**Topics**

- Conflicts and Serializability
- Locks
- Phantom Problem
- Non-2PL Concurrency Protocols

**Conflicts and Serializability**

- Two actions conflict if they operate on the same data object and at least one of these is a write.
- If two actions do not conflict then we can swap their order to create a new schedule without altering effect of schedule on the database.
Conflicting Actions

- We are dealing with only read and write instructions.
- For instruction J belonging to Tj and K belonging to Tk, 4 cases:
  1. J = R(A), K = R(A)
     - The relative order of executing J and K does not matter.
  2. J = R(A), K = W(A)
     - If order is reversed, Tj reads a different value of A.
  3. J = W(A), K = R(A)
     - Similar to case 3.
  4. J = W(A), K = W(A)
     - A subsequent R(A) in eithertxn. is impacted if order is reversed.
     - If there are no other W(A) statements the final value in the database is affected.

Example: Swapping non-conflicting actions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td>W(B)</td>
</tr>
</tbody>
</table>

Conflict Serializable Schedules

- Conflict Equivalence
  - Two schedules are conflict equivalent if:
    - Involve the same actions of the same transactions
    - Every pair of conflicting actions is ordered the same way
- Conflict Serializable
  - Schedule S is conflict serializable if S is conflict equivalent to some serial schedule
  - One way to think about it:
    - Take a serial schedule S
    - Transform it into a new schedule S' by a series of swaps of non-conflicting instructions
    - S' is conflict serializable
Example: Schedule not conflict serializable

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td>W(B)</td>
</tr>
</tbody>
</table>

Conflict Serializable & Serializable

- Every conflict serializable schedule is serializable
  - The converse is not true
- Example:

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A)</td>
<td>W(A)</td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>Commit</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>

Dependency/Precedence Graph

- **Dependency graph:**
  - One node per txn
  - Edge from $T_i$ to $T_j$ if $T_j$ reads/writes an object last written by $T_i$
- **Theorem:**
  - Schedule is conflict serializable if and only if its dependency graph is acyclic
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)

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

Strict 2PL and Serializability

- Protocol:
  - Each txn must obtain a S lock before reading, and an X lock before writing.
  - Hold all locks until the txn completes
  - If a txn holds an X lock on an object, no other txn can have a lock (S or X) on that object.
- Strict 2PL allows only schedules whose precedence graph is acyclic
  -> Schedules are conflict serializable
  -> Schedules are serializable

Topics

- Conflicts and Serializability
  - Locks
    - Lock Management
    - Concurrency Control in B+ trees
    - Multi-Granularity Locking
  - Phantom Problem
  - Non-2PL Concurrency Protocols
**Lock Table**

- Lock and unlock requests are handled by the lock manager.
- Lock table:
  - Hash table: key is object on which lock is held.
- Lock table entry:
  - Linked list of:
    1. Transactions currently holding a lock
    - Type of lock held (shared or exclusive)
    2. Transactions requesting the lock

**Lock Management**

- Lock request arrives:
  - Add entry to the end of the linked list
  - Always grant the first lock request in the entry
- Unlock message:
  - Delete entry for the data item
- Above algorithm prevents starvation
- Locking and unlocking have to be atomic operations

**Topics**

- Conflicts and Serializability
- Locks
  - Lock Management
    - Concurrency Control in B+ trees
    - Multi-Granularity Locking
- Phantom Problem
- Non-2PL Concurrency Protocols
Concurrency Control in B+ Trees

- One conservative approach
  - Treat each page in the tree as data object
  - For search, lock pages in S mode
    - starting from the root page and proceeding down to the leaf
  - For insert/delete/update, lock pages in X mode
    - starting from the root page and proceeding down to the leaf
- The above approach can drastically reduce concurrency in the presence of updates
  - Very high contention for nodes at the top of the tree, in particular, root node

How to Improve?

