数据仓库服务

故障排除

文档版本 31

发布日期 2025-01-03

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。

目 录

1 数据库连接管理

1.1 执行 gsql 连接数据库命令提示 gsql：command not found

问题现象

执行gsql -d postgres -p 26000 -r出现如下错误：

```
gsql：command not found…
```

原因分析

- 没有在gsql的bin目录下执行。
- 未执行环境变量。

处理方法

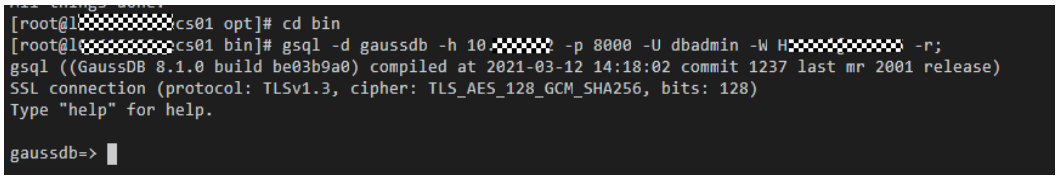
步骤1 在客户端目录下执行环境变量，例如客户端在/opt目录下。

```
cd /opt
source gsql_env.sh
```

```
[root@l[redacted]1 opt]# ll
total 16300
drwxr-xr-x 2 root root     4096 Mar 12  2021 bin
-rw-r--r-- 1 root root 16668016 May  6 14:41 dws_client_8.1.x_redhat_x64.zip
drwxr-xr-x 5 root root     4096 Mar 12  2021 gds
-rwxr-xr-x 1 root root     1465 Mar 12  2021 gsql_env.sh
drwxr-xr-x 3 root root     4096 Mar 12  2021 lib
drwxr-xr-x 3 root root     4096 Mar 12  2021 sample
[root@l[redacted]1 opt]# source gsql_env.sh
Configuring LD_LIBRARY_PATH and PATH for gsql ............... done
All things done.
[root@l[redacted] opt]#
```

步骤2 进入gsql的bin目录下，执行gsql命令进行数据库连接。

```
cd bin
gsql -d gaussdb -h 数据库IP -p 8000 -U dbadmin -W 数据库用户密码 -r;
```



----结束

1.2 通过 gsql 客户端无法连接数据库

问题现象

用户通过客户端工具gsql无法连接到数据库。

原因分析

- 系统连接数量超过了最大连接数量，会显示如下错误信息。
  ```
  gsql -d human_resource -h 10.168.0.74 -U user1 -p 8000 -W password -r
  gsql: FATAL:  sorry, too many clients already
  ```
- 用户不具备访问该数据库的权限，会显示如下错误信息。
  ```
  gsql -d human_resource -h 10.168.0.74 -U user1 -p 8000 -W password -r
  gsql: FATAL:  permission denied for database "human_resource"
  DETAIL:  User does not have CONNECT privilege.
  ```
- 网络连接故障。

解决办法

- 系统连接超过最大连接数量。

  用户可在GaussDB(DWS) 控制台设置最大连接数max_connections。

  max_connections设置方法如下：

  a. 登录GaussDB(DWS) 管理控制台。
  b. 在左侧导航栏中，单击“集群管理”。
  c. 在集群列表中找到所需要的集群，单击集群名称，进入集群“基本信息”页面。
  d. 单击“参数修改”页签，修改参数“max_connections”的值，然后单击“保存”。
  e. 在“修改预览”窗口，确认修改无误后，单击“保存”。

  关于查看用户会话连接数的方法如**表1-1**。

表 1-1 查看会话连接数

描述	命令
查看指定用户的会话连接数上限。	执行如下命令查看连接到指定用户user1的会话连接数上限。其中，-1表示没有对用户user1设置连接数的限制。 SELECT ROLNAME,ROLCONNLIMIT FROM PG_ROLES WHERE ROLNAME='user1'; rolname | rolconnlimit ---------+-------------- user1 | -1 (1 row)
查看指定用户已使用的会话连接数。	执行如下命令查看指定用户user1已使用的会话连接数。其中，1表示user1已使用的会话连接数。 SELECT COUNT(*) FROM V$SESSION WHERE USERNAME='user1'; count ------- 1 (1 row)
查看指定数据库的会话连接数上限。	执行如下命令查看连接到指定数据库db_demo的会话连接数上限。其中，-1表示没有对数据库db_demo设置连接数的限制。 SELECT DATNAME,DATCONNLIMIT FROM PG_DATABASE WHERE DATNAME='db_demo'; datname | datconnlimit ----------+-------------- db_demo | -1 (1 row)
查看指定数据库已使用的会话连接数。	执行如下命令查看指定数据库db_demo上已使用的会话连接数。其中，1表示数据库db_demo上已使用的会话连接数。 SELECT COUNT(*) FROM PG_STAT_ACTIVITY WHERE DATNAME='db_demo'; count ------- 1 (1 row)
查看所有用户已使用会话连接数。	执行如下命令查看所有用户已使用的会话连接数。 SELECT COUNT(*) FROM PG_STAT_ACTIVITY; count ------- 10 (1 row)

- 用户不具备访问该数据库的权限。
  a. 使用管理员用户dbadmin连接数据库。
     gsql -d *human_resource* -h *10.168.0.74* -U *dbadmin* -p 8000 -W *password* -r
  b. 赋予user1用户访问数据库的权限。
     GRANT CONNECT ON DATABASE human_resource TO user1;

说明

实际上，常见的许多错误操作也可能产生用户无法连接上数据库的现象。例如，用户连接的数据库不存在，用户名或密码输入错误等。这些错误操作在客户端工具会有相应的提示信息。

```
gsql -d human_resource -p 8000
gsql: FATAL:  database "human_resource" does not exist

gsql -d human_resource -U user1 -W password -p 8000
gsql: FATAL:  Invalid username/password,login denied.
```

- 网络连接故障。

  请检查客户端与数据库服务器间的网络连接。如果发现从客户端无法PING到数据库服务器端，则说明网络连接出现故障。请联系技术支持工程师提供技术支持。

```
ping -c 4 10.10.10.1
PING 10.10.10.1 (10.10.10.1) 56(84) bytes of data.
From 10.10.10.1: icmp_seq=2 Destination Host Unreachable
From 10.10.10.1 icmp_seq=2 Destination Host Unreachable
From 10.10.10.1 icmp_seq=3 Destination Host Unreachable
From 10.10.10.1 icmp_seq=4 Destination Host Unreachable
--- 10.10.10.1 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 2999ms
```

1.3 连接 GaussDB(DWS)数据库时，提示客户端连接数太多

问题现象

连接GaussDB(DWS) 数据库时报错，提示客户端连接数太多。

- 使用gsql等SQL客户端工具连接数据库时，出现如下报错信息：

```
FATAL: Already too many clients, active/non-active/reserved: 5/508/3.
```

- 使用客户端并发连接数据库时，出现如下报错信息：

```
[2019/12/25 08:30:35] [ERROR] ERROR: pooler: failed to create connections in parallel mode for thread 140530192938752, Error Message: FATAL:  dn_6001_6002: Too many clients already, active/non-active: 468/63.
FATAL: dn_6001_6002: Too many clients already, active/non-active: 468/63.
```

原因分析

1. **当前数据库连接已经超过了最大连接数。**

   错误信息中，non-active的个数表示空闲连接数，例如，non-active为508，说明当前有大量的空闲连接。

2. **创建用户时设置了该用户的最大连接数**。

   查询数据库连接数，如果显示连接数未达设定上限，可能是由于创建用户时设置了该用户的最大连接数。

处理方法

可优先通过如下方法进行应急处理：

1. 临时将所有non-active的连接释放掉。

```
SELECT PG_TERMINATE_BACKEND(pid) from pg_stat_activity WHERE state='idle';
```

2. 在GaussDB(DWS)控制台设置会话闲置超时时长session_timeout，在闲置会话超过所设定的时间后服务端将主动关闭连接。

   session_timeout默认值为600秒，设置为0表示关闭超时限制，一般不建议设置为0。

session_timeout设置方法如下：

a. 登录GaussDB(DWS) 管理控制台。
b. 在左侧导航栏中，单击“集群管理”。
c. 在集群列表中找到所需要的集群，单击集群名称，进入集群“基本信息”页面。
d. 单击“参数修改”页签，修改参数“session_timeout”，然后单击“保存”。
e. 在“修改预览”窗口，确认修改无误后，单击“保存”。

如果上述方式未能满足业务需求，可继续参考以下方式进行处理：

当前数据库连接已经超过了最大连接数场景处理方式：

1. 查看CN上的连接来自哪里，总数量以及是否超过当前max_connections（默认值CN节点为800，DN节点为5000）。
   ```
   SELECT coorname, client_addr, count(1) FROM pgxc_stat_activity group by coorname, client_addr order by coorname;
   ```
2. 判断是否可以调大max_connections，调整策略如下：
   - 调大CN的max_connections会造成并发连接到DN的查询变多，所以需要同步调大DN的max_connections和comm_max_stream。
   - CN/DN的max_connections一般按照原来的2倍调大，原值比较小的集群可以调节到4倍。
   - 为避免在准备步骤失败，max_prepared_transactions的值不能小于max_connections，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。
3. 若上一步中判断可以调整max_connections，则可通过管理控制台调整最大连接数max_connections。

   在管理控制台上，集群“基本信息”页面，单击“参数修改”页签，修改参数“max_connections”为合适的值，然后单击“保存”。

针对设置了用户最大连接数的场景处理方式：

在创建用户时由CREATE ROLE命令的CONNECTION LIMIT connlimit子句直接设定，也可以在设定以后用ALTER ROLE的CONNECTION LIMIT connlimit子句修改。

1. 使用PG_ROLES视图查看指定用户的最大连接数。
   ```
   SELECT ROLNAME,ROLCONNLIMIT FROM PG_ROLES WHERE ROLNAME='role1';
    rolname | rolconnlimit
   ---------+--------------
    role1   |           10
   (1 row)
   ```
2. 修改用户的最大连接数。
   ```
   ALTER ROLE role1 connection limit 20;
   ```

1.4 无法 ping 通集群访问地址

问题现象

在客户端主机上，无法ping通GaussDB(DWS)集群访问地址。

原因分析

- **网络不通**

  如果客户端主机通过GaussDB(DWS)集群的内网地址进行连接，需要排查客户端主机跟GaussDB(DWS)集群是否在相同的VPC和子网内，如果不在相同的VPC和子网内，则网络不通。

- **安全组规则禁止ping**

  GaussDB(DWS)集群所属的安全组入规则需要放开ICMP协议端口才能允许ping，如果未开放ICMP协议端口，就无法ping通。创建GaussDB(DWS)集群时自动创建的安全组默认**只放开了TCP协议和8000端口**。

  如果安全组入规则已开放ICMP协议端口，需要检查相应入规则的源地址是否涵盖了客户端主机的IP地址，如果没有，也无法ping通。

处理方法

- **网络不通**

  如果客户端主机通过GaussDB(DWS)集群的内网地址进行连接，应重新申请一台弹性云服务器作为客户端主机，且该弹性云服务器必须和GaussDB(DWS)集群处于相同的VPC和子网内。

- **安全组规则禁止ping**

  检查GaussDB(DWS)集群所属的安全组规则，查看是否对客户端主机的IP地址开放了ICMP协议端口。具体操作如下：

  a. 登录GaussDB(DWS)管理控制台。
  b. 在“集群管理”页面，找到所需要的集群，单击集群名称进入“基本信息”页面。
  c. 在“基本信息”页面，找到“安全组”参数，单击安全组名称，进入相应的安全组详情页面。
  d. 进入“入方向规则”页签，检查是否存在开放ICMP协议端口的入规则，如果不存在，请单击“添加规则”按钮，添加入方向规则开放ICMP协议端口。

     - 协议端口：选择“ICMP”和“全部”。
     - 源地址：选择“IP地址”，然后根据客户端主机的IP地址输入相应的IP地址与掩码。0.0.0.0/0表示任意地址。

图 1-1 入方向规则

e. 单击“确定”，完成入规则的添加。

1.5 业务执行中报错：An I/O error occurred while sending to the backend

问题现象

使用客户端连接GaussDB(DWS)执行业务过程中出现报错“An I/O error occurred while sending to the backend.”。

原因分析

客户端与数据库之前建好的连接再使用时已经中断导致上述报错，连接中断通常有两种情况：

- 数据库侧主动断开。

  当数据库侧由于某种原因将连接断开，应用侧再次使用此连接就会产生该报错。而数据库侧连接异常断开主要有三种原因：

  a. CN进程异常重启。

  b. session超时。

  c. 用户手动执行命令终止了session。

- 客户端主动断开连接。

处理方法

数据库侧主动断开。针对以上三种情况的原因分析，对应的处理的方式如下：

1. 查看CN进程是否异常重启：

   ```
   ps -eo pid,lstart,etime,cmd | grep coo
   ```

   如果连接在CN进程启动之前就存在，那么CN进程重启之后，连接就会断开，业务侧继续使用则会报错。

2. 会话设置了session_timeout（默认值10min，0表示关闭超时设置。）时间，当超过此时间，数据库会自动清理连接。对于需要保持长连接的场景。可通过客户端对此session设置session_timeout为预期时长。设置方式可参考**session_timeout设置导致JDBC业务报错**。

3. 排查CN日志是否含有due to。

根据日志中相应时间分析，是否有用户手动执行select pg_terminate_backend(pid); 终止了会话，大多数情况是用户操作产生。

```
ERROR: dn_6003_6004: abort transaction due to concurrent update test 289502.
FATAL: terminating connection due to administrator command
```

客户端侧断开连接的处理方式：

如果经过数据库侧的排查，未发现数据库侧主动断开，则可能是客户端侧断开的情况。

1. 可以排查客户端是否设置超时参数socketTimeout，调整该参数为合适的值。
2. 排查网络，是否有网络问题导致连接中断。

2 JDBC/ODBC 类

2.1 JDBC 问题定位

JDBC（Java Database Connectivity，java数据库连接）是应用程序访问数据库的统一标准接口，应用程序可使用JDBC连接数据库并执行SQL。 GaussDB(DWS)提供了对JDBC 4.0特性的支持，本章节提供了JCDB常见问题定位及对应报错和问题的处理方法。

产生JDBC问题的原因主要分为以下三个方面：

- 应用程序和应用程序框架问题。
- JDBC业务功能问题。
- 数据库配置问题。

JDBC问题在具体业务中的表现主要分为以下三个方面：

- 执行报错，JDBC抛出异常。
- 执行效率低，耗时异常。
- 特性不支持，JDBC未实现的JDK接口。

JDBC问题具体分类可参见**表2-1**。

表 2-1 JDBC 问题分类

问题分类	问题原因
建立数据库连接失败	JDBC客户端配置问题：包括URL格式不对，或用户名密码错误。
	网络不通。
	Jar包冲突。
	数据库配置问题，数据库未配置远程访问权限。
执行业务抛出异常	传入SQL有误，GaussDB(DWS)不支持。
	业务处理异常，返回异常报文。

问题分类	问题原因
	网络故障。
	数据库连接超时，socket已关闭。
性能问题	SQL执行慢。
	结果集过大，导致应用程序端响应慢。
	用户传入SQL过长，JDBC解析慢。
功能支持问题	JDK未提供标准接口。
	JDBC未实现接口。

2.2 建立数据库连接失败

Check that the hostname and port are correct and that the postmaster is accepting TCP/IP connections.

问题分析：客户端与服务端网络不通、端口错误或待连接CN异常。

处理方法：

- 客户端ping服务端IP，查看网络是否畅通，网络不通则需解决网络问题。
- 检查URL中连接CN的端口是否正确，若端口不正确则需修改为正确的端口（默认为8000）。

FATAL: Invalid username/password,login denied.

问题分析：用户名或密码配置错误。

处理方法：检查用户名密码是否为数据库用户名和密码，将其修改为正确的数据库用户名和密码。

No suitable driver found for XXXX

问题分析：通过JDBC建连时URL格式错误。

处理方法：将URL格式修改为正确的格式。

- gsjdbc4.jar对应URL格式为： jdbc:postgresql://host:port/database
  - 在使用pom依赖时对应8.1.x版本
- gsjdbc200.jar对应URL格式为：jdbc:gaussdb://host:port/database
  - 在使用pom依赖时对应8.1.x-200版本

conflict

问题分析：JDBC jar包和应用程序冲突。例如JDBC和应用程序拥有相同路径相同名称的类导致：

- gsjdbc4.jar和开源postgresql.jar冲突，两者具有完全相同的类名。
- gsjdbc4.jar由于IAM特性引入了一些其他工具，例如fastjson，和应用程序中的fastjson冲突。

处理方法：

- 针对和开源postgresql.jar的冲突，DWS提供了gsjdbc200.jar，使用和开源驱动不同的url格式和驱动路径，驱动名由**org.postgresql.Driver**修改为**com.huawei.gauss200.jdbc.Driver**，URL格式由**org:postgresql://host:port/database**改为**jdbc:gaussdb://host:port/database**，彻底解决了和开源jar包的冲突。
- 针对JDBC引入的jar和应用程序中引入jar的冲突，可以通过maven的**shade**修改了jar中类的路径，解决此类冲突。
- 排查使用的JDBC驱动是gsjdbc4.jar还是gsjdbc200.jar，如果是gsjdbc4.jar应该替换为gsjdbc200.jar，尝试建立连接。

  说明

  对于pom依赖，将8.1.x版本替换为8.1.x-200版本。

org.postgresql.util.PSQLException: FATAL: terminating connection due to administrator command Session unused timeout

问题分析：会话超时导致连接断开。

处理方法：排查CN和客户端JDBC上的超时配置，按业务实际情况调长超时时间或关闭超时设置。

1. 查看报错的CN日志，如果有session unused timeout这样的日志，说明是会话超时导致的。

   解决办法：

   a. 登录控制台单击指定集群名称。

   b. 在左侧导航栏选择“参数修改”，搜索参数“session_timeout”查看超时时间参数。

   c. 将session_timeout的CN、DN参数值设置为0，详情可参见**修改数据库参数**。

Connection refused: connect.

问题分析：第三方工具的默认驱动不兼容。

处理方法：用户可更换JDBC驱动包，查看是否正常连接。

Connections could not be acquired from the underlying database!

问题分析：按照新建连接排查项进行排查：

- 驱动配置是否有误。
- 数据库连接地址是否有误。
- 密码或账号是否有误。
- 数据库未启动或无权访问。
- 项目未引入对应的驱动jar包。

处理方法：

- 排查驱动配置，将其修改为正确的驱动配置。
  - gsjdbc4.jar driver=org.postgresql.Driver
  - gsjdbc200.jar driver=com.huawei.gauss200.jdbc.Driver
- 排查数据库连接地址，将其修改为正确的数据库连接地址。
  - gsjdbc4.jar对应jdbc:postgresql://host:port/database
  - gsjdbc200.jar对应jdbc:gaussdb://host:port/database
- 排查用户名密码是否为数据库用户名或密码，将其修改为正确的数据库用户名或密码。
- 排查数据库是否启动或有权限访问。
- 检查使用的JDBC驱动是gsjdbc4.jar还是gsjdbc200.jar，请使用正确JDBC驱动jar包。
  - gsjdbc4.jar：与PostgreSQL保持兼容，其中类名、类结构与PostgreSQL驱动完全一致，曾经运行于PostgreSQL的应用程序可以直接移植到当前系统中使用。
  - gsjdbc200.jar：如果同一JVM进程内需要同时访问PostgreSQL及GaussDB(DWS) 请使用该驱动包。该包主类名为“com.huawei.gauss200.jdbc.Driver”（即将“org.postgresql”替换为“com.huawei.gauss200.jdbc”），数据库连接的URL前缀为“jdbc:gaussdb”，其余与gsjdbc4.jar相同。

JDBC DEV 环境没问题，测试环境连接出错报空指针或 URI 报错 uri is not hierarchical

问题分析：某些虚拟环境不支持获取扩展参数，需关闭。

处理方法：在连接时可设置连接参数“connectionExtraInfo=false”，详情可参见**使用JDBC连接数据库**。

```
jdbc:postgresql://host:port/database?connectionExtraInfo=false
```

使用开源 JDBC SSL 方式连接 DWS 报错

问题分析：使用开源JDBC会进行SSL全校验，校验url是否完全匹配。

处理方法：在开源连接时可设置连接参数“sslmode=require”。

```
jdbc:postgresql://host:port/database?sslmode=require
```

在 CopyManager 场景使用连接池获取连接，Connection 无法转换为 BaseConnection

问题分析：BaseConnection为非公开类，需要对连接池对象解封装然后获取原始PGConnection。

处理方法： 对当前对象解包装，返回原始对象以允许访问未公开方法。

```
//解封装
PGConnection unwrap = connection.unwrap(PGConnection.class);
//转换BaseConnection
BaseConnection baseConnection = (BaseConnection)unwrap;
CopyManager copyManager = new CopyManager(baseConnection);
```

2.3 执行业务抛出异常

Broken pipe, connection reset by peer

问题分析：网络故障，数据库连接超时。

处理方法：检查网络状态，修复网络故障，影响数据库连接超时的因素。例如，数据库参数session_timeout。

步骤1 登录控制台单击指定集群名称。

步骤2 在左侧导航栏选择“参数修改”，搜索参数“session_timeout”查看超时时间参数。

步骤3 将session_timeout的CN、DN参数值设置为0，详情可参见**修改数据库参数**。

----结束

The column index is out of range

问题分析：应用程序获取的结果集和预期不一致，列数不一致。

处理方法：检查数据库表定义和查询SQL，对返回结果集做一个正确预期。例如结果集只有3列，取值时传入的index最大为3。

SQL 包含大量参数导致报错：Tried to send an out-of-range integer as a 2-byte value

问题分析：JDBC协议规定，变量总数不能超过32767，short Int最大值。

处理方法：

数据查询：建议将大SQL进行拆分，确保每个SQL变量数小于32767。

数据导入：建议分批导入或使用copymanager，参考**CopyManager**。

调用存储过程报错 ERROR: cached plan must not change result type

问题分析：由于JDBC中使用了PreparedStatement，默认重复执行5次就会缓存plan，在此之后有如果重建表操作（例如修改表定义），再次执行就会报错ERROR: cached plan must not change result type.

处理方法：在JDBC连接字串中指定prepareThreshold=0，不再cache plan。例如：

```
String url = "jdbc:postgresql:// 192.168.0.10:2000/postgres?prepareThreshold=0";
```

使用 JDBC 执行 sql 语句报错 ERROR: insufficient data left in message

问题分析：服务端无法处理字符串中的'\0'字符，'\0'即数值0x00、UTF编码'\u0000'的字符串。

处理方法：排查客户执行的SQL中是否包含'\0'字符，去掉特殊字符，可用空格代替。

使用 JDBC 执行 create table as 语句报错 ERROR: relation "xx" already exists

问题分析：JDBC调用preparedStatement.getParameterMetaData()时会发送P报文，该报文会在数据库中创建表，导致execute执行时报表已存在。

处理方法：使用preparedStatement时，建议将CREATE TABLE AS拆开执行或者使用resultSet.getMetaData()。

2.4 性能问题

在 processResult 阶段耗时

设置loglevel=3，打开JDBC日志，主要耗时在processResult阶段，可分为两种情况：

1. JDBC端等待数据库返回的报文时间过长。

   问题分析：用户可查看FE=> Syncr日志和<=BE ParseComplete日志之间的时间间隔，如果时间间隔较久，则判断为数据库执行慢。

   处理方法：分析SQL执行慢的原因，详情可参见**SQL执行很慢，性能低，有时长时间运行未结束**。

2. 结果集过大，一次性全部加载，消耗大量时间。

   问题分析：查看日志，如果<=BE DataRow日志出现次数过多，或直接执行**SELECT count(*);**查询结果数目过大，则判断为结果集过大。

   处理方法：设置fetchSize参数为一个较小的值，使数据按批次返回，客户端得到快速响应。

   ```
   statement.setFetchSize(10);
   ```

在 modifyJdbcCall 和 createParameterizedQuery 阶段耗时

问题分析：如果主要耗时在modifyJdbcCall阶段（校验传入的SQL是否符合规范）和createParameterizedQuery阶段（将传入的SQL解析为preparedQuery，以获取由simplequery组成的subqueries），则需要确认是否传入的SQL过长导致。

处理方法：JDBC本身没办法优化这部分耗时，可在应用端查看是否可优化传入的SQL语句，详情可参见**SQL语句改写规则**。

2.5 功能支持问题

not yet implemented

问题分析：JDBC未实现接口。

处理方法：需要技术人员研究接口是否可实现，或是否有其他接口已提供相同功能，调整业务使用已提供接口。

JDK 标准接口中未提供相关功能

问题分析：JDK未提供标准接口。

处理方法：理论上如果JDK未提供接口，则JDBC不支持。实际使用中可以使用JDBC类中的public方法获取部分过程数据，绝大部分情况下明确不支持。

3 数据导入/导出

3.1 使用 COPY FROM 导入时报错“invalid byte sequence for encoding "UTF8": 0x00”

问题现象

使用COPY FROM导入GaussDB(DWS)时，报错: “ invalid byte sequence for encoding "UTF8": 0x00.”。

原因分析

业务数据文件从Oracle导入，文件编码为utf-8。该报错还会提示行数，由于文件特别大，vim命令打不开文件，于是用sed命令把报错行数提出来，再用vim命令打开，发现并没有什么异常。用split命令按行数切割后，部分文件也可以导入。

经分析GaussDB(DWS)的varchar型的字段或变量不允许含有'\0'（也即数值0x00、UTF编码'\u0000'）的字符串 ，需在导入前去掉字符串中的'\0'。

处理方法

用sed命令替换0x00后，即可成功导入。

```
sed -i 's/\x00//g;' file
```

参数说明：

- -i表示在原文件直接替换。
- s/表示替换。
- /g表示全局替换。

3.2 GDS 导入/导出类问题

GDS导入/导出容易遇到字符集的问题，特别是不同类型的数据库或者不同编码类型的数据库进行迁移的过程中，往往会导致数据入不了库，严重阻塞数据迁移场景相关业务。

区域支持

区域支持指的是应用遵守文化偏好的问题，包括字母表、排序、数字格式等。区域是在使用initdb创建一个数据库时自动被初始化的。默认情况下，initdb将会按照它的执行环境的区域设置初始化数据库，即系统已经设置好的区域。如果想要使用其他的区域，可以使用手工指定（initdb –locale=xx）。

如果想要将几种区域的规则混合起来，可以使用以下区域子类来控制本地化规则的某些方面。这些类名转换成initdb的选项名来覆盖某个特定分类的区域选择。

表 3-1 区域支持

字段	描述
LC_COLLATE	字符串排序顺序。
LC_CTYPE	字符分类（什么是一个字符？它的大写形式是否等效？）
LC_MESSAGES	消息使用的语言Language of messages。
LC_MONETARY	货币数量使用的格式。
LC_NUMERIC	数字的格式。
LC_TIME	日期和时间的格式。

如果想要系统表现得没有区域支持，可以使用区域C或者等效的POSIX。使用非C或非POSIX区域的缺点是性能影响。它降低了字符处理的速度并且阻止了在LIKE中对普通索引的使用。因此，只能在真正需要的时候才使用它。

一些区域分类的值必须在数据库被创建时的就被固定。不同的数据库可以使用不同的设置，但是一旦一个数据库被创建，就不能在数据库上修改这些区域分类的值。LC_COLLATE和LC_CTYPE就属于上述情形。它们影响索引的排序顺序，因此它们必须保持固定，否则在文本列上的索引将会崩溃。这些分类的默认值在initdb运行时被确定，并且这些值在新数据库被创建时使用，除非在CREATE DATABASE命令中特别指定。其它区域分类可以在任何时候被更改，更改的方式是设置与区域分类同名的服务器配置参数。被initdb选中的值实际上只是被写入到配置文件postgresql.conf中作为服务器启动时的默认值。如果你将这些赋值从postgresql.conf中除去，那么服务器将会从其执行环境中继承该设置。

区域设置特别影响下面的SQL特性：

- 在文本数据上使用ORDER BY或标准比较操作符的查询中的排序顺序
- 函数upper、lower和initcap
- 模式匹配操作符（LIKE、SIMILAR TO和POSIX风格的正则表达式）；区域影响大小写不敏感匹配和通过字符类正则表达式的字符分类
- to_char函数家族

因此，在上述场景遇到查询结果集不一致的情况，就可以猜测可能是字符集问题。

排序规则支持

排序规则特性允许指定每一列甚至每一个操作的数据的排序顺序和字符分类行为。这放松了数据库的LC_COLLATE和LC_CTYPE设置自创建以后就不能更改这一限制。

一个表达式的排序规则可以是"默认"排序规则，它表示数据库的区域设置。一个表达式的排序规则也可能是不确定的。在这种情况下，排序操作和其他需要知道排序规则的操作会失败。

当数据库系统必须要执行一次排序或者字符分类时，它使用输入表达式的排序规则。这会在使用例如ORDER BY子句以及函数或操作符调用（如<）时发生。应用于ORDER BY子句的排序规则就是排序键的排序规则。应用于函数或操作符调用的排序规则从它们的参数得来，具体如下文所述。除比较操作符之外，在大小写字母之间转换的函数会考虑排序规则，例如lower、upper和initcap。模式匹配操作符和to_char及相关函数也会考虑排序规则。

对于一个函数或操作符调用，其排序规则通过检查在执行指定操作时参数的排序规则来获得。如果该函数或操作符调用的结果是一种可排序的数据类型，万一有外围表达式要求函数或操作符表达式的排序规则，在解析时结果的排序规则也会被用作函数或操作符表达式的排序规则。

一个表达式的排序规则派生可以是显式或隐式。该区别会影响多个不同的排序规则出现在同一个表达式中时如何组合它们。当使用一个COLLATE子句时，将发生显式排序规则派生。所有其他排序规则派生都是隐式的。当多个排序规则需要被组合时（例如在一个函数调用中），将使用下面的规则：

1. 如果任何一个输入表达式具有一个显式排序规则派生，则在输入表达式之间的所有显式派生的排序规则必须相同，否则将产生一个错误。如果任何一个显式派生的排序规则存在，它就是排序规则组合的结果。
2. 否则，所有输入表达式必须具有相同的隐式排序规则派生或默认排序规则。如果任何一个非默认排序规则存在，它就是排序规则组合的结果。否则，结果是默认排序规则
3. 如果在输入表达式之间存在冲突的非默认隐式排序规则，则组合被认为是具有不确定排序规则。这并非一种错误情况，除非被调用的特定函数要求提供排序规则的知识。如果它确实这样做，运行时将发生一个错误。

字符集

PG里面的字符集支持各种字符集存储文本，包括单字节字符集，比如ISO 8859系列，以及多字节字符集，比如EUC（扩展Unix编码Extended Unix Code）、UTF-8和Mule内部编码。MPPDB中目前主要使用的字符集包括GBK、UTF-8和LATIN1。所有被支持的字符集都可以被客户端透明地使用，但少数只能在服务器上使用（即作为一种服务器端编码，GBK编码在PG中只是客户端编码，不是服务端编码，MPPDB将GBK引入到服务端编码，这是很多问题的根源）。默认的字符集是在使用initdb初始化PG数据库时选择的。在创建一个数据库实例时可以重载字符集，因此可能会有多个数据库实例并且每一个使用不同的字符集。一个重要的限制是每个数据库的字符集必须和数据库LC_CTYPE（字符分类）和LC_COLLATE （字符串排序顺序）设置兼容。对于C或POSIX，任何字符集都是允许的，但是对于其他区域只有一种字符集可以正确工作。不过，在Windows上UTF-8编码可以和任何区域配合使用。

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PG将无法帮助你转换或者校验非ASCII字符。

数据库系统支持某种编码，主要涉及三个方面：数据库服务器支持，数据访问接口支持以及客户端工具支持。

- 数据库服务器字符编码

  数据库服务器支持某种编码，是指数据库服务器能够从客户端接收、存储以及向客户端提供该种编码的字符（包括标识符、字符型字段值），并能将该种编码的字符转换到其它编码（如UTF-8编码转到GBK编码）。

  指定数据库服务器编码：创建数据库时指定：CREATE DATABASE … ENCODING … //可以取ASCII、UTF-8、EUC_CN、……；

  查看数据库编码：show server_encoding。

- 数据库访问接口编码

  数据库访问接口支持某种编码，是指数据库访问接口要做到能对该种编码的字符进行正确读写，不应出现数据丢失、数据失真等情况。以JDBC接口为例：

  JDBC接口一般根据JVM的file.encoding设置client_encoding：set client_encoding to file_encoding；

  将String转换成client_encoding编码的字节流，传给服务器端：原型String.getBytes(client_encoding)；

  收到服务器的字节流后，使用client_encoding构造String对象作为getString的返回值给应用程序：原型String(byte[], …, client_encoding)。

- 客户端编码

  客户端工具支持某种编码，是指客户端工具能够显示从数据库读取该种编码的字符，也能通过本工具将该种编码的字符提交到服务器端。

  指定会话的客户端编码：SET CLIENT_ENCODING TO 'value';

  查看数据库编码：Show client_encoding。

GDS 导入/导出遇到的字符集问题和解决办法

问题一：0x00字符无法入库：ERROR: invalid byte sequence for encoding "UTF8": 0x00

原因：PG本身不允许文本数据中出现0x00字符，基线问题，其他数据库不存在该问题。

解决方法：

1. 替换0x00字符。
2. Copy、GDS都有“compatible_illegal_chars”这个选项，把这个开关打开（COPY命令、GDS外表可Alter），会把单字节/多字节的非法字符替换成“（空格）”/“？”。这样可以顺利导入数据，但会更改原数据。
3. 建立encoding为SQL_ASCII的库，然后client_encoding也设置为SQL_ASCII（COPY命令中可设置，GDS外表也可设置），这种情况下可以避免字符集的特殊处理和转换，所有库内相关的排序、比较以及处理全部按照单字节处理。

问题二： GBK字符无法导入UTF-8库

原因：缺少GBK到UTF-8的转换函数。

解决方法：目前在r8c10已经补充了缺少的转换函数，包含106个字符，会尽快同步到r7c10的发货版本。具体字符，参考如下。

a2e3 : €
a6 d9-df : ， 。 、 ： ； ！ ？
a6 ec ed : ㎎ ㎏
a6 f3 : ¦
a8 bc bf
ḿ ń
a9 89-8f 90-95
fe 50-5f
fe 60-6f
fe 70-7E
fe 80-8f
fe 90-9f
fe a0

问题三：GBK转义符\（0x5C）问题。

原因：gbk本身作为server端编码存在很多问题，违背了PG的设计原则，即ascii码不能作为多字节字符的一部分（多字节字符每个字节的首位必须为1），不遵循这个原则可能会出现误判的问题。例子如下所示，gbk多字节字符可能会使用'\'作为第二个字符。

```
/* Else, it's the traditional escaped style */
for (bc = 0, tp = inputText; *tp != '\0'; bc++)
{
if (tp[0] != '\\')
tp++;
```

上述问题属于PG基线的代码实现问题。当然可以通过枚举所有可能出现的非法情形来解决，但是实现难度比较大，更为关键的是会增加判断逻辑降低处理效率，这也是社区坚决不允许引入gbk、SJIS（日文）等字符集的原因所在。

解决方法：尽量不使用GBK作为server端字符编码，可使用utf-8替换。另外，尽量不使用SQL_ASCII作为server端编码，无论导入的数据是什么字符集，SQL_ASCII都会按单字节入库，内部逻辑也都会按单字节处理，同样会遇到上述问题。GBK中包含0x5C的两字节字符参考如下。

815c乘 825c倥 835c僭 845c刭 855c匾 865c哼 875c嘰 885c坳 895c堍 8a5c獎 8b5c煥 8c5c攣 8d5c峮 8e5c嚼 8f5c廮 905c怵 915c憖 925c抃 935c挨 945c擻 955c晒 965c朵 975c柒 985c榎 995c橦 9a5c歕 9b5c汨 9c5c淩 9d5c溦 9e5c潬 9f5c烜 a05c燻 a85c ┗ a95c ─ aa5c猵 ab5c玕 ac5c瑗 ad5c璯 ae5c甜 af5c疶 b05c癹 b15c眐 b25c瞗 b35c砜 b45c磲 b55c礬 b65c禱 b75c稯 b85c窟 b95c箊 ba5c篦 bb5c籠 bc5c糪 bd5c絓 be5c綷 bf5c線 c05c繶 c15c罶 c25c耟 c35c肯 c45c膆 c55c蕆 c65c芒 c75c苺 c85c算 c95c箔 ca5c蔦 cb5c蘕 cc5c蘚 cd5c蚰 ce5c蜩 cf5c蝻 d05c衆 d15c褊 d25c襖 d35c覾 d45c診 d55c誠 d65c謀 d75c譖 d85c豭 d95c賊 da5c趄 db5c踔 dc5c躝 dd5c軮 de5c鄨 df5c運 e05c郹 e15c醱 e25c鈇 e35c鉢 e45c鋅 e55c錦 e65c鍖 e75c鍥 e85c鐳 e95c閆 ea5c闬 eb5c隲 ec5c靄 ed5c韁 ee5c頫 ef5c颶 f05c餼 f15c駛 f25c驚 f35c骫 f45c鬨 f55c鮿 f65c鰊 f75c鰳 f85c瑪 f95c鵟 fa5c鼕 fb5c鴆 fc5c黑 fd5c齠 fe5c噢

3.3 创建 GDS 外表失败，提示不支持 ROUNDROBIN

问题现象

创建GDS外表失败，提示不支持ROUNDROBIN，报错信息如下所示：


```
ERROR:  For foreign table ROUNDROBIN distribution type is built-in support.
```

原因分析

GDS外表系统内部默认以ROUNDROBIN分布方式创建，不支持在创建外表时显式添加ROUNDROBIN分布信息。

处理方法

在创建GDS外表时，去除指定的分布信息，即去掉语句中显示指定的“DISTRIBUTE BY ROUNDROBIN”即可。

3.4 通过 CDM 将 MySQL 数据导入 GaussDB(DWS)时出现字段超长，数据同步失败

问题现象

MySQL 5.*x*版本字段长度varchar(n)，用CDM同步数据到GaussDB(DWS)，同样设置长度为varchar(n) ，但是会出现字段超长，数据同步失败的问题。

原因分析

- MySQL5.0.3之前varchar(n)这里的n表示字节数。
- MySQL5.0.3之后varchar(n)这里的n表示字符数，比如varchar(200)，不管是英文还是中文都可以存放200个。
- GaussDB(DWS)的varchar(n)这里的n表示字节数。

根据字符集，字符类型如果为gbk，每个字符占用2个字节；字符类型如果为utf8，每个字符最多占用3个字节。根据转换规则，同样的字段长度，会导致GaussDB(DWS)出现字段超长的问题。

处理方法

假设MySQL字段为varchar(n)，则将GaussDB(DWS)对应的字段长度设置为varchar(n*3)即可。

3.5 执行创建 OBS 外表的 SQL 语句时，提示 Access Denied

问题现象

执行创建OBS外表的SQL语句时，返回OBS错误信息，提示访问被拒绝“Access Denied”。

原因分析

- 创建OBS外表语句中的访问密钥AK和SK错误，会出现如下所示的错误信息：
  ```
  ERROR: Fail to connect OBS in node:cn_5001 with error code: AccessDenied
  ```
- 账户OBS权限不足，对OBS桶没有读、写权限，会出现如下所示的错误信息：
  ```
  dn_6001_6002: Datanode 'dn_6001_6002' fail to read OBS object bucket:'obs-bucket-name'
  key:'xxx/xxx/xxx.csv' with OBS error code:AccessDenied message: Access Denied
  ```

默认情况下，您不具备访问其他账号的OBS数据的权限，此外，IAM用户（相当于子用户）也不具备访问其所属账号的OBS数据的权限。

处理方法

- **创建OBS外表语句中的访问密钥AK和SK错误**

  请获取正确的访问密钥AK和SK，写入创建OBS外表的SQL语句中。获取访问密钥的步骤如下：

  a. 登录GaussDB(DWS)管理控制台。
  b. 将鼠标移至右上角的用户名，单击“我的凭证”。
  c. 进入“我的凭证”后，在左侧导航树单击“访问密钥”。
     在访问密钥页面，可以查看已有的访问密钥ID（即AK）。
  d. 如果要同时获取AK和SK，单击“新增访问密钥”创建并下载访问密钥。

- **账户OBS权限不足，对OBS桶没有读、写权限**

  您必须给指定的用户授予所需的OBS访问权限：

  – 通过OBS外表导入数据到GaussDB(DWS)时，执行导入操作的用户必须具备数据源文件所在的OBS桶和对象的**读取**权限。
  – 通过OBS外表导出数据时，执行导出操作的用户必须具备数据导出路径所在的OBS桶和对象的**读取和写入**权限。

  有关配置OBS权限的具体操作，请参见《对象存储服务控制台指南》中的**配置桶ACL**和“配置对象ACL”章节。

3.6 GDS 导入失败后，磁盘占用空间增大

问题背景与现象

使用GDS导入数据失败，触发作业重跑。重新开始数据导入，完成导入作业后查看磁盘空间，发现磁盘占用空间比导入数据量大很多。

原因分析

在导入数据失败后，占用的磁盘空间没有释放。

解决办法

步骤1 检测GDS导入作业的日志，查看是否有执行失败的现象。

步骤2 对表或者分区执行清理操作。

```
vacuum [full] table_name;
```

----结束

3.7 GDS 导入数据时，脚本执行报错：out of memory

问题现象

在使用GDS导入数据时，脚本执行报错“out of memory”。

原因分析

1. 使用copy命令或者导入数据时，源数据单行数据的大小超过1GB。
2. 由于源文件中的格式符不成对出现，比如引号，文件格式异常导致系统识别的单行数据过大超过1GB。

处理方法

1. 确保源文件中的引号是成对的。
2. 检查创建外表时命令中参数的取值、格式设置是否合理。
3. 检查源文件单行数据是否超过1GB，参考报错行号进行检查，可根据实际情况手动调整或删除该行数据。

3.8 使用 GDS 传输数据的过程中，报错：connection failure error

问题现象

在使用GDS传输数据的过程中，报错“connection failure error”。

原因分析

1. GDS进程崩溃。执行命令检查GDS进程是否崩溃：

   ```
   ps ux|grep gds
   ```

   如果返回结果如下，则说明GDS进程启动成功：

2. GDS启动参数-H配置不正确。

   -H *address_string*：允许哪些主机连接和使用GDS服务。参数需为CIDR格式。此参数配置的目的是允许GaussDB(DWS)集群可以访问GDS服务进行数据导入，请保证所配置的网段包含GaussDB(DWS)集群各主机。

处理方法

1. 重新启动GDS。具体步骤参见**安装配置和启动GDS**。
2. 修改GDS启动命令中的–H参数，可以尝试修改成–H参数为0/0，验证是否是该原因。如果修改参数为0/0后命令可以执行，说明原本的参数设置不合理，所配置的网段未包含GaussDB(DWS)集群各主机，需要修改。

3.9 使用 DataArts Studio 服务创建 GaussDB(DWS)外表时不支持中文，如何处理

问题现象

使用DataArts Studio服务创建GaussDB(DWS)的OBS外表，并且在创建外表语句中指定OBS文件编码格式是UTF-8，但是导入数据时报错，如何处理?

