预估总时间，单位 ms。

名称	类型	描述
estimate_left_time	bigint	语句执行预估剩余时间，单位 ms。
enqueue	text	工作负载管理资源状态。
resource_pool	name	用户使用的资源池。
control_group	text	语句所使用的 Cgroup。
estimate_memory	integer	语句预估使用内存，单位 MB。
min_peak_memory	integer	语句在数据库节点上的最小内存峰值，单位 MB。
max_peak_memory	integer	语句在数据库节点上的最大内存峰值，单位 MB。
average_peak_memory	integer	语句执行过程中的内存使用平均值，单位 MB。
memory_skew_percent	integer	语句在数据库节点间的内存使用倾斜率。
spill_info	text	语句在数据库节点上的下盘信息： None：数据库节点均未下盘。 All：数据库节点均下盘。 [a:b]：数量为 b 个数据库节点中有 a 个数据库节点下盘。

名称	类型	描述
min_spill_size	integer	若发生下盘，数据库节点上下盘的最小数据量，单位 MB，默认为 0。
max_spill_size	integer	若发生下盘，数据库节点上下盘的最大数据量，单位 MB，默认为 0。
average_spill_size	integer	若发生下盘，数据库节点上下盘的平均数据量，单位 MB，默认为 0。
spill_skew_percent	integer	若发生下盘，数据库节点间下盘倾斜率。
min_dn_time	bigint	语句在数据库节点上的最小执行时间，单位 ms。
max_dn_time	bigint	语句在数据库节点上的最大执行时间，单位 ms。
average_dn_time	bigint	语句在数据库节点上的平均执行时间，单位 ms。
dntime_skew_percent	integer	语句在数据库节点的执行时间倾斜率。
min_cpu_time	bigint	语句在数据库节点上的最小 CPU 时间，单位 ms。
max_cpu_time	bigint	语句在数据库节点上的最大 CPU 时间，单位 ms。

名称	类型	描述
total_cpu_time	bigint	语句在数据库节点上的 CPU 总时间，单位 ms。
cpu_skew_percent	integer	语句在数据库节点间的 CPU 时间倾斜率。
min_peak_iops	integer	语句在数据库节点上的每秒最小 IO 峰值（列存单位是次/s，行存单位是万次/s）。
max_peak_iops	integer	语句在数据库节点上的每秒最大 IO 峰值（列存单位是次/s，行存单位是万次/s）。
average_peak_iops	integer	语句在数据库节点上的每秒平均 IO 峰值（列存单位是次/s，行存单位是万次/s）。
iops_skew_percent	integer	语句在数据库节点间的 IO 倾斜率。
warning	text	主要显示如下几类告警信息： Spill file size large than 256MB。 Broadcast size large than 100MB。 Early spill。 Spill times is greater than 3。 Spill on memory adaptive。 Hash table conflict。
queryid	bigint	语句执行使用的内部 query id。
query	text	正在执行的语句。

名称	类型	描述
query_plan	text	语句的执行计划。
node_group	text	语句所属用户对应的逻辑 GBase 8s。
top_cpu_dn	text	cpu 使用量 topN 信息。
top_mem_dn	text	内存使用量 topN 信息。

26.2.9.9 STATEMENT_RESPONSETIME_PERCENTILE

获取 GBase 8sSQL 响应时间 P80，P95 分布信息。

名称	类型	描述
p80	bigint	GBase 8s80%的 SQL 的响应时间（单位：微秒）。
p95	bigint	GBase 8s95%的 SQL 的响应时间（单位：微秒）。

26.2.9.10 STATEMENT_USER_COMPLEX_HISTORY

STATEMENT_USER_COMPLEX_HISTORY 系统表显示数据库主节点执行作业结束后的负载管理记录。此数据是从内核中转储到系统表中的数据。具体的字段请参考表 GS_SESSION_MEMORY_DETAIL。

26.2.9.11 STATEMENT_COMPLEX_HISTORY_TABLE

STATEMENT_COMPLEX_HISTORY_TABLE 系统表显示数据库主节点执行作业结束后的负载管理记录。此数据是从内核中转储到系统表中的数据。具体的字段请参考表 GS_SESSION_MEMORY_DETAIL。

26.2.9.12 STATEMENT_COMPLEX_HISTORY

STATEMENT_COMPLEX_HISTORY 视图显示在数据库主节点上执行作业结束后的负载管理记录。具体的字段请参考表 GS_SESSION_MEMORY_DETAIL。

26.2.9.13 STATEMENT_COMPLEX_RUNTIME

STATEMENT_COMPLEX_RUNTIME 视图显示当前用户在数据库主节点上正在执行的作业的负载管理记录。

名称	类型	描述
datid	oid	连接后端的数据 OID。
dbname	name	连接后端的数据库名称。
schemaname	text	模式的名称。
nodename	text	数据库进程名称。
username	name	连接到后端的用户名。
application_name	text	连接到后端的应用名。
client_addr	inet	连接到后端的客户端的 IP 地址。如果此字段是 null，它表明通过服务器机器上 UNIX 套接字连接客户端或者这是内部进程，如 autovacuum。
client_hostname	text	客户端的主机名，这个字段是通过 client_addr 的反向 DNS 查找得到。这个字段只有在启动 log_hostname 且使用 IP 连接时才非空。

名称	类型	描述
client_port	integer	客户端用于与后端通讯的 TCP 端口号，如果使用 Unix 套接字，则为-1。
query_band	text	用于标示作业类型，可通过 GUC 参数 query_band 进行设置，默认为空字符串。
pid	bigint	后端线程 ID。
block_time	bigint	语句执行前的阻塞时间，单位 ms。
start_time	timestamp with time zone	语句执行的开始时间。
duration	bigint	语句已经执行的时间，单位 ms。
estimate_total_time	bigint	语句执行预估总时间，单位 ms。
estimate_left_time	bigint	语句执行预估剩余时间，单位 ms。
enqueue	text	工作负载管理资源状态。
resource_pool	name	用户使用的资源池。
control_group	text	语句所使用的 Cgroup。
estimate_memory	integer	语句预估使用内存，单位 MB。

名称	类型	描述
min_peak_memory	integer	语句在数据库节点上的最小内存峰值，单位MB。
max_peak_memory	integer	语句在数据库节点上的最大内存峰值，单位MB。
average_peak_memory	integer	语句执行过程中的内存使用平均值，单位MB。
memory_skew_percent	integer	语句在数据库节点间的内存使用倾斜率。
spill_info	text	语句在数据库节点上的下盘信息： None：数据库节点均未下盘。 All：数据库节点均下盘。 [a:b]：数量为 b 个数据库节点中有 a 个数据库节点下盘。
min_spill_size	integer	若发生下盘，数据库节点上下盘的最小数据量，单位 MB，默认为 0。
max_spill_size	integer	若发生下盘，数据库节点上下盘的最大数据量，单位 MB，默认为 0。
average_spill_size	integer	若发生下盘，数据库节点上下盘的平均数据量，单位 MB，默认为 0。
spill_skew_percent	integer	若发生下盘，数据库节点间下盘倾斜率。

名称	类型	描述
min_dn_time	bigint	语句在数据库节点上的最小执行时间，单位ms。
max_dn_time	bigint	语句在数据库节点上的最大执行时间，单位ms。
average_dn_time	bigint	语句在数据库节点上的平均执行时间，单位ms。
dntime_skew_percent	integer	语句在数据库节点的执行时间倾斜率。
min_cpu_time	bigint	语句在数据库节点上的最小 CPU 时间，单位ms。
max_cpu_time	bigint	语句在数据库节点上的最大 CPU 时间，单位ms。
total_cpu_time	bigint	语句在数据库节点上的 CPU 总时间，单位ms。
cpu_skew_percent	integer	语句在数据库节点间的 CPU 时间倾斜率。
min_peak_iops	integer	语句在数据库节点上的每秒最小 IO 峰值（列存单位是次/s，行存单位是万次/s）。
max_peak_iops	integer	语句在数据库节点上的每秒最大 IO 峰值（列存单位是次/s，行存单位是万次/s）。

名称	类型	描述
average_peak_iops	integer	语句在数据库节点上的每秒平均 IO 峰值（列存单位是次/s，行存单位是万次/s）。
iops_skew_percent	integer	语句在数据库节点间的 IO 倾斜率。
warning	text	主要显示如下几类告警信息： Spill file size large than 256MB。 Broadcast size large than 100MB。 Early spill。 Spill times is greater than 3。 Spill on memory adaptive。 Hash table conflict。
queryid	bigint	语句执行使用的内部 query id。
query	text	正在执行的语句。
query_plan	text	语句的执行计划。
node_group	text	语句所属用户对应的逻辑 GBase 8s。
top_cpu_dn	text	cpu 使用量 topN 信息。
top_mem_dn	text	内存使用量 topN 信息。

26.2.9.14 STATEMENT_WLMSTAT_COMPLEX_RUNTIME

STATEMENT_WLMSTAT_COMPLEX_RUNTIME 视图显示和当前用户执行作业正在运

行时的负载管理相关信息。

名称	类型	描述
datid	oid	连接后端的数据库 OID。
datname	name	连接后端的数据库名称。
threadid	bigint	后端线程 ID。
processid	integer	后端线程的 pid。
usesysid	oid	登录后端的用户 OID。
appname	text	连接到后端的应用名。
usename	name	登录到该后端的用户名。
priority	bigint	语句所在 Cgroups 的优先级。
attribute	text	语句的属性： Ordinary：语句发送到数据库后被解析前的默认属性。 Simple：简单语句。 Complicated：复杂语句。 Internal：数据库内部语句。
block_time	bigint	语句当前为止的 pending 的时间，单位 s。

名称	类型	描述
elapsed_time	bigint	语句当前为止的实际执行时间，单位 s。
total_cpu_time	bigint	语句在上一时间周期内的数据库节点上 CPU 使用的总时间，单位 s。
cpu_skew_percent	integer	语句在上一时间周期内的数据库节点上 CPU 使用的倾斜率。
statement_mem	integer	语句执行使用的 statement_mem，预留字段。
active_points	integer	语句占用的资源池并发点数。
dop_value	integer	语句的从资源池中获取的 dop 值。
control_group	text	该字段不支持。
status	text	该字段不支持。
enqueue	text	语句当前的排队情况，包括： Global：在全局队列中排队。 Respool：在资源池队列中排队。 CentralQueue：在中心协调节点（CCN）中排队。 Transaction：语句处于一个事务块中。 StoredProc：句处于一个存储过程中。 None：未在排队。

名称	类型	描述
		Forced None：事务块语句或存储过程语句由于超出设定的等待时间而强制执行。
resource_pool	name	语句当前所在的资源池。
query	text	该后端的最新查询。如果 state 状态是 active（活的），此字段显示当前正在执行的查询。所有其他情况表示上一个查询。
is_plana	boolean	逻辑 GBase 8s 模式下，语句当前是否占用其他逻辑 GBase 8s 的资源执行。该值默认为 f（否）。
node_group	text	语句所属用户对应的逻辑 GBase 8s。

26.2.9.15 STATEMENT_HISTORY

获得当前节点的执行语句的信息。查询系统表必须具有 sysadmin 权限。只可在系统库中查询到结果，用户库中无法查询。

对于此系统表查询有如下约束：

必须在 postgres 库内查询，其它库中不存数据。

此系统表受 track_stmt_stat_level 控制，默认为“OFF,L0”，第一部分控制 Full SQL，第二部分控制 Slow SQL，具体字段记录级别见下表。

对于 Slow SQL，当 track_stmt_stat_level 的值为非 OFF 时，且 SQL 执行时间超过 log_min_duration_statement，会记录为慢 SQL。

名称	类型	描述	记录级别
db_name	name	数据库名称。	L0
schema_name	name	schema 名称。	L0
origin_node	integer	节点名称。	L0
user_name	name	用户名。	L0
application_name	text	用户发起的请求的应用程序名称。	L0
client_addr	text	用户发起的请求的客户端地址。	L0
client_port	integer	用户发起的请求的客户端端口。	L0
unique_query_id	bigint	归一化 SQL ID。	L0
debug_query_id	bigint	唯一 SQL ID。	L0
query	text	归一化 SQL。	L0
start_time	timestamp with time zone	语句启动的时间。	L0

名称	类型	描述	记录级别
finish_time	timestamp with time zone	语句结束的时间。	L0
slow_sql_threshold	bigint	语句执行时慢 SQL 的标准。	L0
transaction_id	bigint	事务 ID。	L0
thread_id	bigint	执行线程 ID。	L0
session_id	bigint	用户 session id。	L0
n_soft_parse	bigint	软解析次数，n_soft_parse + n_hard_parse 可能大于 n_calls，因为子查询未计入 n_calls。	L0
n_hard_parse	bigint	硬解析次数，n_soft_parse + n_hard_parse 可能大于 n_calls，因为子查询未计入 n_calls。	L0
query_plan	text	语句执行计划。	L1
n_returned_rows	bigint	SELECT 返回的结果集行数。	L0
n_tuples_fetched	bigint	随机扫描行。	L0

名称	类型	描述	记录级别
n_tuples_returned	bigint	顺序扫描行。	L0
n_tuples_inserted	bigint	插入行。	L0
n_tuples_updated	bigint	更新行。	L0
n_tuples_deleted	bigint	删除行。	L0
n_blocks_fetched	bigint	buffer 的块访问次数。	L0
n_blocks_hit	bigint	buffer 的块命中次数。	L0
db_time	bigint	有效的 DB 时间花费，多线程将累加（单位：微秒）。	L0
cpu_time	bigint	CPU 时间（单位：微秒）。	L0
execution_time	bigint	执行器内执行时间（单位：微秒）。	L0
parse_time	bigint	SQL 解析时间（单位：微秒）。	L0
plan_time	bigint	SQL 生成计划时间（单位：微秒）。	L0
rewrite_time	bigint	SQL 重写时间（单位：微秒）。	L0

名称	类型	描述	记录级别
pl_execution_time	bigint	plpgsql 上的执行时间（单位：微秒）。	L0
pl_compilation_time	bigint	plpgsql 上的编译时间（单位：微秒）。	L0
data_io_time	bigint	IO 上的时间花费（单位：微秒）。	L0
net_send_info	text	通过物理连接发送消息的网络状态，包含时间（微秒）、调用次数、吞吐量（字节）。单机模式下不支持该字段。例如：{“time”:xxx, “n_calls”:xxx, “size”:xxx}。	L0
net_recv_info	text	通过物理连接接收消息的网络状态，包含时间（微秒）、调用次数、吞吐量（字节）。单机模式下不支持该字段。例如：{“time”:xxx, “n_calls”:xxx, “size”:xxx}。	L0
net_stream_send_info	text	通过逻辑连接发送消息的网络状态，包含时间（微秒）、调用次数、吞吐量（字节）。单机模式下不支持该字段。例如：{“time”:xxx, “n_calls”:xxx, “size”:xxx}。	L0
net_stream_recv_info	text	通过逻辑连接接收消息的网络状态，包含时间（微秒）、调用次数、吞吐量（字节）。单机模式下不支持该字段。例如：{“time”:xxx, “n_calls”:xxx, “size”:xxx}。	L0

