数据仓库服务
9.1.0
实时数仓

文档版本 03

发布日期 2024-11-27

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 实时数仓简介

实时数仓需要支持将insert+upsert+update等操作实时快速入库，数据来源于上游的其他数据库或者应用，同时要求入库后的数据要能及时查询，对于查询的效率要求很高。

目前GaussDB(DWS)传统数仓已有的行存表或者列存表都无法同时满足实时入库和实时查询两个诉求。其中行存表实时入库能力强，支持高并发更新，但是磁盘占用高，查询效率低；列存表数据压缩率高，AP查询性能好，但是不能很好的支持并发更新，并发入库存在严重的锁冲突。

为了解决上面的问题，需要在使用列存储格式尽量降低磁盘占用的同时，支持高并发的更新操作入库以及高性能的查询效率。GaussDB(DWS)的实时数仓中的HStore表就是针对这种情况设计和实现的，面向对于实时入库和实时查询有较强诉求的场景，同时拥有处理传统TP场景的事务能力。

GaussDB(DWS)提供的实时数仓中实现了一种全新的HStore表，可以做到单条或者小批量IUD操作的高并发实时入库，也可以支持大批量的定期入库。数据入库提交后即可查询，无任何时延。支持主键等传统索引能力去重和加速点查，也支持分区、多维字典、局部排序等方式进一步加速AP查询，也可以在TPCC这种强事务压力场景下保证数据强一致性。

说明

- 实时数仓的HStore表仅8.2.0.100及以上集群版本支持。
- 实时数仓为一库两用，生产即分析；适用于交易、分析混合型业务场景；分为单机、集群两种模式。关于如何创建实时数仓请参见**创建DWS 2.0集群**。
- HStore表支持冷热数据管理，具体可参考**冷热数据管理**，该功能仅8.2.0.101及以上集群版本支持。
- HStore表是实时数仓中设计的一种表类型，与SQL参数hstore没有任何关系。

与标准数仓的区别

实时数仓与标准数仓是GaussDB(DWS)的两种不同类型产品，在使用上也存在一定差异，具体可参考**表1-1**进行对比分析。

表 1-1 实时数仓与标准数仓的差异

数仓类型	标准数仓	实时数仓
适用场景	融合分析业务，一体化OLAP分析场景。主要应用于金融、政企、电商、能源等领域。	实时入库+分析混合业务，上游数据实时入库+数据入库后实时高效查询场景。主要用于电商、金融等实时入库要求高的场景。
产品优势	性价比高，使用场景广泛。 支持冷热数据分析，存储、计算弹性伸缩，无限算力、无限容量等。	混合负载，入库性能强。 提供与列存相当的高性能查询效率与高压缩率的数据压缩能力。同时拥有处理传统TP场景的事务能力。
功能特点	支持海量数据离线处理和交互查询，数据规模大、复杂数据挖掘具有很好的性能优势。	支持海量数据高并发的更新操作入库以及高性能的查询效率。在数据规模大、入库并发高、查询要求高的场景下具有很好的性能优势。
SQL语法	SQL语法兼容性高，语法通用，易于使用。	兼容列存语法。
GUC参数	丰富的GUC参数，根据客户业务场景适配最适合客户的数仓环境。	兼容标准数仓GUC参数，同时支持实时数仓调优参数。

技术特点

- 完整的事务一致性

  体现在数据插入或者更新后提交即可见，不存在时延；并发更新后数据保证强一致，不会出现乱序导致的结果预期不一致。

- 查询性能好

  多表关联等复杂AP查询场景下，更完善的分布式查询计划与分布式执行器带来的性能优势，支持复杂的子查询和存储过程。

- 入库快

  彻底解决列存CU锁冲突问题，支持高并发的更新入库操作，典型场景下，并发更新性能是之前的百倍以上。

- 高压缩

  数据在MERGE进入列存主表后，按列存储具有天然的压缩优势，能极大地节省磁盘空间与IO资源。

- 查询加速

  支持主键等传统索引能力去重和加速点查，也支持分区、多维字典、局部排序等方式进一步加速AP查询。

行存、列存、HStore 表对比

表 1-2 行存、列存、HStore 表对比

表类型	行存表	列存表	HStore表
数据存储方式	以元组为单位，将每一条数据的所有属性值存储到临近的空间里。	以CU（Compress Unit）为单位，将单个属性的所有值存储到临近的空间里。	数据主要以CU形式存储在列存主表上，对于被更新的列、小批量插入的数据将被序列化后存储到新设计的Delta表上。
数据写入	行存压缩暂未商用，数据按原始状态存储，磁盘空间占用较大。	按列存储时，由于属性值类型相同具有天然的压缩优势。数据写入时能极大节省IO资源与磁盘空间占用。	批量插入的数据直接写入CU，具有与列存一致的压缩优势。 被更新的列、小批量插入的数据会序列化后压缩。同时定期merge到主表CU。
数据更新	数据按行更新，没有CU锁问题，并发更新（update/upsert/delete等）性能好。	即使更新单条数据，也要获取整个CU的锁，基本无法支持并发更新（update/upsert/delete等）。	彻底解决列存更新的CU锁问题，并发更新（update/upsert/delete等）的性能达到行存的60%以上。
数据读取	按行读取，即使只需访问某一列的数据，也需要将一整行的数据取出。查询性能较差。	按列读取时只需访问该列的CU，再加上CU的压缩优势导致需要占用的IO资源更少，读取性能很好。	对于列存主表的数据按列读取，对于被更新的列、小批量插入的数据会反序列化后取出，数据merge到主表后具有与列存一致的数据读取优势。
优点	并发更新性能好。	查询性能好，磁盘占用空间少。	并发更新性能好，数据MERGE后具有与列存一致的查询性能优势与压缩优势。
缺点	占用磁盘空间多，查询性能差。	基本无法支持并发更新。	需要后台常驻线程对HStore表进行merge清理操作。先merge到CU主表再进行清理，与SQL语法中的Merge无关。
适用场景	1. 更新删除操作频繁的TP事务场景。 2. 点查询（基于索引的、返回数据量小的简单查询）。	1. 查询分析为主的AP场景。 2. 数据量大，存入后的更新删除操作少。	1. 实时并发入库场景。 2. 需要支持高并发的更新入库操作以及高性能的查询效率。

2 支持与限制

实时数仓兼容所有列存语法，具体情况如下：

表 2-1 支持的语法

语法	是否支持
CREATE TABLE	支持
CREATE TABLE LIKE	支持
DROP TABLE	支持
INSERT	支持
COPY	支持
SELECT	支持
TRUNCATE	支持
EXPLAIN	支持
ANALYZE	支持
VACUUM	支持
ALTER TABLE DROP PARTITION	支持
ALTER TABLE ADD PARTITION	支持
ALTER TABLE SET WITH OPTION	支持
ALTER TABLE DROP COLUMN	支持
ALTER TABLE ADD COLUMN	支持
ALTER TABLE ADD NODELIST	支持
ALTER TABLE CHANGE OWNER	支持
ALTER TABLE RENAME COLUMN	支持
ALTER TABLE TRUNCATE PARTITION	支持

语法	是否支持
CREATE INDEX	支持
DROP INDEX	支持
DELETE	支持
ALTER TABLE 其他	支持
ALTER INDEX	支持
MERGE	支持
SELECT INTO	支持
UPDATE	支持
CREATE TABLE AS	支持

约束限制

1. 当需要使用HStore表时，需要同步修改以下几个参数的默认值，否则会导致HStore表性能严重劣化。

   推荐的参数修改配置是：autovacuum_max_workers_hstore=3，autovacuum_max_workers=6，autovacuum=true。
2. 当前HStore与列存都不支持使用vacuum清理索引脏数据，在频繁update场景可能会导致索引膨胀，后续版本会支持。
3. 当使用HStore异步排序功能时，有两点需要注意。
   - 异步排序期间可能会阻塞部分数据的DML操作，最大的阻塞粒度为异步排序的行数阈值，因此在反复DML的场景下不建议使用此功能。
   - 自动异步排序与列存vacuum互斥，当autovacuum流程中满足列存vacuum的条件，则直接跳过流程中的异步排序，等待下次触发（极端场景下因为反复大批量DML操作的执行，列存vacuum会不停被触发，因此异步排序总是无法被触发）。

3 实时数仓语法

3.1 CREATE TABLE

功能描述

在当前数据库中创建一个新的空白HStore表，该表由命令执行者所有。

实时数仓提供创建HStore表DDL语句。创建HStore表DDL需要指定enable_hstore为true, 同时需要将orientation属性设置为column。

注意事项

- 创建HStore表时，必须确保数据库GUC参数设置满足以下条件：
  - autovacuum设置为on。
  - **autovacuum_max_workers_hstore**取值大于0。
  - **autovacuum_max_workers**取值大于autovacuum_max_workers_hstore的取值。
- 创建HStore表的用户需要拥有schema cstore的USAGE权限。
- 表级参数enable_delta与enable_hstore无法同时开启，因为enable_delta用于控制普通列存表的delta开启，会与enable_hstore冲突。
- 每一个HStore表绑定一张delta表，delta表的oid记录在pg_class中reldeltaidx字段（reldelta字段被列存表的delta表使用）。

语法格式

```
CREATE TABLE [ IF NOT EXISTS ] table_name
({ column_name data_type
   | LIKE source_table [like_option [...] ] }
}
   [, ... ])
[ WITH ( {storage_parameter = value}  [, ... ] ) ]
[ TABLESPACE tablespace_name ]
[ DISTRIBUTE BY  HASH ( column_name [,...])]
[ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
[ PARTITION BY {
      {RANGE (partition_key) ( partition_less_than_item [, ... ] )}
 } [ { ENABLE | DISABLE } ROW MOVEMENT ] ];
```

其中like选项like_option为：


```
{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | PARTITION
| RELOPTIONS | DISTRIBUTION | ALL }
```

