Lab9 PostgreSQL事务处理初探

本次实验带领大家初步了解PostgreSQL如何处理并发执行的事务，保证事务运行过程中的隔离性与数据一致性。

由于PostgreSQL事务处理模块具体的原理与实现十分复杂，因此本次实验内容仅包含PostgreSQL多版本并发控制（MVCC）与快照机制部分最基础的原理与实现，
以学习为主，供大家对PostgreSQL并发控制原理有初步的认识。

如果想深入学习PostgreSQL事务处理原理与实现，可以阅读参考资料中的书籍和文档。

一、事务的ACID属性与隔离级别

事务的ACID属性;

- 原子性（Atomicity）：一个事务中的操作要么全部都做，要么全都不做
- 一致性（Consistency）：一个事务将数据库从一个一致状态转化为另一个一致状态（满足各种约束，保证数据一致）
- 隔离性（Isolation）：在并发环境下，事务与其他并发的事务运行时是相互隔离的，并发的事务之间不会相互影响
- 持久性（Durability）：事务提交之后，对数据库中状态的变更是永久性的

通过一个例子展示事务的ACID属性：
假设有一个事务，包含银行转账逻辑：A B两个账户分别各有1000元，现需要从A账户转100元给B账户，该事务可抽象为下述操作序列

1. 从A账户里减去100元
2. 为B账户里增加100元

先考虑事务的原子性。如果事务仅执行了第一步操作后，系统由于各种原因导致宕机，重新启动后发现了A账户为900，B账户为1000。为了维护事务的原子性，
数据库系统需要将“A账户减去100元”这个操作回滚，恢复到事务执行前的状态，就好像转账事务未发生一样。也就是说数据库系统要保证一个事务中的操作，
要么全部都做，要么全都不做，保证事务的原子性。

再考虑事务的隔离性。如果在转账事务执行完第一步还未执行第二步时，有一个并发的读取事务分别读取A、B账户的余额，发现此时A为900、B为1000，AB账户
的总存款凭空少了100元。显然读取事务读到了不一致的数据，主要原因是并发的转账事务影响到了读取事务的执行，违背了事务的隔离性。

上述例子中，读取事务读到了不一致的数据状态，也违反了事务的一致性。事务的一致性也有另一种解释。数据库中存在对数据一致的各种约束，例如账户余额不能少于0元；
如果某次转账会导致转出的账户余额为负，则会触发约束条件，因此事务无法提交，维护事务的一致性。

事务持久性是指，事务提交后，对数据的修改不会由于断电、宕机的意外事故导致丢失。目前传统的OLTP数据库系统通常是依靠WAL（预写式日志），通过将事务的日志

持久化到磁盘等非易失的存储设备中，以保证事务的持久性。

总的来说，数据库系统需要并发控制机制与基于日志的故障恢复技术来维护事务的ACID属性。本次实验主要探讨postgreSQL的多版本并发控制技术，日志与故障恢复技术不在本次lab的范围内。

二、事务的隔离级别

通常情况下，在一些典型的OLTP应用场景中，事务的原子性、一致性与持久性是不能违背的，否则可能会出现严重的数据异常且难以恢复。不过，不同的应用场景下对事务的隔离性可以做一定的“放松”。
ANSI SQL92标准定义了数据异常的概念：脏读（dirty read）、不可重复读（unrepeatable read）、幻读（phantom read）；并根据禁止数据异常的能力定义了不同的隔离级别：读未提交（read uncommited）、读已提交（read committed）、可重复读（repeatable read）与可串行化（serializable）。

1.读未提交

顾名思义，读未提交隔离级别表示，当事务读取一个数据对象时，不用考虑创建该数据对象的事务的状态，直接读取即可。 在该隔离级别下，并发事务执行并不受限制，因此事务执行效率很高，但是会出现脏读、不可重复读、幻读的异常现象。

脏读（dirty read） 示例：

```
T1          T2
  begin
              begin
  W(A)
              R(A)
  abort
              commit
```

上述两个并发执行的事务T1、T2，T1修改元素A后，T2读到了T1修改后A的值。但是由于事务T1由于某种原因导致了abort（没有提交），因此需要将A的值回滚为修改之前的值，
这就导致T2读到了一个实际上并不存在的值。这就是 **脏读** 数据异常现象。

2.读已提交

为了避免脏读数据异常，“读已提交”隔离级别对读取的数据做了一定的限制：只能读取已经提交的数据。这样不会出现读取一个并不存在的值，避免出现“脏读”异常。
但是“读已提交”依然会出现不可重复读与幻读的异常现象。

不可重复读（phantom read）示例：

```
T1          T2
  begin
              begin
  R(A)
              W(A)
              commit
  R(A)
  commit
```

如示例所示。T1先读取了A的旧值，T2写入了A的新值并提交，之后T1再次读取A的值，此时读取的是T2写入新值。
该执行序列满足“读已提交”隔离级别，但是对于T1来说，前后读取同一个数据对象A得到的值却不同，出现了 **不可重复读** 数据异常。

“读已提交”由于可以保证最基本的隔离性能够满足多数应用的需求，并且能提供较高的并发度，因此是多数商业数据库系统默认的隔离级别。

3.可重复读

“可重复读”隔离级别保证整个事务执行期间，前后读取同一个数据对象的值均相同。
但是“可重复读”隔离级别会出现“幻读”异常。

幻读（phantom read）示例：

```
T1              T2
  begin
                  begin
  R(condition)
                  W(condition)
                  commit
  R(conditon)
  commit
```

