《数据库系统原理》课程实验指导

openGauss 事务管理

2023 年 9 月

目录

前 言

实验环境说明

本实验环境为 virtualBOX 虚拟机 openEuler20.03 系统上的 openGauss1.1.0/openGauss2.0.0 数据库和华为云 GaussDB(openGauss)数据库，实验数据采用 TPC-H 供应商和采购商数据库的八张表。

1 事务管理实验

1.1 实验目的

通过单事务、串行事务、并发事务实验，了解 openGauss 数据库中

1）事务组成方式和执行模式；

2）对单事务和串行事务的原子性保障机制；

3）基于锁和隔离级别的事务并发控制和对并发事务的一致性、独立性保障机制。

1.2 实验原理/背景

1.2.1 事务两种运行模式

事务通常以 2 种模式运行，具体如下所述。

1. 自动提交事务

每一条单独的 SQL 语句都在其执行完成后进行自动提交事务，即执行 SQL 语句后就会马上自动隐式执行 COMMIT 操作。如果出现错误，则进行事务回滚至之前状态。

openGauss 中默认开启自动提交事务，现版本的 openGauss 数据库暂时不提供关闭自动提交的功能，具体如下：

1）查看参数，值为 on，自动提交开启

```
show autocommit;
```

```
 autocommit
------------
 on
(1 row)
```

2）关闭自动提交功能

```
set session autocommit=0;
```

```
-- 0 是关闭，1 是开启；
```

关闭失败，不再支持将自动提交设置为关闭

```
ERROR:  SET AUTOCOMMIT TO OFF is no longer supported
```

2. 显式事务

通过指定事务开始语句来显式开启事务来作为开始，并由以提交命令 COMMIT 或者回滚命令 ROLLBACK 来提交或者回滚事务作为结束的一段代码就是一个用户定义的显式事务。具体如下：

openGauss 以 START TRANSACTION | BEGIN TRANSACTION 作为开始，以 COMMIT 或者 ROLLBACK 或者 END 作为结束。

```
tpch=# START TRANSACTION;
START TRANSACTION
tpch=# COMMIT;
COMMIT
```

```
tpch=# BEGIN TRANSACTION;
BEGIN
tpch=# ROLLBACK;
ROLLBACK
```

1.2.2 保存点

在 openGauss 中，可以定义保存点 savepoint，用于实现事务的部分回滚。使用保存点的基本语法为：

```
SAVEPOINT identifier;
ROLLBACK TO identifier;
RELEASE SAVEPOINT identifier;
```

说明：

（1）SAVEPOINT identifier：针对特定时刻 T 的数据库状态/实例，创建名为 identifier 的回滚点/保存点；

（2）ROLLBACK TO identifier：回滚到指定名称为 identifier 的 SAVEPOINT；

（3）RELEASE SAVEPOINT identifier：对于不再需要的保存点 identifier，释放删除该保存点；

（4）如果回滚到 SAVEPOINT 语句返回以下错误，表示不存在具有指定名称的保存点：

```
ERROR:  no such savepoint
```

（5）执行事务控制语句 COMMIT 和 ROLLBACK，将删除当前事务的所有保存点。

1.2.3 事务并发访问问题

脏读（Dirty Read）:读到了其他事务未提交的数据，未提交意味着这些数据可能会回滚，也就是可能最终不会存到数据库中，也就是不存在的数据。

例如，事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据。

	Session A	Session B

1	start transaction;	start transaction;
2		update partsupp_1 set PS_AVIALQTY=9999 where PS_PARTKEY='2022' and PS_SUPPKEY='1526' ;
3	select PS_PARTKEY, PS_SUPPKEY, PS_AVIALQTY from partsupp_1 where PS_PARTKEY='2022' and PS_SUPPKEY='1526' ; //这里查询到了错误的数据 9999	
4	commit;	
5		rollback;

不可重复读（Non-Repeatable Read）:在同一事务内，不同的时刻读到的同一批数据不一样，可能会受到其他事务的影响，例如，在一个事务两次读取同一批数据期间，其他事务修改了这批数据并提交，导致该事务读取的同一批数据内容前后不一致。通常针对数据更新（UPDATE）操作。

例如，事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果不一致。

	Session A	Session B（为自动提交事务，每条语句自动 commit）
1	start transaction;	
2	select PS_AVIALQTY from partsupp_1 where PS_PARTKEY='2022' and PS_SUPPKEY='1526' ; //结果为 1	
3		update partsupp_1 set PS_AVIALQTY=9999 where PS_PARTKEY='2022' and PS_SUPPKEY='1526' ;

4	select PS_AVIALQTY from partsupp_1 where PS_PARTKEY='2022' and PS_SUPPKEY='1526' ; //此时查出来为 9999	
5		update partsupp_1 set PS_AVIALQTY=9998 where PS_PARTKEY='2022' and PS_SUPPKEY='1526' ;
6	select PS_AVIALQTY from partsupp_1 where PS_PARTKEY='2022' and PS_SUPPKEY='1526' ; //此时查出来为 9998	
7	commit;	

幻读（Phantom Read）:针对数据插入（INSERT）操作。假设事务A对某些行的内容作了更改，但是还未提交，此时事务B插入了与事务A更改前的记录相同的记录行，并且在事务A提交之前先提交了。而这时，在事务A中查询，会发现好像刚刚的更改对于某些数据未起作用，但其实是事务B刚插入进来的，让用户感觉出现了幻觉，称为幻读。

导致幻读的原因是一个事务执行两次查询，第二次结果集包含第一次中没有或某些行已经被删除的数据，造成两次结果不一致，只是另一个事务在这两次查询中间插入或删除了数据造成的。如下例所示。

	Session A	Session B（为自动提交事务，每条语句自动 commit）
1	start transaction;	
2	select PS_AVIALQTY from partsupp_1 where PS_PARTKEY='2022' and PS_SUPPKEY='2022' ; //结果为空	
3		insert into partsupp_1 values('2022','2022',0,'comment');
4	select PS_AVIALQTY	

	from partsupp_1 where PS_PARTKEY='2022' and PS_SUPPKEY='2022' ; //会查到上面插入的元组	
5	commit;	

丢失更新。两个事务同时更新关系表中一行数据，后提交（或撤销）的事务覆盖了之前事务提交的数据覆。丢失更新可分为第一类丢失更新和第二类丢失更新。

（1）第一类丢失更新。两个事务同时操作同一个数据时，第一个事务撤销时，将已经提交的第二个事务的更新数据覆盖了，造成第二个事务的数据丢失。

（2）第二类丢失更新.当两个事务同时操作同一个数据时，第一个事务将修改结果成功提交后，对第二个事务已经提交的修改结果进行了覆盖，造成第二个事务的数据丢失。

1.2.4 事务隔离级别

隔离级别定义

事务隔离级别定义了当一个事务与其它事务同时并发访问相同资源或数据时，为避免多个事务并发访问带来的副作用，必须与由其它事务间保持的资源或数据更改相隔离的程度。

多个事务并发执行必须同时满足 ACID 四个特性，即原子性、一致性、隔离性和持久性。当多个事务访问同一个数据项时，如果并发控制机制不当，可能会出现以下数据并发访问问题/副作用。

为了避免上述事务并发访问问题的出现，SQL 规范定义了四种事务隔离级别，分别允许不同的并发访问副作用。这四种事务的隔离级别如下：

读未提交（READ UNCOMMITTED）：

事务B可以读取事务A修改过但未提交的数据，即B在A执行commit操作之前读取了A产生的数据。此隔离级别可防止丢失更新，但可能导致脏读、不可重复读和幻读问题。一般很少使用此隔离级别。

在此隔离级别下当事务 A 写数据项 item 时，不允许另外一个事务 B 同时写 item，但允许事务 B 读数据 item。

读提交 （READ COMMITTED）：

事务B只能在事务A修改过数据项item并且已commit/提交后才能读取到事务A修改的item。此隔离级别下，未提交的写事务A将会禁止其他事务B访问数据项item，可有效防止脏读，但可能导致不可重复读和幻读。

可重复读 （REPEATABLE READ）：

事务B只能在事务A修改过数据并提交后，自己也提交事务后，才能读取到事务A修改的数据。可重复

读隔离级别解决了脏读和不可重复读的问题，确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。

可串行化（SERIALIZABLE）：

最高的事务隔离级别，在该级别下，事务串行化顺序执行，可以避免脏读、不可重复读与幻读。但是这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

较低的隔离级别可以增强许多用户同时访问数据的能力，但也增加了用户可能遇到的并发副作用（例如脏读或丢失更新）的数量。相反，较高的隔离级别减少了用户可能遇到的并发副作用的类型，但需要更多的系统资源，并增加了一个事务阻塞其他事务的可能性。应综合考虑应用程序的数据完整性要求与隔离级别的开销，选择相应的隔离级别。可串行化作为最高隔离级别，可以保证事务在每次重复读取操作时都能准确检索到相同的数据，但需要通过执行某种级别的加锁来完成此操作，而加锁可能会影响多用户系统中的其他用户。最低隔离级别（未提交读）可以检索其他事务已经修改、但未提交的数据。在未提交读中，所有并发副作用都可能发生，但因为没有读取加锁或版本控制，所以开销最少。

openGauss 的隔离级别

目前 openGauss 只支持两种隔离级别，READ COMMITTED 与 REPEATABLE READ。

READ COMMITTED：读已提交隔离级别，事务只能读到已提交的数据而不会读到未提交的数据，这是 openGauss 的默认隔离级别。

实际上，SELECT 查询会查看到在查询开始运行的瞬间该数据库的一个快照。不过，SELECT 能查看到其自身所在事务中先前更新的执行结果。即使先前更新尚未提交。请注意，在同一个事务里两个相邻的 SELECT 命令可能会查看到不同的快照，因为其它事务会在第一个 SELECT 执行期间提交。

因为在读已提交模式里，每个新的命令都是从一个新的快照开始的，而这个快照包含所有到该时刻为止已提交的事务，因此同一事务中后面的命令将看到任何已提交的其它事务的效果。这里关心的问题是在单个命令里是否看到数据库里绝对一致的视图。

读已提交模式提供的部分事务隔离对于许多应用而言是足够的，并且这个模式速度快，使用简单。不过，对于做复杂查询和更新的应用，可能需要保证数据库有比读已提交模式更加严格的一致性视图。

REPEATABLE READ：事务可重复读隔离级别，事务只能读到事务开始之前已提交的数据，不能读到未提交的数据以及事务执行期间其它并发事务提交的修改（但是，查询能查看到自身所在事务中先前更新的执行结果，即使先前更新尚未提交）。

这个级别和读已提交是不一样的，因为可重复读事务中的查询看到的是事务开始时的快照，不是该事务内部当前查询开始时的快照，就是说，单个事务内部的 select 命令总是查看到同样的数据，查看不到自身事务开始之后其他并发事务修改后提交的数据。使用该级别的应用必须准备好重试事务，因为可能会发生串行化失败。

上述两种隔离级别可以分别处理不同程度的数据并发访问问题/副作用，具体如下表所示：

事务隔离级别	脏读	不可重复读	幻读
读提交 (read-committed)	否	是	是
可重复读 (repeatable-read)	否	否	否

注意:

- 在 openGauss 中，目前，READ UNCOMMITTED 等价于 READ COMMITTED，都是读提交隔离级别，SERIALIZABLE 等价于 REPEATABLE READ，都是可重复读隔离级别。

- 由于openGauss采用MVCC机制（多版本并发控制），可以使用快照读，当数据项A写数据项item时，其它事务B也可以读item，B而不会被阻塞。

设置/查看隔离级别

设置隔离级别有两种方法:

(1) 在开启事务时设置隔离级别

```
START | BEGIN TRANSACTION ISOLATION LEVEL read committed | repeatable read;
```

开启事务时若不表明隔离级别，则为默认的 read committed 隔离级别

(2) 开启事务后，在事务内部设置隔离级别

```
SET TRANSACTION ISOLATION LEVEL read committed | repeatable read;
```

注意：在事务中第一个数据修改语句（SELECT，INSERT，DELETE，UPDATE，FETCH，COPY）执行之后，事务隔离级别就不能再次设置。

查看隔离级别有两种方法:

(1) 在开启事务前，查看数据库的默认隔离级别

```
SHOW TRANSACTION ISOLATION LEVEL;
或者
SHOW TRANSACTION_ISOLATION;
```