列存表的 Delta 表差异

表 3-1 HStore 表与列存表的辅助 Delta 表差异

数仓类型	列存的delta表	HStore的delta表	HStore Opt的delta表
表结构	与列存主表的表定义一致。	与主表定义不一致。	不同于主表，与HStore表相同。
功能	用于暂存小批量insert的数据，满阈值后再merge到主表，避免直接insert到主表产生大量小CU。	用于持久化存储update/delete/insert信息。在产生故障后用于恢复内存更新链等管理并发更新的内存结构。	用于持久化存储update/delete/insert信息。在产生故障后用于恢复内存更新链等管理并发更新的内存结构。相比HStore进一步优化。
缺陷	来不及merge导致delta表膨胀，影响查询性能，同时无法解决并发update的锁冲突问题	依赖后台常驻autovacuum来做merge操作。	依赖后台常驻autovacuum来做merge操作。

数仓类型	列存的delta表	HStore的delta表	HStore Opt的delta表
入库建议	1. 综合入库、查询和空间情况，建议选择hstore_opt表，微批copy无更新入库场景性能要求高的情况下选择hstore表。 2. HStore/HStore Opt共同点： ● update入库性能差，建议修改为upsert。 ● delete入库，确定计划走索引扫描即可，用JDBC batch方式入库最佳。 ● merge into入库建议在单次入库数据量超过100W/dn，且无并发数据保证无重复的情况下使用。 ● 尽量避免对冷分区的数据进行修改和新增。 3. HStore的upsert入库建议： ● 选择流程： 第1步：部分列upsert选择**方式二**，全列upsert（遇到冲突update所有列到新值且没有表达式）进入第2步判断。 第2步：数据入库是否会并发更新到同一个key，没有冲突时选择**方式一**，有冲突时进入第3步判断。 第3步：入库的数据可能遇到重复数据，选择**方式二**，否则进入第4步判断。 第4步：入库的方式是copy攒批临时表时，选择**方式三**，否则（包括jdbc的batch入库，简单的单行入库）选择**方式二**。 ● 配置方式选项解释： – 方式一：enable_hstore_nonconflict_upsert_optimization开启+enable_hstore_partial_upsert_optimization关闭。 – 方式二：enable_hstore_nonconflict_upsert_optimization关闭+enable_hstore_partial_upsert_optimization开启。 – 方式三：enable_hstore_nonconflict_upsert_optimization关闭+enable_hstore_partial_upsert_optimization关闭。 ● 附加说明：攒批小于2000使用batch模式入库，大于2000使用copy临时表模式入库。 4. HStore Opt的upsert入库建议： 无并发冲突下开启enable_hstore_nonconflict_upsert_optimization参数，其他场景都关闭即可，会自动选择最优路径。		

数仓类型	列存的delta表	HStore的delta表	HStore Opt的delta表
点查建议	1. 综合点查场景，建议使用HStore Opt表。 2. HStore/HStore Opt共同点： 在等值过滤条件使用最多且distinct值分布相对均匀的一个列上创建二级分区。 3. HStore的点查建议： ● 使用主键之外的索引加速不能得到预期效果，不建议开启。 ● 如果数据类型多为numeric或者16字节以内的字符串，建议使用turbo加速。 4. HStore Opt的点查建议： ● 除了二级分区之外的等值过滤列，如果过滤条件涉及的列在查询中基本固定，使用cbtree索引，如果不断变化建议使用GIN索引，注意创建索引的列数不要超过5列。 ● 所有涉及等值过滤的字符串列，都可以建表时指定bitmap索引，不限列数，但后续不可修改。 ● 时间范围过滤的列，指定为分区列。 ● 点查返回数据量超过10W/dn的场景，索引扫描的性能提升有限，建议针对这种情况使用guc参数enable_seqscan对比测试，灵活选择。		

参数说明

- **IF NOT EXISTS**

  指定IF NOT EXISTS时，若不存在同名表，则可以成功创建表。若已存在同名表，创建时不会报错，仅会提示该表已存在并跳过创建。

- **table_name**

  要创建的表名。

  表名长度不超过63个字符，以字母或下划线开头，可包含字母、数字、下划线、$、#。

- **column_name**

  新表中要创建的字段名。

  字段名长度不超过63个字符，以字母或下划线开头，可包含字母、数字、下划线、$、#。

- **data_type**

  字段的数据类型。

- **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。

  新表与原表之间在创建动作完毕之后是完全无关的。在原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原表的时候包含新表的数据。

  被复制的列并不使用相同的名字进行融合。如果明确的指定了相同的名字或者在另外一个LIKE子句中，将会报错。

  HStore表只能从HStore表中进行继承。

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

  这个子句为表指定一个可选的存储参数。

  - ORIENTATION

    指定表数据的存储方式，即时序方式、行存方式、列存方式，该参数设置成功后就不再支持修改。对于HStore表，应当使用列存方式，同时设置enable_hstore为on。

    取值范围：

    - TIMESERIES，表示表的数据将以时序方式存储。
    - COLUMN，表示表的数据将以列存方式存储。
    - ROW，表示表的数据将以行方式存储。

    默认值：ROW。

  - 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：

    - HStore表和列存表的有效值为YES/NO和/LOW/MIDDLE/HIGH，默认值为LOW。
    - 行存表的有效值为YES/NO，默认值为NO。

  - 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。该参数只对时序表和列存表有效。

    取值范围：0~3

    默认值：0

  - 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。该参数只对时序表和列存表有效。

    取值范围：10000~60000

    默认值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。该参数只对时序表和列存表有效。

    取值范围：600000~2147483647

  - enable_delta

    指定了在列存表是否开启delta表。对HStore表不能开启该参数。

    默认值：off

  - enable_hstore

    指定了是否创建为HStore表（基于列存表实现）。该参数只对列存表有效。该参数仅8.2.0.100及以上集群版本支持。

默认值：off

说明

打开该参数时必须设置以下GUC参数用于保证HStore表的清理，推荐值如下：
autovacuum=true，autovacuum_max_workers=6，
autovacuum_max_workers_hstore=3。

- enable_disaster_cstore

  指定了列存表是否开启细粒度容灾功能。该参数仅适用于COLVERSION为2.0的列存表，并且不能和enable_hstore同时打开。该参数仅8.2.0.100及以上集群版本支持。

  默认值：off

注意

设置该参数为on前需要先设置GUC参数enable_metadata_tracking=on，否则可能开启细粒度容灾功能失败。

- SUB_PARTITION_COUNT

  指定二级分区的个数。该参数用于设置在导入阶段二级分区个数。在建表时进行设置，建表后不支持修改。不建议用户随意设置该默认值，可能会影响导入和查询的性能。

  取值范围：1～1024

  默认值：32

- DELTAROW_THRESHOLD

  指定HStore表导入时小于多少行(SUB_PARTITION_COUNT * DELTAROW_THRESHOLD)的数据进入delta表。

  取值范围：0～60000

  默认值为60000

- COLVERSION

  指定存储格式的版本。 HStore表只支持2.0版本，enable_hstore_opt表支持2.0和3.0版本。

  取值范围：

  1.0：列存表的每列以一个单独的文件进行存储，文件名以relfilenode.C1.0、relfilenode.C2.0、relfilenode.C3.0等命名。

  2.0：列存表的每列合并存储在一个文件中，文件名以relfilenode.C1.0命名。

  默认值：2.0

- enable_binlog

  用于控制hstore表是否开启binlog功能。该参数仅8.3.0.100及以上集群版本支持。

  取值范围：on/off

  默认值：off

- enable_binlog_timestamp

  用于控制hstore表是否开启带时间戳的binlog功能。该参数与enable_binlog无法同时打开，且仅9.1.0.200及以上集群版本支持。

  取值范围：on/off

默认值：off

- DISTRIBUTE BY

  指定表如何在节点之间分布或者复制。

  取值范围：

  HASH（column_name）：对指定的列进行Hash，通过映射，把数据分布到指定DN。

- TO { GROUP groupname | NODE（ nodename [, ...] ）}

  TO GROUP指定创建表所在的Node Group，目前不支持hdfs表使用。TO NODE主要供内部扩容工具使用，一般用户不应该使用。

- PARTITION BY

  指定HStore表的初始分区。

- secondary_part_column

  指定列存表二级分区列的列名，仅能指定一列作为二级分区列且只适用于HStore列存表。该参数仅8.3.0及以上集群版本支持，其中v3表不支持该参数，会走hashbucket剪枝。

  说明

  - 被指定为二级分区的列不能被删除或者修改。
  - 只能在建表时指定二级分区列，建表后不支持修改。
  - 不建议指定分布列作为二级分区列。
  - 指定二级分区列后，该表将会在DN上按照二级分区列进行逻辑上的HASH分区存储，从而有效提升该列等值查询的性能。

- secondary_part_num

  指定列存表二级分区的数量，仅适用于HStore列存表。该参数仅8.3.0及以上集群版本支持，v3表不支持该参数，会走hashbucket剪枝。

  取值范围：1~32

  默认值：8

  说明

  - 只有secondary_part_column被指定时，该参数才可以指定。
  - 只能在建表时指定二级分区的数量，建表后不支持修改。
  - 不建议用户随意设置该默认值，否则会影响导入和查询的性能。

示例

创建简单的HStore表:

```
CREATE TABLE 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)
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);

CREATE TABLE warehouse_t2 (LIKE warehouse_t1 INCLUDING ALL);
```

3.2 INSERT

功能描述

往HStore表中插入一行或多行数据。

注意事项

- 当单次插入的数据量大于等于表级参数DELTAROW_THRESHOLD时，数据会直接插入主表生成CU（Compress Unit）。
- 当插入的数据量小于表级参数DELTAROW_THRESHOLD时，会在辅助Delta表上插入一条类型为I的记录，同时将数据序列化存储到这条记录的values字段。
- 插入到Delta表的数据跟主表使用全局统一分配的cuid。
- 插入到delta表的数据依赖autovacuum 来merge到主表CU。