原因分析

存储在OBS中的源文件含有非UTF-8的数据。

处理方法

排查报错的源文件，检查是否含有非UTF-8的数据，例如中文字符。如果源文件中含有非UTF-8的数据，请先将源文件转换成UTF-8的格式，并重新上传到OBS，然后再执行导入数据的操作。

4 数据库参数修改

4.1 数据库时间与系统时间不一致，如何更改数据库默认时区

问题现象

数据库时间与操作系统不一致，查询GaussDB(DWS)数据库默认时间SYSDATE，结果数据库时间比北京时间慢8个小时，导致无法准确定位到更新数据。

原因分析

GaussDB(DWS)数据库显示和解释时间类型数值时使用的时区默认为“UTC”。如果操作系统时间所设置的时区不是UTC时区，就会出现GaussDB(DWS)数据库时间和系统时间不一致的情况。通常情况下集群时区不需要进行修改，设置客户端时区可以对SQL执行产生影响。

前提条件

建议在业务低峰期修改“timezone”参数。

处理方法

方法一：更改某个GaussDB(DWS)集群的数据库默认时区。

步骤1 登录GaussDB(DWS)管理控制台。

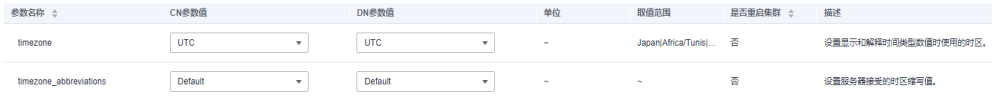
步骤2 在左侧导航栏中，单击“集群管理”。

步骤3 在集群列表中找到所需要的集群，单击集群名称，进入集群“基本信息”页面。

步骤4 单击“参数修改”页签，修改参数“timezone”，修改为您所在的时区，然后单击“保存”。

步骤5 在“修改预览”窗口，确认修改无误后，单击“保存”。

步骤6 用户可根据界面中参数“timezone”所在行的“是否重启”列，判断修改参数后无需进行重启操作。



说明

修改“timezone”参数后无需重启集群操作 ，则修改后立即生效。

----结束

方法二：通过后台命令查询和更改数据库时区。

步骤1 查询客户端时区和当前时间。其中客户端时区为UTC时区，now()函数返回当前时间。

```
SHOW time zone;
 TimeZone
----------
 UTC
(1 row)

select now();
              now
-------------------------------
 2022-05-16 06:05:58.711454+00
(1 row)
```

步骤2 创建数据表，其中timestamp、timestamptz是常用的时间类型。timestamp不保存时区，timestamptz保存时区。

```
CREATE TABLE timezone_test (id int, t1 timestamp, t2 timestamptz) DISTRIBUTE BY HASH (id);

\d timezone_test
          Table "public.timezone_test"
 Column |            Type             | Modifiers
--------+-----------------------------+-----------
 id     | integer                     |
 t1     | timestamp without time zone |
 t2     | timestamp with time zone    |
```

步骤3 向timezone_test表插入当前时间并查询当前表。

```
INSERT INTO timezone_test values (1, now(), now() );
show time zone;
 TimeZone
----------
 UTC
(1 row)

SELECT * FROM timezone_test;
 id |            t1             |            t2
----+---------------------------+-------------------------------
  1 | 2022-05-16 06:10:04.564599 | 2022-05-16 06:10:04.564599+00
(1 row)
```

t1（timestamp类型）在保存数据时丢弃了时区信息，t2（timestamptz类型）保存了时区信息。

步骤4 把客户端时区设置为东8区（UTC-8），再次查询timezone_test表。

```
SET time zone 'UTC-8';
SHOW time zone;
 TimeZone
----------
 UTC-8
(1 row)

SELECT now();
              now
-------------------------------
 2022-05-16 14:13:43.175416+08
(1 row)
```

步骤5 继续插入当前时间到timezone_test表，并查询。此时t1新插入的值使用的是东8区时间，t2根据客户端时区对查询结果进行转换。


```
INSERT INTO timezone_test values (2, now(), now() );
SELECT * FROM timezone_test;
 id |            t1             |            t2
----+---------------------------+-------------------------------
  1 | 2022-05-16 06:10:04.564599 | 2022-05-16 14:10:04.564599+08
  2 | 2022-05-16 14:15:03.715265 | 2022-05-16 14:15:03.715265+08
(2 rows)
```

说明

- timestamp类型只受数据在插入时的时区影响，查询结果不受客户端时区影响。
- timestamptz类型在数据插入时记录了时区信息，查询时会根据客户端时区做转换，以客户端时区显示数据。

----结束

4.2 业务报错：Cannot get stream index, maybe comm_max_stream is not enough

问题现象

用户执行业务报错“ERROR: Failed to connect dn_6001_6002, detail:1021 Cannot get stream index, maybe comm_max_stream is not enough.”。

原因分析

用户数据库的comm_max_datanode参数为默认值1024，在正常批量业务运行时查到DN之间stream数量大约为600~700，当批量任务运行时如果有临时查询，就会超过上限，导致上述报错。

分析过程

1. GUC参数comm_max_stream表示任意两个DN之间stream的最大数量。

   在CN上查询当前任意两个DN之间stream情况：

   ```
   SELECT node_name,remote_name,count(*) FROM pgxc_comm_send_stream group by 1,2 order by 3 desc limit 100;
   ```

   在DN上查询当前DN与其他DN之间stream情况：

   ```
   SELECT node_name,remote_name,count(*) FROM pg_comm_send_stream group by 1,2 order by 3 desc limit 100;
   ```

2. comm_max_stream参数值必须大于并发数*每并发平均stream算子数*（smp的平方）。

   该参数默认值为：通过公式min(query_dop_limit * query_dop_limit * 2 * 20, max_process_memory(字节) * 0.025 /(最大CN数+当前DN数) / 260)计算，小于1024按照1024取值，其中query_dop_limit = 单个机器CPU核数 / 单个机器DN数。

   - 不建议该参数值设置过大，因为comm_max_stream会占用内存（占用内存=256byte*comm_max_stream*comm_max_datanode），如果并发数据流数过大，查询较为复杂及smp过大都会导致内存不足。
   - 如果comm_max_datanode参数值较小，进程内存充足，可以适当将comm_max_stream值调大。

处理方法

根据评估，内存充足，将comm_max_stream参数值调大为2048。（参数值2048仅适用示例场景，请根据实际业务的内存查询结果进行参数值调整。）

步骤1 登录GaussDB(DWS) 管理控制台。

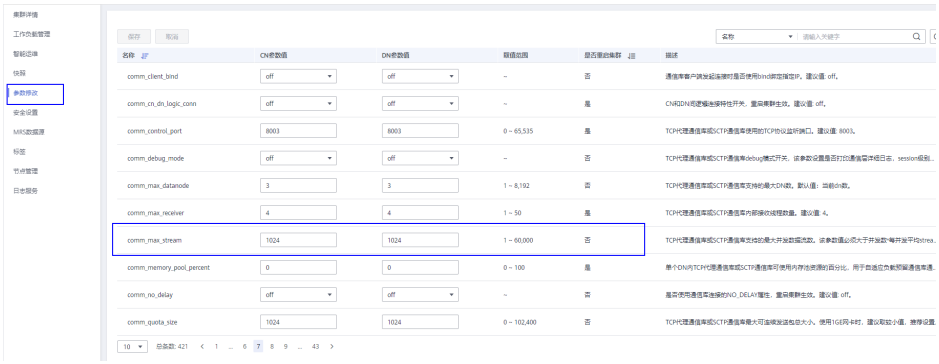
步骤2 在左侧导航栏中，单击“集群管理”。

步骤3 在集群列表中找到所需要的集群，单击集群名称，进入集群“基本信息”页面。

步骤4 单击“参数修改”页签，修改参数“comm_max_stream”，然后单击“保存”。

步骤5 在“修改预览”窗口，确认修改无误后，单击“保存”。

步骤6 参数“comm_max_stream”所在行“是否重启”列显示为“否”，表示该参数修改后无需进行重启操作，即修改后立即生效。



----结束

4.3 SQL 语句执行失败，报错：canceling statement due to statement timeout

问题现象

某SQL语句执行超过两小时，返回如下报错信息：

```
ERROR: canceling statement due to statement timeoutTime.
```

原因分析

当语句执行时间超过**statement_timeout**参数设置的时间时，该语句将会报错并退出执行。

处理方法

方式一：通过控制台修改statement_timeout参数。

步骤1 登录GaussDB(DWS)管理控制台。

步骤2 在左侧导航栏中，单击“集群管理”。

步骤3 在集群列表中找到所需要的集群，单击集群名称，进入集群“基本信息”页面。

步骤4 单击“参数修改”页签，根据需要修改参数“**statement_timeout**”默认值，然后单击“保存”。

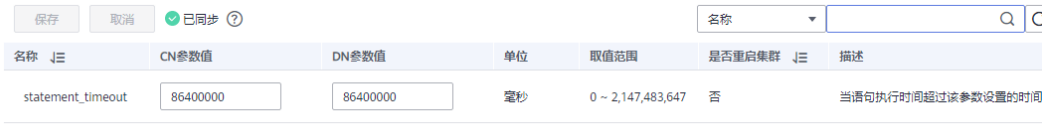
说明

GaussDB(DWS)默认不限制SQL超时，即**statement_timeout**默认值为0。如用户手动修改过该参数，建议恢复原默认值0，或者根据需要调整SQL超时时间，避免设置的SQL超时时间对其它任务产生影响。

步骤5 在“修改预览”窗口，确认修改无误后，单击“保存”。

步骤6 参数“**statement_timeout**”所在行“是否重启”列显示为“否”，表示该参数修改后无需进行重启操作，即修改后立即生效。

图 4-1 修改参数 statement_timeout



----结束

方式二：成功连接集群后，通过SQL命令修改statement_timeout参数。

- 使用SET语句修改（会话级别）：
  SET statement_timeout **TO** *0;*
- 使用ALTER ROLE语句修改（用户级别）：
  ALTER USER *username* **SET** statement_timeout **TO** 600000;
  其中，username为要设置SQL语句超时时间的数据库用户名。

5 账号/权限/密码

5.1 账号被锁住了，如何解锁?

问题现象

连接集群时报错“The account has been locked.”。

原因分析

在连接集群中的数据库时，如果连续输错密码的次数过多，错误次数（输入密码错误的次数由GUC参数failed_login_attempts控制，默认值为10次）达到上限时，会导致账号被锁。

说明

账号被锁的原因可以通过查看审计日志进行定位，详见**重置密码后再次登录仍提示用户被锁**。

管理员用户（默认为 dbadmin）解锁方法

您可以登录GaussDB(DWS) 管理控制台重置管理员密码，重置密码后账号即可自动解锁。在GaussDB(DWS) 管理控制台，进入“集群管理”页面，找到所需要的集群，然后单击“更多 > 重置密码”。

数据库普通用户解锁方法

使用管理员用户（默认为dbadmin）连接数据库，然后执行以下命令进行解锁，其中user_name请替换为需要解锁的用户名：

```
gsql -d gaussdb -p 8000 -U dbadmin -W 密码 -h 集群IP
ALTER USER user_name ACCOUNT UNLOCK;
```

设置尝试登录失败次数

输错密码的次数上限可以在集群的“参数修改”页面通过参数failed_login_attempts进行设置，当failed_login_attempts配置为0时表示不限制密码输入错误的次数，为保证数据库安全，不推荐设置为0。

您可以通过如下步骤修改参数：

1. 登录GaussDB(DWS) 管理控制台。
2. 在左侧导航树，单击“集群管理”。
3. 在集群列表中找到所需要的集群，然后单击集群名称。
4. 进入集群的“参数修改”页面，找到“failed_login_attempts”参数，修改其参数值，然后单击“保存”，确认无误后再单击“保存”。

设置账户被锁定后自动解锁的时间

账户被锁定后，可通过设置参数password_lock_time指定自动解锁的时间，当锁定时间超过password_lock_time设定的值时，账户将会被自行解锁。参数password_lock_time的整数部分表示天数，小数部分可以换算成时、分、秒。

您可以通过如下步骤修改参数：

1. 登录GaussDB(DWS) 管理控制台。
2. 在左侧导航树，单击“集群管理”。
3. 在集群列表中找到所需要的集群，然后单击集群名称。
4. 进入集群的“参数修改”页面，找到“password_lock_time”参数，修改其参数值，然后单击“保存”，确认无误后再单击“保存”。

5.2 重置密码后再次登录仍提示用户被锁

问题现象

某用户连接集群时提示账户被锁，重置用户密码后再次登录，仍提示用户被锁。

```
FATAL: The account has been locked.
```

原因分析

DWS数据库默认情况下，在连续输入错误密码10次后会锁定该用户（允许输入的错误次数由GUC参数failed_login_attempts（默认值为10）控制，可以参见**设置尝试登录失败次数**登录DWS管理控制台进行修改）。

当用户重置密码后，重试还是被锁，说明在重置后，有其他用户或者应用又用错误的密码再次尝试连接了10次或failed_login_attempts的设置值，再次导致了用户被锁。

处理方法

步骤1 使用系统管理员dbadmin用户连接数据库，执行以下SQL语句查看系统时间。

```
SELECT now();
```

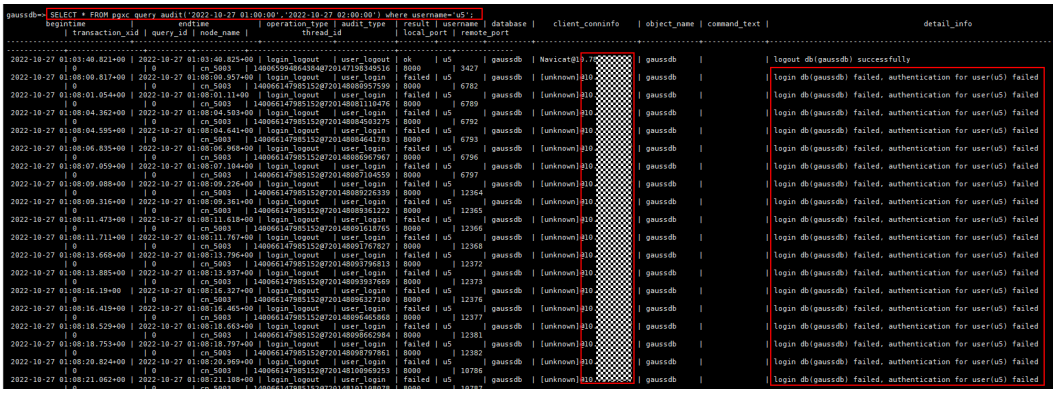
```
gaussdb=> SELECT NOW();
              now
-------------------------------
 2022-10-27 01:14:09.24637+00
(1 row)
```

查询结果显示当前DWS默认显示的系统时间是UTC时间，即北京时间-8H。

步骤2 执行以下SQL语句查询客户端连接情况。其中，

- username请替换为实际被锁的用户名。
- 时间段，请根据实际时间修改。例如要查北京时间当天上午9点到10点这个时间段的连接情况，需由北京时间转为UTC时间，即-8H，则UTC时间为当天上午1点到2点的时间段。

```
SELECT * FROM pgxc_query_audit('2022-10-27 01:00:00','2022-10-27 02:00:00') where
username='username';
```



查询结果显示：IP地址为x.x.x.x的客户端一直使用错误的密码进行连接。

步骤3 根据实际业务情况选择以下对应的处理办法：

- 如果确认步骤2查询到的IP地址是某个作业，可以先停止该作业连接，再通过系统管理员dbadmin连接数据库后，使用以下SQL语句解锁该用户。然后调整该作业设置，使用正确的密码进行连接。
  ```
  ALTER USER username ACCOUNT UNLOCK;
  ```
- 如果不确认该IP地址是哪个作业，可以先参见**设置尝试登录失败次数**将参数failed_login_attempts修改为0，然后使用以下SQL语句重置一个新密码，此时正在连接的其他作业即使再次输错10次密码也不会被锁定。
  ```
  ALTER USER username IDENTIFIED BY '{Password}';
  ```

须知

failed_login_attempts修改为0的方式仅为临时解决方案，为保证数据库安全，不建议failed_login_attempts设置为0。建议后续定位到对应作业后，尽快将作业的连接设置调整为正确密码，继续将failed_login_attempts设置为10。

----结束

5.3 将 Schema 中的表的查询权限赋给其他用户，赋权后仍无法查询 Schema 中的表

问题现象

有权限的用户使用“GRANT SELECT ON all tables in schema *schema_name* TO *u1*”命令给u1用户赋予schema下表的权限后，u1用户仍然无法访问该schema下的表。

原因分析

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，如果没有该权限，则只能看到这些对象的名字，并不能实际进行对象访问。

如果要将该schema下未来创建的表的权限也赋予u1用户，则需使用ALTER DEFAULT PRIVILEGES更改默认权限。

处理方法

请使用具有schema权限的用户登录数据库，执行以下命令将schema中的表权限赋给指定的用户：

```
GRANT USAGE ON SCHEMA schema_name TO u1;
GRANT SELECT ON ALL TABLES IN SCHEMA schema_name TO u1;
```

执行以下命令，将schema中未来新建的表的权限也赋予指定的用户：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA schema_name GRANT SELECT ON TABLES TO u1;
```

上述SQL语句中的“GRANT SELECT”表示赋予的是表的查询权限，如果需要给其他用户赋予表的其他权限，请参考**GRANT**章节。

说明

如果需要给某个用户赋权“可查询数据库所有schema里所有表”，可通过系统表PG_NAMESPACE查询出schema后授权。例如：

```
SELECT 'grant select on all tables in '|| nspname || 'to u1' FROM pg_namespace;
```

5.4 某张表执行过 grant select on table t1 to public，如何针对某用户回收权限

问题现象

假设当前有两个普通用户user1和user2，当前数据库testdb下有两张表t1和t2， 使用GRANT语句进行赋权：

```
GRANT SELECT ON table t1 TO public;
```

用户user1和user2对该表t1有访问权限，随后新建用户user3后，新用户user3对该表也有访问权限，且执行**REVOKE SELECT on table t1 FROM user3;**语句撤销user3查询t1表的权限不生效。

```
testdb=# REVOKE SELECT ON table t1 FROM user3;
REVOKE
testdb=# \c - user3
Password for user user3:
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "testdb" as user "user3".
testdb=> SELECT * FROM t1;
 a
---
(0 rows)

test=> SELECT relname, relacl FROM pg_class WHERE relname = 't1';
 relname |                  relacl
---------+-------------------------------------------
 t1      | {user3=arwdDxt/user3,=r/user3}
(1 row)
```

原因分析

上述问题中撤销user3对表t1的访问权限未生效是因为：之前执行过**GRANT SELECT ON table t1 TO public;**这条SQL语句，该语句中关键字public表示该权限要赋予给所有角色，包括之后新创建的角色，所以新用户user3对该表也有访问权限。public可以看做是一个隐含定义好的组，它包含所有角色。

因此，执行完**REVOKE SELECT ON table t1 FROM user3;**之后，虽然user3用户没有了表t1的访问权限（通过系统表pg_class的relacl字段可查看t1表的权限），但是他仍然有public的权限，所以仍能访问该表。

处理方法

撤销public的权限后对user3用户的权限单独管控。但是由于REVOKE回public的权限后可能导致原来能访问该表的用户（user1和user2）无法访问该表，影响正在使用的业务，因此需要先对这些用户执行GRANT赋予相应权限，然后REVOKE回public的权限。

步骤1 查看所有用户。

```
SELECT * FROM pg_user WHERE usesysid >= 16384;
 usename | usesysid | usecreatedb | usesuper | usecatupd | userepl |  passwd  | valbegin | valuntil |
respool    | parent | spacelimit | useconfig | nodegroup | tempspacelimit | spillspacelimit
---------+----------+-------------+----------+-----------+---------+----------+----------+----------+--------------
+--------+------------+-----------+-----------+----------------+-----------------
 jack    |    16408 | f          | f        | f         | f       | ******** |          |          | default_pool |      0 |
|          |           |              |
 tom     |    16412 | f          | f        | f         | f       | ******** |          |          | default_pool |      0 |
|          |           |              |
```

```
 user1   |   16437 | f          | f       | f       | f      | ******** |         |         | default_pool |     0 |
|         |         |            |
 user2   |   16441 | f          | f       | f       | f      | ******** |         |         | default_pool |     0 |
|         |         |            |
 user3   |   16448 | f          | f       | f       | f      | ******** |         |         | default_pool |     0 |
|         |         |            |
(5 rows)
```

步骤2 对原用户执行GRANT语句进行赋权。

```
GRANT select on table t1 TO jack,tom,user1,user2;
```

步骤3 撤销示例表t1的public权限。

```
REVOKE select on table t1 FROM public;
```

步骤4 切换至用户user3，再次查询示例表t1，结果显示user3访t1表的权限已成功撤销。

```
testdb=# \c - user3
Password for user user3:
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "testdb" as user "user3".
testdb=> SELECT * FROM t1;
ERROR:  permission denied for relation t1
```

----结束

5.5 普通用户创建或删除 GDS/OBS 外表语句时报错，提示没有权限或权限不足

问题现象

创建GDS或OBS外表语句时，管理员用户可以执行成功，但普通用户执行时报错“ERROR: permission denied to create foreign table in security mode.”。

```
CREATE USER u1 PASSWORD '{password}';
SET current_schema = u;
CREATE FOREIGN TABLE customer_ft
(
    c_customer_sk             integer               ,
    c_customer_id             char(16)              ,
    c_current_cdemo_sk        integer               ,
    c_current_hdemo_sk        integer               ,
    c_current_addr_sk         integer
)
    SERVER gsmpp_server
    OPTIONS
(
    location 'gsfs://192.168.0.90:5000/customer1*.dat',
    FORMAT 'TEXT' ,
    DELIMITER '|',
    encoding 'utf8',
    mode 'Normal')
READ ONLY;
ERROR:  permission denied to create foreign table in security mode
```

原因分析

该错误信息说明普通用户没有创建外表的权限。

查询该示例中用户权限：

```
SELECT rolname,roluseft FROM pg_roles WHERE rolname ='u1' ORDER BY rolname desc;
 rolname | roluseft
---------+----------
```

```
 u1      | f
(1 row)
```

处理方法

可使用ALTER USER或者ALTER ROLE语法指定USEFT参数赋予角色或用户使用外表的权限。

参数USEFT | NOUSEFT决定一个新角色或用户是否能操作外表，包括：新建外表、删除外表、修改外表、读写外表。

- 指定USEFT表示角色或用户可操作外表。
- 缺省为NOUSEFT。表示新角色或用户没有操作外表的权限。

请使用数据库管理员用户给普通用户或角色赋予使用外表的权限，示例如下：

```
ALTER USER user_name USEFT;
```

修改用户或角色权限等信息的详细内容请参见**ALTER USER**或者**ALTER ROLE**。

对普通用户或角色赋予使用外表的权限后即可创建外表。

5.6 赋予用户 schema 的 all 权限后建表仍然报错 ERROR: current user does not have privilege to role tom

问题现象

有两个用户tom和jerry，jerry需要在tom的同名schema下创建表，于是tom把该schema的all权限赋予jerry，但是创建表时仍然报错：

```
dbtest=# GRANT all on schema tom to jerry;
GRANT
dbtest=# \c - jerry
Password for user jerry:
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "dbtest" as user "jerry".
dbtest=>
dbtest=> CREATE TABLE tom.t(a int);
ERROR:  current user does not have privilege to role tom
```

原因分析

根据报错内容，jerry需要角色tom的权限。

处理方法

把角色tom的权限赋予jerry后，建表执行成功。

```
dbtest=# GRANT tom to jerry;
GRANT ROLE
dbtest=# \c - jerry
Password for user jerry:
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "dbtest" as user "jerry".
dbtest=>
dbtest=> CREATE TABLE tom.t(a int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
```

5.7 执行语句过程中报错：无权限操作

问题现象

执行语句后提示

```
ERROR:  permission denied for xxx
```

原因分析

该用户无对应的权限，无法对表或schema进行操作或访问。

处理方法

步骤1 使用GRANT语法对表/schema进行赋权，示例： 假设当前有两个用户tom和jerry，如果想要用户jerry能够对当前tom创建的所有表以及将来创建的表都有查询权限，如何处理：

- 将用户tom下的同名schema权限赋给jerry。
  ```
  GRANT USAGE ON SCHEMA tom TO jerry;
  ```
- 将用户tom已经创建的表的select权限赋给jerry。
  ```
  GRANT SELECT ON ALL TABLES IN SCHEMA tom TO jerry;
  ```
- 将用户tom未来在同名schema下创建的表的select权限赋给jerry。
  ```
  ALTER DEFAULT PRIVILEGES FOR USER tom IN SCHEMA tom GRANT SELECT ON TABLES TO jerry;
  ```

----结束

5.8 用户存在依赖关系无法删除如何处理

问题现象

数据库使用中遇到某些用户离职或者角色变更时，要对其账号进行销户、权限进行回收等操作，但是删除用户时出现类似：**role “u1” cannot be dropped because some objects depend on it**的报错信息，无法删除该用户。

例如，要删除用户u1时，出现如下提示：

```
testdb=# DROP USER u1;
ERROR:  role "u1" cannot be dropped because some objects depend on it
DETAIL:  owner of database testdb
3 objects in database gaussdb
```

原因分析

如果用户或角色的权限比较复杂、依赖较多，删除用户时就会报错存在依赖关系无法删除。通过上述问题现象中的ERROR的提示信息，获取以下信息：

- 要删除的用户为数据库testdb的owner。
- 有3个依赖的对象在gaussdb数据库中。

处理方法

- **要删除的用户为一个数据库的owner，需要将对象的所有权重新分配给其他用户。**有以下两种处理方法：

  方式一：将数据库owner转移给其他用户。例如，使用ALTER语句将数据库testdb的owner用户u1修改为u2。

```
testdb=# ALTER DATABASE testdb OWNER to u2;
ALTER DATABASE
testdb=# \l
                                  List of databases
   Name    |   Owner  | Encoding |   Collate   |    Ctype    |    Access privileges
-----------+----------+----------+-------------+-------------+-------------------------
 testdb    | u2       | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
(4 rows)
```

  执行删除u1用户的命令，不再提示“owner of database testdb”的信息。

```
testdb=# DROP USER u1;
ERROR:  role "u1" cannot be dropped because some objects depend on it
DETAIL:  3 objects in database gaussdb
```

  方式二：如果已确认不需要数据库testdb，也可直接将其删除。将所有u1拥有的数据库对象的属主更改为u2。

```
testdb=# REASSIGN OWNED BY u1 TO u2;
REASSIGN OWNED
```

  清理owner是u1的对象。请谨慎使用，会将用户同名的schema也一同清理掉。

```
testdb=# DROP OWNED by u1;
DROP OWNED
```

- **要删除的用户存在依赖关系，需要解除依赖关系**。处理方法如下：

  a. 识别依赖关系。根据报错信息“3 objects in database gaussdb”可知gaussdb数据库里有3个对象依赖u1。由于数据库内系统表的依赖，在其他数据库中不会打印出详细的依赖对象信息，那么在gaussdb库中执行DROP USER的时候，会打印出具体的信息。

     连接到gaussdb库执行如下命令：

```
gaussdb=# DROP USER u1;
ERROR:  role "u1" cannot be dropped because some objects depend on it
DETAIL:  privileges for table pg_class
privileges for schema u2
```

     获取到依赖项的详细信息如下：

     i. privileges for table pg_class：pg_class上u1用户的权限。

     ii. schema u2上u1用户的权限。

  b. 撤销依赖对象的权限。

```
gaussdb=# SELECT relname,relacl FROM pg_class WHERE relname = 'pg_class';
 relname  |              relacl
----------+-----------------------------------
 pg_class | {=r/Ruby,u1=r/Ruby}
(1 row)

gaussdb=#SELECT nspname,nspacl FROM pg_namespace WHERE nspname = 'u2';
 nspname |           nspacl
---------+----------------------------
 u2      | {u2=UC/u2,u2=LP/u2,u1=U/u2}

gaussdb=# REVOKE SELECT ON TABLE pg_class FROM u1;
REVOKE
gaussdb=# REVOKE USAGE ON SCHEMA u2 FROM u1;
REVOKE
```

  c. 再删除用户，可成功删除，不再提示有依赖。

```
gaussdb=# DROP USER u1;
DROP USER
```


- **有场景中存在不知道要删除用户的具体依赖对象，还是不能成功删除，要如何处理呢？** 此处以构造的案例进行演示，新建用户u3， 并赋予u2的SELECT权限。

```
testdb2=# DROP USER u3;
ERROR:  role "u3" cannot be dropped because some objects depend on it
DETAIL:  2 objects in database gaussdb
```

a. pg_shdepend系统表里记录了各个有依赖的对象的oid及其依赖关系。首先获取到用户的oid，再去系统表中找对应的依赖记录。

```
testdb2=# SELECT oid ,rolname FROM pg_roles WHERE rolname = 'u3';
    oid     | rolname
------------+---------
 2147484573 | u3
(1 row)

gaussdb=# SELECT * FROM pg_shdepend WHERE refobjid = 2147484573;
 dbid  | classid |   objid    | objsubid | refclassid |  refobjid  | deptype | objfile
-------+---------+------------+----------+------------+------------+---------+---------
 16073 |    2615 | 2147484575 |        0 |       1260 | 2147484573 | o       |
 16073 |    2615 | 2147484025 |        0 |       1260 | 2147484573 | a       |
(2 rows)
```

由于dependType不同，因此有两条记录，一条记录代表权限依赖(a)，另一条记录代表自身是一个对象的owner。

b. 获取到的classid，代表依赖当前用户的对象的记录表的oid，在pg_class表中找到对应依赖即可。

```
gaussdb=# SELECT relname,relacl FROM pg_class WHERE oid = 2615;
   relname    |     relacl
--------------+----------------
 pg_namespace | {=r/role23}
(1 row)
```

c. 通过查询记录表是pg_namespace，那么就可以确认依赖用户的是一个schema。再到pg_namespace系统表中，查询**1**获取到的objid，可确认具体对象。

```
gaussdb=# SELECT nspname,nspacl FROM pg_namespace WHERE oid in
(2147484575,2147484025);
 nspname |                 nspacl
---------+----------------------------------------------------------
 u3      |
 u2      | {u2=UC/u2,u2=LP/u2,u3=U/u2}
(2 rows)
```

这里看到有两个schema，u3是用户同名的schema，u2是赋权的schema，撤销赋权schema权限。

```
gaussdb=# REVOKE USAGE ON SCHEMA u2 FROM u3;
REVOKE
```

d. 再删除用户u3，可成功删除。

```
gaussdb=# DROP USER u3;
DROP USER
```


6 集群性能

6.1 锁等待检测

操作场景

在日常作业开发中，数据库事务管理中的锁一般指的是**表级锁**，GaussDB(DWS)中支持的锁模式有8种，按排他级别分别为1~8。每种锁模式都有与之相冲突的锁模式，由锁冲突表定义相关的信息，锁冲突表如表6-1所示。

举例：用户u1对某张表test执行INSERT事务时，此时持有RowExclusiveLock锁；此时用户u2也对test表进行VACUUM FULL事务，则该事务与INSERT事务产生锁冲突，处于锁等待状态。

常用的锁等待检测主要通过查询视图**pgxc_lock_conflicts**、**pgxc_stat_activity**、**pgxc_thread_wait_status**、**pg_locks**进行。其中**pgxc_lock_conflicts**视图在8.1.x版本后支持，根据集群版本号不同，检测方式不同。

表 6-1 锁冲突

编号	名称	用途	冲突关系
1	AccessShareLock	SELECT	8
2	RowShareLock	SELECT FOR UPDATE/FOR SHARE	7 | 8
3	RowExclusiveLock	INSERT/UPDATE/DELETE	5 | 6 | 7 | 8
4	ShareUpdateExclusiveLock	VACUUM	4 | 5 | 6 | 7 | 8
5	ShareLock	CREATE INDEX	3 | 4 | 6 | 7 | 8
6	ShareRowExclusiveLock	ROW SELECT...FOR UPDATE	3 | 4 | 5 | 6 | 7 | 8

编号	名称	用途	冲突关系
7	ExclusiveLock	BLOCK ROW SHARE/ SELECT...FOR UPDATE	2 | 3 | 4 | 5 | 6 | 7 | 8
8	AccessExclusiveLock	DROP CLASS/ VACUUM FULL	1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

操作步骤

构造锁等待场景：

步骤1 打开一个新的连接会话，使用普通用户u1连接GaussDB(DWS)数据库，在自己的同名SCHEMA u1下创建测试表u1.test。

```
CREATE TABLE test (id int, name varchar(50));
```

步骤2 开启事务1，进行INSERT操作。

```
START TRANSACTION;
INSERT INTO test VALUES (1, 'lily');
```

步骤3 打开一个新的连接会话，使用系统管理员dbadmin连接GaussDB(DWS)数据库，执行VACUUM FULL操作，发现语句阻塞。