名称	类型	描述	记录级别
lock_count	bigint	加锁次数。	L0
lock_time	bigint	加锁耗时。	L1
lock_wait_count	bigint	加锁等待次数。	L0
lock_wait_time	bigint	加锁等待耗时。	L1
lock_max_count	bigint	最大持锁数量。	L0
lwlock_count	bigint	轻量级加锁次数（预留）。	L0
lwlock_wait_count	bigint	轻量级等锁次数。	L0
lwlock_time	bigint	轻量级加锁时间（预留）。	L1
lwlock_wait_time	bigint	轻量级等锁时间。	L1
details	bytea	语句锁事件的列表，该列表按时间书序记录事件，记录的数量受参数 track_stmt_details_size 的影响。该字段为二进制，需要借助解析函数 pg_catalog.statement_detail_decode 读取。 事件包括：	L2

名称	类型	描述	记录级别
		加锁开始 加锁结束 等锁开始 等锁结束 放锁开始 放锁结束 轻量级等锁开始 轻量级等锁结束	
is_slow_sql	boolean	该 SQL 是否为 slow SQL。 t（true）：表示是。 f（false）：表示不是。	L0
trace_id	text	驱动传入的 trace id，与应用的一次请求相关联。	L0

26.2.10Cache/IO

26.2.10.1 STATIO_USER_TABLES

STATIO_USER_TABLES 视图显示命名空间中所有用户关系表的 IO 状态信息。

名称	类型	描述

名称	类型	描述
relid	oid	表 OID。
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	bigint	从该表中读取的磁盘块数。
heap_blks_hit	bigint	该表缓存命中数。
idx_blks_read	bigint	从表中所有索引读取的磁盘块数。
idx_blks_hit	bigint	表中所有索引命中缓存数。
toast_blks_read	bigint	该表的 TOAST 表读取的磁盘块数（如果存在）。
toast_blks_hit	bigint	该表的 TOAST 表命中缓冲区数（如果存在）。
tidx_blks_read	bigint	该表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	bigint	该表的 TOAST 表索引命中缓冲区数（如果存在）。

26.2.10.2 SUMMARY_STATIO_USER_TABLES

SUMMARY_STATIO_USER_TABLES 视图显示 GBase 8s 内汇聚的命名空间中所有用户关系表的 IO 状态信息。

名称	类型	描述

名称	类型	描述
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	numeric	从该表中读取的磁盘块数。
heap_blks_hit	numeric	此表缓存命中数。
idx_blks_read	numeric	从表中所有索引读取的磁盘块数。
idx_blks_hit	numeric	表中所有索引命中缓存数。
toast_blks_read	numeric	此表的 TOAST 表读取的磁盘块数（如果存在）。
toast_blks_hit	numeric	此表的 TOAST 表命中缓冲区数（如果存在）。
tidx_blks_read	numeric	此表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	numeric	此表的 TOAST 表索引命中缓冲区数（如果存在）。

26.2.10.3 GLOBAL_STATIO_USER_TABLES

GLOBAL_STATIO_USER_TABLES 视图显示各节点的命名空间中所有用户关系表的 IO 状态信息。

名称	类型	描述
node_name	name	节点名称。

名称	类型	描述
relid	oid	表 OID。
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	bigint	从该表中读取的磁盘块数。
heap_blks_hit	bigint	此表缓存命中数。
idx_blks_read	bigint	从表中所有索引读取的磁盘块数。
idx_blks_hit	bigint	表中所有索引命中缓存数。
toast_blks_read	bigint	此表的 TOAST 表读取的磁盘块数（如果存在）。
toast_blks_hit	bigint	此表的 TOAST 表命中缓冲区数（如果存在）。
tidx_blks_read	bigint	此表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	bigint	此表的 TOAST 表索引命中缓冲区数（如果存在）。

STATIO_USER_INDEXES

STATIO_USER_INDEXES 视图显示当前节点命名空间中所有用户关系表索引的 IO 状态信息。

名称	类型	描述

名称	类型	描述
relid	oid	索引的表的 OID。
indexrelid	oid	该索引的 OID。
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。
idx_blks_read	bigint	从索引中读取的磁盘块数。
idx_blks_hit	bigint	索引命中缓存数。

26.2.10.4 SUMMARY_STATIO_USER_INDEXES

SUMMARY_STATIO_USER_INDEXES 视图显示 GBase 8s 内汇聚的命名空间中所有用户关系表索引的 IO 状态信息。

名称	类型	描述
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。
idx_blks_read	numeric	从索引中读取的磁盘块数。

名称	类型	描述
idx_blks_hit	numeric	索引命中缓存数。

26.2.10.5 GLOBAL_STATIO_USER_INDEXES

GLOBAL_STATIO_USER_INDEXES 视图显示各节点的命名空间中所有用户关系表索引的 IO 状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	索引的表的 OID。
indexrelid	oid	该索引的 OID。
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。
idx_blks_read	numeric	从索引中读取的磁盘块数。
idx_blks_hit	numeric	索引命中缓存数。

26.2.10.6 STATIO_USER_SEQUENCES

STATIO_USER_SEQUENCE 视图显示当前节点的命名空间中所有用户关系表类型为序列的 IO 状态信息。

名称	类型	描述
relid	oid	序列 OID。
schemaname	name	序列中模式名。
relname	name	序列名。
blks_read	bigint	从序列中读取的磁盘块数。
blks_hit	bigint	序列中缓存命中数。

26.2.10.7 SUMMARY_STATIO_USER_SEQUENCES

SUMMARY_STATIO_USER_SEQUENCES 视图显示 GBase 8s 内汇聚的命名空间中所有用户关系表类型为序列的 IO 状态信息。

名称	类型	描述
schemaname	name	序列中模式名。
relname	name	序列名。
blks_read	numeric	从序列中读取的磁盘块数。
blks_hit	numeric	序列中缓存命中数。

26.2.10.8 GLOBAL_STATIO_USER_SEQUENCES

GLOBAL_STATIO_USER_SEQUENCES 视图显示各节点的命名空间中所有用户关系表类型为序列的 IO 状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	序列 OID。
schemaname	name	序列中模式名。
relname	name	序列名。
blks_read	bigint	从序列中读取的磁盘块数。
blks_hit	bigint	序列中缓存命中数。

26.2.10.9 STATIO_SYS_TABLES

STATIO_SYS_TABLES 视图显示命名空间中所有系统表的 IO 状态信息。

名称	类型	描述
relid	oid	表 OID。
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	bigint	从该表中读取的磁盘块数。
heap_blks_hit	bigint	该表缓存命中数。

名称	类型	描述
idx_blks_read	bigint	从表中所有索引读取的磁盘块数。
idx_blks_hit	bigint	表中所有索引命中缓存数。
toast_blks_read	bigint	该表的 TOAST 表读取的磁盘块数（如果存在）。
toast_blks_hit	bigint	该表的 TOAST 表命中缓冲区数（如果存在）。
tidx_blks_read	bigint	该表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	bigint	该表的 TOAST 表索引命中缓冲区数（如果存在）。

26.2.10.10 SUMMARY_STATIO_SYS_TABLES

SUMMARY_STATIO_SYS_TABLES 视图显示 GBase 8s 内汇聚的命名空间中所有系统表的 IO 状态信息。

名称	类型	描述
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	numeric	从该表中读取的磁盘块数。
heap_blks_hit	numeric	此表缓存命中数。
idx_blks_read	numeric	从表中所有索引读取的磁盘块数。

名称	类型	描述
idx_blks_hit	numeric	表中所有索引命中缓存数。
toast_blks_read	numeric	此表的 TOAST 表读取的磁盘块数（如果存在）。
toast_blks_hit	numeric	此表的 TOAST 表命中缓冲区数（如果存在）。
tidx_blks_read	numeric	此表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	numeric	此表的 TOAST 表索引命中缓冲区数（如果存在）。

26.2.10.11 GLOBAL_STATIO_SYS_TABLES

GLOBAL_STATIO_SYS_TABLES 视图显示各节点的命名空间中所有系统表的 IO 状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	表 OID。
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	bigint	从该表中读取的磁盘块数。
heap_blks_hit	bigint	此表缓存命中数。

名称	类型	描述
idx_blks_read	bigint	从表中所有索引读取的磁盘块数。
idx_blks_hit	bigint	表中所有索引命中缓存数。
toast_blks_read	bigint	此表的 TOAST 表读取的磁盘块数（如果存在）。
toast_blks_hit	bigint	此表的 TOAST 表命中缓冲区数（如果存在）。
tidx_blks_read	bigint	此表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	bigint	此表的 TOAST 表索引命中缓冲区数（如果存在）。

26.2.10.12 STATIO_SYS_INDEXES

STATIO_SYS_INDEXES 显示命名空间中所有系统表索引的 IO 状态信息。

名称	类型	描述
relid	oid	索引的表的 OID。
indexrelid	oid	该索引的 OID。
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。

名称	类型	描述
idx_blks_read	bigint	从索引中读取的磁盘块数。
idx_blks_hit	bigint	索引命中缓存数。

26.2.10.13 SUMMARY_STATIO_SYS_INDEXES

SUMMARY_STATIO_SYS_INDEXES 视图显示 GBase 8s 内汇聚的命名空间中所有系统表索引的 IO 状态信息。

名称	类型	描述
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。
idx_blks_read	numeric	从索引中读取的磁盘块数。
idx_blks_hit	numeric	索引命中缓存数。

26.2.10.14 GLOBAL_STATIO_SYS_INDEXES

GLOBAL_STATIO_SYS_INDEXES 视图显示各节点的命名空间中所有系统表索引的 IO 状态信息。

名称	类型	描述
node_name	name	数据库进程名称。

名称	类型	描述
relid	oid	索引的表的 OID。
indexrelid	oid	该索引的 OID。
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。
idx_blks_read	numeric	从索引中读取的磁盘块数。
idx_blks_hit	numeric	索引命中缓存数。

26.2.10.15 STATIO_SYS_SEQUENCES

STATIO_SYS_SEQUENCES 显示命名空间中所有系统表为序列的 IO 状态信息。

名称	类型	描述
relid	oid	序列 OID。
schemaname	name	序列中模式名。
relname	name	序列名。
blks_read	bigint	从序列中读取的磁盘块数。

名称	类型	描述
blks_hit	bigint	序列中缓存命中数。

26.2.10.16 SUMMARY_STATIO_SYS_SEQUENCES

SUMMARY_STATIO_SYS_SEQUENCES 视图显示 GBase 8s 内汇聚的命名空间中所有系统表为序列的 IO 状态信息。

名称	类型	描述
schemaname	name	序列中模式名。
relname	name	序列名。
blks_read	numeric	从序列中读取的磁盘块数。
blks_hit	numeric	序列中缓存命中数。

26.2.10.17 GLOBAL_STATIO_SYS_SEQUENCES

GLOBAL_STATIO_SYS_SEQUENCES 视图显示各节点的命名空间中所有系统表为序列的 IO 状态信息。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	序列 OID。
schemaname	name	序列中模式名。

名称	类型	描述
relname	name	序列名。
blks_read	bigint	从序列中读取的磁盘块数。
blks_hit	bigint	序列中缓存命中数。

26.2.10.18 STATIO_ALL_TABLES

STATIO_ALL_TABLES 视图将包含数据库中每个表（包括 TOAST 表）的一行，显示出特定表 I/O 的统计。

名称	类型	描述
relid	oid	表 OID。
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	bigint	从该表中读取的磁盘块数。
heap_blks_hit	bigint	该表缓存命中数。
idx_blks_read	bigint	从表中所有索引读取的磁盘块数。
idx_blks_hit	bigint	表中所有索引命中缓存数。
toast_blks_read	bigint	该表的 TOAST 表读取的磁盘块数（如果存在）。

名称	类型	描述
toast_blks_hit	bigint	该表的 TOAST 表命中缓冲区数（如果存在）。
tidx_blks_read	bigint	该表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	bigint	该表的 TOAST 表索引命中缓冲区数（如果存在）。

26.2.10.19 SUMMARY_STATIO_ALL_TABLES

SUMMARY_STATIO_ALL_TABLES 视图将包含 GBase 8s 内汇聚的数据库中每个表(包括 TOAST 表）的 I/O 的统计。

名称	类型	描述
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	numeric	从该表中读取的磁盘块数。
heap_blks_hit	numeric	此表缓存命中数。
idx_blks_read	numeric	从表中所有索引读取的磁盘块数。
idx_blks_hit	numeric	表中所有索引命中缓存数。
toast_blks_read	numeric	此表的 TOAST 表读取的磁盘块数（如果存在）。
toast_blks_hit	numeric	此表的 TOAST 表命中缓冲区数（如果存在）。

名称	类型	描述
tidx_blks_read	numeric	此表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	numeric	此表的 TOAST 表索引命中缓冲区数（如果存在）。

26.2.10.20 GLOBAL_STATIO_ALL_TABLES

GLOBAL_STATIO_ALL_TABLES 视图将包含各节点的数据库中每个表（包括 TOAST 表）的 I/O 的统计。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	表 OID。
schemaname	name	该表模式名。
relname	name	表名。
heap_blks_read	bigint	从该表中读取的磁盘块数。
heap_blks_hit	bigint	此表缓存命中数。
idx_blks_read	bigint	从表中所有索引读取的磁盘块数。
idx_blks_hit	bigint	表中所有索引命中缓存数。
toast_blks_read	bigint	此表的 TOAST 表读取的磁盘块数（如果存在）。

名称	类型	描述
toast_blks_hit	bigint	此表的 TOAST 表命中缓冲区数（如果存在）。
tidx_blks_read	bigint	此表的 TOAST 表索引读取的磁盘块数（如果存在）。
tidx_blks_hit	bigint	此表的 TOAST 表索引命中缓冲区数（如果存在）。

26.2.10.21 STATIO_ALL_INDEXES

STATIO_ALL_INDEXES 视图包含数据库中的每个索引行，显示特定索引的 I/O 的统计。

名称	类型	描述
relid	oid	索引的表的 OID。
indexrelid	oid	该索引的 OID。
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。
idx_blks_read	bigint	从索引中读取的磁盘块数。
idx_blks_hit	bigint	索引命中缓存数。