语法格式

```
INSERT [/*+ plan_hint */] [ IGNORE | OVERWRITE ] INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    { DEFAULT VALUES
    | VALUES {( { expression | DEFAULT } [, ...] ) }[, ...] | query }
```

参数说明

- **table_name**
  要插入数据的目标表名。
  取值范围：已存在的表名。
- **AS**
  用于给目标表table_name指定别名。alias即为别名的名字。
- **column_name**
  目标表中的字段名。
- **query**
  一个查询语句（SELECT语句），将查询结果作为插入的数据。

示例

创建表reason_t1：

```
-- 创建表reason_t1。
CREATE TABLE reason_t1
(
    TABLE_SK          INTEGER               ,
    TABLE_ID          VARCHAR(20)           ,
    TABLE_NA          VARCHAR(20)
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
```

向表中插入一条记录：

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NA) VALUES (1, 'S01', 'StudentA');
```

向表中插入多条记录：


```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
SELECT * FROM reason_t1 ORDER BY 1;
 TABLE_SK | TABLE_ID | TABLE_NAME
----------+----------+------------
        1 |      S01 |   StudentA
        2 |      T01 |   TeacherA
        3 |      T02 |   TeacherB
(3 rows)
```

3.3 DELETE

功能描述

删除HStore表中的数据。

注意事项

- 如果需要删除表上的所有数据，建议使用TRUNCATE语法，可以有效提高性能同时减少空间膨胀。
- HStore表上的单条Delete操作，会往Delta中插入一条type是D的记录，同时在更新内存更新链用于管理并发。
- HStore表上的批量Delete操作，对于每个CU上的连续delete，会插入一条type是D的记录。
- 对于并发delete场景， 传统列存储格式由于同时操作相同CU时会阻塞所以并发性能较差， 对于HStore表由于不需要阻塞等待，并发delete性能可达到列存的百倍以上。
- 语法完全兼容列存，更多信息可以参考UPDATE语法。

语法格式

```
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition ]
```

参数说明

- **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**

  目标表的名字（可以有模式修饰）。

  取值范围：已存在的表名。
- **alias**

  目标表的别名。

  取值范围：字符串，符合标识符命名规范。
- **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。

示例

创建表reason_t2：

```
CREATE TABLE reason_t2
(
   TABLE_SK         INTEGER              ,
   TABLE_ID         VARCHAR(20)          ,
   TABLE_NA          VARCHAR(20)
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
INSERT INTO reason_t2 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
```

使用WHERE 条件删除：

```
DELETE FROM reason_t2 WHERE TABLE_SK = 2;
DELETE FROM reason_t2 AS rt2 WHERE rt2.TABLE_SK = 2;
```

使用IN语法删除：

```
DELETE FROM reason_t2 WHERE TABLE_SK in (1,3);
```

3.4 UPDATE

功能描述

更新HStore表上指定的数据。

注意事项

- 与列存一样，当前版本HStore上的UPDATE操作始终先DELETE再INSERT。全局GUC参数可控制打开HStore的轻量化UPDATE，当前版本默认关闭。
- 对于并发UPDATE场景， 传统列存储格式由于同时操作相同CU时会产生锁冲突所以并发性能较差， 对于HStore表由于不需要阻塞等待，并发UPDATE性能可达到列存的百倍以上。

语法格式

```
UPDATE [/*+ plan_hint */] [ ONLY ] table_name [ * ] [ [ AS ] alias ]
SET {column_name = { expression | DEFAULT }
   |( column_name [, ...] ) = {( { expression | DEFAULT } [, ...] ) |sub_query }}[, ...]
   [ FROM from_list] [ WHERE condition ];
```

参数说明

- **plan_hint子句**

  以/*+ */的形式在关键字后，用于对指定语句块生成的计划进行hint调优，详细用法请参见**使用Plan Hint进行调优**。
- **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。
- **alias**

  目标表的别名。

  取值范围：字符串，符合标识符命名规范。
- **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。

- **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

  目标表禁止出现在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。

示例

创建表reason_update：

```
CREATE TABLE reason_update
(
   TABLE_SK         INTEGER              ,
   TABLE_ID         VARCHAR(20)          ,
   TABLE_NA          VARCHAR(20)
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
```

向表reason_update中插入数据：

```
INSERT INTO reason_update VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
```

对表reason_update执行UPDATE操作：

```
UPDATE reason_update SET TABLE_NA = 'TeacherD' where TABLE_SK = 3;
```

3.5 UPSERT

功能描述

HStore兼容UPSERT语法，向表中添加一行或多行数据。当出现主键或者唯一约束冲突时更新或者忽略冲突的数据。

注意事项

- 目标表上必须包含主键或者唯一索引才可以执行UPSERT的冲突更新语句。
- 与列存一样，当UPSERT触发更新操作时，当前版本HStore上的更新操作始终先DELETE再INSERT。
- 对于并发UPSERT场景，传统列存储格式由于同时操作相同CU时会产生锁冲突所以并发性能较差，对于HStore表由于不需要阻塞等待，并发UPSERT性能可达到列存的百倍以上。

语法格式

表 3-2 UPSERT 语法格式

语法格式	冲突更新	冲突忽略
第一种：不指定索引	INSERT INTO ON DUPLICATE KEY UPDATE	INSERT IGNORE INSERT INTO ON CONFLICT DO NOTHING
第二种：从指定列名或者约束上可以推断唯一约束	INSERT INTO ON CONFLICT(...) DO UPDATE SET INSERT INTO ON CONFLICT ON CONSTRAINT con_name DO UPDATE SET	INSERT INTO ON CONFLICT(...) DO NOTHING INSERT INTO ON CONFLICT ON CONSTRAINT con_name DO NOTHING

参数说明

第一种不指定索引。会在所有主键或唯一索引上检查冲突，有冲突就会忽略或者更新。

第二种指定索引。会从ON CONFLICT子句中指定列名、包含列名的表达式或者约束名上推断主键或者唯一索引。

- 唯一索引推断

  对于第二种语法形式，通过指定列名或者约束名推断主键或者唯一索引。列名可以是单一列名，或者由多个列名组成的表达式，比如column1，column2，column3。

- UPDATE子句

  UPDATE子句可以通过VALUES(colname)或者EXCLUDED.colname引用插入的数据。EXCLUDED表示因冲突原本该排除的数据行。

- WHERE子句
  - 用于在数据冲突时，判断是否满足指定条件。如果满足，则更新冲突数据。否则忽略。
  - 只有第二种语法形式的冲突更新语法可以指定WHERE子句。即 INSERT INTO ON CONFLICT(...) DO UPDATE SET WHERE

示例

创建表reason_upsert并向表中插入数据：

```
CREATE TABLE reason_upsert
(
  a    int primary key,
  b    int,
  c    int
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
INSERT INTO reason_upsert VALUES (1, 2, 3);
```

忽略冲突的数据：

```
INSERT INTO reason_upsert  VALUES (1, 4, 5),(2, 6, 7) ON CONFLICT(a) DO NOTHING;
```

更新冲突的数据：

```
INSERT INTO reason_upsert  VALUES (1, 4, 5),(3, 8, 9) ON CONFLICT(a) DO UPDATE SET b = EXCLUDED.b,
c = EXCLUDED.c;
```

3.6 MERGE 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

对于并发MERGE INTO场景，触发UPDATE时，传统列存储格式由于同时操作相同CU时会产生锁冲突所以并发性能较差，对于HStore表由于不需要阻塞等待，并发MERGE INTO性能可达到列存的百倍以上。

语法格式

```
MERGE INTO table_name [ [ AS ] alias ]
USING { { table_name | view_name } | subquery } [ [ AS ] alias ]
ON ( condition )
[
  WHEN MATCHED THEN
  UPDATE SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
  [ WHERE condition ]
]
[
  WHEN NOT MATCHED THEN
  INSERT { DEFAULT VALUES |
  [ ( column_name [, ...] ) ] VALUES ( { expression | DEFAULT } [, ...] ) [, ...] [ WHERE condition ] }
];
```

参数说明

- **INTO子句**

  指定正在更新或插入的目标表。

  - **table_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。

- **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。ON关联条件可以是ctid, xc_node_id, tableoid这三个系统列。

- **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。

  说明

  不支持更新分布列、系统表以及系统列。

- **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。

  说明

  - 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。

示例

创建进行MERGE INTO的目标：

```
CREATE TABLE target(a int, b int)WITH(ORIENTATION = COLUMN, ENABLE_HSTORE = ON);
INSERT INTO target VALUES(1, 1),(2, 2);
```

创建数据源表：

```
CREATE TABLE source(a int, b int)WITH(ORIENTATION = COLUMN, ENABLE_HSTORE = ON);
INSERT INTO source VALUES(1, 1),(2, 2),(3, 3),(4, 4),(5, 5);
```

执行MERGE INTO操作：

```
MERGE INTO target t
USING source s
ON (t.a = s.a)
WHEN MATCHED THEN
  UPDATE SET t.b = t.b + 1
WHEN NOT MATCHED THEN
  INSERT VALUES (s.a, s.b) WHERE s.b % 2 = 0;
```

3.7 SELECT

功能描述

从HStore表读取数据。

注意事项

- 列存表与HStore表都暂不支持SELECT FOR UPDATE语法。
- 对HStore表执行SELECT查询时，会扫描列存主表CU上的数据、delta表上的I记录中的数据、内存中每行数据更新信息，并将三种信息合并后返回。
- 在通过主键索引或唯一索引查询数据的场景中：

  对于传统列存表，唯一索引会同时存储行存Delta表上的数据位置信息（blocknum，offset）与列存主表的数据位置信息（cuid，offset），数据MERGE到主表后又会插入新的索引元组，索引会持续膨胀。

  对于HStore表，由于实现了全局CUID的统一分配，索引元组中始终只存储（cuid，offset）， 数据MERGE后不会产生新的索引元组。

语法格式

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [/*+ plan_hint */] [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
```



```
[ HAVING condition [, ...] ]
[ { 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 ] ] } | { LIMIT start, { count | ALL } } ]
```

