Overview
- Serializability
- Scheduling schemes
  - Locking
  - Timestamp
  - Validation
- Recoverability
- Distributed Databases and 2PC

Transaction
- A collection of database operations that should be treated as a whole
  - Atomicity (ACID)
  - commit
  - abort (rollback)
  - Isolation (ACID)

Consistent States
- Database elements
  - relations, tuples, disk pages ...
- State – a "snapshot" of all elements
- Consistent state (ACID)
  - Explicit constraints
  - Implicit constraints

Correctness Assumption
- Each transaction, executed in isolation from other transactions, brings a database from one consistent state to another consistent state.
  - consistent before and after
  - not necessarily during

Interleaving of Transactions
- Why do we want to do that??
- Notations
  - r(X,t)
  - w(X,t)
Transactions Example ...

- Consistency constraint: A = B

\[ T_1: \]
\[ r(A, t) \quad t = t + 10 \quad w(A, t) \quad r(B, t) \quad t = t + 10 \quad w(B, t) \]

\[ T_2: \]
\[ r(A, t) \quad t = 2^*t \quad w(A, t) \quad r(B, t) \quad t = t + 2 \quad w(B, t) \]

Further Abstraction

- What if T2 multiply A and B by 1 instead of 2??
- Omit local operations (a.k.a. what could go wrong will go wrong)
  - *(A)
  - *(X), *(Y)

Serial Schedules

- What is a correct schedule??
- Serial schedules
  - If any action of T proceeds any action of \( T' \), all actions of T proceed all actions of \( T' \)

Serializable Schedules

- A serializable schedule has the same effect on the database as some serial schedule

\[ \text{serial:} \]
\[ r_1(x), w_1(x), r_2(x), w_2(x), r_2(y), w_2(y) \]

??
Conflicts

Consider action $a_i(E)$ from $T_i$ and action $a_j(E')$ from $T_j$, assuming $i \neq j$

- action could be either $r$ or $w$
- $E$ and $E'$ could be same or different

- When can we interchange the order of $a_i(E), a_j(E')$ to $a_j(E'), a_i(E)$?

Conflicts Table

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$E$</th>
<th>$a_j$</th>
<th>$E'$</th>
<th>conflict?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_i$</td>
<td>$X$</td>
<td>$r_j$</td>
<td>$X$</td>
<td></td>
</tr>
<tr>
<td>$w_i$</td>
<td>$X$</td>
<td>$w_j$</td>
<td>$X$</td>
<td></td>
</tr>
<tr>
<td>$w_i$</td>
<td>$X$</td>
<td>$r_j$</td>
<td>$X$</td>
<td></td>
</tr>
<tr>
<td>$w_j$</td>
<td>$X$</td>
<td>$w_j$</td>
<td>$X$</td>
<td></td>
</tr>
</tbody>
</table>

Conflict-Serializability

- Conflict-equivalent
- Conflict-serializable schedule
  - A schedule that is conflict-equivalent to a serial schedule
  - Example:
    - $r_i(a), w_j(a), r_j(a), w_i(b), r_i(b), w_j(b), r_j(b), w_i(b)$

Conflict-serializable vs. Serializable

<table>
<thead>
<tr>
<th>Conflict-serializable schedule</th>
<th>Serializable schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>$w_j(Y), w_i(Y), w_j(X), w_i(X)$</td>
<td>$w_i(X), w_j(X)$</td>
</tr>
</tbody>
</table>

Precedence Graph for a Schedule $S$

- The nodes of the graph are transactions $T_i$
- There's an arc from node $T_i$ to node $T_j$ if $T_i$ takes precedence of $T_j$, or $T_i < s T_j$
  - There is an action $a_i$, proceed an action $a_j$ in $S$
  - $a_i$ and $a_j$ operate on the same database element
  - At least one of $a_i$ and $a_j$ is a write

Precedence Graph Examples

- $r_2(a), r_1(b), w_3(a), r_3(a), w_2(b), w_3(a), r_2(b)$
- $w_1(b)$
- $w_2(b)$
- $r_2(a), r_1(b), w_3(a), r_3(b), r_3(a), w_1(b), w_3(a)$
- $w_2(b)$
**Precedence Graph Test**
- Acyclic graph \( \Rightarrow \) conflict-serializable
  - Proof

**Overview**
- Serializability
- Scheduling schemes
  - Locking
  - Timestamp
  - Validation
- Recoverability
- Distributed Databases and 2PC