26.2.10.22 SUMMARY_STATIO_ALL_INDEXES

SUMMARY_STATIO_ALL_INDEXES 视图包含含 GBase 8s 内汇聚的数据库中的每个索引行， 显示特定索引的 I/O 的统计。

名称	类型	描述
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。
idx_blks_read	numeric	从索引中读取的磁盘块数。
idx_blks_hit	numeric	索引命中缓存数。

26.2.10.23 GLOBAL_STATIO_ALL_INDEXES

GLOBAL_STATIO_ALL_INDEXES 视图包含各节点的数据库中的每个索引行，显示特定索引的 I/O 的统计。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	索引的表的 OID。
indexrelid	oid	该索引的 OID。
schemaname	name	该索引的模式名。
relname	name	该索引的表名。
indexrelname	name	索引名称。

名称	类型	描述
idx_blks_read	numeric	从索引中读取的磁盘块数。
idx_blks_hit	numeric	索引命中缓存数。

26.2.10.24 STATIO_ALL_SEQUENCES

STATIO_ALL_SEQUENCES 视图包含数据库中每个序列的每一行，显示特定序列关于 I/O 的统计。

名称	类型	描述
relid	oid	序列 OID。
schemaname	name	序列中模式名。
relname	name	序列名。
blks_read	bigint	从序列中读取的磁盘块数。
blks_hit	bigint	序列中缓存命中数。

26.2.10.25 SUMMARY_STATIO_ALL_SEQUENCES

SUMMARY_STATIO_ALL_SEQUENCES 视图包含 GBase 8s 内汇聚的数据库中每个序列的每一行,显示特定序列关于 I/O 的统计。

名称	类型	描述
schemaname	name	序列中模式名。

名称	类型	描述
relname	name	序列名。
blks_read	numeric	从序列中读取的磁盘块数。
blks_hit	numeric	序列中缓存命中数。

26.2.10.26 GLOBAL_STATIO_ALL_SEQUENCES

GLOBAL_STATIO_ALL_SEQUENCES 包含各节点的数据库中每个序列的每一行，显示特定序列关于 I/O 的统计。

名称	类型	描述
node_name	name	数据库进程名称。
relid	oid	序列 OID。
schemaname	name	序列中模式名。
relname	name	序列名。
blks_read	bigint	从序列中读取的磁盘块数。
blks_hit	bigint	序列中缓存命中数。

26.2.10.27 GLOBAL_STAT_SESSION_CU

GLOBAL_STAT_DB_CU 视图用于查询 GBase 8s，每个数据库的 CU 命中情况。可以通过 pg_stat_reset()进行清零。

名称	类型	描述
node_name1	text	数据库进程名称。
db_name	text	数据库名。
mem_hit	bigint	内存命中次数。
hdd_sync_read	bigint	硬盘同步读次数。
hdd_asyn_read	bigint	硬盘异步读次数。

26.2.10.28 GLOBAL_STAT_DB_CU

GLOBAL_STAT_SESSION_CU 用于查询 GBase 8s 各个节点，当前运行 session 的 CU 命中情况。session 退出相应的统计数据会清零。GBase 8s 重启后，统计数据也会清零。

名称	类型	描述
mem_hit	integer	内存命中次数。
hdd_sync_read	integer	硬盘同步读次数。
hdd_asyn_read	integer	硬盘异步读次数。

26.2.11 Utility

26.2.11.1 REPLICATION_STAT

REPLICATION_STAT 用于描述日志同步状态信息，如发起端发送日志位置、收端接收日志位置等。

名称	类型	描述

名称	类型	描述
pid	bigint	线程的 PID。
usesysid	oid	用户系统 ID。
usename	name	用户名。
application_name	text	程序名称。
client_addr	inet	客户端地址。
client_hostname	text	客户端名。
client_port	integer	客户端端口。
backend_start	timestamp with time zone	程序启动时间。
state	text	日志复制的状态： 追赶状态。 一致的流状态。
sender_sent_location	text	发送端发送日志位置。
receiver_write_location	text	接收端 write 日志位置。
receiver_flush_location	text	接收端 flush 日志位置。

名称	类型	描述
receiver_replay_location	text	接收端 replay 日志位置。
sync_priority	integer	同步复制的优先级（0 表示异步）。
sync_state	text	同步状态： 异步复制。 同步复制。 潜在同步者。

26.2.11.2 GLOBAL_REPLICATION_STAT

GLOBAL_REPLICATION_STAT 视图用于获得各节点描述日志同步状态信息，如发起端发送日志位置、收端接收日志位置等。

名称	类型	描述
node_name	name	数据库进程名称。
pid	bigint	线程的 PID。
usesysid	oid	用户系统 ID。
usename	name	用户名。
application_name	text	程序名称。
client_addr	inet	客户端地址。

名称	类型	描述
client_hostname	text	客户端名。
client_port	integer	客户端端口。
backend_start	timestamp with time zone	程序启动时间。
state	text	日志复制的状态： 追赶状态。 一致的流状态。
sender_sent_location	text	发送端发送日志位置。
receiver_write_location	text	接收端 write 日志位置。
receiver_flush_location	text	接收端 flush 日志位置。
receiver_replay_location	text	接收端 replay 日志位置。
sync_priority	integer	同步复制的优先级（0 表示异步）。
sync_state	text	同步状态： 异步复制。 同步复制。 潜在同步者。

REPLICATION_SLOTS

REPLICATION_SLOTS 视图用于查看复制节点的信息。

名称	类型	描述
slot_name	text	复制节点的名称。
plugin	text	插件名称。
slot_type	text	复制节点的类型。
datoid	oid	复制节点的数据库 OID。
database	name	复制节点的数据库名称。
active	boolean	复制节点是否为激活状态。
xmin	xid	复制节点事务标识。
catalog_xmin	xid	逻辑复制槽对应的最早解码事务标识。
restart_lsn	text	复制节点的 Xlog 文件信息。
dummy_standby	boolean	复制节点是否为假备。

26.2.11.3 GLOBAL_REPLICATION_SLOTS

GLOBAL_REPLICATION_SLOTS 视图用于查看 GBase 8s 各节点的复制节点的信息。

名称	类型	描述

名称	类型	描述
node_name	name	数据库进程名称。
slot_name	text	复制节点的名称。
plugin	text	插件名称。
slot_type	text	复制节点的类型。
datoid	oid	复制节点的数据库 OID。
database	name	复制节点的数据库名称。
active	boolean	复制节点是否为激活状态。
x_min	xid	复制节点事务标识。
catalog_xmin	xid	逻辑复制槽对应的最早解码事务标识.
restart_lsn	text	复制节点的 Xlog 文件信息。
dummy_standby	boolean	复制节点是否为假备。

26.2.11.4 BGWRITER_STAT

BGWRITER_STAT 视图显示关于后端写进程活动的统计信息。

名称	类型	描述

名称	类型	描述
checkpoints_timed	bigint	执行的定期检查点数。
checkpoints_req	bigint	执行的需求检查点数。
checkpoint_write_time	double precision	花费在检查点处理部分的时间总量，其中文件被写入到磁盘，以毫秒为单位。
checkpoint_sync_time	double precision	花费在检查点处理部分的时间总量，其中文件被同步到磁盘，以毫秒为单位。
buffers_checkpoint	bigint	检查点写缓冲区数量。
buffers_clean	bigint	后端写进程写缓冲区数量。
maxwritten_clean	bigint	后端写进程停止清理扫描时间数，因为它写了太多缓冲区。
buffers_backend	bigint	通过后端直接写缓冲区数。
buffers_backend_fsync	bigint	后端不得不执行自己的 fsync 调用的时间数（通常后端写进程处理这些即使后端确实自己写）。
buffers_alloc	bigint	分配的缓冲区数量。
stats_reset	timestamp with time zone	这些统计被重置的时间。

26.2.11.5 GLOBAL_BGWRITER_STAT

GLOBAL_BGWRITER_STAT 视图显示各节点关于后端写进程活动的统计信息。

名称	类型	描述
node_name	name	数据库进程名称。
checkpoints_timed	bigint	执行的定期检查点数。
checkpoints_req	bigint	执行的需求检查点数。
checkpoint_write_time	double precision	花费在检查点处理部分的时间总量，其中文件被写入到磁盘，以毫秒为单位。
checkpoint_sync_time	double precision	花费在检查点处理部分的时间总量，其中文件被同步到磁盘，以毫秒为单位。
buffers_checkpoint	bigint	检查点写缓冲区数量。
buffers_clean	bigint	后端写进程写缓冲区数量。
maxwritten_clean	bigint	后端写进程停止清理扫描时间数，因为它写了太多缓冲区。
buffers_backend	bigint	通过后端直接写缓冲区数。
buffers_backend_fsync	bigint	后端不得不执行自己的 fsync 调用的时间数（通常后端写进程处理这些即

名称	类型	描述
		使后端确实自己写）。
buffers_alloc	bigint	分配的缓冲区数量。
stats_reset	timestamp with time zone	这些统计被重置的时间。

26.2.11.6 GLOBAL_CKPT_STATUS

GLOBAL_CKPT_STATUS 视图用于显示 GBase 8s 所有实例的检查点信息和各类日志刷页情况。

名称	类型	描述
node_name	text	数据库进程名称。
ckpt_redo_point	test	当前实例的检查点。
ckpt_clog_flush_num	bigint	从启动到当前时间 clog 刷盘页面数。
ckpt_csnlog_flush_num	bigint	从启动到当前时间 csnlog 刷盘页面数。
ckpt_multixact_flush_num	bigint	从启动到当前时间 multixact 刷盘页面数。
ckpt_predicate_flush_num	bigint	从启动到当前时间 predicate 刷盘页面数。

名称	类型	描述
ckpt_twophase_flush_num	bigint	从启动到当前时间 twophase 刷盘页面数。

26.2.11.7 GLOBAL_DOUBLE_WRITE_STATUS

GLOBAL_DOUBLE_WRITE_STATUS 视图显示 GBase 8s 所有实例的双写文件的情况。它是由每个节点的 local_double_write_stat 视图组成，属性完全一致。

名称	类型	描述
node_name	text	节点名称。
curr_dwn	bigint	当前双写文件的序列号。
curr_start_page	bigint	当前双写文件恢复起始页面。
file_trunc_num	bigint	当前双写文件复用的次数。
file_reset_num	bigint	当前双写文件写满后发生重置的次数。
total_writes	bigint	当前双写文件总的 I/O 次数。
low_threshold_writes	bigint	低效率写双写文件的 I/O 次数（一次 I/O 刷页数量少于 16 页面）。
high_threshold_writes	bigint	高效率写双写文件的 I/O 次数（一次 I/O 刷页数量多于一批，421 个页面）。

名称	类型	描述
total_pages	bigint	当前刷页到双写文件区的总的页面个数。
low_threshold_pages	bigint	低效率刷页的页面个数。
high_threshold_pages	bigint	高效率刷页的页面个数。
file_id	bigint	当前双写文件的 id 号。

26.2.11.8 GLOBAL_PAGEWRITER_STATUS

GLOBAL_PAGEWRITER_STATUS 视图显示 GBase 8s 实例的刷页信息和检查点信息。

名称	类型	描述
node_name	text	数据库进程名称。
pgwr_actual_flush_total_num	bigint	从启动到当前时间总计刷脏页数量。
pgwr_last_flush_num	integer	上一批刷脏页数量。
remain_dirty_page_num	bigint	当前预计还剩余多少脏页。
queue_head_page_rec_lsn	text	当前实例的脏页队列第一个脏页的 recovery_lsn。
queue_rec_lsn	text	当前实例的脏页队列的 recovery_lsn。

名称	类型	描述
current_xlog_insert_lsn	text	当前实例 XLog 写入的位置。
ckpt_redo_point	text	当前实例的检查点。

26.2.11.9 GLOBAL_RECORD_RESET_TIME

GLOBAL_RECORD_RESET_TIME 用于重置(重启、主备倒换、数据库删除)汇聚 GBase 8s 统计信息时间。

名称	类型	描述
node_name	text	数据库进程名称。
reset_time	timestamp with time zone	重置时间点。

26.2.11.10 GLOBAL_REDO_STATUS

GLOBAL_REDO_STATUS 视图显示 GBase 8s 实例的日志回放情况。

名称	类型	描述
node_name	text	数据库进程名称。
redo_start_ptr	bigint	当前实例日志回放的起始点。
redo_start_time	bigint	当前实例日志回放的起始 UTC 时间。

名称	类型	描述
redo_done_time	bigint	当前实例日志回放的结束 UTC 时间。
curr_time	bigint	当前实例的当前 UTC 时间。
min_recovery_point	bigint	当前实例日志的最小一致性点位置。
read_ptr	bigint	当前实例日志的读取位置。
last_replayed_read_ptr	bigint	当前实例的日志回放位置。
recovery_done_ptr	bigint	当前实例启动完成时的回放位置。
read_xlog_io_counter	bigint	当前实例读取回放日志的 io 次数计数。
read_xlog_io_total_dur	bigint	当前实例读取回放日志的 io 总时延。
read_data_io_counter	bigint	当前实例回放过程中读取数据页面的 io 次数计数。
read_data_io_total_dur	bigint	当前实例回放过程中读取数据页面的 io 总时延。
write_data_io_counter	bigint	当前实例回放过程中写数据页面的 io 次数计数。
write_data_io_total_dur	bigint	当前实例回放过程中写数据页面的 io 总时延。

名称	类型	描述
process_pending_counter	bigint	当前实例回放过程中日志分发线程的同步次数计数。
process_pending_total_dur	bigint	当前实例回放过程中日志分发线程的同步总时延。
apply_counter	bigint	当前实例回放过程中回放线程的同步次数计数。
apply_total_dur	bigint	当前实例回放过程中回放线程的同步总时延。
speed	bigint	当前实例日志回放速率。
local_max_ptr	bigint	当前实例启动成功后本地收到的回放日志的最大值。
primary_flush_ptr	bigint	主机落盘日志的位置。
worker_info	text	当前实例回放线程信息，若没有开并行回放则该值为空。

26.2.11.11 GLOBAL_RECOVERY_STATUS

GLOBAL_RECOVERY_STATUS 视图显示关于主机和备机的日志流控信息。

名称	类型	描述

名称	类型	描述
node_name	text	主机进程名称，包含主机和备机。
standby_node_name	text	备机进程名称。
source_ip	text	主机的 IP 地址。
source_port	integer	主机的端口号。
dest_ip	text	备机的 IP 地址。
dest_port	integer	备机的端口号。
current_rto	bigint	备机当前的日志流控时间，单位秒。
target_rto	bigint	备机通过 GUC 参数设置的预期流控时间，单位秒。
current_sleep_time	bigint	为了达到这个预期主机所需要的睡眠时间，单位微妙。

