DEADLOCK AND ISOLATION LEVELS

Kathleen Durant PhD
Lesson 10
CS3200
Outline for the day

• Precedence graph
• Deadlock prevention and detection
  • Waits-for graph
• My SQL Granular Locking
• Concurrency without locking
  • Optimistic Concurrency Control
  • Timestamp based concurrency control
To determine if a schedule is conflict serializable we use a precedence graph.
Transactions are vertices of the graph.
There is an edge from T1 to T2 if T1 must happen before T2 in any equivalent serial schedule.
Edge T1 -> T2 if in the schedule we have:
- T1 Read(R) followed by T2 Write(R) for the same resource R
- T1 Write(R) followed by T2 Read(R)
- T1 Write(R) followed by T2 Write(R)
- The schedule is serializable if there are no cycles.
Example 1: Precedence Graph

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

Fill in the edges
## Example 2: Precedence graph

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

**Fill in the edges**
Example 3: Precedence Graph

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
</tr>
<tr>
<td>Read(A)</td>
<td>Write(A)</td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>

Fill in the edges
Concurrency Control Techniques

• Two basic concurrency control techniques:
  • Locking
  • Timestamping
• Both are conservative approaches: delay transactions in case they conflict with other transactions.
• Optimistic methods assume conflict is rare and only check for conflicts at commit.
Locking

Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

- Most widely used approach to ensure serializability.
- Generally, a transaction must claim a *shared* (read) or *exclusive* (write) lock on a data item before read or write.
- Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.
Locking - Basic Rules

- If transaction has shared lock on item, can read but not update item.
- If transaction has exclusive lock on item, can both read and update item.
- Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
- Exclusive lock gives transaction exclusive access to that item.
Locking - Basic Rules

- Some systems allow transaction to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.
Deadlock

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁₇</th>
<th>T₁₈</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>begin_transaction</td>
<td>begin_transaction</td>
</tr>
<tr>
<td>t₂</td>
<td>write_lock(balₓ)</td>
<td>write_lock(balᵧ)</td>
</tr>
<tr>
<td>t₃</td>
<td>read(balₓ)</td>
<td>read(balᵧ)</td>
</tr>
<tr>
<td>t₄</td>
<td>balₓ = balₓ - 10</td>
<td>balᵧ = balᵧ + 100</td>
</tr>
<tr>
<td>t₅</td>
<td>write(balₓ)</td>
<td>write(balᵧ)</td>
</tr>
<tr>
<td>t₆</td>
<td>write_lock(balᵧ)</td>
<td>write_lock(balₓ)</td>
</tr>
<tr>
<td>t₇</td>
<td>WAIT</td>
<td>WAIT</td>
</tr>
<tr>
<td>t₈</td>
<td>WAIT</td>
<td>WAIT</td>
</tr>
<tr>
<td>t₉</td>
<td>WAIT</td>
<td>WAIT</td>
</tr>
<tr>
<td>t₁₀</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>t₁₁</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
Handling Deadlocks

• Three general techniques for handling deadlock:
  • Timeouts.
  • Deadlock prevention.
  • Deadlock detection and recovery.
Timeouts

- Transaction that requests lock will only wait for a system-defined period of time.
- If lock has not been granted within this period, lock request times out.
- In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.
Deadlock Detection and Recovery

• DBMS allows deadlock to occur but recognizes it and breaks it.
• Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:
  • Create a node for each transaction.
  • Create edge $T_i \rightarrow T_j$, if $T_i$ waiting to lock item locked by $T_j$.
• Deadlock exists if and only if WFG contains cycle.
• WFG is created at regular intervals.
Example - Wait-For-Graph (WFG)
Timestamping – preventing deadlocks

- Transactions ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.
- Conflict is resolved by rolling back and restarting transaction.
- No locks so no deadlock.
Deadlock Prevention

- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
- Could order transactions using transaction timestamps:
  - **Wait-Die** - only an older transaction can wait for younger one, otherwise transaction is aborted (*dies*) and restarted with same timestamp.
  - **Wound-Wait** - only a younger transaction can wait for an older one. If older transaction requests lock held by younger one, younger one is aborted (*wounded*).
Timestamping

**Timestamp**

A unique identifier created by DBMS that indicates relative starting time of a transaction.

- Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.
Timestamping (No locks)

- Read/write proceeds only if last update on that data item was carried out by an older transaction.
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.
- Also timestamps for data items (stored in the DB):
  - read-timestamp - timestamp of last transaction to read item;
  - write-timestamp - timestamp of last transaction to write item.
Timestamping - Read(x)

- Consider a transaction $T$ with timestamp $\text{ts}(T)$:

  $\text{ts}(T) < \text{write\_timestamp}(x)$
  - $x$ already updated by younger (later) transaction.
  - Transaction must be aborted and restarted with a new timestamp.

