Internals of Postgres

Multi Version Concurrency Control and Transactions

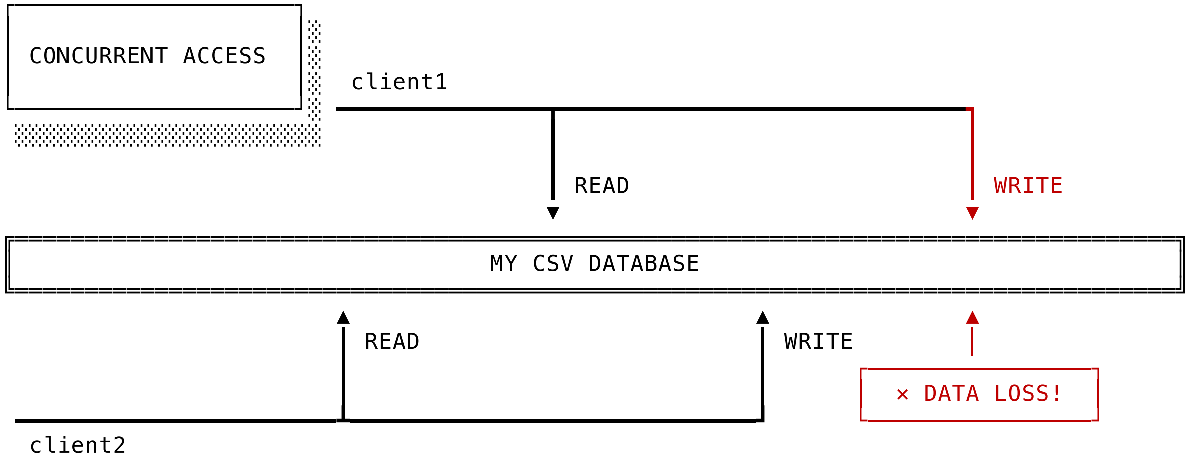
Chun Li

Agenda

- MVCC and how Postgres handles transactions
- How is it implemented?
- Extras in between

In the beginning there was nothing...

- Imagine a simple database reading and writing to a single file
- Life is good until...



What do we do?

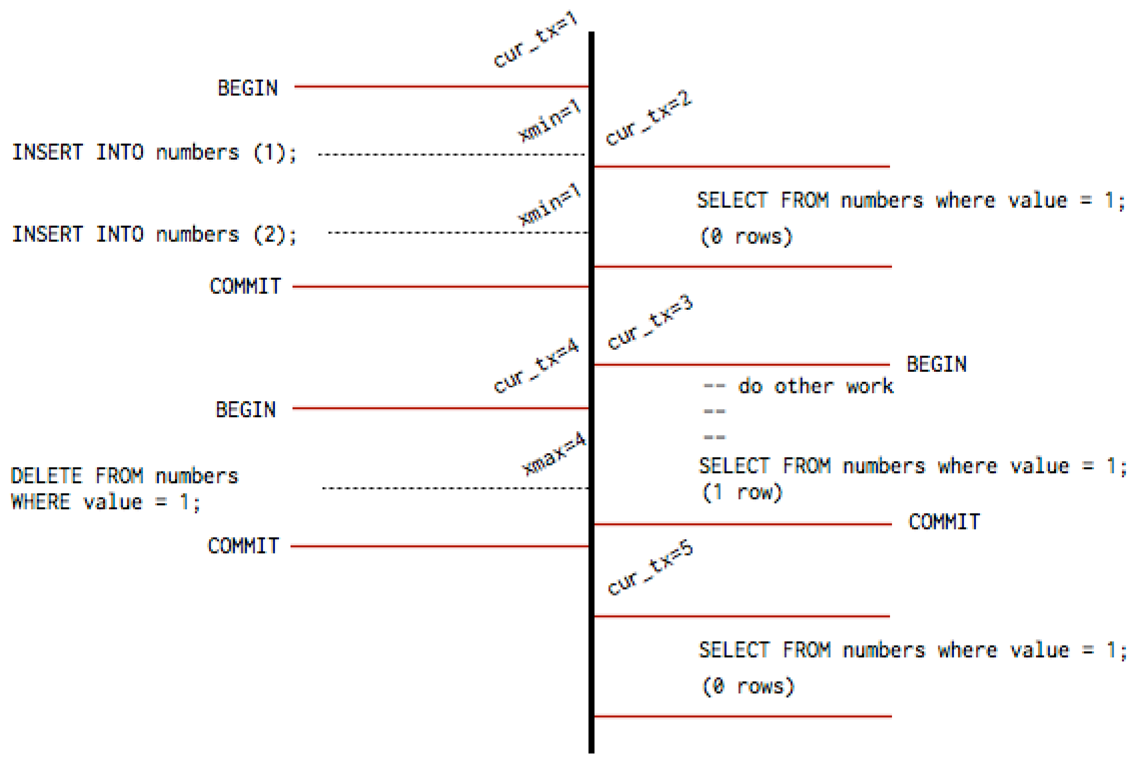
There are a few solutions to this:

- A global read-write lock on every table
- A single control flow point so execution happens one by one

Let there be light

Postgres has an efficient and elegant solution called MVCC:

- Every transaction has a `XID`
- When you update a row, instead of changing the row create a new version
- This allows all transactions to see a consistent `snapshot`
- Rows keep track of `xmin` and `xmax`
    - Commited rows are only visible to transactions with `xmin < XID < xmax`



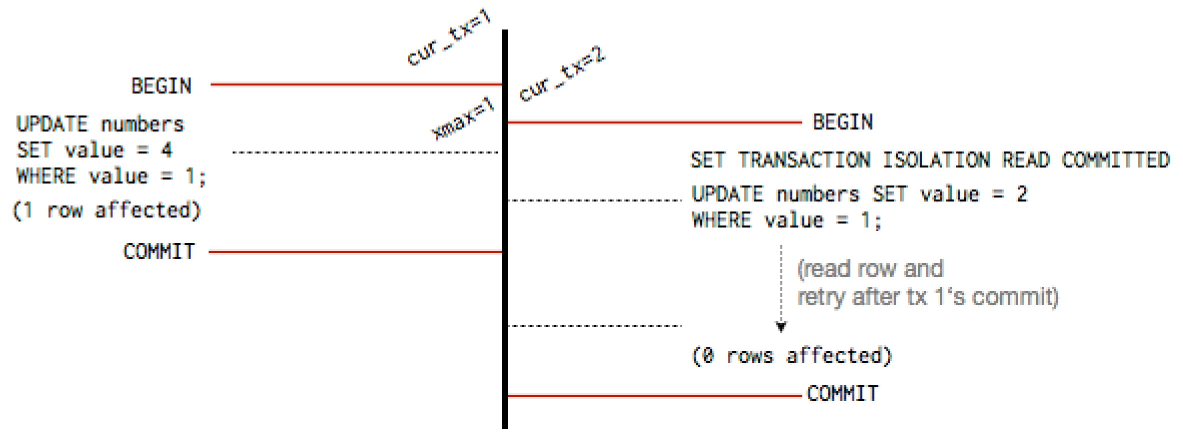
Note you can access xmin and xmax as hidden columns, as well as the current XID easily

```
SELECT *, xmin, xmax FROM numbers;
SELECT txid_current();
```

Two updates at the same time

`READ COMMITTED` - By default, Postgres just reruns if the row has commited

`SERIALIZABLE` - Just fail and let the application handle it



But wait, there's more!

What about the old rows? Eventually they are no longer needed.

- Stop-the-world and "vacuum" them up
- This is not the only approach but it's the one Postgres uses

Extras: Postgres `VACUUM`

Especially given write-heavy loads, you may consider tuning your auto-vacuum settings

- ex. Running `VACUUM` at night during low traffic

There are two main vacuums: standard `VACUUM` and `VACUUM FULL`

- `VACUUM` can usually run in parallel with production operations
  - `SELECT`, `INSERT`, `DELETE`
- `VACUUM FULL` can reclaim more space, but requires an exclusive lock on the table.

