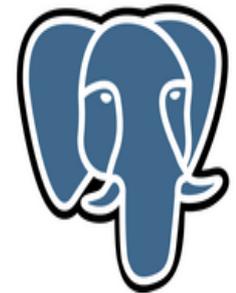




Transactions

Intel (TX memory):
Transactional
Synchronization
Extensions (TSX)

PostgreSQL



Goal – A Distributed Transaction

- We want a transaction that involves multiple nodes
- Review of transactions and their properties
- Things we need to implement transactions
 - * Locks
 - * Achieving atomicity through logging
 - Roll ahead, roll back, write ahead logging
- Finally, 2 Phase Commit (aka 2PC) and 3PC
- Lead into Paxos (again!)

Transactions - Definition

- A transaction is a sequence of data operations with the following properties:
 - * **A** Atomic
 - All or nothing
 - * **C** Consistent
 - Consistent state in => consistent state out
 - * **I** Independent (Isolated)
 - Partial results are not visible to concurrent transactions
 - * **D** Durable
 - Once completed, new state survives crashes

Isolation and serializability

● Definitions

* isolation

- no transaction can see incomplete results of another

* serializability

- actual execution same as some serial order

● Algorithms (based on locks)

* two-phase locking

- serializability

* strict two-phase locking

- isolation and serializability

Two Possible (pessimistic) Approaches

- Two Phase Locking
- Strict Two Phase Locking

Two Phase Locking

- Locks

- * reader/writer locks
- * acquired **as** transaction proceeds
- * no more acquires after first release

- Phase 1

- acquire locks and access data, but release no locks

- Phase 2

- access data, release locks, but acquire no new locks

Q Semantics of two-phase locking

- Does the Two-Phase Locking protocol ensure
 - * serializability?
 - * independence?

- How?

Semantics of two-phase locking

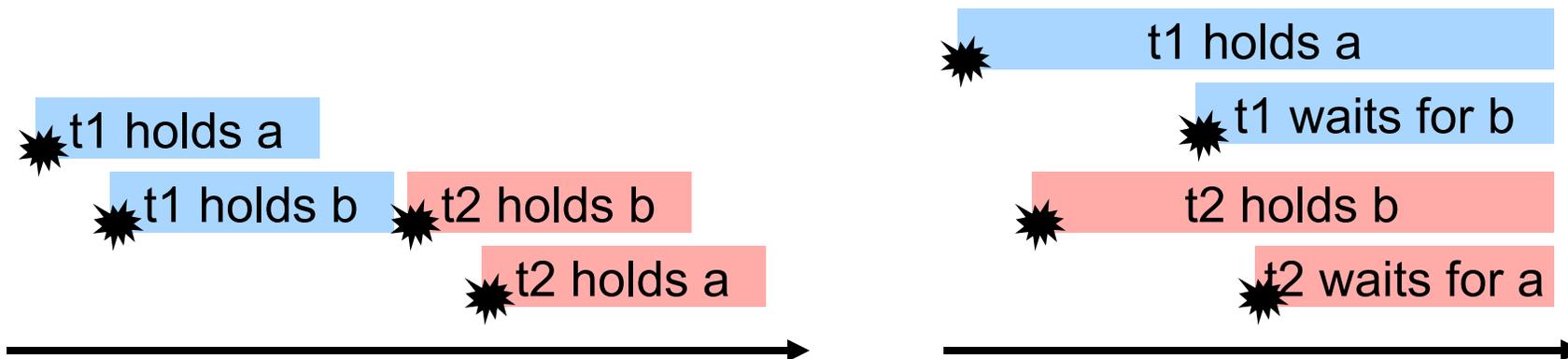
- Ensures serializability
 - * if transactions have no conflicting lock access
 - order arbitrarily
 - * for any transactions with conflicting lock access
 - order transactions based on order lock is acquired
 - * transactions are serialized
 - because, no lock is acquired after first release
 - deadlocks are still possible
- Does not ensure independence
 - * we still have *premature write* problem
 - * t1 releases x, t2 acquires x, then t1 aborts