如示例所示，T1首先根据where查询条件，读取复合条件的元组；然后，事务T2插入新的元组或者更新旧的元组，这些元组满足T1查询的where条件；
T1后续以相同的谓词条件读取数据，会发现读出了新的数据（假设T2插入了复合谓词条件的数据）。对于T1来说，前后两次使用相同的谓词条件， 但是读取到的数据却并不相同，因此产生了 **幻读** 异常。

tip：不可重复读与幻读的异同：

首先，这两种异常对于事务T1而言，都是先读取了数据，之后因事务T2“写”数据，导致T1再次读数据时前后两次数据不一样，发生了异常。
但是，不可重复读异常针对的是一个 **存在的确定的一个数据对象** ，T1前后两次读取该数据对象都存在，T2的写操作改变了已经存在的数据对象的值，
导致T1读取该对象的值不相同；而幻读异常针对的是一个 **满足谓词条件的数据集合**，结果集并不确定，T2的“写”操作可以是插入一个新的数据对象，
导致T1根据前后两次读取了不同的数据集合。

4.可串行化

可串行化是最高的隔离级别。在可串行化隔离级别下，事务并发执行后的数据状态，等价于将事务按照某个串行执行顺序执行的结果。
事务在可串行化调度下，可以避免脏读、不可重复读、幻读数据异常。

总结一下，ANSI SQL92标准中隔离级别与数据异常的关系如下表所示：

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能

隔离级别	脏读	不可重复读	幻读
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

三、多版本并发控制理论基础

为了满足事务的隔离性，避免并发事务在执行的过程中出现数据异常现象，数据库系统需要一套并发控制协议以对事务访问数据做一定的限制。
在水杉平台“当代数据管理系统”课程第八章“数据正确性与事务处理”的课程中，讲解了两阶段封锁协议是如何处理并发事务间的访问数据冲突。
两阶段封锁协议通过对需要访问的数据对象加锁，并将整个事务划分为加锁阶段和放锁阶段，以获得访问数据对象的权限，限制并发事务在事务粒度上访问同一个数据对象。

在两阶段封锁协议下，当并发的两个事务存在读写冲突时（访问同一个数据对象），由于封锁协议，一个事务的读（写）操作需要等待另一个事务的写（读）锁释放后，才可以继续执行。
对于只读事务（不存在写操作），访问数据访问广、持锁时间较长，会频繁的与并发的写事务出现冲突，导致严重的性能下降。

多版本并发控制协议有效地解决了读写并发、只读事务优化的问题。

多版本并发控制举例：

```
T1(TS(T1)=1)     T2(TS(T2)=2)
     begin
                      begin
     R(A)
                      W(A)

     R(A)
     commit
                      commit
```

数据对象A的初始版本为:

version	data	begin	end
A_0	10	0	-

事务T1开始时，系统会给其分配一个时间戳TS(T1)=1。在第一次读取A时，遍历到数据版本A_0，其开始时间戳begin=0，结束时间戳end=无穷，
表示A_0的“可见时间”为 0 ~ 无穷。 显然T1的时间戳落在了A_0的“可见范围内”，因此T1第一次读取A读到了A_0。

后续T2开始时，系统为其分配一个时间戳TS(T2)=2。当T2写A时，由于采用多版本并发控制协议，T2并不会直接将A的旧值覆盖掉，而是创建一个新的版本A_1，
保留A_0并将其end时间戳置为2，表示A_0 版本对于时间戳大于2的事务不可见。

T2执行W(A)后数据库状态如下：

version	data	begin	end
A_0	10	0	2
A_1	20	2	-

当T1第二次读取R(A)时，依然根据时间戳TS(T1)=1遍历A的版本链，发现版本A_0的begin - end时间戳范围包含T1的时间戳，因此A_0对事务T1可见。
T1第二次读取A时，读到的依然是A_0，T2创建的新版本A_1对T1不可见。

通过上述多版本并发控制示例可以看到，多版本并发控制协议会给事务分配一个用于元组版本可见性判断数据结构（在上述示例中是“时间戳”，在PostgreSQL中是快照，下文会讲）。
事务根据这个“时间戳”（postgresql中是快照），会读取对自己“可见”的元组版本；当更新元组时，不会在旧版本上直接覆盖新值，而是创建一个新版本，并对旧版本的时间戳进行修改，标识版本的“可见范围”。

通过多版本并发控制协议，并发的两个事务对同一个数据对象 **读写互不阻塞**，相较两阶段封锁协议提高了并发度。
由于其 **读写互不阻塞** 的优秀特性，目前很多商业数据库以经支持了多版本并发控制，例如postgreSQL、mysql、oracle等。

接下来内容，将对PostgreSQL数据库多版本并发控制原理进行讲解。

四、PostgreSQL并发控制基础数据结构与操作

经过上一节对多版本并发控制协议基础理论的讲解，可以有初步的认识：

- 事务需要有一个数据结构，标识该事务的“可见范围”
- 每个数据对象的每个版本需要利用“时间戳”决定版本的“可见范围”