(2) 开启事务后，在事务内部查看该事务的隔离级别

```
SHOW TRANSACTION ISOLATION LEVEL;
或者
SHOW TRANSACTION_ISOLATION;
```

注意：数据库的默认隔离级别无法更改

1.2.5 锁机制

锁的类型：

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。
行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

行锁一般分为以下两种类型：

共享锁（S）：又称读锁。允许一个事务去读一行，阻止其他事务获得相同数据集的互斥锁。若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。
排他/互斥锁（X）：又称写锁。允许获取互斥锁的事务更新数据，阻止其他事务取得相同的数据集共享读锁和排他写锁。若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。

另外，为了允许行锁和表锁共存，实现多粒度锁机制，还有两种内部使用的表级意向锁（Intention Locks）：

意向共享锁（IS）：事务打算给数据行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。
意向互斥锁（IX）：事务打算给数据行加互斥锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。

openGauss 的加锁：

openGauss 提供了多种锁模式用于控制对表中数据的并发访问。这些模式可以用在 MVCC（多版本并发控制）无法给出期望行为的场合。同样，大多数 openGauss 命令自动施加恰当的锁，以保证被引用的表在命令的执行过程中不会以一种不兼容的方式被删除或者修改。通常情况下，采用行级锁。

对于 UPDATE、DELETE 和 INSERT 语句，openGauss 会自动给这些查询语句所涉及的数据集加互斥锁（X）；

对于 SELECT 语句，openGauss 不会加任何锁，但事务可以通过以下语句显示给记录集互斥锁（X）：

```
SELECT * FROM table_name WHERE ... FOR UPDATE;
```

1.3 实验内容

1. 通过 update 操作对现有关系表进行修改，观察违反 check 约束的 update 执行结果，理解 openGauss 对单个事务提供的原子性保障机制；

2. 在 openGauss 两种事务执行模式下，完成串行事务提交与回滚实验，对数据库表进行查询（select）、更新（update）、插入(insert)、删除(delete)，了解 openGauss 事务组成方式和 commit/rollback 对查询结果的影响，了解数据库内容/状态发生变化时 openGauss 提供的事务原子性的保障机制；

3. 在显式执行模式下，通过串行执行多个事务，对现有数据库表增加、删除属性列，观察 commit/rollback 对执行结果的影响，了解数据库模式结构发生变化时 openGauss 提供的事务原子性保障机制；

4. 观察单事务或串行事务中保存点（Save Point）回滚 rollback 对数据库访问结果的影响，了解 openGauss 保存点机制；

5. 观察 openGauss 的加锁机制，包括对数据库、关系表、数据行等不同粒度的数据对象所施加的各种不同类型的锁；

6. 了解 openGauss 提供的各种事务隔离级别。观察分析在“读提交”隔离级别下，并发事务执行导致的数据不一致性，如丢失修改（写-写错误）、读脏数据（写-读错误）、不可重复读（读-写错误）、幻象；

7. 分析对比多种隔离级别下，如“读提交”与“可重复读”，并发事务执行导致的数据不一致性，了解 openGauss 提供的事务一致性和独立性保障机制。

1.4 实验要求

1. 按照 1.3 节要求，完成全部实验内容。

2. 参照第二部分所给示例，选择 TPC-H 供应商和采购商数据库中不同的关系表，完成各个实验内容。

1.5 实验步骤

按照下述步骤完成本实验。

步骤 1. 实验准备

以课堂所学事务管理相关内容为基础，课后查阅、自学 openGuass 事务管理相关内容，包括事务组成、事务执行模式、隔离级别、锁类型/粒度、多版本协议、日志与故障、备份恢复等。

步骤 2. 单事务/串行事务执行原子性保障机制验证

依据示例，编写事务程序，验证以下六种情况下，DBMS 提供的原子性保障机制：单一事务执行 update 操作、多个串行事务执行 commit/rollback 操作、使用 alter table drop/add 修改数据库表结构、串行事务中多条 insert/delete 语句、利用保存点回滚事务、事务内部单条语句失败对事务的整体影响。

步骤 3. 事务并发执行时的独立性保障机制验证

依据示例，编写事务程序，观察验证 Read-committed 隔离级别下是否会发生脏读、不可重复读、幻读，Repeatable-read 隔离级别下是否会发生脏读、不可重复读、幻读，并对比两种隔离级别下多事务访问结果。

步骤 4. 事务加锁机制

使用 DBMS 提供的控制命令，观察事务并发执行时的加锁信息。

依据示例程序，构造分析并发事务执行时的死锁发生及处理机制。

1.6 实验总结

在实验中有哪些重要问题或者事件？你如何处理的？你的收获是什么？有何建议和意见等等。

2 实验示例

2.1 单事务与串行事务

2.1.1 违反 check 约束的 update 操作

实验要求

在 TPC-H 供应商和采购商数据库中，零部件供应表 partsupp 中的零件供应数量不能小于 0。 在关系表 partsupp（注意：实验前备份该表，以防实验造成数据丢失）上，用 Alter table add check 添加约束，并在该备份表上完成以下实验内容：

Step1. 查询零部件供应表的零件供应数量（PS_AVAILQTY）小于 20 的 PS_PARTKEY、PS_SUPPKEY 和 PS_AVAILQTY；

Step2. 更新零部件供应表将 step1 中的 PS_AVAILQTY 设置为当前值减去 15（注意此时有可能违反 check 约束）

Step3. 查询零部件供应表的零件供应数量（PS_AVAILQTY）小于 20 的 PS_PARTKEY、PS_SUPPKEY 和 PS_AVAILQTY；

针对以上操作分别进行如下的操作：

(1) 将以上操作组织成普通的 SQL 语句，顺序执行。

(2) 将以上操作组织成事务执行（以 start transaction; 开始，以 end;结束）。

查看数据库，观察两次的执行结果有何异同。

实验过程

以下示例是按照“零件供应数量（PS_AVAILQTY）小于 10”、“PS_AVAILQTY 设置为当前值减去 8”，完成的实验内容，供参考。

为方便起见，创建 partsupp 表的副本 partsupp_1，partsupp_2，并将 partsupp 表的数据导入进去

```
CREATE TABLE partsupp_1(
PS_PARTKEY integer NOT NULL,
PS_SUPPKEY integer NOT NULL,
```

```
PS_AVAILQTY integer NOT NULL,
PS_SUPPLYCOST decimal(15, 2) NOT NULL,
PS_COMMENT varchar(199) NOT NULL
);

insert into partsupp_1
select *
from partsupp;
```

用 Alter table add check 添加约束

```
alter table partsupp_1 add constraint partsupp_chk_1 check(PS_AVAILQTY >=0);
alter table partsupp_2 add constraint partsupp_chk_2 check(PS_AVAILQTY >=0);
```

将实验内容在 partsupp_1 表上组织成普通的 SQL 语句，顺序执行

```
select *
from partsupp_1
where PS_AVAILQTY<10;

update partsupp_1
set PS_AVAILQTY=PS_AVAILQTY-8
where PS_AVAILQTY<10;

select *
from partsupp_1
where PS_AVAILQTY<10;
```

在更新操作时，报错

```
ERROR:  new row for relation "partsupp_1" violates check constraint "partsupp_chk_1"
DETAIL:  Failing row contains (1, 2, -7, 771.64, , even theodolites. regular, final theodolites eat
after the car...).
```

两次查询结果一样

```
 ps_partkey | ps_suppkey | ps_availqty | ps_supplycost |
                                                ps_comment

------------+------------+-------------+---------------+----------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------
          1 |          2 |           1 |        771.64 | , even theodolites. regular, final theodoli
tes eat after the carefully pending foxes. furiously regular deposits sleep slyly. carefully bold re
alms above the ironic dependencies haggle careful
          1 |        502 |           1 |        993.49 | ven ideas. quickly even packages print. pen
ding multipliers must have to are fluff
          1 |       1002 |           1 |        337.09 | after the fluffily ironic deposits? blithel
y special dependencies integrate furiously even excuses. blithely silent theodolites could have to h
aggle pending, express requests; fu
          1 |       1502 |           1 |        357.84 | al, regular dependencies serve carefully af
ter the quickly final pinto beans. furiously even deposits sleep quickly final, silent pinto beans.
fluffily reg
          2 |          3 |           1 |        378.49 | nic accounts. final accounts sleep furiousl
y about the ironic, bold packages. regular, regular accounts
          2 |        503 |           1 |        915.27 | ptotes. quickly pending dependencies integr
ate furiously. fluffily ironic ideas impress blithely above the express accounts. furiously even epi
taphs need to wak
          2 |       1003 |           1 |        438.37 | blithely bold ideas. furiously stealthy pac
kages sleep fluffily. slyly special deposits snooze furiously carefully regular accounts. regular de
posits according to the accounts nag carefully slyl
          2 |       1503 |           1 |        306.39 | olites. deposits wake carefully. even, expr
ess requests cajole. carefully regular ex
          3 |          4 |           1 |        920.92 | ilent foxes affix furiously quickly unusual
 requests. even packages across the carefully even theodolites nag above the sp
          3 |        504 |           1 |        498.13 | ending dependencies haggle fluffily. regula
r deposits boost quickly carefully regular requests. deposits affix furiously around the pinto beans
. ironic, unusual platelets across the p
          3 |       1004 |           1 |        645.40 | of the blithely regular theodolites. final
theodolites haggle blithely carefully unusual ideas. blithely even f
          3 |       1504 |           1 |        191.92 |  unusual, ironic foxes according to the ide
as detect furiously alongside of the even, express requests. blithely regular the
--More--_
```

将实验内容在 partsupp_2 表上组织成事务执行（以 start transaction; 开始，以 end;结束）。

```
START TRANSACTION;
```

```
tpch=# START TRANSACTION;
START TRANSACTION
```

在事务中查询“零件供应数量（PS_AVAILQTY）小于 10”的零件供应记录，结果与上面的查询结果一样

```
select *
from partsupp_2
where PS_AVAILQTY<10;
```

```
 ps_partkey | ps_suppkey | ps_availqty | ps_supplycost |
                                                ps_comment

------------+------------+-------------+---------------+----------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------
          1 |          2 |           1 |        771.64 | , even theodolites. regular, final theodoli
tes eat after the carefully pending foxes. furiously regular deposits sleep slyly. carefully bold re
alms above the ironic dependencies haggle careful
          1 |        502 |           1 |        993.49 | ven ideas. quickly even packages print. pen
ding multipliers must have to are fluff
          1 |       1002 |           1 |        337.09 | after the fluffily ironic deposits? blithel
y special dependencies integrate furiously even excuses. blithely silent theodolites could have to h
aggle pending, express requests; fu
          1 |       1502 |           1 |        357.84 | al, regular dependencies serve carefully af
ter the quickly final pinto beans. furiously even deposits sleep quickly final, silent pinto beans.
fluffily reg
          2 |          3 |           1 |        378.49 | nic accounts. final accounts sleep furiousl
y about the ironic, bold packages. regular, regular accounts
          2 |        503 |           1 |        915.27 | ptotes. quickly pending dependencies integr
ate furiously. fluffily ironic ideas impress blithely above the express accounts. furiously even epi
taphs need to wak
          2 |       1003 |           1 |        438.37 | blithely bold ideas. furiously stealthy pac
kages sleep fluffily. slyly special deposits snooze furiously carefully regular accounts. regular de
posits according to the accounts nag carefully slyl
          2 |       1503 |           1 |        306.39 | olites. deposits wake carefully. even, expr
ess requests cajole. carefully regular ex
          3 |          4 |           1 |        920.92 | ilent foxes affix furiously quickly unusual
 requests. even packages across the carefully even theodolites nag above the sp
          3 |        504 |           1 |        498.13 | ending dependencies haggle fluffily. regula
r deposits boost quickly carefully regular requests. deposits affix furiously around the pinto beans
. ironic, unusual platelets across the p
          3 |       1004 |           1 |        645.40 | of the blithely regular theodolites. final
theodolites haggle blithely carefully unusual ideas. blithely even f
          3 |       1504 |           1 |        191.92 |  unusual, ironic foxes according to the ide
as detect furiously alongside of the even, express requests. blithely regular the
--More--
```

然后，在事务中进行更新

```
update partsupp_2
set PS_AVAILQTY=PS_AVAILQTY-8
where PS_AVAILQTY<10;
```