参数说明

- **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。

- **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。

- **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。

- **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：

- **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。

- **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。

- **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：

示例

创建表reason_select并向表中插入数据：

```
CREATE TABLE reason_select
(
  r_reason_sk integer,
  r_reason_id integer,
  r_reason_desc character(100)
)WITH(ORIENTATION = COLUMN, ENABLE_HSTORE=ON);
INSERT INTO reason_select values(3, 1,'reason 1'),(10, 2,'reason 2'),(4, 3,'reason 3'),(10, 4,'reason 4');
```

执行GROUP BY分组操作：

```
SELECT COUNT(*), r_reason_sk FROM reason_select GROUP BY r_reason_sk;
```

执行HAVING过滤操作：

```
SELECT COUNT(*) c,r_reason_sk FROM reason_select GROUP BY r_reason_sk HAVING c > 1;
```

执行ORDER BY操作：


```
SELECT * FROM reason_select ORDER BY r_reason_sk;
```

3.8 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、添加/更新多个列、将列存改为HStore表等。

注意事项

- 通过ALTER修改enable_hstore值可以将列存表变成HStore表，或者将HStore修改成列存表。但需要注意enable_delta为on时，无法设置enable_hstore为on。
- 对于部分ALTER操作（修改列类型，分区合并，添加NOT NULL约束，添加主键约束），HStore表需要先将数据MERGE到主表然后再执行原有的ALTER逻辑，这可能会带来额外的性能开销， 性能影响大小与delta表的数据量相关。
- 当需要增加列时， 建议不要与其它类型的ALTER（比如修改列类型等）组合使用，在一条ALTER里只有ADD COLUMN情况下，由于不需要做FULL MERGE, 性能会有很大提升。
- 不支持修改存储参数ORIENTATION。

修改表属性

使用语法：

```
ALTER TABLE [ IF EXISTS ] <table_name> SET （{ENABLE_HSTORE = ON} [, ... ] );
```

将列存表修改成HStore表：

```
CREATE TABLE alter_test(a int, b int) WITH(ORIENTATION = COLUMN);
ALTER TABLE alter_test SET (ENABLE_HSTORE = ON);
```

须知

当需要使用HStore表时，一定要同步修改如下参数的默认值，否则会导致HStore性能严重劣化，推荐的默认配置是

autovacuum_max_workers_hstore=3，autovacuum_max_workers=6，autovacuum=true。

增加列

使用语法：

```
ALTER TABLE [ IF EXISTS ] <table_name> ADD COLUMN <new_column> <data_type> [ DEFAULT
<default_value>];
```

示例：

创建表alter_test2并对其增加列：

```
CREATE TABLE alter_test2(a int, b int) WITH(ORIENTATION = COLUMN,ENABLE_HSTORE = ON);
ALTER TABLE alter_test ADD COLUMN c int;
```

说明

增加列时不建议在同一个SQL中与其它ALTER组合使用。

重命名

使用语法：

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

示例：

创建表alter_test3并对其重命名为alter_new：

```
CREATE TABLE alter_test3(a int, b int) WITH(ORIENTATION = COLUMN,ENABLE_HSTORE = ON);
ALTER TABLE alter_test3 RENAME TO alter_new;
```

4 实时数仓函数

hstore_light_merge(rel_name text)

描述：该函数用于手动对HStore表进行轻量化清理操作，持有目标表的三级锁。

返回值类型：int

示例：

```
SELECT hstore_light_merge('reason_select');
```

hstore_full_merge(rel_name text，partitionName text)

描述：该函数用于手动对HStore表进行全量清理操作，其中第二个入参为可选参数，用于指定单分区进行操作。

返回值类型：int

须知

- 执行该操作会强制将DELTA表上的所有可见操作Merge到主表，然后建一张新的空Delta表， 期间持有该表的八级锁。
- 该操作的耗时长度与DELTA表上的数据量有关，务必打开HStore的清理线程，确保HStore表的及时清理。
- 仅在8.3.0.100及以上集群版本支持第二个可选参数partitionName，因此8.3.0.100及以上集群版本不支持使用call调用此函数（ call不支持重载 ）

示例：

```
SELECT hstore_full_merge('reason_select'，'part1');
```

pgxc_get_small_cu_info(rel_name text, row_count int)

描述：该函数用于获取目标表的小CU信息。其中第二个参数row_count 为可选参数，表示小CU的阈值，存活元组数小于这个值的就会被认为是小CU，默认值200。该函数仅8.2.1.300及以上集群版本支持。

返回值类型：record

返回值：

node_name：DN节点名。

part_name：分区名，非分区表此列为空。

zero_cu_count：0CU的数量。当一个CU中的所有数据都被删除时，称之为0 CU。

small_cu_count：小CU数量。当一个CU中**有存活数据**且**存活数量小于阈值**时，称之为小CU。

total_cu_count：总的CU的数量。

sec_part_cu_num：每个二级分区的CU数量。当secondary_part_column被指定时，该列才会显示。该字段仅8.3.0及以上集群版本支持。

需要注意的是，这里的CU是跨列的概念，并非一列一个的CU。

示例：

```
SELECT * FROM pgxc_get_small_cu_info('hs');
node_name | part_name | zero_cu_count | small_cu_count | total_cu_count |
sec_part_cu_num
-----------+-----------+---------------+----------------+----------------+-----------------------------------------
 datanode1 |           |             0 |              4 |              4 | p1:1 p2:0 p3:1 p4:0 p5:1 p6:0 p7:1 p8:0
 datanode2 |           |             0 |              4 |              4 | p1:0 p2:1 p3:0 p4:1 p5:0 p6:1 p7:0 p8:1
(2 rows)
```

gs_hstore_compaction(rel_name text, row_count int)

描述：该函数用于合并目标表的小CU。其中第二个参数row_count为可选参数，表示小CU的阈值，存活元组数小于这个值的就会被认为是小CU，默认值100。该函数仅8.2.1.300及以上版本支持。

返回值类型：int

返回值：numCompactCU: 合并小CU的数量。

说明

- 这里的CU是跨列的概念，并非一列一个的CU。
- 函数中并不能传入分区名，该函数暂不支持指定单分区。

示例：

```
SELECT gs_hstore_compaction('hs'，10);
```

pgxc_get_hstore_delta_info(rel_name text)

描述：该函数用于获取目标表的delta表信息，包括delta表的大小，insert/delete/update各种类型记录的数量等。该函数仅8.2.1.100及以上集群版本支持。

返回值类型：record

返回值：

node_name：DN节点名。

part_name：分区名，非分区表此列为non partition table。

live_tup：存活的元组数量。

n_ui_type：type 是 ui (小cu合并和upsert走update插入) 的记录的数量。一条ui记录表示一次插入，可以是单插或者批插。该参数仅8.3.0.100及以上版本支持。

n_i_type：type 是 i (insert) 的记录的数量。一条i记录表示一次插入，可以是单插或者批插。

n_d_type：type 是 d (delete) 的记录的数量。一条d记录表示一次删除，可以是单条删或者批量删除。

n_x_type：type 是 x (由update产生的delete) 记录的数量。

n_u_type：type 是 u (轻量化update) 的记录数量。

n_m_type：type 是 m (merge) 的记录数量。

data_size：delta表的总大小（包括delta上的索引与toast数据的大小）。

示例：

```
SELECT * FROM pgxc_get_hstore_delta_info('hs_part');
 node_name | part_name | live_tup | n_ui_type | n_i_type | n_d_type | n_x_type | n_u_type | n_m_type |
data_size
-----------+-----------+----------+-----------+----------+----------+----------+----------+----------+-----------
 dn_1      | p1        |        2 |         0 |        2 |        0 |        0 |        0 |        0 |      8192
 dn_1      | p2        |        2 |         0 |        2 |        0 |        0 |        0 |        0 |      8192
 dn_1      | p3        |        2 |         0 |        2 |        0 |        0 |        0 |        0 |      8192
(3 rows)
```

pgxc_get_binlog_sync_point(rel_name text, slot_name text, checkpoint bool, node_id int)

描述：该函数用于用于从pg_binlog_slots系统表上获取槽位对应的同步点信息，只能对开启binlog或binlog时间戳的表使用。该函数仅9.1.0.200及以上集群版本支持。

返回值类型：record

返回值：

node_name：节点名

node_id：节点id

last_sync_point：上次同步点

latest_sync_point：当前最新同步点

xmin：同步点对应xmin

示例：

```
select * from pg_catalog.pgxc_get_binlog_sync_point('hstore_binlog_source', 'slot1', false, 0);
 node_name |   node_id   | last_sync_point | latest_sync_point | xmin
-----------+-------------+-----------------+-------------------+-------
 dn_2      | -1051926843 |               0 |             10512 | 10507
 dn_1      | -1300059100 |               0 |             10512 | 10508
(2 rows)
```

pgxc_get_binlog_changes(rel_name text, node_id int, start_csn bigint, end_csn bigInt)

描述：该函数用于获取目标表在指定DN上指定同步点区间的增量数据(node_id给0表示指定所有DN)，只能对开启binlog或binlog时间戳的表使用。该函数仅9.1.0.200及以上集群版本支持。

返回值类型：record

返回值：

gs_binlog_sync_point ： 同步点

gs_binlog_event_sequence：用于表示同一事务内的先后顺序

gs_binlog_event_type：binlog类型

gs_binlog_timestamp_us：binlog记录的时间戳，对于enable_binlog_timestamp为false的binlog表，该列返回空

value columns：目标表上各个用户字段的数据

示例：

```
select * from pgxc_get_binlog_changes('hstore_binlog_source', 0, 0 , 9999999999);
 gs_binlog_sync_point | gs_binlog_event_sequence | gs_binlog_event_type | gs_binlog_timestamp_us | c1  | c2 | c3
----------------------+--------------------------+----------------------+------------------------+-----+----+----
               10516 |                        2 | I                    |      1731570520900211 | 100 |  1 |  1
               10517 |                        3 | d                    |      1731570520904425 | 100 |  1 |  1
               10518 |                        2 | I                    |      1731570520909055 | 200 |  1 |  1
               10519 |                        3 | B                    |      1731570520914102 | 200 |  1 |  1
               10519 |                        4 | U                    |      1731570520914154 | 200 |  2 |  1
```

pgxc_register_binlog_sync_point(rel_name text, slot_name text, node_id int, end_csn bigInt, checkpoint bool, xmin bigint)

