DTCC

2016中国数据库技术大会

DATABASE TECHNOLOGY CONFERENCE CHINA 2016

数 据 定 义 未 来

SequeMedia 盛拓传媒　ITI68.com　ChinaUnix　ITPUB

数据库事务处理原理与实例剖析

叶 涛

yetao1@huawei.com

2016/5/13

前言

- RDBMS = 查询(SQL) + 事务(ACID)
  - 查询(SQL): 查询优化 + 查询执行
  - **事务(ACID)：并发控制 + 故障恢复**
- 事务处理涉及许多基础的原理和算法，还包括许多实现细节。
- **主题目标:** 用20-30分钟时间，勾勒出事务处理的主线，主要原理和关键技术
- ACID: what, why, how, example, future

提纲

- **事务处理原理**
  - 事务概念
  - 并发控制
  - 故障恢复
- **PostgreSQL事务处理实现**
  - MVCC和锁
  - 日志和恢复
- **总结和展望**

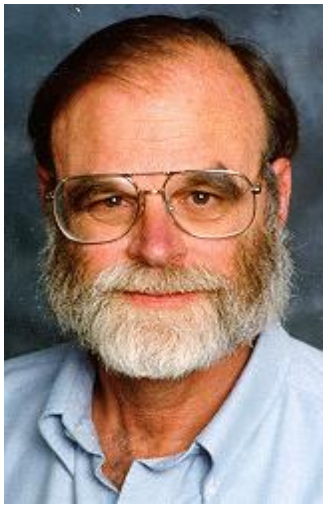
(一) 事务处理原理

什么是事务？

- **Definition**: *a transaction is a group of SQL commands whose results will be made visible to the rest of the system as a unit when the transaction commits --- or not at all, if the transaction aborts. Transactions are expected to be atomic, consistent, isolated, and durable.*
- 例子: 从A账户转账50元到B账户

begin
1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

commit



Jim Gray
1998年图灵奖

什么是ACID？ 为什么要ACID?

- 什么是ACID? (from wiki)
  - **Atomicity:** A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.
  - **Consistency:** A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.
  - **Isolation:** Even though transactions execute concurrently, it appears to each transaction T, that others executed either before T or after T, but not both.
  - **Durability:** Once a transaction completes successfully (commits), its changes to the state survive failures.
- 为什么提出ACID这种事务模型？
  - 给应用开发人员抽象出一个好用的计算机模型：串行执行；执行中间不出错。
  - 屏蔽了事务并发执行的串扰，各种各样的故障，等带来的繁杂的处理逻辑。
  - 事务模型是整个商业世界稳定有序运作的基石。
- P.S. 如果把ACID作为我们的工作方法会怎样？ :)
  - 每做一件事都：(A)要么不做，要么做到底；(C) 一直沿着一个方向做；(I) 心无旁骛；(D)必有总结和输出。 → 绩效很好！

怎样实现ACID?

- **实现ACID的核心技术是并发控制和日志技术**
  - 并发控制：MVCC, 2PL, OCC； 保证并发操作的正确性
  - 日志：Undo/Redo, WAL协议； 保证故障场景下可恢复
- **Atomic: 并发控制 + 日志**
  - 查询和事务执行的中间状态别人看不到。
  - 事务Abort，可以回退，消除影响。
- **Consistent: 应用层定义的完整性约束**
  - Integrity constraints，约束系统从一个合法状态转换到新的合法状态。
- **Isolated: 并发控制**
  - 调度并发的事务对数据库的操作，消除之间的干扰造成的异常结果。
- **Durable: 日志技术**
  - 保证系统Crash，可以通过日志来恢复数据

事务并发执行可能有什么问题？

Dirty Reads

T1	T2
WRITE(A)	
	READ(A)
ABORT	

Last Update

T1	T2
READ(A)	
	READ(A);
A := A+5	
	A := A*1.3
WRITE(A)	
	WRITE(A)

Inconsistent Read

T1	T2
A := 20;	
B := 20;	
WRITE(A)	
	READ(A);
	READ(B);
WRITE(B)	

如果事务的并发执行不加控制，相互之间可能产生各种各样的串扰，影响系统的一致性

理论上，什么是事务正确的并发执行？

- Serial execution: 什么样的事务执行方式肯定是正确的？ **不并发！**所有事务按照某个序列一个个**串行执行**。
- Serializable: 对于N个事务的**一个并发执行**，说它是**可串行化的**(Serializable)，如果它**从计算上等价**于这N个事务的**某种串行执行**；即该并发执行产生的输出以及对数据库的影响和某个串行执行相同。
- 并发控制技术的核心是使用某种算法调度N个事务的执行，使得该执行是可串行化的。**这种算法的正确性都是由严格的数学定理保障的。**