Strict two phase locking

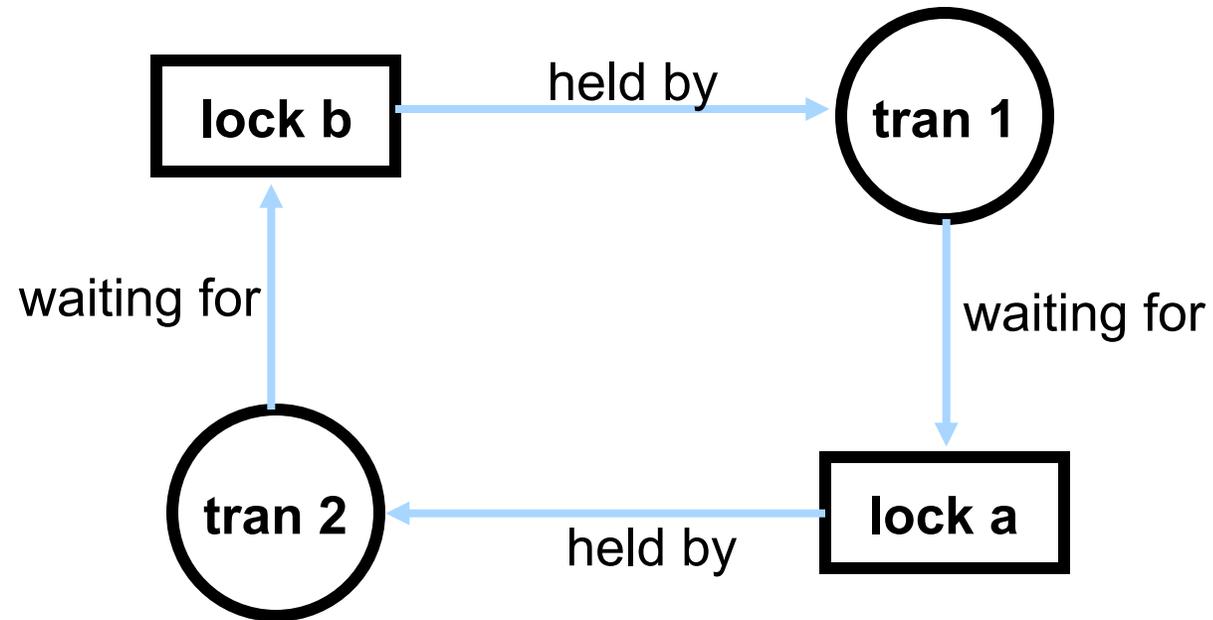
- Like two-phase locking, but
 - * release no locks until transaction commits
- Phase 1:
 - acquire locks and access data, but release no locks
- Phase 2:
 - Commit/abort transaction and then release all locks
- Ensures both serializability and independence

Serializability and two-phase locking

- Two-phase locking and ordering
 - * serial order is acquisition order for shared locks
 - * two-phase ensures that ordering is unambiguous
- Simple illustration of potential deadlock
 - * t1 acquires a then b
 - * t2 acquires b then a