The autovacuum daemon only runs `VACUUM` dynamically in response to update activity, to try and maintain a steady usage of disk space

Running VACUUM yourself is usually NOT a great idea (see Sentry later) unless you have a predictable load

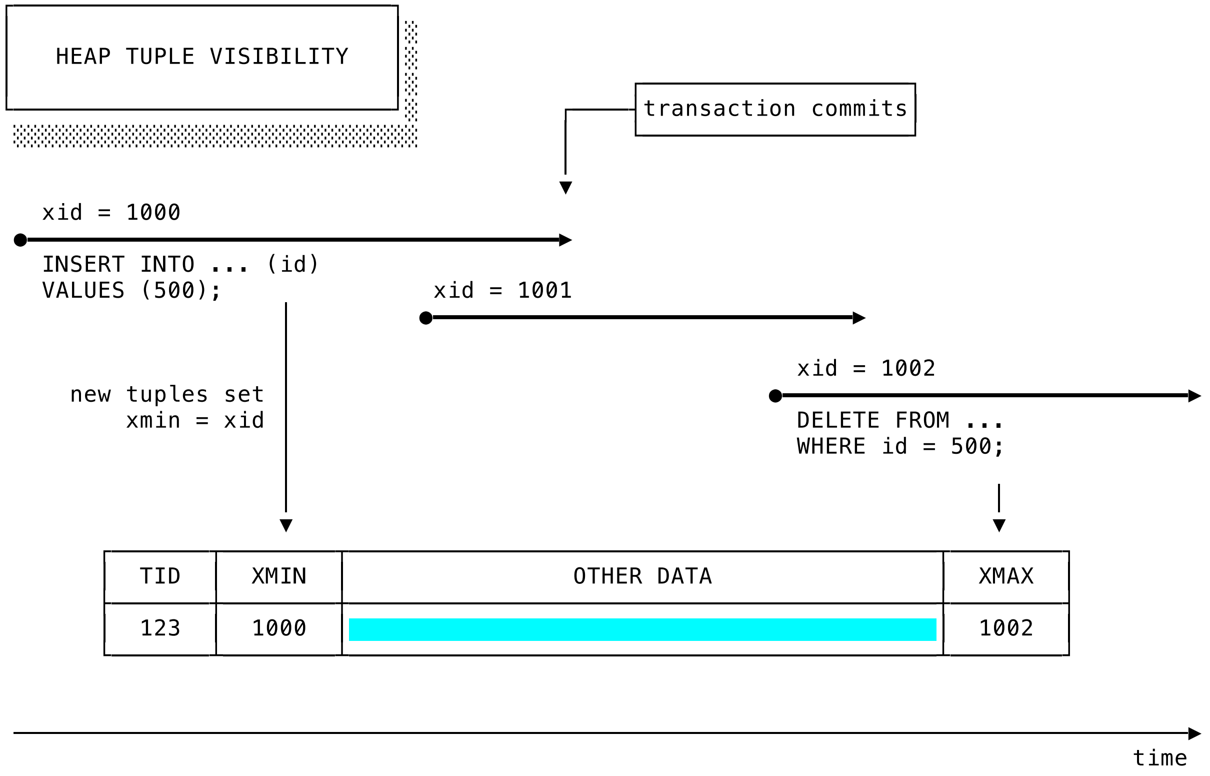
How is it implemented?

```
// proc.h
typedef struct PGXACT
{
        TransactionId xid;
    /* id of top-level transaction currently being
     * executed by this proc, if running and XID
     * is assigned; else InvalidTransactionId */

        TransactionId xmin;
    /* minimal running XID as it was when we were
         * starting our xact, excluding LAZY VACUUM:
         * vacuum must not remove tuples deleted by
         * xid >= xmin ! */

        uint8           vacuumFlags;
    /* vacuum-related flags, see above */
        ...
} PGXACT;
```

How is it implemented?

```
// htup_details.h
/* referenced by HeapTupleHeaderData */
typedef struct HeapTupleFields
{
    TransactionId t_xmin;
    /* inserting xact ID */
    TransactionId t_xmax;
    /* deleting or locking xact ID */
    ...
} HeapTupleFields;
```

How is it implemented?