在上一节的例子中，每个事务在开始时获取一个时间戳，标识事务的“可见范围”，每个版本在被创建和创建更新版本时，
会记录相应时间戳，以确定版本的“可见范围”。

PostgreSQL也采用了类似的思路，在事务开始时，系统会为事务分配一个单调递增的事务号，与前文示例的多版本并发控制协议不同，
PostgreSQL并非将事务号作为自己“可见范围”标准，而是利用快照作为标识“可见范围”的数据结构。

PostgreSQL快照数据结构（简化，只列出了最核心的数据项，更多内容请查阅“参考资料”）：

```
snapshot{
  xmin
```

```
  xmax
  active_trans_list
}
```

下面解释各数据项的含义：

- xmin表示事务获取该快照时，事务号小于xmin的事务已经全部结束了
- xmax表示事务获取该快照时，事务号大于等于xmax的事务还未开始或者正在运行（还未结束）
- active_trans_list表示事务获取快照时，事务号在 [xmin, xmax) 之间的处在活跃状态的事务列表

快照数据结构记录了事务获取快照时，系统中事务的状态信息，事务可以判断出来在获取快照时哪些事务已经结束（事务号小于xmin，或者事务号不在active_trans_list内）；
哪些事务还在运行（事务号大于等于xmax，或者事务号在active_trans_list内）。

下面进行实际测试。

在本地启动postgreSQL服务（见lab6启动方法），打开1个sql shell连接本地postgreSQL，相当于创建1个连接postgreSQL的会话。

注：建议优先使用本地数据库进行实验。如果在水杉平台terminal中访问阿里云postgreSQL数据库，每位同学请使用自己的私有数据库（stu+学号）不要使用新创建的数据库。 如果使用阿里云数据库，可能有多位同学同时建立多个会话，并发运行事务，可能会导致事务的快照信息不是很容易理解，在进行分析时，关注自己启动的事务的事务号与快照即可。 如果使用阿里云postgreSQL数据库，后文中启动sql shell等价于启动水杉平台的terminal建立连接，连接数据库操作见lab4

目前水杉阿里云postgreSQL不支持安装pageinspect扩展，元组头部信息无法展示。本次实验示例展示了完整的信息，本地无法运行postgreSQL的同学如果想实际运行下SQL语句，可以借用下别的同学的电脑操作一下实验

输入下述命令：
设置当前会话中事务的隔离级别是serializable。

```
set session characteristics as transaction isolation level
serializable;
```

在sql shell中启动一个事务：

```
testdb=# begin;
BEGIN
```

查询当前事务的事务号与快照（不等于885很正常，不同的pg数据库事务号不一定相同）：

```
testdb=*# select * from txid_current();
 txid_current
--------------
          885

testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
```

```
 885:885:
(1 行记录)
```

表示该事务当前的事务号为885，其获取到的快照是 (xmin:885 xmax:885 active_trans_list: 空)

不要让这个事务结束，我们再启动另外一个sql shell，创建另一个连接postgreSQL数据库的会话，依次输入如下命令：

```
testdb=# set session characteristics as transaction isolation level serializable;
SET

testdb=# begin;
BEGIN

testdb=*# select * from txid_current();
 txid_current
--------------
          886

testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 885:885:
(1 行记录)

testdb=*# end;
COMMIT
```

启动一个事物，该事务的事务号为886（相较上一个递增1），其快照为 (xmin:885 xmax:885 active_trans_list: 空)。
最后输入了"end;"，将该事务结束（提交）。

在第二个sql shell中，继续输入如下命令：

```
testdb=# begin;
BEGIN
testdb=*# select * from txid_current();
 txid_current
--------------
          887
(1 行记录)


testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 885:887:885
(1 行记录)
```

上述命令开启新的一个事务，其事务号为887，其快照为 (xmin:885 xmax:887 active_trans_list: 885)。
表示对于该事务来说，事务号小于885的事务已经结束，大于等于887的事务还在运行或者没有开始，

而介于885 - 887之间的事务，886号事务已经结束，885号事务还在运行，因此885在快照的活跃事务列表中。

事务在开始运行时获取快照，在快照中已经结束的事务产生的影响对该事务“可见”，而在快照中处于活跃状态的事务的操作对当前事务“不可见”。

上文中讲解了事务的快照数据结构，接下来讲解postgreSQL元组上有关事务“可见范围”的数据结构。

postgreSQL采用多版本并发控制协议，其实现方式与上文提到的基础的多版本并发控制协议类似。每个元组都存在“头部信息”，
里面存放了很多与该元组版本相关的数据结构，此次lab仅关注与并发控制最紧密相关的数据项: t_xmin, t_xmax。
当对元组进行更新时，会创建一个新的元组版本，并更新“头部信息”中与元组可见性相关的数据项，用来进行元组可见性判断。

获取元组的数据可以通过select from where语句查询得到，但是获得元组的“头部信息”需要插件 pageinspect。

首先，将之前开启的事务全部关闭（end;），在其中一个sql shell中执行下述命令：

在本次实验的数据库中创建pageinspect扩展（目前阿里云postgreSQL数据库并不支持创建extension）：

```
testdb=# create extension pageinspect;
CREATE EXTENSION
```

创建用来进行实验的表trans，包含id、data属性，id为主键：

```
testdb=# create table trans (id int primary key, data int);
CREATE TABLE
```

运行下述命令，查找trans表第0块数据页面的内容：

```
testdb=# select * from heap_page_items(get_raw_page('trans',0));
错误:  block number 0 is out of range for relation "trans"
```