Deadlock Wait Graph

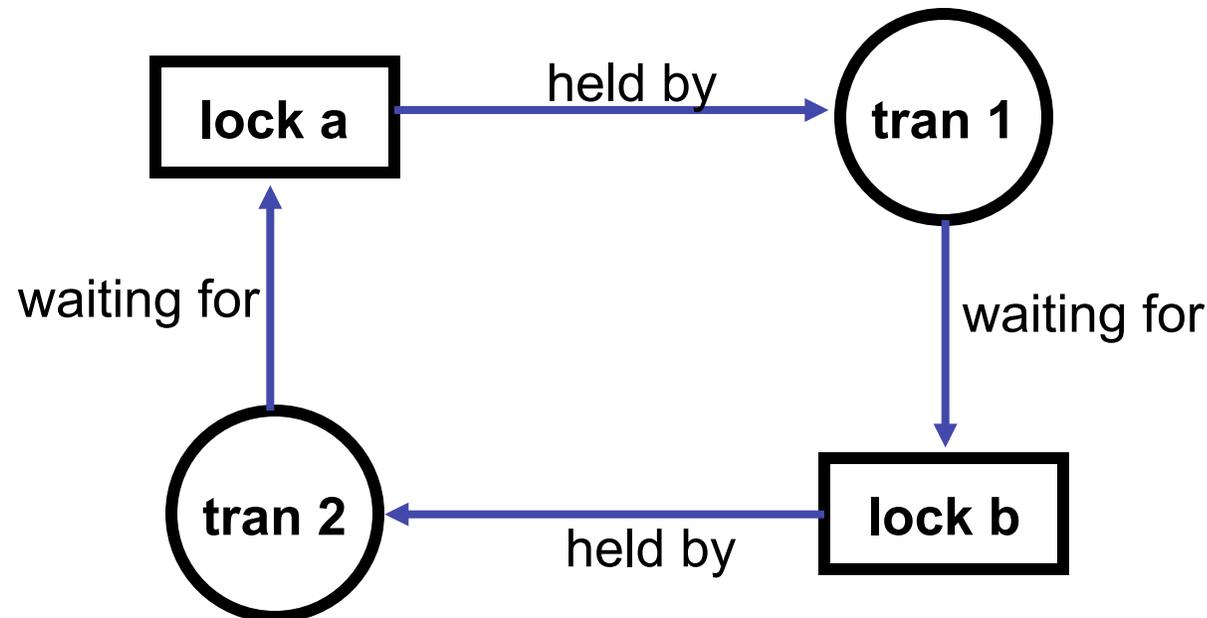


Deadlock

- Transactions increase likelihood of deadlock
 - * must hold lock until transaction commits
 - * model encourages programmers to forget about locks
- Dealing with deadlock
 - * try to prevent it
 - * detect it and abort transactions to break deadlock

Detecting and breaking deadlock

- Construct a Wait Graph as program executes
 - * all deadlocks appear as cycles in graph
- Abort transactions until cycles are broken



