Transaction Management: Concurrency Control, part 1

CS634 Class 15

Slides based on “Database Management Systems” 3rd ed, Ramakrishnan and Gehrke

Transaction Execution

- Example: Reading Uncommitted Data (Dirty Reads)

```
T1:   R(A), W(A),                                 R(B), W(B)
T2:               R(A), W(A), R(B), W(B)
```

- We are assuming each transaction is single-threaded
  - Usually the case in practice, though not universal
- And, for simplicity, that operations for the whole DB happen in some order, possibly interleaving the transactions
  - This is not true in reality: in fact, parallel execution of transactions happens on multi-processors,
  - But it’s close enough to show the important behaviors

Transaction Schedule Notation

- Example: Reading Uncommitted Data (Dirty Reads)

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

Another notation: Using subscripts for transaction ids

- Arrows mark conflicts, yield arcs in PG: T1->T2, T2->T1

$$R_1(A)\ W_1(A)\ R_2(A)\ W_2(A)\ R_2(B)\ W_2(B)\ R_1(B)\ W_1(B)$$

Note: commits are not involved in locating conflicts

Example: RW Conflicts

- Unrepeatable Reads

T1:	R(A),		R(A), W(A), Commit
T2:		R(A), W(A), Commit	

- Alternatively:

$$R_1(A)\ R_2(A)\ W_2(A)\ C_2\ R_1(A)\ W_1(A)\ C_1$$

- Again T1->T2, T2->T1, cycle in PG, not conflict serializable
- See conflicts reaching across a commit here

Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule *S* is conflict serializable if *S* is conflict equivalent to some serial schedule
- Example: T1->T2 only, and conflict serializable, as shown below

$$R_1(A)\ R_1(B)\ W_1(C)\ R_2(B)\ W_2(A)\ R_2(C)\ R_1(B)\ C_1 C_2$$

$$R_1(A)\ R_1(B)\ W_1(C)\ R_1(B)\ C_1\ R_2(B)\ W_2(A)\ R_2(C)\ C_2$$

Dependency Graph

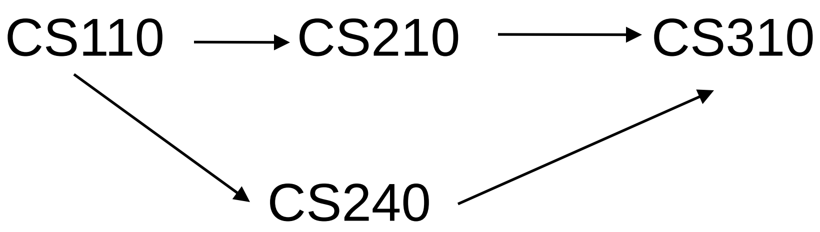
- *Dependency graph*:
  - one node per transaction
  - edge from *Ti* to *Tj* if action of *Ti* precedes and conflicts with action of *Tj*
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic
  - Equivalent serial schedule given by topological sort of dependency graph

From cs310: Definitions

- Path
  - A sequence of vertices $w_1..w_n$ connected by edges s.t. $\{w_i,w_{i+1}\}\varepsilon$ E for each i=1..n.
- Path length
  - Number of edges on the path
- Cycle
  - A path that begins and ends at the same vertex and contains at least one edge
- Directed Acyclic Graph (DAG)
  - A type of directed graphs that has no cycles

A cycle in the graph, DAG

- A cycle in a digraph is a path that returns to its starting vertex.
- An acyclic digraph is also called a DAG, short for directed acyclic graph. These graphs show up in lots of applications. For example, the graph of course prerequisites.



- is a DAG. A cycle in prerequisites would be ridiculous.

DAG's and topological sorts

- A DAG induces a partial order on the nodes.
- Not all element pairs have an order, but some do, and the ones that do must be consistent. So CS110 < CS210 < CS310, and so CS110 < CS310, but CS210 and CS240 have no order between them.
- Suppose a student took only one course per term in CS. Then they would be finding a sequence that satisfies the partial order requirements, for example CS110, CS210, CS240, CS310. Another possible sequence is CS110, CS240, CS210, CS310.
- One of these fully ordered sequences that satisfy a partial order or DAG is called a *topological sort* of the DAG.
- A topological sort orders the nodes such that if there is a path between two nodes u and v, u will appear before v.

Finding a topological sort

- Weiss (author of cs310 book) presents a non-recursive algorithm for finding a topological sort of a DAG, checking that it really has no cycles.
- The first step of this algorithm is to determine the in-degree of all vertices in the graph.
- The in-degree of a vertex is the number of edges in the graph with this vertex as the to-vertex.
- Once we have all the in-degree numbers for the vertices, we look for a vertex with in-degree 0.
- It has no incoming edges, and so can be the vertex at the start of a topological sort, like CS110.

Finding a topological sort (cont.)