26.2.11.12 CLASS_VITAL_INFO

CLASS_VITAL_INFO 视图用于做 WDR 时校验相同的表或者索引的 oid 是否一致。

名称	类型	描述
relid	oid	表的 oid。

名称	类型	描述
schemaname	name	schema 名称。
relname	name	表名。
relkind	“char”	表示对象类型，取值范围如下： r：表示普通表。 t：表示 toast 表。 i：表示索引。

26.2.11.13 USER_LOGIN

USER_LOGIN 用来记录用户登录和退出次数的相关信息。

名称	类型	描述
node_name	text	数据库进程名称。
user_name	text	用户名称。
user_id	integer	用户 oid（同 pg_authid 中的 oid 字段）。
login_counter	bigint	登录次数。
logout_counter	bigint	退出次数。

26.2.11.14 SUMMARY_USER_LOGIN

USER_LOGIN 用来记录用户登录和退出次数的相关信息。

名称	类型	描述
node_name	text	数据库进程名称。
user_name	text	用户名称。
user_id	integer	用户 oid（同 pg_authid 中的 oid 字段）。
login_counter	bigint	登录次数。
logout_counter	bigint	退出次数。

26.2.11.15 GLOBAL_GET_BGWRITER_STATUS

GLOBAL_GET_BGWRITER_STATUS 视图显示所有实例 bgwriter 线程刷页信息，候选 buffer 链中页面个数，buffer 淘汰信息。

名称	类型	描述
node_name	text	实例名称。
bgwr_actual_flush_total_num	bigint	从启动到当前时间 bgwriter 线程总计刷脏页数量。
bgwr_last_flush_num	integer	bgwriter 线程上一批刷脏页数量。
candidate_slots	integer	当前候选 buffer 链中页面个数。

名称	类型	描述
get_buffer_from_list	bigint	buffer 淘汰从候选 buffer 链中获取页面的次数。
get_buffer_clock_sweep	bigint	buffer 淘汰从原淘汰方案中获取页面的次数。

26.2.11.16 GLOBAL_SINGLE_FLUSH_DW_STATUS

GLOBAL_SINGLE_FLUSH_DW_STATUS 视图显示数据库所有实例单页面淘汰双写文件信息。展示内容中，/前是第一个版本双写文件刷页情况，/后是第二个版本双写文件刷页情况。

名称	类型	描述
node_name	text	实例名称。
curr_dwn	text	当前双写文件的序列号。
curr_start_page	text	当前双写文件 start 位置。
total_writes	text	当前双写文件总计写数据页面个数。
file_trunc_num	text	当前双写文件复用的次数。
file_reset_num	text	当前双写文件写满后发生重置的次数。

26.2.11.17 GLOBAL_CANDIDATE_STATUS

GLOBAL_CANDIDATE_STATUS 视图显示整个数据库所有实例候选 buffer 个数，buffer 淘汰信息。

名称	类型	描述
node_name	text	节点名称。
candidate_slots	integer	当前 Normal Buffer Pool 候选 buffer 链中页面个数。
get_buf_from_list	bigint	Normal Buffer Pool，buffer 淘汰从候选 buffer 链中获取页面的次数。
get_buf_clock_sweep	bigint	Normal Buffer Pool，buffer 淘汰从原淘汰方案中获取页面的次数。
seg_candidate_slots	integer	当前 Segment Buffer Pool 候选 buffer 链中页面个数。
seg_get_buf_from_list	bigint	Segment Buffer Pool，buffer 淘汰从候选 buffer 链中获取页面的次数。
seg_get_buf_clock_sweep	bigint	Segment Buffer Pool，buffer 淘汰从原淘汰方案中获取页面的次数。

26.2.12Lock

26.2.12.1 LOCKS

LOCKS 视图用于查看各打开事务所持有的锁信息。

名称	类型	描述
locktype	text	被锁定对象的类型：relation、extend、page、tuple、transactionid、virtualxid、object、userlock、advisory。

名称	类型	描述
database	oid	被锁定对象所在数据库的 OID： 如果被锁定的对象是共享对象，则 OID 为 0。 如果是一个事务 ID，则为 NULL。
relation	oid	关系的 OID，如果锁定的对象不是关系，也不是关系的一部分，则为 NULL。
page	integer	关系内部的页面编号，如果对象不是关系页或者不是行页，则为 NULL。
tuple	smallint	页面里边的行编号，如果对象不是行，则为 NULL。
bucket	integer	哈希桶号。
virtualxid	text	事务的虚拟 ID，如果对象不是一个虚拟事务 ID，则为 NULL。
transactionid	xid	事务的 ID，如果对象不是一个事务 ID，则为 NULL。
classid	oid	包含该对象的系统表的 OID，如果对象不是普通的数据库对象，则为 NULL。
objid	oid	对象在其系统表内的 OID，如果对象不是普通数据库对象，则为 NULL。
objsubid	smallint	对于表的一个字段，这是字段编号；对于其他对象类型，这个字段是 0；如果这个对象不是普通数据库对

名称	类型	描述
		象，则为 NULL。
virtualtransaction	text	持有此锁或者在等待此锁的事务的虚拟 ID。
pid	bigint	持有或者等待这个锁的服务器线程的逻辑 ID。如果锁是被一个预备事务持有的，则为 NULL。
sessionid	bigint	持有或者等待这个锁的会话 ID。如果锁是被一个预备事务持有的，则为 NULL。
mode	text	这个线程持有的或者是期望的锁模式。
granted	boolean	如果锁是持有锁，则为 TRUE。 如果锁是等待锁，则为 FALSE。
fastpath	boolean	如果通过 fast-path 获得锁，则为 TRUE；如果通过主要的锁表获得，则为 FALSE。
locktag	text	会话等待锁信息，可通过 locktag_decode()函数解析。
global_sessionid	text	全局会话 ID。

26.2.12.2 GLOBAL_LOCKS

GLOBAL_LOCKS 视图用于查看各节点各打开事务所持有的锁信息。

名称	类型	描述
node_name	name	数据库进程名称。

名称	类型	描述
locktype	text	被锁定对象的类型：relation、extend、page、tuple、transactionid、virtualxid、object、userlock、advisory。
database	oid	被锁定对象所在数据库的 OID： 如果被锁定的对象是共享对象，则 OID 为 0。 如果是一个事务 ID，则为 NULL。
relation	oid	关系的 OID，如果锁定的对象不是关系，也不是关系的一部分，则为 NULL。
page	integer	关系内部的页面编号，如果对象不是关系页或者不是行页，则为 NULL。
tuple	smallint	页面里边的行编号，如果对象不是行，则为 NULL。
virtualxid	text	事务的虚拟 ID，如果对象不是一个虚拟事务 ID，则为 NULL。
transactionid	xid	事务的 ID，如果对象不是一个事务 ID，则为 NULL。
classid	oid	包含该对象的系统表的 OID，如果对象不是普通的数据库对象，则为 NULL。
objid	oid	对象在其系统表内的 OID，如果对象不是普通数据库对象，则为 NULL。
objsubid	smallint	对于表的一个字段，这是字段编号；对于其他对象类

名称	类型	描述
		型，这个字段是零；如果这个对象不是普通数据库对象，则为 NULL。
virtualtransaction	text	持有此锁或者在等待此锁的事务的虚拟 ID。
pid	bigint	持有或者等待这个锁的服务器线程的逻辑 ID。如果锁是被一个预备事务持有的，则为 NULL。
mode	text	这个线程持有的或者是期望的锁模式。
granted	boolean	如果锁是持有锁，则为 TRUE。 如果锁是等待锁，则为 FALSE。
fastpath	boolean	如果通过 fast-path 获得锁，则为 TRUE；如果通过主要的锁表获得，则为 FALSE。

26.2.13Wait Events

26.2.13.1 WAIT_EVENTS

WAIT_EVENTS 显示当前节点的 event 的等待相关的统计信息。具体事件信息见 PG_THREAD_WAIT_STATUS 中等待状态列表、轻量级锁等待事件列表、IO 等待事件列表和事务锁等待事件列表。关于每种事务锁对业务的影响程度，请参考 LOCK 语法小节的详细描述。

名称	类型	描述
nodename	text	数据库进程名称。
type	text	event 类型。

名称	类型	描述
event	text	event 名称。
wait	bigint	等待次数。
failed_wait	bigint	失败的等待次数。
total_wait_time	bigint	总等待时间（单位：微秒）。
avg_wait_time	bigint	平均等待时间（单位：微秒）。
max_wait_time	bigint	最大等待时间（单位：微秒）。
min_wait_time	bigint	最小等待时间（单位：微秒）。
last_updated	timestamp with time zone	最后一次更新该事件的时间。

26.2.13.2 GLOBAL_WAIT_EVENTS

WAIT_EVENTS 显示当前节点的 event 的等待相关的统计信息。具体事件信息见 PG_THREAD_WAIT_STATUS 中等待状态列表、轻量级锁等待事件列表、IO 等待事件列表和事务锁等待事件列表。关于每种事务锁对业务的影响程度，请参考 LOCK 语法小节的详细描述。

名称	类型	描述
nodename	text	数据库进程名称。

type	text	event 类型。
event	text	event 名称。
wait	bigint	等待次数。
failed_wait	bigint	失败的等待次数。
total_wait_time	bigint	总等待时间（单位：微秒）。
avg_wait_time	bigint	平均等待时间（单位：微秒）。
max_wait_time	bigint	最大等待时间（单位：微秒）。
min_wait_time	bigint	最小等待时间（单位：微秒）。
last_updated	timestamp with time zone	最后一次更新该事件的时间。

26.2.14Configuration

26.2.14.1 CONFIG_SETTINGS

CONFIG_SETTINGS 视图显示数据库运行时参数的相关信息。

名称	类型	描述
name	text	参数名称。

名称	类型	描述
setting	text	参数当前值。
unit	text	参数的隐式结构。
category	text	参数的逻辑组。
short_desc	text	参数的简单描述。
extra_desc	text	参数的详细描述。
context	text	设置参数值的上下文，包括 internal、postmaster、sighup、backend、superuser、user。
vartype	text	参数类型，包括 bool、enum、integer、real、string。
source	text	参数的赋值方式。
min_val	text	参数最大值。如果参数类型不是数值型，那么该字段值为 null。
max_val	text	参数最小值。如果参数类型不是数值型，那么该字段值为 null。
enumvals	text[]	enum 类型参数合法值。如果参数类型不是 enum 型，那么该字段值为 null。
boot_val	text	数据库启动时参数默认值。

名称	类型	描述
reset_val	text	数据库重置时参数默认值。
sourcefile	text	设置参数值的配置文件。如果参数不是通过配置文件赋值，那么该字段值为 null。
sourceline	integer	设置参数值的配置文件的行号。如果参数不是通过配置文件赋值，那么该字段值为 null。

26.2.14.2 GLOBAL_CONFIG_SETTINGS

GLOBAL_CONFIG_SETTINGS 显示各节点数据库运行时参数的相关信息。

名称	类型	描述
node_name	text	数据库进程名称。
name	text	参数名称。
setting	text	参数当前值。
unit	text	参数的隐式结构。
category	text	参数的逻辑组。
short_desc	text	参数的简单描述。
extra_desc	text	参数的详细描述。

名称	类型	描述
context	text	设置参数值的上下文，包括 internal、postmaster、sighup、backend、superuser、user。
vartype	text	参数类型，包括 bool、enum、integer、real、string。
source	text	参数的赋值方式。
min_val	text	参数最小值。如果参数类型不是数值型，那么该字段值为 null。
max_val	text	参数最大值。如果参数类型不是数值型，那么该字段值为 null。
enumvals	text[]	enum 类型参数合法值。如果参数类型不是 enum 型，那么该字段值为 null。
boot_val	text	数据库启动时参数默认值。
reset_val	text	数据库重置时参数默认值。
sourcefile	text	设置参数值的配置文件。如果参数不是通过配置文件赋值，那么该字段值为 null。
sourceline	integer	设置参数值的配置文件的行号。如果参数不是通过配置文件赋值，那么该字段值为 null。

26.2.15Operator

26.2.15.1 OPERATOR_HISTORY_TABLE

OPERATOR_HISTORY_TABLE 系统表显示执行作业结束后的算子相关的记录。此数据是从内核中转储到系统表中的数据。

名称	类型	描述
queryid	bigint	语句执行使用的内部 query_id。
pid	bigint	后端线程 id。
plan_node_id	integer	查询对应的执行计划的 plan node id。
plan_node_name	text	对应于 plan_node_id 的算子的名称。
start_time	timestamp with time zone	该算子处理第一条数据的开始时间。
duration	bigint	该算子到结束时候总的执行时间（ms）。
query_dop	integer	当前算子执行时的并行度。
estimated_rows	bigint	优化器估算的行数信息。
tuple_processed	bigint	当前算子返回的元素个数。
min_peak_memory	integer	当前算子在数据库节点上的最小内存峰值（MB）。
max_peak_memory	integer	当前算子在数据库节点上的最大内存峰值（MB）。

名称	类型	描述
average_peak_memory	integer	当前算子在数据库节点上的平均内存峰值（MB）。
memory_skew_percent	integer	当前算子在数据库节点间的内存使用倾斜率。
min_spill_size	integer	若发生下盘，数据库节点上下盘的最小数据量（MB），默认为 0。
max_spill_size	integer	若发生下盘，数据库节点上下盘的最大数据量（MB），默认为 0。
average_spill_size	integer	若发生下盘，数据库节点上下盘的平均数据量（MB），默认为 0。
spill_skew_percent	integer	若发生下盘，数据库节点间下盘倾斜率。
min_cpu_time	bigint	该算子在数据库节点上的最小执行时间（ms）。
max_cpu_time	bigint	该算子在数据库节点上的最大执行时间（ms）。
total_cpu_time	bigint	该算子在数据库节点上的总执行时间（ms）。
cpu_skew_percent	integer	数据库节点间执行时间的倾斜率。
warning	text	主要显示如下几类告警信息： Sort/SetOp/HashAgg/HashJoin spill。 Spill file size large than 256MB。 Broadcast size large than 100MB。

名称	类型	描述
		Early spill。 Spill times is greater than 3。 Spill on memory adaptive。 Hash table conflict。

26.2.15.2 OPERATOR_HISTORY

OPERATOR_HISTORY 视图显示的是当前用户数据库主节点上执行作业结束后的算子的相关记录。记录的数据同表 GS_WLM_OPERATOR_INFO。

26.2.15.3 OPERATOR_RUNTIME

OPERATOR_RUNTIME 视图显示当前用户正在执行的作业的算子相关信息。

名称	类型	描述
queryid	bigint	语句执行使用的内部 query_id。
pid	bigint	后端线程 id。
plan_node_id	integer	查询对应的执行计划的 plan node id。
plan_node_name	text	对应于 plan_node_id 的算子的名称。
start_time	timestamp with time zone	该算子处理第一条数据的开始时间。
duration	bigint	该算子到结束时候总的执行时间（ms）。