```
VACUUM FULL u1.test;
```

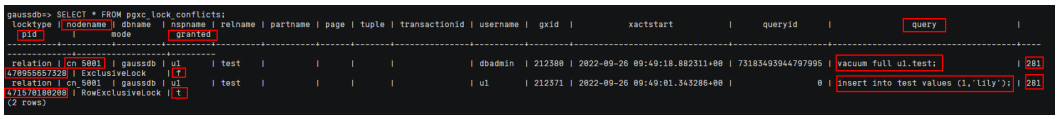
----结束

锁等待检测（8.1.x及以上版本）

步骤1 打开一个新的连接会话，使用系统管理员dbadmin连接GaussDB(DWS)数据库，通过pgxc_lock_conflicts视图查看锁冲突情况。

如下图，回显中查看granted字段为“f”，表示VACUUM FULL语句正在等待其他锁。granted字段为“t”，表示INSERT语句是持有锁。nodename，表示锁产生在的位置，即CN或DN位置，例如cn_5001。

```
SELECT * FROM pgxc_lock_conflicts;
```



步骤2 据语句内容确认是否中止持锁语句。如果终止，则执行以下语句。*pid*从**步骤1**获取，cn_5001为上面查询到的nodename。

```
execute direct on (cn_5001) 'SELECT PG_TERMINATE_BACKEND(pid)';
```

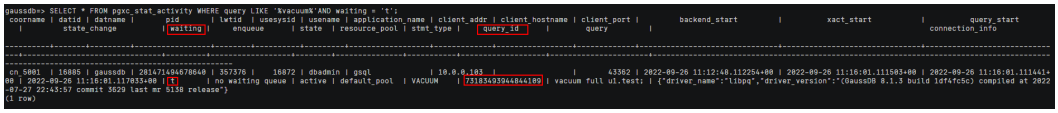
```
gaussdb=> execute direct on (cn_5001) 'SELECT PG_TERMINATE_BACKEND( 281471570180208 )';
 pg_terminate_backend
----------------------
 t
(1 row)
```

----结束

锁等待检测（8.0.x及以前版本）

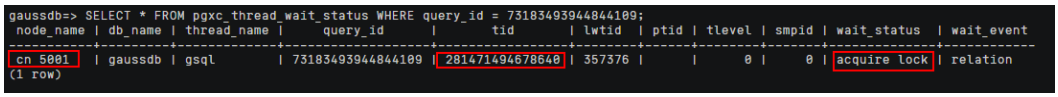
步骤1 在数据库中执行以下语句，获取VACUUM FULL操作对应的query_id。

```
SELECT * FROM pgxc_stat_activity WHERE query LIKE '%vacuum%'AND waiting = 't';
```



步骤2 根据获取的query_id，执行以下语句查看是否存在锁等待，并获取对应的tid。其中，*{query_id}*从**步骤1**获取。

```
SELECT * FROM pgxc_thread_wait_status WHERE query_id = {query_id};
```

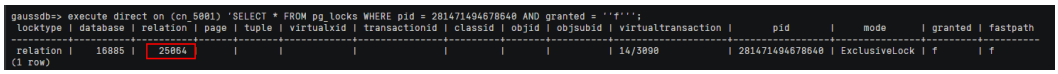


回显中“wait_status”存在“acquire lock”表示存在锁等待。同时查看“node_name”显示在对应的CN或DN上存在锁等待，记录相应的CN或DN名称，例如cn_5001或dn_600x_600y。

步骤3 执行以下语句，到等锁的对应CN或DN上通过查询pg_locks系统表查看VACUUM FULL操作在等待哪个锁。以下以cn_5001为例，如果在DN上等锁，则改为相应的DN名称。pid为**步骤2**获取的tid。

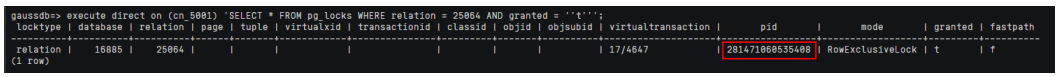
回显中记录relation的值。

```
execute direct on (cn_5001) 'SELECT * FROM pg_locks WHERE pid = {tid} AND granted = ''f''';
```



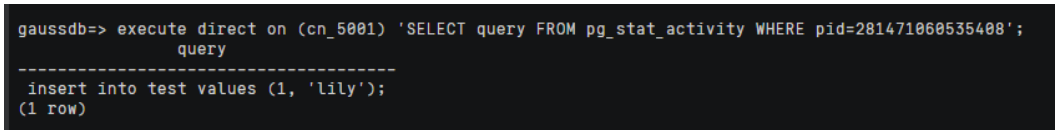
步骤4 根据获取的relation，通过查询pg_locks系统表查看当前持有锁的pid。*{relation}*从**步骤3**获取。

```
execute direct on (cn_5001) 'SELECT * FROM pg_locks WHERE relation = {relation} AND granted = ''t''';
```



步骤5 根据pid，执行以下语句，查到对应的SQL语句。*{pid}*从**步骤4**获取。

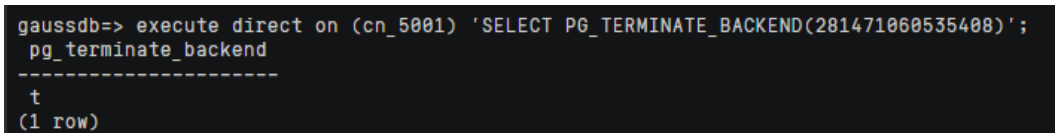
```
execute direct on (cn_5001) 'SELECT query FROM pg_stat_activity WHERE pid={pid}';
```



步骤6 根据语句内容确认是中止持锁语句还是待持锁语句结束再重新执行VACUUM FULL。如果终止，则执行以下语句。*pid*从**步骤4**获取。

中止结束后，再尝试重新执行VACUUM FULL。

```
execute direct on (cn_5001) 'SELECT PG_TERMINATE_BACKEND(pid)';
```



----结束

6.2 执行 SQL 时出现表死锁，提示 LOCK_WAIT_TIMEOUT 锁等待超时

问题现象

执行SQL时出现LOCK_WAIT_TIMEOUT锁等待超时的错误。

原因分析

锁等待超时一般是因为有其他的SQL语句已经持有了锁，当前SQL语句需要等待持有锁的SQL语句执行完毕释放锁之后才能执行。当申请的锁等待时间超过GUC参数lockwait_timeout的设定值时，系统会报LOCK_WAIT_TIMEOUT的错误。

处理方法

1. 8.1.x及以上集群版本，通过pgxc_lock_conflicts视图查看锁冲突情况。
   ```
   SELECT * FROM pgxc_lock_conflicts;
   ```
   8.0.x及之前集群版本，执行以下SQL查询查看是否有阻塞的SQL语句。
   ```
   SELECT w.query as waiting_query,
   w.pid as w_pid,
   w.usename as w_user,
   l.query as locking_query,
   l.pid as l_pid,
   l.usename as l_user,
   n.nspname || '.' || c.relname as tablename
   from pg_stat_activity w join pg_locks l1 on w.pid = l1.pid
   and not l1.granted join pg_locks l2 on l1.relation = l2.relation
   and l2.granted join pg_stat_activity l on l2.pid = l.pid join pg_class c on c.oid = l1.relation join
   pg_namespace n on n.oid=c.relnamespace
   where w.waiting;
   ```
2. 查询到阻塞的表及模式信息后，请根据实际查询语句ID结束会话：
   ```
   SELECT pgxc_terminate_query(7233906901463861);
   ```
   或
   ```
   EXECUTE DIRECT ON(cn_5002) 'SELECT pg_terminate_backend(139666091022080)';
   ```
3. 这种情况一般是因为业务调度不太合理，建议合理安排各个业务的调度时间。
4. 还可以通过设置GUC参数lockwait_timeout，控制单个锁的最长等待时间，即单个锁的等待超时时间。

   lockwait_timeout单位为毫秒（ms），默认值为20分钟。

   lockwait_timeout参数属于SUSET类型参数，请参考**设置GUC参数**中对应的设置方法进行设置。

6.3 执行 SQL 时报错：abort transaction due to concurrent update

问题现象

执行SQL时出现abort transaction due to concurrent update锁等待超时的错误。

原因分析

两个不同的事务对同一个表中的同一行数据进行并发更新/操作，导致后操作的事务发生了回滚。

举例说明：

步骤1 打开一个连接Session A，使用普通用户u1连接GaussDB(DWS)数据库，在u1的同名SCHEMA u1下创建测试表u1.test，并插入数据。

```
CREATE TABLE test (id int, name varchar(50));
INSERT INTO test VALUES (1, 'lily');
```

步骤2 打开一个新的连接会话Session B，使用系统管理员dbadmin连接GaussDB(DWS)数据库，开启事务1，执行UPDATE操作，UPDATE语句执行成功。

```
START TRANSACTION;

UPDATE u1.test SET id = 3 WHERE name = 'lily';
UPDATE 1
```

步骤3 在会话Session A中开启事务2，执行相同的UPDATE语句，执行报错。

```
START TRANSACTION;

UPDATE test SET id = 3 WHERE name = 'lily';
ERROR:  dn_6003_6004: abort transaction due to concurrent update test 289502.
```

----结束

针对上述案例，两个不同的事务并发更新了同一条记录，而并发更新同一条记录发生冲突不会等待锁，直接报错：abort transaction due to concurrent update。

在实际业务中，并不是只有并发UPDATE同一条记录会报错，select、delete等其他SQL并发操作，也有可能报错：abort transaction due to concurrent update。

处理方法

- 调整业务逻辑sql执行顺序

  避免update/delete长时间持有锁的sql在事务前面。
- 避免大事务

  尽量将大事务拆分成多个小事务来处理，小事务缩短锁定资源的时间，发生冲突的几率也降低。
- 控制并发度

  尽可能减少并发会话的数量，以减少冲突的几率。

6.4 磁盘使用率高&集群只读处理方案

查看磁盘使用率

对于用户来说，DWS的磁盘使用率是一个非常需要关注的高价值资源，并且这个资源与集群的可用性息息相关。因此，当出现以下场景时，需要用户密切关注磁盘空间并做出相应的磁盘空间处置（以下磁盘均指数据磁盘）。

查看磁盘空间方法：

步骤1 登录DWS管理控制台，左侧单击“集群管理”，在集群列表中单击指定集群所在行右侧的“监控面板”进入到DMS监控界面。

步骤2 选择“监控 > 节点监控 > 磁盘”，单击“磁盘使用率”右侧的进行排序，可查看当前集群各个节点的磁盘使用率。

数据磁盘识别方法：一般根据容量进行查看，容量为购买容量的磁盘是数据盘

----结束

故障场景

- **场景一：磁盘使用率过高**，当前集群所有磁盘或超过半数以上的磁盘使用率>=70%。
- **场景二：磁盘倾斜**，使用率最高的磁盘和最低的磁盘使用率之差>=10%。
- **场景三：集群只读**，当前只读阈值为单数据盘使用率>=90%。

日常处理作业中，DBA可以通过**空间管控**识别出一些异常业务，阻拦这些不合理的作业，以避免以上场景出现。

须知

1. 只读是DWS系统自我保护的一种机制，避免出现因为磁盘使用率达到100%，导致DWS实例无法正常启动，引发业务彻底中断的风险。
2. 以上场景均会由DMS进行告警通告（告警阈值为80%，可自行配置，参见**修改告警规则**）。
3. 以上场景一，可通过使用告警订阅功能，当磁盘超过70%或75%时给用户发送短信或邮件进行通知，以便提前进行数据清理，如何设置参见**磁盘空间告警订阅**。

场景一：磁盘使用率高清理方式

根据数据表的查询结果，定期进行**脏数据**清理，不同版本方式不同，根据对应版本进行选择。

- **8.1.3及以上版本：通过管理控制台“智能运维”功能进行自动清理。**

步骤1 登录GaussDB(DWS) 管理控制台。

步骤2 在集群列表中单击指定集群名称。

步骤3 进入“集群详情”页面，切换至“智能运维”页签。

步骤4 在运维详情部分切换至运维计划模块。单击“添加运维任务”按钮。

步骤5 弹出添加运维任务边栏，

- 运维任务选择“Vacuum”。
- 调度模式选择“自动”，DWS将自动扫描Vacuum目标。
- Vacuum目标选择系统表或用户表：
  - 如果用户业务UPDATE、DELETE较多，选择用户表。
  - 如果创建表、删除表较多，选择系统表。

步骤6 单击“下一步：定时配置”，配置Vacuum类型，推荐选择“周期型任务”，GaussDB(DWS)将自动在自定义时间窗内执行Vacuum。

说明

对于自动Vacuum运维任务，系统对于用户表的处理方法实际采用的是VACUUM FULL操作。VACUUM FULL执行过程中，本身持有8级锁，会阻塞其他业务，导致锁冲突产生，业务本身会陷入锁等待，20分钟后超时报错。因此，在用户配置时间窗内，应尽量避开执行所有处理表的相关业务。

步骤7 确认无误后，单击“下一步：配置确认”，完成配置。

----结束

- **8.1.2及以前版本：手动执行VACUUM FULL进行清理。**

须知

1. VACUUM FULL操作**会锁表**，VACUUM FULL期间，该表的所有访问会阻塞，并等待VACUUM FULL结束，请合理安排调度时间，避免锁表影响业务。
2. VACUUM FULL是对当前表的有效数据抽出来重新整理，同时清理脏数据，该操作会临时占用**额外的整理空间**（这部分空间待整理完成后释放），因此空间会先增后降，请提前计算好VACUUM FULL所需要的空间再行处理（**额外的整理空间大小=表大小* (1 – 脏页率)**）。

步骤1 连接数据库，执行以下SQL语句查询脏页率超过30%的较大表，并且按照表大小从大到小排序。

```
SELECT schemaname AS schema, relname AS table_name, n_live_tup AS analyze_count,
pg_size_pretty(pg_table_size(relid)) as table_size, dirty_page_rate
FROM PGXC_GET_STAT_ALL_TABLES
WHERE schemaName NOT IN ('pg_toast', 'pg_catalog', 'information_schema', 'cstore', 'pmk')
AND dirty_page_rate > 30
ORDER BY table_size DESC, dirty_page_rate DESC;
```

步骤2 判断是否有回显结果。

- 是，对于表大小超过10G的表，则执行**步骤3**。
- 否，操作结束。

步骤3 将脏页Top5的表，进行VACUUM FULL清理（清理时，如果最高磁盘空间>70%，请串行清理）。

```
VACUUM FULL ANALYZE schema.table_name;
```

步骤4 如果无脏页率较高的表，并且磁盘使用率已经接近或者超过75%，根据以下数仓类型，对集群进行节点扩容或磁盘扩容，避免触发只读导致业务中断。

- 云数仓+SSD云盘：参见**磁盘扩容**进行磁盘扩容。
- 云数仓+SSD本地盘及旧的标准数仓(不支持磁盘扩容)：请联系技术支持进行**在线扩容**。

----结束

场景二：磁盘倾斜，倾斜表清理方式

倾斜表，针对单DN倾斜率>5%的表，建议对表进行重选**分布列**，并对数据进行**重分布。**

须知

- 倾斜表的危害：倾斜表可能引发算子计算/下盘倾斜严重，导致数据倾斜的DN处理压力过大，而无法发挥DWS的分布式计算的优势，影响业务性能，并且容易造成单DN磁盘满。
- 8.1.3版本开始默认建表为轮询表（参见**轮询表RoundRobin**），如果不熟悉分布键，可以在**建表**时，使用ROUNDROBIN关键字，来降低业务开发难度（参见**轮询表、哈希表适用场景**）。

步骤1 连接数据库，执行以下SQL语句查询倾斜表。

```
WITH skew AS
(
        SELECT
                schemaname,
                tablename,
                pg_catalog.sum(dnsize) AS totalsize,
                pg_catalog.avg(dnsize) AS avgsize,
                pg_catalog.max(dnsize) AS maxsize,
                pg_catalog.min(dnsize) AS minsize,
                (pg_catalog.max(dnsize) - pg_catalog.min(dnsize)) AS skewsize,
                pg_catalog.stddev(dnsize) AS skewstddev
        FROM pg_catalog.pg_class c
        INNER JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
        INNER JOIN pg_catalog.gs_table_distribution() s ON s.schemaname = n.nspname AND s.tablename =
c.relname
        INNER JOIN pg_catalog.pgxc_class x ON c.oid = x.pcrelid AND x.pclocatortype IN('H', 'N')
        GROUP BY schemaname,tablename
)
SELECT
        schemaname,
        tablename,
        totalsize,
        avgsize::numeric(1000),
        (maxsize/totalsize)::numeric(4,3)  AS maxratio,
        (minsize/totalsize)::numeric(4,3)  AS minratio,
        skewsize,
```

```
    (skewsize/avgsize)::numeric(4,3)  AS skewratio,
    skewstddev::numeric(1000)
FROM skew
WHERE totalsize > 0;
```

有关查询倾斜表的详细内容，可参考**数据倾斜导致SQL执行慢，大表SQL执行无结果**章节。

步骤2 根据表的大小和倾斜率，将倾斜严重的表重新选择分布列，8.1.0及以上版本直接通过ALTER TABLE语法进行调整，其他版本参见**如何调整分布列**。

----结束

场景三：集群只读

当集群单磁盘使用率超过90%时，系统会自动触发集群只读，此时继续执行写类型业务（DML或DDL）会出现类似报错“ERROR: cannot execute xxx in a read-only transaction.”。

集群只读是DWS对用户数据的一种自我保护机制，防止磁盘写满引发实例无法启动。

集群只读后，由于剩余空间较小，需要尽快通过DROP、TRUNCATE等操作清理无用数据，将空间降至80%以下。之后再处理场景一和场景二中的相关表，避免VACUUM FULL和倾斜处理引发磁盘使用率先增后降，导致磁盘写满。

- **8.1.3及以上集群版本处理方法：**

步骤1 集群处于“只读”状态时，应立即停止写入任务，避免磁盘被写满造成数据丢失的风险。

步骤2 使用客户端连接数据库，通过显示事务关闭只读，DROP/TRUNCATE TABLE清理无用数据，尽量将磁盘使用率清理至80%以下。

清理数据方式1：

```
START TRANSACTION READ WRITE;
drop/truncate table table_name;
COMMIT;
```

清理数据方式2：

```
START TRANSACTION;
SET transaction_read_only=off;
drop/truncate table table_name;
COMMIT;
```

清理完成后，系统会自动解除只读。

步骤3 排查场景一和场景二相关表，查看是否有需要整改的表，如果没有，建议尽快对集群进行扩容，根据数仓类型不同，分为节点扩容和磁盘扩容。

- 云数仓+SSD云盘：参见**磁盘扩容**进行磁盘扩容。
- 云数仓+SSD本地盘及旧的标准数仓(不支持磁盘扩容)：请联系技术支持进行**在线扩容**。

----结束

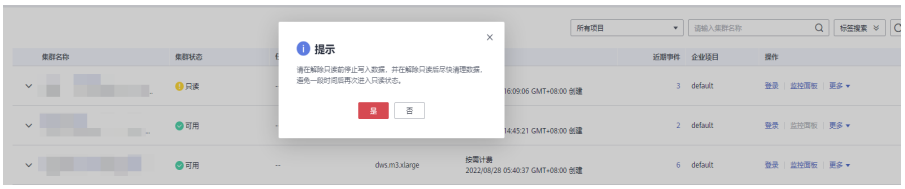
- **8.1.2及以前集群版本处理方法：**

步骤1 集群处于“只读”状态时，应立即停止写入任务，避免磁盘被写满造成数据丢失的风险。

步骤2 集群进入只读状态时，通过登录GaussDB(DWS) 管理控制台，解除只读状态。

1. 登录GaussDB(DWS) 管理控制台。单击“集群管理”。默认显示用户所有的集群列表。
2. 在集群列表中，在指定集群所在行的“操作”列，选择“更多 > 解除只读”。

3. 在弹出对话框中，单击“确定”，再次进行解除只读确认，对集群进行解除只读操作。



步骤3 解除只读操作成功后，通过DROP/TRUNCATE清理相关的无用数据。尽量将磁盘使用率清理至80%以下。

步骤4 排查场景一和场景二相关表，查看是否有需要整改的表，如果没有，建议尽快对集群进行扩容，根据数仓类型不同，分为节点扩容和磁盘扩容。

- 云数仓+SSD云盘：参见**磁盘扩容**进行磁盘扩容。
- 云数仓+SSD本地盘及旧的标准数仓(不支持磁盘扩容)：请联系技术支持进行**在线扩容**。

----结束

单语句空间管控

GaussDB(DWS)支持语句磁盘空间管控相关的参数sql_use_spacelimit和temp_file_limit，用于业务运行时，避免由于不合理的数据量，引发磁盘空间暴增，触发告警/只读，主动识别不合理的大批量倒数业务或者高数据量入库业务。

步骤1 登录DWS控制台，左侧单击“集群管理”，在集群列表中单击指定集群，进入集群详情页面。

步骤2 单击“参数修改”，搜索栏中搜索sql_use_spacelimit和temp_file_limit（参见**磁盘空间**），根据业务进行调整。

建议设置sql_use_spacelimit为实例所在磁盘空间总容量的10%（例如购买时空间为100G/每节点，则配置该参数为10G）。

须知

上述配置生效后，如果单语句运行过程中超过该配置参数，则SQL语句会被主动中止。如需临时放开，可以在session会话中执行以下语句进行临时关闭。

```
SET sql_use_spacelimit=0;
```

----结束

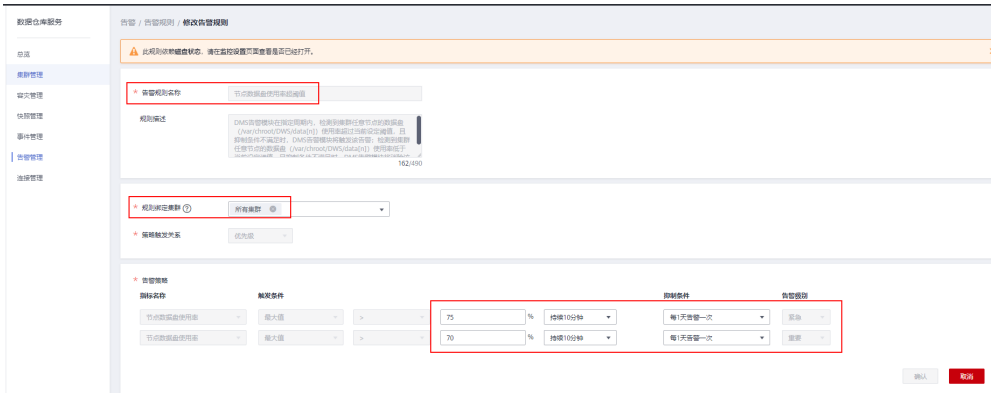
磁盘空间告警订阅

为了减轻客户自行的运维压力，DWS提供了告警订阅的功能：即当集群的磁盘使用率大于设置的阈值时，系统会以短信、邮件形式通知到用户。

步骤1 设置告警阈值：

1. 登录DWS控制台，左侧选择“告警管理”，单击“告警规则管理”。

2. 在告警规则列表中，单击“**节点数据盘使用率超阈值**”右侧的“修改”。
3. 在集群列表中单击指定集群，进入集群详情页面。告警策略设置如下：
   - 规则绑定集群：所有集群
   - 告警策略：将数据盘的重要告警配置为70%，持续10分钟，紧急告警配置为75%，持续10分钟。如下设置：



步骤2 在消息通知服务(SMN)创建主题。

1. 切换到消息通知服务控制台，单击“创建主题”。如下设置“主题名称”和“企业项目”。

2. 主题创建成功后，单击右侧的“添加订阅”。根据需要选择“短信”、“邮件”方式，订阅终端输入对应的手机号或邮箱地址。

3. 输入的手机号或邮箱地址会收到确认短信或邮件，单击确认，即可完成订阅。

步骤3 添加告警订阅。

1. 回到DWS控制台，左侧选择“告警管理”，切换到“订阅”，单击“创建订阅”。
2. 订阅名称输入“dms_alarm”，告警级别选择“紧急”、“重要”，消息通知主题名称选择上一步创建的主题“dms_alarm”。

3. 单击“确认”。整个告警订阅成功，后续磁盘使用率大于70%和75%则发出通知。

----结束

6.5 SQL 执行很慢，性能低，有时长时间运行未结束

问题现象

SQL执行很慢，性能低，有时长时间运行未结束。

原因分析

SQL运行慢可从以下几方面进行分析：

1. 使用EXPLAIN命令查看SQL执行计划，根据执行计划判断是否需要进行SQL调优。
2. 分析查询是否被阻塞，导致语句运行时间过长，可以强制结束有问题的会话。
3. 审视和修改表定义。选择合适的分布列，避免数据倾斜。
4. 分析SQL语句是否使用了不下推的函数，建议更换为支持下推的语法或函数。
5. 对表定期执行VACUUM FULL和ANALYZE操作，可回收已更新或已删除的数据所占据的磁盘空间。

6. 检查表有无索引支撑，建议例行重建索引。

   数据库经过多次删除操作后，索引页面上的索引键将被删除，造成索引膨胀。例行重建索引，可有效的提高查询效率。

7. 对业务进行优化，分析能否将大表进行分表设计。

处理方法

GaussDB(DWS)提供了分析查询和改进查询的方法，并且为用户提供了一些常见案例以及错误处理办法。您可以参考**性能调优**章节对SQL进行性能调优。常见问题也可以优先参考以下两种方法进行分析：

- 方法一：对表定期做统计优化查询。

  如果频繁对表执行INSERT语句插入数据，需要定期对表执行ANALYZE：

  ```
  ANALYZE table_name;
  ```

  如果频繁对表执行DELETE语句删除数据，需要定期对表执行VACUUM FULL：

  ```
  VACUUM FULL table_name;
  ```

  说明

  执行VACUUM FULL语句时需选择空闲时间窗或停止业务时操作。

  查询表大小，如果表非常大，而实际只有很少数据，那么应该执行VACUUM FULL对表进行磁盘碎片整理。

  ```
  SELECT * FROM pg_size_pretty(pg_table_size('tablename'));
  VACUUM FULL table_name;
  ```

方法二：通过PGXC_STAT_ACTIVITY视图查询正在运行的SQL语句信息。

步骤1 查看当前正在运行（非idle）的SQL信息：

```
SELECT pid,datname,usename,state,waiting,query FROM pgxc_stat_activity WHERE state <> 'idle';
```

步骤2 查看当前处于阻塞状态的查询语句：

```
SELECT pid,datname, usename, state,waiting,query FROM pgxc_stat_activity WHERE state <> 'idle' and waiting=true;
```

步骤3 判断查询语句是否阻塞。

- 如果没有阻塞，查找相关业务表，按照**方法一**中的建议方法进行处理。
- 如果存在语句阻塞，根据所查找的问题会话的线程ID，结束阻塞的执行语句。

  ```
  SELECT pg_terminate_backend(pid);
  ```

----结束

6.6 数据倾斜导致 SQL 执行慢，大表 SQL 执行无结果

问题现象

某场景下SQL执行慢，涉及大表的SQL执行不出来结果。

原因分析

GaussDB(DWS)支持Hash、REPLICATION和ROUNDROBIN（8.1.2集群及以上版本支持ROUNDROBIN）分布方式。如果创建了Hash分布的表，未指定分布键，则选择表的第一列作为分布键，这种情况就可能存在倾斜。倾斜造成以下负面影响：

- SQL的性能会非常差，因为数据只分布在部分DN，那么SQL运行的时候就只有部分DN参与计算，没有发挥分布式的优势。
- 会导致资源倾斜，尤其是磁盘。可能部分磁盘的空间已经接近极限，但是其他磁盘利用率很低。
- 可能出现部分节点CPU过高等问题。

分析过程

步骤1 登录GaussDB(DWS)管理控制台。在“集群管理”页面，找到需要查看监控的集群。在指定集群所在行的“操作”列，单击“监控面板”。选择“监控 > 节点监控 > 磁盘”，查看磁盘使用率。

说明

各个数据磁盘的利用率，会有不均衡的现象。正常情况下，利用率最高和利用率最低的磁盘空间相差不大，如果磁盘利用率相差超过了5%就要注意是不是有资源倾斜的情况。

步骤2 连接数据库，通过等待视图查看作业的运行情况，发现作业总是等待部分DN或者个别DN。

```
SELECT wait_status, count(*) as cnt FROM pgxc_thread_wait_status WHERE wait_status not like '%cmd%' AND wait_status not like '%none%' and wait_status not like '%quit%' group by 1 order by 2 desc;
```

步骤3 执行慢语句的explain performance显示，发现各个DN的基表scan的时间和行数不均衡。

```
explain performance select avg(ss_wholesale_cost) from store_sales;
```

```
rmance select avg(ss_wholesale_cost) from store_sales;

      operation                                  |      A-time       |  A-rows  | E-rows  | E-distinct |   Peak Memory   | E-memory | A-wi
-------------------------------------------------+-------------------+----------+---------+------------+-----------------+----------+-----
                                                 | 2452.321          |        1 |       1 |            | 11KB            |          |
te                                               | 2452.238          |        1 |       1 |            | 184KB           |          |
aming (type: GATHER)                             | 2452.010          |        4 |       4 |            | 108KB           |          |
ggregate                                         | [12.219,2425.225] |        4 |       4 |            | [183KB, 184KB] | 1MB      |
r Partition Iterator                             | [12.139,1189.187] | 22831616 | 2880404 |            | [17KB, 17KB]    | 1MB      |
rtitioned CStore Scan on public.store_sales      | [5.929,1173.555]  | 22831616 | 2880404 |            | [482KB, 499KB] | 1MB      |


identified by plan id)
--------------------------
rator

can on public.store_sales
s:  1..7
```

- 基表scan的时间：最快的DN耗时5ms，最慢的DN耗时1173ms。
- 数据分布情况：某些DN有22831616行，其他DN都是0行，数据有严重倾斜。

```
5 --Vector Partition Iterator
      datanode1 (actual time=0.620..1189.187 rows=22831616 loops=1)
      datanode2 (actual time=14.346..14.346 rows=0 loops=1)
      datanode3 (actual time=14.424..14.424 rows=0 loops=1)
      datanode4 (actual time=12.139..12.139 rows=0 loops=1)
      datanode1 (CPU: ex c/r=1, ex row=22831616, ex cyc=40540820, inc cyc=3088825848)
      datanode2 (CPU: ex c/r=0, ex row=0, ex cyc=20852952, inc cyc=37297912)
      datanode3 (CPU: ex c/r=0, ex row=0, ex cyc=20005612, inc cyc=37501436)
      datanode4 (CPU: ex c/r=0, ex row=0, ex cyc=16147884, inc cyc=31560524)
6 --Partitioned CStore Scan on public.store_sales
      datanode1 (actual time=2.611..1173.555 rows=22831616 loops=7)
      datanode2 (actual time=6.327..6.327 rows=0 loops=7)
      datanode3 (actual time=6.732..6.732 rows=0 loops=7)
      datanode4 (actual time=5.929..5.929 rows=0 loops=7)
      datanode1 (Buffers: shared hit=1359)
      datanode2 (Buffers: shared hit=55)
      datanode3 (Buffers: shared hit=55)
      datanode4 (Buffers: shared hit=55)
      datanode1 (CPU: ex c/r=133, ex row=22831616, ex cyc=3048285028, inc cyc=3048285028)
      datanode2 (CPU: ex c/r=0, ex row=0, ex cyc=16444960, inc cyc=16444960)
      datanode3 (CPU: ex c/r=0, ex row=0, ex cyc=17495824, inc cyc=17495824)
      datanode4 (CPU: ex c/r=0, ex row=0, ex cyc=15412640, inc cyc=15412640)
```

步骤4 通过倾斜检查接口可以发现数据倾斜。

```
SELECT table_skewness('store_sales');
```



```
tpcds1xcpm=# select table_skewness('store_sales');
                table_skewness
-----------------------------------------------
 ("datanode1            ",22831616,100.000%)
 ("datanode2            ",0,0.000%)
 ("datanode3            ",0,0.000%)
 ("datanode4            ",0,0.000%)
(4 rows)
```

```
SELECT table_distribution('public','store_sales');
```

```
tpcds1xcpm=# select table_distribution('public','store_sales');
           table_distribution
-----------------------------------------
 (public,store_sales,datanode1,1011752960)
 (public,store_sales,datanode4,33792000)
 (public,store_sales,datanode2,32129024)
 (public,store_sales,datanode3,33349632)
(4 rows)
```

步骤5 通过资源监控发现，个别节点的CPU和I/O明显比其他节点高。

----结束

处理方法

如何找到倾斜的表：

1. 数据库中表个数少于1W的场景下，可直接使用倾斜视图查询当前库内所有表的数据倾斜情况。
   ```
   SELECT * FROM pgxc_get_table_skewness ORDER BY totalsize DESC;
   ```
2. 数据库中表个数非常多（至少大于1W）的场景，因PGXC_GET_TABLE_SKEWNESS视图涉及全库查并计算非常全面的倾斜字段，所以可能会花费比较长的时间（小时级），建议参考PGXC_GET_TABLE_SKEWNESS视图定义，执行以下操作：
   - 8.1.2及之前集群版本中使用table_distribution()函数自定义输出，减少输出列进行计算优化，例如：
     ```
     SELECT schemaname,tablename,max(dnsize) AS maxsize, min(dnsize) AS minsize
     FROM pg_catalog.pg_class c
     INNER JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
     INNER JOIN pg_catalog.table_distribution() s ON s.schemaname = n.nspname AND s.tablename
     = c.relname
     INNER JOIN pg_catalog.pgxc_class x ON c.oid = x.pcrelid AND x.pclocatortype = 'H'
     GROUP BY schemaname,tablename;
     ```
   - 8.1.3及以上集群版本中支持使用gs_table_distribution()函数，全库查询所有表的数据倾斜情况。全库表查询时，gs_table_distribution()函数优于table_distribution()函数；在大集群大数据量场景下，如果进行全库表查询，建议优先使用gs_table_distribution()函数。
     ```
     SELECT schemaname,tablename,max(dnsize) AS maxsize, min(dnsize) AS minsize
     FROM pg_catalog.pg_class c
     INNER JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
     INNER JOIN pg_catalog.gs_table_distribution() s ON s.schemaname = n.nspname AND
     s.tablename = c.relname
     INNER JOIN pg_catalog.pgxc_class x ON c.oid = x.pcrelid AND x.pclocatortype = 'H'
     GROUP BY schemaname,tablename;
     ```


说明

使用如下语句可快速查询到大表：

```
SELECT schemaname||'.'||tablename as table, sum(dnsize) as size FROM
gs_table_distribution() group by 1 order by 2 desc limit 10;
```

使用如下语句可快速查询表的倾斜率：

```
WITH skew AS
(
        SELECT
                schemaname,
                tablename,
                pg_catalog.sum(dnsize) AS totalsize,
                pg_catalog.avg(dnsize) AS avgsize,
                pg_catalog.max(dnsize) AS maxsize,
                pg_catalog.min(dnsize) AS minsize,
                (pg_catalog.max(dnsize) - pg_catalog.min(dnsize)) AS skewsize,
                pg_catalog.stddev(dnsize) AS skewstddev
        FROM pg_catalog.pg_class c
        INNER JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
        INNER JOIN pg_catalog.gs_table_distribution() s ON s.schemaname = n.nspname
AND s.tablename = c.relname
        INNER JOIN pg_catalog.pgxc_class x ON c.oid = x.pcrelid AND x.pclocatortype IN('H',
'N')
        GROUP BY schemaname,tablename
)
SELECT
        schemaname,
        tablename,
        totalsize,
        avgsize::numeric(1000),
        (maxsize/totalsize)::numeric(4,3)  AS maxratio,
        (minsize/totalsize)::numeric(4,3)  AS minratio,
        skewsize,
        (skewsize/avgsize)::numeric(4,3)  AS skewratio,
        skewstddev::numeric(1000)
FROM skew
WHERE totalsize > 0;
```

表的分布键的选择方法：

1. 如果此列的distinct值比较大，并且没有明显的数据倾斜，也可以把多列定义成分布列。

   查看distinct的大小：

   ```
   SELECT count(distinct column1) FROM table;
   ```

   查看数据是否存在倾斜：

   ```
   SELECT count(*) cnt, column1 FROM table group by column1 order by cnt limit 100;
   ```

2. 选用经常做JOIN或group by的列，可以减少STREAM运算。
3. 不推荐以下分布键选择方式：
   a. 分布列用默认值（第一列）。
   b. 分布列用sequence自增生成。
   c. 分布列用随机数生成（除非任意列，或者任意两列的组合做分布键都是倾斜的，一般不选用这种方法）。

6.7 VACUUM FULL 一张表后，表文件大小无变化

问题现象

使用VACUUM FULL命令对一张表进行清理，清理完成后表大小和清理前一样大。

原因分析

假定该表的名称为table_name，对于该现象可能有以下两种原因：

1. 表本身没有delete过数据，使用VACUUM FULL table_name后无需清理delete的数据，因此表大小清理前后一样大。
2. 在执行VACUUM FULL table_name时有并发的事务存在，可能会导致VACUUM FULL跳过清理最近删除的数据，导致清理不完全。

解决办法

对于可能原因的第二种情况，给出如下两种处理方法：

- 如果在VACUUM FULL时有并发的事务存在，此时需要等待所有事务结束，再次执行VACUUM FULL命令对该表进行清理。
- 如果使用上面的方法清理后，表文件大小仍然无变化，在确认集群中没有正在运行的任务，且所有数据都已经保存后，可使用如下操作方式：

步骤1 执行以下命令查询当前的事务XID。