- Notice that there must be a node with in-degree 0.
- If there weren't, then we could start a path anywhere, extend backwards along some in-edge from another vertex and from there to another, etc.
- Eventually we would have to start repeating vertices.
- For example, if we have managed to avoid repeating vertices and have visited all the vertices, then the last vertex still has an in-edge not yet used, and it goes to another vertex, completing a cycle.
- Thus the lack of an in-degree-0 vertex is a sure sign of a cycle and a DAG doesn't have any cycles.
- OK, we have the very first vertex, but what about the rest? Think recursively!

A Topological Sort Example

The image part with relationship ID rId3 was not found in the file.

In-degree

The topological order is:

$V_2, V_0, V_1, V_3, V_4, V_6, V_5$

Back to our text: Dependency Graph

- *Dependency graph*:
  - one node per transaction
  - edge from *Ti* to *Tj* if action of *Ti* precedes and conflicts with action of *Tj*

- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic
  - Equivalent serial schedule given by topological sort of dependency graph

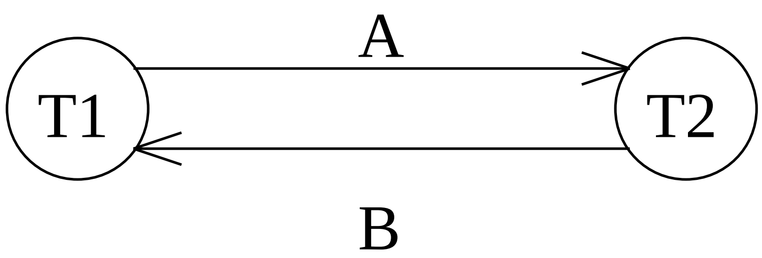
Example

- A schedule that is not conflict serializable:

T1: R(A), W(A), R(B), W(B)

T2: R(A), W(A), R(B), W(B)

$R_1(A)\ W_1(A)\ R_2(A)\ W_2(A)\ R_2(B)_2\ W(B)_2\ R_1\ (B)\ W_1\ (B)$



Dependency graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Strict Two-Phase Locking (Strict 2PL)

- Protocol steps
  - Each transaction must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held are released when the transaction completes
    - (Non-strict) 2PL: Release locks anytime, but cannot acquire locks after releasing any lock.

- Strict 2PL allows only serializable schedules of R/W ops.
  - It simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Strict 2PL Example

T1: S(A) R(A) S(B) R(B) C
T2: S(A) R(A) X(B) R(B)W(B) C

where S_1 (B) blocked

Using subscripted notation: blow-by-blow actions

$S_1(A)\ R_1(A)\ S_2(A)\ R_2(A)\ X_2(B) < S_1(B)\text{-blocked} > R_2(B)\ W_2(B)$
$C_2 < S_1(B)\text{-unblocked} > R_1(B)\ C_1$

Aborting Transactions

- When *Ti* is aborted, all its actions have to be undone
  - if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
  - *cascading aborts* can be avoided by releasing locks only at commit
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits

- In Strict 2PL, cascading aborts are prevented
  - At the cost of decreased concurrency
  - No free lunch!
  - Increased parallelism leads to locking protocol complexity

Deadlocks

- Cycle of transactions waiting for locks to be released by each other: case of "deadly embrace"

T1: X(A) W(A) S(B) [R(B) …]
T2: X(B) W(B) S(A) [R(A) …]

Using subscripted notation:

$X_1(A)\ W_1\ (A)\ X_2(B)\ W_2(B)$ <$S_2(A)$ blocked> <$S_1(B)$ blocked>…

Deadlock Detection

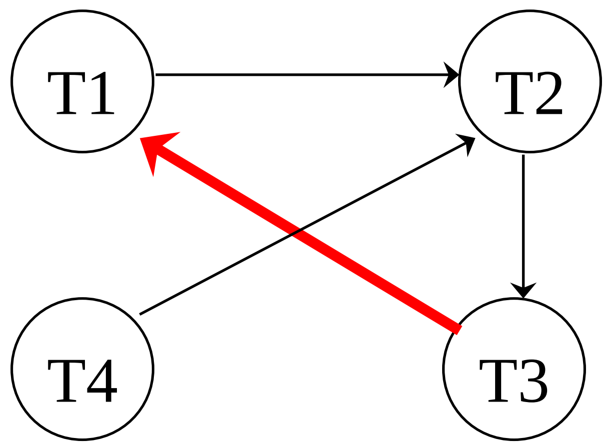
- Create a waits-for graph:
  - Nodes are transactions
  - Edge from *Ti* to *Tj* if *Ti* is waiting for *Tj* to release a lock

T1: S(A), R(A), S(B)

T2: X(B),W(B) X(C)

T3: S(C), R(C) X(A)

T4: X(B)



