# 第五章 并发控制与分布式事务

## 引言

并发控制

## 5.1 事务隔离和两阶段提交

### 5.1.1 事务隔离

我们首先对事务的隔离进行说明。我们知道，并发执行的事务之间的相互影响可能导致数据库状态的不一致性，即使各个事务能保持状态的正确性，而且也没有任何故障发生。SQL 标准用三个必须在并行的事务之间避免的现象定义了四个级别的事务隔离。 这些不希望发生的现象是：

- 脏读（dirty reads）：

  一个事务读取了另一个未提交的并行事务写的数据。

- 不可重复读（non-repeatable reads）：

  一个事务重新读取前面读取过的数据， 发现该数据已经被另一个已提交的事务修改过。

- 幻读（phantom read）：

  一个事务重新执行一个查询，返回一套符合查询条件的行， 发现这些行因为其他最近提交的事务而发生了改变。

这四种隔离级别和对应的行为在表 7-1 中描述。

| 隔离级别 | 脏读（Dirty Read） | 不可重复读（NonRepeatable Read） | 幻读（Phantom Read） |
| --- | --- | --- | --- |
| 读未提交（Read uncommitted） | 可能 | 可能 | 可能 |
| 读已提交（Read committed） | 不可能 | 可能 | 可能 |
| 可重复读（Repeatable read） | 不可能 | 不可能 | 可能 |
| 可串行化（Serializable ） | 不可能 | 不可能 | 不可能 |

表 7-1 四种隔离级别

### 5.1.2 锁机制

数据库往往由多个用户同时使用，数据库在面临多个用户并行地读写数据时会出现一些问题，如前面提到的脏读，幻度，重复读等，破坏数据库的一致性。类似于操作系统的加锁一样，我们也可以对数据库中的共享资源进行加锁。当事务在对某个数据对象进行操作前，先向系统发出请求，申请获得锁。加锁后事务就对该数据对象有了一定的控制，在该事务释放锁之前，其他的事务不能对此数据对象进行更新操作。

锁主要分为两种：互斥锁和共享锁。互斥锁独占资源，不允许其他事务对资源进行读写操作；共享锁允许事务间并行的对资源进行写操作，但只允许同时存在一个事务对资源进行读操作。这两种锁都可以作用在如下级别上：

Access：锁定整个表模式。

Rows：　锁定单独的元组。

### 5.1.3 MVCC

MVCC（Multi-Version Concurrency Control）的中文意思是多版本并发控制。使用 MVCC 可以避开传统数据库的加锁方法，对检索(读)数据的锁请求与写数据的锁请求不冲突，所以读不会阻塞写，而写也从不阻塞读。最大限度地减小锁竞争的可能性以提高数据库的性能。可以把锁 MVCC 看做锁的一种妥协，它在很多情况下避免使用了锁。而对于那些无法轻松接受 MVCC 行为的执行，可以使用锁来进行操作。

MVCC 的实现方法有两种：

1.写新数据时，把旧数据移到一个单独的地方，如回滚段中，其他人读数据时，从回滚段中把旧的数据读出来；

2.写数据时，旧数据不删除，而是把新数据插入。

PostgreSQL 数据库使用第二种方法，而 Oracle 数据库和 MySQL 中的 innodb 引擎使用的是第一种方法。

### 5.1.4 两阶段提交协议

在分布式系统中，各个节点之间在物理上相互独立，通过网络进行沟通和协调。由于存在事务机制，可以保证每个独立节点上的数据

操作可以满足 ACID。但是， 相互独立的节点之间无法准确的知道其他节点中的事务执行情况。所以从理论上讲，两台机器理论上无法达到一致的状态。如果想让分布式部署的多台机器中的数据保持一致性，那么就要保证在所有节点的数据写操作，要不全部都执行，要么全部的都不执行。但是，一台机器在执行本地事务的时候无法知道其他机器中的本地事务的执行结果。所以他也就不知道本次事务到底应该 commit 还是 roolback。所以，常规的解决办法就是引入一个“协调者”的组件来统一调度所有分布式节点的执行。