报了个错误，不要慌，正常现象，因为trans是刚创建的，还没有插入数据，自然没有数据页面。

接下来开始真正的操作：

首先，开始一个事务，并查看事务号：

```
testdb=# begin;
BEGIN
testdb=*# select * from txid_current();
 txid_current
--------------
          893
(1 行记录)
```

插入一条数据，(id = 1, data = 1)：

```
testdb=*# insert into trans values (1, 1);
INSERT 0 1
```

运行下述命令，查看trans表第0个数据页的内容（没有安装pageinspect插件的没法运行下述命令）：

```
testdb=*# select * from heap_page_items(get_raw_page('trans',0));
 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 | t_ctid
| t_infomask2 | t_infomask | t_hoff | t_bits | t_oid |       t_data
----+--------+----------+--------+--------+--------+----------+--------
+-------------+------------+--------+--------+-------+----------------
---
  1 |   8160 |        1 |     32 |    893 |      0 |        0 | (0,1)
|           2 |       2048 |     24 |        |       |
\x0100000001000000
(1 行记录)
```

上述输出的内容，t_data存储的是元组的数据，使用16进制表示，可以看到表达的数据内容是(id=1, data=1)。

其余的内容全是元组的“头部信息”，也成为元组的“元信息”，由于内容很多，作用各不相同，此次lab仅关注 "t_xmin""t_xmax"两个与并发控制紧密相关的数据项。

值得注意的是，t_infomask t_infomask2等数据项也参与了事务处理过程，但是原理与实现较为复杂，此次试验不再讲解。感兴趣的同学可以阅读“参考资料”中5、6两本书。

由输出的t_xmin t_max可知，trans表中 id=1 的元组由事务号893的事务创建（t_xmin = 893），且该元组版本是最新的版本，并没有对id=1的元组后续的更新（t_xmax）。

将该事务提交：

```
testdb=*# end;
COMMIT
```

重新启动新的事务，新的事务的事务号为894：

```
testdb=# begin;
BEGIN
testdb=*# select * from txid_current();
 txid_current
--------------
          894
(1 行记录)
```

对trans数据表id=1的记录进行更新操作，并查看数据页内容：

```
testdb=*# update trans set data = 2 where id = 1;
UPDATE 1

testdb=*# select t_xmin, t_xmax, t_data  from
heap_page_items(get_raw_page('trans',0));
 t_xmin | t_xmax |       t_data
--------+--------+--------------------
    893 |    894 | \x0100000001000000
    894 |      0 | \x0100000002000000
(2 行记录)
```

通过输出我们可以看到，postgreSQL更新一个已经存在的元组，会创建一个新的版本。

在上述示例中，创建了(id=1, data=2)这个数据版本，且将新版本元数据t_xmin置为更新

事务的事务号894。
同时会将旧的数据版本(id=1, data=1)的元数据"t_xmax"，置为更新事务的事务号894。

下一节中，我们将通过并发的事务之间实际执行的示例，来展示postgreSQL数据库是如何通过事务快照与元组版本的头部信息实现多版本并发控制协议。

在此之前，先清理一下数据：

```
//先将上一个事务提交
testdb=*# END;
COMMIT

//删除数据
testdb=# delete from trans;
DELETE 1

//再看一下trans表0号数据页，发现数据已经标记为删除，但是依然占用着物理空间，我们再继续深度清理一下
testdb=# select t_xmin, t_xmax, t_data  from
heap_page_items(get_raw_page('trans',0));
 t_xmin | t_xmax |       t_data
--------+--------+--------------------
    893 |    894 | \x0100000001000000
    894 |    895 | \x0100000002000000

//对trans表执行VACUUM，回收物理空间，执行时尽量没有事务访问trans表所在的数据库
testdb=# vacuum full trans;
VACUUM


//再查看一下，发现数据被彻底清除
testdb=# select t_xmin, t_xmax, t_data  from
heap_page_items(get_raw_page('trans',0));
错误:  block number 0 is out of range for relation "trans"
```

注意：VACUUM进程会物理回收掉数据库中确定不再需要的数据版本（对系统中运行的活跃事务以及未来事务都不可见），在postgreSQL内部会自动调动VACUUM及时回收物理空间。
由于VACUUM进程运行时会损耗系统性能，手动调用VACUUM FULL会消耗很多资源，本实验由于数据量小且不涉及生产环境，因此不会有影响；但在真实的生产环境中，需要谨慎手动调用VACUUM FULL。

五、PostgreSQL并发控制示例

PostgreSQL一共支持三种隔离级别：**读已提交（read committed）、可重复读（repeatable read）、可串行化（serializable）**。其中，读已提交是PostgreSQL默认使用的隔离级别。

接下来将在不同的隔离级别下，通过并发事务在执行过程中展示事务的快照与数据版本的元数据变化情况，讲解postgreSQL并发控制的基本原理。

注意：下述实验不考虑事务未成功执行导致回滚的情况，事务均以正确提交（COMMIT）结束事务

PostgreSQL读已提交

首先，关闭之前的终端，重新开启两个sql shell，相当于建立起两个连接postgreSQL的会话。

在两个终端中均输入下述命令，设置事务的隔离级别为读已提交：

```
testdb=# set session characteristics as transaction isolation level
read committed;
SET
```