报错

```
ERROR:  new row for relation "partsupp_2" violates check constraint "partsupp_chk_2"
DETAIL:  Failing row contains (1, 2, -7, 771.64, , even theodolites. regular, final theodolites eat
after the car...).
```

再次，在事务中查询“零件供应数量（PS_AVAILQTY）小于 10”的零件供应记录

```
select *
from partsupp_2
where PS_AVAILQTY<10;
```

报错，由于事务中前面的命令执行失败，之后的命令便无法执行了，需要将事务回滚

```
ERROR:  current transaction is aborted, commands ignored until end of transaction block, firstChar[Q
]
```

用 end 结束本事务，事务自动回滚

```
END;
```

```
tpch=# end;
ROLLBACK
```

退出事务后，再次查询

```
select *
from partsupp_2
where PS_AVAILQTY<10;
```

与上面的查询结果一致，数据没有发生变动

```
 ps_partkey | ps_suppkey | ps_availqty | ps_supplycost |
                                                ps_comment

------------+------------+-------------+---------------+----------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------
          1 |          2 |           1 |        771.64 | , even theodolites. regular, final theodoli
tes eat after the carefully pending foxes. furiously regular deposits sleep slyly. carefully bold re
alms above the ironic dependencies haggle careful
          1 |        502 |           1 |        993.49 | ven ideas. quickly even packages print. pen
ding multipliers must have to are fluff
          1 |       1002 |           1 |        337.09 | after the fluffily ironic deposits? blithel
y special dependencies integrate furiously even excuses. blithely silent theodolites could have to h
aggle pending, express requests; fu
          1 |       1502 |           1 |        357.84 | al, regular dependencies serve carefully af
ter the quickly final pinto beans. furiously even deposits sleep quickly final, silent pinto beans.
fluffily reg
          2 |          3 |           1 |        378.49 | nic accounts. final accounts sleep furiousl
y about the ironic, bold packages. regular, regular accounts
          2 |        503 |           1 |        915.27 | ptotes. quickly pending dependencies integr
ate furiously. fluffily ironic ideas impress blithely above the express accounts. furiously even epi
taphs need to wak
          2 |       1003 |           1 |        438.37 | blithely bold ideas. furiously stealthy pac
kages sleep fluffily. slyly special deposits snooze furiously carefully regular accounts. regular de
posits according to the accounts nag carefully slyl
          2 |       1503 |           1 |        306.39 | olites. deposits wake carefully. even, expr
ess requests cajole. carefully regular ex
          3 |          4 |           1 |        920.92 | ilent foxes affix furiously quickly unusual
 requests. even packages across the carefully even theodolites nag above the sp
          3 |        504 |           1 |        498.13 | ending dependencies haggle fluffily. regula
r deposits boost quickly carefully regular requests. deposits affix furiously around the pinto beans
. ironic, unusual platelets across the p
          3 |       1004 |           1 |        645.40 | of the blithely regular theodolites. final
theodolites haggle blithely carefully unusual ideas. blithely even f
          3 |       1504 |           1 |        191.92 |  unusual, ironic foxes according to the ide
as detect furiously alongside of the even, express requests. blithely regular the
--More--
```

分析：当执行 update 语句之后发现报错，check 约束不合法，会导致回滚，单语句顺序执行的话，会导致单语句回滚，事务执行的话，则是整个事务回滚，说明当有 check 约束时，某行更新失败会使得整条语句（或者整个事务）全部回滚，并非是只跳过 check 不通过的那些行。

2.1.2 事务 commit/rollback 操作

实验要求

分别以两种事务执行模式，即自动提交、显式提交，在关系表 partsupp 上执行以下操作，并观察、分析、解释执行结果。注意：实验前备份该表，以防实验造成数据丢失。

Step1. 查看零件 key 在'2020'和'2022'之间的零件供应记录的零件供应成本；

Step2. 将零件 key 在'2020'和'2022'之间的零件供应记录的零件供应成本更新为 200;

Step3. 再次查看零件 key 在'2020'和'2022'之间的零件供应记录的零件供应成本。

将 step1、step2 和 step3 的数据库访问组织成 1 个单一事务 T1，再将 step3 作为 1 个独立事务，提交 DBMS，串行执行这 2 个事务，观察 T1 中的 rollback、commit 对事务执行结果的影响。

由 step1、step2 和 step3 组成的事务 T1 采用以下 2 种结束方式：

（1）以 commit 结束。

（2）以 rollback 结束。

实验过程

以下为示例程序，供参考

以 commit 结束的 T1 事务

```
START TRANSACTION;
select PS_PARTKEY,PS_SUPPLYCOST
from partsupp_1
where PS_PARTKEY between '2020' and '2022';
update partsupp_1
set PS_SUPPLYCOST=200
where PS_PARTKEY between '2020' and '2022';
select PS_PARTKEY,PS_SUPPLYCOST
from partsupp_1
where PS_PARTKEY between '2020' and '2022';
commit;
```

共查询到 12 条数据，并将 PS_SUPPLYCOST 更新为 200，更新成功，将事务提交。

```
tpch=# START TRANSACTION;
START TRANSACTION
tpch=# select PS_PARTKEY, PS_SUPPLYCOST
from partsupp_1
where PS_PARTKEY between '2020' and '2022';
 ps_partkey | ps_supplycost
------------+---------------
       2020 |        764.16
       2020 |        301.31
       2020 |        440.30
       2020 |        652.16
       2021 |        995.42
       2021 |        848.23
       2021 |        587.94
       2021 |        536.95
       2022 |        766.84
       2022 |        343.68
       2022 |        165.81
       2022 |        714.06
(12 rows)
```

```
tpch=# update partsupp_1
tpch-# set PS_SUPPLYCOST=200
tpch-# where PS_PARTKEY between '2020' and '2022';
UPDATE 12
tpch=# select PS_PARTKEY, PS_SUPPLYCOST
from partsupp_1
where PS_PARTKEY between '2020' and '2022';
 ps_partkey | ps_supplycost
------------+---------------
       2020 |        200.00
       2020 |        200.00
       2020 |        200.00
       2020 |        200.00
       2021 |        200.00
       2021 |        200.00
       2021 |        200.00
       2021 |        200.00
       2022 |        200.00
       2022 |        200.00
       2022 |        200.00
       2022 |        200.00
(12 rows)

tpch=# commit;
COMMIT
```

作为独立事务，再次进行查询

```
select PS_PARTKEY,PS_SUPPLYCOST
from partsupp_1
where PS_PARTKEY between '2020' and '2022';
```

数据为更新后的数据，可知，T1 事务内的更新操作有效，数据表发生更改。

```
tpch=# select PS_PARTKEY, PS_SUPPLYCOST
from partsupp_1
where PS_PARTKEY between '2020' and '2022';
 ps_partkey | ps_supplycost
------------+---------------
       2020 |        200.00
       2020 |        200.00
       2020 |        200.00
       2020 |        200.00
       2021 |        200.00
       2021 |        200.00
       2021 |        200.00
       2021 |        200.00
       2022 |        200.00
       2022 |        200.00
       2022 |        200.00
       2022 |        200.00
(12 rows)
```

以 rollback 结束的 T1 事务

```
START TRANSACTION;
select PS_PARTKEY,PS_SUPPLYCOST
from partsupp_2
where PS_PARTKEY between '2020' and '2022';
update partsupp_2
set PS_SUPPLYCOST=200
where PS_PARTKEY between '2020' and '2022';
select PS_PARTKEY,PS_SUPPLYCOST
```

```
from partsupp_2
where PS_PARTKEY between '2020' and '2022';
rollback;
```

共查询到 12 条数据，并将 EARFCN 更新为 38950，更新成功，将事务回滚。

```
tpch=# START TRANSACTION;
START TRANSACTION
tpch=# select PS_PARTKEY, PS_SUPPLYCOST
from partsupp_2
where PS_PARTKEY between '2020' and '2022';
 ps_partkey | ps_supplycost
------------+---------------
       2020 |        764.16
       2020 |        301.31
       2020 |        440.30
       2020 |        652.16
       2021 |        995.42
       2021 |        848.23
       2021 |        587.94
       2021 |        536.95
       2022 |        766.84
       2022 |        343.68
       2022 |        165.81
       2022 |        714.06
(12 rows)
```

```
tpch=# update partsupp_2
set PS_SUPPLYCOST=200
where PS_PARTKEY between '2020' and '2022';
UPDATE 12
tpch=# select PS_PARTKEY, PS_SUPPLYCOST
from partsupp_2
where PS_PARTKEY between '2020' and '2022';
 ps_partkey | ps_supplycost
------------+---------------
       2020 |        200.00
       2020 |        200.00
       2020 |        200.00
       2020 |        200.00
       2021 |        200.00
       2021 |        200.00
       2021 |        200.00
       2021 |        200.00
       2022 |        200.00
       2022 |        200.00
       2022 |        200.00
       2022 |        200.00
(12 rows)

tpch=# rollback;
ROLLBACK
```

作为独立事务，再次进行查询

```
select PS_PARTKEY,PS_SUPPLYCOST
from partsupp_2
where PS_PARTKEY between '2020' and '2022';
```

数据为更新前的数据，由于事务回滚，事务内的更新操作无效，数据表不发生更改。

```
tpch=# select PS_PARTKEY, PS_SUPPLYCOST
from partsupp_2
where PS_PARTKEY between '2020' and '2022';
 ps_partkey | ps_supplycost
------------+---------------
       2020 |        764.16
       2020 |        301.31
       2020 |        440.30
       2020 |        652.16
       2021 |        995.42
       2021 |        848.23
       2021 |        587.94
       2021 |        536.95
       2022 |        766.84
       2022 |        343.68
       2022 |        165.81
       2022 |        714.06
(12 rows)
```

分析：

当显式执行事务时，一定要记得在事务结尾处执行 commit 操作，否则对数据表的更改就不会持久化生效。

2.1.3 修改数据库模式

实验要求

针对零部件供应表 partsupp（注意备份原表），

Step1. 修改 TPC-H 数据库中的 partsupp 表，删除列 PS_AVAILQTY（使用 alter table drop）；

Step2. 修改 partsupp 表，增加列 PS_AVAILQTY（使用 alter table add）。

将 step1、step2 的数据库访问组织成 1 个单一事务（以显式事务的方式），采用以下 2 种结束方式：

（1）以 commit 结束；

（2）以 rollback 结束。

查看数据库（用 select 语句查看被删除/增加的列），观察数据库模式修改语句（alter table），是否会受到 rollback，commit 语句的影响。

也可以自行选择或创建表、删除其它关系表，重复以上两步，查看数据库，观察数据库模式定义语句（create table）模式修改语句（drop table）是否会受到 rollback，commit 语句的影响。

实验过程

删除 PS_AVAILQTY 列，以 commit 结束

```
START TRANSACTION;
alter table partsupp_1 drop column PS_AVAILQTY;
COMMIT;
```

查看 PS_AVAILQTY 列

```
select PS_AVAILQTY from partsupp_1;
```

报错，显示该列以不存在，事务提交后，删除操作有效

```
ERROR:  column "ps_availqty" does not exist
LINE 1: select PS_AVAILQTY from partsupp_1;
               ^
CONTEXT:  referenced column: ps_availqty
```

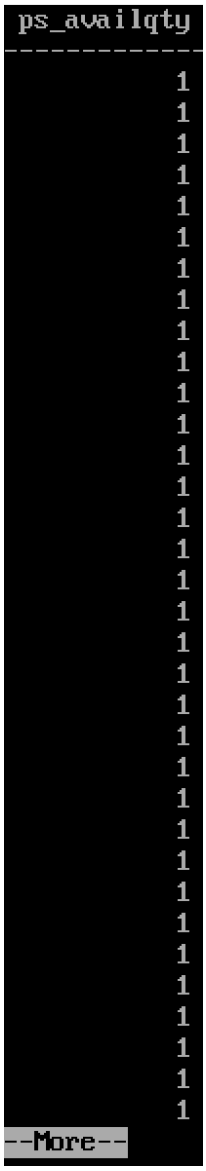
删除 PS_AVAILQTY 列，以 rollback 结束

```
START TRANSACTION;
alter table partsupp_2 drop column PS_AVAILQTY;
ROLLBACK;
```