TID	XMIN	OTHER DATA	XMAX
123	1000		1002

```c
// snapshot.h
typedef struct SnapshotData
{
    /* ...
     * An MVCC snapshot can never see the effects of
     * XIDs >= xmax. It can see the effects of all older
     * XIDs except those listed in the snapshot. xmin is
     * stored as an optimization to avoid needing to
     * search the XID arrays for most tuples.
     */
    TransactionId xmin;
    /* all XID < xmin are visible to me */
    TransactionId xmax;
    /* all XID >= xmax are invisible to me */

    /*
     * For normal MVCC snapshot this contains the all
     * xact IDs that are in progress, unless the snapshot
     * was taken during recovery in which case it's empty
     * ...
     * note: all ids in xip[] satisfy xmin <= xip[i] < xm
     */
    TransactionId *xip;
    uint32        xcnt; /* # of xact ids in xip[] */
    ...
}
```

How is it implemented?

```c
// postgres.c
static void
exec_simple_query(const char *query_string)
{
    ...
    /*
     * Set up a snapshot if parse analysis/planning
     * will need one.
     */
    if (analyze_requires_snapshot(parsetree))
    {
        PushActiveSnapshot(GetTransactionSnapshot());
        snapshot_set = true;
    }
    ...
}
```

How is it implemented?

```
Snapshot
GetSnapshotData(Snapshot snapshot)
{
    /* xmax is always latestCompletedXid + 1 */
    xmax = ShmemVariableCache->latestCompletedXid;
    Assert(TransactionIdIsNormal(xmax));
    TransactionIdAdvance(xmax);

    ...

    snapshot->xmax = xmax;
}
```

How is it implemented?

```
#define InvalidTransactionId        ((TransactionId) 0)
#define BootstrapTransactionId      ((TransactionId) 1)
#define FrozenTransactionId         ((TransactionId) 2)
#define FirstNormalTransactionId    ((TransactionId) 3)
...
/* advance a transaction ID variable,
handling wraparound correctly */
#define TransactionIdAdvance(dest)    \
    do { \
        (dest)++; \
        if ((dest) < FirstNormalTransactionId) \
            (dest) = FirstNormalTransactionId; \
    } while(0)
```

Note this doesn't show how Postgres solves the problem with `XID` wrapping:

- https://www.postgresql.org/docs/9.3/static/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND

```
for (index = 0; index < numProcs; index++)
{
    volatile PGXACT *pgxact = &allPgXact[pgprocno];
    TransactionId xid;
    xid = pgxact->xmin; /* fetch just once */

    /*
     * If the transaction has no XID assigned, we can
     * skip it; it won't have sub-XIDs either.  If
     * the XID is >= xmax, we can also skip it; such
     * transactions will be treated as running anyway
     * (and any sub-XIDs will also be >= xmax).
     */
    if (!TransactionIdIsNormal(xid)
        || !NormalTransactionIdPrecedes(xid, xmax))
        continue;

    if (NormalTransactionIdPrecedes(xid, xmin))
        xmin = xid;

    /* Add XID to snapshot. */
    snapshot->xip[count++] = xid;
    ...
}
...
snapshot->xmin = xmin;
```

How is it implemented?

```
static void
CommitTransaction(void)
{
    ...
    /*
     * We need to mark our XIDs as committed in pg_xact.
     * This is where we durably commit.
     */
    latestXid = RecordTransactionCommit();

    /*
     * Let others know about no transaction in progress
     * by me. Note that this must be done _before_
     * releasing locks we hold and _after_
     * RecordTransactionCommit.
     */
    ProcArrayEndTransaction(MyProc, latestXid);

    ...
}
```