```
SELECT txid_current();
```

步骤2 再执行以下命令查看活跃事务列表：

```
SELECT txid_current_snapshot();
```

步骤3 如果发现活跃事务列表中有XID比当前的事务XID小时，重启集群后，再次使用VACUUM FULL命令对该表进行清理。

----结束

6.8 删除表数据后执行了 VACUUM，但存储空间并没有释放

问题现象

删除表数据后执行了VACUUM，但是存储空间并没有释放。

原因分析

- 执行VACUUM时，对某些表可能没有权限，或者数据库本身并没有太多的数据膨胀。
- 执行VACUUM，默认清理当前用户在数据库中拥有权限的每一个表，没有权限的表则直接跳过回收操作。
- 参数vacuum_defer_cleanup_age不是0，该参数在老版本默认为8000，表示最近8000个事务产生的脏数据不进行回收。
- 为了保证事务可见性，产生脏数据的事务号，如果大于当前活跃的老事务号，则这部分脏数据也不会清理。

处理方法

- 建议对单个表执行VACUUM FULL命令，命令格式为“VACUUM FULL 表名”。
- 如果您对表没有权限，请联系数据库管理员或表的所有者进行处理。
- 对于vacuum_defer_cleanup_age不是0的场景，可以将此参数改为0，取消VACUUM的事务延迟。

- 对于存在老事务的场景，重启集群再重新执行VACUUM FULL可以保证空间一定回收，否则只能等老事务结束再执行VACUUM FULL。

6.9 执行 VACUUM FULL 命令时报错：Lock wait timeout

问题现象

执行VACUUM FULL命令时报错：

```
[0]ERROR: dn_6009_6010: Lock wait timeout: thread 140158632457984 on node dn_6009_6010 waiting for AccessExclusiveLock on relation 2299036 of database 14522 after 1202001.968 ms
Detail: blocked by hold lock thread 140150147380992, statement <<backend information not available>>, hold lockmode AccessShareLock.
Line Number: 1
```

原因分析

日志中的“Lock wait timeout”说明锁等待超时。锁等待超时一般是因为有其他的SQL语句已经持有了锁，当前SQL语句需要等待持有锁的SQL语句执行完毕释放锁之后才能执行。当申请的锁等待时间超过GUC参数lockwait_timeout的设定值时，系统会报LOCK_WAIT_TIMEOUT的错误。

执行VACUUM FULL命令时出现报错的原因一般为执行命令超时，如果对整个数据库执行VACUUM FULL执行时间较长可能会超时。

处理方法

建议对单个表执行VACUUM FULL命令，命令格式为“VACUUM FULL 表名”，同时增加执行“VACUUM FULL”命令的频率。尤其是对于频繁增、删、改的表，建议定期做VACUUM FULL操作。

6.10 VACUUM FULL 执行慢

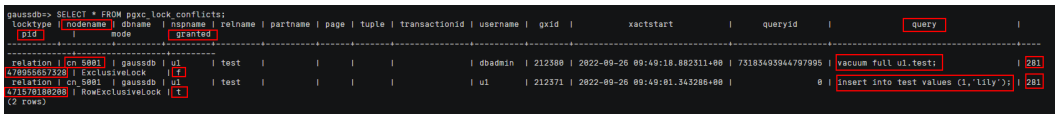
VACUUM FULL执行慢的常见场景及处理方法如下：

场景一：存在锁等待导致 VACUUM FULL 执行慢

- **8.1.x及以上集群版本的处理方法：**

步骤1 通过查询pgxc_lock_conflicts视图查看锁冲突情况。

```
SELECT * FROM pgxc_lock_conflicts;
```



- 在查询结果中查看granted字段为“f”，表示VACUUM FULL语句正在等待其他锁。granted字段为“t”，表示INSERT语句是持有锁。nodename，表示锁产生的位置，即CN或DN位置，例如cn_5001，继续执行2。
- 如果查询结果为0 rows，则表示不存在锁冲突。则需排查其它场景。

步骤2 根据语句内容判断是终止持锁语句后继续执行VACUUM FULL还是在业务低峰期选择合适的时间执行VACUUM FULL。

如果要终止持锁语句，则执行以下语句。*pid*从上述步骤1获取，cn_5001为所查询到的nodename。

```
execute direct on (cn_5001) 'SELECT PG_TERMINATE_BACKEND(pid)';
```

```
gaussdb=> execute direct on (cn_5001) 'SELECT PG_TERMINATE_BACKEND(281470955657328)';
 pg_terminate_backend
----------------------
 t
(1 row)
```

步骤3 语句终止后，再重新执行VACUUM FULL。

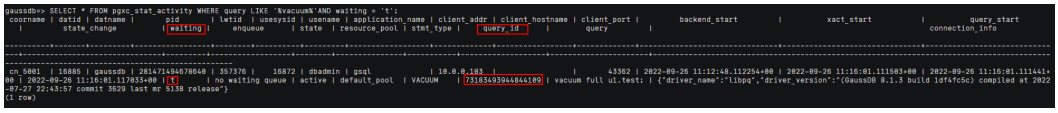
```
VACUUM FULL table_name;
```

----结束

- **8.0.x及以前版本的处理方法：**

步骤1 在数据库中执行语句，获取VACUUM FULL操作对应的query_id。

```
SELECT * FROM pgxc_stat_activity WHERE query LIKE '%vacuum%'AND waiting = 't';
```



步骤2 根据**步骤1**获取的query_id，执行以下语句查看是否存在锁等待。

```
SELECT * FROM pgxc_thread_wait_status WHERE query_id = {query_id};
```

```
gaussdb=> SELECT * FROM pgxc_thread_wait_status WHERE query_id = 73183493944844109;
 node_name | db_name | thread_name |      query_id       |       tid       | lwtid  | ptid | tlevel | smpid | wait_status  | wait_event
-----------+---------+-------------+---------------------+-----------------+--------+------+--------+-------+--------------+------------
 cn_5001   | gaussdb | gsql        | 73183493944844109 | 281471494678640 | 357376 |      |      0 |     0 | acquire lock | relation
(1 row)
```

- 查询结果中“wait_status”存在“acquire lock”表示存在锁等待。同时查看“node_name”显示在对应的CN或DN上存在锁等待，记录相应的CN或DN名称，例如cn_5001或dn_600x_600y，继续执行**步骤3**。
- 查询结果中“wait_status”不存在“acquire lock”，排除锁等待情况，继续排查其它场景。

步骤3 执行以下语句，到等锁的对应CN或DN上从pg_locks中查看VACUUM FULL操作在等待哪个锁。以cn_5001为例，如果在DN上等锁，则改为相应的DN名称。pid为**步骤2**获取的tid。

回显中记录relation的值。

```
execute direct on (cn_5001) 'SELECT * FROM pg_locks WHERE pid = {tid} AND granted = ''f''';
```

```
gaussdb=> execute direct on (cn_5001) 'SELECT * FROM pg_locks WHERE pid = 281471494678640 AND granted = ''f''';
 locktype | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtransaction |       pid       |     mode      | granted | fastpath
----------+----------+----------+------+-------+------------+---------------+---------+-------+----------+--------------------+-----------------+---------------+---------+----------
 relation |    16885 |    25064 |      |       |            |               |         |       |          | 14/3090            | 281471494678640 | ExclusiveLock | f       | f
(1 row)
```

步骤4 根据获取的relation，从pg_locks中查看当前持有锁的pid。*relation*从**步骤3**获取。

```
execute direct on (cn_5001) 'SELECT * FROM pg_locks WHERE relation = {relation} AND granted = ''t''';
```

```
gaussdb=> execute direct on (cn_5001) 'SELECT * FROM pg_locks WHERE relation = 25064 AND granted = ''t''';
 locktype | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtransaction |       pid       |       mode       | granted | fastpath
----------+----------+----------+------+-------+------------+---------------+---------+-------+----------+--------------------+-----------------+------------------+---------+----------
 relation |    16885 |    25064 |      |       |            |               |         |       |          | 17/4647            | 281471060535408 | RowExclusiveLock | t       | f
(1 row)
```

步骤5 根据pid，执行以下语句查到对应的SQL语句。*pid*从**步骤4**获取。

```
execute direct on (cn_5001) 'SELECT query FROM pg_stat_activity WHERE pid ={pid}';
```

```
gaussdb=> execute direct on (cn_5001) 'SELECT query FROM pg_stat_activity WHERE pid=281471060535408';
                query
-------------------------------------
 insert into test values (1, 'lily');
(1 row)
```

步骤6 根据语句内容判断是终止持锁语句后继续执行VACUUM FULL还是在业务低峰期选择合适的时间执行VACUUM FULL。

如果要终止持锁语句，则执行以下语句。*pid*从上述**步骤4**获取，cn_5001为所查询到的nodename。

```
execute direct on (cn_5001) 'SELECT PG_TERMINATE_BACKEND(pid)';
```

```
gaussdb=> execute direct on (cn_5001) 'SELECT PG_TERMINATE_BACKEND(281471060535408)';
 pg_terminate_backend
----------------------
 t
(1 row)
```

步骤7 语句终止后，再执行VACUUM FULL。

```
VACUUM FULL table_name;
```

----结束

场景二：存在 I/O 和网络问题导致事务无法提交

处理方法：执行一个简单的CRETAE TABLE语句，如果CRETAE TABLE语句执行也很慢，说明可能存在I/O和网络问题，可以进一步排查I/O和网络情况。

场景三：系统表过大导致 VACUUM FULL 执行慢

在排除I/O和网络问题后，对空表执行VACUUM FULL，即使是空表执行VACUUM FULL也比较慢，则说明是系统表较大导致。因为VACUUM FULL任意一张表时，都会扫描pg_class、pg_partition、pg_proc三张系统表，当这三个系统表过大时，也会导致VACUUM FULL执行较慢。

处理方法：GaussDB(DWS)支持对系统表执行VACUUM FULL，但是会产生八级锁，涉及相关系统表的业务会被阻塞，**注意要在业务空闲时间窗或停止业务期间且没有DDL操作时**清理系统表。

场景四：列存表使用了局部聚簇（PCK）时，VACUUM FULL 执行慢

对列存表执行VACUUM FULL时，如果存在PCK，就会将PARTIAL_CLUSTER_ROWS中多条记录全都加载到内存中再进行排序，如果表较大或psort_work_mem设置较小，会导致PCK排序时产生下盘（数据库选择将临时结果暂存到磁盘），进行外部排序；一旦进行外部排序，时间消耗就会增加很多。

处理方法：根据表中数据的tuple length，合理调整PARTIAL_CLUSTER_ROWS和psort_work_mem的大小。

1. 执行以下语句查看表定义。回显中存在“PARTIAL CLUSTER KEY”信息，表示存在PCK。
   ```
   SELECT * FROM pg_get_tabledef('table name');
   ```



```
postgres=> select * from pg_get_tabledef('customer_t1');
                        pg_get_tabledef
-----------------------------------------------------------------------
 SET search_path = dbadmin;                                           +
 CREATE  TABLE customer_t1 (                                          +
         c_customer_sk integer,                                       +
         c_customer_id character(5),                                  +
         c_first_name character(6),                                   +
         c_last_name character(8)                                     +
 )                                                                    +
 WITH (orientation=column, compression=middle)                        +
 DISTRIBUTE BY HASH(c_last_name)                                      +
 TO GROUP group_version1;                                             +
 ALTER TABLE customer_t1 ADD PARTIAL CLUSTER KEY (c_customer_sk);
(1 row)
```

2. 登录DWS管理控制台，左侧选择“集群管理”。
3. 单击对应的集群名称，进入集群详情页。
4. 左侧选择“参数修改”，在搜索栏中输入psort_work_mem进行搜索，将CN参数值和DN参数值同时调大，单击“保存”。
5. 再重新执行VACUUM FULL操作。

6.11 表数据膨胀导致 SQL 查询慢，用户前台页面数据加载不出

问题现象

数据库中原先执行几秒钟的SQL语句，现在执行二十几秒未出结果，导致前台页面数据加载超时，无法对用户提供图表显示。

原因分析

- 大量表频繁增删改，未及时清理，导致脏数据过多、表数据膨胀、查询慢。
- 内存参数设置不合理。

分析过程

步骤1 和用户确认是部分业务慢，获取部分慢SQL后，打印执行计划，分析出耗时主要在index scan上，可能是I/O争抢导致，通过监控I/O，发现并没有I/O资源使用瓶颈。

```
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------
   3 --Hash Join (4,18)
       Hash Cond: ((t1.area_code)::text = (t5.area_code)::text)
   5 --Hash Join (6,16)
       Hash Cond: ((t1.measure_unit_code)::text = (t4.measure_unit_code)::text)
   7 --Hash Join (8,14)
       Hash Cond: ((t1.value_type_code)::text = (t3.value_type_code)::text)
   9 --Hash Join (10,12)
       Hash Cond: ((t1.index_code)::text = (t2.index_code)::text)
  11 --Index Scan using idx_m_ss_index_event_index on ioc_dm.m_ss_index_event t1
       Filter: ((t1.data_status = 1::numeric) AND (length((t1.occur_period)::text) = 2) AND ((t1.index_code)::bigint = 10010001001l::bigint) AND ((t
1.area_code)::bigint = 440307000000::bigint) AND ((t1.time_type_code)::bigint = 8) AND (to_char(t1.update_time, 'yyyymmdd'::text) = to_char(('2020-02-
25'::date)::timestamp with time zone, 'yyyymmdd'::text)))
       Rows Removed by Filter: 51913
(11 rows)
```



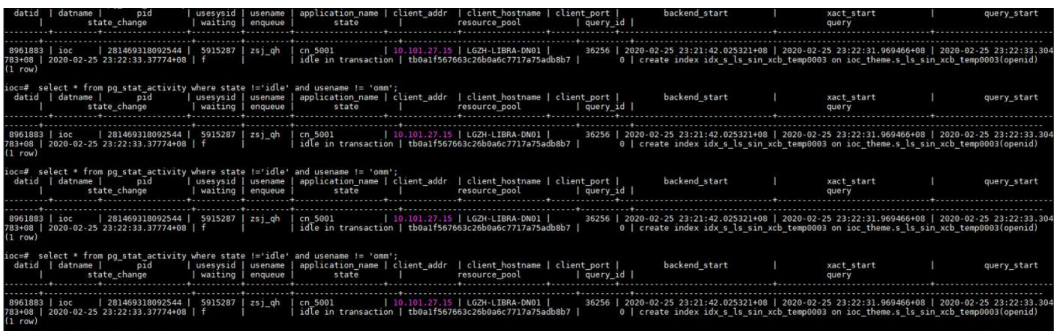
```
id |                                                operation                                                |        A-time         | A-rows | E
-rows |  Peak Memory   | A-width | E-width | E-costs
------+----------------+---------+---------+----------
  1 | ->  Streaming (type: GATHER)                                                                            | 21306.396             |     16 |
  15 | 303KB          |         |     339 | 15154.56
  2 |    ->  Sort                                                                                             | [42.902,21259.337]    |     16 |
  15 | [31KB, 33KB]   | [0,58]  |     339 | 15153.23
  3 |       ->  Hash Join (4,18)                                                                              | [42.872,21259.214]    |     16 |
  15 | [8KB, 8KB]     |         |     339 | 15153.22
  4 |          ->  Streaming(type: REDISTRIBUTE)                                                              | [21213.855,21216.616] |     16 |
  15 | [142KB, 144KB] |         |     270 | 15124.36
  5 |             ->  Hash Join (6,16)                                                                        | [21218.306,21249.577] |     16 |
  15 | [7KB, 7KB]     |         |     270 | 15122.84
  6 |                ->  Streaming(type: BROADCAST)                                                           | [21217.594,21248.730] |    240 |
 225 | [144KB, 144KB] |         |     190 | 15097.21
  7 |                   ->  Hash Join (8,14)                                                                  | [0.313,21225.478]     |     16 |
  15 | [7KB, 7KB]     |         |     190 | 15092.59
  8 |                      ->  Streaming(type: BROADCAST)                                                     | [21198.318,21224.950] |     96 |
 225 | [144KB, 144KB] |         |     110 | 15066.96
  9 |                         ->  Hash Join (10,12)                                                           | [21208.801,21218.437] |     16 |
  15 | [7KB, 7KB]     |         |     110 | 15064.65
 10 |                            ->  Streaming(type: BROADCAST)                                               | [21161.272,21215.516] |    240 |
  15 | [144KB, 144KB] |         |      41 | 15039.07
 11 |                               ->  Index Scan using idx_m_ss_index_event_index on ioc_dm.m_ss_index_event t1 | [762.317,21177.799] |     16 |
   1 | [40KB, 40KB]   |         |      41 | 15036.76
 12 |                            ->  Hash                                                                     | [0.469,55.742]        |   2310 |
 152 | [291KB, 291KB] | [46,48] |     116 | 25.25
 13 |                               ->  Seq Scan on ioc_ods.o_gwd_ioc_index t2                                | [0.335,55.571]        |   2310 |
 150 | [16KB, 16KB]   |         |     116 | 25.25
 14 |                         ->  Hash                                                                        | [0.003,15.530]        |      8 |
 152 | [259KB, 291KB] | [0,32]  |     116 | 25.25
 15 |                            ->  Seq Scan on ioc_ods.o_gwd_ioc_value_type t3                              | [0.001,15.516]        |      8 |
 150 | [15KB, 15KB]   |         |     116 | 25.25
 16 |                      ->  Hash                                                                           | [0.049,0.105]         |    160 |
 152 | [291KB, 291KB] | [25,29] |     116 | 25.25
 17 |                         ->  Seq Scan on ioc_ods.o_gwd_ioc_measure_unit t4                               | [0.037,0.085]         |    160 |
```



```
11 --Index Scan using idx_m_ss_index_event_index on ioc_dm.m_ss_index_event t1
      dn_6001_6002 (actual time=0.209..2142.458 rows=3 loops=1)
      dn_6003_6004 (actual time=762.317..762.317 rows=0 loops=1)
      dn_6005_6006 (actual time=9.738..20835.282 rows=2 loops=1)
      dn_6007_6008 (actual time=7345.547..7345.547 rows=0 loops=1)
      dn_6009_6010 (actual time=0.257..7235.551 rows=4 loops=1)
      dn_6011_6012 (actual time=20024.688..20024.688 rows=0 loops=1)
      dn_6013_6014 (actual time=17878.685..17878.685 rows=0 loops=1)
      dn_6015_6016 (actual time=17078.916..17078.916 rows=0 loops=1)
      dn_6017_6018 (actual time=17827.799..17827.799 rows=0 loops=1)
      dn_6019_6020 (actual time=0.253..15975.299 rows=2 loops=1)
      dn_6021_6022 (actual time=21177.799..21177.799 rows=0 loops=1)
      dn_6023_6024 (actual time=0.278..15016.516 rows=1 loops=1)
      dn_6025_6026 (actual time=0.264..16628.971 rows=2 loops=1)
      dn_6027_6028 (actual time=0.270..16635.989 rows=2 loops=1)
      dn_6029_6030 (actual time=12725.526..12725.526 rows=0 loops=1)
      dn_6001_6002 (Buffers: shared hit=485 read=22013 written=5)
      dn_6003_6004 (Buffers: shared hit=512 read=22041 written=4)
      dn_6005_6006 (Buffers: shared hit=481 read=22080 written=43)
      dn_6007_6008 (Buffers: shared hit=539 read=22105 written=12)
      dn_6009_6010 (Buffers: shared hit=463 read=22074 written=13)
      dn_6011_6012 (Buffers: shared hit=481 read=22128 written=65)
      dn_6013_6014 (Buffers: shared hit=534 read=22067 written=73)
      dn_6015_6016 (Buffers: shared hit=560 read=22153 written=50)
      dn_6017_6018 (Buffers: shared hit=535 read=21961 written=44)
      dn_6019_6020 (Buffers: shared hit=507 read=22133 written=58)
      dn_6021_6022 (Buffers: shared hit=466 read=22190 written=41)
      dn_6023_6024 (Buffers: shared hit=476 read=22087 written=44)
      dn_6025_6026 (Buffers: shared hit=502 read=21973 written=44)
      dn_6027_6028 (Buffers: shared hit=442 read=22111 written=44)
      dn_6029_6030 (Buffers: shared hit=476 read=22009 written=39)
      dn_6001_6002 (CPU: ex c/r=35707713, ex cyc=107123139, inc cyc=107123139)
      dn_6003_6004 (CPU: ex c/r=0, ex cyc=38115922, inc cyc=38115922)
      dn_6005_6006 (CPU: ex c/r=520883939, ex cyc=1041767878, inc cyc=1041767878)
      dn_6007_6008 (CPU: ex c/r=0, ex cyc=367278665, inc cyc=367278665)
      dn_6009_6010 (CPU: ex c/r=90444697, ex cyc=361778791, inc cyc=361778791)
      dn_6011_6012 (CPU: ex c/r=0, ex cyc=1001235015, inc cyc=1001235015)
      dn_6013_6014 (CPU: ex c/r=0, ex cyc=893934788, inc cyc=893934788)
      dn_6015_6016 (CPU: ex c/r=0, ex cyc=853946318, inc cyc=853946318)
      dn_6017_6018 (CPU: ex c/r=0, ex cyc=891390498, inc cyc=891390498)
      dn_6019_6020 (CPU: ex c/r=399382685, ex cyc=798765371, inc cyc=798765371)
      dn_6021_6022 (CPU: ex c/r=0, ex cyc=1058894369, inc cyc=1058894369)
      dn_6023_6024 (CPU: ex c/r=750828892, ex cyc=750828892, inc cyc=750828892)
      dn_6025_6026 (CPU: ex c/r=415725991, ex cyc=831451982, inc
```

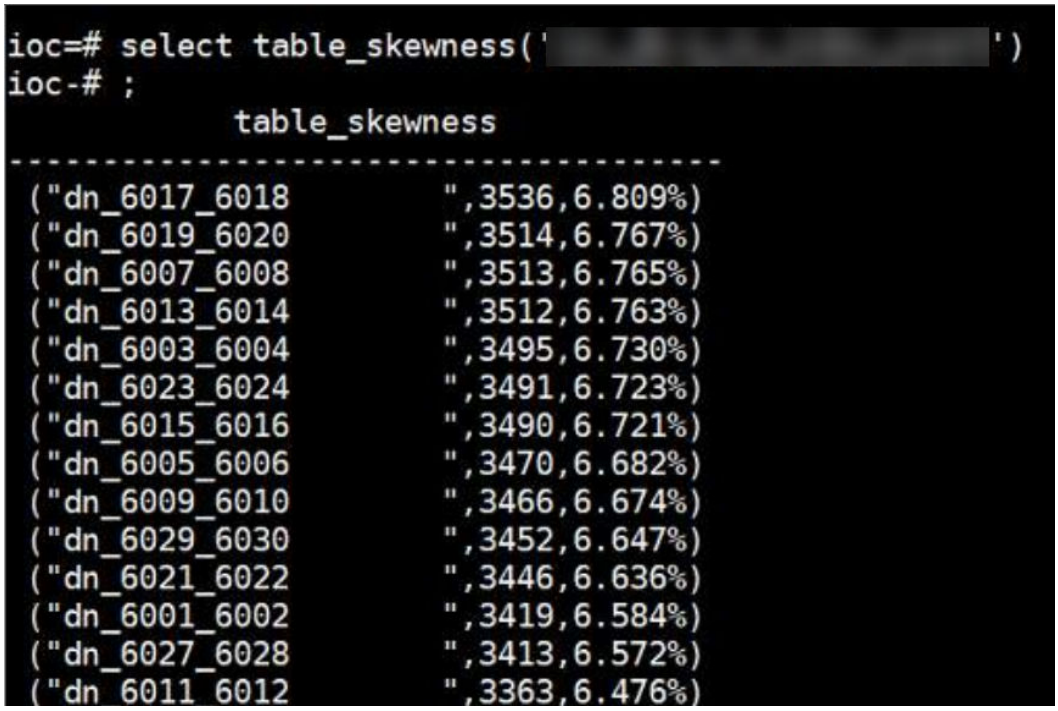
步骤2 查询当前活跃SQL，发现有大量的CREATE INDEX语句，需要和用户确认该业务是否合理。

```
SELECT * from pg_stat_activity where state !='idle' and usename !='Ruby';
```



步骤3 根据执行计划，发现在部分DN上耗时较高，查询表的倾斜情况，并未发现有倾斜的情况。

```
SELECT table_skewness('table name');
```



步骤4 联系运维人员登录集群实例，检查内存相关参数，设置不合理，需要优化。

- 单节点总内存大小为256GB。
- max_process_memory为12GB，设置过小。
- shared_buffers为32MB，设置过小。
- work_mem：CN：64MB 、DN：64MB。
- max_active_statements为-1（不限制并发数）。

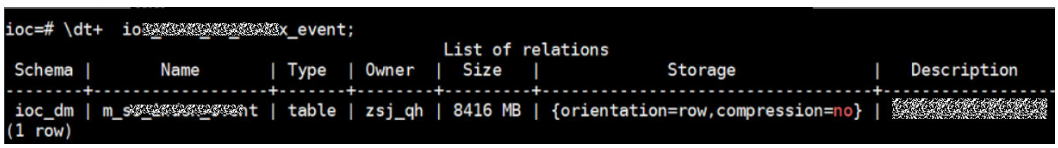
步骤5 按以下值设置：

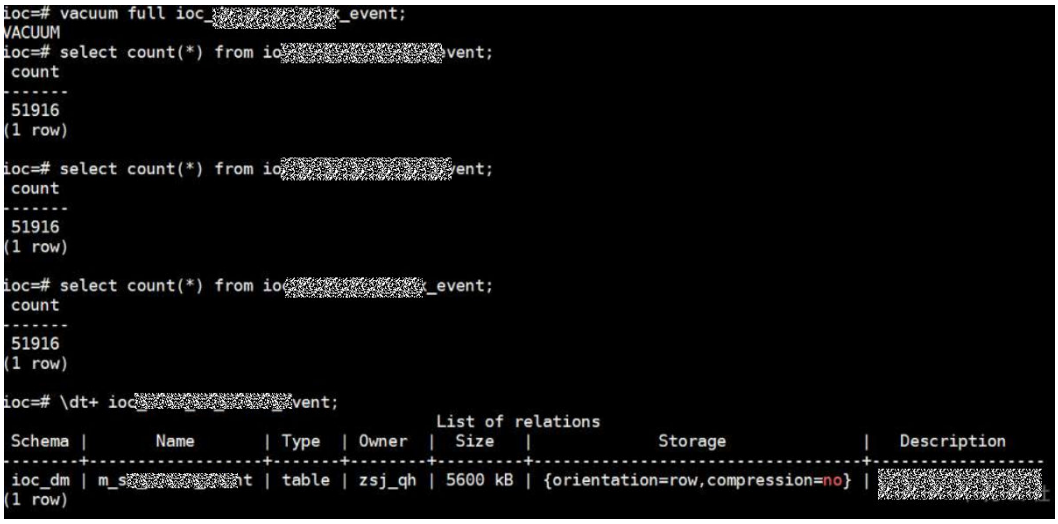
gs_guc set -Z coordinator -Z datanode -N all -I all -c "max_process_memory=25GB"

gs_guc set -Z coordinator -Z datanode -N all -I all -c "shared_buffers=8GB"

gs_guc set -Z coordinator -Z datanode -N all -I all -c "work_mem=128MB"

步骤6 进一步分析扫描慢的原因，发现表数据膨胀严重，对其中一张8GB大小的表，总数据量5万条，做完VACUUM FULL后大小减小为5.6MB。





----结束

处理方法

步骤1 对业务涉及到的常用的大表，执行VACUUM FULL操作，清理脏数据。

步骤2 设置GUC内存参数。

----结束

6.12 集群报错内存溢出

问题现象

查看日志提示：

```
[ERROR] Mpp task queryDataAnalyseById or updateDataAnalyseHistoryEndTimesAndResult fail, dataAnalyseId:17615 org.postgresql.util.PSQLException: ERROR: memory is temporarily unavailable
sql：vacuum full dws_customer_360.t_user_resource;
```

原因分析

存在部分SQL语句使用内存资源过多，造成内存资源耗尽，其余语句执行作业时无法分配到内存就提示内存不足。

处理方法

1. 调整业务执行时间窗，与高并发执行业务的时间错峰执行。
2. 查询当前集群的内存使用情况，找到内存使用过高的语句并及时终止，释放资源之后集群内存就会恢复。具体的操作步骤如下：

- **8.1.1及之前集群版本连接数据库后执行以下步骤：**

步骤1 执行以下语句查询当前集群的内存使用情况，观察是否有实例的dynamic_used_memory已经大于或者接近于该实例的max_dynamic_memory，出现上述报错，一般为dynamic_used_memory达到上限。

```
SELECT * FROM pgxc_total_memory_detail;
```


步骤2 开启topsql的情况下，使用实时topsql查询正在执行的高内存query语句，根据结果中的max_peak_memory以及memory_skew_percent值，较大的值就是消耗内存较多的语句。

```
SELECT
nodename,pid,dbname,username,application_name,min_peak_memory,max_peak_memory,average_peak_memory,memory_skew_percent,substr(query,0,50) as query FROM pgxc_wlm_session_statistics;
```

步骤3 根据步骤2查询的会话信息，通过执行pg_terminate_backend函数结束相应会话，即可恢复内存。恢复后可根据实际业务情况重新优化该SQL语句。

```
SELECT pg_terminate_backend(pid);
```

----结束

- **8.1.2及以上集群版本支持登录GaussDB(DWS) 管理控制台，在实时查询监控页面执行以下步骤：**

须知

- 实时查询仅8.1.2及以上集群版本支持。
- 启动实时查询功能需要在“监控设置>监控采集”页面打开“实时查询监控”指标项。

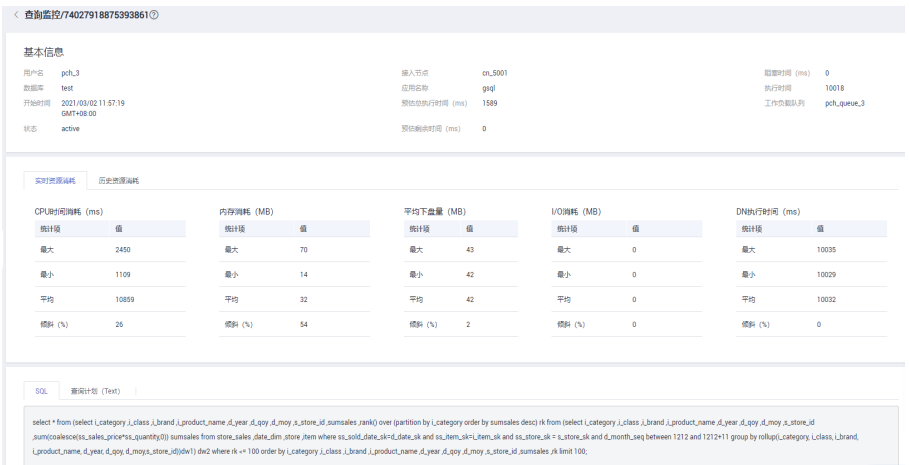
步骤1 登录GaussDB(DWS) 管理控制台，在“集群管理”页面，找到需要查看监控的集群，在集群所在行的“操作”列，单击“监控面板”，系统将显示数据库监控页面。

步骤2 在左侧导航栏选择“监控 > 实时查询”，进入实时查询监控页面。

步骤3 根据选择的指定时间段浏览集群中正在运行的所有查询信息。

步骤4 单击指定实时查询监控的会话查询ID，进入该会话ID的实时查询的详情页面，在详情页面中会展示当前监控的详细内容。例如用户名称、数据库名称、执行时间、查询语句、查询状态、排队状态、dn最小内存峰值、dn最大内存峰值、dn每秒最大io峰值、dn每秒最小io峰值和内存使用平均值等信息。

根据“dn最大内存峰值”和“内存使用平均值”查询结果，值较大的就是消耗内存较多的语句。



步骤5 如果已确认所查询的占用内存高的语句需要被终止，勾选指定的查询ID后，单击“终止查询”按钮，终止查询。

如果设置了细粒度权限控制功能，只有配置了操作权限的用户才能使用终止查询按钮。只读权限用户登录后终止查询按钮为灰色。

----结束

6.13 带自定义函数的语句不下推

问题现象

SQL语句不下推。

原因分析

目前GaussDB(DWS)可以支持绝大多数常用函数的下推，不下推函数的场景主要出现在自定义函数属性定义错误的情况下。

不下推语句的执行方式没有利用分布式的优势，其在执行过程中，相当于把大量的数据和计算过程汇集到一个节点上去做，因此性能通常非常差。

分析过程

步骤1 通过EXPLAIN VERBOSE打印语句执行计划。

```
tpcds1xcpm=#  explain verbose
 select func_add_sql(sr_customer_sk,sr_store_sk )
 ,sum(SR_FEE) as ctr_total_return
 from store_returns
 ,date_dim
 where sr_returned_date_sk = d_date_sk
 and d_year =2000
 group by 1;
                                                             QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=1820.37..2419.67 rows=47944 width=46)
   Output: ((store_returns.sr_customer_sk + store_returns.sr_store_sk)), sum(store_returns.sr_fee)
   Group By Key: (store_returns.sr_customer_sk + store_returns.sr_store_sk)
   ->  Hash Join  (cost=4.56..1580.65 rows=47944 width=14)
         Output: (store_returns.sr_customer_sk + store_returns.sr_store_sk), store_returns.sr_fee
         Hash Cond: (store_returns.sr_returned_date_sk = date_dim.d_date_sk)
         ->  Data Node Scan on store_returns "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00 rows=287514 distinct=1993 width=18)
               Output: store_returns.sr_customer_sk, store_returns.sr_store_sk, store_returns.sr_fee, store_returns.sr_returned_date_sk
               Node/s: All datanodes
               Remote query: SELECT sr_customer_sk, sr_store_sk, sr_fee, sr_returned_date_sk FROM ONLY public.store_returns WHERE true
         ->  Hash  (cost=0.00..0.00 rows=365 distinct=365 width=4)
               Output: date_dim.d_date_sk
               ->  Data Node Scan on date_dim "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00 rows=365 width=4)
                     Output: date_dim.d_date_sk
                     Node/s: All datanodes
                     Remote query: SELECT d_date_sk FROM ONLY public.date_dim WHERE d_year = 2000
(16 rows)
```


上述执行计划中出现__REMOTE关键字，表示当前的语句为不下推执行。

步骤2 不下推语句在pg_log中会打印不下推的原因，上述语句在CN的日志中会找到类似以下的日志。

```
2020-11-07 17:20:28.894 CST zyl tpcdslxcpm [local] 140573226825472 0 [BACKEND] LOG:  SQL can't be shipped, reason: Function func_add_sql() can not be shipped
2020-11-07 17:20:28.894 CST zyl tpcdslxcpm [local] 140573226825472 0 [BACKEND] STATEMENT:  explain verbose
        select func_add_sql(sr_customer_sk,sr_store_sk )
        ,sum(SR_FEE) as ctr_total_return
        from store_returns
        ,date_dim
        where sr_returned_date_sk = d_date_sk
        and d_year =2000
        group by 1;
```

----结束

处理方法

审视用户自定义函数的provolatile属性是否定义正确。如果定义不正确，要修改对应的属性，使它能够下推执行。

具体判断方法可以参考如下说明：

- 函数相关的所有属性都在pg_proc系统表中可以查到，与函数能否下推相关的两个属性是provolatile和proshippable：
  - 如果函数的provolatile属性为i，则无论proshippable的值是否为t，则函数始终可以下推。
  - 如果函数的provolatile属性为s或v，则仅当proshippable的值为t时，函数可以下推。
- provolatile的本质含义是描述函数的易变属性，取值为i/s/v。i代表IMMUTABLE，s代表STABLE，v代表VOLATILE。

  例如：
  - 如果一个函数对于同样的输入，一定有相同的输出，那么这类函数就是IMMUTABLE的。例如，绝大部分的字符串处理函数，这类函数始终可以下推。
  - 如果一个函数的返回结果在一个SQL语句的调用过程中，结果是相同的，那么它就是STABLE的。例如，时间相关的处理函数，它的最终显示结果可能与具体的GUC参数相关（例如控制时间显示格式的参数），这类函数都是STABLE的，此类函数仅当其属性是SHIPPABLE的时候，才能下推。
  - 如果一个函数的返回结果可能随着每一次的调用而返回不同的结果。例如，nextval、random这种函数，每次调用结果都是不可预期的，这类函数就是VOLATILE的，此类函数仅当其属性是SHIPPABLE的时候，才能下推。
- proshippable字段表示函数是否可以下推到DN上执行，默认值是false，取值范围为t/f/NULL。

6.14 列存表更新失败或多次更新后出现表膨胀

问题现象

- 对列存表更新或UPDATE会失败。
- 多次对列存表UPDATE，发现表大小膨胀了十多倍。

原因分析

- 列存表不支持并发更新。

- 列存表的更新操作，空间不会回收旧记录。

处理方法

- **方法一**

说明

该处理方法仅8.1.3及以上集群版本支持。

步骤1 登录GaussDB(DWS) 管理控制台。

步骤2 在集群列表中单击指定集群名称。

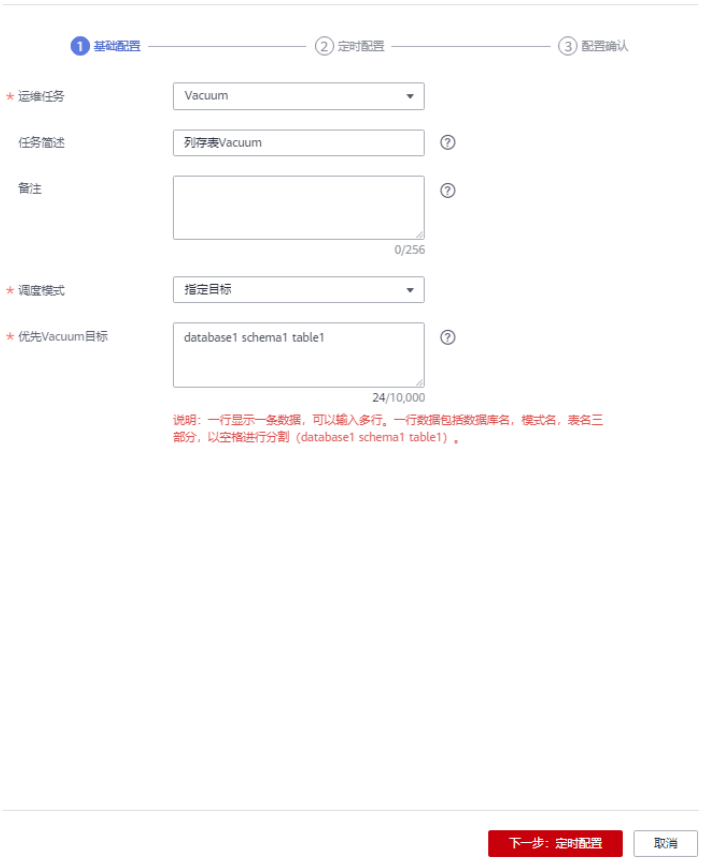
步骤3 进入“集群详情”页面，切换至“智能运维”页签。

步骤4 在运维详情部分切换至运维计划模块。单击“添加运维任务”按钮。

步骤5 弹出添加运维任务边栏。

- 运维任务选择“Vacuum”。
- 调度模式选择“指定目标”，智能运维将在指定时间窗内，自动下发表级Vacuum任务。

  用户可配置需要Vacuum的列存表，其中一行对应一张表，每张表以数据库名、模式名、表名表示，以空格进行分割。



步骤6 单击“下一步：定时配置”，配置Vacuum类型，推荐选择“周期型任务”，GaussDB(DWS)将自动在自定义时间窗内执行Vacuum。

步骤7 确认无误后，单击“下一步：配置确认”，完成配置。

----结束

- **方法二**

对列存表更新操作后，需要进行VACUUM FULL清理，更多用法请参见**VACUUM**的“VACUUM”章节。

