HAWQ TRANSACTION INTRODUCTION

Ming Li （李 明）
Email: mli@pivotal.io
mli@apache.org

Agenda

- PostgreSQL transaction Introduction
- HAWQ Transaction Design
- Performance Tuning
- Future work
- About Us
- Recovery consideration

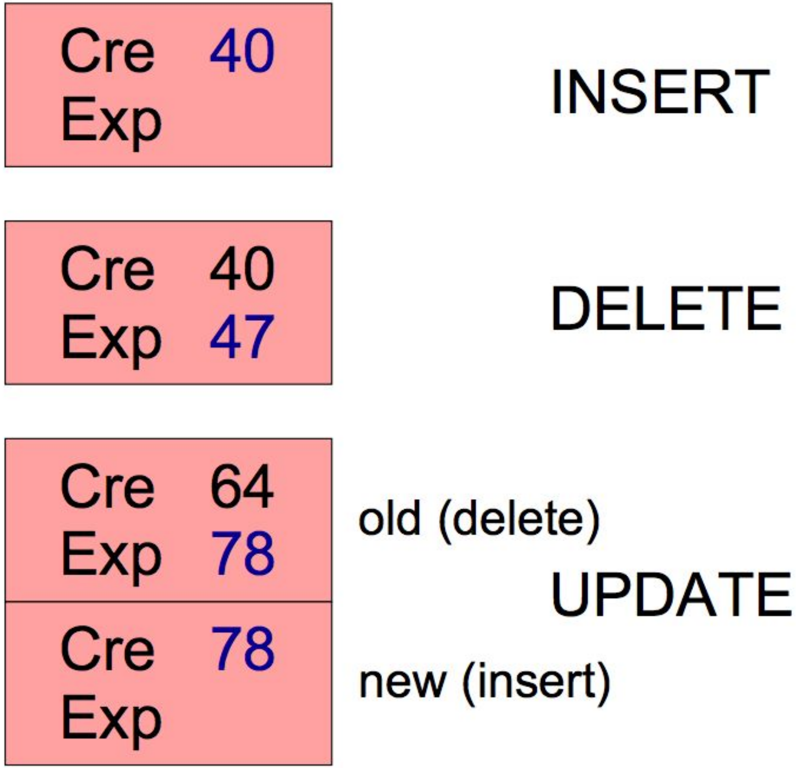
PostgreSQL TRANSACTION INTRODUCTION

MVCC (Multiple Version Concurrency Control)

- Multiple Version representation:
  - Version indication
  - Relationship between tuple and transaction

- The relationship between old version and new version
- Data struct & functions

MVCC Behavior



Cre 40
Exp

INSERT

Cre 40
Exp 47

DELETE

Cre 64
Exp 78

old (delete)

UPDATE

Cre 78
Exp

new (insert)

Snapshot – filter to see the specific version

MVCC snapshots control which tuples are visible for specific statement in a transaction

MVCC Snapshots Determine Row Visibility

Create−Only

Cre 30 Exp	Visible
Cre 50 Exp	Invisible
Cre 110 Exp	Invisible

Create & Expire

Cre 30 Exp 80	Invisible
Cre 30 Exp 75	Visible
Cre 30 Exp 110	Visible

Sequential Scan

Snapshot

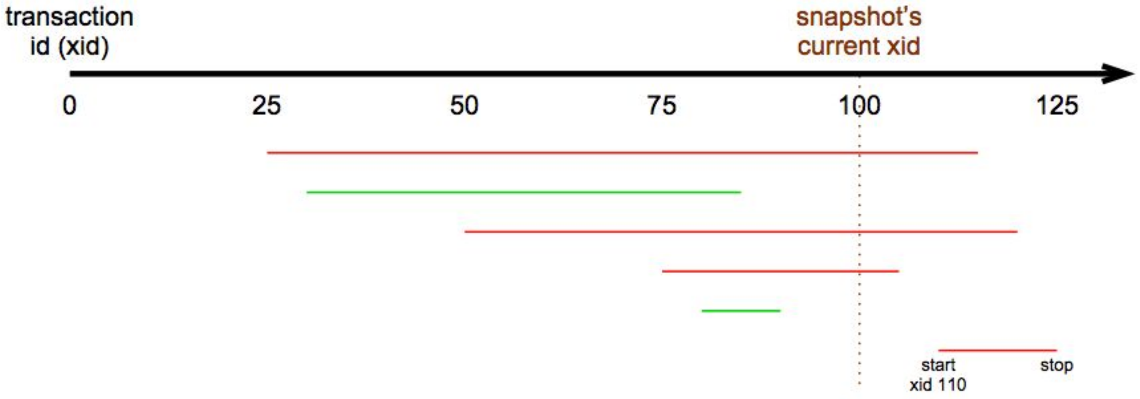
The highest−numbered committed transaction: 100