首先构造测试用的数据，在其中一个终端中输入下述命令，插入(id=1, data=1)：

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

//结束事务（提交）
testdb=# END;
COMMIT
```

根据第二节事务隔离级别中读已提交示例，进行下述操作，展示postgreSQL在读已提交隔离级别下如何避免脏读，以及为什么无法避免不可重复读：

为了方便表述，用T1 T2分别表示两个sql shell，按照顺序执行下述SQL：

```
T1:
testdb=# begin;
BEGIN
testdb=*# select * from txid_current();
 txid_current
--------------
          898
```

```
                                                    T2:
                                                    testdb=# begin;
                                                    BEGIN
                                                    testdb=*#
select * from txid_current();
                                                     txid_current
                                                    --------------
                                                              899

                                                    //查看事务899当
前的快照
                                                    testdb=*#
select * from txid_current_snapshot();

txid_current_snapshot

                                                    ---------------
--------
```

```
                                                                898:898:
```

```
T1:
testdb=*# update trans set data = 2 where id = 1;
UPDATE 1
```

```
                                                                T2:
                                                                testdb=*#
```

```
select * from trans where id = 1;
```

```
                                                                 id | data
                                                                ----+------
                                                                  1 |    1
```

```
                                                                testdb=*#
```

```
select t_xmin, t_xmax, t_data  from
heap_page_items(get_raw_page('trans',0));
```

```
                                                                 t_xmin |
t_xmax |       t_data
                                                                --------+------
--+--------------------
                                                                    897 |
898 | \x0100000001000000
                                                                    898 |
0 | \x0100000002000000
```

```
T1:
testdb=*# commit;
COMMIT
```

```
                                                                T2:
                                                                testdb=*#
```

```
commit;
```

```
                                                                COMMIT
```

在上述SQL序列中，T1先对id=1的记录更新，将data更新为2，然后T2读取id=1的记录。 由于T2读取时使用的快照是(xmin:898 xmax:898 active_trans_list: 空)，
表示当T2执行读取操作时，事务号小于898的事务对于其已经结束（提交），这些事务对数据库的改变对T2来说是可见的；而对于事务号大于等于898的事务依然活跃（未提交），
这些事务对数据库的改变对于事务T2来说是不可见的。

postgreSQL事务可见性判断核心原则： **事务只读取相对快照来说已经提交的事务对数据库造成的修改** 。

因此根据上述可见性原则，事务T2在读取id=1的元组时，它可以“看到”事务号为897的事务对数据库的修改，但是看不到事务号为898事务对数据库的修改。
因此(id=1,data=1)的这个元组版本对于事务T2来说并没有被删除，而是“可见的”；与之相反，由于(id=1, data=2)的元组版本的t_xmin=898，表示是被事务号898的事务创建的，
因此该元组版本对于事务T2并不可见。 综上所述，事务T2的读取id=1的操作依然读取的是旧版本的数据（data=1）。

显然，即使T1最后没有提交，而是回滚（rollback）。由于T2根本看不到T1对于数据库的修改，因此不会发生脏读的数据异常。

但是值得注意的是，postgreSQL在读已提交隔离级别下，同一个事务内，每次执行一条SQL语句都会重新获取一次快照（当前系统事务的状态）。
因此在读已提交隔离级别下，postgreSQL并不能阻止不可重复读异常。 见下述例子：

在两个终端中分别输入下述事务执行序列：

```
T1:
testdb=# begin;
BEGIN
testdb=*# select * from txid_current();
 txid_current
--------------
          900

testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 900:900:

testdb=*# select * from trans where id = 1;
 id | data
----+------
  1 |    2
```

```
                                                                T2:

testdb=# begin;
                                                                BEGIN

testdb=*# select * from txid_current();

txid_current
                                                                -------
-------

901


testdb=*# update trans set data = 3 where id = 1;
                                                                UPDATE
1

                                                                //T2事
务提交

testdb=*# commit;
                                                                COMMIT
```

```
T1:
testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 900:902:
```

```
testdb=*# select * from trans where id = 1;
 id | data
----+------
  1 |    3

testdb=*# select t_xmin, t_xmax, t_data  from
heap_page_items(get_raw_page('trans',0));
 t_xmin | t_xmax |       t_data
--------+--------+--------------------
    897 |    898 | \x0100000001000000
    898 |    901 | \x0100000002000000
    901 |      0 | \x0100000003000000

//T1事务提交
testdb=*# end;
COMMIT
```

通过上述示例可以看到，T1首先根据快照(xmin:900 xmax:900 active_trans_list: 空)，判断出(id=1, data=2)版本对当前事务可见
（因为(id=1,data=1)版本的t_xmax=898，事务号898的事务在T1当前的快照中已经提交，因此对于T1来说，(id=1,data=1)版本已经“删除”，不可见）。

随后T2开始执行，T2事务号901，执行更新操作后（将id=1的元组data改为3）提交。

后面T1又再次执行读取id=1元组的操作。由于是读已提交隔离级别，事务T1会重新快照，以获得系统最新的事务状态。
新快照(xmin:900 xmax:902 active_trans_list: 空)，表示事务号小于900的事务已经结束；事务号大于等于902的事务正在活跃或者还未开始；
对于900 901两个中间事务，900是T1自己的事务号，而901由于不在活跃列表中，因此在这个快照中事务号为901的事务已经提交。

因此事务T1第二次执行读取id=1的元组时，使用了与第一次不相同的快照。在新快照中，897 898 901三个事务均已提交，因此(1,1)(1,2)两个版本对于T1均不可见，
T2事务并发创建的(1,3)版本对T1可见。 所以T2第二次读取了(1,3)，发生了不可重复读异常。

PostgreSQL可重复读

postgreSQL解决不可重复读异常的思路较为简单，只需要让事务在开始时获取一次快照，并在整个事务运行过程中均使用该快照即可。

可以执行如下事务操作序列：

首先，在两个终端中均执行下述语句，设置隔离级别为可重复读：