两阶段提交协议是很常见的解决分布式事务的方式，他可以保证分布式事务中，要么所有参与的进程都提交事务成功，要么都取消事务，这样做可以在分布式环境中保持 ACID 中 A(原子性)。

在两阶段提交协议中，包含了两种角色：协调者与参与者。参与者就是实际处理事务的机器，而协调者就是其中一台单独的处理分布式事务的机器。

该算法分为两个阶段：

1.投票阶段

2.提交阶段

阶段 1：请求阶段（commit-requestphase，或称表决阶段，voting phase）

在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。在表决过程中，参与者将告知协调者自己的决策：

同意（事务参与者本地作业执行成功）或取消（本地作业执行故障）。这里的取消是指该参与者所在的机器没有准备好，或者出现了故障。因此无法执行该事务。

阶段 2：提交阶段（commitphase）

在该阶段，协调者将基于第一个阶段的投票结果进行决策，进行提交或取消。当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。以下是两种不同情况下的说明。

当协调者节点从所有参与者节点获得的相应消息都为”同意”时，协调者节点向所有参与者节点发出”正式提交(commit)”的请求。

如果任一参与者节点在第一阶段返回的响应消息为“中止”，或者协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时，协调者节点向所有参与者节点发出”回滚操作(rollback)”的请求。：

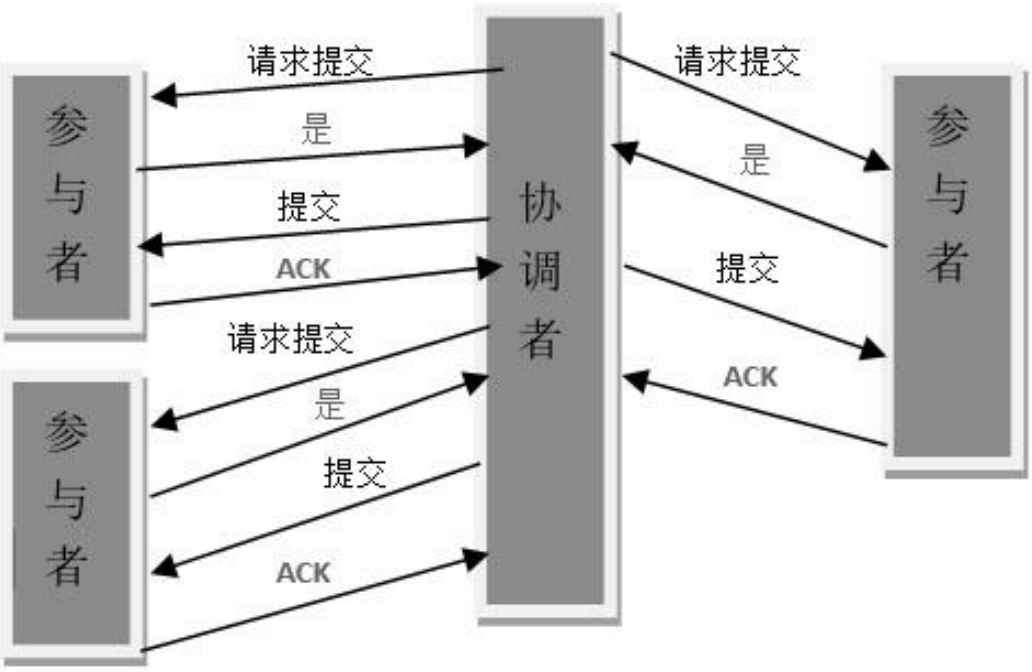


图 7-2 所有节点返回“同意”