Open Transactions: 25, 50, 75

For simplicity, assume all other transactions are committed.

Internally, the creation xid is stored in the system column 'xmin', and expire in 'xmax'.

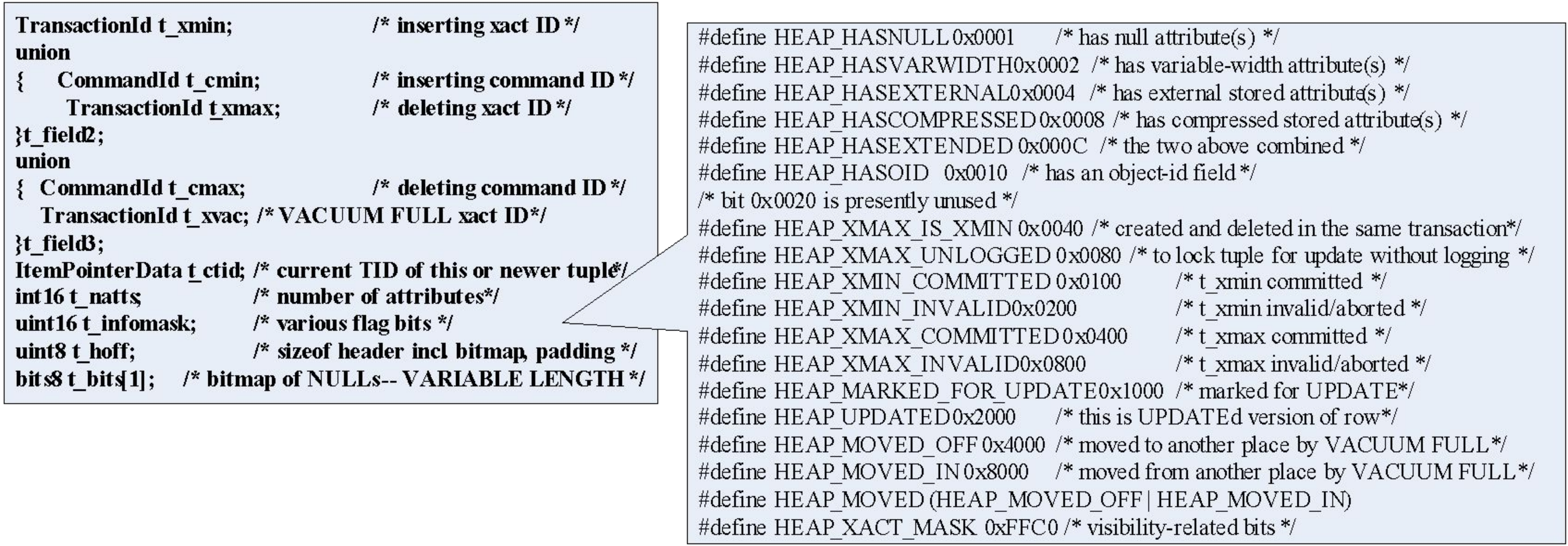
MVCC Snapshot Timeline



Green is visible. Red is invisible.

Only transactions completed before transaction id 100 started are visible.