- Two observations allow us to use properties of B+ tree
  1. The higher levels of the tree only direct searches, but don’t contain real data
  2. For inserts a node needs to be locked only if a split can propagate from a leaf to this node

Crabbing Protocol

- When searching for a key value:
  - Lock the root node in S mode
  - When traversing down the tree, acquire an S lock on the child node
  - After acquiring the lock on the child node, release the lock on the parent node
  - Continue till leaf node is reached
- Justification:
  - Searches never go back up the tree
Crabbing Protocol - contd

- When inserting a key value:
  - Follow the same protocol as for searching except
  1. If the child node is full, then don’t release the lock on the parent node when the child is locked
  2. When leaf node is reached, lock in X mode
  3. If the node needs to be split, lock the parent in X mode

- Justification:
  - Keep a node locked only if there is the possibility that the node may need to be split

- Of course, deadlocks can occur when an attempt to lock the parent node is made

- The DBMS can handle these by just re-starting one of the operations from the root node after finishing the other operation

Topics

- Conflicts and Serializability
- Locks
  - Management
  - Concurrency Control in B+ trees
    - Multi-Granularity Locking
  - Phantom Problem
  - Non-2PL Concurrency Protocols

Multi-Granularity Locking

- Up to this point, we have used individual data items as the unit on which synchronization is performed
- In some situations a group of items needs to be treated as one synchronization unit
  - E.g.,
    - If Ti wants to modify all the records in a table, lock the whole table
    - If Tj wants to modify only one record in a table, lock only that row
- Issue: Locks at one granularity can conflict with locks at another granularity
  - How can the DBMS efficiently manage these?
Granularity Hierarchy
- Exploit the hierarchical nature of ‘contains’ relationship among physical database objects
- Data “containers” are nested:

<table>
<thead>
<tr>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
</tr>
<tr>
<td>Tables</td>
</tr>
<tr>
<td>Pages</td>
</tr>
<tr>
<td>Tuples</td>
</tr>
</tbody>
</table>

Solution: New Lock Modes, Protocol
- Allow txns to lock at each level, but with a special protocol using new “intention” locks
- Before locking an item, txn must set “intention locks” on all its ancestors
- For unlock, go from specific to general (i.e., bottom-up)
- Must release locks in bottom-up order

Example
- T1 wants to read rows from R
- T2 wants to modify R
- T1 gets:
  - IS on DB, Table, Page
  - S on rows in R
- T2 wants to get X on Table
  - Can’t get it
Topics

- Conflicts and Serializability
- Locks
- Phantom Problem
- Non-2PL Concurrency Protocols

Dynamic Databases

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (say, age = 71)
  - Next, T2 inserts a new sailor; rating = 1, age = 96
  - T2 also deletes oldest sailor with rating = 2 (and, say, age = 80), and commits
  - T1 now locks all pages containing sailor records with rating = 2, and finds oldest (say, age = 63)
- No consistent DB state where T1 is “correct”!

Phantom Problem

- T1 implicitly assumes that it has locked the set of all sailor records with rating = 1
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!
Handling Phantoms: Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. $\text{age} > 2 \times \text{salary}$
- Index locking:
  - One mechanism for predicate locking
- In general, predicate locking has a lot of locking overhead

Topics

- Conflicts and Serializability
- Locks
- Phantom Problem
- Non-2PL Concurrency Protocols
  - Timestamp Concurrency Control
  - Multiversion Concurrency Control

Timestamp Concurrency Control

- Idea:
  - Give each txn a timestamp (TS) when it begins
    - Txn timestamps have increasing values
  - Give each object X
    - A read-timestamp (RTS)
    - Highest timestamp of txn that has read X
    - A write-timestamp (WTS)
    - Highest timestamp of txn that has written X
  - Timestamps of transactions determine the serializability order
    - i.e., timestamp order of txns is the serial order in which they must appear to execute
When Txn T wants to read Object O

- If TS(T) < WTS(O)
  - T wants to read a value that was already overwritten
  - Abort T and restart it with a new, larger TS
    - If restarted with same TS, T will fail again!
- If TS(T) ≥ WTS(O):
  - Allow T to read O
  - Reset RTS(O) to max(RTS(O), TS(T))
- Change to RTS(O) on reads must be written to disk!
  - This write and restarts represent overheads

When Txn T wants to Write Object O

- If TS(T) < RTS(O)
  - Value that T wants to over-write has been read by a txn started after T
  - Abort and restart T
- If TS(T) < WTS(O)
  - Value that T wants to over-write was written by a txn started after T
  - Abort and restart T
- Else, allow T to write O
  - Reset WTS(O) = TS(T)

Improvement: Thomas Write Rule

- If TS(T) < RTS(O), same as before
- If TS(T) < WTS(O)
  - Thomas Write Rule:
    - Don’t restart T and just ignore such outdated writes
      - T’s write is effectively followed by another write, with no intervening reads
      - Allows nonconflict serializable schedules
      - But are still serializable
  - Else, same as before
**Timestamp CC vs. 2PL**

- Timestamp:
  - No deadlocks. In 2PL, deadlocks possible
  - Starvation possible. No starvation in 2PL
- Unfortunately, unrecoverable schedules are allowed
- Timestamp CC can be modified to allow only recoverable schedules
  - Will need locking!
  - Still not the same as 2PL
- In practice, use is restricted to distributed databases

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td>Commit</td>
</tr>
</tbody>
</table>

If T1 aborts, T2 has committed an unrecoverable commit.