**Locking Mechanisms**
- Lock
  - request a lock on a db element: \( l(e) \)
  - release a lock on a db element: \( u(e) \)
- Transaction
  - a transaction can only access an element if it’s holding a lock on that element
  - after a transaction locks an element, it must unlock it *later.*

**Scheduling with Exclusive Locks**
- Scheduling
  - Lock table
  - No two transactions can hold the lock for the same element at the same time
  - Lock-based scheduling is *not* enough

**Two-Phase Locking (2PL)**
- In every transaction, all lock requests proceed all unlock requests
- 2PL transactions
  - \( 2PL \) Transactions + Lock-based Scheduling

\[ \text{Conflict-serializable schedule} \]

**Why 2PL Works**
- Example: \( r_i(a), w_j(b), w_k(b) \)
  - Add an action from \( T_j \) to make the schedule non-conflict-serializable??
  - Why it is not possible with 2PL??
- Proof
Shared Locks

- We need locks to reads; concurrent read should be allowed
  - Shared lock (read lock): $s_l(e), u_l(e)$
  - Exclusive lock (write lock): $x_l(e), u(e)$
- Compatibility matrix

<table>
<thead>
<tr>
<th>lock held</th>
<th>lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Y N</td>
</tr>
<tr>
<td>X</td>
<td>N N</td>
</tr>
</tbody>
</table>

Lock Upgrading

- Acquire shared lock first, then only upgrade it to exclusive lock when necessary
- Why do we want to do it??
- Why do we not want to do it??

Update Lock

- Update lock: $u_l(e)$
  - read privilege
  - can be upgraded to exclusive lock, while shared locks cannot

<table>
<thead>
<tr>
<th>lock requested</th>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td>I</td>
</tr>
</tbody>
</table>

Increment Action and Increment Lock

- Increment action: $inc_l(e)$
- Increment lock: $i_l(e)$

<table>
<thead>
<tr>
<th>lock requested</th>
<th>S</th>
<th>X</th>
<th>U</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Y</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lock-based Scheduler

- Insert locking and unlocking operations into transactions
- Accept or delay operations according to a lock table

Conceptual Lock Table

<table>
<thead>
<tr>
<th>Every possible object</th>
<th>Lock info for B</th>
<th>Lock info for C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation with a Hash Table

If object not found in hash table, it is unlocked

Lock Table Example

<table>
<thead>
<tr>
<th>Object: A</th>
<th>Group mode: U</th>
<th>Waiting: yes</th>
<th>List:</th>
</tr>
</thead>
<tbody>
<tr>
<td>tran mode wait?</td>
<td>Not T_link</td>
<td>T1</td>
<td>S</td>
</tr>
<tr>
<td>T2</td>
<td>U</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>X</td>
<td>yes</td>
<td>A</td>
</tr>
</tbody>
</table>

Lock Granularity

- fewer locks but less concurrency
  - tuple
  - page
  - relation
- more locks but better concurrency

DB Elements in a Hierarchy

- relation
- block
  - B1
  - B2
- tuple
  - t1
  - t2
  - t3
  - t4

Intension Locks

- Called *Warning Locks* in the textbook
- IS – “intend to acquire a shared lock”
- IX – “intend to acquire an exclusive lock”

Multi-Granularity Locking

- Always start at the root node and work downward
- Place an intension lock on each node along the path
- Place an lock on the target node
Compatibility Matrix with Intension Locks

<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>Y</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tree Index Locking

- Why the hierarchical locking scheme does not work for tree indexes?
- How to make tree locking more efficient
  - locking does not have to start with the root node
  - does not have to be strictly 2PL

Tree Protocol

- First lock may be any node
- Can only lock child node when there’s a lock on the parent node
- Can Unlock at any time
- Cannot re-lock a node

Tree Protocol Example

- $T_1, T_2, ..., T_{10}$
- Exclusive lock only
- Lock order
  - R: $T_1, T_2, T_3$
  - $C_2$: $T_0, T_5, T_7, T_9, T_3, T_2$
  - $C_2$: $T_6, T_8, T_9, T_{10}, T_1$
- Give a serial schedule based on the locking order

Optimistic Concurrency Control

- No locks
- Just let the transactions run ...
- ... until something bad happens, then we abort and restart the transaction

Timestamp-based Scheduler

- Each transaction is given a timestamp $TS(T)$
  - hardware clock
  - software counter
- For each db element $X$, maintain
  - $RT(X)$ – latest timestamp of read
  - $WT(X)$ – latest timestamp of write
  - $C(X)$ – latest write has committed
**Timestamp Example**

\[ r_2(x), r_1(x), r_1(y), w_1(y), c, w_2(y), c_2 \]

**Physically Unrealizable**

- Serialize transactions by timestamp
- Unserializable behavior is called physically unrealizable
  - read too late
  - write too late

**Timestamp-based Scheduling**

<table>
<thead>
<tr>
<th>( TS(T_i) &lt; RT(x) )</th>
<th>( R(x) )</th>
<th>( W(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( TS(T_i) &lt; WT(x) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Multi-version Timestamps**

- Why do we want to do it??
- How do we do it??
  - When can older versions be removed??

