15-823

Advanced Topics in Database Systems Performance

Concurrency Control
(based on David DeWitt's notes)

Definitions

- **Database**
  - a fixed set of named resources (entities)
- **Consistency constraints**
  - must be true for DB to be considered consistent
  - **Example:**

    Σ(ACCT-BALS) = Σ(ASSETS)

    ACCT-BAL >= 0
- **Key point**

consistent database S1 —transaction T→ consistent database S2

Statement of Problem

- Concurrent execution of independent transactions
  - utilization/throughput ("hide" waiting for I/Os.)
  - response time
  - fairness
- Example:

	T1:	**T2:**
t0:	tmp1 := read(X)	
t1:		tmp2 := read(X)
t2:	tmp1 := tmp1 – 20	
t3:		tmp2 := tmp2 + 10
t4:	write tmp1 into X	
t5:		write tmp2 into X

Statement of problem (cont.)

- Arbitrary interleaving can lead to
  - Temporary inconsistency (ok, unavoidable)
  - “Permanent” inconsistency
- Need correctness criteria:
  - **schedule**: a particular action sequencing for a set of transactions
  - **consistent schedule**: each transaction sees consistent view of DB

Serializability

Assumption: all serial schedules are consistent

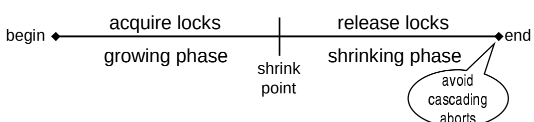
- Dependencies:
  - T1 reads X, …, T2 writes X --- **RW**
  - T1 writes X, …, T2 reads X --- **WR**
  - T1 writes X, …, T2 writes X --- **WW**
- Serialization graph
  - Nodes are Transactions T1, T2, …
  - Edges: Ti $\rightarrow$ Tj if there is RW, WR, or WW from Ti to Tj

Theorem: schedule S serializable $\Leftrightarrow$ SG(S) acyclic

- suggests (bad) technique for CC:
  build SG(S), topological sort, see if it works

Locking

- Basic idea: **lock** <entity> / **unlock** <entity>
- **Well-formed Xact**: lock, action, unlock, lock…
- **Two-phased Xact**: <lock> <actions> <unlock>



Theorem:

all Xacts well-formed or 2-phased $\Rightarrow$ any S is serializable

Gray et al. paper

- Goal: correctness and performance
- Granularity tradeoff
  - small granularity ⇒ high concurrency / high overhead
  - large granularity ⇒ low overhead / low concurrency
- Possible granularities for CC:
  - DB
  - Areas
  - Files/Relations
  - Pages
  - Records/Tuples
  - Record Fields
- Large xacts set coarse locks, small xacts set fine locks

Carnegie Mellon

Solution: Hierarchical Locking

- Shared locks S for reading
- Exclusive locks X for writing

Problem:

T1 locks (S) a record in a file, then T2 locks (X) whole file

How can T2 discover that T1 has locked the record?

Solution: Intention locks: IS and IX

Example: T1 IS file, then T1 S record

T2 cannot X file – however, T3 can IS or S file

- For more concurrency: SIX (e.g., read all – lock parts)
  - More concurrency than X lock
  - Write permission (unlike S lock)
  - Low overhead (when compared to IX lock)

Carnegie Mellon

How Does This Work?

- Let's build the lock compatibility matrix.
- Transactions lock top-down; unlock bottom-up
- Exact rules:
  - S or IS (Q) ⇐ have IS or IX on ancestors (Q)
  - X, SIX, or IX (Q) ⇐ SIX or IX on ancestors (Q)
  - Release locks bottom-up
- Tricky special case: update index field
- Examples?

Carnegie Mellon

Consistency

- "Dirty" writes
  - Until committed at end of transaction
- Levels

Degree 0: short write locks on updated items

Degree 1: long write locks on updated items
("long" means to hold until the transaction finishes)

Degree 2: long write locks on updated items, and short read locks on items read

Degree 3: long write locks on updated items, and long read locks on items read

Prevention of Inconsistency (0/1)

- Garbage reads
  T1: update(X); T2: update(X)
  - Who knows what value X will wind up holding?
  - Solution: set short write locks. **(→ degree 0)**
- Lost Updates
  T1: update(X);
  T2: update(X);
  T1: abort (restoring X to pre-T1 value)
  - At this point the update due to T2 is lost.
    (note: log contains (T1, X, [oldval, newval])
  - Solution: set long write locks. (→ **degree 1**)

Prevention of Inconsistency (2)

- Dirty Reads
  T1: update(X)
  T2: read(X)
  T1: abort
  - Now T2's read is bogus
  - Solution: long exclusive locks + short read locks **(→ degree 2)**
  - Systems often run long queries at level 2

Prevention of Inconsistency (3)

- Unrepeatable Reads

  T1: update(X)
  T1: complete transaction
  T2: read(X)
  T3: update(X)
  T3: complete transaction
  T2: read(X)

  - Now T2 has read two different values for X
  - Solution: long read locks. **(→ degree 3)**

  2-phase well-formed → degree 3 consistent

Pragmatics

- Maintain lock table as a hashed data structure
  - Preferably in main memory
- Lock/unlock must be atomic (critical section)
- Typically lock/unlock cost is 100s of instructions
- Getting this right on an SMP is a real challenge!