  $\text{ts}(T) < \text{read\_timestamp}(x)$
  - $x$ already read by younger transaction.
  - Roll back transaction and restart it using a later timestamp.
Timestamping - Write(x)

\[ ts(T) < \text{write\_timestamp}(x) \]

- x already written by younger transaction.
- Write can safely be ignored - *ignore obsolete write* rule.

- Otherwise, operation is accepted and executed.
Example–Basic Timestamp Ordering

<table>
<thead>
<tr>
<th>Time</th>
<th>Op</th>
<th>T₁₉</th>
<th>T₂₀</th>
<th>T₂₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td></td>
<td>begin_transaction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₂</td>
<td>read(balₓ)</td>
<td>read(balₓ)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₃</td>
<td>balₓ = balₓ + 10</td>
<td>balₓ = balₓ + 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₄</td>
<td>write(balₓ)</td>
<td>write(balₓ)</td>
<td>begin_transaction</td>
<td></td>
</tr>
<tr>
<td>t₅</td>
<td>read(balᵧ)</td>
<td></td>
<td></td>
<td>read(balᵧ)</td>
</tr>
<tr>
<td>t₆</td>
<td>balᵧ = balᵧ + 20</td>
<td>balᵧ = balᵧ + 20</td>
<td>begin_transaction</td>
<td>balᵧ = balᵧ + 20</td>
</tr>
<tr>
<td>t₇</td>
<td>read(balᵧ)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₈</td>
<td>write(balᵧ)</td>
<td></td>
<td></td>
<td>write(balᵧ)⁺</td>
</tr>
<tr>
<td>t₉</td>
<td>balᵧ = balᵧ + 30</td>
<td>balᵧ = balᵧ + 30</td>
<td></td>
<td>balᵧ = balᵧ + 30</td>
</tr>
<tr>
<td>t₁₀</td>
<td>write(balᵧ)</td>
<td></td>
<td>balᵧ = balᵧ + 30</td>
<td></td>
</tr>
<tr>
<td>t₁₁</td>
<td>bal₂ = 100</td>
<td>bal₂ = 100</td>
<td>bal₂ = 100</td>
<td>bal₂ = 100</td>
</tr>
<tr>
<td>t₁₂</td>
<td>write(bal₂)</td>
<td></td>
<td>bal₂ = 100</td>
<td></td>
</tr>
<tr>
<td>t₁₃</td>
<td>bal₂ = 50</td>
<td>bal₂ = 50</td>
<td>bal₂ = 50</td>
<td>bal₂ = 50</td>
</tr>
<tr>
<td>t₁₄</td>
<td>write(bal₂)</td>
<td>write(bal₂)⁺</td>
<td>begin_transaction</td>
<td></td>
</tr>
<tr>
<td>t₁₅</td>
<td>read(balᵧ)</td>
<td></td>
<td>read(balᵧ)</td>
<td></td>
</tr>
<tr>
<td>t₁₆</td>
<td>balᵧ = balᵧ + 20</td>
<td>balᵧ = balᵧ + 20</td>
<td>balᵧ = balᵧ + 20</td>
<td></td>
</tr>
<tr>
<td>t₁₇</td>
<td>write(balᵧ)</td>
<td></td>
<td>balᵧ = balᵧ + 20</td>
<td>write(balᵧ)</td>
</tr>
<tr>
<td>t₁₈</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

⁺ At time t₈, the write by transaction T₂₀ violates the first timestamping write rule described above and therefore is aborted and restarted at time t₁₄.

‡ At time t₁₄, the write by transaction T₁₉ can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T₂₁ at time t₁₂.
Optimistic Techniques

• Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serializability.
• At commit, check is made to determine whether conflict has occurred.
• If there is a conflict, transaction must be rolled back and restarted.
• Potentially allows greater concurrency than traditional protocols.
Performance of Locking

- Locks force transactions to wait
  - Abort and restart due to deadlock wastes the work done by the aborted transaction
  - In practice, deadlocks are rare, e.g., due to lock downgrades approach
- Waiting for locks becomes bigger problem as more transactions execute concurrently
  - Allowing more concurrent transactions initially increases throughput, but at some point leads to thrashing
  - Need to limit maximum number of concurrent transactions to prevent thrashing
  - Minimize lock contention by reducing the time a transaction holds locks and by avoiding hotspots (objects frequently accessed)
Controlling Locking Overhead

• Declaring transaction as “READ ONLY” increases concurrency
• Isolation level: trade off concurrency against exposure of transaction to other transaction’s uncommitted changes
  • Degrees of serializability