```
VACUUM FULL table_name;
```

6.15 列存表多次插入后出现表膨胀

问题现象

列存表多次执行INSERT后，发现表膨胀。

原因分析

列存表数据按列存储，一列的每60000行存储为一个CU，同一列的CU连续存储在一个文件中，当该文件大于1GB时，切换到新文件中。CU文件数据不能更改只能追加写。对频繁进行删除和更新的列存表VACUUM后，由于列存表的CU无法更改，即使标识为可用的空间也是无法进行复用的（复用需要更改CU）。因此不建议在GaussDB(DWS)中对列存表频繁进行删除和更新 。

处理方法

建议开启列存表的delta表功能。

```
ALTER TABLE table_name SET (ENABLE_DELTA = ON);
```

说明

- 开启列存表的delta表功能，在导入单条或者小规模数据进入表中时，能够防止小CU的产生，所以开启delta表能够带来显著的性能提升，例如在3CN、6DN的集群上操作，每次导入100条数据，导入时间能减少25%，存储空间减少97%，所以在需要多次插入小批量数据前应该先开启delta表，等到确定接下来没有小批量数据导入了再关闭。
- delta表就是列存表附带的行存表，那么将数据插入delta表后将失去列存表的高压缩比等优势，正常情况下使用列存表的场景都是大批量数据导入，所以默认关闭delta表，如果开启delta表做大批量数据导入，反而会额外消耗更多时间和空间，同样在3CN、6DN的集群上操作，每次导入10000条数据时，开启delta表会比不开启时慢4倍，额外消耗10倍以上的空间。所以开启delta表需谨慎，根据实际业务需要来选择开启和关闭。

6.16 往 GaussDB(DWS)写数据慢，客户端数据会有积压

问题现象

客户端往GaussDB(DWS)写入数据较慢，客户端数据会有积压。

原因分析

如果通过单条INSERT INTO语句的方式单并发写数据入库，客户端很可能会出现瓶颈。INSERT是最简单的一种数据写入方式，适合数据写入量不大，并发度不高的场景。

处理方法

如果遇到写数据慢的问题，建议通过以下两种方式进行处理：

- 建议选择其他更加高效的数据导入方式，例如使用COPY方式导入数据。
  有关导入方式的详细信息，请参见**导入方式说明**。
- 增大客户端并发数。

6.17 分析查询效率异常降低的问题

通常在几十毫秒内完成的查询，有时会突然需要几秒的时间完成；而通常需要几秒完成的查询，有时需要半小时才能完成。如何分析这种查询效率异常降低的问题呢?

解决办法

通过下列的操作步骤，可以分析出查询效率异常降低的原因。

步骤1 使用ANALYZE命令分析数据库。

ANALYZE命令更新所有表中数据大小以及属性等相关统计信息，该命令较为轻量级，可以经常执行。如果此命令执行后性能恢复或者有所提升，则表明AUTOVACUUM未能很好的完成它的工作，有待进一步分析。

步骤2 检查查询语句是否返回了多余的数据信息。

例如，如果查询语句先查询一个表中所有的记录，而只用到结果中的前10条记录。对于包含50条记录的表，查询起来是很快的；但是，当表中包含的记录达到50000，查询效率将会有所下降。

若业务应用中存在只需要部分数据信息，但是查询语句却是返回所有信息的情况，建议修改查询语句，增加LIMIT子句来限制返回的记录数。这样至少使数据库优化器有了一定的优化空间，一定程度上会提升查询效率。

步骤3 检查查询语句单独运行时是否仍然较慢。

尝试在数据库没有其他查询或查询较少的时候运行查询语句，并观察运行效率。如果效率较高，则说明可能是由于之前运行数据库系统的主机负载过大导致查询低效。此外，还可能是由于执行计划比较低效，但是由于主机硬件较快使得查询效率较高。

步骤4 检查重复相同查询语句的执行效率。

查询效率低的一个重要原因是查询所需信息没有缓存在内存中，这可能是由于内存资源紧张，缓存信息被其他查询处理覆盖。

重复执行相同的查询语句，如果后续执行的查询语句效率提升，则可能是由于上述原因导致。

----结束

6.18 未收集统计信息导致查询性能差

问题现象

SQL查询性能差，对语句执行EXPLAIN VERBOSE时有Warning信息。

原因分析

查询中涉及到的表或列没有收集统计信息。统计信息是优化器生成执行计划的基础，没有收集统计信息，优化器生成的执行计划会非常差，如果统计信息未收集，会导致多种多样表现形式的性能问题。例如，等值关联走NestLoop，大表broadcast，集群CPU持续增高等问题。

分析过程

步骤1 通过EXPLAIN VERBOSE/EXPLAIN PERFORMANCE打印语句的执行计划。

执行计划中会有语句未收集统计信息的告警，并且通常E-rows估算非常小。

```
tpcdslxcpm=#  explain verbose
tpcdslxcpm=#  select sr_customer_sk as ctr_customer_sk
tpcdslxcpm=#  ,sr_store_sk as ctr_store_sk
tpcdslxcpm=#  ,sum(SR_FEE) as ctr_total_return
tpcdslxcpm=#  from store_returns_1
tpcdslxcpm=#  ,date_dim_1
tpcdslxcpm=#  where sr_returned_date_sk = d_date_sk
tpcdslxcpm=#  and d_year =2000
tpcdslxcpm=#  group by sr_customer_sk
tpcdslxcpm=#  ,sr_store_sk;
WARNING:  Statistics in some tables or columns(public.store_returns_1.sr_returned_date_sk, public.store_returns_1.sr_customer_sk, public.store_returns_1.sr_store_sk, publi
c.date_dim_1.d_date_sk, public.date_dim_1.d_year) are not collected.
HINT:  Do analyze for them in order to generate optimized plan.
 id |                       operation                       | E-rows | E-distinct | E-width | E-costs
----+-------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Streaming (type: GATHER)                          |      4 |            |      54 | 42.64
  2 |    ->  HashAggregate                                  |      4 |            |      54 | 32.64
  3 |       ->  Streaming(type: REDISTRIBUTE)               |      4 |            |      22 | 32.62
  4 |          ->  Nested Loop (5,6)                        |      4 |            |      22 | 32.56
  5 |             ->  Seq Scan on public.date_dim_1         |      1 |            |       4 | 16.20
  6 |             ->  Seq Scan on public.store_returns_1    |     40 |            |      26 | 16.16
(6 rows)

                  Predicate Information (identified by plan id)
--------------------------------------------------------------------------------
   4 --Nested Loop (5,6)
         Join Filter: (store_returns_1.sr_returned_date_sk = date_dim_1.d_date_sk)
   5 --Seq Scan on public.date_dim_1
         Filter: (date_dim_1.d_year = 2000)
(4 rows)
```

步骤2 上述例子中，在打印的执行计划中有Warning提示信息，提示有哪些列在这个执行计划中用到了，但是这些列没有统计信息。

在CN的pg_log日志中也有会有类似的Warning信息。同时，E-rows会比实际值小很多。

----结束

处理方法

周期性地运行ANALYZE，或者在对表的大部分内容执行更改操作后立即执行ANALYZE。

6.19 执行计划中有 NestLoop 导致 SQL 语句执行慢

问题现象

某业务场景中SQL语句执行慢，打印执行计划发现存在NestLoop。

分析过程

步骤1 通过EXPLAIN VERBOSE打印语句执行计划，查看执行计划发现SQL语句中存在not in语句。

```
explain verbose
select sr_customer_sk as ctr_customer_sk
,sr_store_sk as ctr_store_sk
,sum(SR_FEE) as ctr_total_return
from store_returns
where sr_returned_date_sk not in ( select d_date_sk from date_dim where d_year = 2000)
group by sr_customer_sk
,sr_store_sk;
```

步骤2 执行计划中存在NestLoop。

```
 id |                                operation                                | E-rows | E-distinct | E-width |  E-costs
----+-------------------------------------------------------------------------+--------+------------+---------+-----------
  1 | ->  Row Adapter                                                         | 218254 |            |      46 | 310182.72
  2 |    ->  Vector Streaming (type: GATHER)                                  | 218254 |            |      46 | 310182.72
  3 |       ->  Vector Hash Aggregate                                         | 218254 |            |      46 | 309792.10
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                       | 218254 |            |      14 | 308837.23
  5 |             ->  Vector Nest Loop Anti Join (6, 8)                       | 218254 |            |      14 | 307372.95
  6 |                ->  Vector Partition Iterator                            | 287514 |            |      18 | 2249.88
  7 |                   ->  Partitioned CStore Scan on public.store_returns   | 287514 |            |      18 | 2249.88
  8 |                ->  Vector Materialize                                   |   1460 |            |       4 | 966.86
  9 |                   ->  Vector Streaming(type: BROADCAST)                 |   1460 |            |       4 | 965.03
 10 |                      ->  Vector Partition Iterator                      |    365 |            |       4 | 928.92
 11 |                         ->  Partitioned CStore Scan on public.date_dim  |    365 |            |       4 | 928.92
(11 rows)

                                    Predicate Information (identified by plan id)
--------------------------------------------------------------------------------------------------------------------------------
   5 --Vector Nest Loop Anti Join (6, 8)
       Join Filter: ((store_returns.sr_returned_date_sk = date_dim.d_date_sk) OR (store_returns.sr_returned_date_sk IS NULL))
   6 --Vector Partition Iterator
       Iterations: 7
   7 --Partitioned CStore Scan on public.store_returns
       Selected Partitions:  1..7
  10 --Vector Partition Iterator
       Iterations: 1
  11 --Partitioned CStore Scan on public.date_dim
       Filter: (date_dim.d_year = 2000)
       Selected Partitions:  3
(11 rows)
```

----结束

分析结果

- NestLoop是导致语句性能慢的主要原因。
- 由于NOT IN对于NULL值的特殊处理，导致语句无法使用高效的HashJoin进行高效处理，性能较差。

处理方法

步骤1 若业务场景中用户不关注NULL值的处理，或者数据中根本不存在NULL值，则可以通过等价改写将NOT IN改写为NOT EXISTS来进行优化。

```
tpcds1xcpm=#  explain verbose
tpcds1xcpm-#  select sr_customer_sk as ctr_customer_sk
tpcds1xcpm-#  ,sr_store_sk as ctr_store_sk
tpcds1xcpm-#  ,sum(SR_FEE) as ctr_total_return
tpcds1xcpm-#  from store_returns
tpcds1xcpm-#  where not exists ( select 1 from date_dim where d_date_sk =sr_returned_date_sk and  d_year = 2000)
tpcds1xcpm-#  group by sr_customer_sk
tpcds1xcpm-#  ,sr_store_sk;
 id |                              operation                              | E-rows | E-distinct | E-width | E-costs
----+---------------------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Row Adapter                                                     | 239700 |            |      46 | 7068.30
  2 |    ->  Vector Streaming (type: GATHER)                              | 239700 |            |      46 | 7068.30
  3 |       ->  Vector Hash Aggregate                                     | 239700 |            |      46 | 6677.68
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                   | 239701 |            |      14 | 5628.99
  5 |             ->  Vector Hash Anti Join (6, 8)                        | 239701 |            |      14 | 4020.85
  6 |                ->  Vector Partition Iterator                        | 287514 |       1990 |      18 | 2249.88
  7 |                   ->  Partitioned CStore Scan on public.store_returns | 287514 |            |      18 | 2249.88
  8 |                ->  Vector Streaming(type: BROADCAST)                |   1460 |        365 |       4 | 965.03
  9 |                   ->  Vector Partition Iterator                     |    365 |            |       4 | 928.92
 10 |                      ->  Partitioned CStore Scan on public.date_dim |    365 |            |       4 | 928.92
(10 rows)

                Predicate Information (identified by plan id)
--------------------------------------------------------------------------------
   5 --Vector Hash Anti Join (6, 8)
       Hash Cond: (store_returns.sr_returned_date_sk = date_dim.d_date_sk)
   6 --Vector Partition Iterator
       Iterations: 7
   7 --Partitioned CStore Scan on public.store_returns
       Selected Partitions:  1..7
   9 --Vector Partition Iterator
       Iterations: 1
  10 --Partitioned CStore Scan on public.date_dim
       Filter: (date_dim.d_year = 2000)
       Selected Partitions:  3
(11 rows)
```


----结束

6.20 未分区剪枝导致 SQL 查询慢

问题现象

SQL语句查询慢，查询的分区表总共185亿条数据，查询条件中没有涉及分区键。

```
SELECT passtime FROM table where passtime<'2020-02-19 15:28:14' and passtime>'2020-02-18 15:28:37' order by passtime desc limit 10;
SELECT max(passtime) FROM table where passtime<'2020-02-19 15:28:14' and passtime>'2020-02-18 15:28:37';
```

列存表，分区键为createtime，哈希分布键为motorvehicleid。

原因分析

慢SQL过滤条件中未涉及分区字段，导致执行计划未分区剪枝，进行了全表扫描，性能严重劣化。

分析过程

步骤1 和用户确认部分业务慢，执行慢的业务中都涉及到了同一张表tb_motor_vehicle。

步骤2 收集几个典型的慢SQL语句，分别打印执行计划。从执行计划中可以看出来，两条SQL的耗时都集中在Partitioned CStore Scan on public.tb_motor_vehicle列存表的分区扫描上。


```
 id |                          operation                          |        A-time         |  A-rows  | E-rows | E-distinct |  Peak Memory   | E-memory
----+-------------------------------------------------------------+-----------------------+----------+--------+------------+----------------+----------
  1 | ->  Row Adapter                                             | 658657.403            |        1 |      1 |            | 10KB           |
  2 |   ->  Vector Aggregate                                      | 658657.395            |        1 |      1 |            | 176KB          |
  3 |     ->  Vector Streaming (type: GATHER)                     | 658657.203            |       24 |     24 |            | 157KB          |
  4 |       ->  Vector Aggregate                                  | [293523.439,657533.737] |       24 |     24 |            | [176KB, 176KB] | 1MB
  5 |         ->  Vector Partition Iterator                       | [292832.499,656876.182] | 58042877 |    864 |            | [17KB, 17KB]   | 1MB
  6 |           ->  Partitioned CStore Scan on public.tb_motor_vehicle | [290866.340,652888.555] | 58042877 |    864 |            | [1MB, 1MB]     | 1MB
(6 rows)

                                Predicate Information (identified by plan id)
--------------------------------------------------------------------------------------------------------------------------------------------------------------
  5 --Vector Partition Iterator
      Iterations: 398
  6 --Partitioned CStore Scan on public.tb_motor_vehicle
      Filter: ((tb_motor_vehicle.passtime < '2020-02-19 15:28:14'::timestamp without time zone) AND (tb_motor_vehicle.passtime > '2020-02-18 15:28:37'::timestam
      Rows Removed by Filter: 1638714321
      Selected Partitions:  1..398
(6 rows)
```

```
 iyd |                          operation                          |        A-time         |  A-rows  | E-rows | E-distinct |  Peak Memory   | E-memory
-----+-------------------------------------------------------------+-----------------------+----------+--------+------------+----------------+----------
  1 | ->  Row Adapter                                             | 712811.232            |       10 |     10 |            | 10KB           |
  2 |   ->  Vector Limit                                          | 712811.206            |       10 |     10 |            | 1KB            |
  3 |     ->  Vector Streaming (type: GATHER)                     | 712811.200            |      240 |    240 |            | 672KB          |
  4 |       ->  Vector Limit                                      | [365578.263,712268.299] |      240 |    240 |            | [1KB, 1KB]     | 1MB
  5 |         ->  Vector Sort                                     | [365578.257,712268.294] |      240 |    864 |            | [254KB, 254KB] | 16MB
  6 |           ->  Vector Partition Iterator                     | [365216.092,711913.426] | 58042877 |    864 |            | [17KB, 17KB]   | 1MB
  7 |             ->  Partitioned CStore Scan on public.tb_motor_vehicle | [360943.136,701785.728] | 58042877 |    864 |            | [1MB, 1MB]     | 1MB
(7 rows)

                                Predicate Information (identified by plan id)
--------------------------------------------------------------------------------------------------------------------------------------------------------------
  6 --Vector Partition Iterator
      Iterations: 398
  7 --Partitioned CStore Scan on public.tb_motor_vehicle
      Filter: ((tb_motor_vehicle.passtime < '2020-02-19 15:28:14'::timestamp without time zone) AND (tb_motor_vehicle.passtime > '2020-02-18 15:28:37'::timestamp w
      Rows Removed by Filter: 1638678418
      Selected Partitions:  1..398
(6 rows)
```

步骤3 已确认该表的分区键为createtime，而涉及的SQL中无任何createtime的筛选和过滤条件，基本可以确认是由于慢SQL的计划没有走分区剪枝，导致了全表扫描，对于185亿条数据量的表，全表扫描性能会很差。

----结束

处理方法

在慢SQL的过滤条件中增加分区筛选条件，避免走全表扫描。

优化后的SQL和执行计划如下，性能从十几分钟，优化到了12秒左右，性能有明显提升。

```
SELECT passtime FROM tb_motor_vehicle WHERE createtime > '2020-02-19 00:00:00' AND createtime < '2020-02-20 00:00:00' AND passtime > '2020-02-19 00:00:00' AND passtime < '2020-02-20 00:00:00' ORDER BY passtime DESC LIMIT 10000;
```

```
explain performance SELECT passtime FROM tb_motor_vehicle WHERE createtime > '2020-02-19 00:00:00' AND createtime < '2020-02-20 00:00:00' AND passtime > '2020-02-19 00
ORDER BY passtime DESC LIMIT 10000;
+----+-------------------------------------------------------------+-------------------------+----------+----------+------------+----------------+----------+
| id | operation                                                   | A-time                  | A-rows   | E-rows   | E-distinct | Peak Memory    | E-memory |
+----+-------------------------------------------------------------+-------------------------+----------+----------+------------+----------------+----------+
|  1 | ->  Row Adapter                                             | 12285.727               |    10000 |    10000 | NULL       | 10KB           |          |
|  2 |   ->  Vector Limit                                          | 12284.854               |    10000 |    10000 | NULL       | 1KB            |          |
|  3 |     ->  Vector Streaming (type: GATHER)                     | 12284.840               |    10000 |   240000 | NULL       | 1464KB         |          |
|  4 |       ->  Vector Limit                                      | [10300.383,12227.410]   |   240000 |   240000 | NULL       | [1KB, 1KB]     | 1MB      |
|  5 |         ->  Vector Sort                                     | [10300.368,12227.399]   |   240000 | 11542800 | NULL       | [962KB, 962KB] | 16MB     |
|  6 |           ->  Vector Partition Iterator                     | [8461.684,10419.017]    | 57415974 | 11542791 | NULL       | [25KB, 25KB]   | 1MB      |
|  7 |             ->  Partitioned CStore Scan on public.tb_motor_vehicle | [8389.677,10356.890]    | 57415974 | 11542791 | NULL       | [1MB, 1MB]     | 1MB      |
+----+-------------------------------------------------------------+-------------------------+----------+----------+------------+----------------+----------+
7 rows in set
```

6.21 行数估算过小，优化器选择走 NestLoop 导致性能下降

问题现象

查询语句执行慢，卡住无法返回结果。SQL语句的特点是2~3张表left join，然后通过SELECT查询结果，执行计划如下：


```
id |                                                                operation                                                                | E-rows  | E-distinct | E-memory | E-width |  E-costs
---+-----------------------------------------------------------------------------------------------------------------------------------------+---------+------------+----------+---------+-----------
 1 | ->  Row Adapter                                                                                                                         |       2 |            |          |     771 | 116895.08
 2 |   ->  Vector Streaming (type: GATHER)                                                                                                   |       2 |            |          |     771 | 116895.08
 3 |     -> Vector Nest Loop Left Join (4, 9)                                                                                                |       2 |            | 1MB      |     771 | 91866.92
 4 |       ->  Vector Streaming(type: LOCAL GATHER dop: 1/8)                                                                                 |       1 |            | 16MB     |     118 | 89602.67
 5 |         ->  Vector Hash Aggregate                                                                                                       |       1 |            | 16MB     |      94 | 89602.63
 6 |           ->  Vector Streaming(type: LOCAL REDISTRIBUTE dop: 8/8)                                                                       |       1 |            | 17MB     |      22 | 89602.61
 7 |             ->  Vector Partition Iterator                                                                                               |       1 |            | 1MB      |      22 | 89602.58
 8 |               ->  Partitioned CStore Scan on scm_sdrplus.t58_pppoe_h a                                                                  |       1 |            | 1MB      |      22 | 89602.57
 9 |       ->  Vector Hash Left Join (10, 22)                                                                                                |       2 | 200, 77    | 16MB     |     664 | 2264.24
10 |         -> Vector Nest Loop Left Join (11, 21)                                                                                          |       2 |            | 1MB      |     654 | 2186.18
11 |           ->  Vector Hash Right Join (12, 13)                                                                                           |       2 | 7220, 2256 | 16MB     |     643 | 1987.21
12 |             ->  CStore Scan on scm_sdrplus.ne_location 1                                                                                | 43320   |            | 1MB      |     115 | 1735.22
13 |             ->  Vector Partition Iterator                                                                                               |       2 |            | 1MB      |     555 | 231.10
14 |               ->  Partitioned CStore Index Scan using user_database_account_idx on scm_sdrplus.user_database a                          |       2 |            | 16MB     |     555 | 231.10
15 |       ->  Row Adapter  [14, InitPlan 1 (returns $1)]                                                                                   |       6 |            | 1MB      |      43 | 25026.85
16 |         ->  Vector Aggregate                                                                                                            |       6 |            | 1MB      |      43 | 25026.85
17 |           ->  Vector Streaming(type: BROADCAST)                                                                                         |      36 |            | 2MB      |      43 | 25026.85
18 |             ->  Vector Aggregate                                                                                                        |       6 |            | 1MB      |      43 | 25026.72
19 |               ->  Vector Partition Iterator                                                                                             | 3330361 |            | 1MB      |      11 | 23639.06
20 |                 ->  Partitioned CStore Scan on scm_sdrplus.user_database a                                                              | 3330361 |            | 1MB      |      11 | 23639.06
21 |           ->  CStore Index Scan using i_mac_oui on scm_sdrplus.mac_oui c                                                                |       6 |            | 1MB      |      21 | 198.96
22 |         ->  CStore Scan on scm_sdrplus.thai_province m                                                                                  |     462 |            | 1MB      |      10 | 77.08
```

原因分析

优化器在选择执行计划时，对结果集评估较小，导致执行计划走了NestLoop，性能下降。

分析过程

步骤1 排查当前的I/O、内存、CPU使用情况，没有发现资源占用高的情况。

步骤2 查看慢SQL的线程等待状态。

根据线程等待状态，并没有出现都在等待某个DN的情况，初步排除中间结果集偏斜到了同一个DN的情况。

```
SELECT * FROM pg_thread_wait_status WHERE query_id='149181737656737395';
```

```
  node_name   | db_name  | thread_name |      query_id       |       tid       | lwtid | ptid  | tlevel | smpid |          wait_status          | wait_event
--------------+----------+-------------+---------------------+-----------------+-------+-------+--------+-------+-------------------------------+------------
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140346588657408 | 34622 | 21227 |     17 |     0 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345923331840 | 34624 | 21227 |      4 |     0 | flush data: dn_6001_6002(0)   |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140346705573632 | 34626 | 21227 |      4 |     1 | flush data: dn_6001_6002(0)   |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345734571776 | 34630 | 21227 |      4 |     2 | flush data: dn_6001_6002(0)   |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345708902144 | 34631 | 21227 |      4 |     3 | flush data: dn_6001_6002(0)   |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345664861952 | 34634 | 21227 |      4 |     5 | flush data: dn_6001_6002(0)   |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345690023680 | 34633 | 21227 |      4 |     4 | flush data: dn_6001_6002(0)   |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345645983488 | 34636 | 21227 |      4 |     6 | flush data: dn_6001_6002(0)   |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345620297472 | 34637 | 21227 |      4 |     7 | flush data: dn_6001_6002(0)   |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345593034496 | 34640 | 34624 |      6 |     0 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140345140049664 | 34641 | 34624 |      6 |     1 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140344882620160 | 34643 | 34624 |      6 |     2 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140342933812992 | 34645 | 34624 |      6 |     3 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140342917031680 | 34646 | 34624 |      6 |     4 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140342900250368 | 34648 | 34624 |      6 |     5 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140342883469056 | 34649 | 34624 |      6 |     6 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140342711473920 | 34652 | 34624 |      6 |     7 | synchronize quit              |
 dn_6001_6002 | db_sdrplus | cn_5002     | 149181737656737395 | 140342430512896 | 21227 |       |      0 |     0 | none                          |
 cn_5002      | db_sdrplus | gsql        | 149181737656737395 | 140085516814080 | 21777 |       |      0 |     0 | wait node: dn_6011_6012, total 6 |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139965483169536 | 34621 |       |      0 |     0 | none                          |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139965415552768 | 34623 | 34621 |     17 |     0 | synchronize quit              |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139978527995648 | 34625 | 34621 |      4 |     0 | flush data: dn_6003_6004(0)   |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139965444912896 | 34627 | 34621 |      4 |     1 | flush data: dn_6003_6004(0)   |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139965396674304 | 34628 | 34621 |      4 |     2 | flush data: dn_6003_6004(0)   |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139965370988288 | 34629 | 34621 |      4 |     3 | flush data: dn_6003_6004(0)   |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139964910663424 | 34632 | 34621 |      4 |     4 | flush data: dn_6003_6004(0)   |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139964893882112 | 34635 | 34621 |      4 |     5 | flush data: dn_6003_6004(0)   |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139964644316928 | 34638 | 34621 |      4 |     6 | flush data: dn_6003_6004(0)   |
 dn_6003_6004 | db_sdrplus | cn_5002     | 149181737656737395 | 139964627535616 | 34639 | 34621 |      4 |     7 | flush data: dn_6003_6004(0)   |
```

步骤3 联系运维人员登录到相应的实例节点上，打印等待状态为none的线程堆栈信息如下。

通过反复打印堆栈信息，发现堆栈在变化，并没有hang死，所以初步判断该问题为性能慢的问题。另堆栈中有VecNestLoopRuntime，结合执行计划，初步判断是由于统计信息不准，优化器评估结果集较少，执行计划使用了NestLoop导致性能下降。

gstack *14104*


```
Thread 1 (Thread 0x7f4fa20ff700 (LWP 14104)):
#0  0x000055b3faed6b37 in heap_hot_search_buffer(ItemPointerData*, RelationData*, int, SnapshotData*, HeapTupleData*, HeapTupleHeaderData*, bool*, bool) ()
#1  0x000055b3faef0890 in index_fetch_heap(IndexScanDescData*) ()
#2  0x000055b3faef0b21 in index_getnext(IndexScanDescData*, ScanDirection) ()
#3  0x000055b3faeefeb3 in systable_getnext_ordered(SysScanDescData*, ScanDirection) ()
#4  0x000055b3fadef4b1 in CStore::LoadCUDesc(int, LoadCUDescCtl*, bool, SnapshotData*) ()
#5  0x000055b3fadf4728 in CStore::LoadCUDescIfNeed() ()
#6  0x000055b3fadf4fbb in CStore::CStoreScan(CStoreScanState*, VectorBatch*) ()
#7  0x000055b3fb22e569 in ExecCStoreScan(CStoreScanState*) ()
#8  0x000055b3fb230468 in VectorBatch* ExecCstoreIndexScanT<(IndexType)2>(CStoreIndexScanState*) ()
#9  0x000055b3fb23080a in ExecCstoreIndexScan(CStoreIndexScanState*) ()
#10 0x000055b3fb2707b4 in VectorEngine(PlanState*) ()
#11 0x000055b3fb241d11 in VectorBatch* VecNestLoopRuntime::JoinQualT<(JoinType)1, true, false, false>() ()
#12 0x000055b3fb245bc1 in VectorBatch* VecNestLoopRuntime::JoinT<true>() ()
#13 0x000055b3fb2707b4 in VectorEngine(PlanState*) ()
#14 0x000055b3fb2190c8 in HashJoinTbl::probeHashTable(hashSource*) ()
#15 0x000055b3fb2182e5 in ExecVecHashJoin(VecHashJoinState*) ()
#16 0x000055b3fb2707b4 in VectorEngine(PlanState*) ()
#17 0x000055b3fb241d11 in VectorBatch* VecNestLoopRuntime::JoinQualT<(JoinType)1, true, false, false>() ()
#18 0x000055b3fb245bc1 in VectorBatch* VecNestLoopRuntime::JoinT<true>() ()
#19 0x000055b3fb2707b4 in VectorEngine(PlanState*) ()
#20 0x000055b3fb1562a5 in standard_ExecutorRun(QueryDesc*, ScanDirection, long) ()
#21 0x000055b3fb156a9d in ExecutorRun(QueryDesc*, ScanDirection, long) ()
#22 0x000055b3fb6fc0e8 in PortalRunSelect(PortalData*, bool, long, _DestReceiver*) ()
#23 0x000055b3fb6fc860 in PortalRun(PortalData*, long, bool, _DestReceiver*, _DestReceiver*, char*) ()
#24 0x000055b3fb6ededa in exec_simple_plan(PlannedStmt*) ()
#25 0x000055b3fb6f45c0 in PostgresMain(int, char**, char const*, char const*) ()
```

```
#0  0x000055b3fadef4b1 in CStore::LoadCUDesc(int, LoadCUDescCtl*, bool, SnapshotData*) ()
#1  0x000055b3fadf4728 in CStore::LoadCUDescIfNeed() ()
#2  0x000055b3fadf4fbb in CStore::CStoreScan(CStoreScanState*, VectorBatch*) ()
#3  0x000055b3fb22e569 in ExecCStoreScan(CStoreScanState*) ()
#4  0x000055b3fb230468 in VectorBatch* ExecCstoreIndexScanT<(IndexType)2>(CStoreIndexScanState*) ()
#5  0x000055b3fb23080a in ExecCstoreIndexScan(CStoreIndexScanState*) ()
#6  0x000055b3fb2707b4 in VectorEngine(PlanState*) ()
#7  0x000055b3fb241d11 in VectorBatch* VecNestLoopRuntime::JoinQualT<(JoinType)1, true, false, false>() ()
#8  0x000055b3fb245bc1 in VectorBatch* VecNestLoopRuntime::JoinT<true>() ()
#9  0x000055b3fb2707b4 in VectorEngine(PlanState*) ()
#10 0x000055b3fb2190c8 in HashJoinTbl::probeHashTable(hashSource*) ()
#11 0x000055b3fb2182e5 in ExecVecHashJoin(VecHashJoinState*) ()
#12 0x000055b3fb2707b4 in VectorEngine(PlanState*) ()
#13 0x000055b3fb241d11 in VectorBatch* VecNestLoopRuntime::JoinQualT<(JoinType)1, true, false, false>() ()
#14 0x000055b3fb245bc1 in VectorBatch* VecNestLoopRuntime::JoinT<true>() ()
#15 0x000055b3fb2707b4 in VectorEngine(PlanState*) ()
#16 0x000055b3fb1562a5 in standard_ExecutorRun(QueryDesc*, ScanDirection, long) ()
#17 0x000055b3fb156a9d in ExecutorRun(QueryDesc*, ScanDirection, long) ()
#18 0x000055b3fb6fc0e8 in PortalRunSelect(PortalData*, bool, long, _DestReceiver*) ()
#19 0x000055b3fb6fc860 in PortalRun(PortalData*, long, bool, _DestReceiver*, _DestReceiver*, char*) ()
#20 0x000055b3fb6ededa in exec_simple_plan(PlannedStmt*) ()
#21 0x000055b3fb6f45c0 in PostgresMain(int, char**, char const*, char const*) ()
#22 0x000055b3fb59593e in SubPostmasterMain(int, char**) ()
```

步骤4 对表执行ANALYZE后性能并没有太大改善。

步骤5 对SQL增加hint关闭索引，让优化器强行使用hashjoin，发现hint功能没有生效，原因是hint无法改变子查询中的计划。

步骤6 通过SET enable_indexscan = off，执行计划被改变，使用了Hash Left Join，慢SQL在3秒左右返回结果，查询性能恢复。

id	operation	A-time	A-rows	E-rows	Peak Memory	E-memory	A-width	E-width	E-costs
1	-> Row Adapter	3464.656	357053	6	260KB			768	154118.20
2	-> Vector Streaming (type: GATHER)	3376.601	357053	6	1503KB			768	154118.20
3	-> Vector Streaming(type: LOCAL GATHER dop: 1/8)	[3271.144,3381.566]	357053	6	[789KB,789KB]	16MB		768	129090.04
4	-> Vector Hash Right Join (5, 20)	[3207.552,3371.885]	357053	2	[2MB,2MB]	16MB		768	129089.89
5	-> Vector Streaming(type: LOCAL REDISTRIBUTE dop: 8/1)	[544.447,606.631]	1110265	1110155	[809KB,825KB]	2MB		664	39371.57
6	-> Vector Hash Left Join (7, 19)	[951.633,1195.812]	1110265	1110155	[4MB,4MB]	16MB		664	24171.42
7	-> Vector Hash Left Join (8, 16)	[250.763,270.695]	1110265	1110120	[3MB,3MB]	16MB		653	15628.64
8	-> Vector Partition Iterator	[60.531,69.685]	1110265	1110120	[17KB,17KB]	1MB		555	9256.68
9	-> Partitioned CStore Scan on user_database t2	[58.111,67.092]	1110265	1110120	[3MB,3MB]	1MB		555	9256.68
10	-> Row Adapter [9, InitPlan 1 (returns $1)]	[0.178,0.212]	6	6	[10KB,10KB]	1MB		43	25026.85
11	-> Vector Aggregate	[0.168,0.202]	6	6	[176KB,176KB]	1MB		43	25026.85
12	-> Vector Streaming(type: BROADCAST)	[0.129,0.165]	36	36	[96KB,96KB]	2MB		43	25026.85
13	-> Vector Aggregate	[35.907,70.240]	6	6	[177KB,177KB]	1MB		43	25026.72
14	-> Vector Partition Iterator	[16.292,31.784]	3330361	3330361	[17KB,17KB]	1MB		11	23639.06
15	-> Partitioned CStore Scan on user_database	[13.908,27.591]	3330361	3330361	[580KB,580KB]	1MB		11	23639.06
16	-> Vector Hash Left Join (17, 18)	[115.823,125.157]	43320	43320	[418KB,418KB]	16MB	[304,304]	125	1912.42
17	-> CStore Scan on ne_location 1	[103.422,111.513]	43320	43320	[1MB,1MB]	1MB		115	1735.22
18	-> CStore Scan on thai_province m	[0.385,0.731]	462	462	[219KB,219KB]	1MB	[26,26]	10	77.08
19	-> CStore Scan on mac_oui c	[515.285,755.601]	188418	188418	[478KB,478KB]	1MB	[53,53]	21	1151.40
20	-> Vector Hash Aggregate	[2619.810,2781.678]	357053	1	[2MB,2MB]	16MB	[133,134]	91	89602.63
21	-> Vector Streaming(type: LOCAL REDISTRIBUTE dop: 8/8)	[2338.077,2505.698]	39697152	1	[155KB,155KB]	17MB		19	89602.61
22	-> Vector Partition Iterator	[1633.245,2331.055]	39697152	1	[25KB,25KB]	1MB		19	89602.58
23	-> Partitioned CStore Scan on t58_pppoe_h t1	[1629.094,2325.774]	39697152	1	[1MB,1MB]	1MB			

(23 rows)

----结束

处理方法

通过**set enable_indexscan = off**关闭索引功能，让优化器生成的执行计划不走NestLoop，而走Hashjoin。

6.22 语句中存在“in 常量”导致 SQL 执行无结果

问题现象

简单的大表过滤的SQL语句中有一个“in 常量”的过滤条件，常量的个数非常多(约有2000多个)，基表数据量比较大，SQL语句执行无结果。

原因分析

执行计划中，in条件还是作为普通的过滤条件存在。这种场景下，join操作的性能优于in条件，最优的执行计划应该是将“in 常量”转化为join操作。

分析过程

步骤1 打印语句的执行计划：

```
QUERY PLAN
 id |                                operation                                |  E-rows  | E-distinct | E-memory | E-width |  E-c
----+-------------------------------------------------------------------------+----------+------------+----------+---------+------
  1 | ->  Row Adapter                                                         | 14586792 |            |          |      44 | 26121
  2 |   ->  Vector Streaming (type: GATHER)                                   | 14586792 |            |          |      44 | 26121
  3 |     ->  Vector Streaming(type: LOCAL GATHER dop: 1/24)                  | 14586792 |            |   48MB   |      44 | 26073
  4 |       ->  Vector Partition Iterator                                     | 14586798 |            |   1MB    |      44 | 26073
  5 |         ->  Partitioned CStore Scan on public.tb_motor_vehicle          | 14586798 |            |   1MB    |      44 | 26073
(5 rows)
```

步骤2 执行计划中，in条件还是作为普通的过滤条件存在。这种场景下，join操作的性能优于in条件，最优的执行计划应该是将“in 常量”转化为join操作。

----结束

处理方法

步骤1 GUC参数**qrw_inlist2join_optmode**可以控制把“in 常量”转join的行为，默认是cost_base的。如果优化器估算不准，可能会出现需要转化的场景没有做转化，导致性能较差。

步骤2 这种情况下可以通过设置qrw_inlist2join_optmode为rule_base来规避解决。