查看 PS_AVAILQTY 列

```
select PS_AVAILQTY from partsupp_2;
```

该列仍然存在，事务回滚，删除操作无效。



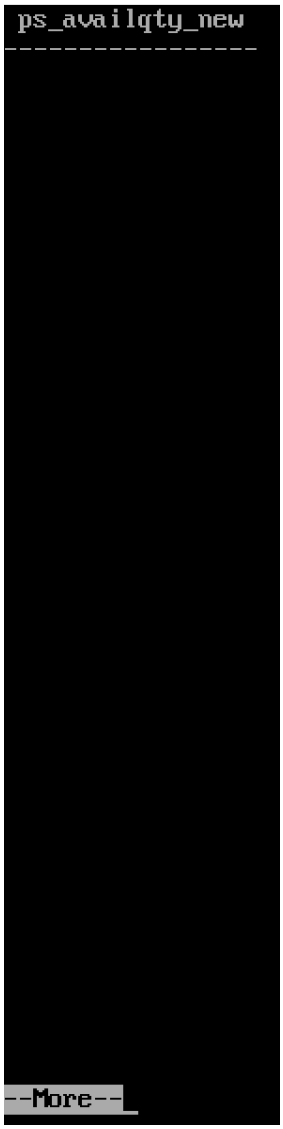
增加 PS_AVAILQTY_NEW 列，以 commit 结束

```
START TRANSACTION;
alter table partsupp_1 add column PS_AVAILQTY_NEW integer;
COMMIT;
```

查看 PS_AVAILQTY_NEW 列

```
select PS_AVAILQTY_NEW from partsupp_1;
```

显示该列存在，内容为空，事务提交后，增加操作有效



增加 PS_AVAILQTY_NEW 列，以 rollback 结束

```
START TRANSACTION;
alter table partsupp_2 add column PS_AVAILQTY_NEW integer;
ROLLBACK;
```

查看 PS_AVAILQTY_NEW 列

```
select PS_AVAILQTY_NEW from partsupp_2;
```

显示该列不存在，事务回滚，增加操作无效

```
ERROR:  column "ps_availqty_new" does not exist
LINE 1: select PS_AVAILQTY_NEW from partsupp_2;
               ^
CONTEXT:  referenced column: ps_availqty_new
```

分析：

数据库模式定义语句（比如：create table），模式修改语句（比如：drop table）是会受到 rollback，commit 语句的影响。

2.1.4 多条 insert/delete 操作执行比较

实验要求

针对零部件供应表 partsupp（注意备份原表），

Step1. 查询零部件供应表的零件供应数量（PS_AVAILQTY）小于 7 的 PS_PARTKEY、PS_SUPPKEY 和 PS_AVAILQTY；

Step2. 在零部件供应表中，添加一条 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“2022”、PS_AVAILQTY 为 6 的信息；

Step3. 删除 step2 所添加的信息；

Step4. 查询零部件供应表的零件供应数量（PS_AVAILQTY）小于 7 的 PS_PARTKEY、PS_SUPPKEY 和 PS_AVAILQTY；

针对以上操作分别进行如下的操作：

（1）将以上操作组织成普通的 SQL 语句，顺序执行；

（2）将以上操作组织成事务执行（以 start transaction; 开始，以 commit;结束）

查看数据库，观察两次的执行结果有何异同。

实验过程

以下实验查询为“零件供应数量（PS_AVAILQTY）小于 10”，插入的 PS_PARTKEY 为“2022”，PS_SUPPKEY 为“2022”，PS_AVAILQTY 为“7”然后删除,再次查询，供参考。

组织成普通的 SQL 语句，顺序执行

```
select PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY
from partsupp_1
where PS_AVAILQTY<10;

INSERT INTO partsupp_1
values(2022,2022,7,0,'comment');

delete from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY='2022' ;

select PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY
from partsupp_1
where PS_AVAILQTY<10;
```

两次查询的结果相同

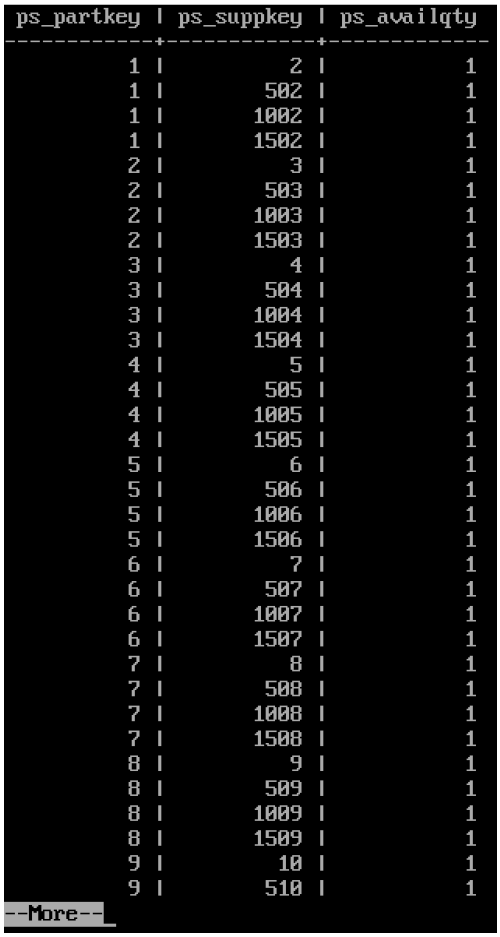
ps_partkey	ps_suppkey	ps_availqty
1	2	1
1	502	1
1	1002	1
1	1502	1
2	3	1
2	503	1
2	1003	1
2	1503	1
3	4	1
3	504	1
3	1004	1
3	1504	1
4	5	1
4	505	1
4	1005	1
4	1505	1
5	6	1
5	506	1
5	1006	1
5	1506	1
6	7	1
6	507	1
6	1007	1
6	1507	1
7	8	1
7	508	1
7	1008	1
7	1508	1
8	9	1
8	509	1
8	1009	1
8	1509	1
9	10	1
9	510	1

--More--

组织成事务执行（以 start transaction; 开始，以 commit;结束）

```
START TRANSACTION;
select PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY
from partsupp_2
where PS_AVAILQTY<10;
INSERT INTO partsupp_2
values(2022,2022,7,0,'comment');
delete from partsupp_2
where PS_PARTKEY='2022' and PS_SUPPKEY='2022';
select PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY
from partsupp_2
where PS_AVAILQTY<10;
COMMIT;
```

两次查询的结果相同（一共 602 行）



ps_partkey	ps_suppkey	ps_availqty
1	2	1
1	502	1
1	1002	1
1	1502	1
2	3	1
2	503	1
2	1003	1
2	1503	1
3	4	1
3	504	1
3	1004	1
3	1504	1
4	5	1
4	505	1
4	1005	1
4	1505	1
5	6	1
5	506	1
5	1006	1
5	1506	1
6	7	1
6	507	1
6	1007	1
6	1507	1
7	8	1
7	508	1
7	1008	1
7	1508	1
8	9	1
8	509	1
8	1009	1
8	1509	1
9	10	1
9	510	1

--More--

分析：

两次执行的结果没有不同，多条 insert/delete 操作的显式事务操作跟隐式事务操作的结果集一样。

2.1.5 保存点 Savepoint 设置与回滚实验

实验要求

本实验要求在事务内部不同执行位置设置，例如添加之后、添加之前、删除之后等，使用 SAVE TRANSACTION savepoint_name 语句创建保存点,使用 ROLLBACK savepoint_name 语句将事务回滚，观察每次操作的结果。保存点提供了回滚部分事务的机制，而不是回滚到事务的开始。

以零部件表 part 为访问对象，在创建的事务中 insert 插入语句后设置保存点，然后删除添加的信息，并回滚至保存点并提交事务；事务完成后再查询相应的行，观察执行结果是否插入成功，具体如下：

Step1. 查询零部件表 part 的零件大小 P_SIZE 为 300 的零件的 P_PARTKEY；

Step2. 在零部件表中，添加一条 P_PARTKEY 为 202200、P_SIZE 为 300 的信息；

Step3. 设置保存点；

Step4. 删除 step2 所添加的信息；

Step5. 回滚至保存点；

Step6. 事务提交结束；

Step7. 查询零部件表的零件大小 P_SIZE 为 300 的零件的 P_PARTKEY；

事务结构如下：

start transaction;

select 语句（检查表中原始数据）

insert 语句（向表中添加一行新的数据）

savepoint sp（设置保存点）

delete 语句（删除 insert 语句添加的行）

rollback to sp（回滚至保存点）

commit;（提交事务）

select 语句（检查插入数据是否成功）

实验过程

以下事务以 partsupp 为访问对象，在插入操作之后设置了保存点：

```
START TRANSACTION;
select PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY
from partsupp_1
where PS_AVAILQTY<10;
INSERT INTO partsupp_1
values(2022,2022,7,0,'comment');
savepoint sp;
delete from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY='2022';
rollback to sp;
COMMIT;
```

查看是否插入成功，插入的数据是否被删除

```
select PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY
from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY='2022';
```

局部回滚在插入操作之后，删除操作之前，插入成功，插入的数据没有被删除

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       2022 |           7
(1 row)
```

分析：

在事务内部，用 savepoint_name 语句创建保存点,使用 ROLLBACK to savepoint_name 语句将事务回滚到保存点的时刻，而不是回滚到事务的开始。保存点提供了回滚部分事务的机制,利用 savepoint 来达到局部回滚的目的。

2.1.6 事务内某条语句执行失败对其余语句的影响

实验要求

根据实际情况，零部件表 part 中的零件大小 P_SIZE 在 1 到 50 之间。 在 part 的备份表 part_new 上，用 Alter table add check 添加约束，并在该备份表上完成以下实验内容：

Step1. 在 part_new 表上添加约束：加入约束 check(P_SIZE between 1 and 50)。

Step2. 以备份表 part_new 表为访问对象，依次添加 P_SIZE 为‘30’和‘60’的两条数据，将 2 条对备份表 part_new 表进行顺序访问的 insert 语句组织成 1 个显示执行模式下的事务；

Step3. 观察并对比当事务执行违反约束时，事务结束后 part_new 的内容。

实验过程

以下实验内容以 partsupp 为访问对象，首先 insert 一条 PS_PARTKEY=999999 、PS_SUPPKEY=999999、PS_AVAILQTY=7 的数据，然后在 PS_AVAILQTY<10 的条件下更新所有的零件供应数量（PS_AVAILQTY）为当前数量减去 8。实验要求中的 sql 语句请同学们自行完成。

在 partsupp_1 表上加入 PS_AVAILQTY >= 0 的约束

```
alter table partsupp_1 add constraint partsupp_chk_1 check(PS_AVAILQTY>=0);
```

插入数据，并更新 PS_AVAILQTY 的值

```
START TRANSACTION;
select PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY
from partsupp_1
where PS_AVAILQTY<10;
INSERT INTO partsupp_1
values(999999,999999,7,0,'comment');
update partsupp_1
set PS_AVAILQTY=PS_AVAILQTY-8
where PS_AVAILQTY<10;
COMMIT;
```

由于不满足 check 约束，更新 PS_AVAILQTY 为当前数量减去 8 时失败

```
ERROR:  new row for relation "partsupp_1" violates check constraint "partsupp_chk_1"
DETAIL:  Failing row contains (1, 2, -7, 771.64, , even theodolites. regular, final theodolites eat
after the car...).
```

用 COMMIT 提交该事务时，由于事务内有命令执行失败，则该事务自动回滚

```
tpch=# COMMIT;
ROLLBACK
```

查询在上面事务中插入的数据

```
select PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY
from partsupp_1
where PS_PARTKEY='999999' and PS_SUPPKEY='999999';
```

为查询到该数据，不只是更新语句回滚，插入语句也跟着回滚，回滚到整个事务开始前

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
(0 rows)
```

分析：

事务内某条语句的错误执行会导致该事务在提交时强制回滚，回滚到事务开始前，从而使事务内其他语句的执行无效。

2.2 并发事务控制

在不同隔离级别下，并发事务读写导致的不一致性（脏读，不可重复读，幻读），设计如下实验，加以验证分析。

2.2.1 read committed 隔离级别下的脏读，不可重复读，幻读

实验要求

针对零部件供应表 partsupp（注意备份原表），在 read committed 隔离级别下，在并发事务内部进行数据的查询，更新，添加等，观察并发事务之间的影响，验证 read committed 隔离级别下是否存在脏读，不可重复读，幻读。