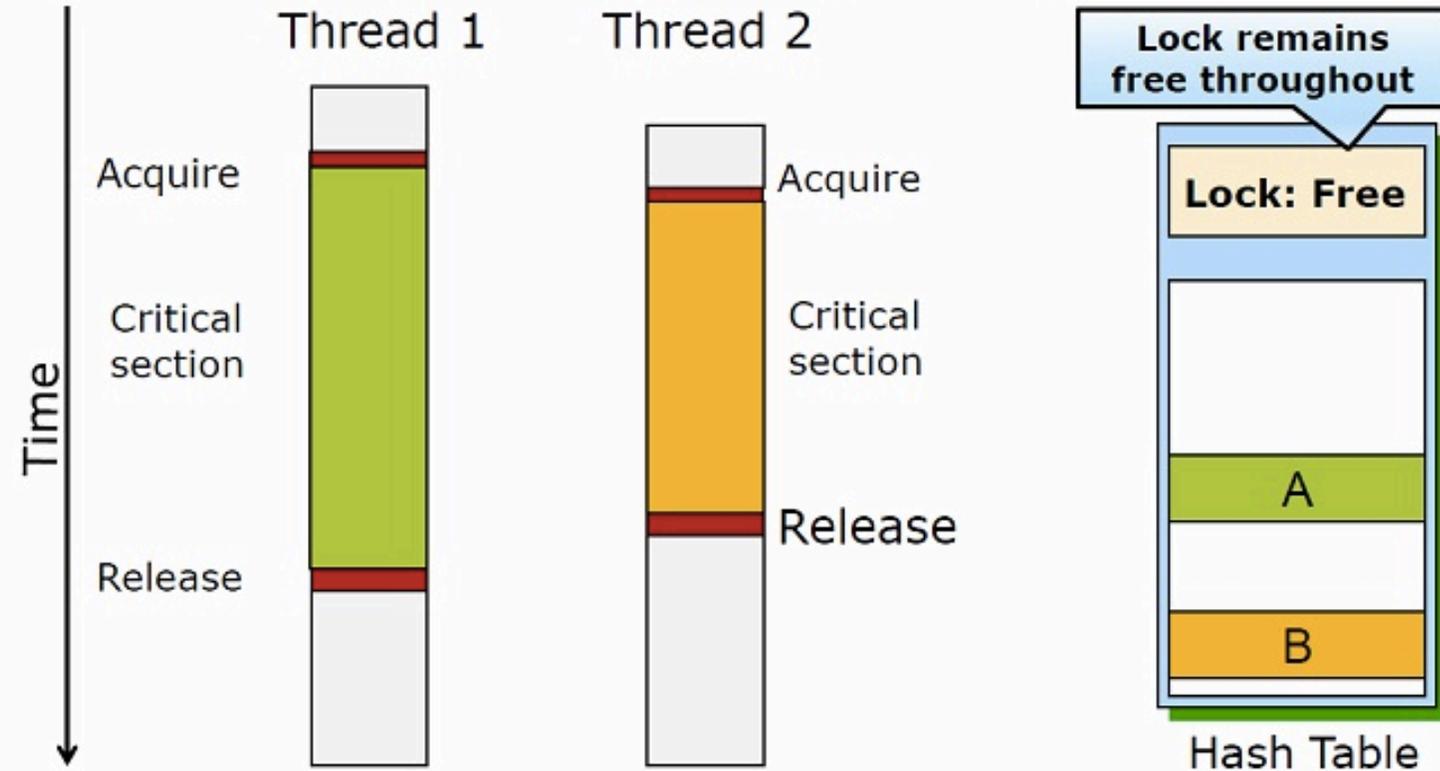
Optimistic concurrency control

- Two-Phase locking is a **paranoid** approach
 - * creates more lock conflicts than necessary
 - * especially for long running transactions
- Optimistic concurrency control
 - * no locks – process works on copies of data
 - * during commit, check for conflicts and abort if any otherwise write the copies
- Analysis
 - * (+) no overhead locking when there's no conflict
 - * (-) copies of data
 - * (-) if conflicts are common overhead much higher

Optimistic concurrency control: TX memory (note: no durability!)

Hardware TX memory (Intel's Haswell)

A Canonical Intel® TSX Execution



No Serialization and No Communication if No Data Conflicts

Recoverability (Atomicity)

- Problem

- * ensure atomic update in face of failure

- If no failure, it's easy

- * just do the updates

- If failure occurs while updates are performed

- * Roll back to remove updates or

- * Roll forward to complete updates

- * What we need to do and when will depend on just when we crash

Logging

- **Persistent (on disk) log**
 - * records information to support recovery and abort
- **Types of logging**
 - * redo logging --- roll forward
 - * undo logging --- roll back (and abort)
 - * Write-ahead logging --- roll forward and back
- **Types of log records**
 - * *begin, update, abort, commit, and truncate*
- **Atomic update**
 - * atomic operation is write of *commit* record to disk
 - * transaction committed iff *commit* record in log