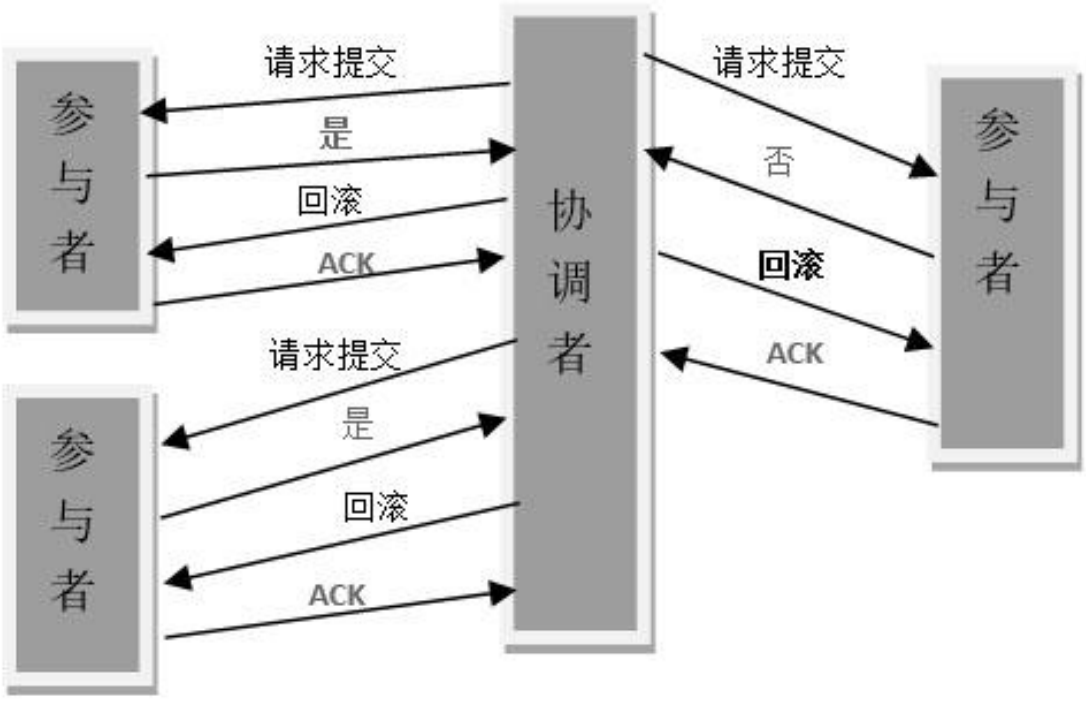


图 7-3 某一节点返回“中止”

两阶段提交协议一定程度上能够保证操作的原子性。但是不幸的

是二阶段提交协议还是会出现一些问题：

1） 同步阻塞：在提交过程中参与节点执行的是事务阻塞型的操作。当参与者占有公共资源时，其他事务操作被阻塞。

2） 单点故障：在提交的阶段参与节点会锁定事务资源，如果协调者发生故障，由于接受不到协调者的响应信息，所有的参与者就会一直阻塞下去。

3） 数据不一致：当协调者向参与者发送 commit 请求之后，发生了局部网络异常或者在发送 commit 请求过程中协调者发生了故障，会导致一部分参与者执行提交操作而另一部分由于没有接收到信息而没有进行任何操作。那么就会导致分布式数据库出现不一致的问题。

## 5.2 Greenplum 中的实现

- 事务隔离

在 PostgreSQL 里，你可以请求四种可能的事务隔离级别中的任意一种。 但是在内部，实际上只有两种独立的隔离级别，分别对应读已提交和可串行化。 如果你选择了读未提交的级别，实际上你用的是读已提交， 在你选择可重复的读级别的时候，实际上你用的是可串行化，所以实际的隔离级别可能比你选择的更严格。 这是 SQL 标准允许的：四种隔离级别只定义了哪种现象不能发生，但是没有定义那种现象一定发生。 PostgreSQL 只提供两种隔离级别的原因是，这是把标准的隔离级别与多版本并发控制架构映射相关的唯一的合

理方法。用户可以使用 set transaction 命令在事务中改变隔离级别。

读已提交（Read Committed） 是 PostgreSQL 里的缺省隔离级别。 当一个事务运行在这个隔离级别时， 一个 SELECT 查询只能看到查询开始之前提交的数据而永远无法看到未提交的数据或者是在查询执行时其他并行的事务提交做的改变。 （不过 SELECT 的确看得见同一次事务中前面更新的结果。即使它们还没提交也看得到。）