名称	类型	描述
status	text	当前算子的执行状态，包括 finished 和 running。
query_dop	integer	当前算子执行时的并行度。
estimated_rows	bigint	优化器估算的行数信息。
tuple_processed	bigint	当前算子返回的元素个数。
min_peak_memory	integer	当前算子在数据库节点上的最小内存峰值（MB）。
max_peak_memory	integer	当前算子在数据库节点上的最大内存峰值（MB）。
average_peak_memory	integer	当前算子在数据库节点上的平均内存峰值（MB）。
memory_skew_percent	integer	当前算子在数据库节点的内存使用倾斜率。
min_spill_size	integer	若发生下盘，数据库节点上下盘的最小数据量（MB），默认为 0。
max_spill_size	integer	若发生下盘，数据库节点上下盘的最大数据量（MB），默认为 0。
average_spill_size	integer	若发生下盘，数据库节点上下盘的平均数据量（MB），默认为 0。
spill_skew_percent	integer	若发生下盘，数据库节点间下盘倾斜率。

名称	类型	描述
min_cpu_time	bigint	该算子在数据库节点上的最小执行时间（ms）。
max_cpu_time	bigint	该算子在数据库节点上的最大执行时间(ms)。
total_cpu_time	bigint	该算子在数据库节点上的总执行时间(ms)。
cpu_skew_percent	integer	数据库节点间执行时间的倾斜率。
warning	text	主要显示如下几类告警信息： Sort/SetOp/HashAgg/HashJoin spill。 Spill file size large than 256MB。 Broadcast size large than 100MB。 Early spill。 Spill times is greater than 3。 Spill on memory adaptive。 Hash table conflict。

26.2.15.4 GLOBAL_OPERATOR_HISTORY

GLOBAL_OPERATOR_HISTORY 系统视图显示的是当前用户在数据库主节点上执行作业结束后的算子的相关记录。

名称	类型	描述
queryid	bigint	语句执行使用的内部 query_id。

名称	类型	描述
pid	bigint	后端线程 id。
plan_node_id	integer	查询对应的执行计划的 plan node id。
plan_node_name	text	对应于 plan_node_id 的算子的名称。
start_time	timestamp with time zone	该算子处理第一条数据的开始时间。
duration	bigint	该算子到结束时候总的执行时间(ms)。
query_dop	integer	当前算子执行时的并行度。
estimated_rows	bigint	优化器估算的行数信息。
tuple_processed	bigint	当前算子返回的元素个数。
min_peak_memory	integer	当前算子在数据库节点上的最小内存峰值(MB)。
max_peak_memory	integer	当前算子在数据库节点上的最大内存峰值(MB)。
average_peak_memory	integer	当前算子在数据库节点上的平均内存峰值(MB)。
memory_skew_percent	integer	当前算子在数据库节点间的内存使用倾斜率。

名称	类型	描述
min_spill_size	integer	若发生下盘，数据库节点上下盘的最小数据量(MB)，默认为 0。
max_spill_size	integer	若发生下盘，数据库节点上下盘的最大数据量(MB)，默认为 0。
average_spill_size	integer	若发生下盘，数据库节点上下盘的平均数据量(MB)，默认为 0。
spill_skew_percent	integer	若发生下盘，数据库节点间下盘倾斜率。
min_cpu_time	bigint	该算子在数据库节点上的最小执行时间(ms)。
max_cpu_time	bigint	该算子在数据库节点上的最大执行时间(ms)。
total_cpu_time	bigint	该算子在数据库节点上的总执行时间(ms)。
cpu_skew_percent	integer	数据库节点间执行时间的倾斜率。
warning	text	主要显示如下几类告警信息： Sort/SetOp/HashAgg/HashJoin spill。 Spill file size large than 256MB。 Broadcast size large than 100MB。 Early spill。 Spill times is greater than 3。 Spill on memory adaptive。

名称	类型	描述
		Hash table conflict。

26.2.15.5 GLOBAL_OPERATOR_HISTORY_TABLE

GLOBAL_OPERATOR_HISTORY_TABLE 视图显示数据库主节点执行作业结束后的算子相关的记录。此数据是从内核中转储到系统表 GS_WLM_OPERATOR_INFO 中的数据。该视图是查询数据库主节点系统表 GS_WLM_OPERATOR_INFO 的汇聚视图。表字段同表 GLOBAL_OPERATOR_HISTORY。

26.2.15.6 GLOBAL_OPERATOR_RUNTIME

GLOBAL_OPERATOR_RUNTIME 视图显示当前用户在数据库主节点上正在执行的作业的算子相关信息。

名称	类型	描述
queryid	bigint	语句执行使用的内部 query_id。
pid	bigint	后端线程 id。
plan_node_id	integer	查询对应的执行计划的 plan node id。
plan_node_name	text	对应于 plan_node_id 的算子的名称。
start_time	timestamp with time zone	该算子处理第一条数据的开始时间。
duration	bigint	该算子到结束时候总的执行时间(ms)。

名称	类型	描述
status	text	当前算子的执行状态，包括 finished 和 running。
query_dop	integer	当前算子执行时的并行度。
estimated_rows	bigint	优化器估算的行数信息。
tuple_processed	bigint	当前算子返回的元素个数。
min_peak_memory	integer	当前算子在数据库节点上的最小内存峰值(MB)。
max_peak_memory	integer	当前算子在数据库节点上的最大内存峰值(MB)。
average_peak_memory	integer	当前算子在数据库节点上的平均内存峰值(MB)。
memory_skew_percent	integer	当前算子在数据库节点的内存使用倾斜率。
min_spill_size	integer	若发生下盘，数据库节点上下盘的最小数据量(MB)，默认为 0。
max_spill_size	integer	若发生下盘，数据库节点上下盘的最大数据量(MB)，默认为 0。
average_spill_size	integer	若发生下盘，数据库节点上下盘的平均数据量(MB)，默认为 0。
spill_skew_percent	integer	若发生下盘，数据库节点间下盘倾斜率。

名称	类型	描述
min_cpu_time	bigint	该算子在数据库节点上的最小执行时间(ms)。
max_cpu_time	bigint	该算子在数据库节点上的最大执行时间(ms)。
total_cpu_time	bigint	该算子在数据库节点上的总执行时间(ms)。
cpu_skew_percent	integer	数据库节点间执行时间的倾斜率。
warning	text	主要显示如下几类告警信息： Sort/SetOp/HashAgg/HashJoin spill。 Spill file size large than 256MB。 Broadcast size large than 100MB。 Early spill。 Spill times is greater than 3。 Spill on memory adaptive。 Hash table conflict。

26.2.16Workload Manager

26.2.16.1 WLM_USER_RESOURCE_CONFIG

WLM_USER_RESOURCE_CONFIG 视图显示用户的资源配置信息。

名称	类型	描述
userid	oid	用户 oid。
username	name	用户名称。

名称	类型	描述
sysadmin	boolean	是否是 sysadmin。
rpoid	oid	资源池的 oid。
respool	name	资源池的名称。
parentid	oid	父用户的 oid。
totalspace	bigint	占用总空间大小。
spacelimit	bigint	空间大上限。
childcount	integer	子用户数量。
childlist	text	子用户的列表。

26.2.16.2 WLM_USER_RESOURCE_RUNTIME

WLM_USER_RESOURCE_RUNTIME 视图显示所有用户资源使用情况，需要使用管理员用户进行查询。此视图在 GUC 参数“use_workload_manager”为“on”时才有效。

名称	类型	描述
username	name	用户名。
used_memory	integer	正在使用的内存大小，单位 MB。
total_memory	integer	可以使用的内存大小，单位 MB。值为 0 表示未限制最大可用内存，其限制取决于数据库最大可

名称	类型	描述
		用内存。
used_cpu	integer	正在使用的 CPU 核数。
total_cpu	integer	在该机器节点上，用户关联控制组的 CPU 核数总和。
used_space	bigint	已使用的存储空间大小，单位 KB。
total_space	bigint	可使用的存储空间大小，单位 KB，值为-1 表示未限制最大存储空间。
used_temp_space	bigint	已使用的临时空间大小（预留字段，暂未使用），单位 KB。
total_temp_space	bigint	可使用的临时空间大小（预留字段，暂未使用），单位 KB，值为-1 表示未限制最大临时存储空间。
used_spill_space	bigint	已使用的下盘空间大小（预留字段，暂未使用），单位 KB。
total_spill_space	bigint	可使用的下盘空间大小（预留字段，暂未使用），单位 KB，值为-1 表示未限制最大下盘空间。

26.2.17Global Plancache

GPC 相关视图在 enable_global_plancache 打开且线程池打开的状态下才有效。

26.2.17.1 GLOBAL_PLANCACHE_STATUS

GLOBAL_PLANCACHE_STATUS 视图显示 GPC 全局计划缓存状态信息。

名称	类型	描述
nodename	text	所属节点名称。
query	text	查询语句 text。
refcount	integer	被引用次数。
valid	bool	是否合法。
databaseid	oid	所属数据库 id。
schema_name	text	所属 schema。
params_num	integer	参数数量。
func_id	oid	该 plancache 所在存储过程 oid，如果不属于存储过程则为 0。

26.2.17.2 GLOBAL_PLANCACHE_CLEAN

GLOBAL_PLANCACHE_CLEAN 视图用于清理所有节点上无人使用的全局计划缓存。返回值为 Boolean 类型。

26.2.18RTO & RPO

26.2.18.1 global_rto_status

global_rto_status 视图显示关于主机和备机的日志流控信息（本节点除外、备 DN 上不可使用）。

参数	类型	描述

参数	类型	描述
node_name	text	节点的名称，包含主机和备机。
rto_info	text	流控的信息，包含了备机当前的日志流控时间（单位：秒），备机通过 GUC 参数设置的预期流控时间（单位：秒），为了达到这个预期主机所需要的睡眠时间（单位：微秒）。

26.3 WDR Snapshot Schema

WDR Snasphot 在启动后(打开参数 enable_wdr_snapshot)，会在用户表空间”pg_default”，数据库” postgres” 下新建 schema “snapshot”，用于持久化 WDR 快照数据。默认初始化用户或 monadmin 用户可以访问 Snapshot Schema。

根据参数 wdr_snapshot_retention_days 来自动管理快照的生命周期。

26.3.1 WDR Snapshot 原信息表

26.3.1.1.1 SNAPSHOT.SNAPSHOT

SNAPSHOT 表记录当前系统中存储的 WDR 快照数据的索引信息、开始时间和结束时间。只能在系统库中查询到结果，在用户库中无法查询。

名称	类型	描述	示例
snapshot_id	bigint	WDR 快照序号。	1
start_ts	timestamp	WDR 快照的开始时间。	2019-12-28 17:11:27.423742+08
end_ts	timestamp	WDR 快照的结束时间。	2019-12-28 17:11:43.67726+08

26.3.1.1.2 SNAPSHOT.TABLES_SNAP_TIMESTAMP

TABLES_SNAP_TIMESTAMP 表记录所有存储的 WDR snapshot 中数据库、表对象、以及数据采集的开始和结束时间。

名称	类型	描述	示例
snapshot_id	bigint	WDR 快照序号。	1
db_name	text	WDR snapshot 对应的 database。	tpcc1000
tablename	text	WDR snasphot 对应的 table。	snap_xc_statio_all_indexes
start_ts	timestamp	WDR 快照的开始时间。	2019-12-28 17:11:27.425849+08
end_ts	timestamp	WDR 快照的结束时间。	2019-12-28 17:11:27.707398+08

须知：

用户应该禁止对 Snapshot schema 下的表进行增删改等操作，人为对这些表的修改或破坏可能会导致 WDR 各种异常情况甚至 WDR 不可用。

26.3.1.1.3 SNAP_SEQ

snap_seq 是一个递增的 sequence，其为 WDR snapshot 提供快照的 ID。

26.3.2 WDR Snapshot 数据表

WDR Snapshot 数据表命名原则：snap_{源数据表}。

WDR Snapshot 数据表来源为 DBE_PERF Schema 下的视图。

26.3.3 WDR Snapshot 生成性能报告

基于 WDR Snapshot 数据表汇总、统计，生成性能报告，默认初始化用户或监控管理员用户可以生成报告，在 V8.8.500R001C20SPC003 之前的版本初始化用户或者 sysadmin 用户可以生成报告。

26.3.3.1.1 前提条件

WDR Snasphot 启动（即参数 enable_wdr_snapshot 为 on 时），且快照数量大于等于 2。

26.3.3.1.2 操作步骤

(1) 执行如下命令新建报告文件。

```
touch /home/data/wdrTestNode.html
```

(2) 执行以下命令连接 postgres 数据库。

```
gsql -d postgres -p 端口号 -r
```

(3) 执行如下命令查询已经生成的快照，以获取快照的 snapshot_id。

```
select * from snapshot.snapshot;
```

(4) （可选）执行如下命令手动创建快照。数据库中只有一个快照或者需要查看在当前时间段数据库的监控数据，可以选择手动执行快照操作，该命令需要用户具有 sysadmin 权限。

```
select create_wdr_snapshot();
```

说明：执行“cm_ctl query -Cdvi”，回显中“Central Coordinator State”下显示的信息即为 CCN 信息。

(5) 执行如下命令，在本地生成 HTML 格式的 WDR 报告。

执行如下命令，设置报告格式。\a：不显示表行列符号， \t：不显示列名 ，\o：指定输出文件。

```
gsql> \a
gsql> \t
gsql> \o /home/om/wdrTestNode.html
```

执行如下命令，生成 HTML 格式的 WDR 报告。

```
gsql> select generate_wdr_report(begin_snap_id Oid, end_snap_id Oid, int
report_type, int report_scope, int node_name );
```

示例一，生成集群级别的报告：

```
select generate_wdr_report(1, 2, 'all', 'cluster',null);
```

示例二，生成某个节点的报告：

```
select generate_wdr_report(1, 2, 'all', 'node', pgxc_node_str()::cstring);
```

说明：当前 GBase 8s 的节点名固定是“dn_6001_6002_6003”，也可直接代入。

表 26-1 generate_wdr_report 函数参数说明

参数	说明	取值范围
begin_snap_id	查询时间段开始的 snapshot 的 id（表 snapshot.snaoshot 中的 snapshot_id）。	-
end_snap_id	查询时间段结束 snapshot 的 id。默认 end_snap_id 大于 begin_snap_id （表 snapshot.snaoshot 中的 snapshot_id）。	-
report_type	指定生成 report 的类型。例如，summary/detail/all。	summary：汇总数据。 detail：明细数据。 all：包含 summary 和 detail。
report_scope	指定生成 report 的范围，可以为 cluster 或者 node。	cluster：数据库级别的信息。 node：节点级别的信息。
node_name	在 report_scope 指定为 node 时，需要把该参数指定为对应节点的名称。（节点名称可以执行 select * from	node：GBase 8s 中的节点名称。 cluster：省略/空/NULL。