Approaches to logging an update

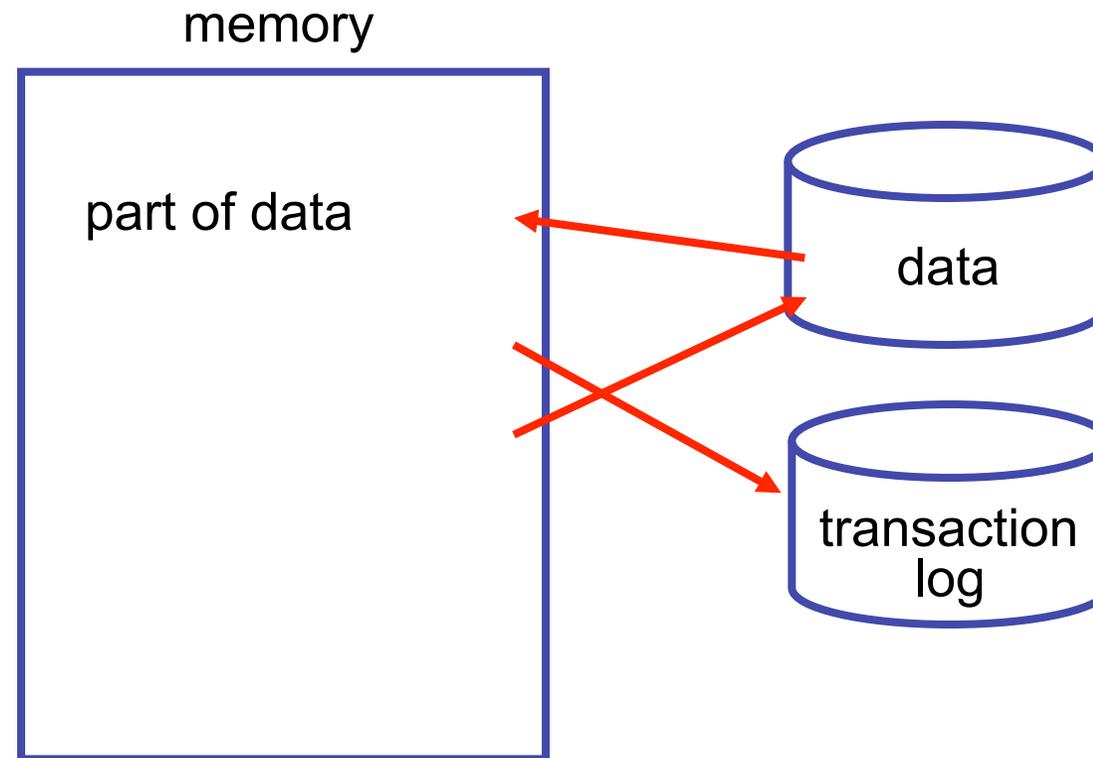
● Value logging

- * write old or new value of modified data to log
- * simple, but not always space efficient or easy
 - E.g., hard for some things such as malloc and system calls

● Operation logging

- * write name of operation and its arguments
- * usually used for redo logging
 - undo is possible, but requires a reversing operation

Transaction and persistent data



Redo logging - roll forward

Normal operation



- For each transactional update
 - * change in-memory copy (or work on a disk copy)
 - * **write new value to log**
 - * do not change on-disk copy until commit
- Commit
 - * write *commit* record to log
 - * write changed data to disk
 - * write *truncate* record to log
- Abort
 - * write *abort* record to log
 - * invalidate in-memory data
 - * reread from disk

Log what you
need to redo

Redo logging - roll forward Recovery

- When the system restarts after a failure
 - * use log to roll forward committed transactions
 - * normal access stopped until recovery is completed
- Complete committed, but untruncated transaction
 - * for every trans with a *commit* but no *truncate*
 - * read new values from log and update disk values
 - * write *truncate* record to log
- Abort all uncommitted transactions
 - * for every transaction with no *commit* or *abort*
 - write *abort* record to log

Redo logging - roll forward

Disadvantage

- No disk writes until commit so you have lots of I/O at the end to commit the transaction
- Must integrate cache of data in memory and transaction logging
 - * complicates design of both systems
- This lock-in of memory degrades performance
 - * particularly if transactions are long running or modify lots of data

Undo logging - roll backward

Normal operation



- For each transactional update
 - * write **old** value to log
 - * modify data and then write new value to disk any time
- Commit
 - * ensure that all updates have been written to disk
 - i.e., “force” or ‘flush’ updates to disk
 - * write commit record to log
- Abort
 - * use log to recover disk to old values

Log what you
need to undo

Undo logging - roll backward

Recovery

- When the system restarts after a failure
 - * use log to rollback uncommitted transactions
 - * normal access stopped until recovery completed
- Undo effect with many uncommitted transactions
 - * For every trans with no *commit* or *abort*
 - use log to recover disk to old values
 - write *abort* record to log

Undo logging - roll backward

Log records

- Begin

- * log += [b, tid]

- Update

- * log += [u, tid, addr, size, oldValue], update disk anytime

- Commit

- * complete disk update, log += [c, tid]

- Abort and Recovery

- * reapply old values for trans with b but no c or a,
log += [a, tid]

Undo logging - roll backward

Disadvantage

- Must modify disk data before commit can be written to log
- Performance impact
 - * slows commit (can't commit until all data is modified)
 - transactions hold locks longer
 - higher chance of conflicts

Write-ahead logging

● Idea

- * combine undo and redo logging

● How

- * write old values to log
- * modify data
- * write new values to log anytime before commit
- * write commit record to log
- * write data back to disk at anytime, when done write truncate record to log

Failure Recovery

- Commit but no truncate
 - * Use roll forward based on new values
- No commit
 - * Use old value to roll back

Shrinking the Log File (Truncation)

- Truncation is the process of
 - * removing unneeded records from transaction log
- For redo logging
 - * remove transactions with t or a
- For undo logging
 - * remove transactions with c or a

Transactions summary

- Key properties
 - * ACID
- Serializability and Independence
 - * two phase locking
 - serializability
 - * strict two phase locking
 - Serializability and Independence
- Recovery
 - * redo and/or undo logging