Transaction Processing in PostgreSQL

Tom Lane
Great Bridge, LLC
tgl@sss.pgh.pa.us

Outline

Introduction

- What is a transaction?

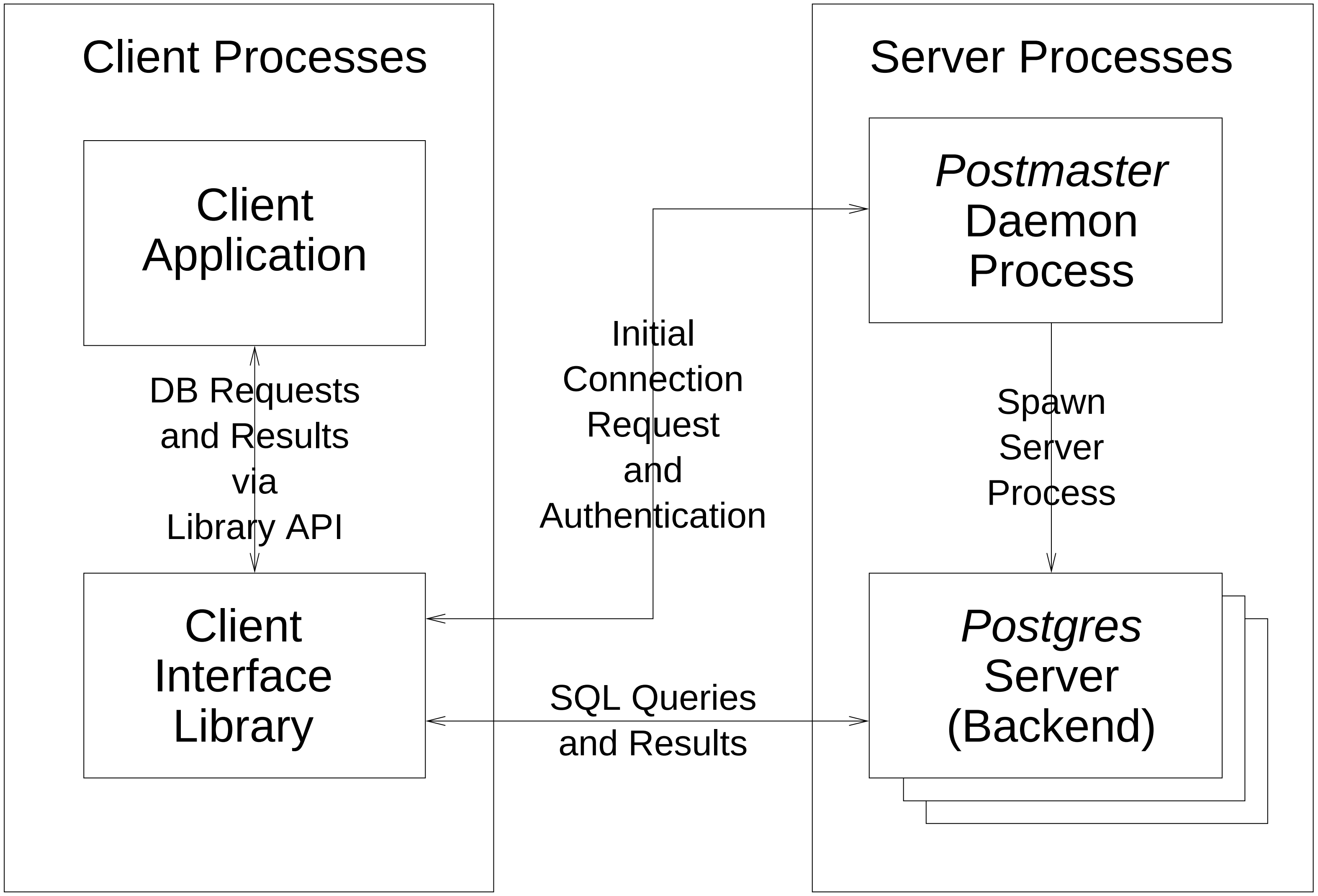
User's view

- Multi-version concurrency control

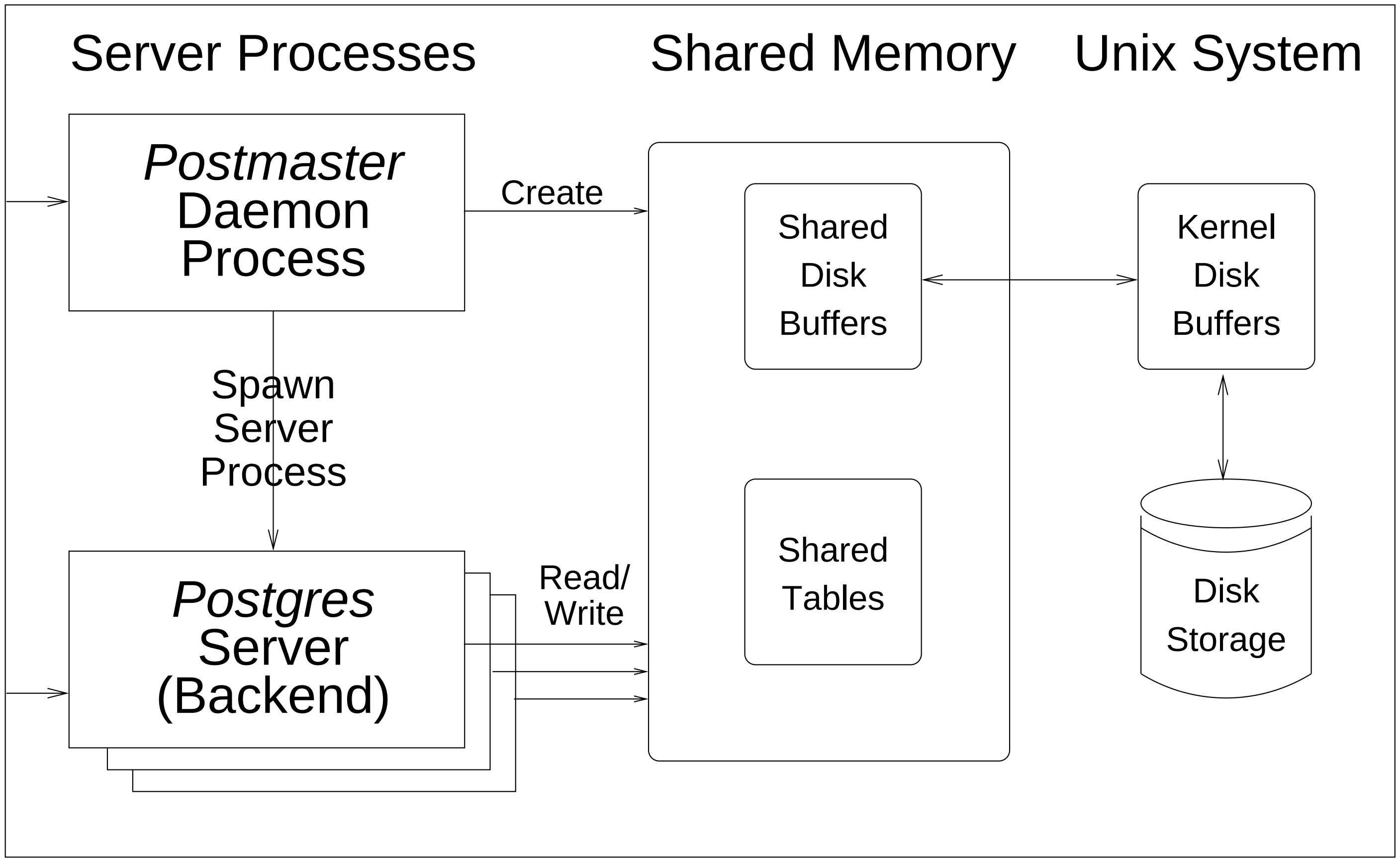
Implementation

- Tuple visibility
- Storage management
- Locks

PostgreSQL system overview


Client Processes
Client Application
DB Requests and Results via Library API
Client Interface Library
Initial Connection Request and Authentication
SQL Queries and Results
Server Processes
Postmaster Daemon Process
Spawn Server Process
Postgres Server (Backend)

PostgreSQL system overview



- Database files are accessed through shared buffer pool
  - Hence, two backends can never see inconsistent views of a file
- Unix kernel usually provides additional buffering

What is a transaction, anyway?

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.*

- Postgres does not support distributed transactions, so all commands of a transaction are executed by one backend.
- We don’t currently handle nested transactions, either.

The ACID test: atomic, consistent, isolated, durable

Atomic: results of a transaction are seen entirely or not at all within other transactions. (A transaction need not appear atomic to itself.)

Consistent: system-defined consistency constraints are enforced on the results of transactions. (Not going to discuss constraint checking today.)

Isolated: transactions are not affected by the behavior of concurrently-running transactions.