描述：该函数用于登记同步点，只能对开启binlog或binlog时间戳的表使用。该函数仅9.1.0.200及以上集群版本支持。

返回值类型：int

返回值: 登记成功的节点数量

示例：

```
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, false, 100);
 pgxc_register_binlog_sync_point
---------------------------------
                               2
(1 row)
```

pgxc_consumed_binlog_records(rel_name text, node_id int)

描述：该函数用于获取目标表在指定DN上的消费状态，只能对开启binlog或binlog时间戳的表使用。该函数仅9.1.0.200及以上集群版本支持。

返回值类型：int

返回值：返回0表示目标表的binlog仍没有被消费完毕（包含所有槽位以及checkpoint同步点），返回1表示目标表的binlog被全部消费完毕。

示例：

```
select * from pgxc_consumed_binlog_records('hstore_binlog_source',0);
 pgxc_consumed_binlog_records
------------------------------
                             1
(1 row)
```


pgxc_get_binlog_cursor_by_timestamp(rel_name text, timestamp timestampTz, node_id int)

描述：该函数用于获取目标表上指定时间点后的第一个binlog记录的信息，只能对开启binlog时间戳的表使用。

该函数仅9.1.0.200及以上集群版本支持。

返回值类型：record

返回值：

node_name：节点名

node_id：节点id

atest_sync_point：当前最新同步点

binlog_sync_point：时间点后的第一个binlog记录的同步点

binlog_timestamp_us：时间点后的第一个binlog记录的时间戳

binlog_xmin：时间点后的第一个binlog记录的xmin

示例：

```
select * from pgxc_get_binlog_cursor_by_timestamp('hstore_binlog_source','2024-11-14
15:48:40.900211+08', 0);
 node_name |   node_id   | latest_sync_point | binlog_sync_point | binlog_timestamp_us | binlog_xmin
-----------+-------------+-------------------+-------------------+---------------------+-------------
 dn_2      | -1051926843 |             10532 |             10516 |    1731570520900211 |       10510
 dn_1      | -1300059100 |             10532 |             10518 |    1731570520909055 |       10510
(2 rows)
```

pgxc_get_binlog_cursor_by_syncpoint(rel_name text, csn int8, node_id int)

描述：该函数用于获取目标表上指定同步点后的第一个binlog记录的信息，只能对开启binlog时间戳的表使用。

该函数仅9.1.0.200及以上集群版本支持。

返回值类型：record

返回值：

node_name：节点名

node_id：节点id

atest_sync_point：当前最新同步点

binlog_sync_point：同步点后的第一个binlog记录的同步点

binlog_timestamp_us：同步点后的第一个binlog记录的时间戳

binlog_xmin：同步点后的第一个binlog记录的xmin

示例：

```
select * from pgxc_get_binlog_cursor_by_syncpoint('hstore_binlog_source',10516,0);
 node_name |   node_id   | latest_sync_point | binlog_sync_point | binlog_timestamp_us | binlog_xmin
-----------+-------------+-------------------+-------------------+---------------------+-------------
 dn_1      | -1300059100 |             11187 |             10518 |    1731570520909055 |       10510
 dn_2      | -1051926843 |             11187 |             10516 |    1731570520900211 |       10510
(2 rows)
```

pgxc_get_cstore_dirty_ratio(rel_name text, partition_name)

描述：该函数用于获取各个DN上目标表的cu、delta以及cudesc的脏页率以及大小，仅支持HStore_opt表。

其中partition_name为可选参数，如果传入分区名则只返回该分区对应的信息；如果没传入分区名且是主表时分区表返回所有分区对应的信息。该函数仅9.1.0.100及以上集群版本支持。

返回值类型：record

返回值：

node_name：DN节点名

database_name：表所在数据库名称

rel_name：主表名

part_name：分区名

cu_dirty_ratio：cu文件的脏页率

cu_size：cu文件大小

delta_dirty_ratio：delta表脏页率

delta_size：delta表大小

cudesc_dirty_ratio：cudesc表脏页率

cudesc_size：cudesc表大小

示例：

```
SELECT * FROM pgxc_get_cstore_dirty_ratio('hs_opt_part');
 node_name | database_name |      rel_name       | partition_name | cu_dirty_ratio | cu_size | delta_dirty_ratio
| delta_size | cudesc_dirty_ratio | cudesc_size
-----------+---------------+---------------------+----------------+----------------+---------+-------------------
+------------+--------------------+-------------
 dn_1      | postgres      | public.hs_opt_part | p1             |              0 |       0 |               0 |      16384
|               0 |       24576
 dn_1      | postgres      | public.hs_opt_part | p2             |              0 |       0 |               0 |      16384
|               0 |       24576
 dn_1      | postgres      | public.hs_opt_part | p3             |              0 |       0 |               0 |      16384
|               0 |       24576
 dn_1      | postgres      | public.hs_opt_part | p4             |              0 |       0 |               0 |      16384
|               0 |       24576
 dn_1      | postgres      | public.hs_opt_part | other          |              0 | 1105920 |               0 |     524288
|               0 |       40960
```


5 实时数仓 GUC 参数

autovacuum

参数说明：控制是否启动数据库自动清理进程（autovacuum）。

参数类型：SIGHUP

取值范围：布尔型

- on表示开启数据库自动清理进程。
- off表示关闭数据库自动清理进程。

默认值：on

autovacuum_compaction_rows_limit

参数说明：小CU的阈值，存活元组数小于这个值的就会被认为是小CU。该参数仅8.2.1.300及以上集群版本支持。

参数类型：USERSET

取值范围：整型，-1~5000

默认值：2500

须知

版本低于9.1.0.100，禁止设置该参数，否则可能会导致**主键数据重复**。

说明

- 版本低于9.1.0.100，默认值为-1，表示关闭0CU开关。
- 9.1.0.100版本，该参数默认值为0。
- 9.1.0.200及以上版本，该参数默认值为2500。
- **该参数不建议自行修改，如需修改请联系技术支持。**

autovacuum_compaction_time_limit

参数说明：小CU清理的时间间隔，每间隔一段时间就会执行一次小CU合并。该参数仅8.2.1.300及以上集群版本支持。

参数类型：SIGHUP

取值范围：整型，0~10080，单位为分钟。

默认值：0

autovacuum_max_workers

参数说明：设置能同时运行的自动清理线程的最大数量，该参数的取值上限与max_connections和job_queue_processes大小有关。

参数类型：SIGHUP

取值范围：整型

- 最小值为0，表示不会自动进行autovacuum。
- 理论最大值为262143，实际最大值为动态值。计算公式为“262143 - max_inner_tool_connections - max_connections - job_queue_processes - 辅助线程数 – autovacuum的launcher线程数 - 1”，其中辅助线程数和autovacuum的launcher线程数由两个宏来指定，当前版本的默认值分别为20和2。

默认值：4

autovacuum_max_workers_hstore

参数说明：设置Autovacuum_max_workers里面，能同时运行的专用于清理hstore的自动清理线程的最大数量。

参数类型：SIGHUP

取值范围：整型

默认值：1

说明

当需要使用hstore表时，一定要同步修改以下几个参数的默认值，否则会导致hstore表性能严重劣化，推荐的修改配置是：

autovacuum_max_workers_hstore=3，autovacuum_max_workers=6，autovacuum=true。

hstore_buffer_size

参数说明：用于控制HStore的CU槽位数量，该槽位用于存储每个CU的更新链，能显著提升更新与查询效率。

为避免占用内存过大，系统会根据内存大小计算出一个槽位值，再与该参数相比取最小的值。

参数类型：POSTMASTER

取值范围：整型，100~100000

默认值：100000

gtm_option

参数说明：GaussDB(DWS)场景下，指定GTM运行模式。该参数仅8.2.1及以上集群版本支持。

- GTM模式：常规模式，由GTM统一管理运行中的事务，以及XID和CSN的分配工作。
- GTM-Lite模式：GTM只负责XID的分配和CSN的更新，不再负责全局事务管理。GTM-Lite模式适用于高并发，短查询的TP场景，可以在保证事务一致性的情况下提升查询性能。
- GTM-Free模式：分布式事务只支持写外部一致性，不具有读外部一致性。实时数仓场景下设置该模式不生效。

参数类型：POSTMASTER

取值范围：枚举类型

- gtm或0：表示开启GTM模式。
- gtm-lite或1：表示开启GTM-Lite模式。
- gtm-free或2：表示开始GTM-Free模式。

默认值：gtm

须知

1. GaussDB(DWS)实例和GTM实例都有相同含义的gtm_option参数，对于GTM和GTM-Lite两种模式，一定要在gaussdb和gtm上设置为相同的模式，否则会出现业务报错无法执行的问题。
2. GTM-Free模式可以通过enable_gtm_free = on或gtm_option = gtm-free的任一方法开启。
3. 设置非GTM-Free模式时，enable_gtm_free 必须设置为off。
4. GTM-Free模式仅在混合云与ESL场景设置后生效。

defer_xid_cleanup_time

参数说明：指定实时数仓中GTM-Lite模式下全局OldestXmin的维护周期。每个维护周期内，由CCN或FCN进行全局OldestXmin的收集判断和统一下发。该参数仅8.2.1及以上集群版本支持。

此参数仅在GTM-Lite模式生效，不建议修改此参数。

参数类型：SIGHUP

取值范围：整型，1~INT_MAX ，单位为ms。

默认值：5000

enable_hstore_keyby_upsert

参数说明：用于控制是否开启hstore表对于批量upsert的特定优化，针对前端可以保证不会并发upsert到同一行且是全列upsert的场景，建议开启，性能会有较大提升，该参数仅8.3.0及以上集群版本支持。