（1）脏读

Step1. 用隐式的独立事务查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值；

Step2. 创建 read committed 隔离级别下的并发事务 T1 和 T2；

Step3. 在事务 T1 中，将 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 修改为 6；

Step4. 在事务 T2 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值，并 COMMIT 提交事务 T2；

Step5. 用 ROLLBACK 回滚事务 T1；

要求：

观察 Step4 的 PS_AVAILQTY 值，与 Step1 和 Step3 的 PS_AVAILQTY 值进行对比，说明事务 T2 是否读取了事务 T1 未提交的数据，是否存在脏读；

（2）不可重复读

Step1. 创建 read committed 隔离级别下的并发事务 T1 和 T2；

Step2. 在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值；

Step3. 在事务 T2 中，将 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 修改为 6，并 COMMIT 提交事务 T2；

Step4. 在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值，并 COMMIT 提交事务 T1；

要求：

观察 Step2 和 Step4 两次查询中的 PS_AVAILQTY 值是否相等，说明是否为不可重复读；

（3）幻读

Step1. 创建 read committed 隔离级别下的并发事务 T1 和 T2；

Step2. 在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 在“23”与“2022”之间的元组；

Step3. 在事务 T2 中，插入 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“2022”的元组，并 COMMIT 提交事务 T2；

比如：values('2022','2022',0,0,'comment')

Step4. 在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 在“23”与“2022”之间的元组，并 COMMIT 提交事务 T1；

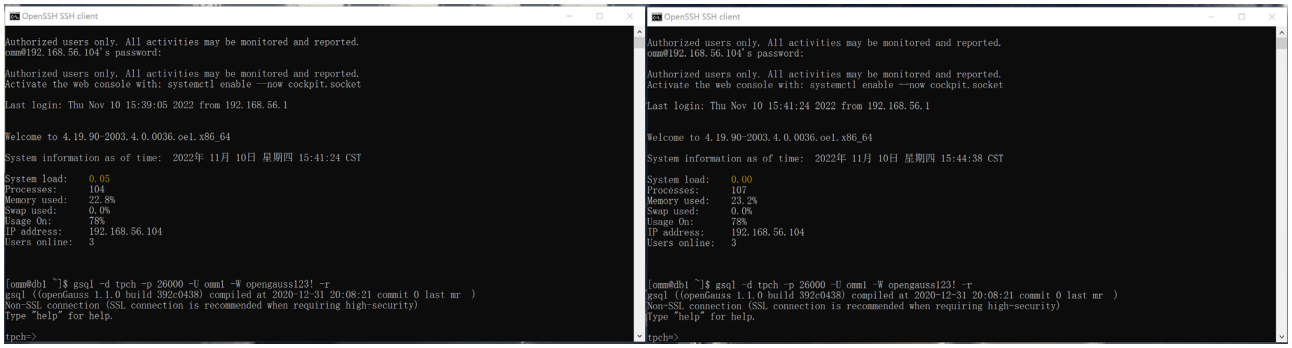
要求：

观察 Step2 和 Step4 两次查询到的元组是否相同，事务 T1 在 Step4 查询到的元组是否包含事务 T2 中插入的元组，说明是否为幻读；

实验过程

（1）

先打开两个 putty 窗口，并连接相同的数据库



查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值

```
select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
```

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       1526 |           1
(1 row)
```

在第一个窗口创建 read committed 隔离级别下的事务 T1

在第二个窗口创建 read committed 隔离级别下的事务 T2

```
START TRANSACTION ISOLATION LEVEL read committed;
```

```
START TRANSACTION
```

在事务 T1 中，将 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 修改为 6；

```
update partsupp_1
set PS_AVAILQTY=6
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
```

在事务 T2 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值；

```
select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
```

与之前查询的结果相同，都为 1。还未提交的事务 T1 对 PS_AVAILQTY 的更改并没有影响之后事务 T2 的查询结果，这样若事务 T1 之后回滚，事务 T2 也不会读到脏数据

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       1526 |           1
(1 row)
```

将事务 T2 提交，并将事务 T1 回滚

分析：

read committed 隔离级别下只能读到已经提交的数据，无法读到未提交的数据，不会出现脏读。

(2)

在第一个窗口创建 read committed 隔离级别下的事务 T1

在第二个窗口创建 read committed 隔离级别下的事务 T2

```
START TRANSACTION ISOLATION LEVEL read committed;
```

```
START TRANSACTION
```

在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值;

```
select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
```

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       1526 |           1
(1 row)
```

在事务 T2 中，将 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 修改为 6，并 COMMIT 提交事务 T2

```
update partsupp_1
set PS_AVAILQTY=6
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
commit;
```

在事务 T1 中，再次查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值，并 COMMIT 提交事务 T1

```
select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
commit;
```

两次查询的值不一样，第二次查询的值为 6，与之前提交的事务 T2 中更改的值相同

```
mmit; ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       1526 |           6
(1 row)
```

分析：

受到其它已经提交的事务的影响，不同的时刻读到的同一批数据不一样。该例中，T1 事务的查询操作受到之前已经提交的 T2 事务的数据更改操作的影响，使得两次对同一个数据的查询结果不一样。read committed 隔离级别下可能出现不可重复读。

（3）

在第一个窗口创建 read committed 隔离级别下的事务 T1

在第二个窗口创建 read committed 隔离级别下的事务 T2

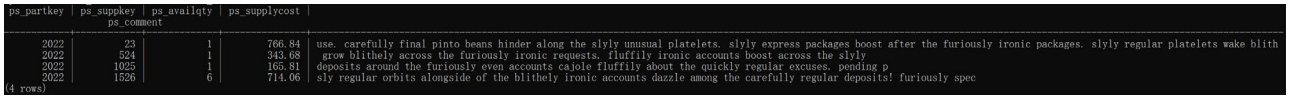
```
START TRANSACTION ISOLATION LEVEL read committed;
```

```
START TRANSACTION
```

在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 在“23”和“2022”之间的元组

```
select *
from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY between '23' and '2022';
```

一共有 4 行数据



ps_partkey	ps_suppkey	ps_availqty	ps_supplycost	ps_comment
2022	23	1	766.84	use. carefully final pinto beans hinder along the slyly unusual platelets. slyly express packages boost after the furiously ironic packages. slyly regular platelets wake blith
2022	524	1	343.68	grow blithely across the furiously ironic requests. fluffily ironic accounts boost across the slyly
2022	1025	1	165.81	deposits around the furiously even accounts cajole fluffily about the quickly regular excuses. pending p
2022	1526	6	714.06	sly regular orbits alongside of the blithely ironic accounts dazzle among the carefully regular deposits! furiously spec

(4 rows)

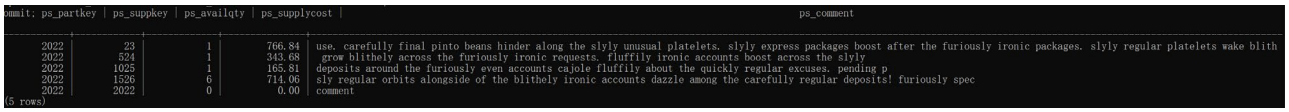
在事务 T2 中，插入 PS_PARTKEY 为“2022”、PS_SUPPKEY”为“2022”的元组，并 COMMIT 提交事务 T2

```
INSERT INTO partsupp_1
values('2022','2022',0,0,'comment');
commit;
```

在事务 T1 中，再次查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 在“23”和“2022”之间的元组,并 COMMIT 提交事务

```
select *
from partsupp_1
where PS_PARTKEY='2022' and PS_SUPPKEY between '23' and '2022';
commit;
```

查询结果一共有 5 行，多出的一行数据为在已经提交的事务 T2 中插入的新元组



ommit; ps_partkey	ps_suppkey	ps_availqty	ps_supplycost	ps_comment
2022	23	1	766.84	use. carefully final pinto beans hinder along the slyly unusual platelets. slyly express packages boost after the furiously ironic packages. slyly regular platelets wake blith
2022	524	1	343.68	grow blithely across the furiously ironic requests. fluffily ironic accounts boost across the slyly
2022	1025	1	165.81	deposits around the furiously even accounts cajole fluffily about the quickly regular excuses. pending p
2022	1526	6	714.06	sly regular orbits alongside of the blithely ironic accounts dazzle among the carefully regular deposits! furiously spec
2022	2022	0	0.00	comment

(5 rows)

分析：

在事务 T1 第一次查询以后，事务 T1 还未提交，事务 T2 就插入了一条在事务 T1 的查询范围内的元组，并且事务 T2 提交了，之后事务 T1 在进行一遍相同的查询，受事务 T2 的插入操作的影响，查询结果多出了一个数据。read committed 隔离级别下可能出现幻读。

2.2.2 repeatable read 隔离级别下的脏读，不可重复读，幻读

实验要求

针对零部件供应表 partsupp（注意备份原表），在 repeatable read 隔离级别下，在并发事务内部进行数据的查询，更新，添加等，观察并发事务之间的影响，验证 repeatable read 隔离级别下是否存在脏读，不可重复读，幻读。

（1）脏读

Step1. 用隐式的独立事务查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值；

Step2. 创建 repeatable read 隔离级别下的并发事务 T1 和 T2；

Step3. 在事务 T1 中，将 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 修改为 6；

Step4. 在事务 T2 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值，并 COMMIT 提交事务 T2；

Step5. 用 ROLLBACK 回滚事务 T1；

要求：

观察 Step4 的 PS_AVAILQTY 值，与 Step1 和 Step3 的 PS_AVAILQTY 值进行对比，说明事务 T2 是否读取了事务 T1 未提交的数据，是否存在脏读；

（2）不可重复读

Step1. 创建 repeatable read 隔离级别下的并发事务 T1 和 T2；

Step2. 在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值；

Step3. 在事务 T2 中，将 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 修改为 6，并 COMMIT 提交事务 T2；

Step4. 在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值，并 COMMIT 提交事务 T1；

要求：

观察 Step2 和 Step4 两次查询中的 PS_AVAILQTY 值是否相等，说明是否为不可重复读；

（3）幻读

Step1. 创建 repeatable read 隔离级别下的并发事务 T1 和 T2；

Step2. 在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 在“23”与“2022”之间的元组；

Step3. 在事务 T2 中，插入 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“2022”的元组，并 COMMIT 提交事务 T2；

比如：values('2022','2022',0,0,'comment')

Step4. 在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 在“23”与“2022”之间的元组，并 COMMIT 提交事务 T1；

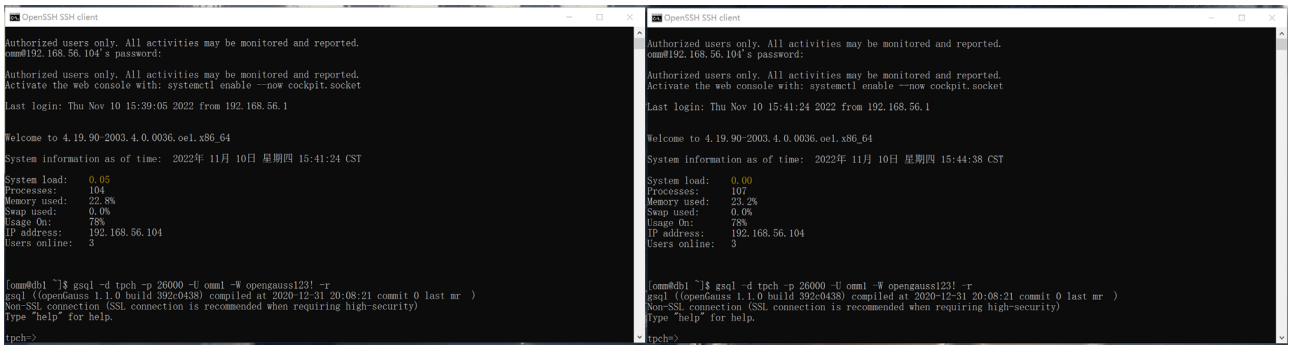
要求：

观察 Step2 和 Step4 两次查询到的元组是否相同，事务 T1 在 Step4 查询到的元组是否包含事务 T2 中插入的元组，说明是否为幻读；

实验过程

(1)

先打开两个 putty 窗口，并连接相同的数据库



查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值

```
select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
from partsupp_2
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
```

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       1526 |           1
(1 row)
```