```
testdb=# set session characteristics as transaction isolation level
repeatable read;
SET
```

在两个终端中分别输入下述事务执行序列：

```
T1:
testdb=# begin;
BEGIN
testdb=*# select * from txid_current();
 txid_current
--------------
          902

testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 902:902:

testdb=*# select * from trans where id = 1;
 id | data
----+------
  1 |    3
```

```
T2:

testdb=# begin;

BEGIN

testdb=*# select * from txid_current();

txid_current

-------

903
```

```
testdb=*# update trans set data = 4 where id = 1;

UPDATE

1

//T2事务提交

testdb=*# commit;

COMMIT
```

```
T1:
testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 902:902:

testdb=*# select * from trans where id = 1;
 id | data
----+------
  1 |    3

testdb=*# select t_xmin, t_xmax, t_data  from
heap_page_items(get_raw_page('trans',0));
 t_xmin | t_xmax |       t_data
```

```
--------+--------+--------------------
    897 |    898 | \x0100000001000000
    898 |    901 | \x0100000002000000
    901 |    903 | \x0100000003000000
    903 |      0 | \x0100000004000000
```

```
//T1事务提交
testdb=*# end;
COMMIT
```

上述事务操作序列与前面的操作序列相同，唯一区别是事务的隔离级别由读已提交，变为了可重复读；T2的更新操作将data改为4。
但是发现，T1前后两次读取id=1的元组data却相同（等于3），并未出现不可重复读异常。

原因在于，T1在最开始获取一次快照(xmin:902 xmax:902 active_trans_list: 空)后，整个事务运行期间均使用该快照，因此两次读取均使用相同的快照读取数据。
虽然事务T2在T1的两次读取之间，创建了(id=1, data=4)的数据版本，并提交。 但是T2第二次读取数据的时候，根据快照(xmin:902 xmax:902 active_trans_list: 空)，
事务T2事务号为903，903相对该快照为活跃状态（未提交），因此事务T1读取id=1的元组时，T2对id=1的元组的修改操作对事务T1并不可见。

在可重复读隔离级别下，事务一开始获取一个快照，并在整个事务执行期间均使用同一个快照。这种设计方式也可以避免幻读的数据异常。 可见如下示例。

在两个终端中分别输入下述事务执行序列：

```
T1:
testdb=# begin;
BEGIN
testdb=*# select * from txid_current();
 txid_current
--------------
          904

testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 904:904:

testdb=*# select * from trans where data > 2;
 id | data
----+------
  1 |    4
```

```
                                                                T2:

testdb=# begin;
                                                                BEGIN

testdb=*# select * from txid_current();

txid_current
```

```
-------
-------

905


testdb=*# insert into trans values (2, 5);
                                                                INSERT
0 1

                                                                //T2事
务提交

testdb=*# commit;
                                                                COMMIT

T1:
testdb=*# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 904:904:

testdb=*# select * from trans where data > 2;
 id | data
----+------
  1 |    4

testdb=*# select t_xmin, t_xmax, t_data  from
heap_page_items(get_raw_page('trans',0));
 t_xmin | t_xmax |       t_data
--------+--------+--------------------
    897 |    898 | \x0100000001000000
    898 |    901 | \x0100000002000000
    901 |    903 | \x0100000003000000
    903 |      0 | \x0100000004000000
    905 |      0 | \x0200000005000000

//T1事务提交
testdb=*# end;
COMMIT
```

上述示例中，T1前后进行了两次查询，查询谓词为data > 2。显然，第一次查询符合条件的只有id=1的元组，最新版本data=4对T1可见。
后续T2开始新事务，插入了新的元组 (id=2, data=5)，这个新元组在T1查询的谓词条件内（data > 2），T2事务提交。 随后，T1继续按照data>2谓词条件进行查询。

根据第二节隔离级别定义，在可重复读下无法解决幻象问题，正常来讲T1第二次读data>2的数据，会将(id=2, data=5)也包含在输出结果中。但是pg实际执行结果和第一次执行一样，并不包含新插入的元组。

可以利用事务快照和元组版本的头部信息进行推理一下。在可重复读隔离级别下，T1前后两次查询使用相同的快照，T2插入新的数据，新元组的t_xmin=905。当事务T1进行第二次查询时，
由于在快照(xmin:904 xmax:904 active_trans_list: 空)中，事务号905为“活跃事务”，因

此事务号905的事务对数据库的修改对于该快照来说不可见（未提交）。
因此，元组(2, 5)对事务T1不可见，事务T1第二次查询data>2的元组，结果并不包含(2, 5)。

既然在可重复读隔离级别下，postgreSQL数据库并不会出现幻读的数据异常，那么为什么并不称其设定为可串行化隔离级别?

原因在于，**postgreSQL在可重复读隔离级别下，会出现写偏序数据异常。**

PostgreSQL写偏序异常

起初ANSI SQL92标准定义了三种数据异常，分别为脏读、不可重复读与幻读。但在1995年，有几位数据库领域的专家经过研究发现，在多版本并发控制协议中，会出现"写偏序"异常。

接下来通过示例表明"写偏序"数据异常现象：

在两个终端中依次输入下述事务执行序列（依然是可重复读隔离级别，不用更改）：