实际上，一个 SELECT 查询看到一个在该查询开始运行的瞬间该数据库的一个快照。 请注意两个相邻的 SELECT 命令可能看到不同的数据，哪怕它们是在同一个事务里， 因为其它事务会在第一个 SELECT 执行的时候提交。

可串行化（Serializable） 级别提供最严格的事务隔离。 这个级别模拟串行的事务执行， 就好象事务将被一个接着一个那样串行的，而不是并行的执行。 不过，使用这个级别的应用必须准备在串行化失败的时候重新启动事务。

当一个事务处于可串行化级别， 一个 SELECT 查询只能看到在事务开始之前提交的数据而永远看不到未提交的数据或事务执行中其他并行事务提交的修改。 （不过，SELECT 的确看得到同一次事务中前面的更新的效果。即使事务还没有提交也一样。） 这个行为和读已提交级别是不太一样，它的 SELECT 看到的是该事务开始时的快照，而不是该事务内部当前查询开始时的快照。 这样，一个事务内部后面的 SELECT 命令总是看到同样的数据。

PostgreSQL 通过对隔离级别的控制，从而可以保证数据库并发执

行的一致性。而对隔离级别的控制，主要通过锁和 MVCC 两种机制来实现。

❖ 锁

Greenplum 中的锁基本上与 PostgreSQL 的锁一样。但由于 Greenplum 是一个分布式的数据库，锁机制较为复杂一些。因为需要统一所有节点上的锁，这个锁的控制是在 Master 上执行的。

Greenplum 支持八种锁，当执行特定的命令会隐式申请锁。这八种锁都是表级别锁：

| 锁类型 | 说明 |
| --- | --- |
| ACCESS SHARE | 查询命令（Select command）将会在它查询的表上获取“Access Shared”锁,一般地，任何一个对表上的只读查询操作都将获取这种类型的锁。 |
| ROW SHARE | “Select for update”和“Select for share”命令将获得这种类型锁,并且所有被引用但没有 FOR UPDATE 的表上会加上“Access shared locks”锁。 |
| ROW EXCLUSIVE | “Update,Delete,Insert”命令会在目标表上获得这种锁，并且在其它被引用的表上加上“Access shared”锁,一般地，更改表数据的命令都将在这张表上获得“Row exclusive”锁。 |
| SHARE UPDATE EXCLUSIVE | 这种模式保护一张表不被并发的模式更改和 VACUUM;“Vacuum(without full), Analyze ”和 “Create index concurrently”命令会获得这种类型锁。 |
| SHARE | 这种模式保护一张表数据不被并发的更改;“Create index”命令会获得这种锁模式。 |
| SHARE ROW EXCLUSIVE | 任何 Postgresql 命令不会自动获得这种锁。 |
| EXCLUSIVE | 这种锁模式仅能与 Access Share 模式并发,换句话说，只有读操作可以和持有“EXCLUSIVE”锁的事务并行；任何 Postgresql 命令不会自动获得这种类型的锁； |
| ACCESS EXCLUSIVE | 与所有模式锁冲突，这种模式保证了当前只有一个事务访问这张表;“ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL” 命令会获得这种类型锁，在 |

| | |
|---|---|
| | Lock table 命令中，如果没有申明其它模式，它也是缺省模式。 |

表 7-2 DDL 语句处理过程

这里对 row share lock 做一下说明，虽然 select for update 和 select for share 都会申请这种锁，但表现略有不同：

① select for update 使那些被检索出来的行被锁住，就像要更新它们一样，从而避免在当前事务执行的过程中被其他事物修改。任何企图对锁住的元组的 update，delete 以及 select for update 操作都会被阻塞。同样，其他事务如果执行了 update，delete 或者 select for update 都会阻塞当前事物。

② select for share 在每个被检索出来的行上要求一个共享锁。一个共享锁阻塞其他事物在这些行上执行 update，delete 和 select for update，却不阻止他们执行 select for share。