参数	说明	取值范围
	pg_node_env;查询）。 在 report_scope 为 cluster 时，该值可以省略或者指定为空或 NULL。	

执行如下命令关闭输出选项及格式化输出命令。

```
\o \a \t
```

(6) 在/home/data/下根据需要查看 WDR 报告。

26.3.3.1.3 示例

```
--创建报告文件
touch  /home/data/wdrTestNode.html

--连接数据库
gsql -d postgres -p 15400 -r

--查询已经生成的快照。
postgres=# select * from snapshot.snapshot;
 snapshot_id |           start_ts            |            end_ts
-------------+-------------------------------+-------------------------------
           1 | 2023-09-07 10:20:36.763244+08 | 2023-09-07 10:20:42.166511+08
           2 | 2023-09-07 10:21:13.416352+08 | 2023-09-07 10:21:19.470911+08
(2 rows)


--生成格式化性能报告 wdrTestNode.html。
postgres=# \a \t \o /home/om/wdrTestNode.html
Output format is unaligned.
Showing only tuples.

--向性能报告 wdrTestNode.html 中写入数据。
postgres=# select generate_wdr_report(1, 2, 'all', 'node', 'dn_6001_6002_6003');

--关闭性能报告 wdrTestNode.html。
postgres=# \o
```

```
--生成格式化性能报告 wdrTestCluster.html。
postgres=# \o /home/om/wdrTestCluster.html

--向格式化性能报告 wdrTestCluster.html 中写入数据。
postgres=# select generate_wdr_report(1, 2, 'all', 'cluster');

--关闭性能报告 wdrTestCluster.html。
postgres=# \o \a \t
Output format is aligned.
Tuples only is off.
```

26.3.4 查看 WDR 报告

26.3.4.1 Database Stat

Database Stat 列名称及描述如下表所示。

列名称	描述
DB Name	数据库名称。
Backends	连接到该数据库的后端数。
Xact commit	此数据库中已经提交的事务数。
Xact Rollback	此数据库中已经回滚的事务数。
Blks Read	在这个数据库中读取的磁盘块的数量。
Blks Hit	高速缓存中已经发现的磁盘块的次数。
Tuple Returned	顺序扫描的行数。
Tuple Fetched	随机扫描的行数。

列名称	描述
Tuple Inserted	通过数据库查询插入的行数。
Tuple Updated	通过数据库查询更新的行数。
Tup Deleted	通过数据库查询删除的行数。
Conflicts	由于数据库恢复冲突取消的查询数量。
Temp Files	通过数据库查询创建的临时文件数量。
Temp Bytes	通过数据库查询写入临时文件的数据总量。
Deadlocks	在该数据库中检索的死锁数。
Blk Read Time	通过数据库后端读取数据文件块花费的时间，以毫秒计算。
Blk Write Time	通过数据库后端写入数据文件块花费的时间，以毫秒计算。
Stats Reset	重置当前状态统计的时间。

26.3.4.2 Load Profile

Load Profile 指标名称及描述如下表所示。

指标名称	描述
DB Time(us)	作业运行的 elapse time 总和。

指标名称	描述
CPU Time(us)	作业运行的 CPU 时间总和。
Redo size(blocks)	产生的 WAL 的大小（块数）。
Logical read (blocks)	表或者索引文件的逻辑读（块数）。
Physical read (blocks)	表或者索引的物理读（块数）。
Physical write (blocks)	表或者索引的物理写（块数）。
Read IO requests	表或者索引的读次数。
Write IO requests	表或者索引的写次数。
Read IO (MB)	表或者索引的读大小（MB）。
Write IO (MB)	表或者索引的写大小（MB）。
Logons	登录次数。
Executes (SQL)	SQL 执行次数。
Rollbacks	回滚事务数。
Transactions	事务数。
SQL response time P95(us)	95%的 SQL 的响应时间。

指标名称	描述
SQL response time P80(us)	80%的 SQL 的响应时间。

26.3.4.3 Instance Efficiency Percentages

Instance Efficiency Percentages 指标名称及描述如下表所示。

指标名称	描述
Buffer Hit %	Buffer Pool 命中率。
Effective CPU %	CPU time 占 DB time 的比例。
WalWrite NoWait %	访问 WAL Buffer 的 event 次数占总 wait event 的比例。
Soft Parse %	软解析的次数占总的解析次数的比例。
Non-Parse CPU %	非 parse 的时间占执行总时间的比例。

26.3.4.4 Top 10 Events by Total Wait Time

Top 10 Events by Total Wait Time 列名称及描述如下表所示。

列名称	描述
Event	Wait Event 名称。
Waits	wait 次数。

列名称	描述
Total Wait Time(us)	总 wait 时间（微秒）。
Avg Wait Time(us)	平均 wait 时间（微秒）。
Type	Wait Event 类别。

26.3.4.5 Wait Classes by Total Wait Time

Wait Classes by Total Wait Time 列名称及描述如下表所示。

列名称	描述
Type	Wait Event 类别名称： STATUS。 LWLOCK_EVENT。 LOCK_EVENT。 IO_EVENT。
Waits	Wait 次数。
Total Wait Time(us)	总 Wait 时间（微秒）。
Avg Wait Time(us)	平均 Wait 时间（微秒）。

26.3.4.6 Host CPU

Host CPU 列名称及描述如下表所示。

列名称	描述
Cpus	CPU 数量。
Cores	CPU 核数。
Sockets	CPU Sockets 数量。
Load Average Begin	开始 snapshot 的 Load Average 值。
Load Average End	结束 snapshot 的 Load Average 值。
%User	用户态在 CPU 时间上的占比。
%System	内核态在 CPU 时间上的占比。
%WIO	Wait IO 在 CPU 时间上的占比。
%Idle	空闲时间在 CPU 时间上的占比。

26.3.4.7 IO Profile

IO Profile 指标名称及描述如下表所示。

指标名称	描述
Database requests	Database IO 次数。
Database (MB)	Database IO 数据量。
Database (blocks)	Database IO 数据块。
Redo requests	Redo IO 次数。
Redo (MB)	Redo IO 量。

26.3.4.8 Memory Statistics

Memory Statistics 指标名称及描述如下表所示。

指标名称	描述
shared_used_memory	已经使用共享内存大小（MB）。
max_shared_memory	最大共享内存（MB）。
process_used_memory	进程已经使用内存（MB）。
max_process_memory	最大进程内存（MB）。

26.3.4.9 Time Model

Time Model 名称及描述如下表所示。

名称	描述
DB_TIME	所有线程端到端的墙上时间（WALL TIME）消耗总和（单位：微秒）。
EXECUTION_TIME	消耗在执行器上的时间总和（单位：微秒)。
PL_EXECUTION_TIME	消耗在 PL/SQL 执行上的时间总和（单位：微秒）。
CPU_TIME	所有线程 CPU 时间消耗总和（单位：微秒）。
PLAN_TIME	消耗在执行计划生成上的时间总和（单位：微秒）。
REWRITE_TIME	消耗在查询重写上的时间总和（单位：微秒）。
PL_COMPILATION_TIME	消耗在 SQL 编译上的时间总和（单位：微秒）。
PARSE_TIME	消耗在 SQL 解析上的时间总和（单位：微秒）。
NET_SEND_TIME	消耗在网络发送上的时间总和（单位：微秒）。
DATA_IO_TIME	消耗在数据读写上的时间总和（单位：微秒）。

26.3.4.10 SQL Statistics

SQL Statistics 列名称及描述如下表所示。

列名称	描述

列名称	描述
Unique SQL Id	归一化的 SQL ID。
Node Name	节点名称。
User Name	用户名称。
Tuples Read	访问的元组数量。
Calls	调用次数。
Min Elapse Time(us)	最小执行时间（us）。
Max Elapse Time(us)	最大执行时间（us）。
Total Elapse Time(us)	总执行时间（us）。
Avg Elapse Time(us)	平均执行时间（us）。
Returned Rows	SELECT 返回行数。
Tuples	Insert/Update/Delete 行数。

列名称	描述
Affected	
Logical Read	Buffer 逻辑读次数。
Physical Read	Buffer 物理读次数。
CPU Time(us)	CPU 时间（us）。
Data IO Time(us)	IO 上的时间花费（us）。
Sort Count	排序执行的次数。
Sort Time(us)	排序执行的时间（us）。
Sort Mem Used(KB)	排序过程中使用的 work memory 大小（KB）。
Sort Spill Count	排序过程中，若发生落盘，写文件的次数。
Sort Spill Size(KB)	排序过程中，若发生落盘，使用的文件大小（KB）。

列名称	描述
Hash Count	hash 执行的次数。
Hash Time(us)	hash 执行的时间（us）。
Hash Mem Used(KB)	hash 过程中使用的 work memory 大小（KB）。
Hash Spill Count	hash 过程中，若发生落盘，写文件的次数。
Hash Spill Size(KB)	hash 过程中，若发生落盘，使用的文件大小（KB）。
SQL Text	归一化 SQL 字符串。

26.3.4.11 Wait Events

Wait Events 列名称及描述如下表所示。

列名称	描述
Type	Wait Event 类别名称： STATUS。 LWLOCK_EVENT。 LOCK_EVENT。 IO_EVENT。

列名称	描述
Event	Wait Event 名称。
Total Wait Time (us)	总 Wait 时间（us）。
Waits	总 Wait 次数。
Failed Waits	Wait 失败次数。
Avg Wait Time (us)	平均 Wait 时间（us）。
Max Wait Time (us)	最大 Wait 时间（us）。

26.3.4.12 Cache IO Stats

Cache IO Stats 包含 User table 和 User index 两张表，列名称及描述如下所示。

26.3.4.12.1 User table IO activity ordered by heap blks hit ratio

列名称	描述
DB Name	Database 名称。
Schema Name	Schema 名称。
Table Name	Table 名称。

列名称	描述
%Heap Blks Hit Ratio	此表的 Buffer Pool 命中率。
Heap Blks Read	该表中读取的磁盘块数。
Heap Blks Hit	此表缓存命中数。
Idx Blks Read	表中所有索引读取的磁盘块数。
Idx Blks Hit	表中所有索引命中缓存数。
Toast Blks Read	此表的 TOAST 表读取的磁盘块数（如果存在）。
Toast Blks Hit	此表的 TOAST 表命中缓冲区数（如果存在）。
Tidx Blks Read	此表的 TOAST 表索引读取的磁盘块数（如果存在）。
Tidx Blks Hit	此表的 TOAST 表索引命中缓冲区数（如果存在）。

26.3.4.12.2 User index IO activity ordered by idx blks hit ratio

列名称	描述
DB Name	Database 名称。
Schema Name	Schema 名称。

列名称	描述
Table Name	Table 名称。
Index Name	Index 名称。
%Idx Blks Hit Ratio	Index 的命中率。
Idx Blks Read	所有索引读取的磁盘块数。
Idx Blks Hit	所有索引命中缓存数。

26.3.4.13 Utility status

Utility status 包含 Replication slot 和 Replication stat 两张表，列名称及描述如下所示。

26.3.4.13.1 Replication slot

列名称	描述
Slot Name	复制节点名。
Slot Type	复制节点类型。
DB Name	复制节点数据库名称。
Active	复制节点状态。
Xmin	复制节点事务标识。

列名称	描述
Restart Lsn	复制节点的 Xlog 文件信息。
Dummy Standby	复制节点假备。

26.3.4.13.2 Replication stat

列名称	描述
Thread Id	线程的 PID。
Usesys Id	用户系统 ID。
Usename	用户名称。
Application Name	应用程序。
Client Addr	客户端地址。
Client Hostname	客户端主机名。
Client Port	客户端端口。
Backend Start	程序起始时间。
State	日志复制状态。
Sender Sent Location	发送端发送日志位置。

列名称	描述
Receiver Write Location	接收端 write 日志位置。
Receiver Flush Location	接收端 flush 日志位置。
Receiver Replay Location	接收端 replay 日志位置。
Sync Priority	同步优先级。
Sync State	同步状态。

26.3.4.14 Object stats

Object stats 包含 User Tables stats、User index stats 和 Bad lock stats 三张表，列名称及描述如下所示。

26.3.4.14.1 User Tables stats

列名称	描述
DB Name	Database 名称。
Schema	Schema 名称。
Relname	Relation 名称。
Seq Scan	此表发起的顺序扫描数。
Seq Tup Read	顺序扫描抓取的活跃行数。

列名称	描述
Index Scan	此表发起的索引扫描数。
Index Tup Fetch	索引扫描抓取的活跃行数。
Tuple Insert	插入行数。
Tuple Update	更新行数。
Tuple Delete	删除行数。
Tuple Hot Update	HOT 更新行数（即没有更新所需的单独索引）。
Live Tuple	估计活跃行数。
Dead Tuple	估计死行数。
Last Vacuum	最后一次此表是手动清理的（不计算 VACUUM FULL）时间。
Last Autovacuum	上次被 autovacuum 守护进程清理的时间。
Last Analyze	上次手动分析这个表的时间。
Last Autoanalyze	上次被 autovacuum 守护进程分析的时间。

列名称	描述
Vacuum Count	这个表被手动清理的次数（不计算 VACUUM FULL）。
Autovacuum Count	这个表被 autovacuum 清理的次数。
Analyze Count	这个表被手动分析的次数。
Autoanalyze Count	这个表被 autovacuum 守护进程分析的次数。

26.3.4.14.2 User index stats

列名称	描述
DB Name	Database 名称。
Schema	Schema 名称。
Relname	Relation 名称。
Index Relname	Index 名称。
Index Scan	索引上开始的索引扫描数。
Index Tuple Read	通过索引上扫描返回的索引项数。

列名称	描述
Index Tuple Fetch	通过使用索引的简单索引扫描抓取的表行数。

26.3.4.14.3 Bad lock stats

列名称	描述
DB Id	数据库的 OID。
Tablespace Id	表空间的 OID。
Relfilenode	文件对象 ID。
Fork Number	文件类型。
Error Count	失败计数。
First Time	第一次发生时间。
Last Time	最近一次发生时间。

26.3.4.14.4 Configuration settings

Configuration settings 列名称及描述如下表所示。

列名称	描述
Name	GUC 名称。

列名称	描述
Abstract	GUC 描述。
Type	数据类型。
Curent Value	当前值。
Min Value	合法最小值。
Max Value	合法最大值。
Category	GUC 类别。
Enum Values	如果是枚举值，列举所有枚举值。
Default Value	数据库启动时参数默认值。
Reset Value	数据库重置时参数默认值。