```
T1:
testdb=# begin;
BEGIN

testdb=*# select count(*) as count from trans where data >= 4;
 count
-------
     2
```

```
T2:
testdb=# begin;
BEGIN

testdb=*#
select count(*) as count from trans where data >= 4;
 count
-------
     2
```

```
//应用层判断，不在sql语句内，不要在sql shell中执行
if count == 2

//判断成立，将下述语句输入到sql shell中
T1:
testdb=*# update trans set data = 3 where id = 1;
UPDATE 1

testdb=*# end;
COMMIT
```

```
if count == 2

T2:
```

```
                                                    testdb=*#
update trans set data = 3 where id = 2;
                                                    UPDATE 1
                                                    testdb=*# end;
                                                    COMMIT
```

上述示例中，事务T1执行的逻辑是：如果trans表中data>=4的元组数量等于2，那么T1就将id=1的元组data更新为3；
事务T2执行的逻辑是：如果trans表中data>=4的元组数量等于2，那么T2就将id=2的元组更新为3。 if判断并不在sql语句中而在应用逻辑层实现。

由于trans表初始数据状态是：(1,4) (2,5)，那么两个事务开始运行查询时，得到的结果均为2，由于符合条件，后续两个事务分别会继续执行更新操作。

上述事务运行结束后，trans表的数据状态如下：

```
testdb=# select * from trans;
 id | data
----+------
  1 |    3
  2 |    3
```

会发现，经过T1 T2并发执行后，数据从(1,4) (2,5) 转化为了(1,3) (2,3)。

根据可串行化调度的定义：**事务并发执行的结果与事务按照某种串行顺序执行后的数据结果相同** 。 显然按照这个定义，应存在一个串行执行顺序 T1->T2 或者 T2->T1，其中一个运行结果等于(1,3) (2,3)。

但是，我们在大脑中串行执行一下两个事务：
初始数据为(1,4) (2,5)，如果先运行T1，那么结果是data>=4的元组数量等于2，将(1,4)更新为(1,3)，T1提交；后续继续执行T2，data>=4的元组数量等于1，并不执行更新，T1结束。最终的数据状态为(1,3) (2,5)。 反过来，如果先执行T2，后执行T1。那么最终的数据状态是(1,4) (2,3)。

由此可见，上述示例中T1与T2并发执行的结果并不等价于任何一种串行执行的结果，因此不满足可串行化调度。

而上述出现的数据异常现象，称为**写偏序**异常。

PostgreSQL可串行化

上文介绍了PostgreSQL在可重复读隔离级别下，虽然避免了脏读、不可重复读与幻读，但是存在写偏序异常。

下面通过示例展示PostgreSQL在可串行化隔离级别下，是如何处理写偏序异常的。

首先利用下述更新，将数据会恢复到初始状态(1,4) (2,5)：

```
testdb=# update trans set data = 4 where id = 1;
UPDATE 1
testdb=# update trans set data = 5 where id = 2;
UPDATE 1
```

接下来，在两个终端中均输入下述命令，设置事务隔离级别为可串行化：

```
testdb=# set session characteristics as transaction isolation level
serializable;
SET
```

在两个终端中依次输入下述事务执行序列：

```
T1:
testdb=# begin;
BEGIN

testdb=*# select count(*) as count from trans where data >= 4;
 count
-------
     2
```

```
T2:
testdb=# begin;
BEGIN

testdb=*#
select count(*) as count from trans where data >= 4;
 count
-------
     2
```

```
//应用层判断，不在sql语句内，不要在sql shell中执行
if count == 2

//判断成立，将下述语句输入到sql shell中
T1:
testdb=*# update trans set data = 3 where id = 1;
UPDATE 1

//事务正常提交
testdb=*# end;
COMMIT
```

```
if count == 2

T2:
testdb=*#
update trans set data = 3 where id = 2;
错误:  由于多个事务间的读/写依赖而无法串行访问
描述:  Reason code: Canceled on identification as a pivot, during write.
提示:  该事务如果重试，有可能成功.