在第一个窗口创建 repeatable read 隔离级别下的事务 T1

在第二个窗口创建 repeatable read 隔离级别下的事务 T2

```
START TRANSACTION ISOLATION LEVEL repeatable read;
```

```
START TRANSACTION
```

在事务 T1 中，将 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 修改为 6；

```
update partsupp_2
set PS_AVAILQTY=6
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
```

在事务 T2 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值；

```
select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
from partsupp_2
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
```

与之前查询的结果相同，都为 1。还未提交的事务 T1 对 PS_AVAILQTY 的更改并没有影响之后事务 T2 的查询结果，这样若事务 T1 之后回滚，事务 T2 也不会读到脏数据

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       1526 |           1
(1 row)
```

将事务 T2 提交，并将事务 T1 回滚

分析：

repeatable read 隔离级别下只能读到已经提交的数据，无法读到未提交的数据，不会出现脏读。

(2)

在第一个窗口创建 repeatable read 隔离级别下的事务 T1

在第二个窗口创建 repeatable read 隔离级别下的事务 T2

```
START TRANSACTION ISOLATION LEVEL repeatable read;
```

```
START TRANSACTION
```

在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“1526”的 PS_AVAILQTY 值；

```
select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
from partsupp_2
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
```

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       1526 |           1
(1 row)
```

在事务 T2 中，将 SECTOR_ID 为“15501-128”的 HEIGHT 修改为 6，并 COMMIT 提交事务 T2

```
update partsupp_2
set PS_AVAILQTY=6
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
commit;
```

在事务 T1 中，再次查询 SECTOR_ID 为“15500-128”的 HEIGHT 值，并 COMMIT 提交事务 T1

```
select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
from partsupp_2
where PS_PARTKEY='2022' and PS_SUPPKEY='1526';
commit;
```

两次查询的值相同，都为 1，前面已经提交的事务 T2 对 HEIGHT 值的修改未影响事务 T1 的查询

```
 ps_partkey | ps_suppkey | ps_availqty
------------+------------+-------------
       2022 |       1526 |           1
(1 row)
```

分析：

repeatable read 隔离级别下，一个事务仅仅看到本事务开始之前提交的数据，它不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。该例中，T1 事务只能看到在该事务开始之前提交的数据，而 T2 事务是在 T1 事务执行期间提交的，则 T1 事务无法看到 T2 事务修改的数据，T1 事务的查询结果不会产生变化。repeatable read 隔离级别下不会出现不可重复读。

(3)

在第一个窗口创建 repeatable read 隔离级别下的事务 T1

在第二个窗口创建 repeatable read 隔离级别下的事务 T2

```
START TRANSACTION ISOLATION LEVEL repeatable read;
```

```
START TRANSACTION
```

在事务 T1 中，查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 在“23”与“2022”之间的元组

```
select *
from partsupp_2
where PS_PARTKEY='2022' and PS_SUPPKEY between '23' and '2022';
```

一共有 4 行数据

ps_partkey	ps_suppkey	ps_availqty	ps_supplycost	ps_comment
2022	23	1	766.84	use. carefully final pinto beans hinder along the slyly unusual platelets. slyly express packages boost after the furiously ironic packages. slyly regular platelets wake blith
2022	524	1	343.68	grow blithely across the furiously ironic requests. fluffily ironic accounts boost across the slyly
2022	1025	1	165.81	deposits around the furiously even accounts cajole fluffily about the quickly regular excuses. pending p
2022	1526	1	714.06	sly regular orbits alongside of the blithely ironic accounts dazzle among the carefully regular deposits! furiously spec

(4 rows)

在事务 T2 中，插入 PS_PARTKEY 为“2022”、PS_SUPPKEY 为“2022”的元组，并 COMMIT 提交事务 T2

```
INSERT INTO partsupp_2
values('2022','2022',0,0,'comment');
commit;
```

在事务 T1 中，再次查询 PS_PARTKEY 为“2022”、PS_SUPPKEY 在“23”与“2022”之间的元组,并 COMMIT 提交事务

```
select *
from partsupp_2
where PS_PARTKEY='2022' and PS_SUPPKEY between '23' and '2022';
commit;
```

查询结果一共有 4 行，与之前的查询结果相同

ps_partkey	ps_suppkey	ps_availqty	ps_supplycost	ps_comment
2022	23	1	766.84	use. carefully final pinto beans hinder along the slyly unusual platelets. slyly express packages boost after the furiously ironic packages. slyly regular platelets wake blith
2022	524	1	343.68	grow blithely across the furiously ironic requests. fluffily ironic accounts boost across the slyly
2022	1025	1	165.81	deposits around the furiously even accounts cajole fluffily about the quickly regular excuses. pending p
2022	1526	1	714.06	sly regular orbits alongside of the blithely ironic accounts dazzle among the carefully regular deposits! furiously spec

(4 rows)

分析：

该例中，T1 事务只能看到在该事务开始之前提交的数据，而 T2 事务是在 T1 事务执行期间提交的，则 T1 事务无法看到 T2 事务插入的数据，T1 事务的查询结果不会产生变化。repeatable read 隔离级别下不会出现幻读。

2.3 事务锁机制

2.3.1 死锁分析

在 openGauss 中，当两个或以上的事务相互持有和请求锁，并形成一个循环的依赖关系，就会产生死锁。多个事务同时锁定同一个资源时，也可能产生死锁。死锁无法完全避免的，数据库会自动检测事务死锁，立即回滚其中某个事务，并且返回一个错误。它根据某种机制来选择回滚代价最小的事务来进行回滚。

以下两种情况可能导致并发事务死锁：

（1）在REPEATEABLE-READ隔离级别下，如果两个事务同时对关系表中满足where条件的元组记录用select-from-where-for-update加互斥锁，在没有符合条件记录的情况下，两个事务都会加锁成功。当2个事务发现满足where条件的元组记录尚不存在，都试图插入一条新记录，就会出现死锁。这种情况下，将隔离级别改成READ COMMITTED，可以避免死锁问题。

（2）当隔离级别为READ COMMITED时，如果两个并发事务都先执行select-from-where-for-update，判断是否存在符合where条件的记录，如果没有，就插入一条记录。此时，只有一个事务能插入成功，另一个事务会出现锁等待，当第 1 个事务提交后，第 2 个事务会因主键重合出错。但虽然这个事务出错，却会获得一个互斥锁。这时如果有第 3 个事务又来申请互斥锁，也会出现死锁。对于这种情况，可以直接做插入操作，然后再捕获主键重异常，或者在遇到主键重合错误时，总是执行ROLLBACK释放获得的互斥锁。

示例 1. 隔离级别 read-repeatable 下死锁

在 session1 中开启事务，执行 select for update 语句，该语句对表中符合条件的元组/数据行加上互斥锁。

```
START TRANSACTION ISOLATION LEVEL repeatable read;
select PS_PARTKEY from partsupp_1 where PS_SUPPKEY=1526 for update;
```

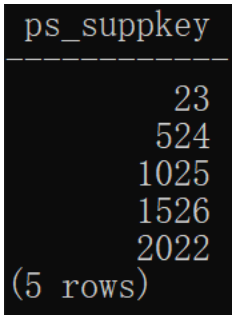
ps_partkey
25
525
1025
1525
2523
3024
3525
4019
4521
5023
5525
6016
6519
7022
7525
8013
8517
9021
9525
10010
10515
11020
11525
12007
12513
13019
13525

在 session2 中开启一个事务，也是执行上述 select for update 语句。此时检测到符合筛选条件的部分数据行上加了互斥锁（两次查询的数据行有重叠，存在 PS_PARTKEY=2022 且 PS_SUPPKEY=1526 的数据行），查询进入申请锁的等待阶段。

```
START TRANSACTION ISOLATION LEVEL repeatable read;
select PS_SUPPKEY from partsupp_1 where PS_PARTKEY=2022 for update;
```

```
tpch=> select PS_SUPPKEY from partsupp_1 where PS_PARTKEY=2022 for update;
```

若在等待过程中，马上释放数据行的互斥锁，比如 COMMIT 提交 session1 上占用数据行互斥锁的事务，则查询正常进行，此查询的业务对符合查询条件的数据行加入新的互斥锁。



```
 ps_suppkey
------------
         23
        524
       1025
       1526
       2022
(5 rows)
```

若在等待一段时间后，仍没有释放互斥锁，达到了设定的锁等待时间，系统会自动判定发生死锁，回滚当前事务。

```
ERROR:  Lock wait timeout: thread 139655381776128 on node dn_6001 waiting for ShareLock on transaction 147996 after 120019.777 ms
DETAIL:  blocked by hold lock thread 139655499347712, statement <select PS_SUPPKEY from partsupp_1 where PS_PARTKEY=2022 for update;>, hold lockmode ExclusiveLock.
```

分析：

在本例中，session1 的业务先占用了符合 PS_SUPPKEY=1526 的元组/数据行的互斥锁，之后 session2 的业务要申请符合 PS_PARTKEY=2022 的元组/数据行的互斥锁时，由于存在 PS_PARTKEY=2022 且 PS_SUPPKEY=1526 的数据行，则在 session1 的业务释放符合 PS_SUPPKEY=1526 的元组/数据行的互斥锁前，session2 的业务无法占用符合 PS_PARTKEY=2022 的元组/数据行的互斥锁。因此，session2 的业务进入申请锁的等待阶段，这个等待阶段设定了最大的锁等待时间，超时后，系统会自动判定发生死锁，回滚当前事务。若在超时前，session1 的业务占用的互斥锁释放了，则 session2 的业务能够对符合 PS_PARTKEY=2022 的元组/数据行加上互斥锁，session2 的业务的查询便能正常进行了。

示例 2. 加的互斥锁的粒度

在 session1 中开启事务，执行 select for update 语句，该语句对表中符合条件的元组/数据行加上互斥锁。

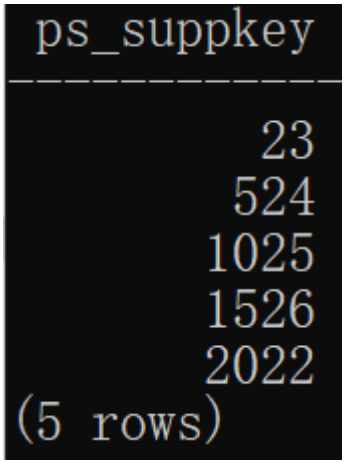
```
START TRANSACTION ISOLATION LEVEL repeatable read;
select PS_PARTKEY from partsupp_1 where PS_SUPPKEY=20 for update;
```

ps_partkey
19
519
1019
1519
2019
2516
3017
3518
4019
4513
5015
5517
6019
6510
7013
7516
8019
8507
9011
9515
10019
10504
11009
11514
12019
12501
13007
13513
14019

--More--

在 session2 中开启一个事务，也是执行上述 select for update 语句。

```
START TRANSACTION ISOLATION LEVEL repeatable read;
select PS_SUPPKEY from partsupp_1 where PS_PARTKEY=2022 for update;
```



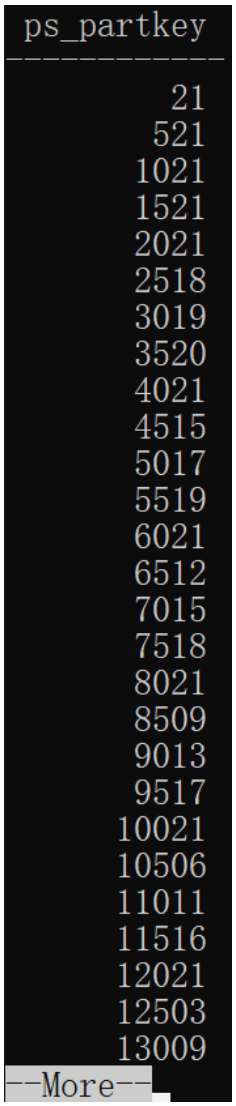
ps_suppkey
23
524
1025
1526
2022

(5 rows)

正常查询，没有发生锁等待，说明 select for update 语句，不是对整表加互斥锁。

在 session2 上，继续执行上述 select for update 语句

```
select PS_PARTKEY from partsupp_1 where PS_SUPPKEY=22 for update;
```



ps_partkey
21
521
1021
1521
2021
2518
3019
3520
4021
4515
5017
5519
6021
6512
7015
7518
8021
8509
9013
9517
10021
10506
11011
11516
12021
12503
13009