Theorem 1 [PAPA77, PAPA79, STEA76]

Let $\mathbf{T} = \{T_1, \ldots, T_m\}$ *be a set of transactions and let E be an execution of these transactions modeled by logs* $\{L_1, \ldots, L_m\}$. *E is serializable if there exists a total ordering of T such that for each pair of conflicting operations* O_i *and* O_j *from distinct transactions* T_i *and* T_j *(respectively),* O_i *precedes* O_j *in any log* $L_1, \ldots, L_m$ *if and only if* T_i *precedes* T_j *in the total ordering.*

Theorem 2 [BERN80a]

Let $\rightarrow_{rwr}$ *and* $\rightarrow_{ww}$ *be associated with execution E. E is serializable if (a)* $\rightarrow_{rwr}$ *and* $\rightarrow_{ww}$ *are acyclic, and (b) there is a total ordering of the transactions consistent with all* $\rightarrow_{rwr}$ *and all* $\rightarrow_{ww}$ *relationships.*

技术上，怎样实现正确的并发执行？

- 数据库的并发控制技术主要有三种：2PL, MVCC, OCC， 都能实现**正确的(可串行化的)**并发执行, 各适合不同的场景 。
- **Strict two-phase Locking (2PL)**: 事务在读(写)每个数据**对象**前需申请持有共享锁(互斥锁)。 所持有的锁直到事务结束时才释放。事务申请不到锁资源时，需排队等候。
- **Multi-Version Concurrency Control (MVCC)**: 事务不持有锁，但是保证在执行过程中只看到整个数据库过去某个时刻的一致性状态。即使在执行过程中，别的并发事务更改了数据。
- **Optimistic Concurrency Control (OCC)**: 多个事务并发执行的时候，互相不阻塞。 执行过程中，记录每个事务读写的历史。在事务提交前，检查事务的读写历史是否会造成不可串行化调度。如果是，挑选某个冲突的事务回滚。
- 2PL方法是最经典的并发控制方法。 MVCC对于TP(短事务)和AP(长查询)混合负载通常有更好的并发度。OCC方法适合于读写冲突较少的场景；否则，大量的事务回滚会造成性能下降。

应用上，怎样权衡并发性能和正确性?

- 严格的可串行化的并发执行， 能保证正确性，但是同时也会造成事务的阻塞和回滚，影响系统的性能(事务吞吐量)。
- 为了**权衡事务的并发执行性能和正确性**，SQL标准定义了四种不同的隔离级别，对并发执行的约束依次从弱到强。
- 在弱隔离级别下，事务的并发度更好，但是也更容易出现并发造成的异常结果(Anomaly)。

	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
串行化	不可能	不可能	不可能

SQL标准的四种隔离级别怎么实现(定义)？

- SQL标准的四种隔离级别，在2PL方法中，可根据是否持有数据对象(元组和谓词)的读锁和写锁，以及持有锁时间的长短来实现(定义)。
- MVCC以及OCC的并发控制方法的许多弱隔离级别行为，并不能很好的用SQL标准的四种隔离级别描述。

Table 2. Degrees of Consistency and Locking Isolation Levels defined in terms of locks.

Consistency Level = Locking Isolation Level	Read Locks on Data Items and Predicates (the same unless noted)	Write Locks on Data Items and Predicates (always the same)
Degree 0	none required	Well-formed Writes
Degree 1 = Locking READ UNCOMMITTED	none required	Well-formed Writes Long duration Write locks
Degree 2 = Locking READ COMMITTED	Well-formed Reads Short duration Read locks (both)	Well-formed Writes, Long duration Write locks
Cursor Stability (see Section 4.1)	Well-formed Reads Read locks held on current of cursor Short duration Read Predicate locks	Well-formed Writes, Long duration Write locks
Locking REPEATABLE READ	Well-formed Reads Long duration data-item Read locks Short duration Read Predicate locks	Well-formed Writes, Long duration Write locks
Degree 3 = Locking SERIALIZABLE	Well-formed Reads Long duration Read locks (both)	Well-formed Writes, Long duration Write locks