Data structure: HeapTupleHeaderData

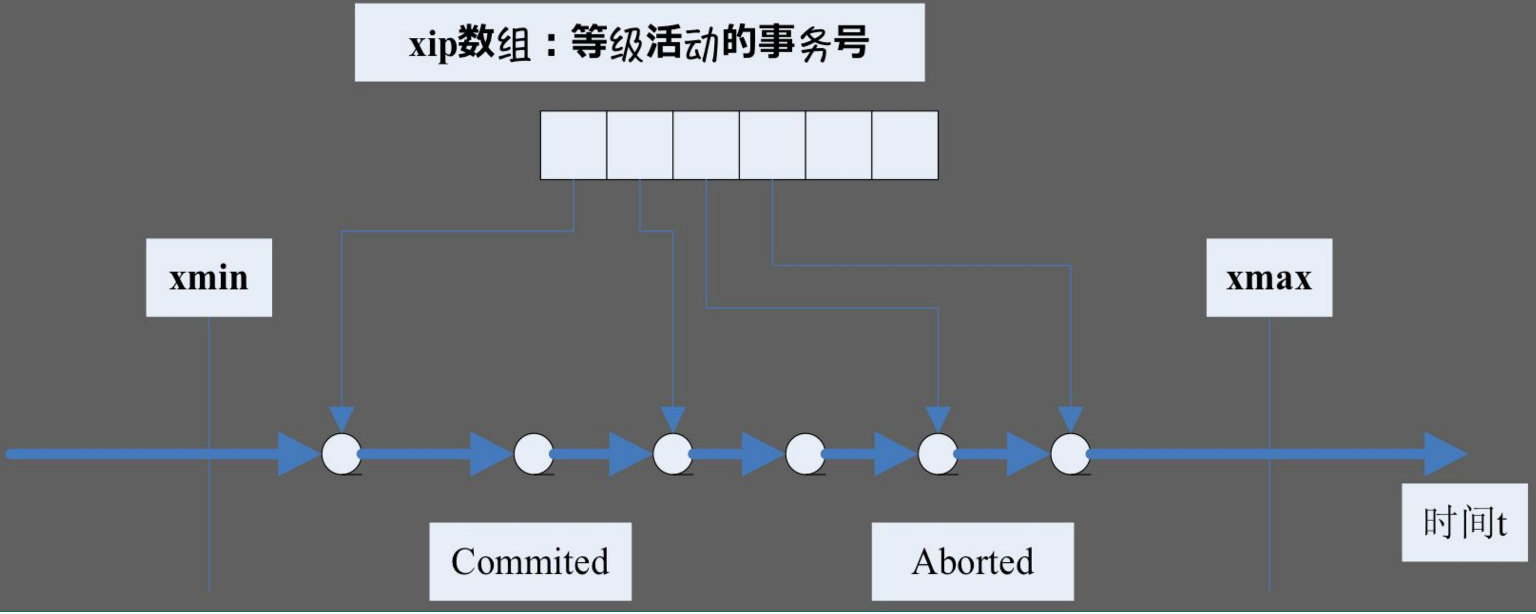


```
TransactionId t_xmin;                /* inserting xact ID */
union
{     CommandId t_cmin;              /* inserting command ID */
        TransactionId t_xmax;        /* deleting xact ID */
}t_field2;
union
{  CommandId t_cmax;                 /* deleting command ID */
   TransactionId t_xvac; /* VACUUM FULL xact ID*/
}t_field3;
ItemPointerData t_ctid; /* current TID of this or newer tuple*/
int16 t_natts;           /* number of attributes*/
uint16 t_infomask;       /* various flag bits */
uint8 t_hoff;            /* sizeof header incl bitmap, padding */
bits8 t_bits[1];    /* bitmap of NULLs-- VARIABLE LENGTH */
```

```
#define HEAP_HASNULL 0x0001      /* has null attribute(s) */
#define HEAP_HASVARWIDTH0x0002  /* has variable-width attribute(s) */
#define HEAP_HASEXTERNAL0x0004  /* has external stored attribute(s) */
#define HEAP_HASCOMPRESSED 0x0008 /* has compressed stored attribute(s) */
#define HEAP_HASEXTENDED 0x000C  /* the two above combined */
#define HEAP_HASOID   0x0010  /* has an object-id field */
/* bit 0x0020 is presently unused */
#define HEAP_XMAX_IS_XMIN 0x0040 /* created and deleted in the same transaction*/
#define HEAP_XMAX_UNLOGGED 0x0080 /* to lock tuple for update without logging */
#define HEAP_XMIN_COMMITTED 0x0100        /* t_xmin committed */
#define HEAP_XMIN_INVALID0x0200           /* t_xmin invalid/aborted */
#define HEAP_XMAX_COMMITTED 0x0400        /* t_xmax committed */
#define HEAP_XMAX_INVALID0x0800           /* t_xmax invalid/aborted */
#define HEAP_MARKED_FOR_UPDATE0x1000 /* marked for UPDATE*/
#define HEAP_UPDATED 0x2000     /* this is UPDATEd version of row*/
#define HEAP_MOVED_OFF 0x4000 /* moved to another place by VACUUM FULL*/
#define HEAP_MOVED_IN 0x8000    /* moved from another place by VACUUM FULL*/
#define HEAP_MOVED (HEAP_MOVED_OFF | HEAP_MOVED_IN)
#define HEAP_XACT_MASK 0xFFC0 /* visibility-related bits */
```