参数类型：USERSET

取值范围：布尔型

on表示开启对hstore表上的upsert特定优化。

off表示关闭对hstore表上的upsert特定优化，走老的流程。

默认值：off

autovacuum_asyncsort_rows_limit

参数说明：自动异步排序的行数阈值。该参数仅9.1.0及以上集群版本支持。

参数类型：SIGHUP

取值范围：整型，120000~4200000

默认值：600000

autovacuum_asyncsort_size_limit

参数说明：自动异步排序的空间占用阈值。该参数仅9.1.0及以上集群版本支持。

参数类型：SIGHUP

取值范围：整型，1048576~104857600，单位为KB。

默认值：10485760（10GB）

autovacuum_asyncsort_time_limit

参数说明：自动异步排序的时间间隔，只有当前表距离上次异步排序的时间间隔超过autovacuum_asyncsort_time_limit才会再次触发异步排序。该参数仅9.1.0及以上集群版本支持。

参数类型：SIGHUP

取值范围：整型，0~10080，单位为分钟。

默认值：10

enable_hstore_binlog_table

参数说明：用于控制是否可以创建binlog表。

参数类型：SIGHUP

取值范围：布尔型

- on表示可以创建binlog表。
- off表示不可以创建binlog表。

默认值：off

enable_generate_binlog

参数说明：用于控制当前会话的binlog表上的DML操作是否产生binlog。该参数仅9.1.0.200以上集群版本支持。

参数类型：USERSET

取值范围：布尔型

- on表示产生binlog。
- off表示不产生binlog。

默认值：on

binlog_consume_timeout

参数说明：用于控制在线扩缩容binlog表或者vacuum full binlog表时，循环判断binlog记录是否都被消费的超时时间。该参数仅8.3.0.100及以上版本支持。单位秒。

参数类型：SIGHUP

取值范围：整型，0~86400

默认值： 3600

enable_hstoreopt_auto_bitmap

参数说明：用于控制是否默认在建表时给HStore Opt表自动设置bitmap columns列。该参数仅9.1.0.100及以上版本支持。

参数类型：SIGHUP

取值范围：布尔型

- on表示默认设置bitmap columns选项。
- off表示默认不设置bitmap columns选项。

默认值：off

enable_cu_predicate_pushdown

参数说明：

1. 功能概述：该功能用于控制是否启用过滤器下推。启用后，能够普遍提升查询性能，尤其是在涉及bitmap_columns列和pck排序列时性能会显著提升。适用于特定的WHERE条件、IS NULL条件、IN条件等场景。该参数仅9.1.0.200及以上版本支持。
2. 支持的列类型：
   - 整数类型：INT2、INT4、INT8
   - 日期和时间类型：DATE、TIMESTAMP、TIMESTAMPTZ
   - 字符串类型：VARCHAR、TEXT
   - 数值类型：NUMERIC（长度在19以内）
3. 支持查询条件：该功能支持多种 WHERE 表达式，主要包括：
   - **IN 表达式**：用于匹配多个值
   - **IS NULL / IS NOT NULL 条件**：检查列值是否为空或非空
   - **比较表达式**：如大于（>）、小于（<）、等于（=）、不等于（<>）等条件，用于范围查询和精确匹配。

参数类型：USERSET

取值范围：布尔型

- on表示开启对filter下推。
- off表示关闭对filter下推。

默认值：on

enable_hstoreopt_insert_sort

参数说明：用于控制HStore Opt表是否开启入库排序（包括Vaccum Full）。

该参数仅9.1.0.100及以上版本支持。

参数类型：SIGHUP

取值范围：布尔型

- on表示开启入库排序。
- off表示在异步排序开启（autovacuum_asyncsort_time_limit大于0）时，关闭入库排序（包括Vacuum Full)，异步排序关闭时，入库排序（包括Vacuum Full) 仍然启用。

默认值：on

6 实时数仓 Binlog

6.1 订阅实时数仓 Binlog

Binlog 使用介绍

当用户需要捕获数据库事件用于数据增量导出Flink等第三方组件，并协同完成数据加工等任务时，DWS实时数仓中的HStore表提供了Binlog功能，通过消费Binlog数据来实现上下游的数据同步，提高数据加工的效率。

传统的数据比如MySQL数据库等，支持通过Binlog来记录数据库中所有数据的变化，但相比于MySQL的Binlog主要用于数据恢复与主从复制，DWS实时数仓Binlog一般只用于实时场景下的数据同步。同时DWS实时数仓Binlog并不会记录DDL操作，只记录Insert/Delete/Update/(Upsert)等DML操作。

GaussDB(DWS)的Binlog的优点如下：

- 表级按需开关：按需给指定表打开或关闭binlog功能，更为灵活。
- 全增量一体消费：支持Flink任务启动后，先全量同步source，再实时消费source端增量。
- 支持消费即清理：对于空间敏感且只关注实时同步与加工的客户，支持消费后即开始异步清理增量，有效减少空间使用。

利用Flink强大的实时处理能力和GaussDB(DWS)的Binlog能力，可以快速构建实时数仓，且无需维护其他组件（如kafka），整体架构分层清晰，数据可以高效流动，并且整体任务链路都可以通过Flink SQL来驱动，便于业务人员理解和使用。

约束与限制

1. 当前仅8.3.0.100及以上版本支持HStore和Hstore-opt记录Binlog功能，且V3表处于试商用阶段，使用前需要联系技术支持进行评估。
2. 使用Binlog的前置条件是必须存在主键约束，并且为HStore表或者Hstore-opt表，分布方式只能是Hash分布。
3. Binlog表仅记录insert/delete/update(upsert)等DML操作进行记录，不会记录DDL。
4. 当前Binlog表不支持的操作： Insert overwrite、修改分布列、给临时表开启Binlog、exchange/merge/split partition。

5. 当前Binlog表并不限制用户进行以下DDL操作，但进行操作后会导致增量数据与同步点位信息会被清空，需要评估后再执行:

   ADD COLUMN 增加列、DROP COLUMN 删除列、SET TYPE 修改列、TRUNCATE 清空表数据。

6. Binlog表在线或者离线扩容期间会等待Binlog记录的消费，只有Binlog记录消费完毕才可以继续进行接下来的扩缩容步骤，默认等待时间为1小时，可通过guc参数**binlog_consume_timeout**来设置，如果等待超时或者等待出错都会退出扩缩容过程，认为该表扩缩容失败。
7. VACUUM FULL Binlog表时，会等待Binlog记录的消费，只有Binlog记录消费完毕才可以进行接下来的VACUUM FULL操作，默认等待时间为1小时，可通过guc参数**binlog_consume_timeout**来设置，如果等待超时或者等待出错都会退出VACUUM FULL过程，认为该表VACUUM FULL失败。且由于需要等待Binlog记录消费完毕，所以即使VACUUM FULL一个分区表，也会对分区的主表上7级锁，阻塞整个表的插入更新或者删除。
8. Binlog表在备份恢复期间，仅会被当做普通HStore表进行备份，恢复后辅助表的增量数据与同步点信息会清空，需要重新开始同步。
9. 支持Binlog时间戳功能，通过设置**enable_binlog_timestamp**打开，同样只有HStore和Hstore-opt两种表支持打开。该约束仅9.1.0.200及以上版本支持。

Binlog 格式与原理

表 6-1 binlog 字段格式

字段名称	字段类型	含义
gs_binlog_sync_point	BIGINT	Binlog系统字段，表示该记录的同步点值，普通GTM模式下，该值唯一且有序。
gs_binlog_event_sequence	BIGINT	Binlog的系统字段， 用于表示同一事务类操作的先后顺序。
gs_binlog_event_type	CHAR	Binlog的系统字段, 表示当前记录的操作类型。 type可能有以下几种取值： • 'I' 即INSERT， 表示当前Binlog是插入一条新记录。 • 'd' 即DELETE，表示当前Binlog是删除一条记录。 • 'B' 即BEFORE_UPDATE，表示当前Binlog是更新前的记录。 • 'U'即AFTER_UPDATE，表示当前Binlog是更新后的记录。
gs_binlog_timestamp_us	BIGINT	Binlog的系统字段, 表示当前记录入库时的时间戳。 只有开启binlog时间戳功能时会有，没开启binlog时间戳时为空。仅9.1.0.200及以上版本支持。
user_column_1	用户列	用户的自定义数据列

字段名称	字段类型	含义
...	...	...
usert_column_n	用户列	用户的自定义数据列

说明

- UPDATE（或者UPSERT触发的更新操作）会产生两条Binlog，分别是BEFORE_UPDATE类型与AFTER_UPDATE类型，其中BEFORE_UDPATE主要用于保证Flink等第三方组件做数据加工后加工结果的正确性。
- 对于用户UPDATE/DELETE操作产生的BEFORE_UPDATE以及DELETE类型的Binlog，DWS实时数仓中这两类Binlog类型并不会在操作执行时就反查出所有用户列并填充，保证了入库的性能。
- 当在DWS实时数仓给某个HStore表开启Binlog功能时， 本质就是给HStore表再创建一张辅助表（也通过HStore实现），该辅助表包含gs_binlog_event_sync_point、gs_binlog_event_event_sequence、gs_binlog_event_type三个系统列以及一个将所有用户列序列化存储到一个字段的value列。
- 当开启binlog时间戳功能时（表级参数**enable_binlog_timestamp**），辅助表上记录的binlog记录会严格保留至超过TTL才会清理，这会带来数倍的额外空间开销（具体倍数与TTL时间内的更新入库量有关）。当开启普通binlog时（表级参数enable_binlog），辅助表上记录的binlog，只要被下游消费就会允许异步清理，能大大减少空间占用。仅9.1.0.200及以上版本支持。