26.3.4.14.5 SQL Detail

SQL Detail 列名称及描述如下表所示。

列名称	描述
Unique SQL Id	归一化 SQL ID。

列名称	描述
User Name	用户名称。
Node Name	节点名称。Node 模式下不显示该字段。
SQL Text	归一化 SQL 文本。

26.4 DBE_PLDEBUGGER Schema

DBE_PLDEBUGGER 下系统函数用于单机下调试存储过程，目前支持的接口及其描述如下所示。仅管理员有权限执行这些调试接口。

须知：当在函数体中创建用户时，调用 attach、next、continue、 info_code、step、info_breakpoint、backtrace、 finish 中会返回密码的明文。因此不建议用户在函数体中创建用户。

对应权限角色为 gs_role_pldebugger，可以由管理员用户通过如下命令将 debugger 权限赋权给该用户。

```
GRANT gs_role_pldebugger to user;
```

需要有两个客户端连接数据库，一个客户端负责执行调试接口作为 debug 端，另一个客户端执行调试函数，控制 server 端存储过程执行。示例如下。

(1) 准备调试

通过 PG_PROC，查找到待调试存储过程的 oid，并执行 DBE_PLDEBUGGER.turn_on(oid)。本客户端就会作为 server 端使用。

```
postgres=# CREATE OR REPLACE PROCEDURE test_debug ( IN  x INT)
AS
BEGIN
    INSERT INTO t1 (a) VALUES (x);
    DELETE FROM t1 WHERE a = x;
END;
/
CREATE PROCEDURE
postgres=# SELECT OID FROM PG_PROC WHERE PRONAME='test_debug';
```

```
 oid
-------
 16389
(1 row)
postgres=# SELECT * FROM DBE_PLDEBUGGER.turn_on(16389);
 nodename | port
----------+------
 datanode |    0
(1 row)
```

(2) 开始调试

server 端执行存储过程，会在存储过程内第一条 SQL 语句前 hang 住，等待 debug 端发送的调试消息。仅支持直接执行存储过程的调试，不支持通过 trigger 调用执行的存储过程调试。

```
postgres=# call test_debug(1);
```

再起一个客户端，作为 debug 端，通过 turn_on 返回的数据，调用 DBE_PLDEBUGGER.attach 关联到该存储过程上进行调试。

```
postgres=# SELECT * FROM DBE_PLDEBUGGER.attach('datanode',0);
 funcoid |  funcname  | lineno |              query
---------+------------+--------+---------------------------------
   16389 | test_debug |      3 |   INSERT INTO t1 (a) VALUES (x);
(1 row)
```

在执行 attach 的客户端调试，执行下一条 statement。

```
postgres=# SELECT * FROM DBE_PLDEBUGGER.next();
 funcoid |  funcname  | lineno |        query
---------+------------+--------+---------------------
   16389 | test_debug |      0 | [EXECUTION FINISHED]
(1 row)
```

在执行 attach 的客户端调试，可以执行以下变量操作

```

postgres=# SELECT * FROM DBE_PLDEBUGGER.info_locals(); --打印全部变量
 varname | vartype | value | package_name | isconst
---------+---------+-------+--------------+---------
 x       | int4    | 1     |              | f
(1 row)
postgres=# SELECT * FROM DBE_PLDEBUGGER.set_var('x', 2); --变量赋值
 set_var
```

```
---------
 t
(1 row)
postgres=# SELECT * FROM DBE_PLDEBUGGER.print_var('x'); --打印单个变量
 varname | vartype | value | package_name | isconst
---------+---------+-------+--------------+---------
 x       | int4    | 2     |              | f
(1 row)
```

直接执行完成当前正在调试的存储过程。

```
postgres=# SELECT * FROM DBE_PLDEBUGGER.continue();
 funcoid |  funcname   | lineno |        query
---------+-------------+--------+---------------------
   16389 | test_debug |      0 | [EXECUTION FINISHED]
(1 row)
```

直接退出当前正在调试的存储过程，不执行尚未执行的语句。

```
postgres=# SELECT * FROM DBE_PLDEBUGGER.abort();
 abort
-------
 t
(1 row)
```

client 端查看代码信息并识别可以设置断点行号。

```
postgres=# SELECT * FROM DBE_PLDEBUGGER.info_code(16389);
 lineno |                            query                             | canbreak
--------+--------------------------------------------------------------+---------
--
        | CREATE OR REPLACE PROCEDURE public.test_debug( IN  x INT) | f
      1 | AS  DECLARE                                                  | f
      2 | BEGIN                                                        | f
      3 |     INSERT INTO t1 (a) VALUES (x);                           | t
      4 |     DELETE FROM t1 WHERE a = x;                              | t
      5 | END;                                                         | f
      6 | /                                                            | f
(7 rows)
```

(3) 设置断点。

```
postgres=# SELECT * FROM DBE_PLDEBUGGER.add_breakpoint(16389,4);
```

```
 lineno |                            query                             | canbreak
--------+--------------------------------------------------------------+--------
--
        | CREATE OR REPLACE PROCEDURE public.test_debug( IN  x INT) | f
      1 | AS  DECLARE                                                  | f
      2 | BEGIN                                                        | f
      3 |     INSERT INTO t1 (a) VALUES (x);                           | t
      4 |     DELETE FROM t1 WHERE a = x;                              | t
      5 | END;                                                         | f
      6 | /                                                            | f
(7 rows)
```

查看断点信息。

```
postgres=# SELECT * FROM DBE_PLDEBUGGER.info_breakpoints();
 breakpointno | funcoid | lineno |              query               | enable
--------------+---------+--------+----------------------------------+--------
            0 |   16389 |      4 |     DELETE FROM t1 WHERE a = x; | t
(1 row)
```

执行至断点。

```
postgres=# SELECT * FROM DBE_PLDEBUGGER.continue();
 funcoid |  funcname   | lineno |              query
---------+-------------+--------+---------------------------------
   16389 | test_debug |      4 |     DELETE FROM t1 WHERE a = x;
(1 row)
```

存储过程执行结束后，调试会自动退出，再进行调试需要重新 attach 关联。如果 server 端不需要继续调试，可执行 turn_off 关闭，或退出 session。具体调试接口请见下面列表。

表 26-2 DBE_PLDEBUGGER

接口名称	描述
DBE_PLDEBUGGER.turn_on	server 端调用，标记存储过程可以调试，调用后执行该存储过程时会 hang 住等待调试信息。
DBE_PLDEBUGGER.turn_off	server 端调用，标记存储过程关闭调试。

接口名称	描述
DBE_PLDEBUGGER.local_debug_server_info	server 端调用，打印本 session 内所有已 turn_on 的存储过程。
DBE_PLDEBUGGER.attach	debug 端调用，关联到正在调试存储过程。
DBE_PLDEBUGGER.info_locals	debug 端调用，打印正在调试的存储过程中的变量当前值。
DBE_PLDEBUGGER.next	debug 端调用，单步执行。
DBE_PLDEBUGGER.continue	debug 端调用，继续执行，直到断点或存储过程结束。
DBE_PLDEBUGGER.abort	debug 端调用，停止调试，server 端报错长跳转。
DBE_PLDEBUGGER.print_var	debug 端调用，打印正在调试的存储过程中指定的变量当前值。
DBE_PLDEBUGGER.info_code	debug 和 server 端都可以调用，打印指定存储过程的源语句和各行对应的行号。
DBE_PLDEBUGGER.step	debug 端调用，单步进入执行。
DBE_PLDEBUGGER.add_breakpoint	debug 端调用，新增断点。
DBE_PLDEBUGGER.delete_breakpoint	debug 端调用，删除断点。

接口名称	描述
DBE_PLDEBUGGER.info_breakpoints	debug 端调用，查看当前的所有断点。
DBE_PLDEBUGGER.backtrace	debug 端调用，查看当前的调用栈。
DBE_PLDEBUGGER.enable_breakpoint	debug 端调用，激活被禁用的断点。
DBE_PLDEBUGGER.disable_breakpoint	debug 端调用，禁用已激活的断点。
DBE_PLDEBUGGER.finish	debug 端调用，继续调试，直到断点或返回上一层调用栈。
DBE_PLDEBUGGER.set_var	debug 端调用，为变量进行赋值操作。

26.4.1 DBE_PLDEBUGGER.info_breakpoints

debug 端调试过程中，调用 info_breakpoints，查看当前的函数断点。

名称	类型	描述
breakpointno	OUT integer	断点编号。
funcoid	OUT oid	函数 ID。
lineno	OUT integer	行号。
query	OUT text	断点内容。

名称	类型	描述
enable	OUT boolean	是否有效

26.4.2 DBE_PLDEBUGGER.backtrace

debug 端调试过程中，调用 backtrace，查看当前的调用堆栈。

名称	类型	描述
frameno	OUT integer	调用栈编号。
funcname	OUT oid	函数名。
lineno	OUT integer	行号。
query	OUT text	断点内容。
funcoid	OUT oid	函数 oid。

26.4.3 DBE_PLDEBUGGER.turn_on

该函数用于标记某一存储过程为可调试，执行 turn_on 后 server 端可以执行该存储过程来进行调试。需要用户根据系统表 PG_PROC 手动获取存储过程 oid，传入函数中。turn_on 后本 session 内执行该存储过程会停在第一条 sql 前等待 debug 端的调试操作。该设置会在 session 断连后默认被清理掉。目前不支持对启用自治事务的存储过程/函数进行调试。

函数原型为：

```
DBE_PLDEBUGGER.turn_on(Oid)
RETURN Record;
```

表 26-3 turn_on 入参和返回值列表

名称	类型	描述
func_oid	IN oid	函数 oid。
nodename	OUT text	节点名称。
port	OUT integer	连接端口号。

26.4.4 DBE_PLDEBUGGER.turn_off

用于去掉 turn_on 添加的调试标记，返回值表示成功或失败。可通过 DBE_PLDEBUGGER.local_debug_server_info 查找已经 turn_on 的存储过程 oid。

函数原型为：

DBE_PLDEBUGGER.turn_off(Oid)

RETURN boolean;

名称	类型	描述
func_oid	IN oid	函数 oid。
turn_off	OUT boolean	turn off 是否成功。

26.4.5 DBE_PLDEBUGGER.local_debug_server_info

用于查找当前连接中已经 turn_on 的存储过程 oid。便于用户确认在调试哪些存储过程，需要通过 funcoid 和 pg_proc 配合使用。

名称	类型	描述
nodename	OUT text	节点名称。

名称	类型	描述
port	OUT bigint	端口号。
funcoid	OUT oid	存储过程 oid。

26.4.6 DBE_PLDEBUGGER.attach

server 端执行存储过程，停在第一条语句前，等待 debug 端关联。debug 端调用 attach，传入 nodename 和 port，关联到该存储过程上。

如果调试过程中报错，attach 会自动失效；如果调试过程中 attach 到其他存储过程上，当前 attach 的调试也会失效。

表 26-4 attach 入参和返回值列表

名称	类型	描述
nodename	IN text	节点名称。
port	IN integer	连接端口号。
funcoid	OUT oid	函数 id。
funcname	OUT text	函数名。
lineno	OUT integer	当前调试运行的下一行行号。
query	OUT text	当前调试的下一行函数源码。

26.4.7 DBE_PLDEBUGGER.next

执行存储过程中当前的 sql，返回执行的下一条的行数和对应 query。

表 26-5 next 返回值列表

名称	类型	描述
funcoid	OUT oid	函数 id。
funcname	OUT text	函数名。
lineno	OUT integer	当前调试运行的下一行行号。
query	OUT text	当前调试的下一行函数源码。

26.4.8 DBE_PLDEBUGGER.continue

执行当前存储过程，直到下一个断点或结束。返回值表示执行的下一条的行数和对应 query。

函数原型为：

DBE_PLDEBUGGER.continue()

RETURN Record;

表 26-6 continue 返回值列表

名称	类型	描述
funcoid	OUT oid	函数 id。
funcname	OUT text	函数名。

名称	类型	描述
lineno	OUT integer	当前调试运行的下一行行号。
query	OUT text	当前调试的下一行函数源码。

26.4.9 DBE_PLDEBUGGER.abort

令 server 端执行的存储过程报错跳出。返回值表示是否成功发送 abort。

函数原型为：

DBE_PLDEBUGGER.abort()

RETURN boolean;

表 26-7 abort 返回值列表

名称	类型	描述
abort	OUT boolean	表示成功或失败。

26.4.10DBE_PLDEBUGGER.print_var

debug 端调试过程中，调用 print_var，打印当前存储过程内变量中指定的变量名及其取值。该函数入参 frameno 表示查询遍历的栈层数，支持不加入该参数调用，缺省为查看最上层栈变量。

表 26-8 print_var 入参和返回值列表

名称	类型	描述
var_name	IN text	变量。

名称	类型	描述
frameno	IN integer（可选）	指定的栈层数，缺省为最顶层。
varname	OUT text	变量名。
vartype	OUT text	变量类型。
value	OUT text	变量值。
package_name	OUT text	变量对应的 package 名，预留使用，当前均为空。
isconst	OUT boolean	是否为常量。

26.4.11 DBE_PLDEBUGGER.info_code

debug 端调试过程中，调用 info_code，查看指定存储过程的源语句和各行对应的行号，行号从函数体开始，函数头部分行号为空。

表 26-9 info_code 入参和返回值列表

名称	类型	描述
funcoid	IN oid	函数 ID。
lineno	OUT integer	行号。
query	OUT text	源语句。

名称	类型	描述
canbreak	OUT bool	当前行是否支持断点。

26.4.12DBE_PLDEBUGGER.step

debug 端调试过程中，如果当前执行的是一个存储过程，则进入该存储过程继续调试，返回该存储过程第一行的行号等信息，如果当前执行的不是存储过程，则和 next 行为一致，执行该 sql 后返回下一行的行号等信息。

表 26-10 step 入参和返回值列表

名称	类型	描述
funcoid	OUT oid	函数 id。
funcname	OUT text	函数名。
lineno	OUT integer	当前调试运行的下一行行号。
query	OUT text	当前调试的下一行函数源码。

26.4.13DBE_PLDEBUGGER.delete_breakpoint

debug 端调试过程中，调用 delete_breakpoint 删除已有的断点。

表 26-11 delete_breakpoint 入参和返回值列表

名称	类型	描述
breakpointno	IN integer	断点编号。

名称	类型	描述
result	OUT bool	是否成功。

26.4.14DBE_PLDEBUGGER.info_breakpoints

debug 端调试过程中，调用 info_breakpoints，查看当前的函数断点。

表 26-12 info_breakpoints 返回值列表

名称	类型	描述
breakpointno	OUT integer	断点编号。
funcoid	OUT oid	函数 ID。
lineno	OUT integer	行号。
query	OUT text	断点内容。
enable	OUT boolean	是否有效