Deadlock Prevention

- Assign priorities based on timestamps
- Assume *Ti* wants a lock that *Tj* holds
  - Wait-Die: It *Ti* > *Tj*, *Ti* waits for *Tj*; otherwise *Ti* aborts
  - Wound-wait: If *Tj* > *Ti*, *Tj* aborts; otherwise *Ti* waits
    - In use in Google Spanner, "The first horizontally scalable, strongly consistent, relational database service", as product released May, 2017
- Fairness is an issue
  - If transaction re-starts, make sure it has its original timestamp
  - Otherwise starvation may occur
- In practice, not used for 2PL locks in centralized DBs (but may be in use for mutex-related mechanisms ("latches") to be covered later), and

More Dynamic Databases

- If the set of DB objects changes, Strict 2PL using row or page locks will not ensure serializability:
  - Phantoms (anomalies involving sets of rows) are still possible
  - Locking whole tables will work but is horribly slow
  - Example (with insert phantom and delete phantom): pg 560
    - ➢T1 finds oldest sailor for each of rating=1 and rating=2
    - ➢T2 does an insertion and a deletion

  1. T1 locks all rows/pages with *rating* = 1, finds oldest sailor (*age* = 71)
  2. Next, T2 inserts a new sailor; *rating* = 1, *age* = 96
  3. T2 deletes oldest sailor with rating = 2 (*age* = 80), commits
  4. T1 locks all rows/pages with *rating* = 2, and finds oldest (*age* = 63)

- No serial schedule gives same outcome!
- T1 sees old set for rating 1, new set for rating 2: can't happen serially.
- Database must prevent this if running at full Serializable isolation.

The “Phantom” Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1
  - Unless running at serializable isolation, it really only locked the ones it accessed, and unlocked them again if running at RC (short-term reads)

- Two mechanisms to address the problem
  - Index locking
  - Predicate locking—not used in practice (except for index locking, considered a type of predicate locking)

Another phantom example

- Table tasks has one row for each worker task, with worker name, task name, number of hours
- Rule that no worker has more than 8 hours total
- Application A to add a task sums hours for worker, adds task if it fits under 8 hours max
  - T1 running A sees 'Joe' has 6 hours, adds task of 2 hours
  - Concurrently, T2 running A sees 'Joe' has 6 hours, adds task of 1 hour.
  - Joe ends up with 9 hours of work.
- Again, the problem is there is no lock on the *set* of rows being examined to make a decision

Index Locking

- Assume index on the *rating* field
- T1 should lock the index page(s) containing the data entries with *rating* = 1, and their immediate neighbors
  - If there are no records with *rating* = 1, T1 must lock the index page where such a data entry *would* be, if it existed!
  - e.g., lock the page with rating = 0 and beginning of rating=2
  - Or lock pages for just one extra data item on one side, if a lock is understood to cover the key value plus gap to one side.
- If there is no suitable index, T1 must lock all data pages, and lock the file to prevent new pages from being added

Index Locking: row locks

- Assume index on the *rating* field
- Row locking is the industry standard now
- T1 should lock all the data entries with *rating* = 1 and at least one neighbor (depending on details of protocol)
  - If there are no records with *rating* = 1, T1 must lock the entries adjacent to where data entry *would* be, if it existed!
  - e.g., lock the last entry with rating = 0 and beginning of rating=2
- If there is no suitable index, T1 must lock all the rows and lock the file to prevent new rows from being added, or use a “table lock”.

Predicate Locking

- Grant lock on all records that satisfy some logical predicate
  - But note that a general predicate can depend on *data* in the row: salary > 50000 + 1000*years
  - Or a whole table: salary > (select avg(salary) in emps)

- Index locking is a special case of predicate locking
  - Index supports efficient implementation of the predicate lock
  - Predicate is specified in WHERE clause

- In general, predicate locking is expensive to implement!
  - Can avoid the runtime cost by using Repeatable Read isolation level, but that opens up anomaly possibilities.

Index Locking, Blow by blow

- Index locking happens in the storage engine, based on FILE calls coming from query processor as directed by the query plan
- Example: Transaction T1 accesses a heap table with certain index, gets row for certain index key value, say 100. Suppose the next data entry is for another key, 102.
  - Storage engine share-locks the accessed data entry for key 100, guarding it and the gap between that key and the next key.
  - Then if another transaction T2 tries to change the row with key 100, can't get necessary X lock, waits. Same with key 101.
  - Original transaction T1 can ask for next key, get 102.
  - But if another transaction updates row with key 102 (not guarded by T1's share lock), then then T1 has to wait for the next key.

Index Locking Scenario, cont.

- There is an underlying assumption in that story: that all the accesses in fact use the index on this column.
- Well, the important thing is that all accesses that change the column value go through the index. It's OK for another reader to access the value.
- An insert or delete needs to change the index, so they are naturally involved.
- An update to this column also needs to change the index, in two places, so it also collides with the old lock.
- You can see this has to be checked out carefully!