开启 Binlog

通过在建HStore表时指定表级参数enable_binlog，开启HStore表的Binlog功能。

```
CREATE TABLE hstore_binlog_source (
    c1  INT PRIMARY KEY,
    c2  INT,
    c3  INT
) WITH (
    ORIENTATION = COLUMN,
    enable_hstore_opt=true,
    enable_binlog=on,
    binlog_ttl = 86400
);
```

说明

- 对于开启了Binlog的表，并不会立刻给入库的操作记录Binlog，需要再给同步任务注册同步点后，才会开始记录Binlog（开启Flink同步binlog任务后，会自动循环进行获取同步点、获取增量数据、注册同步点操作）。
- binlog_ttl是可选参数，当不设置时将使用默认值86400， 单位为秒，当同步任务注册的同步点超过TTL没有进行增量同步时，该同步点位将被清理。最老的同步点位之前的Binlog（即被所有任务消费了的Binlog）会被异步清理来回收空间。
- 空间开销：对于开启普通binlog的表，如果能保证增量被下游及时消费，那么空间就能被及时清理回收。

通过执行ALTER命令给已有的HStore表开启binlog功能：

```
CREATE TABLE hstore_binlog_source (
    c1  INT PRIMARY KEY,
    c2  INT,
    c3  INT
) WITH (
```



```
    ORIENTATION = COLUMN,
    enable_hstore_opt=true
);
ALTER TABLE hstore_binlog_source SET (enable_binlog=on);
```

查询 binlog

通过DWS提供的系统函数，可以直接查询目标表在指定DN上binlog信息，以及是否被下游消费完毕等信息。

```
-- 模拟Flink调用系统函数获取同步点，参数分别表示 表名、槽位名、是否checkPoint点位，目标DN(为0表示所有DN)。
select * from pg_catalog.pgxc_get_binlog_sync_point('hstore_binlog_source', 'slot1', false, 0);
select * from pg_catalog.pgxc_get_binlog_sync_point('hstore_binlog_source', 'slot1', true, 0);
-- 进行增删改产生增量binlog。
INSERT INTO hstore_binlog_source VALUES(100, 1, 1);
delete hstore_binlog_source where c1 = 100;
INSERT INTO hstore_binlog_source VALUES(200, 1, 1);
update hstore_binlog_source set c2 =2 where c1 = 200;
-- 模拟Flink调用系统函数查询指定CSN区间的Binlog，参数分别表示表名，目标DN(为0表示所有DN),起始CSN点位, 终止CSN点位。
select * from pgxc_get_binlog_changes('hstore_binlog_source', 0, 0 , 9999999999);
```

```
postgres=# select * from pgxc_get_binlog_changes('hstore_binlog_source', 0, 0 , 9999999999);
 gs_binlog_sync_point | gs_binlog_event_sequence | gs_binlog_event_type | gs_binlog_timestamp_us | c1  | c2  | c3
----------------------+--------------------------+----------------------+------------------------+-----+-----+----
                10241 |                        2 | I                    |                        | 100 |   1 |  1
                10242 |                        3 | d                    |                        | 100 |   1 |  1
                10243 |                        4 | I                    |                        | 100 |   1 |  1
                10245 |                        5 | B                    |                        | 100 |   1 |  1
                10245 |                        6 | U                    |                        | 100 | 100 |  1
(5 rows)
```

可以看到两次INSERT操作产生了两个gs_binlog_event_type是'I'的记录，DELETE操作产生了type是'd'的记录，UPDATE产生了一行BeforeUpdate的'B'记录以及一条AfterUpdate的'U'记录，分别表示更新前的值以及更新后的值。

通过调用系统函数**pgxc_consumed_binlog_records**可以查询目标表的binlog是否被所有槽位消费完毕。参数分别表示目标表名以及目标DN（为0表示所有DN），返回值。

```
-- 模拟Flink调用系统函数注册同步点，参数分别表示表名，槽位名，注册的点位，是否属于checkPoint, 点位对应的xmin (获取同步点时会提供)。
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, false, 100);
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, true, 100);
-- 查询表上binlog是否被全部消费，返回 1 表示已经被下游槽位全部消费。
select * from pgxc_consumed_binlog_records('hstore_binlog_source',0);
```

```
postgres=# select * from pgxc_consumed_binlog_records('hstore_binlog_source',0);
 pgxc_consumed_binlog_records
------------------------------
                            1
(1 row)
```

开启 Binlog 时间戳功能

如果需要读取指定时间点之后binlog的功能，通过在建HStore表时指定表级参数enable_binlog_timestamp，开启HStore表的Binlog时间戳功能。仅9.1.0.200及以上版本支持。

```
CREATE TABLE hstore_binlog_source(
    c1  INT PRIMARY KEY,
    c2  INT,
    c3  INT
) WITH (
    ORIENTATION = COLUMN,
    enable_hstore_opt=true,
    enable_binlog_timestamp =on,
    binlog_ttl = 86400
);
```


说明

- 对于开启了Binlog时间戳的表，并不会立刻给入库的操作记录Binlog，需要再给同步任务注册同步点后，才会开始记录Binlog（开启Flink同步binlog任务后，会自动循环进行获取同步点、获取增量数据、注册同步点操作）。
- binlog_ttl是可选参数，当不设置时将使用默认值86400， 单位为秒（即默认保留一天），当Binlog记录的时间戳距离现在大于TTL时，会被异步清理。
- 空间开销：对于开启binlog时间戳的表，辅助表上记录的binlog记录会严格保留至超过TTL才会清理，这会带来数倍的额外空间开销（具体倍数与TTL时间内的更新入库量有关）。

查询开启binlog时间戳功能的表上的Binlog：

```
postgres=# select * from pgxc_get_binlog_changes('hstore_binlog_source', 0, 0 , 9999999999);
 gs_binlog_sync_point | gs_binlog_event_sequence | gs_binlog_event_type | gs_binlog_timestamp_us | c1  | c2 | c3
----------------------+--------------------------+----------------------+------------------------+-----+----+----
                10516 |                        2 | I                    |       1731570520900211 | 100 |  1 |  1
                10517 |                        3 | d                    |       1731570520904425 | 100 |  1 |  1
                10518 |                        2 | I                    |       1731570520909055 | 200 |  1 |  1
                10519 |                        3 | B                    |       1731570520914102 | 200 |  1 |  1
                10519 |                        4 | U                    |       1731570520914154 | 200 |  2 |  1
(5 rows)
```

将gs_binlog_timestamp_us从BigInt类型转成可读的时间戳：

```
select to_timestamp(1731569598408661/1000000);
```

```
postgres=#  select to_timestamp(1731570520900211/1000000);
          to_timestamp
-------------------------------
 2024-11-14 15:48:40.900211+08
(1 row)
```

获取各个DN上，目标表指定时间点后的第一条binlog信息（为空表示这个时间点后没有binlog）：

```
select * from pgxc_get_binlog_cursor_by_timestamp('hstore_binlog_source','2024-11-14 15:33:18.40866+08', 0);
```

```
postgres=# select * from pgxc_get_binlog_cursor_by_timestamp('hstore_binlog_source','2024-11-14 15:48:40.900211+08', 0);
 node_name |   node_id   | latest_sync_point | binlog_sync_point | binlog_timestamp_us | binlog_xmin
-----------+-------------+-------------------+-------------------+---------------------+-------------
 dn_2      | -1051926843 |             10532 |             10516 |    1731570520900211 |       10510
 dn_1      | -1300059100 |             10532 |             10518 |    1731570520909055 |       10510
(2 rows)
```

获取开启binlog时间戳功能的表的消费进度：

返回的字段表示最近消费的一条binlog的时间戳，binlog上的最近时间戳，最近消费的一条binlog的CSN点位， binlog上最近CSN点位，未消费的binlog记录数量。

```
-- 模拟Flink调用系统函数注册同步点，参数分别表示表名，槽位名，注册的点位，是否属于checkPoint, 点位对应的xmin (获取同步点时会提供)。
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, false, 100);
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, true, 100);
-- 查询目标表各个槽位的消费进度。
select * from pgxc_get_binlog_consume_progress('hstore_binlog_source', 0);
```

```
postgres=# select * from pgxc_get_binlog_consume_progress('hstore_binlog_source', 0);
 node_name |   node_id   | slot_name | checkpoint | latest_consumed_timestamp |    latest_timestamp    | latest_consumed_csn | latest_csn | unconsumed_binlog_count
-----------+-------------+-----------+------------+---------------------------+------------------------+---------------------+------------+-------------------------
 dn_1      | -1300059100 | slot1     | f          | 2024-11-14 15:48:40+08    | 2024-11-14 15:48:40+08 |               10519 |      10519 |                       0
 dn_1      | -1300059100 | slot1     | t          | 2024-11-14 15:48:40+08    | 2024-11-14 15:48:40+08 |               10519 |      10519 |                       0
 dn_2      | -1051926843 | slot1     | f          | 2024-11-14 15:48:40+08    | 2024-11-14 15:48:40+08 |               10517 |      10517 |                       0
 dn_2      | -1051926843 | slot1     | t          | 2024-11-14 15:48:40+08    | 2024-11-14 15:48:40+08 |               10517 |      10517 |                       0
(4 rows)
```

控制 DML 不产生 Binlog

通过设置会话级参数**enable_generate_binlog**为off，可以控制当前会话的DML，在给开启binlog的表入库时，不产生binlog记录。

6.2 Flink 实时消费 Binlog

注意事项

- 当前仅8.3.0.100及以上的版本支持HStore和HStore-opt记录Binlog功能，且处于试商用阶段，使用前需要进行评估。
- 目前GaussDB(DWS)只有Hstore表支持Binlog功能，表需要包含主键且设置enable_binlog=on。
- 消费的Binlog表名不要带有特殊字符，如.、""等。
- 如果多个任务消费同一张表的Binlog数据，需要保证每个任务的binlogSlotName唯一。
- 为了达到最高的消费速度，建议将任务的并发度和DWS集群DN数设置一致。
- 使用**dws-connector-flink**的Sink能力来写入Binlog数据的话，需要注意以下几点：
  - 如果需要保证DN内的数据写入顺序则需要设置connectionSize设置为1。
  - 如果源端有更新主键操作或者需要flink进行聚合计算的话，将ignoreUpdateBefore设置为false，否则不建议将 ignoreUpdateBefore设置为false（默认true）。