```
set qrw_inlist2join_optmode to rule_base;
```

----结束

6.23 单表点查询性能差

问题现象

单表查询的场景下，预期1s以内返回结果，实际执行耗时超过10s。

原因分析

行列存表选择错误导致的问题，点查询场景应该使用行存表+btree索引。

分析过程

步骤1 通过抓取问题SQL的执行信息，发现大部分的耗时都在“CStore Scan”。

```
 id |                          operation                          |        A-time        | A-rows | E-rows | Peak Memory | A-width | E-width | E-costs
----+-------------------------------------------------------------+----------------------+--------+--------+-------------+---------+---------+---------
  1 | ->  Row Adapter                                             | 15168.721            |      1 |      1 | 37KB        |         |      88 | 9662341.28
  2 |   ->  Vector Limit                                          | 15168.715            |      1 |      1 | 2KB         |         |      88 | 9662341.28
  3 |     ->  Vector Aggregate                                    | 15168.711            |      1 |      1 | 629KB       |         |      88 | 9662341.28
  4 |       ->  Vector Streaming (type: GATHER)                   | 15168.140            |     48 |      1 | 319KB       |         |      88 | 9662341.28
  5 |         ->  Vector Aggregate                                | [1769.545,14168.103] |     48 |      1 | [535KB,535KB] |       |      88 | 9662339.75
  6 |           ->  Vector Partition Iterator                     | [1769.516,14150.029] |      0 |      1 | [17KB,17KB] |         |       8 | 9662339.72
  7 |             ->  Partitioned CStore Scan on sym_saacntxn     | [1769.419,14129.592] |      0 |      1 | [2MB,3MB]   |         |       8 | 9662339.72
(7 rows)
```

步骤2 分析出问题的场景：基表是一张十亿级别的表，每晚有批量增量数据入库，同时会有少量的数据清洗的工作。白天会有高并发的查询操作，查询不涉及表关联，并且返回结果都不大。

----结束

处理方法

调整表定义，将表修改为行存表，同时建立btree索引，索引建立的原则：

1. 在经常需要搜索查询的列上创建索引，可以加快搜索的速度。
2. 不要定义冗余或重复的索引。
3. 建立组合索引时候，要把过滤性比较好的列往前放。
4. 为经常出现在关键字ORDER BY、GROUP BY、DISTINCT后面的字段建立索引。
5. 在经常使用WHERE子句的列上创建索引，加快条件的判断速度。

6.24 动态负载管理下的 CCN 排队

问题现象

业务整体缓慢，只有少量语句在执行，其余业务语句都在排队中（wait in ccn queue）。

原因分析

动态负载管理下，语句会根据估算内存计数排序，例如，最大动态可用内存为10GB（单实例），语句估算使用内存大小为5GB，这样的语句运行两个，其余语句就会等待前两个语句运行完毕才能执行，此时的状态即为wait in ccn queue。

处理方法

- 场景一：语句估算内存过大，造成排队。
  - 查询pg_session_wlmstat视图，查看状态为running的语句是否个数很少，而且statement_mem字段数值是否较大（单位为MB，一般认为大于max_dynamic_memory 1/3即为大内存语句）。如果都符合就可以判断是此类语句占据内存导致整体运行缓慢。
    ```
    SELECT usename,substr(query,0,20),threadid,status,statement_mem FROM pg_session_wlmstat where usename not in ('omm','Ruby') order by statement_mem,status desc;
    ```



```
usename |        substr        |    threadid     | status  | statement_mem
--------+----------------------+-----------------+---------+---------------
 dzx    | explain /*Q18*/ perf | 140635882325760 | running |          1288
 dzx    | explain /*Q18*/ perf | 140635599181568 | running |          1288
 dzx    | explain /*Q18*/ perf | 140635978802944 | pending |          1288
 dzx    | explain /*Q18*/ perf | 140635683088128 | pending |          1288
 dzx    | explain /*Q18*/ perf | 140635632744192 | pending |          1288
 dzx    | explain /*Q18*/ perf | 140635615962880 | pending |          1288
 dzx    | explain /*Q18*/ perf | 140635649525504 | pending |          1288
 dzx    | explain /*Q18*/ perf | 140635808921344 | pending |          1288
 dzx    | explain /*Q18*/ perf | 140635582400256 | pending |          1288
 dzx    | explain /*Q18*/ perf | 140635666306816 | pending |          1288
(10 rows)
```

如上图所示，只有最后一个语句是running状态，其余语句都是pending状态。根据statement_mem可以看到该语句占据2576MB内存。此时根据语句的threadid，执行以下命令终止对应的查询语句，终止后即可释放资源，其余语句正常运行。

```
SELECT pg_terminate_backend(threadid);
```

- 场景二：所有语句状态都是pending状态，没有运行的语句。此时应是管控机制出现异常，直接终止所有线程，即可恢复正常。

```
SELECT pg_terminate_backend(pid) FROM pg_stat_activity where usename not in ('dbadmin','Ruby');
```

6.25 数据膨胀磁盘空间不足，导致性能降低

问题现象

用户数据膨胀严重，磁盘空间不足，性能低。

原因分析

用户可在管控面执行全库Vacuum/Vacuum Full，以定期进行空间回收：

- 用户频繁创建、删除表，导致系统表膨胀严重，需要对系统表执行Vacuum。
- 用户频繁执行UPDATE、DELETE语句，导致用户表膨胀严重，需要对用户表执行Vacuum/Vacuum Full。

说明

仅8.1.3及以上集群版本支持。

处理方法

步骤1 登录GaussDB(DWS) 管理控制台。

步骤2 在集群列表中单击指定集群名称。

步骤3 进入“集群详情”页面，切换至“智能运维”页签。

步骤4 在运维详情部分切换至运维计划模块。单击“添加运维任务”按钮。

步骤5 弹出添加运维任务边栏，

- 运维任务选择“Vacuum”。
- 调度模式选择“自动”，DWS将自动扫描Vacuum目标。
- Vacuum目标选择系统表或用户表：
  - 如果用户业务UPDATE、DELETE较多，选择用户表。
  - 如果创建表、删除表较多，选择系统表。

步骤6 单击“下一步：定时配置”，配置Vacuum类型，推荐选择“周期型任务”，GaussDB(DWS)将自动在自定义时间窗内执行Vacuum。

说明

对于自动Vacuum运维任务，系统对于用户表的处理方法实际采用的是VACUUM FULL操作。VACUUM FULL执行过程中，本身持有8级锁，会阻塞其他业务，导致锁冲突产生，业务本身会陷入锁等待，20分钟后超时报错。因此，在用户配置时间窗内，应尽量避开执行所有处理表的相关业务。

步骤7 确认无误后，单击“下一步：配置确认”，完成配置。

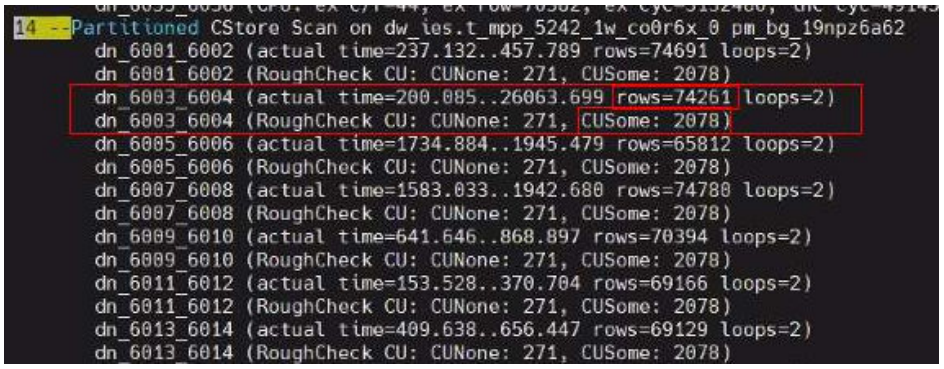
----结束

6.26 列存小 CU 多导致的性能慢问题

实际业务场景中，用户会大量使用列存表，但是列存表使用不当会造严重的性能问题，最常见的就是列存小CU过多导致的性能慢问题。

问题现象

1. 系统I/O长期飙升过高，查询偶发性变慢。
2. 查看偶发慢业务慢时的执行计划信息，慢在cstore scan，且扫描数据量不大但扫描CU个数较多。



如图，一个CU能够存放6W条记录，而计划中7W记录需要扫描2000+ CU，说明当前可能存在小CU较多的情况 。

排查方法

查看相关表CU中数据分布情况，以下操作在DN执行。

1. 查看列存表对应的cudesc表

   针对非分区表：

   ```
   SELECT 'cstore.'||relname FROM pg_class where oid = (SELECT relcudescrelid FROM pg_class c inner join pg_namespace n on c.relnamespace = n.oid where relname = 'table name' and nspname = 'schema name');
   ```

   针对分区表：

   ```
   SELECT 'cstore.'||relname FROM pg_class where oid in (SELECT p.relcudescrelid FROM pg_partition p,pg_class c,pg_namespace n where c.relnamespace = n.oid and p.parentid = c.oid and c.relname = 'table name' and n.nspname = 'schema name' and p.relcudescrelid != 0);
   ```

2. 查看cudesc中各CU的rowcount情况

   查询步骤一返回的cudesc表信息，查询结果类似如下，主要关注row_count过小（远小于6w）的CU数量，如果此数量较大，说明当前小CU多，CU膨胀问题严重，影响存储效率和查询访问效率。

col_id	cu_id	min	max	row_count	cu_mode	size	cu_pointer	magic	extra
1	1001	\r	\r	1	3	0		1624011	
2	1001	\r	\r	1	3	0		1624011	
-10	1001			1		1633951767		1624011	
1	1002	\x11	\x11	1	3	0		1624011	
2	1002	\x11	\x11	1	3	0		1624011	
-10	1002			1		1633951767		1624011	
1	1003	\x13	\x13	1	3	0		1624011	
2	1003	\x13	\x13	1	3	0		1624011	
-10	1003			1		1633951767		1624011	
1	1004	2	2	1	3	0		1624011	
2	1004	2	2	1	3	0		1624011	
-10	1004			1		1633951767		1624011	
1	1005	4	4	1	3	0		1624011	
2	1005	4	4	1	3	0		1624011	
-10	1005			1		1633951767		1624011	
1	1006	B	B	1	3	0		1624011	
2	1006	B	B	1	3	0		1624011	
-10	1006			1		1633951767		1624011	
1	1007	D	D	1	3	0		1624011	
2	1007	D	D	1	3	0		1624011	
-10	1007			1		1633951767		1624011	
1	1008	F	F	1	3	0		1624011	
2	1008	F	F	1	3	0		1624011	
-10	1008			1		1633951767		1624011	
1	1009	J	J	1	3	0		1624011	
2	1009	J	J	1	3	0		1624011	
-10	1009			1		1633951767		1624011	
1	1010	M	M	1	3	0		1624011	
2	1010	M	M	1	3	0		1624011	
-10	1010			1		1633951767		1624011	
1	1011	P	P	1	3	0		1624011	
2	1011	P	P	1	3	0		1624011	
-10	1011			1		1633951767		1624011	

触发场景

列存频繁小批量导入，针对含分区且分区个数比较多的场景，小CU问题更加突出。

解决方案

业务侧：

1. 对列存表进行攒批入库，单次入库量（有分区则针对单分区单次入库量）接近或大于6w*主DN个数。
2. 表数据量不大时建议改为行存表。

运维侧：

当业务侧因业务特征无法调整入库量时，定期对列存表进行vacuum full可达到整合小CU的目的，一定程度缓解小CU问题。

6.27 降低 I/O 的处理方案

问题现象

在DWS实际业务场景中因I/O高、I/O瓶颈导致的性能问题较多，其中应用业务设计不合理导致的问题占大多数。本文从应用业务优化角度，以常见触发I/O慢的业务SQL场景为例，指导如何通过优化业务去提升I/O效率和降低I/O。

确定 I/O 瓶颈&识别高 I/O 的语句

步骤1 通过以下内容掌握SQL级I/O问题分析的基础知识。

- PGXC_THREAD_WAIT_STATUS视图功能。
- EXPLAIN功能，需了解并熟悉Scan算子、A-time、A-rows、E- rows，详细内容参见：SQL执行计划。

步骤2 通过pgxc_thread_wait_status视图查看并确定I/O瓶颈。全部状态信息请参见PG_THREAD_WAIT_STATUS。

```
SELECT wait_status,wait_event,count(*) AS cnt FROM pgxc_thread_wait_status WHERE wait_status <> 'wait cmd' AND wait_status <> 'synchronize quit' AND wait_status <> 'none'  GROUP BY 1,2 ORDER BY 3 DESC limit 50;
```

I/O瓶颈时常见的主要状态如下表所示。

表 6-2 I/O 常见状态

Wait status	Wait event
wait io：等待I/O完成。	• BufFileRead：从临时文件中读取数据到指定buffer。 • BufFileWrite：向临时文件中写入指定buffer中的内容。 • DataFileRead：同步读取表数据文件。 • DataFileWrite：向表数据文件写入内容。 • .....

Wait status	Wait event
acquire lwlock：等待获取轻量级锁。	WALWriteLock：用于避免并发WAL写盘。
wait wal sync：等待特定LSN的wal log完成到备机的同步。	NA
wait data sync：等待完成数据页到备机的同步。	NA
Material | Material - write file：当前是Material算子，write file表示Material算子正在将数据写入磁盘。	NA

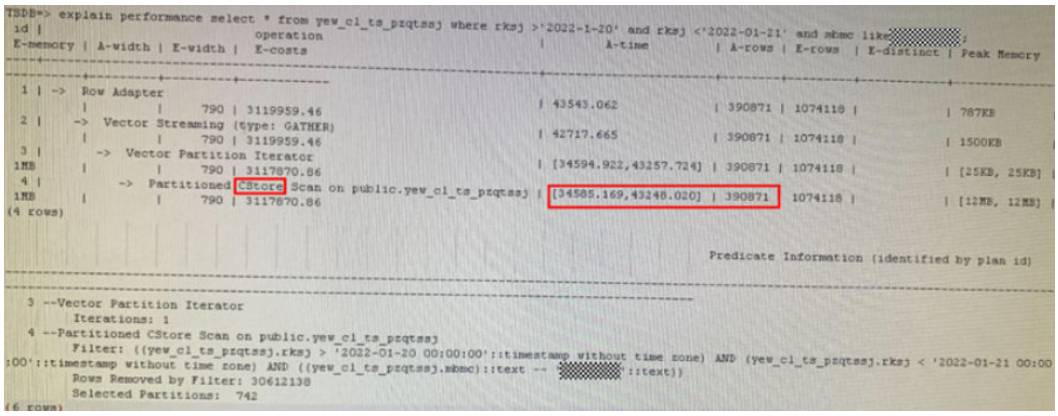
步骤3 获取高I/O消耗的SQL。

先通过OS命令识别消耗高的线程，然后结合DWS的线程号信息找到消耗高的业务SQL，获取工具和操作方法请下载**iowatcher**。

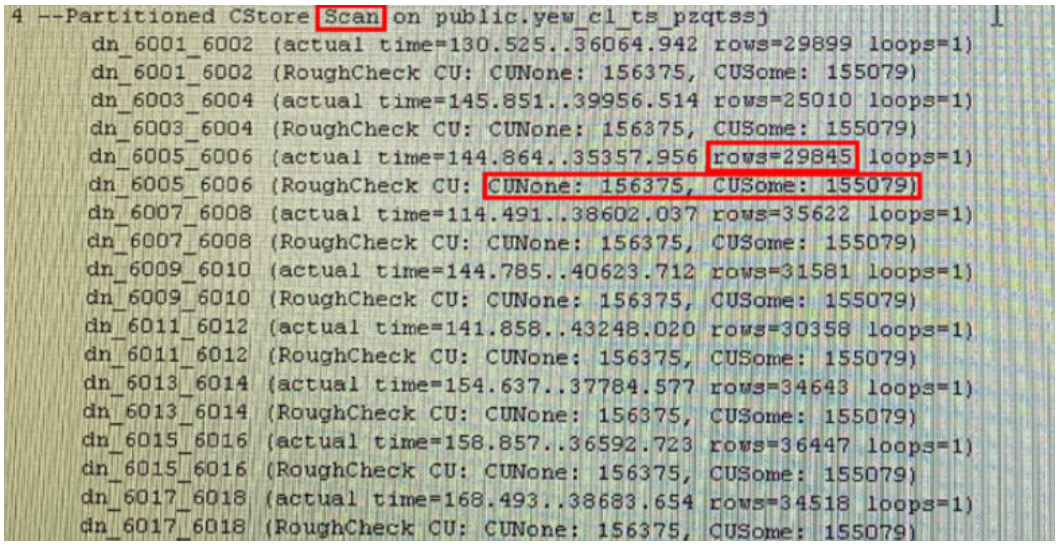
----结束

场景 1：列存小 CU 膨胀

某业务SQL查询出390871条数据需43248ms，分析计划主要耗时在Cstore Scan。



Cstore Scan的详细信息中，每个DN扫描出2w左右的数据，但是扫描了有数据的CU（CUSome）155079个，没有数据的CU（CUNone）156375个，说明当前小CU、未命中数据的CU极多，即CU膨胀严重。



触发因素：对列存表（尤其是分区表）进行高频小批量导入会造成CU膨胀。

处理方法

步骤1 列存表的数据入库方式修改为攒批入库，单分区单批次入库数据量需大于DN个数*6W。

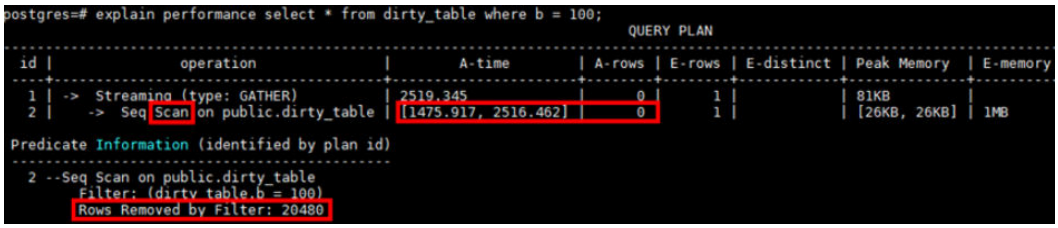
步骤2 如果因业务原因无法攒批入库，则需定期VACUUM FULL此类高频小批量导入的列存表。

步骤3 当小CU膨胀很快时，频繁VACUUM FULL也会消耗大量I/O，甚至加剧整个系统的I/O瓶颈，此场景建议修改列存表为行存表（CU长期膨胀严重的情况下，列存的存储空间优势和顺序扫描性能优势将不复存在）。

----结束

场景 2：脏数据&数据清理

某业务SQL总执行时间2.519s，其中Scan占了2.516s，同时该表的扫描最终只扫描到0条符合条件数据，过滤了20480条数据，即总共扫描了20480+0条数据却消耗了2s+，扫描时间与扫描数据量严重不符，此现象可判断为由于脏数据多从而影响扫描和I/O效率。



查看表脏页率为99%，VACUUM FULL后性能优化到100ms左右。


```
postgres=# SELECT relname
,pg_stat_get_live_tuples(c.oid) AS n_live_tup
,pg_stat_get_dead_tuples(c.oid) AS n_dead_tup
,round(n_dead_tup * 100 / (n_live_tup + n_dead_tup + 1), 2) AS dead_tup_ratio
FROM pg_class c
WHERE relname = 'dirty_table';
   relname   | n_live_tup | n_dead_tup | dead_tup_ratio
-------------+------------+------------+----------------
 dirty_table |      20756 |  204700642 |          99.99
(1 row)

postgres=# VACUUM FULL dirty_table;
VACUUM
postgres=# explain performance select * from dirty_table where b = 100;
                                                                                QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------
 id |               operation                |       A-time        | A-rows | E-rows | E-distinct | Peak Memory  | E-memory
----+----------------------------------------+---------------------+--------+--------+------------+--------------+----------
  1 | ->  Streaming (type: GATHER)           | 123.619             |      0 |      1 |            | 81KB         |
  2 |    ->  Seq Scan on public.dirty_table  | [87.666, 119.871]   |      0 |      1 |            | [18KB, 18KB] | 1MB

 Predicate Information (identified by plan id)
-----------------------------------------------
  2 --Seq Scan on public.dirty_table
        Filter: (dirty_table.b = 100)
        Rows Removed by Filter: 20480
```

触发因素：表频繁执行UPDATE/DELETE导致脏数据过多，且长时间未VACUUM FULL清理。

处理方法

步骤1 对频繁UPDATE/DELETE产生脏数据的表，定期VACUUM FULL，因大表的VACUUM FULL也会消耗大量I/O，因此需要在业务低峰时执行，避免加剧业务高峰期I/O压力。

步骤2 当脏数据产生很快，频繁VACUUM FULL也会消耗大量I/O，甚至加剧整个系统的I/O瓶颈，这时需要考虑脏数据的产生是否合理。针对频繁DELETE的场景，可以考虑如下方案：

1. 全量DELETE修改为TRUNCATE或者使用临时表替代。
2. 定期DELETE某时间段数据，使用分区表并使用TRUNCATE或DROP分区替代。

----结束

场景 3：表存储倾斜

例如表Scan的A-time中，max time DN执行耗时6554ms，min time DN耗时0s，DN之间扫描差异超过10倍以上，这种集合Scan的详细信息，基本可以确定为表存储倾斜导致。

```
postgres=# explain performance select * from skew_table where b = 10000;
                                                                                QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------
 id |               operation                |       A-time        | A-rows | E-rows | E-distinct | Peak Memory  | E-memory
----+----------------------------------------+---------------------+--------+--------+------------+--------------+----------
  1 | ->  Streaming (type: GATHER)           | 6560.625            |    512 |    499 |            | 88KB         |
  2 |    ->  Seq Scan on public.skew_table   | [0.000, 6554.298]   |    512 |    499 |            | [18KB, 46KB] | 1MB

Predicate Information (identified by plan id)
-----------------------------------------------
  2 --Seq Scan on public.skew_table
        Filter: (skew_table.b = 10000)
        Rows Removed by Filter: 5119488
2 --Seq Scan on public.skew_table
        dn_6001_6002 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6003_6004 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6005_6006 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6007_6008 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6009_6010 (actual time=18.618..6554.298 rows=512 loops=1) (filter time=485.531)
        dn_6011_6012 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6013_6014 (actual time=0.000..0.000 rows=0 loops=1) (filter time=0.000)
        dn_6015_6016 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6017_6018 (actual time=0.000..0.000 rows=0 loops=1) (filter time=0.000)
        dn_6019_6020 (actual time=0.000..0.000 rows=0 loops=1) (filter time=0.000)
        dn_6021_6022 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6023_6024 (actual time=0.000..0.000 rows=0 loops=1) (filter time=0.000)
        dn_6025_6026 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6027_6028 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
        dn_6029_6030 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
```


通过table_distribution发现所有数据倾斜到了dn_6009单个DN，修改分布列使得表存储分布均匀后，max dn time和min dn time基本维持在相同水平400ms左右，Scan时间从6554ms优化到431ms。

```
postgres=# select table_distribution('public','skew_table');
             table_distribution
-----------------------------------------------
(public,skew_table,dn_6005_6006,8192)
(public,skew_table,dn_6013_6014,8192)
(public,skew_table,dn_6027_6028,8192)
(public,skew_table,dn_6021_6022,8192)
(public,skew_table,dn_6003_6004,8192)
(public,skew_table,dn_6023_6024,8192)
(public,skew_table,dn_6001_6002,8192)
(public,skew_table,dn_6025_6026,8192)
(public,skew_table,dn_6029_6030,8192)
(public,skew_table,dn_6009_6010,10894655488)
(public,skew_table,dn_6007_6008,8192)
(public,skew_table,dn_6019_6020,8192)
(public,skew_table,dn_6017_6018,8192)
(public,skew_table,dn_6011_6012,8192)
(public,skew_table,dn_6015_6016,8192)
(15 rows)

postgres=# alter table skew_table distribute by hash(b);
ALTER TABLE
postgres=# select table_distribution('public','skew_table');
             table_distribution
-----------------------------------------------
(public,skew_table,dn_6021_6022,727785472)
(public,skew_table,dn_6027_6028,741949440)
(public,skew_table,dn_6015_6016,717979648)
(public,skew_table,dn_6025_6026,738680832)
(public,skew_table,dn_6023_6024,683122688)
(public,skew_table,dn_6019_6020,694018048)
(public,skew_table,dn_6029_6030,777904128)
(public,skew_table,dn_6007_6008,775725056)
(public,skew_table,dn_6009_6010,686391296)
(public,skew_table,dn_6013_6014,728875008)
(public,skew_table,dn_6011_6012,732143616)
(public,skew_table,dn_6003_6004,709263360)
(public,skew_table,dn_6001_6002,739770368)
(public,skew_table,dn_6005_6006,697286656)
(public,skew_table,dn_6017_6018,744128512)
(15 rows)

postgres=# explain performance select * from skew_table where b = 10000;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 id |             operation               |       A-time       | A-rows | E-rows | E-distinct | Peak Memory  | E-memory
----+-------------------------------------+--------------------+--------+--------+------------+--------------+----------
  1 | ->  Streaming (type: GATHER)        | 190.969            |    512 |    499 |            | 87KB         |
  2 |     ->  Seq Scan on public.skew_table | [185.666, 185.666] |    512 |    499 |            | [18KB, 18KB] | 1MB

Predicate Information (identified by plan id)
-----------------------------------------------
  2 --Seq Scan on public.skew_table
        Filter: (skew_table.b = 10000)
        Rows Removed by Filter: 320512
```

触发因素：分布式场景，表分布列选择不合理会导致存储倾斜，同时导致DN间压力失衡，单DN I/O压力大，整体I/O效率下降。

解决办法：修改表的分布列使表的存储分布均匀，分布列选择原则参见**选择分布列**。

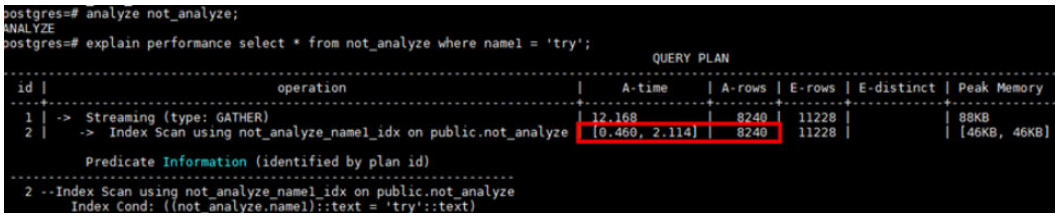
场景 4：无索引、有索引不走

某一次点查询，Seq Scan扫描需要3767ms，因涉及从4096000条数据中获取8240条数据，符合索引扫描的场景（海量数据中寻找少量数据），在对过滤条件列增加索引后，计划依然是Seq Scan而没有走Index Scan。

```
postgres=# create index on not_analyze(name1);
CREATE INDEX
postgres=# explain performance select * from not_analyze where name1 = 'try';
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 id |              operation               |        A-time        | A-rows | E-rows | E-distinct | Peak Memory  |
----+--------------------------------------+----------------------+--------+--------+------------+--------------+
  1 | ->  Streaming (type: GATHER)         | 3773.764             |   8240 |  10000 |            | 88KB         |
  2 |     ->  Seq Scan on public.not_analyze | [2816.353, 3767.258] |   8240 |  10000 |            | [46KB, 46KB] |

      Predicate Information (identified by plan id)
-------------------------------------------------------
  2 --Seq Scan on public.not_analyze
        Filter: ((not_analyze.name1)::text = 'try'::text)
        Rows Removed by Filter: 40960000
```

对目标表ANALYZE后，计划能够自动选择索引，性能从3s+优化到2ms+，极大降低I/O消耗。



常见场景：行存大表的查询场景，从大量数据中访问极少数据，没走索引扫描而是走顺序扫描，导致I/O效率低，不走索引常见有两种情况：

- 过滤条件列上没建索引。
- 有索引但是计划没选索引扫描。

触发因素：

- 常用过滤条件列没有建索引。
- 表中数据因执行DML操作后产生数据变化未及时ANALYZE，导致优化器无法选择索引扫描计划，ANALYZE介绍参见**ANALYZE** 。

处理方式：

步骤1 对行存表常用过滤列增加索引，索引基本设计原则：

- 索引列选择distinct值多，且常用于过滤条件，过滤条件多时可以考虑建组合索引，组合索引中distinct值多的列排在前面，索引个数不宜超过3个。
- 大量数据带索引导入会产生大量I/O，如果该表涉及大量数据导入，需严格控制索引个数，建议导入前先将索引删除，导入完成后再重新建索引。

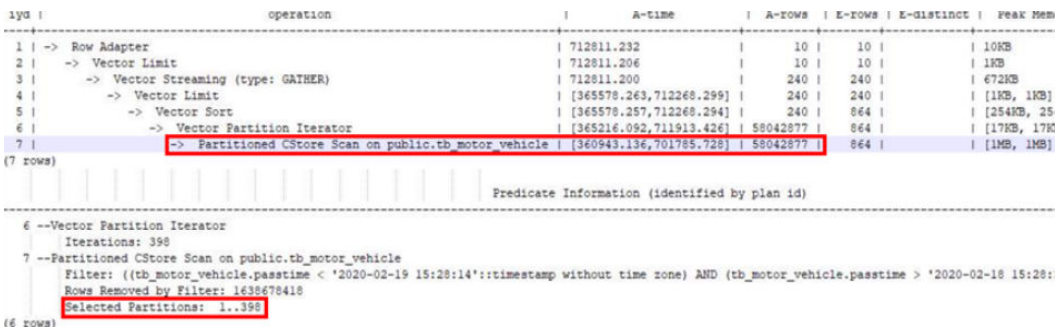
步骤2 对频繁做DML操作的表，业务中加入及时ANALYZE，主要场景：

- 表数据从无到有。
- 表频繁进行INSERT/UPDATE/DELETE。
- 表数据即插即用，需要立即访问且只访问刚插入的数据。

----结束

场景 5：无分区、有分区不剪枝

例如某业务表进场使用createtime时间列作为过滤条件获取特定时间数据，对该表设计为分区表后没有走分区剪枝（Selected Partitions数量多），Scan花了701785ms，I/O效率极低。



在增加分区键createtime作为过滤条件后，Partitioned scan走分区剪枝（Selected Partitions数量极少），性能从700s优化到10s，I/O效率极大提升。


```
explain performance SELECT passtime FROM tb_motor_vehicle WHERE createtime > '2020-02-19 00:00:00' AND createtime < '2020-02-20 00:00:00' AND passtime > '2020-02-19 0(
ORDER BY passtime DESC LIMIT 10000;
+----+-----------------------------------------------------------------------+---------------------+---------+---------+------------+-------------------+----------+
| id | operation                                                             | A-time              | A-rows  | E-rows  | E-distinct | Peak Memory       | E-memory |
+----+-----------------------------------------------------------------------+---------------------+---------+---------+------------+-------------------+----------+
|  1 | ->  Row Adapter                                                       | 12285.727           |   10000 |   10000 | NULL       | 10KB              |          |
|  2 |    ->  Vector Limit                                                   | 12284.854           |   10000 |   10000 | NULL       | 1KB               |          |
|  3 |       ->  Vector Streaming (type: GATHER)                             | 12284.840           |   10000 |  240000 | NULL       | 1464KB            |          |
|  4 |          ->  Vector Limit                                             | [10300.383,12227.410] |  240000 |  240000 | NULL       | [1KB, 1KB]        | 1MB      |
|  5 |             ->  Vector Sort                                           | [10300.368,12227.399] |  240000 | 11542800 | NULL       | [962KB, 962KB]    | 16MB     |
|  6 |                ->  Vector Partition Iterator                          | [8461.684,10419.017] | 57415974 | 11542791 | NULL       | [25KB, 25KB]      | 1MB      |
|  7 |                   ->  Partitioned CStore Scan on public.tb_motor_vehicle | [8389.677,10356.890] | 57415974 | 11542791 | NULL       | [1MB, 1MB]        | 1MB      |
```

常见场景：按照时间存储数据的大表，查询特征大多为访问当天或者某几天的数据，这种情况应该通过分区键进行分区剪枝（只扫描对应少量分区）来极大提升I/O效率，不走分区剪枝常见的情况有：

- 未设计成分区表。
- 设计了分区没使用分区键做过滤条件。
- 分区键做过滤条件时，对列值有函数转换。

触发因素：未合理使用分区表和分区剪枝功能，导致扫描效率低。

处理方式：

- 对按照时间特征存储和访问的大表设计成分区表。
- 分区键一般选离散度高、常用于查询过滤条件中的时间类型的字段。
- 分区间隔一般参考高频的查询所使用的间隔，需要注意的是针对列存表，分区间隔过小（例如按小时）可能会导致小文件过多的问题，一般建议最小间隔为按天。

场景 6：行存表求 count 值

某行存大表频繁全表count（指不带过滤条件或者过滤条件过滤很少数据的count），其中Scan花费43s，持续占用大量I/O，此类作业并发起来后，整体系统I/O持续100%，触发I/O瓶颈，导致整体性能慢。

```
postgres=# explain performance select count(*) from count_row;
                                                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
 id |               operation                |        A-time         | A-rows  | E-rows  | E-distinct | Peak Memory  | E-memory |
----+----------------------------------------+-----------------------+---------+---------+------------+--------------+----------+
  1 | ->  Aggregate                          | 44083.542             |       1 |       1 |            | 7KB          |          |
  2 |    ->  Streaming (type: GATHER)        | 44083.325             |      15 |      15 |            | 94KB         |          |
  3 |       ->  Aggregate                    | [26500.134, 44176.070] |      15 |      15 |            | [8KB, 8KB]   | 1MB      |
  4 |          ->  Seq Scan on public.count_row | [25899.765, 43585.137] | 40960000 | 2559293 |            | [44KB, 44KB] | 1MB      |
```

对比相同数据量的列存表（A-rows均为40960000），列存的Scan只花费14ms，I/O占用极低。

```
postgres=# explain performance select count(*) from count_col;
                                                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
 id |               operation                |        A-time         | A-rows  | E-rows  | E-distinct | Peak Memory  | E-memory |
----+----------------------------------------+-----------------------+---------+---------+------------+--------------+----------+
  1 | ->  Row Adapter                        | 28.796                |       1 |       1 |            | 10KB         |          |
  2 |    ->  Vector Aggregate                | 28.788                |       1 |       1 |            | 176KB        |          |
  3 |       ->  Vector Streaming (type: GATHER) | 28.743             |      15 |      15 |            | 91KB         |          |
  4 |          ->  Vector Aggregate          | [16.145, 25.270]      |      15 |      15 |            | [137KB, 137KB] | 1MB      |
  5 |             ->  CStore Scan on public.count_col | [7.540, 14.494] | 40960000 | 40960000 |            | [364KB, 364KB] | 1MB      |
```

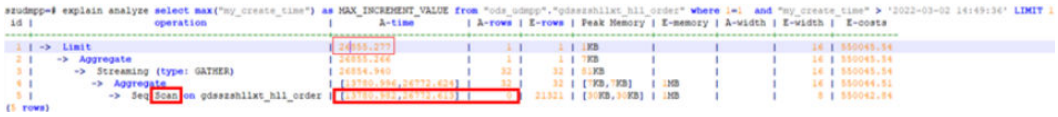
触发因素：行存表因其存储方式的原因，全表scan的效率较低，频繁的对大表全表扫描，导致I/O持续占用。

解决办法：

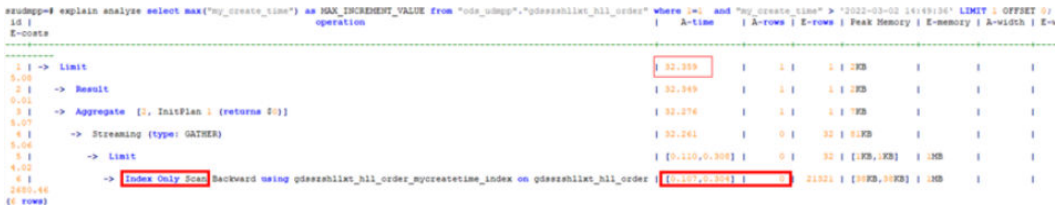
- 业务侧审视频繁全表count的必要性，降低全表count的频率和并发度。
- 如果业务类型符合列存表，则将行存表修改为列存表，提高I/O效率。

场景 7：行存表求 max 值

计算某行存表某列的max值，花费了26772ms，此类作业并发起后，整体系统I/O持续100%，触发I/O瓶颈，导致整体性能慢。



针对max列增加索引后，语句耗时从26s优化到32ms，极大减少I/O消耗。



触发因素：行存表max值逐个scan符合条件的值来计算max，当scan的数据量很大时，会持续消耗I/O。

解决办法：给max列增加索引，凭借btree索引数据有序存储的特征，加速扫描过程，降低I/O消耗。

场景 8：大量数据带索引导入

某业务场景数据往DWS同步时，延迟严重，集群整体I/O压力大。

后台查看等待视图有大量wait wal sync和WALWriteLock状态，均为xlog同步状态。

acquire lwlock	WALWriteLock	135
wait wal sync	(Null)	132
wait io	DataFileRead	36
Sort	(Null)	15
wait node(total 1): dn_6015_6016	(Null)	12
wait node(tlevel 6, total 24): dn_6039_6040	(Null)	9
wait node(total 1): dn_6013_6014	(Null)	9

触发因素：大量数据带索引（一般超过3个）导入（insert/copy/merge into）会产生大量xlog，导致主备同步慢，备机长期Catchup，整体I/O利用率飙高。。

解决方案：

- 严格控制每张表的索引个数，建议3个以内。
- 大量数据导入前先将索引删除，导入完成后再重新建索引。

场景 9：行存大表首次查询

某业务场景出现备DN持续catchup，I/O压力大，观察某个SQL等待视图在wait wal sync。