--More--

正常查询，没有发生锁等待，说明 select for update 语句，不是对查询条件所在的列加互斥锁。

在 session2 上，继续执行上述 select for update 语句，这次与 session1 上的查询条件一样

```
select PS_PARTKEY from partsupp_1 where PS_SUPPKEY=20 for update;
```

```
tpch=> select PS_PARTKEY from partsupp_1 where PS_SUPPKEY=20 for update;
```

发生锁等待，并且等待一段时间后，系统会自动判定发生死锁，回滚当前事务。

```
ERROR:  Lock wait timeout: thread 139655499347712 on node dn_6001 waiting for ShareLock on transaction 147998 after 120000.379 ms
DETAIL:  blocked by hold lock thread 139655381776128, statement <select PS_PARTKEY from partsupp_1 where PS_SUPPKEY=20 for update;>, hold lockmode ExclusiveLock.
```

说明 select for update 语句，是对表中符合查询条件的元组/数据行加上互斥锁

分析：

select for update 语句，加入的互斥锁的粒度一般为行锁。

2.3.2 锁管理参数

deadlock_timeout

参数说明：设置死锁超时检测时间，以毫秒为单位。当申请的锁超过设定值时，系统会检查是否产生了死锁。

死锁的检查代价是比较高的，服务器不会在每次等待锁的时候都运行这个过程。在系统运行过程中死锁是不经常出现的，因此在检查死锁前只需等待一个相对较短的时间。增加这个值就减少了无用的死锁检查浪费的时间，但是会减慢真正的死锁错误报告的速度。在一个负载过重的服务器上，用户可能需要增大它。这个值的设置应该超过事务持续时间，这样就可以减少在锁释放之前就开始死锁检查的问题。

如果要通过设置 log_lock_waits 来将查询执行过程中的锁等待耗时信息写入日志，请确保 log_lock_waits 的设置值小于 deadlock_timeout 的设置值（或默认值）。

该参数属于 SUSET 类型参数。

取值范围：整型，1\~2147483647，单位为毫秒（ms）。

默认值：1s

lockwait_timeout

参数说明：控制单个锁的最长等待时间。当申请的锁等待时间超过设定值时，系统会报错。

该参数属于 SUSET 类型参数。

取值范围：整型，0 \~ INT_MAX，单位为毫秒（ms）。

默认值：20min

update_lockwait_timeout

参数说明：允许并发更新参数开启情况下，该参数控制并发更新同一行时单个锁的最长等待时间。当申请的锁等待时间超过设定值时，系统会报错。

该参数属于 SUSET 类型参数。

取值范围：整型，0 \~ INT_MAX，单位为毫秒（ms）。

默认值：2min

max_locks_per_transaction

参数说明：控制每个事务能够得到的平均的对象锁的数量。

共享的锁表的大小是以假设任意时刻最多只有 max_locks_per_transaction*(max_connections+max_prepared_transactions) 个独立的对象需要被锁住为基础进行计算的。不超过设定数量的多个对象可以在任一时刻同时被锁定。当在一个事务里面修改很多不同的表时，可能需要提高这个默认数值。只能在数据库启动的时候设置。

增大这个参数可能导致 openGauss 请求更多的 System V 共享内存，有可能超过操作系统的缺省配置。

当运行备机时，请将此参数设置不小于主机上的值，否则，在备机上查询操作不会被允许。

该参数属于 POSTMASTER 类型参数。

取值范围：整型，10 \~ INT_MAX

默认值：256

max_pred_locks_per_transaction

参数说明：控制每个事务允许断定锁的最大数量，是一个平均值。

共享的断定锁表的大小是以假设任意时刻最多只有 max_pred_locks_per_transaction*(max_connections+max_prepared_transactions) 个独立的对象需要被锁住为基础进行计算的。不超过设定数量的多个对象可以在任一时刻同时被锁定。当在一个事务里面修改很多不同的表时，可能需要提高这个默认数值。只能在服务器启动的时候设置。

增大这个参数可能导致 openGauss 请求更多的 System V 共享内存，有可能超过操作系统的缺省配置。

该参数属于 POSTMASTER 类型参数。

取值范围：整型，10 \~ INT_MAX

默认值：64

gs_clean_timeout

参数说明：控制 DBnode 周期性调用 gs_clean 工具的时间，是一个平均值。

openGauss 数据库中事务处理使用的是两阶段提交的方法，当有两阶段事务残留时，该事务通常会拿着表级锁，导致其它连接无法加锁，此时需要调用 gs_clean 工具对 openGauss 中两阶段事务进行清理，gs_clean_timeout 是控制 DBnode 周期性调用 gs_clean 的时间。

增大这个参数可能导致 openGauss 周期性调用 gs_clean 工具的时间延长，导致两阶段事务清理时间延长。

该参数属于 SIGHUP 类型参数。

取值范围：整型，0 \~ INT_MAX / 1000，单位为秒（s）。

默认值：5min

partition_lock_upgrade_timeout

参数说明：在执行某些查询语句的过程中，会需要将分区表上的锁级别由允许读的 ExclusiveLock 级别升级到读写阻塞的 AccessExclusiveLock 级别。如果此时已经存在并发的读事务，那么该锁升级操作将阻塞等待。partition_lock_upgrade_timeout 为尝试锁升级的等待超时时间。

在分区表上进行 MERGE PARTITION 和 CLUSTER PARTITION 操作时，都利用了临时表进行数据重排和文件交换，为了最大程度提高分区上的操作并发度，在数据重排阶段给相关分区加锁 ExclusiveLock，在文件交换阶段加锁 AccessExclusiveLock。

常规加锁方式是等待加锁，直到加锁成功，或者等待时间超过 lockwait_timeout 发生超时失败。

在分区表上进行 MERGE PARTITION 或 CLUSTER PARTITION 操作时，进入文件交换阶段需要申请加锁 AccessExclusiveLock，加锁方式是尝试性加锁，加锁成功了则立即返回，不成功则等待 50ms 后继续下次尝试，加锁超时时间使用会话级设置参数 partition_lock_upgrade_timeout。

特殊值：若 partition_lock_upgrade_timeout 取值-1，表示无限等待，即不停的尝试锁升级，直到加锁成功。

该参数属于 USERSET 类型参数。

取值范围：整型，最小值-1，最大值 3000，单位为秒（s）。

默认值：1800

fault_mon_timeout

参数说明：轻量级死锁检测周期。

该参数属于 SIGHUP 类型参数。

取值范围：整型，最小值 0，最大值 1440，单位为分钟（min）

默认值：5min

enable_online_ddl_waitlock

参数说明：控制 DDL 是否会阻塞等待 pg_advisory_lock/pgxc_lock_for_backup 等 openGauss 锁。主要用于 OM 在线操作场景，不建议用户设置。

该参数属于 SIGHUP 类型参数。

取值范围：布尔型，on 表示开启，off 表示关闭。

默认值：off

xloginsert_locks

参数说明：控制用于并发写预写式日志锁的个数。主要用于提高写预写式日志的效率。

该参数属于 POSTMASTER 类型参数。

取值范围：整型，最小值 1，最大值 1000

默认值：8

2.4 备份与恢复

2.4.1 概述

数据备份是保护数据安全的重要手段之一，为了更好的保护数据安全，openGauss 数据库支持两种备份恢复类型、多种备份恢复方案，备份和恢复过程中提供数据的可靠性保障机制。备份与恢复类型可分为逻辑备份与恢复、物理备份与恢复。

- **逻辑备份与恢复**：通过逻辑导出对数据进行备份，逻辑备份只能基于备份时刻进行数据转储，所以恢复时也只能恢复到备份时保存的数据。对于故障点和备份点之间的数据，逻辑备份无能为力，逻辑备份适合备份那些很少变化的数据，当这些数据因误操作被损坏时，可以通过逻辑备份进行快速恢复。如果通过逻辑备份进行全库恢复，通常需要重建数据库，导入备份数据来完成，对于可用性要求很高的数据库，这种恢复时间太长，通常不被采用。由于逻辑备份具有平台无关性，所以更为常见的是，逻辑备份被作为一个数据迁移及移动的主要手段。
- **物理备份与恢复**：通过物理文件拷贝的方式对数据库进行备份，以磁盘块为基本单位将数据备份。通过备份的数据文件及归档日志等文件，数据库可以进行完全恢复。物理备份速度快，一般被用作对数据进行备份和恢复，用于全量备份的场景。通过合理规划，可以低成本进行备份与恢复。

以下为 openGauss 支持的两类数据备份恢复方案，备份方案也决定了当异常发生时该如何恢复。

备份类型	应用场景	支持的介质	优缺点
逻辑备份与	适合于数据量小的场景。	磁盘	可按用户需要进行指定对象

恢复	目前用于表备份恢复，可以备份恢复单表和多表。	SSD	的备份和恢复，灵活度高。 当数据量大时，备份效率低。
物理备份与恢复	适用于数据量大的场景，主要用于全量数据备份恢复，也可对整个数据库中的 WAL 归档日志和运行日志进行备份恢复。		数据量大时，备份效率高。

表 1 两种备份恢复类型对比

当需要进行备份恢复操作时，主要从以下四个方面考虑数据备份方案。

- **备份对业务的影响在可接受范围。**
- **数据库恢复效率：为尽量减小数据库故障的影响，要使恢复时间减到最少，从而使恢复的效率达到最高。**
- **数据可恢复程度：当数据库失效后，要尽量减少数据损失。**
- **数据库恢复成本。**

在现网选择备份策略时参考的因素比较多，如备份对象、数据大小、网络配置等。

表 2 备份策略典型场景

备份策略	**关键性能因素**	**典型数据量**	**性能规格**
集群备份	数据大小 网络配置	数据：PB 级 对象：约 100 万个	备份： 每个主机 80 Mbit/s（NBU/EISOO+磁盘） 约 90%磁盘 I/O 速率（SSD/HDD）
表备份	表所在模式 网络配置（NBU）	数据：10 TB 级	备份：基于查询性能速度+I/O 速度 说明： 多表备份时，备份耗时计算方式： **复制代码总时间 = 表数量 x 起步时间 + 数据总量 / 数据备份速度** 其中：

			磁盘起步时间为 5s 左右，NBU 起步时间比 DISK 长（取决于 NBU 部署方案）。 数据备份速度为单节点 50MB/s 左右（基于 1GB 大小的表，物理机备份到本地磁盘得出此速率）。 表越小，备份性能越低。

物理备份恢复

gs_basebackup

背景信息

openGauss 部署成功后，在数据库运行的过程中，会遇到各种问题及异常状态。openGauss 提供了 gs_basebackup 工具做基础的物理备份。gs_basebackup 的实现目标是对服务器数据库文件进行二进制拷贝，其实现原理使用了复制协议。远程执行 gs_basebackup 时，需要使用系统管理员账户。gs_basebackup 当前支持热备份和压缩格式备份。

说明：

- gs_basebackup 仅支持全量备份，不支持增量。
- gs_basebackup 当前支持热备份模式和压缩格式备份模式。
- gs_basebackup 在备份包含绝对路径的表空间时，不能在同一台机器上进行备份。对于同一台机器，绝对路径是唯一的，因此会产生冲突。可以在不同的机器上备份含绝对路径的表空间。
- 若打开增量检测点功能且打开双写，gs_basebackup 也会备份双写文件。
- 若 pg_xlog 目录为软链接，备份时将不会建立软链接，会直接将数据备份到目的路径的 pg_xlog 目录下。
- 备份过程中收回用户备份权限，可能导致备份失败，或者备份数据不可用。

前提条件

- 可以正常连接 openGauss 数据库。
- 备份过程中用户权限没有被回收。
- pg_hba.conf 中需要配置允许复制链接，且该连接必须由一个系统管理员建立。
- 如果 xlog 传输模式为 stream 模式，需要配置 max_wal_senders 的数量，至少有一个可用。
- 如果 xlog 传输模式为 fetch 模式，有必要把 wal_keep_segments 参数设置得足够高，这样在备份末尾之前日志不会被移除。
- 在进行还原时，需要保证各节点备份目录中存在备份文件，若备份文件丢失，则需要从其他节点进行拷贝。