SnapshotData结构体

```
TransactionId xmin;      /* XID < xmin are visible to me  */
TransactionId xmax;   /* XID >= xmax are invisible to me */
uint32 xcnt;   /* # of xact ids in xip[] */
TransactionId *xip;  /* array of xact IDs in progress */
CommandId curcid;       /* in my xact, CID < curcid are visible */
ItemPointerData tid;  /* required for Dirty snapshot -:( */
```



Isolation Level Processing

- Snapshot is fetched:
    - At the start of each SQL statement in READ COMMITTED transaction isolation level
    - At transaction start in SERIALIZABLE transaction isolation level.
- Different processing when concurrently update/delete one tuple:

通过HeapTupleHeader数据结构中的t_infomask== HeapTupleBeingUpdated来判断

等待是通过调用XactLockTableWait()获取等待事务的ShareLock锁，为了支持这种等待，在每个事务的开始时加一个ExclusiveLock事务锁，在事务结束时，同其他类型的锁一起同时释放

重新计算命令搜索条件（WHERE 子句），看看该行已更新的版本是否仍然符合搜索条件。如果是，则第二个更新获取该行的已更新版本重新计算，并生成数据，实现该元组的更新。

HAWQ TRANSACTION DESIGN

HAWQ Characteristics

- SQL On HDFS
- HDFS doesn’t support UPDATE in place
- HAWQ Supported: Append Only (CREATE / INSERT/ TRUNCATE / SELECT/ NO UPDATE/ NO DELETE)
- HAWQ now only support one active master for catalog management
- Dispatch catalog info to segment if needed
- 2PC: no need

HAWQ Transaction Basic

- Combine MVCC with Append Only
- Most scenario: INSERT in batch in one transaction.
- Concurrency control at seg file level instead of tuple level
- Valid File Length maintained in MVCC catalog.
- How INSERT works:
    - HDFS with truncate: truncate invalid data appended by aborted transaction before next time appending data.
    - HDFS without truncate: This segfile cannot be appended any more because of invalid data at the end of file. Use next segfile instead.

Demo

Valid file length of all seg files in one relation will maintained in one sys table:

- For AO tables in the systable: pg_aoseg.pg_aoseg_$TID
- For PARQUET tables in systable: pg_aoseg.pg_paqseg_$TID

INSERT Optimization

- INSERT parallel processing:
  - Lane model: one segfile only be handled by one executor process
  - Each INSERT query processing one groups of seg files
- Speedup transaction ending:
  - Dispatch the segfile creating/deleting tasking to segments for parallel processing
- Speedup Table dropping and selecting:
  - Change hdfs relation segfile path: now for one relation, all segfile put at one directory, instead of put all relations’ segfiles in one directory.