故障对事务的ACID带来什么挑战？

- 事务处理系统中可能会发生什么样的故障？
  - **Transaction Failure：** 事务执行时，因为某种原因(譬如说违反完整性约束)，**导致不能提交**。此时，需要Abort事务， 并清除事务对数据库的修改，保证事务的**原子性**。
  - **System Failure：** 事务处理系统出现故障(譬如说断电)，**内存中的数据全部丢失**。此时需要从故障中恢复系统，保证故障前已提交的事务对数据库的修改不丢失，保证故障时正在运行和已Abort的事务对数据库的修改可被清除，保证事务的**持久性和原子性**。
  - **Media Failure：** 事务处理系统出现介质故障，**硬盘中的数据丢失**。此时需要从存档的日志中重新恢复数据，保证事务的**持久性**。

缓冲区管理策略对故障恢复有什么影响？

- 在事务执行过程中，是否允许刷脏数据到磁盘？（STEAL/NO-STEAL）
  - 若选STEAL, 则要保证最后Abort的事务修改的磁盘数据可被清除。
- 在提交事务时，是否强制要求事务更改的所有数据都必须刷盘？(FORCE/NO-FORCE)
  - 若选NO-FORCE, 则要保证提交的事务对数据库的修改在系统故障(内存掉电)后可恢复。
- (NO-STEAL,FORCE)策略，对缓冲区管理有较大限制，**对事务提交的性能有较大影响。**
- **(STEAL,NO-FORCE)策略，性能更好，但对故障恢复提出更高要求。**
- 一般数据库都选择STEAL, NO-FORCE策略。

技术上怎样实现故障恢复？

- **常见的数据库一般使用遵守Write Ahead Logging(WAL)协议的日志技术，来应对各种故障场景，实现故障恢复。**WAL协议主要包括三条规则：
  - (1) 事务执行过程中，使用日志记录对数据的每个修改，并要保证**日志比数据先刷到磁盘中**。
  - (2) 每条日志记录按照产生先后有一个单调递增的日志号(LSN)。在刷第r条日志时，要保证r之前的所有日志都已被刷到磁盘中。
  - (3) 事务提交时，要保证它产生的日志都已刷到磁盘上。
- **规则(1)保证：**发生故障时，可使用日志，清除abort事务对磁盘上数据的修改，实现了**事务的原子性**。
- **规则(2)和规则(3) 保证：**发生故障时，可使用日志，恢复已提交事务未刷盘的数据，实现了事**务的持久性**。
- 日志和恢复技术中，有许多细节和优化，具体可参见 ARIES论文。

(二) PostgreSQL事务处理实现

PostgreSQL事务处理实现

- **PostgreSQL的并发控制技术**
  - 使用MVCC 机制实现DML语句读写的并发控制。
  - 使用遵循2PL协议的锁机制，实现了DML语句写写的并发控制。
  - 使用表锁实现DML和DDL，在表级别的并发控制。
  - 使用SSI技术实现了真正的可串行化隔离级别。

- **PostgreSQL的故障恢复技术**
  - 使用遵循WAL协议的REDO日志， 实现事务的持久性。
  - PostgreSQL没有UNDO日志，事务的原子性由MVCC和Clog来实现。

PostgreSQL MVCC机制

- PostgreSQL使用MVCC机制实现对**数据读写的并发访问**，实现了“读不阻塞写，写不阻塞读”。
- **Tuple结构：** 每个元组头中，会记录插入事务的ID (xmin)以及删除事务的ID(xmax)。
- **修改：** 执行删除操作时，仅标注元组的删除事务ID，不真正删除数据。执行更新操作时，标记旧元组删除，插入一个新版本的元组，并使用一个指针从旧元组指向新元组。 这样，表中同一行数据的不同版本，根据新旧程度形成一个链表。
- **查询：** 执行查询时，同一行数据的多个版本，使用快照(活跃事务列表)和Clog的信息，来做**可见性判断**，决定对当前查询可见的**唯一**的一个版本。
- **清理：** 历史上被删除的元组如果对任何事务都不可见（不会再被用到），则可以被清理(真正的删除)。 PostgreSQL中使用Vacuum实现无用数据的清理。