26.4.15DBE_PLDEBUGGER.backtrace

debug 端调试过程中，调用 backtrace，查看当前的调用堆栈。

表 26-13 backtrace 返回值列表

名称	类型	描述
frameno	OUT integer	调用栈编号。
funcname	OUT oid	函数名。

名称	类型	描述
lineno	OUT integer	行号。
query	OUT text	断点内容。
funcoid	OUT oid	函数 oid。

26.4.16DBE_PLDEBUGGER.enable_breakpoint

debug 端调试过程中，调用 enable_breakpoint 激活已被禁用的断点。

表 26-14 enable_breakpoint 入参和返回值列表

名称	类型	描述
breakpointno	IN integer	断点编号
result	OUT bool	是否成功

26.4.17DBE_PLDEBUGGER.add_breakpoint

debug 端调试过程中，调用 add_breakpoint 增加新的断点。如果返回-1 则说明指定的断点不合法，请参考 DBE_PLDEBUGGER.info_code 的 canbreak 字段确定断点合适的位置。

表 26-15 add_breakpoint 入参和返回值列表

名称	类型	描述
funcoid	IN text	函数 ID。
lineno	IN integer	行号。

名称	类型	描述
breakpointno	OUT integer	断点编号。

26.4.18DBE_PLDEBUGGER.disable_breakpoint

debug 端调试过程中，调用 disable_breakpoint 禁用已被激活的断点。

表 26-16 disable_breakpoint 入参和返回值列表

名称	类型	描述
breakpointno	IN integer	断点编号
result	OUT bool	是否成功

26.4.19DBE_PLDEBUGGER.finish

执行存储过程中当前的 SQL 直到下一个断点触发或执行到上层栈的下一行。

表 26-17 finish 入参和返回值列表

名称	类型	描述
funcoid	OUT oid	函数 id。
funcname	OUT text	函数名。
lineno	OUT integer	当前调试运行的下一行行号。
query	OUT text	当前调试的下一行函数源码。

26.4.20DBE_PLDEBUGGER.set_var

将指定的调试的存储过程中最上层栈上的变量修改为入参的取值。如果存储过程中包含同名的变量，set_var 只支持第一个变量值的设置。

表 26-18 set_var 入参和返回值列表

名称	类型	描述
var_name	IN text	变量名。
value	IN text	修改值。
result	OUT boolean	结果，是否成功。

26.5 DB4AI Schema

DB4AI 模式在 AI 特性中主要是用来存储和管理数据集版本。模式中保存数据表的原始视图快照，每一个数据版本的更改记录以及版本快照的管理信息。模式面向普通用户，用户可在该模式下查找特性 DB4AI.SNAPSHOT 创建的快照版本信息。

26.5.1 DB4AI.SNAPSHOT

SNAPSHOT 表记录当前用户通过特性 DB4AI.SNAPSHOT 存储的快照。

名称	类型	描述	实例
id	bigint	当前快照的 ID。	1
parent_id	bigint	父快照的 ID。	0
matrix_id	bigint	CSS 模式下快照的矩阵 ID，否则为 NULL。	0

名称	类型	描述	实例
root_id	bigint	初始快照的ID，通过db4ai.create_snapshot()从操作数据构建。	0
schema	name	导出快照视图的模式。	public
name	name	快照的名称，包括版本后缀。	example0@1.1.0
owner	name	创建此快照的用户的名称。	nw
commands	text[]	记录如何从其根快照生成到此快照的 SQL 语句的完整列表。	{DELETE,“WHERE id > 7”}
comment	text	快照说明。	inherits from @1.0.0
published	boolean	TRUE，当且仅当快照当前已发布。	f
archived	boolean	TRUE，当且仅当快照当前已存档。	f
created	timestamp without time zone	快照创建日期的时间戳。	2021-08-25 10:59:52.955604
row_count	bigint	此快照中的数据行数。	8

26.5.2 DB4AI.CREATE_SNAPSHOT

CREATE_SNAPSHOT 是 DB4AI 特性用于创建快照的接口函数。通过语法 CREATE

SNAPSHOT 调用。

参数	类型	描述
i_schema	IN NAME	快照存储的模式名字，默认值是当前用户或者 PUBLIC。
i_name	IN NAME	快照名称。
i_commands	IN TEXT[]	定义数据获取的 SQL 命令。
i_vers	IN NAME	版本后缀。
i_comment	IN TEXT	快照描述。
res	OUT db4ai.snapshot_name	结果。

26.5.3 DB4AI.CREATE_SNAPSHOT_INTERNAL

CREATE_SNAPSHOT_INTERNAL 是 db4ai.create_snapshot 函数的内置执行函数。函数存在信息校验，无法直接调用。

参数	类型	描述
s_id	IN BIGINT	快照 ID。
i_schema	IN NAME	快照存储的名字空间。
i_name	IN NAME	快照名称。

参数	类型	描述
i_commands	IN TEXT[]	定义数据获取的 SQL 命令。
i_comment	IN TEXT	快照描述。
i_owner	IN NAME	快照拥有者。

26.5.4 DB4AI.PREPARE_SNAPSHOT

PREPARE_SNAPSHOT 是 DB4AI 特性中数据准备模型训练和解释快照进行协作。快照为所有应用更改的数据和文档提供了完整的序列。通过语法 PREPARE SNAPSHOT 调用。

参数	类型	描述
i_schema	IN NAME	快照存储的模式名字，默认值是当前用户或者 PUBLIC。
i_parent	IN NAME	父快照名称。
i_commands	IN TEXT[]	定义快照修改的 DDL 和 DML 命令。
i_vers	IN NAME	版本后缀。
i_comment	IN TEXT	此数据策展单元的说明。
res	OUT db4ai.snapshot_name	结果。

26.5.5 DB4AI.PREPARE_SNAPSHOT_INTERNAL

PREPARE_SNAPSHOT_INTERNAL 是 db4ai.prepare_snapshot 函数的内置执行函数。函数存在信息校验，无法直接调用。

参数	类型	描述
s_id	IN BIGINT	快照 ID。
p_id	IN BIGINT	父快照 ID。
m_id	IN BIGINT	矩阵 id。
r_id	IN BIGINT	根快照 ID。
i_schema	IN NAME	快照模式。
i_name	IN NAME	快照名称。
i_commands	IN TEXT[]	定义快照修改的 DDL 和 DML 命令。
i_comment	IN TEXT	快照描述。
i_owner	IN NAME	快照所有者。
i_idx	INOUT INT	exec_cmds 的索引。
i_exec_cmds	INOUT TEXT[]	用于执行的 DDL 和 DML。
i_mapping	IN NAME[]	将用户列映射到备份列；如

参数	类型	描述
		果不为 NULL，则生成规则。

26.5.6 DB4AI.ARCHIVE_SNAPSHOT

ARCHIVE_SNAPSHOT 是 DB4AI 特性用于存档快照的接口函数。通过语法 ARCHIVE SNAPSHOT 调用。生效后的快照无法参数训练等任务。

参数	类型	描述
i_schema	IN NAME	快照存储的模式名字，默认值是当前用户
i_name	IN NAME	快照名称
res	OUT db4ai.snapshot_name	结果

26.5.7 DB4AI.PUBLISH_SNAPSHOT

PUBLISH_SNAPSHOT 是 DB4AI 特性用于发布快照的接口函数。通过语法 PUBLISH SNAPSHOT 调用。

参数	类型	描述
i_schema	IN NAME	快照存储的模式名字，默认值是当前用户或者 PUBLIC
i_name	IN NAME	快照名称
res	OUT db4ai.snapshot_name	结果

26.5.8 DB4AI.MANAGE_SNAPSHOT_INTERNAL

MANAGE_SNAPSHOT_INTERNAL 是 DB4AI.PUBLISH_SNAPSHOT 和 DB4AI.ARCHIVE_SNAPSHOT 函数的内置执行函数。函数存在信息校验，无法直接调用。

表 26-19 DB4AI.MANAGE_SNAPSHOT_INTERNAL 入参和返回值列表

参数	类型	描述
i_schema	IN NAME	快照存储的模式名字
i_name	IN NAME	快照名称
publish	IN BOOLEN	是否是发布状态
res	OUT db4ai.snapshot_name	结果

26.5.9 DB4AI.SAMPLE_SNAPSHOT

SAMPLE_SNAPSHOT 是 DB4AI 特性用于对基数据进行采样生成快照的接口函数。通过语法 SAMPLE SNAPSHOT 调用。

参数	类型	描述
i_schema	IN NAME	快照存储的模式名字
i_parent	IN NAME	父快照名称
i_sample_infixes	IN NAME[]	示例快照名称中缀
i_sample_ratios	IN NUMBER[]	每个样本的大小，作为父集的比率

参数	类型	描述
i_stratify	IN NAME[]	分层策略
i_sample_comments	IN TEXT[]	示例快照描述
res	OUT db4ai.snapshot_name	结果

26.5.10 DB4AI.PURGE_SNAPSHOT

PURGE_SNAPSHOT 是 DB4AI 特性用于删除快照的接口函数。通过语法 PURGE SNAPSHOT 调用。

参数	类型	描述
i_schema	IN NAME	快照存储的模式名字
i_name	IN NAME	快照名称
res	OUT db4ai.snapshot_name	结果

26.5.11 DB4AI.PURGE_SNAPSHOT_INTERNAL

PURGE_SNAPSHOT_INTERNAL 是 DB4AI.PURGE_SNAPSHOT 函数的内置执行函数。函数存在信息校验，无法直接调用

参数	类型	描述
i_schema	IN NAME	快照存储的模式名字
i_name	IN NAME	快照名称

26.6 DBE_PLDEVELOPER

DBE_PLDEVELOPER 下系统表用于记录 PLPGSQL 包、函数及存储过程编译过程中需要记录的信息。

26.6.1 DBE_PLDEVELOPER.gs_source

用于记录 PLPGSQL 对象（存储过程、函数、包、包体）编译相关信息，具体内容见下列字段描述。

打开 plsql_show_all_error 参数后，会把成功或失败的 PLPGSQL 对象编译信息记录在此表中，如果关闭 plsql_show_all_error 参数则只会将正确的编译相关信息插入此表中。

表 26-20 DBE_PLDEVELOPER.gs_source 字段

名称	类型	描述
id	oid	对象的 ID。
owner	bigint	对象创建用户 ID。
nspid	oid	对象的模式 ID。
name	name	对象名。
type	text	对象类型（procedure/function/package/package body）。
status	boolean	是否创建成功。
src	text	对象创建的原始语句。

26.6.2 DBE_PLDEVELOPER.gs_errors

用于记录 PLPGSQL 对象（存储过程、函数、包、包体）编译过程中遇到的报错信息，具体内容见下列字段描述。

打开 plsql_show_all_error 参数后，如果编译过程中存在报错，则会跳过报错继续编译并把报错信息记录在 gs_errors 中，如果关闭 plsql_show_all_error 参数则不会将相关信息插入此表中。

名称	类型	描述
id	oid	对象的 ID。
owner	bigint	对象创建用户 ID。
nspid	oid	对象的模式 ID。
name	name	对象名。
type	text	对象类型（procedure/function/package/package body）。
line	integer	行号。
src	text	对象创建的原始语句。

26.7 DBE_SQL_UTIL Schema

DBE_SQL_UTIL 模式存储了用于管理 SQL PATCH 的工具，包括创建、删除、开启、禁用 SQL PATCH 等系统函数。普通用户只有 usage 权限，没有 create、alter、drop、comment 等权限。

26.7.1 DBE_SQL_UTIL.create_hint_sql_patch

create_hint_sql_ patch 是用于创建调优 SQL PATCH 的接口函数，返回执行是否成功。限制仅初始用户、 sysadmin、opradmin、monadmin 用户有权限调用。

表 26-21 DBE_SQL_UTIL.create_hint_sql_ patch 入参和返回值列表

参数	类型	描述

patch_ name	IN name	PATCH 名称。
unique_sql_id	IN bigint	查询全局唯一 ID。
hint_string	IN text	Hint 文本。
AboutopenGauss	IN text	PATCH 的备注，默认值为 NULL。
enabled	IN bool	PATCH 是否生效，默认值为 true。

26.7.2 DBE_SQL_UTIL.create_abort_sql_patch

create_abort_sql_ patch 是用于创建避险 SQL PATCH 的接口函数，返回执行是否成功。限制仅初始用户、 sysadmin、opradmin、monadmin 用户有权限调用。

表 26-22 DBE_SQL_UTIL.create_abort_sql_ patch 入参和返回值列表

参数	类型	描述
patch_ name	IN name	PATCH 名称。
unique_sql_id	IN bigint	查询全局唯一 ID。
AboutopenGauss	IN text	PATCH 的备注，默认值为 NULL。
enabled	IN bool	PATCH 是否生效，默认值为 true。
result	OUT bool	执行是否成功。

26.7.3 DBE_SQL_UTIL.drop_sql_patch

drop_sql_ patch 是用于删除 SQL PATCH 的接口函数，返回执行是否成功。 限制仅初始用户、 sysadmin、opradmin、monadmin 用户有权限调用。

表 26-23 DBE_SQL_UTIL.drop_sql_ patch 入参和返回值列表

参数	类型	描述
patch_ name	IN name	PATCH 名称。

result	OUT bool	执行是否成功。

26.7.4 DBE_SQL_UTIL.enable_sql_patch

enable_sql_ patch 是用于开启 SQL PATCH 的接口函数，返回执行是否成功。 限制仅初始用户、sysadmin、opradmin、monadmin 用户有权限调用。

表 26-24 DBE_SQL_UTIL.enable_sql_ patch 入参和返回值列表

参数	类型	描述
patch_ name	IN name	PATCH 名称。
result	OUT bool	执行是否成功。

26.7.5 DBE_SQL_UTIL.disable_sql_patch

disable_sql_ patch 是用于禁用 SQL PATCH 的接口函数，返回执行是否成功。 限制仅初始用户、sysadmin、opradmin、monadmin 用户有权限调用。

表 26-25 DBE_SQL_UTIL.disable_sql_ patch 入参和返回值列表

参数	类型	描述
patch_ name	IN name	PATCH 名称。
result	OUT bool	执行是否成功。

26.7.6 DBE_SQL_UTIL.show_sql_patch

show_sql_ patch 是用于显示给定 patch_ name 对应的 SQL PATCH 的接口函数，返回运行结果。限制仅初始用户、sysadmin、opradmin、monadmin 用户有权限调用。

表 26-26 DBE_SQL_UTIL.show_sql_ patch 入参和返回值列表

参数	类型	描述
patch_ name	IN name	PATCH 名称。
unique_sql_id	OUT bigint	查询全局唯一 ID。

enabled	OUT bool	PATCH 是否生效。
abort	OUT bool	是否是 AbortHint。
hint_str	OUT text	Hint 文本。

GBASE®

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.

微信二维码

技术支持热线：400-013-9696