```
postgres=# select  * from pgxc_thread_wait_status where tid=140372871083776;
  node_name   | db_name | thread_name | query_id |       tid       | lwtid | ptid | tlevel | smpid |  wait_status  | wait_event
--------------+---------+-------------+----------+-----------------+-------+------+--------+-------+---------------+------------
 dn_6029_6030 | GS_DW   | Data Studio |        0 | 140372871083776 | 87672 |      |      0 |     0 | wait wal sync |
(1 row)
```

排查业务发现某查询语句执行时间较长，执行kill命令后恢复。

触发因素：行存表大量数据入库后，首次查询触发page hint产生大量XLOG，触发主备同步慢及大量I/O消耗。

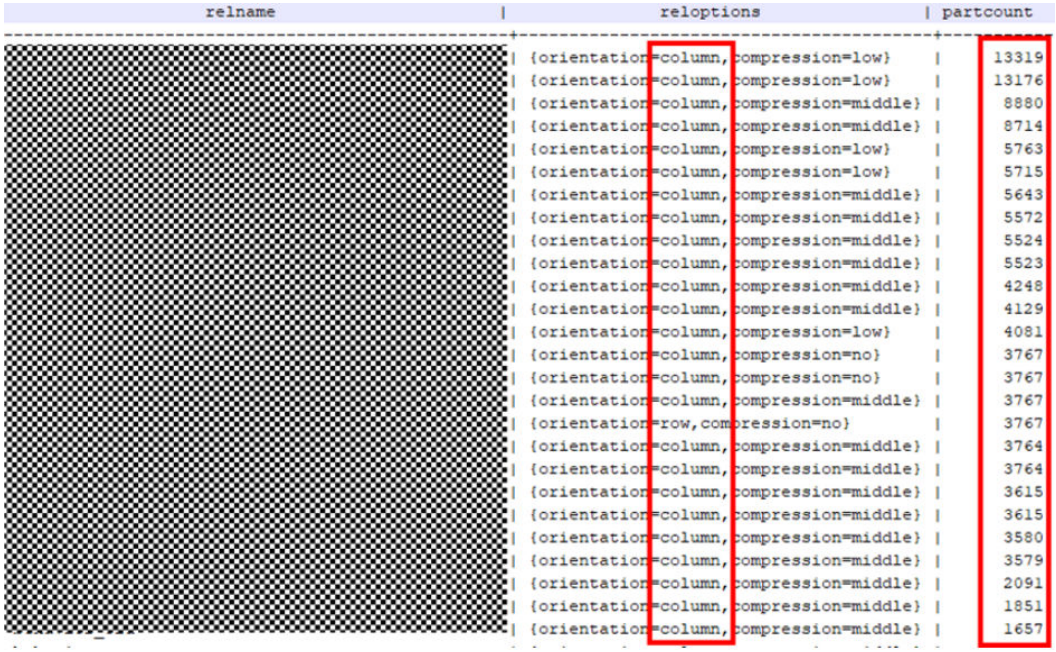
解决措施：

- 对该类一次性访问大量新数据的场景，修改行存表为列存表。
- 可关闭wal_log_hints和enable_crc_check参数（不推荐该方式，因故障期间有数据丢失风险）。

场景 10：小文件多 IOPS 高

某业务执行过程中，整个集群IOPS飙高，另外当出现集群故障后，长期Building不成功，IOPS飙高，相关表信息如下：

```
SELECT relname,reloptions,partcount FROM pg_class c INNER JOIN ( SELECT parentid,count(*) AS partcount FROM pg_partition GROUP BY parentid ) s ON c.oid = s.parentid ORDER BY partcount DESC;
```



触发因素：某业务库大量列存多分区（3000+）的表，导致小文件巨多（单DN文件2000w+），访问效率低，故障恢复Building极慢，同时Building也消耗大量IOPS，反向影响业务性能。

解决办法：

- 整改列存分区间隔，减少分区个数来降低文件个数。
- 列存表修改为行存表，行存的存储特征决定其文件个数不会像列存般膨胀严重。

小结

通过前面的场景总结得出，提升I/O使用效率可分为两个维度，即提升I/O的存储效率和计算效率（又称访问效率）。

- 提升存储效率包括整合小CU、减少脏数据、消除存储倾斜等。
- 提升计算效率包括分区剪枝、索引扫描等，可根据实际业务场景灵活处理。

6.28 高 CPU 系统性能调优方案

如果当前集群CPU负载较高，可参考如下步骤进行优化：

步骤1 检查当前集群业务是否占用CPU过高。

1. 登录GaussDB(DWS)管理控制台。
2. 在“监控 > 告警”界面，单击右上角“集群选择”下拉框，选中告警集群，查看集群最近7天的告警信息，通过定位信息锁定触发告警的节点名称。
3. 在“专属集群 > 集群列表”界面找到告警集群，在所在行操作列单击“监控面板”进入监控界面。
4. 选择“监控 > 节点监控 > 概览”可查看当前集群各节点CPU使用率的具体情况，单击最右的监控按钮，查看最近1/3/12/24小时的CPU性能指标，判断是否有CPU使用率突然增大的情况。

步骤2 设置资源池CPU限额与配额。

- 专属限额其实就是绑核，按照百分比的方式分配CPU核给资源池使用，该资源池上运行的复杂作业只能在分配的CPU上运行。
- 共享配额相当于给资源池按照百分比配置一定的权重，配额不限制资源池使用的CPU核，当一个CPU满负载时，在该CPU上运行作业的资源池按照权重的比例抢占CPU时间片。

可利用专属限额限制语句运行的CPU核心，利用配额指定语句争抢CPU时间片的能力。

步骤3 设置异常规则及时终止高CPU语句。

防止极端场景下某个语句占用CPU资源过多，导致数据库内其他语句因争抢CPU而变得缓慢迟钝的情况，可创建与CPU资源相关的异常规则。具体操作可参考**异常规则**，对超过异常规则阈值的SQL及时终止拦截，保持集群稳定。

步骤4 根据业务场景适当降低作业并发量。

----结束

6.29 降低内存的处理方案

如果当前集群内存负载较高，或出现“memory is temporary unavailable”内存报错，首先利用日志信息确定内存异常节点，然后连接到该节点查询**pv_total_memory_detail**视图确认当前是否还存在内存不足问题，可比较process_used_memory和max_process_memory的关系，如前者明显小于后者，则说明占用内存大的语句已经跑完或者被杀掉，当前系统已经恢复，若已经大于或比较接近，则说明当前内存使用已经或即将超限，若此时dynamic_used_memory过大，说明动态申请的内存过大，这类问题可能和正在运行的SQL强相关，此时可参考如下步骤进行优化：

步骤1 检查当前集群业务是否占用内存过高。

1. 登录GaussDB(DWS)管理控制台。
2. 在“监控 > 告警”界面，单击右上角“集群选择”下拉框，选中告警集群，查看集群最近7天的告警信息，通过定位信息锁定触发告警的节点名称。
3. 在“专属集群 > 集群列表”界面找到告警集群，在所在行操作列单击“监控面板”进入监控界面。
4. 选择“监控 > 节点监控 > 概览”可查看当前集群各节点CPU使用率的具体情况，单击最右的监控按钮 ，查看最近1/3/12/24小时的CPU性能指标，判断是否有CPU使用率突然增大的情况。

步骤2 设置异常规则及时终止高内存语句。

防止极端场景下某些语句使用内存过多，导致其他语句由于内存分配不足而出现算子下盘执行缓慢或者申请不到内存而执行失败的情况，可创建与内存资源相关的异常规则，具体操作可参考**异常规则**，对超过异常规则阈值的SQL进行及时终止拦截，保持集群稳定。

步骤3 分析计划，优化语句。

- 对语句中的相关表进行ANALYZE，矫正内存估算情况，避免该语句申请内存过大导致内存超限报错。
- 是否完全下推，参考**使排序下推**。
- 是否存在对数据量大的表执行broadcast。
- 是否有不合理的join顺序。例如，多表关联时，执行计划中优先关联的两表的中间结果集比较大，导致最终执行代价比较大。

步骤4 根据业务场景适当降低作业并发量。

----结束

7 集群异常

7.1 磁盘监控告警阈值太低，告警频繁

问题现象

DWS集群磁盘使用率达到80%就出现告警，告警频繁。

原因分析

集群配置的告警监控阈值不合理。

处理方法

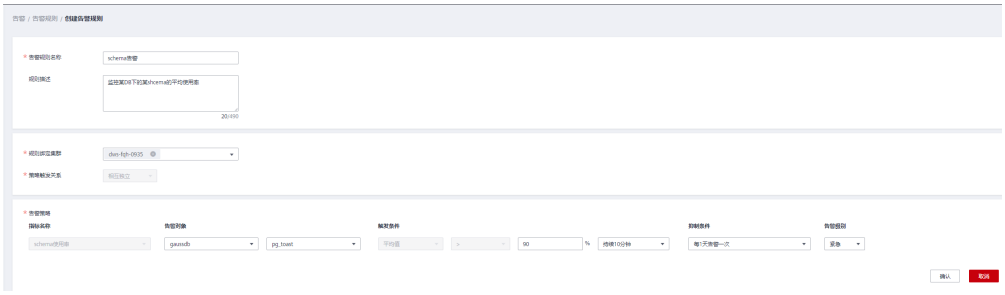
可在GaussDB(DWS)管理控制台设置告警的触发条件，指定达到磁盘使用率、告警持续时间及告警频次。

须知

集群磁盘使用率达到90%就会触发集群只读，需要预留时间来处理问题，避免使用率达到只读阈值。

1. 登录GaussDB(DWS) 管理控制台。
2. 在左侧导航栏，单击“告警管理”，切换至“告警”页签。
3. 单击左上角的“查看告警规则”按钮，进入告警规则页面。
4. 在指定告警规则名称所在行操作列，单击“修改”按钮进入修改告警规则页面。将触发条件修改为平均值大于90%，抑制条件修改为“每1天告警一次”。（此处仅做举例，实际情况以业务诉求为准。）
   - 触发条件：定义对监控指标做阈值判断的计算规则。目前主要使用一段时间内的平均值来降低告警震荡的几率。
   - 抑制条件：在指定的时间段内，抑制同类型告警的反复触发和消除。

图 7-1 设置告警规则



8 数据库使用

8.1 插入或更新数据时报错，提示分布键不能被更新

问题现象

往数据库插入或更新数据时报错，提示分布键不能被更新，错误信息如下所示：

```
ERROR: Distributed key column can't be updated in current version
```

原因分析

GaussDB(DWS)分布键不允许被更新。

处理方法

方法一：分布键目前暂不支持更新，直接跳过该报错。

方法二：将分布列修改为一个不会更新的列（8.1.0版本后，支持调整分布列，以下为示例）。

步骤1 查询当前表定义，返回结果显示该表分布列为c_last_name。

```
SELECT pg_get_tabledef('customer_t1');
```

```
gaussdb=> select pg_get_tabledef ('customer_t1');
                                  pg_get_tabledef
--------------------------------------------------------------------------------
 SET search_path = public;                                                     +
 CREATE  TABLE customer_t1 (                                                   +
         c_customer_sk integer,                                                +
         c_customer_id character(5),                                           +
         c_first_name character(6),                                            +
         c_last_name character(8)                                              +
 )                                                                             +
 WITH (orientation=column, compression=middle, colversion=2.0, enable_delta=false)+
 DISTRIBUTE BY HASH(c_last_name)                                               +
 TO GROUP group_version1;
(1 row)
```

步骤2 更新分布列中的数据时报错。

```
UPDATE customer_t1 SET c_last_name = 'Jimy' WHERE c_customer_sk = 6885;
```



```
gaussdb=> update customer_t1 set c_last_name = 'Jimy' where c_customer_sk = 6885;
ERROR:  Distributed key column can't be updated in current version
```

步骤3 将该表的分布列修改为不会更新的列，例如c_customer_sk。

```
ALTER TABLE customer_t1 DISTRIBUTE BY hash (c_customer_sk);
```

```
gaussdb=> alter table customer_t1 DISTRIBUTE BY hash (c_customer_sk);
ALTER TABLE
```

步骤4 重新执行更新旧的分布列的数据。更新成功。

```
UPDATE customer_t1 SET c_last_name = 'Jimy' WHERE c_customer_sk = 6885;
```

```
gaussdb=> update customer_t1 set c_last_name = 'Jimy' where c_customer_sk = 6885;
UPDATE 1
```

----结束

8.2 执行 SQL 语句时提示“Connection reset by peer”

问题现象

执行SQL语句时，提示“Connection reset by peer”。

```
ERROR: Failed to read response from Datanodes Detail: Connection reset by peer
```

原因分析

在网络压力大的情况下会因为socket通信问题，出现断连现象。

解决办法

- 通过流控机制防止网络压力过大，需要设置以下GUC参数的值控制网络流量峰值。

  comm_quota_size = 400，comm_usable_memory = 100 。

  通过如下步骤修改参数值：

  a. 登录GaussDB(DWS) 管理控制台。
  b. 在左侧导航树，单击“集群管理”。
  c. 在集群列表中找到所需要的集群，然后单击集群名称。
  d. 进入集群的“参数修改”页面，分别找到“comm_quota_size”和“comm_usable_memory”参数，修改其参数值，单击“保存”，确认无误后再单击“保存”。

- 数据库在识别此类错误后，会自动进行重试，重试次数使用GUC变量max_query_retry_times来控制。

  说明

  目前仅支持单条SQL语句的重试，暂不支持事务块中出错SQL重试。

8.3 VARCHAR(n)存储中文字符，提示 value too long for type character varying?

问题现象

VARCHAR(18)的字段，存储8个中文字符长度不够，报如下所示的错误：

```
org.postgresql.util.PSQLException: ERROR: value too long for type character varying(18)
```

原因分析

以UTF-8编码为例，一个中文占3~4个字节，即8个中文占24~32字节，超出VARCHAR(18)的最大18字节限制。

当表中某一字段包含有中文字符时，可使用char_length或length函数来查询字段字符长度，使用lengthb函数来查询字段字节长度。

```
 SELECT length('数据库database');
 length
--------
     11
(1 row)
SELECT lengthb('数据库database');
 length
--------
     17
(1 row)
```

处理方法

varchar(n)为变长类型，n代表可存储的最大字节数。中文字符通常占用3~4个字节。

请根据实际的中文字符长度，增加该字段的字段长度。示例中某字段要存储8个中文字符，则需要设置n至少为32，即VARCHAR(32)。

8.4 SQL 语句中字段名大小写敏感问题

问题现象

某表table01中存在以大小写字母组合的名称为“ColumnA”的字段，使用SELECT语句查询该字段时，提示字段不存在，报错：column "columna" does not exist。

```
select ColumnA from table01 limit 100;
ERROR:  column "columna" does not exist
LINE 1: select columna from TABLE_01;
               ^
CONTEXT:  referenced column: columna
```

原因分析

在GaussDB(DWS)中，SQL语句中的表字段等名称带双引号时大小写敏感；不带双引号时大小写不敏感，按全小写处理。

处理方法

- 在大小写不敏感的场景下，将字段名称的双引号去掉。
- 在大小写敏感的场景下，执行SQL语句时需要给字段名称加上双引号。

  示例：

  table01中，使用SELECT语句查询ColumnA时，ColumnA字段带上双引号，查询成功：

  ```
  SELECT "ColumnA" FROM table01 LIMIT 100;
  ```

8.5 删除表时报错：cannot drop table test because other objects depend on it

问题现象

删除表时出现如下报错“cannot drop table test because other objects depend on it.”。

```
tddb=# create table t1  (a int, b serial) distribute by hash(a);
NOTICE:  CREATE TABLE will create implicit sequence "t1_b_seq" for serial column "t1.b"
CREATE TABLE
tddb=# create table t2  (a int, b int default nextval('t1_b_seq')) distribute by hash(a);
CREATE TABLE
tddb=# drop table t1;
ERROR:  cannot drop table t1 because other objects depend on it
DETAIL:  default for table t2 column b depends on sequence t1_b_seq
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

原因分析

创建t1表后，隐式创建了sequence，然后创建t2表的时候，引用了该sequence。在删除t1表的时候，由于会级联删除sequence，但是该sequence被其他对象依赖，因此导致该报错。

处理方法

如不需保留t2，可通过DROP CASCADE的方式级联删除。

如需保留该表和该sequence，可以通过以下图例中方式进行删除：

```
tddb=# drop table t1;
ERROR:  cannot drop table t1 because other objects depend on it
DETAIL:  default for table t2 column b depends on sequence t1_b_seq
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
tddb=#
tddb=# alter sequence t1_b_seq owned by none;
ALTER SEQUENCE
tddb=#
tddb=# drop table t1;
DROP TABLE
tddb=#
```


8.6 多个表同时进行 MERGE INTO UPDATE 时，执行失败

问题现象

多个表同时进行MERGE INTO UPDATE时，执行失败。

原因分析

查看日志，发现有如下错误日志：

```
dn_6007_6008 YY003 79375943437085786 [BACKEND] DETAIL:  blocked by hold lock thread 0, statement <pending twophase transaction>, hold lockmode (null).
```

这是由于分布式锁导致的，两个DN节点都锁住了自己的数据块，然后又在等待对方的数据块，所以导致锁超时。

这种行为是两阶段锁的特性，分布式情况下都会面临这样的问题。

处理方法

建议对单表执行MERGE，将并发操作改为串行。

8.7 session_timeout 设置导致 JDBC 业务报错

问题现象

通过JDBC连接集群执行COPY导入时报错：

```
org.postgresql.util.PSQLException: Database connection failed when starting copy at org.postgresql.core.v3.QueryExecutorImpl.startCopy(QueryExecutorImpl.java:804) at org.postgresql.copy.CopyManager.copyIn(CopyManager.java:52) at org.postgresql.copy.CopyManager.copyIn(CopyManager.java:161) at org.postgresql.copy.CopyManager.copyIn(CopyManager.java:146) at copy.main(copy.java:95) Caused by: java.io.EOFExceptionat org.postgresql.core.PGStream.ReceiveChar(PGStream.java:284) at org.postgresql.core.v3.QueryExecutorImpl.processCopyResults(QueryExecutorImpl.java:1008) at org.postgresql.core.v3.QueryExecutorImpl.startCopy(QueryExecutorImpl.java:802) ... 4 more
```

原因分析

数据库默认设置session_timeout=10min，即连接空闲超过10分钟，自动断开连接，导致连接失效。

处理方法

可登录GaussDB(DWS)管理控制台，设置session_timeout为0或预期时长，使会话保持长时间连接。

1. 登录GaussDB(DWS)管理控制台。在集群列表中找到所需要的集群，单击集群名称，进入“集群详情”页面。
2. 单击“参数修改”页签，修改session_timeout参数值，然后单击“保存”。

说明

COPY导入执行完成后，建议继续设置session_timeout=10min，因为如果有客户端长时间连接数据库，但对数据库不进行任何操作，该连接将一直占用一个线程，如果这样的客户端连接很多，就会出现大量的线程都被空闲的连接占用，从而导致数据库连接满或者资源浪费。

8.8 DROP TABLE 失败

问题现象

DROP TABLE失败的两种现象：

- 在使用“SELECT * FROM DBA_TABLES;”语句（或者gsql客户端也可以使用\dt +命令）查看数据库中无相关表；CREATE TABLE时报该表已经存在的错误，使用DROP TABLE语句失败，报不存在该表的错误，导致无法再次创建表。
- 在使用“SELECT * FROM DBA_TABLES;”语句（或者gsql客户端也可以使用\dt +命令）查看数据库中有相关表；使用DROP TABLE时失败，报不存在该表的错误，导致无法再次创建表。

原因分析

导致该错误的原因有的节点上有该张表，有的节点上无该张表。

解决办法

当遇到上面两种现象时，使用DROP TABLE语句失败，请再次使用DROP TABLE if exists table_name;命令，使得DROP表可以成功。

8.9 使用 string_agg 函数查询执行结果不稳定

问题现象

SQL语句查询结果不一致。

原因分析

某业务场景中的SQL语句中使用了string_agg函数，语句逻辑如下：

```
postgres=# select * from employee;
 empno | ename  |   job    | mgr  |      hiredate       |  sal  | comm | deptno
-------+--------+----------+------+---------------------+-------+------+--------
  7499 | ALLEN  | SALEMAN | 7698 | 2014-11-12 00:00:00 | 16000 |  300 |     30
  7566 | JONES  | MANAGER | 7839 | 2015-12-12 00:00:00 | 32000 |    0 |     20
  7654 | MARTIN | SALEMAN | 7698 | 2016-09-12 00:00:00 | 12000 | 1400 |     30
(3 rows)
```

执行如下SQL语句：

```
select count(*) from
(select deptno, string_agg(ename, ',') from employee group by deptno) t1 ,
(select deptno, string_agg(ename, ',') from employee group by deptno) t2
where t1.string_agg = t2.string_agg;
```

在循环多次执行这个语句的时候，发现结果不稳定，输出结果有时候是t1，有时候是t2，因此怀疑是数据库有问题，结果集不正确。

String_agg函数的作用是将组内的数据合并成一行，但是如果用户用法是string_agg(ename, ',')这种情况下，结果集就是不稳定的，因为没有指定组合的顺序。

例如，上述SQL语句中的输出结果可以是以下任意一种，且都是合理的。

```
30 | ALLEN,MARTIN
30 | MARTIN,ALLEN
```

因此有可能出现t1这个subquery中的结果和t2这个subquery中的结果对于deptno=30的时候的输出结果是不一样的。

处理方法

String_agg中增加order by，语句修改为如下格式保证ename字段是按照相同的顺序来拼接的，从而满足查询结果是稳定的。

```
select count(*) from
(select deptno, string_agg(ename, ',' order by ename desc) from employee group by deptno) t1 ,
(select deptno, string_agg(ename, ',' order by ename desc) from employee group by deptno) t2
where t1.string_agg = t2.string_agg;
```

8.10 查询表大小时报错“could not open relation with OID xxx”

问题现象

在执行pg_table_size查询表大小时，出现报错“could not open relation with OID xxxx.”。

原因分析

通过执行pg_table_size这个查询接口，对于不存在的表会返回NULL或者报错。

处理方法

1. 通过Function的exception方式屏蔽该报错，将大小统一到一个值，对于不存在的表，可以用大小为-1来表示，函数如下：

```
CREATE OR REPLACE FUNCTION public.pg_t_size(tab_oid OID,OUT retrun_code text)
RETURNS text
LANGUAGE plpgsql
AS $$ DECLARE
v_sql text;
ts text;
BEGIN
V_SQL:='select pg_size_pretty(pg_table_size('||tab_oid||'))';
EXECUTE IMMEDIATE V_SQL into ts;
IF ts IS NULL
THEN RETRUN_CODE:=-1;
ELSE
return ts;
END IF;
EXCEPTION
WHEN OTHERS THEN
RETRUN_CODE:=-1;
END$$;
```

2. 执行如下命令查询结果：

```
call public.pg_t_size('1','');
 retrun_code
```



```
-------------
 -1
(1 row)

select oid from pg_class limit 2;
 oid
------
 2662
 2659
(2 rows)

call public.pg_t_size('2662','');
 retrun_code
-------------
 120 KB
(1 row)
```

8.11 DROP TABLE IF EXISTS 语法误区

问题现象

使用DROP TABLE IF EXISTS语句删除表存在语法误区，理解不当将会删除错误。

原因分析

DROP TABLE IF EXISTS语法可以简单这样理解：

1. 判断当前CN是否存在该table。
2. 如果存在，就给其他CN和DN下发DROP命令；如果不存在，则跳过。

而不是：

1. 将DROP TABLE IF EXISTS下发给所有CN和DN。
2. 各个CN和DN判断自己有没有该table，如果有的话执行DROP。

处理方法

如果出现某些表定义在部分CN/DN存在，部分CN/DN不存在时，是不可以直接用DROP TABLE IF EXISTS修复的。

这种情况的通用修复方式为：

1. CREATE TABLE IF NOT EXISTS将所有CN和DN的表定义补齐。
2. 如果表没有用，就在CN上DROP table删掉所有CN和DN的表定义，如果表还有用，继续使用即可。

8.12 不同用户查询同表显示数据不同

问题现象

2个用户登录相同数据库human_resource，分别执行的查询语句如下：select count(*) from areas，查询同一张表areas时，查询结果却不一致。

原因分析

请先判断同名的表是否确实是同一张表。在关系型数据库中，确定一张表通常需要3个因素：database，schema，table。从问题现象描述看，database，table已经确定，分别是human_resource、areas。接着，需要检查schema。使用dbadmin，user01分别登录发现，search_path依次是public和"$user"。dbadmin作为集群管理员，默认不会创建dbadmin同名的schema，即不指定schema的情况下所有表都会建在public下。而对于普通用户如user01，则会在创建用户时，默认创建同名的schema，即不指定schema时表都会创建在user01的schema下。最终确定该案例发生时，确实因为2个用户之间交错对表进行操作，导致了同名不同表的情况。

解决办法

在操作表时加上schema引用，格式：schema.table。

8.13 修改索引只调用索引名提示索引不存在

问题现象

创建分区表索引HR_staffS_p1_index1，不指定索引分区的名字。

```
CREATE INDEX HR_staffS_p1_index1 ON HR.staffS_p1 (staff_ID) LOCAL;
```

创建分区索引HR_staffS_p1_index2，并指定索引分区的名字。

```
CREATE INDEX HR_staffS_p1_index2 ON HR.staffS_p1 (staff_ID) LOCAL
(
    PARTITION staff_ID1_index,
    PARTITION staff_ID2_index TABLESPACE example3,
    PARTITION staff_ID3_index TABLESPACE example4
) TABLESPACE example;
```

修改索引分区staff_ID1_index的表空间为example1：

调用“ALTER INDEX HR_staffS_p1_index2 MOVE PARTITION staff_ID2_index TABLESPACE example1;”提示索引不存在。

原因分析

重新创建索引CREATE INDEX HR_staffS_p1_index2 MOVE PARTITION staff_ID2_index TABLESPACE example1，提示索引已存在，然后执行以下SQL命令或者gsql客户端元命令\d+ HR.staffS_p1 查询索引时发现索引已存在。

```
SELECT * FROM DBA_INDEXES WHERE index_name = HR.staffS_p1 ;
```

推测是当前模式是public模式，而不是hr模式，导致检索不到该索引。

使用“ALTER INDEX hr.HR_staffS_p1_index2 MOVE PARTITION staff_ID2_index TABLESPACE example1;”验证推测，发现调用成功。

接着调用ALTER SESSION SET CURRENT_SCHEMA TO hr;再次调用“ALTER INDEX HR_staffS_p1_index2 MOVE PARTITION staff_ID2_index TABLESPACE example1;”发现设置成功。

解决办法

在操作表、索引、视图时加上schema引用，格式：schema.table。

8.14 执行 CREATE SCHEMA 语句时，报错 SCHEMA 已存在

问题现象

执行CREATE SCHEMA语句时，报错要创建的schema已存在。

```
ERROR: schema "schema" already exists
```

原因分析

在SQL语句中，字段名称是区分大小写的，默认为小写字段名。

解决办法

在大小写有严格要求的场景下，需要给字段名称加上双引号，如下图重新执行SQL语句，则创建成功。

```
1 CREATE SCHEMA "SCHEMA";
2
```

```
[INFO] 操作影响的记录行数：0
[INFO] 执行时间： 8 sec
[INFO] 执行成功...
```

8.15 删除数据库失败，提示有 session 正在连接

问题现象

删除数据库失败，提示有session正在连接。

原因分析

删除数据库时可能当前仍有session正在连接数据库，或者有session在不停地连接该数据库，故删除数据库失败。需要查看数据库中的session，检查是否仍有session在连接，如果有，排查连接数据库的机器，停止连接后再删除数据库。

处理方法

步骤1 使用SQL客户端工具连接数据库。

步骤2 执行如下命令查看当前会话。

```
SELECT * FROM pg_stat_activity;
```

查询结果中的关键字段，说明如下：

- datname：用户会话所连接的数据库名称。
- usename：连接数据库的用户名。
- client_addr：连接数据库的客户端主机的IP地址。

在查询结果中，找出待删除的数据库名称及对应的客户端主机IP地址。

步骤3 请根据客户端主机的IP地址排查连接数据库的机器及应用，并停止相关的连接。

```
CLEAN CONNECTION TO ALL FOR DATABASE database_name;
```

步骤4 重新执行删除数据库的命令。

```
DROP DATABASE [ IF EXISTS ] database_name;
```

----结束

8.16 在 Java 中，读取 character 类型的表字段时返回类型为什么是 byte?

问题现象

数据库中新建一张表，某个表字段使用character类型，在Java中读取character类型的字段时返回类型是byte。

例如，创建示例表table01：

```
CREATE TABLE IF NOT EXISTS table01(
  msg_id character(36),
  msg character varying(50)
);
```

在Java中，读取character类型的字段代码如下：

```
ColumnMetaInfo(msg_id,1,Byte,true,false,1,true);
```

原因分析

- CHARACTER(n)是定长字符串类型，当实际字符串长度不够时数据库会用空格补全，Java用byte类型接收。
- CHARACTER VARYING(n)是变长字符串类型，Java使用String类型接收。

8.17 执行表分区操作时，报错：start value of partition "xxx" NOT EQUAL up-boundary of last partition

问题现象

执行ALTER TABLE PARTITION时报错：

```
ERROR:start value of partition "XX" NOT EQUAL up-boundary of last partition.
```

原因分析

在同一条ALTER TABLE PARTITION语句中，既存在DROP PARTITION又存在ADD PARTITION时，无论它们的相对顺序是什么，GaussDB(DWS)总会先执行DROP PARTITION再执行ADD PARTITION。执行完DROP PARTITION删除末尾分区后，再执行ADD PARTITION操作会出现分区间隙，导致报错。

解决办法

为防止出现分区间隙，需要将ADD PARTITION的START值前移。

以分区表partitiontest为例：

```
CREATE TABLE partitiontest
(
 c_int integer,
 c_time TIMESTAMP WITHOUT TIME ZONE
)
PARTITION BY range (c_int)
(
 partition p1 start(100)end(108),
 partition p2 start(108)end(120)
);
```

执行如下两种语句会发生报错：

```
ALTER TABLE partitiontest ADD PARTITION p3 start(120)end(130), DROP PARTITION p2;
ERROR:  start value of partition "p3" NOT EQUAL up-boundary of last partition.
ALTER TABLE partitiontest DROP PARTITION p2,ADD PARTITION p3 start(120)end(130);
ERROR:  start value of partition "p3" NOT EQUAL up-boundary of last partition.
```

可以修改语句为：

```
ALTER TABLE partitiontest ADD PARTITION p3 start(108)end(130), DROP PARTITION p2;
ALTER TABLE partitiontest DROP PARTITION p2,ADD PARTITION p3 start(108)end(130);
```

8.18 重建索引失败

问题现象

当Desc表的索引出现损坏时，无法进行一系列操作，报错信息可能为：

```
index \"%s\" contains corrupted page at block
%u" ,RelationGetRelationName(rel),BufferGetBlockNumber(buf),
please reindex it.
```

原因分析

在实际操作中，索引会由于软件问题或者硬件问题引起崩溃。例如，当索引分裂完而磁盘空间不足、出现页面损坏等问题时，会导致索引损坏。

解决办法

若此表是以pg_cudesc_xxxxx_index进行命名则为列存表，则说明desc表的索引表损坏，通过desc表的索引表表名，找到对应主表的oid和表，执行REINDEX INTERNAL TABLE name语句重建cudesc表的索引。

8.19 视图查询时执行失败

问题现象

在连接集群数据库之后，使用某个视图执行查询，可能会操作失败，提示错误信息如下：

```
[GAUSS-01850] : object with oid 16420 is not a partition object
```


原因分析

此视图是建立分区表的某个分区上的，查询此视图时需要访问到对应的分区，从而必须首先查询对应的分区是否存在。当对应的分区已经被删除后，无法访问到此分区，从而导致视图访问也失败，报出如上类似的信息。

解决办法

步骤1 确定是针对视图对象进行的SQL操作，并获得视图的名字。

直接检查SQL语句的FROM对象，确定是否为视图。若是，则直接获得该视图的名字。

步骤2 使用已获得的视图的名字和schema，删除该视图。

步骤3 重新执行SQL语句。对于查询操作，由于对应的分区已经被删除，视图的存在没有意义。

----结束

8.20 全局 SQL 查询

通过pgxc_stat_activity函数和视图实现全局SQL查询。

1. 执行如下命令连接数据库。
   gsql -d *postgres* **-p** 8000
2. 执行如下命令创建pgxc_stat_activity函数。

```
DROP FUNCTION PUBLIC.pgxc_stat_activity() cascade;
CREATE OR REPLACE FUNCTION PUBLIC.pgxc_stat_activity
(
OUT coorname text,
OUT datname text,
OUT usename text,
OUT pid bigint,
OUT application_name text,
OUT client_addr inet,
OUT backend_start timestamptz,
OUT xact_start timestamptz,
OUT query_start timestamptz,
OUT state_change timestamptz,
OUT waiting boolean,
OUT enqueue text,
OUT state text,
OUT query_id bigint,
OUT query text
)
RETURNS setof RECORD
AS $$
DECLARE
row_data pg_stat_activity%rowtype;
coor_name record;
fet_active text;
fetch_coor text;
BEGIN
--Get all the node names
fetch_coor := 'SELECT node_name FROM pg_catalog.pgxc_node WHERE node_type=''C''';
FOR coor_name IN EXECUTE(fetch_coor) LOOP
coorname := coor_name.node_name;
fet_active := 'EXECUTE DIRECT ON (' || coorname || ') ''SELECT * FROM pg_catalog.pg_stat_activity
WHERE pid != pg_catalog.pg_backend_pid() and application_name not in (SELECT node_name FROM
pg_catalog.pgxc_node WHERE node_type=''''C''''); ''';
FOR row_data IN EXECUTE(fet_active) LOOP
datname := row_data.datname;
pid := row_data.pid;
```



```
usename := row_data.usename;
application_name := row_data.application_name;
client_addr := row_data.client_addr;
backend_start := row_data.backend_start;
xact_start := row_data.xact_start;
query_start := row_data.query_start;
state_change := row_data.state_change;
waiting := row_data.waiting;
enqueue := row_data.enqueue;
state := row_data.state;
query_id := row_data.query_id;
query := row_data.query;
RETURN NEXT;
END LOOP;
END LOOP;
return;
END; $$
LANGUAGE 'plpgsql';
```

3. 执行如下命令创建pgxc_stat_activity视图。

```
CREATE VIEW PUBLIC.pgxc_stat_activity AS SELECT * FROM PUBLIC.pgxc_stat_activity();
```

4. 执行如下sql语句查询全局会话信息。

```
SELECT * FROM PUBLIC.pgxc_stat_activity order by coorname;
```

8.21 如何判断表是否执行过 UPDATE 或 DELETE

问题现象

DWS中有两种情况需要关注表是否做过UPDATE及DELETE操作：

1. 对表频繁执行UPDATE或者DELETE操作会产生大量的磁盘页面碎片，从而逐渐降低查询的效率，需要将磁盘页面碎片恢复并交还操作系统，即VACUUM FULL操作，这种场景下需要查找出哪些表执行过UPDATE；
2. 判断一张表是否是维度表，是否可以从Hash表变更为复制表，可以查看这张表是否执行过UPDATE或DELETE，如果执行过UPDATE或DELETE操作，则不能修改为复制表。

处理方法

通过以下命令查找哪些表执行过UPDATE及DELETE操作：

```
ANALYZE tablename;
SELECT
    n.nspname , c.relname,
    pg_stat_get_tuples_deleted(x.pcrelid) as deleted,
pg_stat_get_tuples_updated(x.pcrelid) as updated
FROM pg_class c
INNER JOIN pg_namespace n ON n.oid = c.relnamespace
INNER JOIN pgxc_class x ON x.pcrelid = c.oid
WHERE c.relkind = 'r' and c.relname='tablename' ;
```

8.22 执行业务报错“Can't fit xid into page”

问题现象

场景一：执行VACUUM FULL时报错“Can't fit xid into page, now xid is 34181619720, base is 29832807366, min is 3, max is 3.”。

场景二：其他非VACUUM FULL操作，例如业务场景中给用户赋予函数的操作权限时，报错信息“Can't fit xid into page. relation "pg_proc", now xid is 34181619720, base is 29832807366, min is 3, max is 3.”。

原因分析

系统中存在老事务导致上述报错。

处理方法

场景一处理步骤：

步骤1 排查是否存在老事务。

```
SELECT * FROM pgxc_gtm_snapshot_status();
    xmin    |    xmax     |      csn     | oldestxmin   |   xcnt | running_xids
------------+-------------+--------------+--------------+--------+--------------
 34730350588 | 34730350588 |  34730350553 | 34730350553  |    0
(1 row)
```

- 如果查询结果中oldestxmin与报错中xid“34181619720”十分接近，且大于base+min和base+max, 说明系统中老事务不会影响freeze作业，可直接执行**步骤4**。
- 如果查询结果中oldestxmin小于base+min，且小很多，说明系统中存在老事务，且导致vacuum freeze执行未产生作用，需继续执行**步骤2**。

步骤2 使用如下命令查询集群中老事务信息。

```
SELECT * FROM pgxc_running_xacts where xmin::text::bigint < $base+$min and xmin::text::bigint > 0;
SELECT * FROM pgxc_running_xacts where gxid::text::bigint < $base+$min and gxid::text::bigint > 0;
```

步骤3 通过pgxc_stat_activity视图查询**步骤2**中的业务，确认后执行如下命令终止对应的线程。

```
SELECT pg_terminate_backend(pid) FROM pgxc_running_xacts where xmin::text::bigint <$base+$min  and xmin::text::bigint > 0;
```