MVCC Example: Insert, Update, Delete

```
# insert into customer values('Sally',10000);
INSERT 16472 1
# select current,xmin,xmax,next,oid from customer_page0;
 current | xmin | xmax | next  |  oid
---------+------+------+-------+-------
 (0,1)   | 1390 |    0 | (0,1) | 16472
(1 row)

# update customer set cost=cost+5000 where name='Sally';
UPDATE 1
# select current,xmin,xmax,next,oid from customer_page0;
 current | xmin | xmax | next  |  oid
---------+------+------+-------+-------
 (0,1)   | 1390 | 1391 | (0,2) | 16472
 (0,2)   | 1391 |    0 | (0,2) | 16472
(2 rows)

# delete from customer where name='Sally';
DELETE 1
# select current,xmin,xmax,next,oid from customer_page0;
 current | xmin | xmax | next  |  oid
---------+------+------+-------+-------
 (0,1)   | 1390 | 1391 | (0,2) | 16472
 (0,2)   | 1391 | 1392 | (0,2) | 16472
(2 rows)
```

Insert, xid=1390

Update, xid=1391

Delete, xid=1392

MVCC：Snapshot与可见性判断

PostgreSQL使用活跃事务列表(快照)和Clog来实现多个版本的可见性判断

快照获取时机：

- read committed，每条sql语句执行前；
- repeatable read和serializable， 事务启动时；

元组是可见的，如果：

- xmin 有效， 且
- xmax 无效（活跃，回滚，未开始）

```
# select * from txid_current_snapshot();
 txid_current_snapshot
-----------------------
 1393:1397:1393,1395
(1 row)

最小xid: 最大xid: 活跃xid列表
```

详细版:

```
((Xmin == my-transaction &&                inserted by the current transaction
  Cmin < my-command &&                     before this command, and
  (Xmax is null ||                         the row has not been deleted, or
   (Xmax == my-transaction &&              it was deleted by the current transaction
    Cmax >= my-command)))                  but not before this command,
||                                         or
 (Xmin is committed &&                     the row was inserted by a committed transaction, and
     (Xmax is null ||                      the row has not been deleted, or
      (Xmax == my-transaction &&           the row is being deleted by this transaction
       Cmax >= my-command) ||              but it's not deleted "yet", or
      (Xmax != my-transaction &&           the row was deleted by another transaction
       Xmax is not committed))))           that has not been committed
```

PostgreSQL的锁机制

- PostgreSQL使用表锁实现DML(增删查改)和DDL(表模式更改，创建和删除表和索引) 在表级别并发访问。 使用事务ID锁，元组锁实现元组级别写写的并发访问。
- 表锁和事务ID锁遵循**2PL协议**，事务在执行过程中获取的所有锁在事务提交时统一释放。

编号	锁模式（LOCKMODE）	对应操作	与之冲突的模式
1	AccessShareLock	SELECT	8
2	RowShareLock	SELECT FOR UPDATE	7,8
3	RowExclusiveLock	INSERT, UPDATE, DELETE， ANALYZE	5,6,7,8
4	ShareUpdateExclusiveLock	VACUUM（非FULL）	4,5,6,7,8
5	ShareLock	CREATE INDEX	3,4,6,7,8
6	ShareRowExclusiveLock	—	3,4,5,6,7,8
7	ExclusiveLock	—	2,3,4,5,6,7,8
8	AccessExclusiveLock	DROP TABLE, ALTER TABLE, REINDEX， CLUSTER, VACUUM FULL	全部

Example: PostgreSQL表锁

Session 1:

```
# begin;
BEGIN
# select txid_current();
 txid_current
--------------
         1407
(1 row)
# delete from customer where name='Sam';
DELETE 1
```

Session 2:

```
# begin;
BEGIN
# drop table customer;
Waiting…
```

```
# select * from lockview1;
  pid   | vxid  |   lock_type   |      lock_mode      | granted | xid_lock | relname
--------+-------+---------------+---------------------+---------+----------+----------
  20460 | 2/170 | transactionid | ExclusiveLock       | t       | 1407     |
  20460 | 2/170 | relation      | RowExclusiveLock    | t       |          | customer
 168988 | 4/168 | relation      | AccessExclusiveLock | f       |          | customer
(3 rows)
```

Example: PostgreSQL元组锁

Session 1:

```
# begin;
BEGIN
# update customer set cost=cost+15000
# where name='Sally';
UPDATE 1
```