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Isolation levels

- **SERIALIZABLE**: obtains locks on (sets of) accessed objects and holds them until the end
- **REPEATABLE READ**: same locks as for serializable transaction, but does not lock sets of objects at higher level
- **READ COMMITTED**: obtains X-locks before writing and holds them until the end; obtains S-locks before reading, but releases them immediately after reading
- **READ UNCOMMITTED**: does not obtain S-locks for reading; not allowed to perform any writes
  - Does not request any locks ever
Hierarchy of Granularity

- Could represent granularity of locks in a hierarchical structure.
- Root node represents entire database, level 1s represent files, etc.
- When node is locked, all its descendants are also locked.
- DBMS should check hierarchical path before granting lock.
Lock Modes: State Intent

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Allows transactions to lock at each level but with a special protocol using new ‘intentions’ locks.
  - Can be read intent (intent share) or write intent (intent exclusive)
- Before viewing an item, transaction must set intention locks on all its ancestors (higher level containers)
- Locks are applied top-down, released bottom-up
Granularity of Data Items

- Size of data items chosen as unit of protection by concurrency control protocol.
- Ranging from coarse to fine:
  - The entire database.
  - A file.
  - A table.
  - A page (or area or database spaced).
  - A record.
  - A field value of a record.
Levels of locking

- Each transaction starts from the root of the hierarchy
- To get S or IS lock on a node, must hold IS or IX on parent node
- To get X or IX on a node, must hold IX on parent node
- Must release locks in bottom-up order
- Equivalent to directly setting locks at the leaf levels
Granularity of Data Items

• **Tradeoff:**
  • coarser, the lower the degree of concurrency;
  • finer, more locking information that is needed to be stored.
• **Best item size depends on the types of transactions.**
ISOLATION LEVEL: MYSQL

- **SET TRANSACTION** ISOLATION LEVEL *levels*;
  - SERIALIZABLE
  - REPEATABLE READ
  - READ COMMITTED
  - READ UNCOMMITTED
- Default is that the command affects the next transaction
- Can also set the ISOLATION LEVEL for the current session and globally
  - **SET [GLOBAL|SESSION] TRANSACTION** ISOLATION LEVEL *levels*;
  - **GLOBAL** applies globally for all subsequent sessions. Existing sessions are unaffected.
  - **SESSION** applies to all subsequent transactions performed within the current session
- Can also define the access method for the query
  - **SET TRANSACTION READ ONLY**
  - **SET TRANSACTION READ WRITE**
INNODB and Transactions

• All user activity occurs inside a transaction
• If autocommit mode is enabled, each SQL statement forms a single transaction on its own.
• Perform a multiple-statement transaction by starting it with an explicit START TRANSACTION
• autocommit mode is disabled within a session with SET autocommit = 0,
  • The session will have a transaction open until it is explicitly closed
  • Issue commit or rollback to close the transaction
• Default InnoDB Isolation level is REPEATABLE READ
• InnoDB performs row level locking
  • Only if two transactions try to modify the same row does one of the transactions wait for the other to complete
InnoDB and locks

- InnoDB implements standard row-level locking where there are two types of locks
  - (S) shared locks
    - permits the transaction that holds the lock to read a row.
  - (X) exclusive locks
    - permits the transaction that holds the lock to update or delete a row.
- InnoDB supports *multiple granularity locking* which permits coexistence of record locks and locks on entire tables.
  - Intention locks are table locks in InnoDB that indicate which type of lock a transaction will require later for a row in that table.
  - Intention shared (IS) Transaction T intends to set S locks on individual rows in table t. (SELECT ... LOCK IN SHARE MODE)
  - Intention exclusive (IX) Transaction T intends to set X locks on individual rows in table t (SELECT ... LOCK FOR UPDATE)
Granting locks

- A lock is granted to a requesting transaction if it is compatible with existing locks.
- A transaction waits until the conflicting existing lock is released.
- If a lock request conflicts with an existing lock and cannot be granted because it would cause deadlock, an error occurs.
- Main purpose of IX and IS locks is to show that someone is locking a row, or going to lock a row in the table.
- `SHOW ENGINE INNODB STATUS;`
  - To report on any transactions and deadlock conditions.
Summary

• Precedence graph allow us to represent transactions whose actions involve reading and writing the same data object

• Deadlocks can be assumed, prevented or detected.
  • Assumed if a transaction is waiting longer than the system time limit n – the system aborts and restarts the transaction
  • Detected via waits-for graph
  • Prevented via timestamps

• Optimistic concurrency control aims to minimize the cost of Concurrency Control
  • Best when reads are common and writes are rare