除了使用命令隐式申请锁之外，Greenplum 也支持在事务中显式申请锁。使用 lock * in 命令可以显示申请锁。

Greenplum 的锁之间具有冲突关系，冲突的锁不能被同时申请。这几种锁的冲突如表 7-3 所示：

| 锁类型 | ACCESS SHARE | ROW SHARE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE |
|---|---|---|---|---|---|---|---|---|
| ACCESS SHARE | | | | | | | | × |
| ROW SHARE | | | | | | | × | × |
| ROW EXCLUSIVE | | | | | | × | × | × |
| SHARE UPDATE EXCLUSIVE | | | | | × | × | × | × |
| SHARE | | | | × | × | × | × | × |
| SHARE ROW EXCLUSIVE | | | × | × | × | × | × | × |
| EXCLUSIVE | | × | × | × | × | × | × | × |
| ACCESS EXCLUSIVE | × | × | × | × | × | × | × | × |

表 7-3 锁冲突关系

❖ MVCC

前面提到 Greenplum 对 MVCC 的实现方式是并不删除旧数据，而是把新数据插入。与 oracle 数据库和 MySQL 中的 innodb 引擎相比较，PostgreSQL 的 MVCC 实现方式的优缺点如下。

优点：

1.）事务回滚可以立即完成，无论事务进行了多少操作；

2.）数据可以进行很多更新，不必像 Oracle 和 MySQL 的 Innodb 引擎那样需要经常保证回滚段不会被用完，也不会像 oracle 数据库那样经常遇到“ORA-1555”错误的困扰；