Session 2:

```
# begin;
BEGIN
# delete from customer where name='Sally';
waiting…
```

```
# select * from lockview1;
  pid   | vxid  |   lock_type   |      lock_mode      | granted | xid_lock | relname
--------+-------+---------------+---------------------+---------+----------+----------
  20460 | 2/178 | transactionid | ExclusiveLock       | t       | 1419     |
  20460 | 2/178 | relation      | RowExclusiveLock    | t       |          | customer
 106373 | 3/572 | transactionid | ExclusiveLock       | t       | 1420     |
 106373 | 3/572 | relation      | RowExclusiveLock    | t       |          | customer
 106373 | 3/572 | tuple         | AccessExclusiveLock | t       |          | customer
 106373 | 3/572 | transactionid | ShareLock           | f       | 1419     |
(6 rows)
```

PostgreSQL四种隔离级别的实现

- PostgreSQL内部实现了三种隔离级别：Read Committed，Snapshot Isolation, Serializable Snapshot Isolation。
- PostgreSQL中Repeatable Read级别, 实际为Snapshot Isolation级别，不会出现幻读现象, 但会出现write skew现象, 违反Serializable。
- Read Committed级别和Snapshot Isolation级别，实现的区别在于取快照的时机: 每个SQL执行前 v.s. 事务启动时。
- Serializable Snapshot Isolation在Snapshot Isolation的基础上，通过监控事务间的读写依赖关系，主动Abort部分事务，实现了最严格的可串行化执行。

Read committed

Snapshot Isolation

Table 13-1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

PostgreSQL Redo日志实现

- In PostgreSQL, Redo log files are used by
  - Recovery
  - Incremental Backup and Point In Time Recovery
  - Replication
- Every change made to the database is written to the redo,Redo logs contain a history of all changes made to the database.
- In PostgreSQL, Redo logs obey the Write Ahead Log protocol
  - ✓ Each data page (either heap or index) is marked with the LSN (log sequence number --- in practice, a WAL file location) of the latest XLOG record affecting the page.
  - ✓ Before the bufmgr can write out a dirty page, it must ensure that xlog has been flushed to disk at least up to the page's LSN.
  - ✓ The redo log buffer is flushed to the redo log file when a COMMIT is issued.

Redo Record

INSERT

```
INSERT INTO score
(team, runs, wickets)
VALUES
('AUS',100,4);
```

Header Info : crc, RM_HEAP_ID, Xid,len, XLOG_HEAP_INSERT

Data : tupleid, infobits,
c0: 'AUS'
c1: 100
c2: 4

UPDATE

```
UPDATE score
SET
   runs = 104,
   wickets = 5
WHERE team = 'AUS';
```

Header Info : crc, RM_HEAP_ID, Xid,len, XLOG_HEAP_UPDATE

Data : oldtupid, newtupid, infobits,
c0: 'AUS'
c1: 104
c2: 4

DELETE

```
DELETE FROM score
WHERE team = 'AUS';
```

Header Info : crc, RM_HEAP_ID, Xid,len, XLOG_HEAP_DELETE

Data : tupleid

Algorithm For WAL Action

- Pin and take Exclusive-lock on buffer containing the data page to be modified
- Start the critical section which ensures that any error occur till End of critical section should be a PANIC as buffers might contain unlogged changes.
- Apply changes to buffer.
- Mark the buffer as dirty which ensures bgwriter (checkpoint) will flush this page and marking buffer dirty before writing log record ensures less contention for content latch of buffer.
- Build a record to be inserted in transaction log buffer.
- Update the Page with LSN which will be used by bgwriter or flush operation of page to ensure that corresponding log is flushed from buffer.
- End Critical section.
- Unlock and unpin the buffer.

Algorithm for Recovery

- Read control file to find the location of the checkpoint
- Get the last valid checkpoint record to find the starting point of recovery
- Read WAL records from log file
- Apply every record after checkpoint except the page lsn is bigger than the lsn of the record until all WAL records in exhausted
- Calculate the lastest xid, oid and lsn
- Start the system

总 结

- 数据库事务的ACID属性保障了企业关键业务负载的正确性和可靠性，是整个商业世界稳定有序运作的基石。
- 数据库内核中实现事务ACID属性的关键技术是并发控制和日志恢复技术。
- 最严格的并发执行是可串行化执行。为了提升性能，SQL 92标准提出四种隔离级别。在放松的隔离级别下，存在脏读，不可重复读，幻读等并发问题。
- 实现并发控制的主要技术有2PL, MVCC和OCC；故障恢复的主要技术是WAL。
- PostgreSQL的并发控制技术：MVCC + 2PL。
- PostgreSQL的故障恢复技术：REDO日志 + WAL协议。