- Pain Point at large number of partitions
- ToDo
  - Only allocate one segfile for one physical segment node, instead of each segfile for one vseg.
  - Multiple supplier one consumer mode write

Performance Tuning

Performance Tuning

- Partition number too big lead to too many seg file (besides partition definition)

1. Hash distributed table: affected by the bucket number (the default value is set by guc default_hash_table_bucket_number) .
2. Randomly Distributed and External Tables: affected by virtual segment number (the guc hawq_rm_nvseg_perquery_perseg_limit manages the number of vsegs per host; the guc hawq_rm_nvseg_perquery_limit set the cluster wide number of vsegs per query).

Best practice: set small value when INSERT and set big value when SELECT.

- Manually Trigger Checkpoint if any transaction related to too many segfiles committed.
- VACUUM FULL catalog if too many out-of-date tuples exists

Future Work

Catalog Bloating (Too many out-of-dated tuples)

- Example: gp_fastsequence up to 2G in customer's env, The related index up to 9G (3.4G even after reindex it), it will slow down 15 minutes for some query (eg. Analyze).
- Why not VACUUM FULL? There are more than 100 queries, 10 Applications running at the same time, which always access gp_fastsequence, so the vacuum operation cannot get lock and proceed.
- HAWQ 2.0 removed gp_fastsequence because it is only used in the index of user data table.

- ToDo:

-- What: Other catalogs (need more investigation)

-- How:

(1) Change MVCC to Update-in-place (like Persistent Tables)
(2) Split into many small catalog (like pg_aoseg.pg_aoseg_$TID)

ToDo: UPDATE Supporting

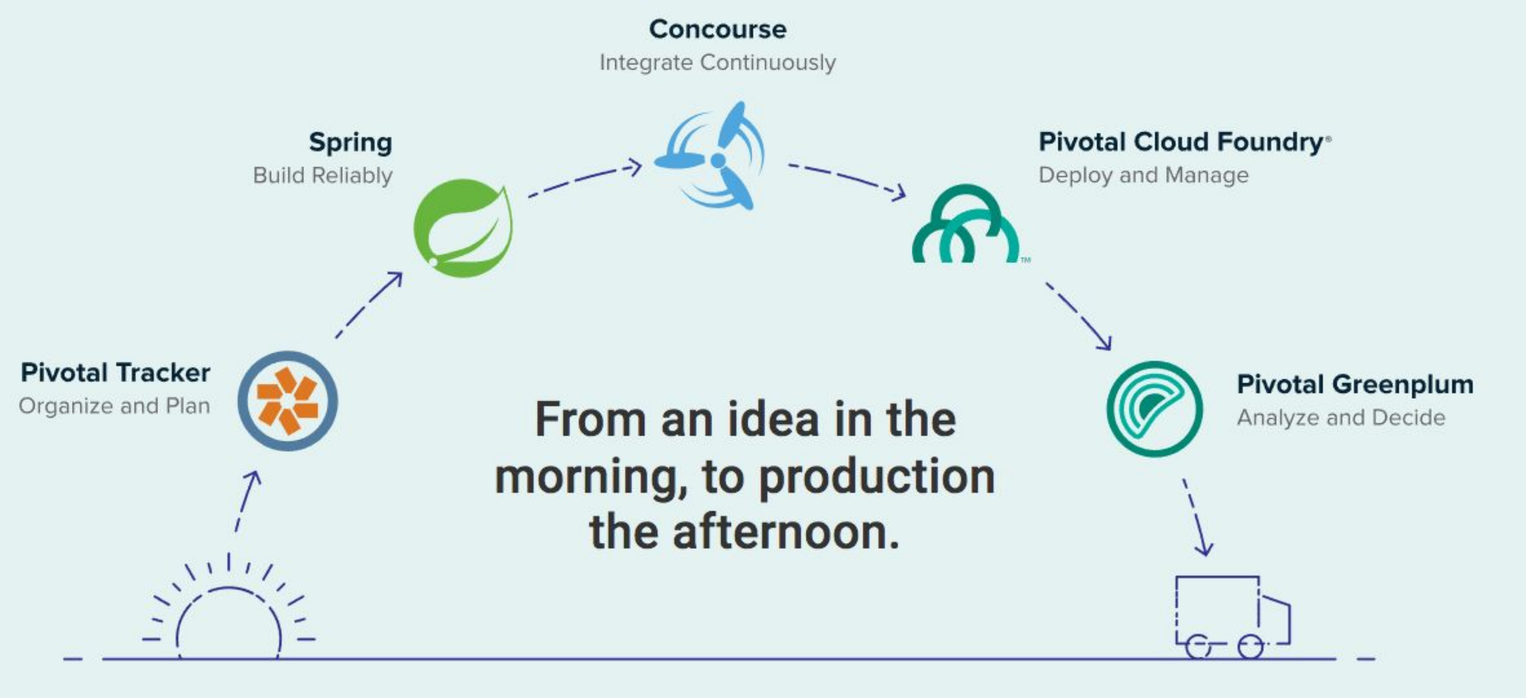
- Update emulation in HAWQ
  - LSMT (Log Structured Merge Tree) vs. Multiple Version
  - Valid File length
  - Relationship between NEW and OLD version tuple (pointer ? Bitmap?)
  - Bloom filter usage for skipping single version tuple
  - Sorted ItemPointer for delta tuples
  - Binary search for each delta block (Random Read)
  - Merge sorting for all delta blocks (Full Scan Read)