语法

显示帮助信息

```
gs_basebackup -?|--help
```

显示版本号信息

```
gs_basebackup -V|--version
```

参数说明

gs_basebackup 参数可以分为如下几类：

-D directory

备份文件输出的目录，必选项。

常用参数：

-c，–checkpoint=fast|spread

设置检查点模式为 fast 或者 spread(默认)。

-l，–label=LABEL

为备份设置标签。

-P，–progress

启用进展报告。

-v， –verbose

启用冗长模式。

-V， –version

打印版本后退出。

-?，–help

显示 gs_basebackup 命令行参数。

-T，–tablespace-mapping=olddir=newdir

在备份期间将目录 olddir 中的表空间重定位到 newdir 中。为使之有效，olddir 必须正好匹配表空间所在的路径（但如果备份中没有包含 olddir 中的表空间也不是错误）。olddir 和 newdir 必须是绝对路径。如果一个路径凑巧包含了一个=符号，可用反斜线对它转义。对于多个表空间可以多次使用这个选项。

-F，–format=plain|tar

设置输出格式为 plain(默认)或者 tar。没有设置该参数的情况下，默认–format=plain。plain 格式把输出写成平面文件，使用和当前数据目录和表空间相同的布局。当集簇没有额外表空间时，整个数据库将被放在目标目录中。如果集簇包含额外的表空间，主数据目录将被放置在目标目录中，但是所有其他表空间将被放在它们位

于服务器上的相同的绝对路径中。tar 模式将输出写成目标目录中的 tar 文件。主数据目录将被写入到一个名为 base.tar 的文件中，并且其他表空间将被以其 OID 命名。生成的 tar 包，需要用 gs_tar 命令解压。

-X， –xlog-method=fetch|stream

设置 xlog 传输方式。没有设置该参数的情况下，默认–xlog-method=stream。在备份中包括所需的预写式日志文件（WAL 文件）。这包括所有在备份期间产生的预写式日志。fetch 方式在备份末尾收集预写式日志文件。因此，有必要把 wal_keep_segments 参数设置得足够高，这样在备份末尾之前日志不会被移除。如果在要传输日志时它已经被轮转，备份将失败并且是不可用的。stream 方式在备份被创建时流传送预写式日志。这将开启一个到服务器的第二连接并且在运行备份时并行开始流传输预写式日志。因此，它将使用最多两个由 max_wal_senders 参数配置的连接。只要客户端能保持接收预写式日志，使用这种模式不需要在主控机上保存额外的预写式日志。

-x，–xlog

使用这个选项等效于和方法 fetch 一起使用-X。

-Z –compress=level

启用对 tar 文件输出的 gzip 压缩，并且制定压缩级别（0 到 9，0 是不压缩，9 是最佳压缩）。只有使用 tar 格式时压缩才可用，并且会在所有 tar 文件名后面自动加上后缀.gz。

-z

启用对 tar 文件输出的 gzip 压缩，使用默认的压缩级别。只有使用 tar 格式时压缩才可用，并且会在所有 tar 文件名后面自动加上后缀.gz。

连接参数:

-h, –host=HOSTNAME

指定正在运行服务器的主机名或者 Unix 域套接字的路径。

-p，–port=PORT

指定数据库服务器的端口号。

可以通过 port 参数修改默认端口号。

-U，–username=USERNAME

指定连接数据库的用户。

-s, –status-interval=INTERVAL

发送到服务器的状态包的时间(以秒为单位)

-w,–no-password

不出现输入密码提示。

-W, –password

当使用-U 参数连接本地数据库或者连接远端数据库时，可通过指定该选项出现输入密码提示。

从备份文件恢复数据

当数据库发生故障时需要从备份文件进行恢复。因为 gs_basebackup 是对数据库按二进制进行备份，因此恢复时可以直接拷贝替换原有的文件， 或者直接在备份的库上启动数据库。

当使用-U 参数连接本地数据库或者连接远端数据库时，可通过指定该选项出现输入密码提示。

2.4.2 示例

步骤 1：备份前准备

在对数据库进行备份前，对数据库进行如下操作：

切换到 omm 用户，以操作系统用户 omm 登录数据库主节点。

```
su - omm
```

启动数据库服务

```
gs_om -t start
```

连接 openGauss 数据库。

```
gsql -d postgres -p 26000 -r
```

创建 customer_t1 表。

```
postgres=#DROP TABLE IF EXISTS customer_t1;


postgres=#CREATE TABLE customer_t1
(
    c_customer_sk             integer,
    c_customer_id             char(5),
    c_first_name              char(6),
    c_last_name               char(8)
);
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

向表中插入数据。

```
postgres=# INSERT INTO customer_t1 (c_customer_sk, c_customer_id, c_first_name) VALUES
(3769, 'hello', DEFAULT) ,
(6885, 'maps', 'Joes'),
(4321, 'tpcds', 'Lily'),
(9527, 'world', 'James');
```

显示结果为：

```
INSERT 0 4
```

查看表中的数据。

```
postgres=# select * from customer_t1;

 c_customer_sk | c_customer_id | c_first_name | c_last_name
---------------+---------------+--------------+-------------
          6885 | maps          | Joes         |
          4321 | tpcds         | Lily         |
          9527 | world         | James        |
          3769 | hello         |              |
(4 rows)
```

创建 customer_t2 表

```
postgres=#DROP TABLE IF EXISTS customer_t2;

postgres=#CREATE TABLE customer_t2
(
    c_customer_sk             integer,
    c_customer_id             char(5),
    c_first_name              char(6),
    c_last_name               char(8)
);
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

向表中插入数据

```
postgres=# INSERT INTO customer_t2 (c_customer_sk, c_customer_id, c_first_name) VALUES
(3769, 'hello', DEFAULT) ,
(6885, 'maps', 'Joes'),
(9527, 'world', 'James');
```

显示结果为：

```
INSERT 0 3
```

查看表中数据。

```
select * from customer_t2;

 c_customer_sk | c_customer_id | c_first_name | c_last_name
---------------+---------------+--------------+-------------
          3769 | hello         |              |
          6885 | maps          | Joes         |
          9527 | world         | James        |
(3 rows)
```

创建用户 lucy。

```
postgres=# DROP user IF EXISTS lucy;
postgres=# CREATE USER lucy WITH PASSWORD "Bigdata@123";
```

切换到 Lucy 用户。

```
postgres=# \c - lucy

Password for user lucy:
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "lucy".
```

创建 lucy shema 的表。

```
postgres=# DROP TABLE IF EXISTS lucy.mytable;

postgres=# CREATE TABLE mytable (firstcol int);
```

显示结果为：

```
CREATE TABLE
```

向表中插入数据。

```
postgres=# INSERT INTO mytable values (100);
```

当结果显示为如下信息，则表示插入数据成功。

```
INSERT 0 1
```

查看表中数据。

```
postgres=# SELECT * from mytable;
```

显示结果为：

```
 firstcol
----------
      100
(1 row)
```

退出数据库：

```
postgres=#\q
```

步骤 2：物理备份

openGauss 部署成功后，在数据库运行的过程中，会遇到各种问题及异常状态。openGauss 提供了 gs_basebackup 工具做基础备份。gs_basebackup 工具由操作系统用户 omm 执行。

在数据库备份之前创建存储备份文件的文件夹。

切换到 omm 用户，以操作系统用户 omm 登录数据库主节点。

```
su - omm
```

创建存储备份文件的文件夹。

```
mkdir -p /home/omm/physical/backup
```

如果数据库服务没有启动，就启动数据库服务(务必以操作系统用户 omm 启动数据库服务，如果没有请切换用户)。

```
gs_om -t start
```

结果显示为：

```
Starting cluster.
=========================================
=========================================
Successfully started.
```

将数据库进行物理备份。

```
gs_basebackup -D /home/omm/physical/backup -p 26000
```

参数-D directory 表示备份文件输出的目录，必选项。

结果显示为：

```
INFO:  The starting position of the xlog copy of the full build is: 1/D5000028. The slot minimum LSN is: 0/0.
begin build tablespace list
finish build tablespace list
begin get xlog by xlogstream
 check identify system success
 send START_REPLICATION 1/D5000000 success
 keepalive message is received
 keepalive message is received
 keepalive message is received
 keepalive message is received
 keepalive message is received
 keepalive message is received
 keepalive message is received
 keepalive message is received
```

切换到存储备份文件夹查看备份文件。

```
cd /home/omm/physical/backup
ls
```

显示如下：

```
backup_label        mot.conf     pg_errorinfo      pg_llog       pg_snapshots  pg_xlog
server.key
base                pg_clog      pg_hba.conf       pg_multixact  pg_stat_tmp   postgresql.conf
server.key.cipher
cacert.pem          pg_copydir   pg_hba.conf.bak   pg_notify     pg_tblspc     postgresql.conf.bak
server.key.rand
global              pg_csnlog    pg_hba.conf.lock  pg_replslot   pg_twophase   postgresql.conf.lock
gswlm_userinfo.cfg  pg_ctl.lock  pg_ident.conf      pg_serial     PG_VERSION    server.crt
```

步骤 3：物理备份恢复

当数据库发生故障时需要从备份文件进行恢复。因为 gs_basebackup 是对数据库按二进制进行备份，因此恢复时可以直接拷贝替换原有的文件， 或者直接在备份的库上启动数据库。

说明：

- 若当前数据库实例正在运行，直接从备份文件启动数据库可能会存在端口冲突，这时，需要修配置文件的 port 参数，或者在启动数据库时指定一下端口。

- 若当前备份文件为主备数据库，可能需要修改一下主备之间的复制连接。即配置文件中的 postgre.conf 中的 replconninfo1，replconninfo2 等。

当数据库发生问题需要从备份进行恢复时，步骤如下：

停止 openGauss((务必以操作系统用户 omm 停止数据库服务，如果没有请切换用户)。

```
gs_om -t stop
```

显示的结果为：

```
Stopping cluster.
=========================================
Successfully stopped cluster.
=========================================
End stop cluster.
```

清理原库中的所有或部分文件。

```
cd /gaussdb/data/
ls
```

查看数据库节点文件夹名称(文件夹名称是数据库安装时定义的，不同数据可能不同)。

```
db1
```

查看文件列表如下：

```
cd db1
ls

base                mot.conf     pg_errorinfo      pg_llog       pg_serial     PG_VERSION
postmaster.opts
cacert.pem          pg_clog      pg_hba.conf       pg_location   pg_snapshots  pg_xlog
server.crt
gaussdb.state       pg_copydir   pg_hba.conf.bak   pg_multixact  pg_stat_tmp   postgresql.conf
server.key
global              pg_csnlog    pg_hba.conf.lock  pg_notify     pg_tblspc     postgresql.conf.bak
server.key.cipher
```

```
gswlm_userinfo.cfg  pg_ctl.lock  pg_ident.conf     pg_replslot   pg_twophase   postgresql.conf.lock
server.key.rand
```

删除文件，对数据库文件进行破坏。

```
rm -rf *
ls
```

删除文件后，列表如下：

```
base    pg_clog     pg_csnlog     pg_llog      pg_multixact  pg_replslot  pg_snapshots  pg_tblspc
pg_xlog
global  pg_copydir  pg_errorinfo  pg_location  pg_notify     pg_serial    pg_stat_tmp   pg_twophase
```

或者为空：

```
[omm@ecs-ecs-c9bf db1]$
```

使用数据库系统用户权限从备份中还原需要的数据库文件，"/gaussdb/data/db1"中 db1 是数据库节点文件夹名称，不同数据库可能不同请查看确认。

```
cp -r /home/omm/physical/backup/.  /gaussdb/data/db1
```

备份时间大概需要几分钟，恢复后文件列表如下：

```
cd /gaussdb/data/db1
ls

backup_label.old    mot.conf      pg_hba.conf       pg_multixact  pg_tblspc            postgresql.conf.lock
server.key.rand
base                pg_clog       pg_hba.conf.bak   pg_notify     pg_twophase          postmaster.opts
cacert.pem          pg_copydir    pg_hba.conf.lock  pg_replslot   PG_VERSION           postmaster.pid
gaussdb.state       pg_csnlog     pg_ident.conf     pg_serial     pg_xlog              server.crt
global              pg_ctl.lock   pg_llog           pg_snapshots  postgresql.conf      server.key
gswlm_userinfo.cfg  pg_errorinfo  pg_location       pg_stat_tmp   postgresql.conf.bak  server.key.cipher
```

若数据库中存在链接文件，需要修改使其链接到正确的文件。

重启数据库服务器， 并检查数据库内容，确保数据库已经恢复到所需的状态。

```
gs_om -t start

Starting cluster.
=========================================
=========================================
Successfully started.
```