Flink 实时消费 Binlog

使用DWS Connector来实时消费Binlog，具体请参见**DWS-Connector**。

如果已使用其他同步工具已经将全量数据同步到了目标端，后续只想进行增量同步。则可以调用以下系统函数来更新同步点。

```
SELECT * FROM pg_catalog.pgxc_register_full_sync_point('table_name', 'slot_name');
```

源表 DDL

Source端会根据操作类型自动为每行数据设置准确的Flink RowKind类型（INTERT、DELETE、UPDATE_BEFORE、UPDATE_AFTER)，这样就能镜像同步表的数据，类似MySQL和Postgres的CDC功能。

```
CREATE TABLE test_binlog_source (
  a int,
  b int,
  c int,
  primary key(a) NOT ENFORCED
) with (
  'connector' = 'dws',
  'url' = 'jdbc:gaussdb://ip:port/gaussdb',
  'binlog' = 'true',
  'tableName' = 'test_binlog_source',
  'binlogSlotName' = 'slot',
  'username'='xxx',
  'password'='xxx')
```

Binlog 相关参数说明

下表仅涉及消费Binlog时的参数。

表 6-2 消费 Binlog 时的参数

参数	说明	数据类型	默认值
binlog	是否读取Binlog信息	Boolean	false
binlogSlotName	槽位信息，可以理解一个标识。由于可能存在多个Flink任务同时消费同一张表的Binlog信息，所以该场景需要保证每个任务的binlogSlotName不同。	String	Flink映射表的表名
binlogBatchReadSize	批量读取binlog的数据行数	Integer	5000
fullSyncBinlogBatchReadSize	全量读取binlog的数据行数	Integer	50000
binlogReadTimeout	增量消费Binlog数据时超时时间，单位毫秒	Integer	600000
fullSyncBinlogReadTimeout	全量消费Binlog数据时超时时间，单位毫秒	Long	1800000
binlogSleepTime	实时消费不到Binlog数据时休眠时间，单位毫秒。如果连续读取不到Binlog数据，则休眠时间为：binlogSleepTime * 次数，最大为binlogMaxSleepTime。读取到数据后，则重置。	Long	500
binlogMaxSleepTime	实时消费不到Binlog数据时最大休眠时间，单位毫秒。	Long	10000
binlogMaxRetryTimes	消费Binlog数据出错后的重试次数。	Integer	1
binlogRetryInterval	消费binlog数据出错后的重试时间间隔。重试时sleep时间：binlogRetryInterval * (1~binlogMaxRetryTimes) +Random（100）。单位毫秒。	Long	100
binlogParallelNum	消费Binlog数据时线程数，只有任务并发度小于DWS集群DN数时，该参数才有效，即此时一个并发度会消费多个DN上的数据，所以可以考虑设置该参数。	Integer	3
connectionPoolSize	JDBC连接池连接大小。	Integer	5
needRedistribution	表示是否兼容扩充重分布（需要升级到对应内核版本，如果是低版本则设置为false）；如果设置成true的话，flink的restart-strategy不能设置为none。	Boolean	true

参数	说明	数据类型	默认值
newSystemValue	表示读取binlog数据时是否使用新的系统字段（需要升级到对应内核版本，如果是低版本则设置为false）。	Boolean	true
checkNodeChangeInterval	检测节点变化的间隔，只有needRedistribution=true才生效。	Long	10000
connectionSocketTimeout	连接处理超时时间（可以看成客户端执行SQL超时时间），单位毫秒；默认值为0，即不设置超时时间。	Integer	0
binlogIgnoreUpdateBefore	是否过滤Binlog记录中的before_update记录，以及delete记录是否只返回主键信息。该参数仅9.1.0.200及以上版本支持。	Boolean	false
binlogStartTime	设置从某个时间点开始消费Binlog（只能增量消费），格式为yyyy-MM-dd hh:mm:ss且表需要开启**enable_binlog_timestamp**。 该参数仅9.1.0.200及以上版本支持。	String	无
binlogSyncPointSize	增量读取binlog同步点区间的大小（增量读取binlog时，如果数据量过大可能涉及下盘，可通过调整该参数控制）。 该参数仅9.1.0.200及以上版本支持。	Integer	5000

数据同步示例

- GaussDB(DWS)侧：

  说明

  新建binlog表时，**enable_hstore_binlog_table**参数需要设置为true，可以通过show enable_hstore_binlog_table来查询。

  -- 源表（产生binlog）

  ```
  CREATE TABLE test_binlog_source(a int, b int, c int, primary key(a)) with(orientation=column, enable_hstore_opt=on, enable_binlog=true);
  ```

  -- 目标表

  ```
  CREATE TABLE test_binlog_sink(a int, b int, c int, primary key(a)) with(orientation=column, enable_hstore_opt=on);
  ```

- Flink侧：

  执行如下命令进行完整数据同步：

  ```
  -- 建立源表的映射表
  CREATE TABLE test_binlog_source (
    a int,
  ```

```
  b int,
  c int,
  primary key(a) NOT ENFORCED
) with (
  'connector' = 'dws',
  'url' = 'jdbc:gaussdb://ip:port/gaussdb',
  'binlog' = 'true',
  'tableName' = 'test_binlog_source',
  'binlogSlotName' = 'slot',
  'username'='xxx',
  'password'='xxx');

-- 建立目标表的映射表
CREATE TABLE test_binlog_sink (
  a int,
  b int,
  c int,
  primary key(a) NOT ENFORCED
) with (
  'connector' = 'dws',
  'url' = 'jdbc:gaussdb://ip:port/gaussdb',
  'tableName' = 'test_binlog_sink',
  'ignoreUpdateBefore'='false',
  'connectionSize' = '1',
  'username'='xxx',
  'password'='xxx');

INSERT INTO test_binlog_sink select * from test_binlog_source;
```

使用 java 程序示例

新建源表和目标表：

```
-- source
create table binlog_test_source(a int, b int, c int, primary key(a)) with(orientation=column,
enable_hstore_opt=on, enable_binlog=true);
-- sink
create table binlog_test_sink(a int, b int, c int, primary key(a)) with(orientation=column,
enable_hstore_opt=on, enable_binlog=true);
```

demo程序：

```
public class BinlogDemo {

    // binlog表的表名
    private static final String BINLOG_TABLE_NAME = "binlog_test_source";

    // binlog表的槽位名
    private static final String BINLOG_SLOT_NAME = "binlog_test_slot";

    // 写入表的表名
    private static final String SINK_TABLE_NAME = "binlog_test_sink";

    public static void main(String[] args) throws Exception {
        DwsConfig dwsConfig = buildDwsConfig();
        DwsClient dwsClient = new DwsClient(dwsConfig);

        TableSchema sourceTableSchema =
dwsClient.getTableSchema(TableName.valueOf(BINLOG_TABLE_NAME));
        TableSchema sinkTableSchema = dwsClient.getTableSchema(TableName.valueOf(SINK_TABLE_NAME));

        // 需要写入哪些列
        List<String> sinkColumns = sinkTableSchema.getColumnNames();

        // 线程池
        DwsConnectionPool dwsConnectionPool = new DwsConnectionPool(dwsConfig);
        // 存放数据的队列
        BlockingQueue<BinlogRecord> queue = new LinkedBlockingQueue<>();
        // 需要同步哪些列
```

```
        List<String> sourceColumnNames = sourceTableSchema.getColumnNames();

        BinlogReader binlogReader = new BinlogReader(dwsConfig, queue, sourceColumnNames,
dwsConnectionPool);

        // 启动读取任务
        binlogReader.start();
        binlogReader.getRecords();

        while (binlogReader.isStart()) {
            try {
                while (!queue.isEmpty() && !binlogReader.hasException()) {
                    // 读取数据
                    BinlogRecord record = queue.poll();
                    if (Objects.isNull(record)) {
                        continue;
                    }
                    BinlogRecordType type = BinlogRecordType.toBinlogRecordType(record.getType());
                    List<Object> columnValues = record.getColumnValues();

                    // 写入数据
                    if (BinlogRecordType.INSERT.equals(type) || BinlogRecordType.UPDATE_AFTER.equals(type)) {
                        Operate upsert = dwsClient.write(sinkTableSchema);
                        for (int i = 0; i < sinkColumns.size(); i++) {
                            upsert.setObject(i, columnValues.get(i), false);
                        }
                        upsert.commit();
                    } else if (BinlogRecordType.DELETE.equals(type) ||
BinlogRecordType.UPDATE_BEFORE.equals(type)) {
                        Operate delete = dwsClient.delete(sinkTableSchema);
                        for (int i = 0; i < sinkColumns.size(); i++) {
                            String field = sinkColumns.get(i);
                            if (!sinkTableSchema.isPrimaryKey(field)) {
                                continue;
                            }
                            delete.setObject(i, columnValues.get(i), false);
                        }
                        delete.commit();
                    }
                }
                binlogReader.checkException();
            } catch (Exception e) {
                throw new DwsClientException(ExceptionCode.GET_BINLOG_ERROR, "get binlog  has error", e);
            }
        }
    }

    private static DwsConfig buildDwsConfig() {
        // 初始化一些配置信息(只列举一些必要的配置,具体配置信息请参考文档)
        TableConfig tableConfig = new TableConfig().withBinlog(true)
                .withNewSystemValue(true)
                .withNeedRedistribution(false)
                .withBinlogSlotName(BINLOG_SLOT_NAME);
        return DwsConfig.builder()
                .withUrl("链接信息")
                .withUsername("用户名")
                .withPassword("密码")
                .withBinlogTableName(BINLOG_TABLE_NAME)
                .withTableConfig(BINLOG_TABLE_NAME, tableConfig)
                .build();
    }
}
```