//事务选择回滚
testdb=!# end;
ROLLBACK
```

可以看到，在可串行化隔离级别下，PostgreSQL会选择将其中一个事务回滚（ROLLBACK），阻止写偏序异常的发生。

虽然多版本并发控制机制存在写偏序数据异常，但并非所有采用了多版本并发控制机制的商业数据库都会阻止写偏序异常的发生，例如，大名鼎鼎的ORACLE就没有提供避免写偏序异常的能力。

六、阅读指南

此次lab不做要求，对并发控制感兴趣的同学提供一个学习路径。

并发控制理论：参考资料1，对基础的并发控制协议有全面的讲解；参考资料7，对数据异常与隔离级别有较深入的讨论 并发控制工程：如果想深入学习postgreSQL并发控制原理与实现，可以先阅读参考资料4；后续可以继续阅读参考资料6与7

七、练习

习题1：

总结PostgreSQL在不同隔离级别下，会出现哪些数据异常现象，能够禁止哪些数据异常

答：

1. **Read Uncommitted（读取未提交）** 这个隔离级别是最低的，事务可以读取到其他事务未提交的数据。尽管 PostgreSQL 实际上会将其转换为 Read Committed，但它的行为在理论上接近于 Read Uncommitted。

   可能出现的异常：脏读（Dirty Read）： 一个事务读取到另一个事务未提交的数据，这些数据如果被回滚，则读取到的数据不再存在。例如，事务 A 修改了数据并提交，事务 B 在 A 提交前读取了该数据。如果事务 A 回滚，事务 B 的数据就变得无效，可能会导致不一致。
   可以禁止的异常：无法禁止脏读（在PostgreSQL中，默认并不允许完全的Read Uncommitted级别）。但通过采用更高隔离级别，脏读能被防止。
2. **Read Committed（读取已提交）** 这个隔离级别是 PostgreSQL 默认的隔离级别，事务只能读取到已提交的事务数据。

   可能出现的异常：不可重复读（Non-repeatable Read）： 在一个事务内读取的数据，在事务期间可能被其他事务修改，导致多次读取同一数据时结果不同。
   可以禁止的异常：可以避免脏读，因为只会读取已提交的事务数据。
3. **Repeatable Read（可重复读）** 在这个隔离级别下，事务在执行过程中会看到同一数据的同一值，不会被其他事务修改。所有读取的数据，在整个事务过程中保持一致。

   可能出现的异常：幻读（Phantom Read）： 虽然事务内读取的数据一致，但当事务执行过程中，其他事务可以插入新的数据行，这些新数据在事务中可能不会出现，导致查询结果不一致。
   可以禁止的异常：可以避免脏读和不可重复读，但是幻读仍然无法完全避免。
4. **Serializable（可串行化）** 这是最高的事务隔离级别，它保证事务的串行执行，即所有事务的结果等同于按某个顺序一个接一个地执行。它强制保证事务之间的完全隔离。

可能出现的异常：在串行化级别下，幻读、不可重复读、脏读等异常都可以被完全禁止。
可以禁止的异常：脏读；不可重复读；幻读。

总结:

隔离级别	脏读	不可重复读	幻读
读未提交	允许	允许	允许
读已提交	禁止	允许	允许
可重复读	禁止	禁止	允许
可串行化	禁止	禁止	禁止

解答：

习题2：

请总结多版本并发控制协议与两阶段封锁协议相比，主要的优势在哪里

答:
1.两阶段封锁协议（2PL）：

- 严格的事务隔离性：2PL 可以保证严格的隔离性，避免了脏读、不可重复读和幻读等异常，特别适用于对事务一致性要求极高的场景。
- 适用于写密集型应用：在某些写密集型场景中，2PL 可以通过锁来确保数据一致性，但这通常会带来性能上的开销。

2.多版本并发控制（MVCC）的优势:

- 提高并发性能: MVCC 的核心优势在于它能显著提高并发性能。在读取操作上，不需要加锁，避免了锁竞争和阻塞。事务可以并行读取数据，只有在写入时才需要考虑事务之间的冲突。
- 无锁读取：事务可以同时读取同一数据的不同版本，而不会被其他事务的锁操作所阻塞。因此，MVCC 特别适合读取密集型的应用场景。
- 减少死锁：由于不会有锁的竞争，MVCC 可以避免传统的死锁问题。
- 更高的可扩展性：MVCC 提供了更高的事务吞吐量，尤其是在高并发环境下，能够更好地处理大量读请求。

3.总结:

特性	MVCC	2PL
并发性能	由于读操作不加锁，能够大幅度提升并发性能。读操作不会阻塞其他事务的执行	读操作需要加锁，可能会导致阻塞，尤其是在高并发的情况下。
锁竞性	几乎不需要加锁，避免了锁竞争问题，因此并发性能较好。	高并发环境下，锁竞争严重，可能导致性能瓶颈。
死锁问题	由于不使用锁，MVCC 避免了死锁的发生	锁机制可能导致死锁，事务需要进行回滚和重试，影响性能。

特性	MVCC	2PL
写操作性能	写操作需要创建新的数据版本，可能导致存储开销和版本管理开销。	写操作会持有锁，可能会导致阻塞，影响并发性能。
可扩展性	在高并发环境下，MVCC 能够更好地处理大量的读请求，提高系统可扩展性。	高并发场景下，2PL 会因为锁的竞争导致系统扩展性差。
事务隔离性	提供较高的事务隔离性（如快照隔离），避免脏读、不可重复读等问题，但可能会遇到幻读。	可以严格控制事务的隔离性，避免脏读、不可重复读、幻读等问题。

解答：

参考资料

1. CMU 15-445 Lecture#15-18: https://15445.courses.cs.cmu.edu/fall2021/schedule.html
2. PopstgreSQL transaction isolation: https://www.postgresql.org/docs/current/transaction-iso.html
3. PostgreSQL pageinspect操作文档: https://www.postgresql.org/docs/15/pageinspect.html
4. The Internals of PostgreSQL - Concurrency Control: https://www.interdb.jp/pg/pgsql05.html
5. PostgreSQL技术内幕：事务处理深度探索，张树杰
6. 数据库事务处理的艺术：事务管理与并发控制，李海翔等
7. A Critique of ANSI SQL Isolation Levels, sigmod, 1995