缺点：

1.）旧版本数据需要清理。PostgreSQL 清理旧版本的命令成为 Vac

uum；

2.）旧版本的数据会导致查询更慢一些，因为旧版本的数据存在于数据文件中，查询时需要扫描更多的数据块。

PostgreSQL 中的 MVCC 实现原理可简单概括如下，见图 7-1：

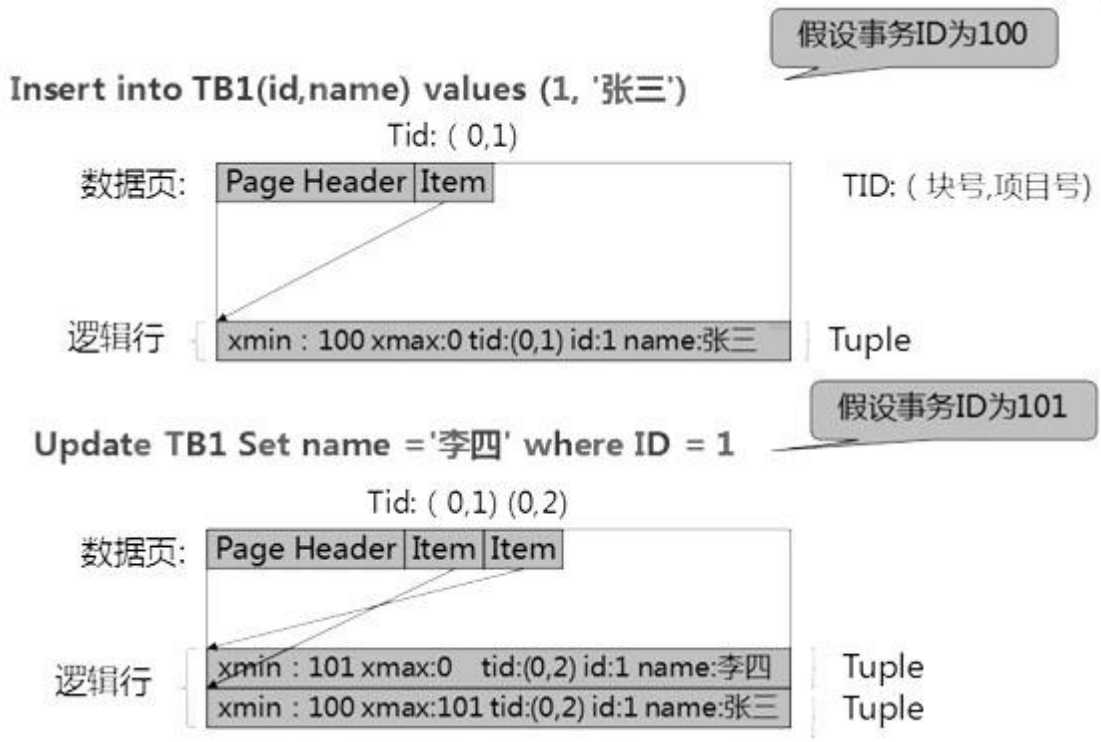


图 7-1 MVCC 示例

1）数据文件中存放同一逻辑行的多个行版本（称为 Tuple）

2）每个行版本的头部记录插入以及删除该行版本的事务的 ID（分别称为 xmin 和 xmax），xmax 为 0 则认为改行记录没有进行过删除操作，更新操作被看做一次删除和一次插入操作。

3）每个事务的状态（运行中，中止或提交）记录在 pg_clog 文件中

4）根据上面的数据并运用一定的规则每个事务只会看到一个特定

的快照。

## 5.3 数据库管理实验导引

### 5.3.1 数据库管理实战

1. 锁机制

在第一个实验中，我们分别打开两个终端，其中一个进行 select 操作，获取 access share lock，另一个进行 alter 操作，获取 access exclusive lock。

1）新建表 test_05，开启一个事务，进行一次 select 操作

```
testdb=# begin;
BEGIN
testdb=# select * from test_05;
 id
----
(0 rows)
```

2）另建一个终端，再次开启一个事务，进行一次 select 操作和 alt 操作

```
testdb=# begin;
BEGIN
testdb=# select * from test_05;
 id
----
(0 rows)

testdb=# alter table test_05 add column uname varchar(10);
```

可以发现，对于同种锁，并不一定冲突，这也符合表中所列的冲突情况。但是对于冲突的锁，如 ACCESS SHARE 和 ACCESS EXCLUSIVE，sql 语句一直在等待，直到第一个事务进行了提交

操作。

3）获取 test_05 的 oid，然后在 pg_lock 里查询到锁信息

```
testdb=# select oid from pg_class where relname = 'test_05';
  oid
-------
 57462
(1 row)

testdb=# select locktype,database,relation,pid,mode from pg_locks where relation
='57462';
 locktype | database | relation | pid  |        mode
----------+----------+----------+------+---------------------
 relation |    24606 |    57462 | 3520 | AccessShareLock
 relation |    24606 |    57462 | 3520 | AccessExclusiveLock
 relation |    24606 |    57462 | 3530 | AccessShareLock
 relation |    24606 |    57462 | 3553 | AccessShareLock
 relation |    24606 |    57462 | 3523 | AccessShareLock
 relation |    24606 |    57462 | 3555 | AccessShareLock
 relation |    24606 |    57462 | 3525 | AccessShareLock
(7 rows)

testdb=# select datname,procpid,usename,current_query from pg_stat_activity wher
e procpid in (3520,3530,3553,3523,3555,3525);
 datname | procpid | usename |
  current_query
---------+---------+---------+-----------------------------------------------
---------------------------------------------------------------
 testdb  |    3520 | gpadmin | alter table test_05 add column uname varchar(10);
 testdb  |    3530 | gpadmin | select datname,procpid,usename,current_query from
 pg_stat_activity where procpid in (3520,3530,3553,3523,3555,3525);
```

4）下面对显式加锁进行试验

```
testdb=# lock test_05 in ROW SHARE mode;
LOCK TABLE
testdb=# select locktype,database,relation,pid,mode from pg_locks where relation='57462';
 locktype | database | relation | pid  |        mode
----------+----------+----------+------+---------------------
 relation |    24606 |    57462 | 3520 | AccessShareLock
 relation |    24606 |    57462 | 3520 | AccessExclusiveLock
 relation |    24606 |    57462 | 3530 | AccessShareLock
 relation |    24606 |    57462 | 3530 | RowShareLock
 relation |    24606 |    57462 | 3553 | AccessShareLock
 relation |    24606 |    57462 | 3553 | RowShareLock
 relation |    24606 |    57462 | 3523 | AccessShareLock
 relation |    24606 |    57462 | 3555 | AccessShareLock
 relation |    24606 |    57462 | 3555 | RowShareLock
 relation |    24606 |    57462 | 3525 | AccessShareLock
(10 rows)
```