---

**Topics**

- Conflicts and Serializability
- Locks
- Phantom Problem
- Non-2PL Concurrency Protocols
  - Timestamp Concurrency Control
  - Multi-version Concurrency Control

---

**Multiversion Timestamp CC**

- Idea:
  - A txn T aborted on read of X when the value T wanted to read had been overwritten by a later txn that wrote X
  - Let writers make a “new” copy while readers use an appropriate “old” copy

![Main Segment](image1.png)

![Version Pool](image2.png)

*MAIN SEGMENT* (Current versions of DB objects)

*VERSION POOL* (Older versions that may be useful for some active readers)
**Multiversion CC - contd.**

- Each version of an object has:
  - Value
  - WTS = its writer’s TS
    - Never updated since associated with a version of an element
  - RTS = TS of the txn that most recently read this version
    - Gets updated on successful reads
- Versions are chained backward
- We can discard versions that are “too old to be of interest”
  - If a version with WTS = m exists, and
  - No active txn T with TS(T) < m, then
  - Discard all versions with WTS < m

**Read Operation**

- For each object to be read:
  - Finds newest version with WTS <= TS(T)
  - Starts with current version in the main segment and chains backward through earlier versions
  - RTS of this version update to max(RTS, TS(T))
- Assuming that some version of every object exists from the beginning of time, *read never blocked*

![WTS timeline](chart)

**Write Operation**

- To write an object:
  - Finds newest version V s.t. WTS <= TS(T)
  - If RTS(V) <= TS(T),
    - If WTS(V) = TS(T) overwrite V
    - Else makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T)
  - Else, reject write

![WTS timeline](chart)
Example 1: Multiversion CC

Version of an object A created by txn X

\[ A_x = <value, RTS, WTS> \]

<table>
<thead>
<tr>
<th>Txn name</th>
<th>Operation</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0, T1(T1) = 10</td>
<td>W(A) = 50</td>
<td>A0 = &lt;50, 10, 10&gt;</td>
</tr>
<tr>
<td>T1, T2(T2) = 12</td>
<td>W(A) = 100</td>
<td>A1 = &lt;100, 12, 12&gt;</td>
</tr>
<tr>
<td>T2, T3(T3) = 15</td>
<td>W(A) = 14</td>
<td>A2 = &lt;120, 15, 15&gt;</td>
</tr>
<tr>
<td>T3, T4(T4) = 15</td>
<td>W(A) = 100</td>
<td>A3 = &lt;200, 15, 15&gt;</td>
</tr>
<tr>
<td>T4</td>
<td>W(A)</td>
<td>Abort T2</td>
</tr>
</tbody>
</table>

Example 2: Multiversion CC

Version of an object A created by txn X

\[ A_x = <value, RTS, WTS> \]

<table>
<thead>
<tr>
<th>Txn name</th>
<th>Operation</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0, T1(T1) = 10</td>
<td>W(A) = 50</td>
<td>A0 = &lt;50, 10, 10&gt;</td>
</tr>
<tr>
<td>T1, T2(T2) = 12</td>
<td>W(A) = 100</td>
<td>A1 = &lt;100, 12, 12&gt;</td>
</tr>
<tr>
<td>T2, T3(T3) = 15</td>
<td>W(A) = 14</td>
<td>A2 = &lt;200, 15, 15&gt;</td>
</tr>
<tr>
<td>T3, T4(T4) = 15</td>
<td>W(A) = 100</td>
<td>A3 = &lt;200, 15, 15&gt;</td>
</tr>
<tr>
<td>T4</td>
<td>W(A)</td>
<td>Abort T2</td>
</tr>
</tbody>
</table>

Multiversion Two-Phase Locking

- In multiversion CC protocol every read requires updating the RTS
  - Can be very inefficient
- For better performance (and recoverability), multiversion two-phase locking protocols are used
  - Differentiate between read-only txns and update txns
  - Update txns perform rigorous two-phase locking
    - Also called Read-Write or RW txns
    - Read txns never get locks
**Timestamps in Multiversion 2PL**