展望1：ACID的不变与变化

- **ACID看起来是一个经典的东西，会落后吗？**
  - ACID本质上解决了什么问题？
  - A？ C？ I？ D?
- **业务需求的变化**
  - 业务量
  - 并发热点
  - 读写比
  - 可用性
  - 业务分布
- **硬件环境的变化**
  - 计算(频率，核数，功耗)：异构众核
  - 存储(时延，带宽，IOPS)： 内存，Flash，非易失性内存
  - 网络(时延，带宽): RDMA，Fabric, 硅光互联

展望2：相关的技术会有什么样的变化？

- 一致性模型的变化？
  - ACID，CAP，BASE..
- 并发控制技术的变化？
  - 大内存，多核，读写比，热点
  - 隔离级别: Snapshot Isolation, SSI
  - 并发控制技术：2PL, MVCC, OCC
- 日志技术的变化？
  - 多核，NVRAM， PAXOS
- 几个新的TP系统： **Hekaton, Spanner, VoltDB**

参考资料

1. Mohan C, Haderle D, Lindsay B, et al. **ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging**. ACM Transactions on Database Systems (TODS), 1992, 17(1): 94-162.
2. Bernstein P A, Goodman N**. Concurrency control in distributed database systems**. ACM Computing Surveys (CSUR), 1981, 13(2): 185-221.
3. Berenson H, Bernstein P, Gray J, et al**. A critique of ANSI SQL isolation levels**//ACM SIGMOD Record. ACM, 1995, 24(2): 1-10.
4. Ports D R K, Grittner K. **Serializable snapshot isolation in PostgreSQL**. Proceedings of the VLDB Endowment, 2012, 5(12): 1850-1861.
5. Momjian B. **MVCC unmasked**. 2013.
6. Momjian B. **Unlocking the Postgres Lock Manager.** 2011
7. Tome Lane. **Transaction Processing in PostgreSQL.** 2000
8. Tome Lane. **PostgreSQL Concurrency Issues.** 2002
9. Amit Kapila. **WAL Internals Of PostgreSQL.** 2012. (pages 28-31 copied)
10. **PostgreSQL 9.5.2 Documentation. 2015.**

谢 谢

- 如果对主题中的内容有任何建议和意见，**希望一起探讨**，琢磨得更细致，欢迎联系我：yetao1@huawei.com or yeetao@gmail.com or 扫我微信。

- 如果对主题中的内容感兴趣，**希望找个工作**，和一伙人一起琢磨和实践数据库内核中的基础技术，也可以联系我，☺

附一：customer_page0视图定义

```
CREATE TABLE CUSTOMER( NAME VARCHAR(20), COST INT ) WITH OIDS;

CREATE VIEW customer_page0 AS
SELECT '(0,' || lp || ')' AS CURRENT,
    CASE lp_flags
        WHEN 0 THEN 'Unused'
        WHEN 1 THEN 'Normal'
        WHEN 2 THEN 'Redirect to ' || lp_off
        WHEN 3 THEN 'Dead' END,
    t_xmin :: text :: int8 AS xmin,
    t_xmax :: text :: int8 AS xmax,
    t_ctid AS NEXT,
    t_oid AS oid
FROM heap_page_items( get_raw_page('customer', 0) )
ORDER BY lp;
```

附二：lockview视图定义

```
CREATE VIEW lockview AS
SELECT pid, virtualtransaction AS vxid, locktype AS lock_type, mode AS lock_mode,
granted,
     CASE
          WHEN virtualxid IS NOT NULL AND transactionid IS NOT NULL
          THEN virtualxid || ' ' || transactionid
          WHEN virtualxid::text IS NOT NULL
          THEN virtualxid
     ELSE transactionid::text
     END AS xid_lock,
     relname, page, tuple, classid, objid, objsubid
FROM pg_locks LEFT OUTER JOIN pg_class
ON (pg_locks.relation = pg_class.oid)
WHERE pid != pg_backend_pid() AND virtualtransaction IS DISTINCT FROM virtualxid
ORDER BY 1, 2, 5 DESC, 6, 3, 4, 7;

CREATE VIEW lockview1 AS
SELECT pid, vxid, lock_type, lock_mode, granted, xid_lock, relname
FROM lockview -- granted is ordered earlier
ORDER BY 1, 2,
```

THANKS

SequeMedia 盛拓传媒 IT168.com ChinaUnix ITPUB