在加锁之后再次查看锁信息，可以看到与原来相比多了 RowShar

eLock

2. MVCC

下面我们来对 MVCC 的特性做一些实验：

1）首先创建一张表：

```
testdb=# create table test01(id int,name varchar(20)) distributed by(id);
CREATE TABLE
```

2）在事务①进行一次事务插入操作，暂时不进行提交：

```
testdb=# begin;
BEGIN
testdb=# insert into test01 values(1,'Mike');
INSERT 0 1
testdb=#
```

3）在事务②中设置事务隔离级别为缺省的 READ COMMITTED 级别，对表中的数据进行查询：

```
testdb=# begin;
BEGIN
testdb=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
testdb=# select * from test01;
 id | name
----+------
(0 rows)
```

4）将事务①进行提交：

```
testdb=# begin;
BEGIN
testdb=# insert into test01 values(1,'Mike');
INSERT 0 1
testdb=#  commit;
COMMIT
```

5）在事务②再次进行一次查询：

```
testdb=# begin;
BEGIN
testdb=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
testdb=# select * from test01;
 id | name
----+------
(0 rows)

testdb=# select * from test01;
 id | name
----+------
  1 | Mike
(1 row)
```

可以发现，事务②中进行的两次查询出现了不同的结果，出现了幻读。这是由于我们设定的隔离级别造成的。数据库会根据我们设定的隔离级别调用不同的快照，用于不同形式的可见性判断。

3. 角色

在本次实验中，分别创建了 undergraduate 和 class1 两个角色，并将其加入到配置文件中。由于将 class1 置为 undergraduate 的 member，并给 class1 赋上 t1 表的 select 权限，那么 undergraduate 应该会自动拥有同样的权限。我们在实验中进行了验证。

1） 创建 undergrade 和 class1 角色

```
testdb=# create role undergraduate LOGIN password '123456';
NOTICE:  resource queue required -- using default resource queue "pg_default"
CREATE ROLE
testdb=# create role class1 LOGIN password '123456';
NOTICE:  resource queue required -- using default resource queue "pg_default"
CREATE ROLE
```

2） 在 pg_hba.conf 加上新的 role 的信息

```
local    testdb      undergraduate            trust
local    testdb      class1            trust
```

3） 将 class  1 设为 undergraduate 的 member

```
testdb=# grant class1 to undergraduate;
GRANT ROLE
```

4） 给 class1 赋上 select on t1 的权限（t1 是建立的一张临时表）

```
testdb=# grant select on t1 to class1;
GRANT
```

5） 使用 undergraduate 登录数据库，查看 t1 表（需要重启数据库使修改生效）

```
[gpadmin@localhost ~]$ psql -U undergraduate
psql (8.3devel)
Type "help" for help.

testdb=> select * from t1 where id = 1;
 id |         name
----+----------------------
  1 | test
(1 row)
```

从结果可以看出，undergraduate 是拥有 t1 表的 select 权限的。这说明一个成员是拥有它的子成员的全部权限的。

### 5.3.2 数据库管理实验要求

1、使用 create role 命令创建一个拥有创建数据库、创建用户、登录权限， 且有效期到 2016 年 12 月 19 日， 密码为‘test’的用户； 使用 grant 和 revoke 命令添加和回收权限；

2、申请锁，通过 pg_locks 系统表查询表的锁信息

3、 查看 Greenplum 的数据目录结构，理清楚 base、 global、pg_log、 pg_clog、 pg_xlog 等目录存储的内容。

4、了解 greenplum 中的 toast 机制，通过不断增大字段占用空间，探索在何种情况下会触发 toast 机制，生成 toast 表，是否有 toast 表可以从 pg_class 中进行查询。