Stronger variant: **serializable.** If the final results of a set of concurrent transactions are the same as if we'd run the transactions serially in some order (not necessarily any predetermined order), then we say the behavior is serializable.

Durable: once a transaction commits, its results will not be lost regardless of subsequent failures.

But how can thousands of changes be made "atomically"?

- The actual tuple insertions/deletions/updates are all marked as *done by transaction N* as they are being made. Concurrently running backends ignore the changes because they know transaction N is not committed yet. When the transaction commits, all those changes become logically visible at once.
- The control file **pg_log** contains 2 status bits per transaction ID, with possible states *in progress, committed, aborted.* Setting those two bits to the value *committed* is the atomic action that marks a transaction committed.
- An aborting transaction will normally set its **pg_log** status to *aborted.* But even if the process crashes without having done so, everything is safe. The next time some backend checks the state of that transaction, it will observe that the transaction is marked *in progress* but is not running on any backend, deduce that it crashed, and update the **pg_log** entry to *aborted* on its behalf. No changes are needed in any table data file during abort.

But is it really atomic and durable, even if the system crashes?

Well ... that depends on how much you trust your kernel and hard disk.

• Postgres transactions are only guaranteed atomic if a disk page write is an atomic action. On most modern hard drives that's true if a page is a physical sector, but most people run with disk pages configured as 8K or so, which makes it a little more dubious whether a page write is all-or-nothing.

• **pg_log** is safe anyway since we're only flipping bits in it, and both bits of a transaction's status must be in the same sector.

• But when moving tuples around in a data page, there's a potential for data corruption if a power failure should manage to abort the page write partway through (perhaps only some of the component sectors get written). This is one reason to keep page sizes small ... and to buy a UPS for your server!

Working through the Unix kernel costs us something, too

It’s critical that we force a transaction’s data page changes down to disk before we write **pg_log.** If the disk writes occur in the wrong order, a power failure could leave us with a transaction that’s marked *committed* in **pg_log** but not all of whose data changes are reflected on disk --- thus failing the atomicity test.

• Unix kernels allow us to force the correct write order via fsync(2), but the performance penalty of fsync’ing many files is pretty high.

• We’re looking at ways to avoid needing so many fsync()s, but that’s a different talk.

User's view: multi-version concurrency control

A PostgreSQL application sees the following behavior of concurrent transactions:

• Each transaction sees a snapshot (database version) as of its start time, no matter what other transactions are doing while it runs

• Readers do not block writers, writers do not block readers

• Writers only block each other when updating the same row

Concurrent updates are tricky

Consider this example: transaction A does

```
UPDATE foo SET x = x + 1 WHERE rowid = 42
```

and before it commits, transaction B comes along and wants to do the same thing on the same row.

- **B clearly must wait to see if A commits or not.**
- **If A aborts then B can go ahead, using the pre-existing value of *x.***
- **But if A commits, what then?**
  - Using the old value of *x* will yield a clearly unacceptable result: *x* ends up incremented by 1 not 2 after both transactions commit.
  - But if B is allowed to increment the new value of *x,* then B is reading data committed since it began execution. This violates the basic principle of transaction isolation.

Read committed vs. serializable transaction level

PostgreSQL offers two answers to the concurrent-update problem (out of four transaction isolation levels defined in the ISO SQL standard):

Read committed level: allow B to use new tuple as input values (after checking to ensure new tuple still satisfies query's WHERE clause). Thus, B is allowed to see just this tuple of A's results.

Serializable level: abort B with "not serializable" error. Client application must redo the whole transaction B, which will then be allowed to see the new value of *x* under strict serializable-behavior rules.

• Serializable level is logically cleaner but requires more code in application, so by default we run in read-committed level which usually produces the desired behavior.

• In either case a pure SELECT transaction only sees data committed before it started. It's just updates and deletes that are interesting.

How it's implemented

"O say, can you see that tuple?"

The most fundamental implementation concept is *tuple visibility:* which versions of which table rows are seen by which transactions.

Ignoring tuples you're not supposed to be able to see is the key to making transactions appear atomic.

Definition: a *tuple* is a specific stored object in a table, representing one version of some logical table *row.* A row may exist in multiple versions simultaneously.

Non-overwriting storage management

We must store multiple versions of every row. A tuple can be removed only after it’s been committed as deleted for long enough that no active transaction can see it anymore.

Fortunately, PostgreSQL has always practiced "non overwriting" storage management: updated tuples are appended to the table, and older versions are removed sometime later.