About Us

ABOUT ME

- 2001 CS Bachelor of HIT, 2004 CS Master of Tsinghua.
- 2004~2006 PostgreSQL dev @ Institute of Information System and Engineering, School of Software, Tsinghua University.
- 2006~2010 DB2 Federation Server dev @ IBM China Software Development Lab
- 2010~2014 AffinityDB dev @ VMware
- 2014~now HAWQ dev @ Pivotal

About Pivotal



Pivotal Tracker
Organize and Plan

Spring
Build Reliably

Concourse
Integrate Continuously

Pivotal Cloud Foundry®
Deploy and Manage

Pivotal Greenplum
Analyze and Decide

From an idea in the morning, to production the afternoon.

Pivotal is Hiring:
Join us: pivotalrnd_china_jobs@pivotal.io

HAWQ官方纯技术讨论群

Pivotal®

Q & A

Pivotal

RECOVERY CONSIDERATION

PostgreSQL Recovery

- Take an e.g. Transaction T1 inserts lot of data
- System crashes before T1 can commit
- System comes back up again, needs recovery
- REDO (Do the same actions again), UNDO (Revert the effect of an action)
- Postgres supports only REDO. No UNDO is supported.
- In fact, does Postgres need UNDO ?
  - Not Really! WHY ?? (MVCC is the answer)
- What is the advantage and what is the downside ?
  - Adv – Fast recovery (only REDO)
  - Downside – Unnecessary space wastage

Persistent Table Introduction: Derived from GPDB

- *Intent – Clean the INCOMPLETE work*
- Persistent Tables (PT) are *‘Database Object Life Managers’*
- Database objects are either relations, databases, table spaces or filespaces
- Manage life of the same object both on the master & standby side
- PT are heap tables but do NOT follow MVCC rules. They do NOT follow transaction snapshotting rules. Newly added tuple becomes visible instantly.
- Mapping: One ‘Persistent Table’ ⇔ Per database object type

gp_persistent_filespace_node gp_persistent_tablespace_node gp_persistent_database_node gp_persistent_relation_node gp_persistent_relfile_node

Persistent Table: Why Non-MVCC?

- Motivation - PT should act as a on-disk Hash Table that manages information about an object life
- Changes made to the Database (e.g. Relation File Creation) by an incomplete transaction should be still visible, so that they can be revoked
- If PT were MVCC compliant, for e.g. a transaction that created a table, inserted some data and never committed due to system crash would lead to never cleaning the created table
- This consume disk space (resources)