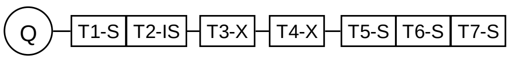
Lock Compatibility

Suppose
- T1 has a share lock on P
- T2 is waiting to gain exclusive access to P
- T3 wants shared access to P

Do we grant T3 an S lock? No! So…

Lock Queue

- For each locked Q with outstanding requests: FCFS queue
- **compatible group =** {adjacent Xacts w/ compatible modes}

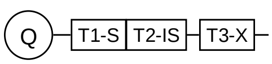


- Granted group: front compatible group
- Mode of granted group = most restrictive mode amongst members (e.g., S for S and IS or X for SIX, IX, and X)

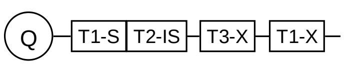
Lock Upgrades

Often want to convert

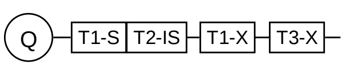
E.g., T1 does a "test-and-then-modify" action



Should T1's request go at the end of the queue?



Deadlock! Instead, put upgrades after granted group

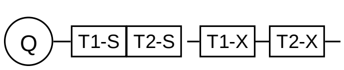


Deadlock

- In OS world, usually due to errors or overloads
- In DB system with 2PL, inherent!!!
- Common cause: Lock mode upgrades

(1) T1: S-lock Q (2) T2: S-lock Q

(3) T1: convert S(Q) to X-lock (4) T2: convert S(Q) to X-lock



- **Deadlock!**

Another deadlock

- Differing access orderings.

  T1: X-lock P
  T2: X-lock Q
  T1: X-lock Q /* block, waiting for T2 */
  T2: X-lock P /* block, waiting for T1 */

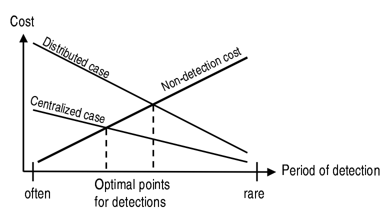
- **Deadlock!**

Deadlock Detection

- Use “waits-for” graph and look for cycles.
- Empirically, in actual systems, in waits-for graph:
  - Cycles fairly rare.
  - Cycle length usually 2, sometimes 3, virtually never > 3.
  - Use DFS to find cycles.
- When should we look for cycles? Options:
  - Whenever a transaction blocks
  - Periodically
  - Never (use timeouts)

Deadlock Detection (cont.)

- **Centralized** systems: deadlock detection upon blocking
  (Cheap – most recently blocked transaction (T) must be the one that caused the deadlock, so just DFS starting from T)
- **Distributed** systems: periodic detection

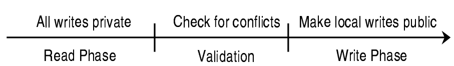


Victim Selection Criteria

- Goals
  - minimize wasted work
  - minimize time to get back to point of restart
- Selecting a victim
  - current blocker
  - youngest XACT
  - least resources used
  - fewest locks held (commonly used)
  - fewest number of restarts

Optimistic CC (Kung&Robinson)

- Assumption: conflicts are rare
- Optimize for the no-conflict case.
- All transactions consist of three phases
  - **Read:** Here, all writes are to private storage.
  - **Validation:** Make sure no conflicts have occurred.
  - **Write**: If Validation was successful, make writes public. (If not, abort!)



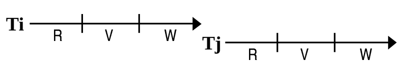
Why Might this Make Sense?

- All transactions are readers
  - The system will be setting and releasing locks for no reason at all
- Lots of transactions, each accessing/modifying only a small amount of data, large total amount of data
  - Low probability of conflict, so again locking is wasted
- Fraction of transaction execution in which conflicts “really take place” is small compared to total path length
  - Locks until of transaction are way too restrictive most of the time

Validation Phase (1)

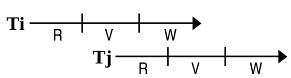
- Goal: guarantee only serializable schedules result.
- Technique:
  - Assign each transaction a TN (transaction number)
  - Require TN order to be the serialization order
  - If TN(Ti) < TN(Tj) ⇒ **ONE** of the following must hold:

1. Ti completes W before Tj starts R



Validation Phase (2)

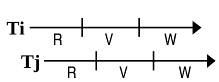
2. WS(Ti) ∩ RS(Tj) = ∅ **and** Ti completes W before Tj starts W



Comments:
- No problem with Tj reading values previous to Ti's writes (nothing in common there)
- No problem with Ti overwriting Tj's writes (no overlap in time)

Validation Phase (3)

3. WS(Ti) ∩ RS(Tj) = ∅ **and**
   WS(Ti) ∩ WS(Tj) = ∅ **and**
   Ti completes its R before Tj completes its W



Comments:
- No problem with Tj getting (or missing) input from Ti, as there is nothing that Ti writes that Tj touches
- Since Ti finishes its R before Tj finishes its R, Ti won't read any output from Tj either
- No overwrite problems as write-sets are disjoint

Correctness

All of conflict types (WR, RW, WW) go one way

- Condition 1: true serial execution
- Condition 2
  - No W-R conflicts since WS(Ti) intersect RS(Tj) = NULL
  - In R-W conflicts, Ti precedes Tj, since Ti's W (and hence R) of Ti precedes that of Tj
  - In W-W conflicts, Ti precedes Tj by definition
- Condition 3
  - No W-R conflicts since WS(Ti) intersect RS(Tj) = NULL
  - No W-W conflicts since WS(Ti) intersect WS(Tj) = NULL
  - In all R-W conflicts, Ti precedes Tj, since the Ti's R precedes Tj's W

Carnegie Mellon

Observations

- Better assign TN's at beginning of validation phase
- T with very long R: check ALL T's within its lifetime
  - Requires unbounded buffer space
  - Solution: bound buffer, toss out when full, abort possibly affected T's
  - Starvation!
- Serial/Parallel validation – Pros & cons?
- [To be continued…]

Carnegie Mellon