**Validation-based Scheduler**

- Transaction
  - Read set RS(T)
  - Write set WS(T)
- Scheduling
  - Read
  - Validate
  - Write

**Transaction Sets**

- START
  - transactions that have started but not validated
    - start(T)
- VAL
  - transactions that are validated but have to finished writing
    - start(T), val(T)
- FIN
  - transactions that have finished
    - start(T), val(T), and fin(T)
Validation Example

- Validate T
  - U in VAL
  - U is not finished
  - RS(T) \cap WS(U) \neq \emptyset, or WS(T) \cap WS(U) \neq \emptyset

Overview

- Serializability
- Scheduling schemes
  - Locking
  - Timestamp
  - Validation
- Recoverability
- Distributed Databases and 2PC

The Recoverability Problem

- Serializability problem
  - Ensure correct execution of T_1, ... , T_k when all transactions successfully commit
- Recoverability problem
  - Ensure correct execution of T_1, ... , T_k when some of the transactions abort

“Seemingly” Unrecoverable Schedule

- Is the schedule serializable??
  - conflict-serializable??
- Are the transactions 2PL??
- Is the schedule recoverable??

Unrecoverable Schedule

- Is the schedule serializable??
  - conflict-serializable??
- Are the transactions 2PL??
- Is the schedule recoverable??

Recoverable Schedule

- Recoverable schedule: each transaction commits only after each transaction from which it has read committed.
- Examples:
  - \( w_1(A), w_2(B), w_3(A), r_1(B), c_1, c_2 \)
  - \( w_2(a), w_1(B), w_3(A), r_2(B), c_1, c_2 \)
ACR Schedules

- Cascading rollback
  - \( w1(A), w1(B), w2(A), r2(B), a1 \)
- A schedule avoids cascading rollback (ACR) if transactions only read values written by committed transactions
  - a.k.a. never read “dirty” data

Scheduling Schemes That Enforce ACR

- Timestamp
- Validation
- Strict 2PL
  - 2PL
  - release any write-related lock (exclusive, update, increment ...) after \(<\text{COMMIT} T>\) or \(<\text{ABORT} T>\) is flushed to disk

Group Commit

- Relaxed Strict 2PL
  - release write-related locks after \(<\text{COMMIT T}>\) is written to memory buffer
  - ??
Overview

- Serializability
- Scheduling schemes
  - Locking
  - Timestamp
  - Validation
- Recoverability
- Distributed Databases and 2PC

Distributed Databases

- Retail chains
- Bank branches
- ...
- Replicated databases for load balancing
  - Great for queries (reads)
  - Not so great for updates (writes)

Distributed Transaction Example

Issues in Distributed Transaction

- Commit/abort
- Serializability
  - distributed locking and timestamp
- Recovery
  - node failure
  - network failure
- Atomicity
- ...

2 Phase Commit (2PC)

- Local log
- Coordinator

2PC – Phase One

- Coordinator
  - log <Prepare T>
  - send message [prepare T]
- Other
  - Commit
    - enter a precommitted state
    - <Ready T>, [Ready T]
  - Abort
    - <Don't commit T>, [Don't commit T]
    - Abort
2PC – Phase Two

- Coordinator
  - All [Ready T]
    - <Commit T>
    - [Commit T]
  - At least one [Don’t commit T]
    - <Abort T>
    - [Abort T]

- Other
  - [Commit T] → <Commit T>
  - [Abort T] → <Abort T>

2PC Recovery When “Other” Fails

- When last log record is:
  - <Start T>
  - <Commit T>
  - <Abort T>
  - <Ready T>
  - <Don’t Commit T>

2PC Recovery When Coordinator Fails

- New coordinator
  - At least one site has <Commit T>
  - At least one site has <Abort T>
  - At least one site has <Don’t commit T>
  - All surviving sites have <Ready T>
  - All surviving sites have <Start T>