说明

pgxc_running_xacts只能查询CN上的活跃事务。如果报错的是DN，需要到对应DN上使用pg_running_xacts视图进行查询。

步骤4 对报错表执行VACUUM FULL FREEZE。

```
VACUUM FULL FREEZE table_name;
```

步骤5 登录GaussDB(DWS)管理控制台，查看vacuum_freeze_min_age参数，如果设置值为50亿，则按照以下方式重新设置为20亿：

在集群列表中找到所需集群，单击集群名称，进入“集群详情”页面。单击“参数修改”页签，修改vacuum_freeze_min_age参数值，然后单击“保存”。

----结束

场景二处理步骤：

非VACUUM FULL操作中的报错信息，可以确认系统中存在老事务，按照场景一中的**步骤2**和**步骤3**清理掉后即可，无需再继续执行VACUUM FULL FREEZE。

8.23 执行业务报错：unable to get a stable set of rows in the source table

问题现象

MERGE INTO的作用是将源表内容根据匹配条件对目标表做更新或插入，当目标表匹配到多行满足条件时，GaussDB(DWS)有以下两种行为：

1. 业务报错：unable to get a stable set of rows in the source table.
2. 随机匹配一行数据，可能会导致实际与预期不符。

原因分析

进行MERGE INTO操作对目标表做更新或插入，目标表匹配到多行满足条件时出现该报错。

处理方法

这两种行为由参数behavior_compat_options控制，当参数behavior_compat_options缺省的情况下，匹配到多行会报错，如果behavior_compat_options设置了merge_update_multi参数项，这种情况下不会报错，而是会随机匹配一行数据。

因此，当出现MERGE INTO的结果与预期不符时，需查看该参数是否被设置，同时排查是否匹配了多行数据，并修改业务逻辑。

8.24 DWS 元数据不一致-分区索引异常

问题现象

某局点查看表定义报错：“The local index xxx on the partition xxx not exist.”。

原因分析

该问题主表索引无损坏，报错原因是由于单个分区索引记录在系统表pg_partition中与其他系统表不一致，未找到对应分区索引信息。

问题复现

1. 创建分区表a_0317，含p1，p2两个分区。
```
CREATE TABLE a_0317(a int) partition by range(a) (partition p1 values less than (4), partition p2 values less than (8));
```
2. 创建主表与分区索引。
```
CREATE INDEX a_0317_index on a_0317(a) local (partition p1_index, partition p2_inde);
```
3. 查看分区表分区索引信息如下：
   a. 查看主表索引信息。
```
SELECT oid,* FROM pg_class where relname ='a_0317_index';
  oid   |    relname    | relnamespace | reltype | reloftype | relowner | relam | relfilenode |
reltablespace | relpages | reltuples | relallvisible | reltoastrelid | reltoastidxid | reldeltarelid | reld
eltaidx | relcudescrelid | relcudescidx | relhasindex | relisshared | relpersistence | relkind | relnatts
| relchecks | relhasoids | relhaspkey | relhasrules | relhastriggers | relhassubclass | relcmprs
 | relhasclusterkey | relrowmovement | parttype | relfrozenxid | relacl | reloptions | relreplident |
relfrozenxid64
--------+--------------+--------------+---------+-----------+----------+-------+-------------
+---------------+----------+-----------+--------------+---------------+--------------+---------------
+-----
--------+----------------+--------------+-------------+-------------+----------------+---------+----------
+-----------+------------+------------+-------------+----------------+----------------+---------
-+------------------+----------------+----------+--------------+--------+------------+--------------
+----------------
 241487 | a_0317_index |         2200 |       0 |         0 |    16393 |   403 |      241487 |             0
|       0 |         0 |           0 |           0 |           0 |           0 |
     0 |           0 |           0 | f          | f          | p           | i      |       1 |         0 | f         |
f         | f          | f           | f           | 0
 | f             | f           | p       | 0          |      |          | n           |             0
(1 row)
```
   b. 根据主表索引信息查看分区索引信息。
```
SELECT * FROM pg_partition where parentid= 241487;
 relname  | parttype | parentid | rangenum | intervalnum | partstrategy | relfilenode |
reltablespace | relpages | reltuples | relallvisible | reltoastrelid | reltoastidxid | indextblid |
indisusable |
reldeltarelid | reldeltaidx | relcudescrelid | relcudescidx | relfrozenxid | intspnum | partkey |
intervaltablespace | interval | boundaries | transit | reloptions | relfrozenxid64
----------+----------+----------+----------+-------------+--------------+-------------+---------------
+----------+-----------+---------------+--------------+---------------+------------+-------------+-
--------------+-------------+----------------+--------------+--------------+----------+---------
+--------------------+----------+------------+---------+------------+----------------
 p1_index | x       |   241487 |       0 |          0 | n           |     241488 |            0 |       0 |
0 |           0 |           0 |           0 |           0 |    241485 | t          |
          0 |          0 |           0 |          0 | 0          |        |        |                |
|          |       |          |           0
 p2_inde  | x       |   241487 |       0 |          0 | n           |     241489 |            0 |       0 |
0 |           0 |           0 |           0 |           0 |    241486 | t          |
          0 |          0 |           0 |          0 | 0          |        |        |                |
|          |       |          |           0
(2 rows)
```
4. 连接CN开启读写事务，从pg_partition系统表删除p1分区的索引信息。
```
START TRANSACTION read write;
DELETE from pg_partition where relname = 'p1_index';
```
5. 查看表定义报错与现场报错相同，问题复现：
```
\d+ a_0317
ERROR:  The local index 700633 on the partition 700647 not exist.CONTEXT:  referenced column: pg_get_indexdef
```

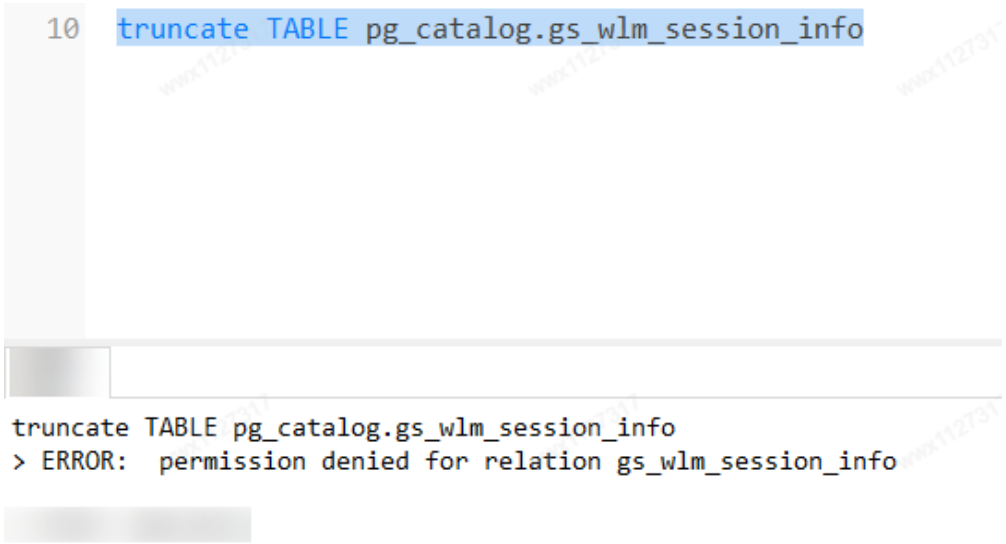

处理方法

1. 删除该表索引信息。
```
DROP INDEX a_0317_index;
```
2. 对该表索引进行重建。
```
CREATE INDEX a_0317_index on a_0317(a) local (partition p1_index, partition p2_inde);
```
3. 查看表定义无报错。
```
\d+ a_0317
                        Table "public.a_0317"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 a      | integer |           | plain   |              |
Indexes:
    "a_0317_index" btree (a) LOCAL(PARTITION p1_index, PARTITION p2_inde)  TABLESPACE
pg_default
Range partition by(a)
Number of partition: 2 (View pg_partition to check each partition range.)
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no
```

8.25 对系统表 gs_wlm_session_info 执行 TRUNCATE 命令报错

问题现象

清理磁盘空间，因系统表pg_catalog.gs_wlm_session_info较大（有20G），在不需要查询历史sql语句的前提下，对此系统表执行TRUNCATE命令，执行时报错“permission denied for relation gs_wlm_session_info”。



原因分析

对系统表执行TRUNCATE命令是版本8.0.x及以上集群才支持的，低版本集群不支持TRUNCATE系统表。

集群版本可登录GaussDB(DWS) 管理控制台，在“集群管理”页面进入对应集群的“集群详情”页面进行查看。

处理方法

- 8.0以下版本集群清理系统表需要先执行DELETE FROM，再执行VACUUM FULL。

  此处仅以gs_wlm_session_info系统表为例：

  ```
  DELETE FROM pg_catalog.gs_wlm_session_info;
  VACUUM FULL pg_catalog.gs_wlm_session_info;
  ```

- 8.0及以上版本集群可执行以下命令清理系统表：

  ```
  TRUNCATE TABLE dbms_om.gs_wlm_session_info;
  ```

  说明

  此系统表的schema是dbms_om。

8.26 分区表插入数据报错：inserted partition key does not map to any table partition

问题现象

给范围分区表插入数据报错：inserted partition key does not map to any table partition。

```
CREATE TABLE startend_pt (c1 INT, c2 INT)
DISTRIBUTE BY HASH (c1)
PARTITION BY RANGE (c2) (
    PARTITION p1 START(1) END(1000) EVERY(200) ,
    PARTITION p2 END(2000),
    PARTITION p3 START(2000) END(2500) ,
    PARTITION p4 START(2500),
    PARTITION p5 START(3000) END(5000) EVERY(1000)
);
SELECT partition_name,high_value FROM dba_tab_partitions WHERE table_name='startend_pt';
 partition_name | high_value
----------------+------------
 p1_0           | 1
 p1_1           | 201
 p1_2           | 401
 p1_3           | 601
 p1_4           | 801
 p1_5           | 1000
 p2             | 2000
 p3             | 2500
 p4             | 3000
 p5_1           | 4000
 p5_2           | 5000
(11 rows)

INSERT INTO startend_pt VALUES (1,5001);
ERROR:  dn_6003_6004: inserted partition key does not map to any table partition
```

原因分析

范围分区是根据表的一列或者多列，将要插入表的数据分为若干个范围，这些范围在不同的分区里没有重叠。划分好分区后，根据分区键值将数据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。

示例中的分区表tpcds.startend_pt是以c2列为partition_key，将插入表的数据分为5个没有重叠的分区。而插入的数据中，c2列对应的数据5001已超过了分区表中划分的分区范围（即5001>5000），因此报错。

处理方法

应根据数据实际情况规划分区，以保证插入的数据都在规划好的分区中。

如果已规划的分区无法满足实际应用条件，可以增加分区后再插入数据。针对上述案例，可以增加分区c2，分区范围介于5000和MAXVALUE之间：

```
ALTER TABLE startend_pt ADD PARTITION P6 VALUES LESS THAN (MAXVALUE);
SELECT partition_name,high_value FROM dba_tab_partitions WHERE table_name='startend_pt';
 partition_name | high_value
----------------+------------
 p1_0           | 1
 p1_1           | 201
 p1_2           | 401
 p1_3           | 601
 p1_4           | 801
 p1_5           | 1000
 p2             | 2000
 p3             | 2500
 p4             | 3000
 p5_1           | 4000
 p5_2           | 5000
 p6             | MAXVALUE
(12 rows)

INSERT INTO startend_pt VALUES (1,5001);
SELECT * FROM startend_pt;
 c1 |  c2
----+------
  1 | 5001
(1 row)
```

8.27 范围分区表添加新分区报错 upper boundary of adding partition MUST overtop last existing partition

问题现象

创建范围分区表后增加新的分区，使用ALTER TABLE ADD PARTITION语句报错upper boundary of adding partition MUST overtop last existing partition。

```
——创建范围分区表studentinfo
CREATE TABLE studentinfo (stuno smallint, sname varchar(20), score varchar(20), examate timestamp)
PARTITION BY RANGE (examate) (
   PARTITION p1 VALUES LESS THAN ('2022-10-10 00:00:00+08'),
   PARTITION p2 VALUES LESS THAN ('2022-10-11 00:00:00+08'),
   PARTITION p3 VALUES LESS THAN ('2022-10-12 00:00:00+08'),
   PARTITION p4 VALUES LESS THAN (maxvalue)
);
——添加边界值为2022-10-9 00:00:00+08的分区p0
ALTER TABLE studentinfo ADD PARTITION p0 values less than ('2022-10-9 00:00:00+08');
ERROR:  the boundary of partition "p0" is less than previous partition's boundary
——添加边界值为2022-10-13 00:00:00+08的分区p5
ALTER TABLE studentinfo ADD PARTITION p5 values less than ('2022-10-13 00:00:00+08');
ERROR:  the boundary of partition "p5" is equal to previous partition's boundary
```

原因分析

增加分区时需同时满足以下条件：

- 新增分区名不能与已有分区名相同。
- 新增分区的边界值必须大于最后一个分区的上边界。

- 新增分区的边界值要和分区表的分区键的类型一致。

已有分区p1的边界为(-∞，20221010)，而新增分区p0的上边界为20221009，落在分区p1内；已有分区p4的边界为[20221012,+∞)，而新增分区p5的上界为20221013，落在分区p4内。新增分区p0、p5不满足使用ADD PARTITION增加分区的条件，因此执行新增分区语句报错。

处理方法

使用ALTER TABLE SPLIT PARTITION分割已有分区，也能达到新增分区的目的。同样， SPLIT PARTITION的新分区名称也不能与已有分区相同。

使用split子句分割p4分区[20221012,+∞)为p4a分区范围为[20221012,20221013)和p4b分区范围为[20221013,+∞)。

```
——SPLIT PARTITION分割前的分区
SELECT relname, boundaries FROM pg_partition p where p.parentid='studentinfo'::regclass ORDER BY 1;
   relname   |       boundaries
-------------+-------------------------
 p1          | {"2022-10-10 00:00:00"}
 p2          | {"2022-10-11 00:00:00"}
 p3          | {"2022-10-12 00:00:00"}
 p4          | {NULL}
 studentinfo |
(5 rows)

ALTER TABLE studentinfo SPLIT PARTITION p1 AT('2022-10-09 00:00:00+08') INTO (PARTITION
P1a,PARTITION P1b);
ALTER TABLE studentinfo SPLIT PARTITION p4 AT('2022-10-13 00:00:00+08') INTO (PARTITION
P4a,PARTITION P4b);

——执行SPLIT PARTITION分割后的分区
SELECT relname, boundaries FROM pg_partition p where p.parentid='studentinfo'::regclass ORDER BY 1;
   relname   |       boundaries
-------------+-------------------------
 p1a         | {"2022-10-09 00:00:00"}
 p1b         | {"2022-10-10 00:00:00"}
 p2          | {"2022-10-11 00:00:00"}
 p3          | {"2022-10-12 00:00:00"}
 p4a         | {"2022-10-13 00:00:00"}
 p4b         | {NULL}
 studentinfo |
(7 rows)
```

如果对分区名称有要求，可以在分割后再使用rename partition统一分区名。

```
ALTER TABLE studentinfo RENAME PARTITION p1a to p0;
```

8.28 查询表报错：missing chunk number %d for toast value %u in pg_toast_XXXX

问题现象

查询表报错：missing chunk number %d for toast value %u in pg_toast_XXXX。

原因分析

表关联的toast表的data发生损坏。

toast是The OverSized Attribute Storage Technique（超尺寸字段存储技术）的缩写，是超长字段在GaussDB(DWS)的一种存储方式。当某表中有超长字段的时候，那

么该表会有与之相关联的toast表。根据toast表的命名规则，假设存在表名为test的OID为2619，那么如果存在与之相关联的toast表，则toast表名为pg_toast_2619。此报错中pg_toast_2619非固定表名，可根据实际报错对pg_toast_2619进行替换。

处理方法

步骤1 通过toast的OID（示例中为2619，来源于报错信息的pg_toast_2619）查询出哪张表发生了损坏：

```
SELECT 2619::regclass;
   regclass
--------------
 pg_statistic
(1 row)
```

步骤2 对已定位的损坏表（**步骤1**中查询得到的表pg_statistic），执行REINDEX和VACUUM ANALYZE操作。显示REINDEX/VACUUM，表示修复完成。如果修复过程中出现报错，请继续执行**步骤3**。

```
REINDEX table pg_toast.pg_toast_2619;
REINDEX table pg_statistic;
VACUUM ANALYZE pg_statistic;
```

步骤3 执行以下的命令定位该表中损坏的数据行。

```
DO $$
declare
 v_rec record;
BEGIN
for v_rec in SELECT * FROM pg_statistic loop
        raise notice 'Parameter is: %', v_rec.ctid;
raise notice 'Parameter is: %', v_rec;
end loop;
END;
$$
LANGUAGE plpgsql;
NOTICE:  00000: Parameter is: (46,9)
ERROR:  XX000: missing chunk number 0 for toast value 30982 in pg_toast_2619
CONTEXT:  PL/pgSQL function inline_code_block line 7 at RAISE
```

步骤4 将**步骤3**中定位的损坏数据行记录删除。

```
DELETE FROM pg_statistic WHERE ctid ='(46,9)';
```

步骤5 重复执行**步骤3**、**步骤4**，直到全部有问题的数据记录被删除。

步骤6 损坏的数据行被删除后，执行**步骤2**中的REINDEX和VACUUM ANALYZE操作对该表重新进行修复。

----结束

8.29 向表中插入数据报错：duplicate key value violates unique constraint "%s"

问题现象

向表中插入数据报错：duplicate key value violates unique constraint "%s"。

```
CREATE TABLE films (
code        char(5) PRIMARY KEY,
title       varchar(40) NOT NULL,
did         integer NOT NULL,
date_prod   date,
```

```
kind        varchar(10),
len         interval hour to minute
);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "films_pkey" for table "films"
CREATE TABLE

INSERT INTO films VALUES ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films VALUES ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
ERROR:  dn_6003_6004: duplicate key value violates unique constraint "films_pkey"
DETAIL:  Key (code)=(UA502) already exists.
```

原因分析

创建表films时带有主键约束，使用PRIMARY KEY声明主键code，即表中的code字段只能包含唯一的非NULL值。同时主键会为表films创建索引films_pkey。

插入表中的数据code字段值UA502与表中已存在的中code字段值重复，因此插入数据报错。

处理方法

- 方法一：检查数据冲突，修改插入数据。例如，修改示例重复字段UA502为UA509。

  ```
  INSERT INTO films VALUES ('UA509', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
  INSERT 0 1
  ```

- 方法二：删除表films主键约束。

  ```
  ALTER TABLE films DROP CONSTRAINT films_pkey;
  ALTER TABLE
  INSERT INTO films VALUES ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
  INSERT 0 1
  ```

8.30 执行业务报错 could not determine which collation to use for string hashing

问题现象

执行SELECT查询时报错could not determine which collation to use for string hashing。

```
CREATE TABLE t(a text collate "C", b text collate case_insensitive);
INSERT INTO t VALUES('Hello','world');
——计算ifnull(a,b)的值的哈希值
SELECT hashtext(ifnull(a,b)) FROM t;
ERROR:  dn_6005_6006: could not determine which collation to use for string hashing.
HINT:  Use the COLLATE clause to set the collation explicitly.
```

说明

hashtext函数用于计算适当数据类型的值的哈希值。此处仅用作示例说明出现collate冲突时应该如何解决。

原因分析

表t中有两个字段，且两个字段的排序规则不同，字段a的排序规则为C（安装数据库时的默认排序规则），字段b的排序规则为case_insensitive（不区分大小写），SELECT语句中表达式hashtext(ifnull(a,b))存在多个collation造成冲突，执行报错。

处理方法

当字符串表达式中collation有多个时，可手动指定COLLATE collation_name。

执行SELECT时，指定表达式ifnull(a,b)的排序规则为C或者case_insensitive。

```
SELECT hashtext(ifnull(a,b) collate "C") FROM t;
 hashtext
-----------
 820977155
(1 row)

SELECT hashtext(ifnull(a,b) collate case_insensitive) FROM t;
 hashtext
-----------
 238052143
(1 row)
```

扩展

除上述场景会出现存在多个collation造成冲突，以下两种场景也会出现多个collation，报错不同但解决方法相同。

- 场景一

  SELECT语句中比较表达式a=b存在多个collation造成冲突，执行报错could not determine which collation to use for string comparison。

  ```
  SELECT a=b FROM t;
  ERROR:  dn_6001_6002: could not determine which collation to use for string comparison
  HINT:  Use the COLLATE clause to set the collation explicitly.
  ```

  执行SELECT时，指定表达式a=b的排序规则为case_insensitive。

  ```
   SELECT a=b COLLATE case_insensitive FROM t;
   ?column?
  ----------
   f
  (1 row)
  ```

- 场景二

  SELECT语句中表达式instr(a,b)存在多个collation造成冲突，执行报错could not determine which collation to use for string searching。

  ```
  SELECT instr(a,b) FROM t;
  ERROR:  dn_6005_6006: could not determine which collation to use for string searching
  HINT:  Use the COLLATE clause to set the collation explicitly.
  ```

  执行SELECT时，指定字段a的排序规则为case_insensitive或者指定字段b的排序规则为C来保证字段排序规则的统一。

  ```
  SELECT instr(a collate case_insensitive,b) FROM t;
   instr
  -------
      0
  (1 row)

  SELECT instr(a,b collate "C") FROM t;
   instr
  -------
      0
  (1 row)
  ```


8.31 使用 GaussDB(DWS) 的 ODBC 驱动，SQL 查询结果中字符类型的字段内容会被截断

问题现象

使用GaussDB(DWS)的ODBC驱动，SQL查询结果中字符类型的字段内容会被截断，需通过SQL语法CAST BYTEA转成二进制才能完整取出字段信息。但是，同样的程序连接ORACLE、SQL SERVER却没有问题。

原因分析

ODBC客户端设置了max varchar=255，导致超过255的字段被截断。

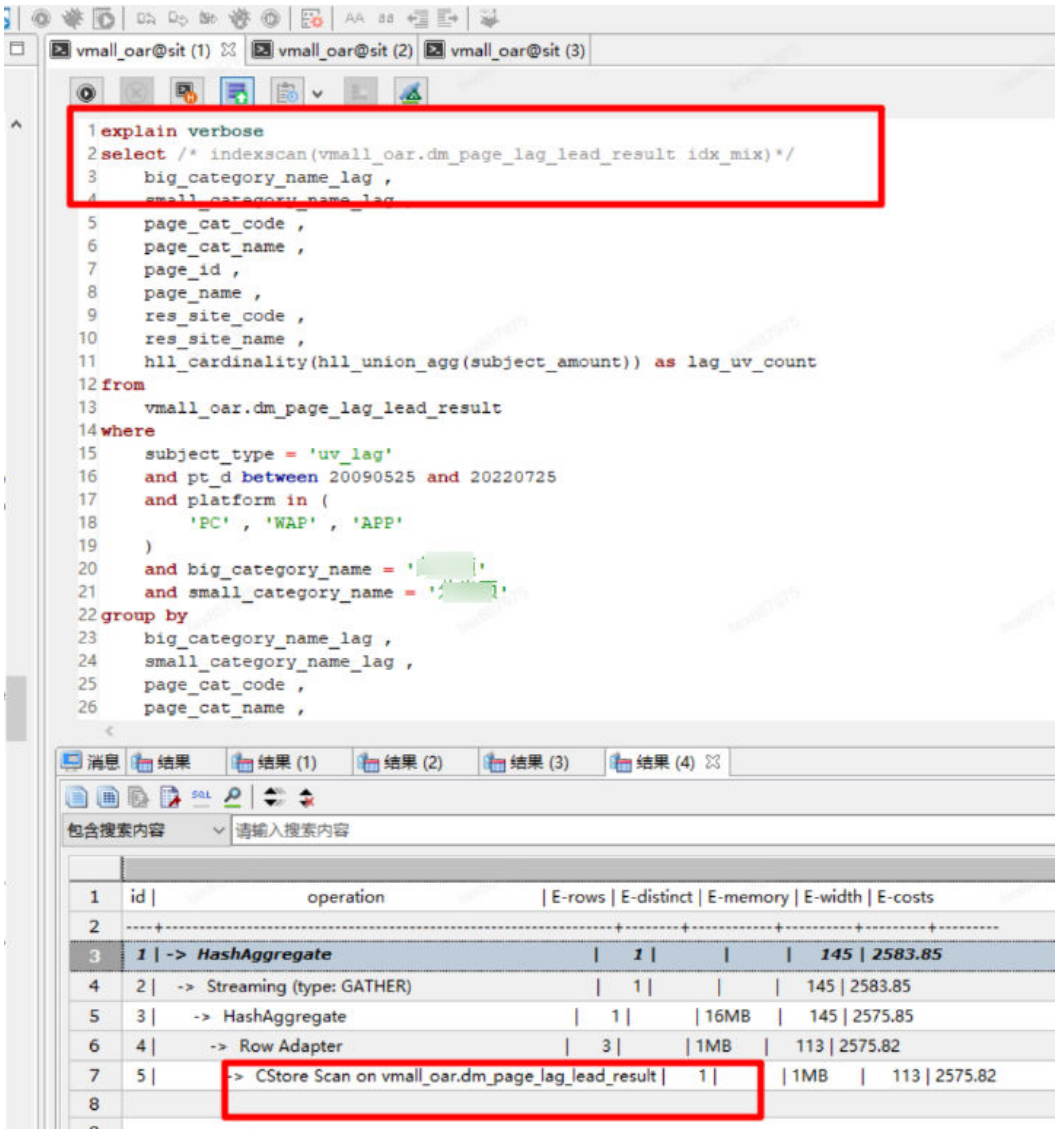
处理方法

在ODBC客户端，增大max varchar的值即可。

8.32 执行 Plan Hint 的 Scan 方式不生效

问题现象

DWS中指定了Plan Hint的scan方式，但是并未生效。



原因分析

Plan Hint语法使用错误。Plan Hint的Scan语法应在SELECT语句中增加“**/*+ indexscan(table_name index_name) */**”，**明显语句中缺少了“+”号** 。

Plan Hint的语法参见**Plan Hint调优概述**。

处理方法

Plan Hint的Scan语法应在SELECT语句中增加**/*+ indexscan(table_name index_name) */** 格式。

```
explain verbose
select /*+ indexscan (vmall_oar.dm_page_lag_lead_result idx_mix)*/
.....
```

8.33 数据类型转换出现报错：invalid input syntax for xxx

问题现象

某张表的某个字段类型为varchar(20)，数据为5.0，在使用cast(xxx as integer)转换成整数型时报错：invalid input syntax for integer 5.0。

```
gaussdb=> SELECT CAST(price AS integer) FROM product;
ERROR:  dn_6005_6006: invalid input syntax for integer: "5.0"
CONTEXT:  referenced column: price
```

原因分析

在SQL语句执行过程中，若遇到类似invalid input syntax for integer/bigint/numeric等报错的问题，基本都是数据类型之间转换导致，如字符a或空格转换成integer、bigint类型时触发。

在熟悉GaussDB(DWS)的数值类型和字符类型后，可以避免数据类型的使用问题，参考**数据类型**章节。

处理方法

以上述问题现象中的“字符串类型varchar直接转换为整数型integer”报错为例，可以先将字段类型修改为decimal（任意精度型）再进行转换来处理。

具体的操作步骤如下：

步骤1 假设报错表名为product，表定义如下：

```
SELECT * FROM PG_GET_TABLEDEF('product');
```

```
gaussdb=> SELECT * FROM PG_GET_TABLEDEF('product');
            pg_get_tabledef
---------------------------------------
 SET search_path = public;             +
 CREATE  TABLE product (               +
         id integer,                   +
         price character varying(20)   +
 )                                     +
 WITH (orientation=row, compression=no)+
 DISTRIBUTE BY ROUNDROBIN              +
 TO GROUP group_version1;
(1 row)
```

步骤2 将查询结果转换为整数型。

```
SELECT CAST(price AS integer) FROM product;
```

出现如下报错：

```
gaussdb=> SELECT CAST(price AS integer) FROM product;
ERROR:  dn_6005_6006: invalid input syntax for integer: "5.0"
CONTEXT:  referenced column: price
```

步骤3 修改表字段的数据类型为decimal。

```
ALTER TABLE product ALTER COLUMN price TYPE decimal(10,1);
```

步骤4 成功转换字段为整数型。

```
SELECT CAST(price AS integer) FROM product;
```

```
gaussdb=> SELECT CAST(price AS integer) FROM product;
 price
-------
     5
     6
(2 rows)
```

----结束

8.34 业务报错：UNION types %s and %s cannot be matched

问题现象

执行union语句时报错：UNION types %s and %s cannot be matched。

原因分析

该报错的原因是UNION的分支中，相同位置的输出列格式类型不同导致。

处理方法

故障构造场景

步骤1 使用客户端连接DWS数据库。

步骤2 执行以下SQL语句。

```
CREATE TABLE t1(a int, b timestamp);
CREATE TABLE
CREATE TABLE t2(a int, b text);
CREATE TABLE
INSERT INTO t1 select 1, current_date;
INSERT 0 1
INSERT INTO t2 select 1, current_date;
INSERT 0 1
SELECT * FROM t1 UNION SELECT * FROM t2;
ERROR:  UNION types timestamp without time zone and text cannot be matched
LINE 1: SELECT * FROM t1 UNION SELECT * FROM t2;
                                ^
```

----结束

解决办法

步骤1 示例中，t1表和t2表在b列上类型不同，导致在UNION操作时出现类型不匹配的报错，应保证UNION各分支相同位置的输出列类型匹配。

说明

t2表b列是text类型，插入的数据是current_date，在插入时发生了隐式类型转换，所以插入不报错；但是在查询时，不会自动进行隐式转换，因此会报错。

解决以上问题，需保证UNION各分支的输出列类型匹配，不满足要求时可以对输出列强制类型转化。

```
SELECT a,b::text FROM t1 UNION SELECT a,b FROM t2;
 a |          b
---+---------------------
 1 | 2023-02-16
 1 | 2023-02-16 00:00:00
(2 rows)
```

----结束

8.35 更新报错 ERROR：Non-deterministic UPDATE

问题现象

执行update语句报错ERROR：Non-deterministic UPDATE。

```
CREATE TABLE public.t1(a int, b int) WITH(orientation = column);
CREATE TABLE

CREATE TABLE public.t2(a int, b int) WITH(orientation = column);
CREATE TABLE

INSERT INTO public.t1 VALUES (1, 1);
INSERT INTO public.t2 VALUES (1, 1),(1, 2);

UPDATE t1 SET t1.b = t2.b FROM t2 WHERE t1.a = t2.a;
ERROR: Non-deterministic
UPDATEDETAIL:  multiple updates to a row by a single query for column store table.
```

原因分析

当一条SQL语句中同一个元组被多次更新，执行便会报错ERROR：Non-deterministic UPDATE。

可以看到更新操作分成两步执行：

1. 通过关联操作查找满足更新条件的元组。
2. 执行更新操作。

针对上述案例，对于表public.t1中元组 (1, 1)来说，表public.t2中满足更新条件t1.a = t2.a的记录有两条，分别为(1, 1)， (1, 2)；按照执行器逻辑表t2的元组 (1, 1)需要被更新两次，那么就可能出现两种情况：

1. 表public.t1和表public.t2关联时先命中(1, 1)，再命中(1, 2)，这时public.t1的元组(1, 1)，先被更新为(1,1)，再被更新为(1,2)，最终结果为(1, 2)。
2. 表public.t1和表public.t2关联时先命中(1, 2)，再命中(1, 1)，这时public.t1的元组(1, 1)，先被更新为(1,2)，再被更新为(1,1)，最终结果为(1, 1)。

实际执行过程中public.t2表输出结果集的顺序会影响update语句的最终输出结果（实际业务中表public.t2的位置可能是一个非常复杂的子查询），导致了**update语句执行结果的随机性**，而这个实际业务中是无法接受的。

解决方案

建议根据业务实际情况调整update语句。比如分析public.t2的字段含义，确定更新的目标字段。针对上述案例，如果期望在a值相等的情况下，把public.t1中字段b更新为public.t2中的最大值，那么可以修改为如下逻辑：


```
UPDATE t1 SET t1.b = t2.b_max FROM (SELECT a, max(b) AS b_max FROM t2 GROUP BY a) t2 WHERE t1.a
= t2.a;
UPDATE 1
SELECT * FROM public.t1;
 a | b
---+---
 1 | 2
(1 row)
```

8.36 插入数据报错：null value in column '%s' violates not-null constraint

问题现象

向表中插入数据报错：null value in column '%s' violates not-null constraint，此处s%指报错的列（字段）名。

```
CREATE TABLE t1(a int, b int not null);

INSERT INTO t1 VALUES (1);
ERROR:  dn_6001_6002: null value in column "b" violates not-null constraint
```

原因分析

针对上述案例，表t1中的字段b在建表时，设置了非空（not null）约束，那么字段b中不能有空值。而插入数据时b列为空，则执行报错。

解决方案

针对上述案例，有两种解决方案：

- 方案一：使用ALTER TABLE删除字段b的非空（not null）约束
  ```
  ALTER TABLE t1 ALTER COLUMN b DROP NOT NULL;
  ALTER TABLE

  INSERT INTO t1 VALUES (1);
  INSERT 0 1
  ```
- 方案二：保持字段b的非空（not null）约束，字段b不再插入空值

在实际业务中，可根据实际情况选择解决方案。

8.37 业务报错：unable to get a stable set of rows in the source table

问题现象

执行MERGE INTO将源表内容根据匹配条件对目标表做更新报错unable to get a stable set of rows in the source table。

现有目标表products和源表newproducts，以源表newproducts中product_id为1502为匹配条件，对目标表进行更新报错：

```
CREATE TABLE products (product_id INTEGER,product_name VARCHAR2(60),category VARCHAR2(60));

INSERT INTO products VALUES (1501, 'vivitar 35mm', 'electrncs'),(1502, 'olympus is50', 'electrncs'),(1600,
```



```
'play gym', 'toys');

CREATE TABLE newproducts (product_id INTEGER,product_name VARCHAR2(60),category VARCHAR2(60));

INSERT INTO newproducts VALUES (1502, 'olympus camera', 'electrncs'),(1600, 'lamaze', 'toys'),(1502,
'skateboard', 'toy');

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 np.product_id = 1502;
ERROR:  dn_6003_6004: unable to get a stable set of rows in the source tables
```

原因分析

源表newproducts中product_id为1502的数据有两条，且参数behavior_compat_options缺省，因此MERGE INTO时匹配到多条数据报错。

MERGE INTO的作用是将源表内容根据匹配条件对目标表做更新或插入，当目标表匹配到多行满足条件时，GaussDB(DWS)有以下两种行为：

1. 业务报错：unable to get a stable set of rows in the source table。
2. 随机匹配一行数据，可能会导致实际与预期不符。

这两种行为由参数behavior_compat_options控制，当参数behavior_compat_options缺省的情况下，匹配到多行会报错，如果behavior_compat_options被设置为merge_update_multi，则不会报错，而是会随机匹配一行数据。

因此，当出现merge into的结果与预期不符的情况时，需查看该参数是否被设置，同时排查是否匹配了多行数据，并根据实际情况修改业务逻辑。

解决方案

- 方案一：设置参数behavior_compat_options为merge_update_multi。

  当目标表匹配到多行满足条件时，该方案不会报错，而是会随机匹配一行数据，有数据遗漏风险。

  ```
  SET behavior_compat_options=merge_update_multi;

  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 np.product_id =
  1502;
  MERGE 1

  SELECT * FROM products ;
   product_id |  product_name  | category
  ------------+----------------+-----------
        1501 | vivitar 35mm   | electrncs
        1502 | olympus camera | electrncs
        1600 | play gym       | toys
  (3 rows)
  ```

- 方案二：修改MERGE INTO匹配条件。

  尽可能选择筛选结果唯一的表达式为匹配条件。

  ```
  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 np.product_id !=
  1502;
  ```



```
MERGE 1

SELECT * FROM products;
 product_id |  product_name  | category
------------+----------------+-----------
       1501 | vivitar 35mm   | electrncs
       1502 | olympus camera | electrncs
       1600 | lamaze         | toys
(3 rows)
```

8.38 Oracle/TD/MySQL 兼容模式下查询结果不一致

问题现象

某业务场景中有两套集群环境，在数据量一致的情况下，对比发现两套环境执行相同的SQL语句但执行结果不同。

步骤1 该场景中使用的语法可以简化为以下逻辑：

```
CREATE TABLE test (a text, b int);
INSERT INTO test values('', 1);
INSERT INTO test values(null, 1);
SELECT count(*) FROM test a, test b WHERE a.a = b.a;
```

步骤2 两套环境中的执行结果分别如下：

结果1：

```
demo_db1=> SELECT count(*) FROM test a, test b WHERE a.a = b.a;
 count
-------
     0
(1 row)
```

结果2：

```
demo_db2=> SELECT count(*) FROM test a, test b WHERE a.a = b.a;
 count
-------
     1
(1 row)
```

----结束

原因分析

GaussDB(DWS)支持Oracle、Teradata和MySQL数据库兼容模式。

在TD/MySQL兼容模式下，空和NULL是不相等的，在ORA兼容模式下，空和NULL是相等的。因此上述场景可能是因为两个环境中数据库的兼容性模式设置不一致导致。

可通过查询PG_DATABASE系统表确认数据库的兼容模式：

```
SELECT datname, datcompatibility FROM pg_database;
```

处理方法

数据库的兼容模式在CREATE DATABASE时由**DBCOMPATIBILITY**参数指定。

- **DBCOMPATIBILITY [=] compatibility_type**
  指定兼容的数据库的类型。

- 取值范围：ORA、TD、MySQL。分别表示兼容Oracle、Teradata和MySQL数据库。

  若创建数据库时不指定该参数，默认为ORA。

为解决DATABASE的兼容性模式问题，需要将两个数据库的兼容模式修改为一致。GaussDB(DWS)不支持ALTER方式修改已有数据库的兼容模式**DBCOMPATIBILITY**，只能通过新建数据库的方式来指定兼容模式。

```
CREATE DATABASE td_db DBCOMPATIBILITY ='TD';
CREATE DATABASE
```

GaussDB(DWS)不同兼容模式下Oracle、Teradata和MySQL语法行为会有一些差异，具体的差异内容可参考**Oracle、Teradata和MySQL语法兼容性差异**。