How is it implemented?

```
void
ProcArrayEndTransaction(PGPROC *proc,
TransactionId latestXid)
{
    /*
     * We must lock ProcArrayLock while clearing our
     * advertised XID, so that we do not exit the set
     * of "running" transactions while someone else
     * is taking a snapshot.  See discussion in
     * src/backend/access/transam/README.
     */
    if (LWLockConditionalAcquire(ProcArrayLock, LW_EXCLUS
    {
        ProcArrayEndTransactionInternal(proc, pgxact, lat
        LWLockRelease(ProcArrayLock);
    }
    ...
}
```

How is it implemented?

```
static inline void
ProcArrayEndTransactionInternal
(PGPROC *proc, PGXACT *pgxact, TransactionId latestXid)
{
    ...
    /* Also advance global latestCompletedXid while
       holding the lock */
    if (TransactionIdPrecedes
        (ShmemVariableCache->latestCompletedXid,
                                      latestXid))
        ShmemVariableCache->latestCompletedXid
                = latestXid;
}
```

Proc Array

- Postgres is unique in that it creates a new process for each connection
- Usually it's happy to let them do things in parallel, but transactions commit and abort serially

How do we scan for tuples visible to a snapshot?

```
static void
heapgettup(HeapScanDesc scan, ScanDirection dir,
           int nkeys, ScanKey key) {
    ...
    /*
     * advance the scan until we find a qualifying tuple
     * or run out of stuff to scan
     */
    lpp = PageGetItemId(dp, lineoff);
    for (;;) {
        /*
         * if current tuple qualifies, return it.
         */
        valid =
        HeapTupleSatisfiesVisibility(tuple, snapshot,
                                     scan->rs_cbuf);
        if (valid) {
            return;
        }
        ++lpp;/* move forward in this page's ItemId array
        ++lineoff;
    }
    ...
}
```

Called by `HeapTupleSatisfiesVisibility`

```
bool
HeapTupleSatisfiesMVCC(HeapTuple htup, Snapshot snapshot,
                       Buffer buffer) {
    ...
    else if (XidInMVCCSnapshot(
        HeapTupleHeaderGetRawXmin(tuple), snapshot))
        return false;
    else if (TransactionIdDidCommit(
        HeapTupleHeaderGetRawXmin(tuple)))
        SetHintBits(tuple, buffer, HEAP_XMIN_COMMITTED,
                    HeapTupleHeaderGetRawXmin(tuple));
    ...

    return false;
}
```

```
static bool
XidInMVCCSnapshot(TransactionId xid, Snapshot snapshot)
{
    /* Any xid < xmin is not in-progress */
    if (TransactionIdPrecedes(xid, snapshot->xmin))
        return false;
    /* Any xid >= xmax is in-progress */
    if (TransactionIdFollowsOrEquals(xid, snapshot->xmax)
        return true;

    ...

    for (i = 0; i < snapshot->xcnt; i++)
    {
        if (TransactionIdEquals(xid, snapshot->xip[i]))
            return true;
    }

    ...
}
```

How is it implemented?

The last thing it does it check if the tuple has commited via the commit log

```
bool /* true if given transaction committed */
TransactionIdDidCommit(TransactionId transactionId)
{
    XidStatus xidstatus;

    xidstatus = TransactionLogFetch(transactionId);

    /*
     * If it's marked committed, it's committed.
     */
    if (xidstatus == TRANSACTION_STATUS_COMMITTED)
        return true;
    ...
}
```

The End

- I have really only glossed over the surface of Postgres's internals
- It really makes me appreciate the hard work and complexity of a system that is performant, robust and stable
- `TODO` : How does Postgres commit? (durability, the WAL, and subcommits)

References

https://brandur.org/postgres-atomicity

https://github.com/postgres/postgres

https://devcenter.heroku.com/articles/postgresql-concurrency

https://www.postgresql.org/docs/9.1/static/routine-vacuuming.html