Persistent Table: Struct & Storage

Free Tuple in Persistent Tables –

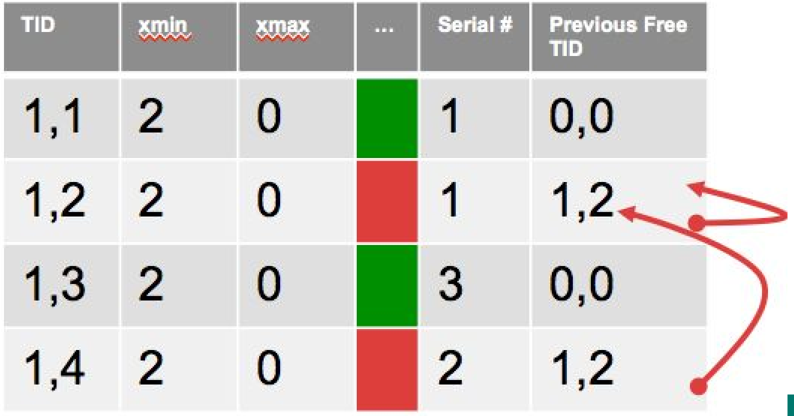
TID	xmin	xmax	…	Serial #	Previous Free TID
1,1	2	0	🟩	1	0,0
1,2	2	0	🟩	2	0,0
1,3	2	0	🟩	3	0,0
1,4	2	0	🟩	4	0,0

Serial #

- Unique in PT, increasing order
- Called as FreeOrder # if tuple is free
- Next Serial #

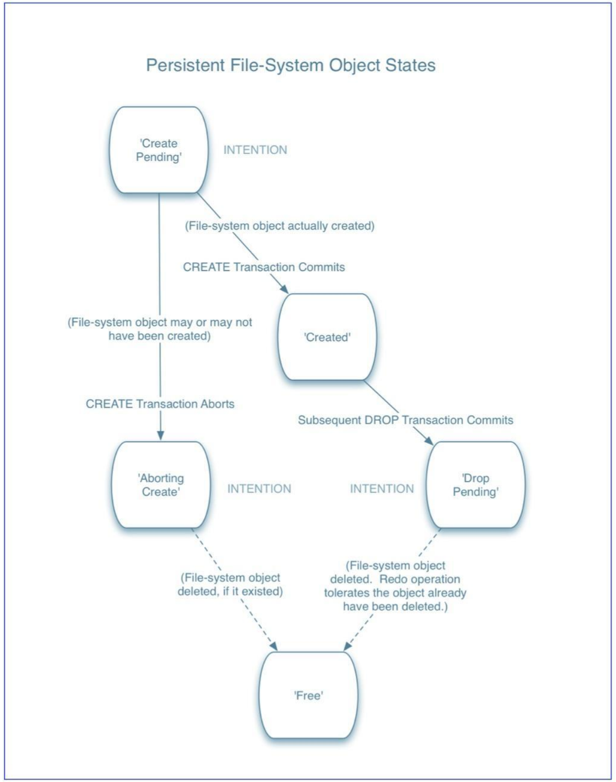
Previous Free TID

- Points to previous free tuple
- 0,0 if the current tuple used
- First free tuple points to itself



TID	xmin	xmax	…	Serial #	Previous Free TID
1,1	2	0	🟩	1	0,0
1,2	2	0	🟥	1	1,2
1,3	2	0	🟩	3	0,0
1,4	2	0	🟥	2	1,2

Persistent Table: Status Transfer Diagram



HAWQ Recovery Stages:

- Pass 1
  - Recover ONLY Persistent Tables & Check their integrity
  - Integrity checks include FreeTID linked list correctness, FreeOrder sequence correctness etc

- Pass 2
  - Using the same redo xlog records as in Pass 1, create infrastructure to find which Database objects need to cleaned and which need to remain etc
  - E.g. Free a create pending state tuple and drop corresponding object

- Pass 3
  - Again, using the same redo xlog records recover only non-PT tables

- Pass 4
  - Some more integrity checks like gp_globalsequence correctness etc

HAWQ Recovery Speedup

- Skip segfile creating/deleting if the transaction is rollback finally.
- ToDo:
  - Persistent Table for relation file creation/deletion tracing is too heavy.
  - Can we just revert relation file creation/deletion operation for rollback transaction directly?