- Timestamps based on a counter `ts-counter`
- Each version of data has a single timestamp
- Read-only txn Ti
  - Assigned a timestamp by reading the value of `ts-counter` just before start of execution
  - Reading object X returns the version of X with largest `TS(X) < TS(Ti)`

---

**Read-Write Txn $T_k$ in Multiversion 2PL**

- Gets a shared lock on the item it wants to read, and
- Gets an exclusive lock on item it wants to write
  - Creates a new version of the item
  - Sets the timestamp to $\infty$
    - i.e., a value larger than any possible timestamp
  - But this new version is not available to any other txn until after commit
- At commit processing
  - Sets the timestamp of every version it has created to `ts-counter + 1`, and
  - Increments `ts-counter` by 1

---

**Multiversion 2PL: Properties**

- Any read txn that:
  - Starts after $T_k$ committed will see the value written by $T_k$
  - Started before $T_k$ committed will see the value before the update by $T_k$
- Read-only txn
  - Never has to wait for locks
  - Never causes an update
    - i.e., no RTS needs to be updated
- Schedules produced are recoverable
Example: Multiversion 2PL

- Version of an object A created by txn X: Ax = <value, WTS>
- Txns T0 and T1 are RW txns, T2 and T3 are read-only txns

<table>
<thead>
<tr>
<th>Txn name</th>
<th>Operation</th>
<th>ts-counter</th>
<th>Local</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>W(A) = 50</td>
<td>9</td>
<td>A0 = &lt;50, 1&gt;</td>
<td></td>
</tr>
<tr>
<td>T0, TS(T1) = 10</td>
<td>Commit</td>
<td>10</td>
<td>A0 = &lt;50, 10&gt;</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>W(A) = 100</td>
<td>10</td>
<td>A1 = &lt;100, 1&gt;</td>
<td></td>
</tr>
<tr>
<td>T2, TS(T2) = 10</td>
<td>R(A) = 50</td>
<td>10</td>
<td>A0 = &lt;50, 10&gt;</td>
<td></td>
</tr>
<tr>
<td>T2, TS(T2) = 10</td>
<td>Commit</td>
<td>11</td>
<td>A1 = &lt;100, 11&gt;</td>
<td></td>
</tr>
<tr>
<td>T3, TS(T3) = 11</td>
<td>R(A) = 100</td>
<td>11</td>
<td>A1 = &lt;100, 11&gt;</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>&quot;Commit&quot;</td>
<td>11</td>
<td>Can now delete A0</td>
<td></td>
</tr>
</tbody>
</table>

Summary

- We looked at different concurrency protocols
  - Strict and non-strict 2PL
  - Timestamp CC
  - Multiversion CC: with and without 2PL
- Not all DBMS use the strict 2PL protocol and the default isolation level may not be the same
- In particular, Oracle uses multi-version concurrency control with 2PL
  - Default isolation level is READ COMMITTED, and not SERIALIZABLE
    - Guarantees statement level read consistency, but not txns level read consistency
- A transactional application designed to run against multiple DBMS must guard against potential difference in behavior

Read Committed Behavior in Oracle

- Example from Slide 46 with READ COMMITTED
- T0 and T1 are RW txns, T2 and T3 are read-only txns

<table>
<thead>
<tr>
<th>Txn name</th>
<th>Operation</th>
<th>ts-counter</th>
<th>Local</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>W(A) = 50</td>
<td>9</td>
<td>A0 = &lt;50, 1&gt;</td>
<td></td>
</tr>
<tr>
<td>T0, TS(T1) = 10</td>
<td>Commit</td>
<td>10</td>
<td>A0 = &lt;50, 10&gt;</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>W(A) = 100</td>
<td>10</td>
<td>A1 = &lt;100, 1&gt;</td>
<td></td>
</tr>
<tr>
<td>T2, TS(T2) = 10</td>
<td>R(A) = 50</td>
<td>10</td>
<td>A0 = &lt;50, 10&gt;</td>
<td></td>
</tr>
<tr>
<td>T2, TS(T2) = 10</td>
<td>Commit</td>
<td>11</td>
<td>A1 = &lt;100, 11&gt;</td>
<td></td>
</tr>
<tr>
<td>T3, TS(T3) = 11</td>
<td>R(A) = 100</td>
<td>11</td>
<td>A1 = &lt;100, 11&gt;</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>&quot;Commit&quot;</td>
<td>11</td>
<td>Can now delete A0</td>
<td></td>
</tr>
</tbody>
</table>