Currently, removal of long-dead tuples is handled by a *VACUUM* maintenance command that must be issued periodically. We are looking at ways to reduce need for *VACUUM* by recycling dead tuples on-the-fly.

Per-tuple status information

Tuple headers contain:

- **xmin:** transaction ID of inserting transaction
- **xmax:** transaction ID of replacing/deleting transaction (initially NULL)
- **forward link:** link to newer version of same logical row, if any

Basic idea: tuple is visible if xmin is valid and xmax is not. "Valid" means "either committed or the current transaction".

If we plan to update rather than delete, we first add new version of row to table, then set xmax and forward link in old tuple. Forward link will be needed by concurrent updaters (but not by readers).

To avoid repeated consultation of **pg_log,** there are also some status bits that indicate "known committed" or "known aborted" for xmin and xmax. These are set by the first backend that inspects xmin or xmax after the referenced transaction commits/aborts.

"Snapshots" filter away active transactions

If Transaction A commits while Transaction B is running, we don't want B to suddenly start seeing A's updates partway through.

- Hence, we make a list at transaction start of which transactions are currently being run by other backends. (Cheap shared-memory communication is essential here: we just look in a shared-memory table, in which each backend records its current transaction number.)
- These transaction IDs will never be considered valid by the current transaction, even if they are shown to be committed in **pg_log** or on-row status bits.
- Nor will a transaction with ID higher than the current transaction's ever be considered valid.
- These rules ensure that no transaction committing after the current transaction's start will be considered committed.
- Validity is in the eye of the beholder.

Table-level locks: still gotta have ’em for some things

Even though readers and writers don’t block each other under MVCC, we still need table-level locking.

This exists mainly to prevent the entire table from being altered or deleted out from under readers or writers.

We also offer various lock levels for application use (mainly for porting applications that take a traditional lock-based approach to concurrency).

Types of locks

	Lock type	Acquired by system for	Conflicts with
1	AccessShareLock	SELECT	7
2	RowShareLock	SELECT FOR UPDATE	6,7
3	RowExclusiveLock	UPDATE, INSERT, DELETE	4,5,6,7
4	ShareLock	CREATE INDEX	3,5,6,7
5	ShareRowExclusiveLock		3,4,5,6,7
6	ExclusiveLock		2,3,4,5,6,7
7	AccessExclusiveLock	DROP TABLE, ALTER TABLE, VACUUM	all

All lock types can be obtained by user LOCK TABLE commands.

Locks are held till end of transaction: you can grab a lock, but you can’t release it except by ending your transaction.

Lock implementation

Locks are recorded in a shared-memory hash table keyed by kind and ID of object being locked. Each item shows the types and numbers of locks held or pending on its object. Would-be lockers who have a conflict with an existing lock must wait.

Waiting is handled by waiting on a per-process IPC semaphore, which will be signaled when another process releases the wanted lock. Note we need only one semaphore per concurrent backend, not one per potentially lockable object.

Deadlock detection

Deadlock is possible if two transactions try to grab conflicting locks in different orders.

If a would-be locker sleeps for more than a second without getting the desired lock, it runs a deadlock-check algorithm that searches the lock hash table for circular lock dependencies. If it finds any, then obtaining the lock will be impossible, so it gives up and reports an error. Else it goes back to sleep and waits till granted the lock (or till client application gives up and requests transaction cancel).

- The delay before running the deadlock check algorithm can be tuned to match the typical transaction time in a particular server’s workload. In this way, unnecessary deadlock checks are seldom performed, but real deadlocks are detected reasonably quickly.

Short-term locks

Short-term locks protect datastructures in shared memory, such as the lock hashtable described above.

These locks should only be held for long enough to examine and/or update a shared item --- in particular a backend should never block while holding one.

Implementation: *spin locks* based on platform-specific atomic test-and-set instructions. This allows the lock code to fall through extremely quickly in the common case where there is no contention for the lock. If the test-and-set fails, we sleep for a short period (using select(2)) and try again. No deadlock detection as such, but we give up and report error if fail too many times.

Summary

PostgreSQL offers true ACID semantics for transactions, given some reasonable assumptions about the behavior of the underlying Unix kernel and hardware.

Multi-version concurrency control **allows concurrent reading and writing of tables, blocking only for concurrent updates of same row.**

MVCC is practical because of non-overwriting storage manager that we inherited from the Berkeley ***POSTQUEL*** **project. Traditional row-overwriting storage management would have a much harder time.**