ACID Properties of DB Transaction

- Atomicity
- Consistency
- Isolation
- Durability

Need for Concurrent Execution

- Fully utilize system resources to maximize performance
- Enhance user experience by improving responsiveness

Problem of Concurrent Transactions ...

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>milk</td>
<td>2.99</td>
</tr>
<tr>
<td>2</td>
<td>beer</td>
<td>6.99</td>
</tr>
</tbody>
</table>

Transaction #1:

- `MIN` select min(price) from items;
- `MAX` select max(price) from items;

... Problem of Concurrent Transactions

Transaction #2:

- `DELETE` delete from items;
- `INSERT` insert into items values (3, 'water', 0.99);

Concurrency Control

- Ensure the correct execution of concurrent transactions

Consider the interleaving of T1 and T2:

MIN, DELETE, INSERT, MAX
**Transaction**

```sql
start transaction;
select balance from accounts
where id=1;
update accounts
set balance=balance-100
where id=1;
update accounts
set balance=balance+100
where id=2;
commit;
```

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

---

**Schedule**

- A schedule is the interleaving of the transactions as executed by the DBMS
- **Example:**

  **Two transactions**

  \(T_1: r_1(x), w_1(x), r_1(y), w_1(y)\)

  \(T_2: r_2(y), w_2(y), w_1(x)\)

  **One possible schedule:**

  \(r_1(x), w_1(x), r_2(y), w_1(y), r_1(y), w_1(y), w_2(x)\)

---

**Serial Schedule**

- A serial schedule is a schedule in which the transactions are not interleaved

---

**Serializable Schedule**

- A serializable schedule is a schedule that produces the same result as some serial schedule
- A schedule is **correct** if and only if it is serializable

---

**Example: Serializable Schedules**

- Are the following schedules serializable??

  \(r_1(x), w_1(x), r_1(y), w_1(y), r_1(y), w_2(x)\)

  \(r_2(x), w_1(x), r_2(y), r_1(y), w_2(y), w_2(x)\)

  \(r_1(x), w_1(x), r_2(y), r_1(y), w_2(y), w_2(x)\)

  \(r_1(x), w_1(x), r_1(y), w_1(y), r_1(y), w_2(y), w_2(x)\)

---

**Conflict Operations**

- Two operations **conflict** if the order in which they are executed can produce different results
  - Write-write conflict, e.g. \(w_1(x)\) and \(w_2(x)\)
  - Read-write conflict, e.g. \(r_1(y)\) and \(w_2(y)\)
Precedence Graph of Schedule $S$

- The nodes of the graph are transactions $T_i$.
- There is an arc from node $T_i$ to node $T_j$ if there are two conflicting actions $a_i$ and $a_j$, and $a_i$ proceeds $a_j$ in $S$.

Example: Precedence Graph

![Example Graph](image)

Determine Serializability

- A schedule is serializable if its precedence graph is acyclic.

Scheduling

![Scheduling Diagram](image)

Locking

- Produce serializable schedules using **locks**.
- Lock
  - `lock()` — returns immediately if the lock is available or is already owned by the current thread/process; otherwise wait.
  - `unlock()` — release the lock, i.e., make the lock available again.

Basic Locking Scheme

- A transaction must acquire a lock on some data before performing any operation on it.
  - E.g. $l_1(x), r_1(x), u_l_1(x), l_2(x), w_1(x), u_2(x)$
- Problem: concurrent reads are not allowed.
Shared Locks and Exclusive Locks

- Multiple transactions can each hold a shared lock on the same data.
- If a transaction holds an exclusive lock on some data, no other transaction can hold any kind of lock on the same data.

Example:
\[ s_l, r, x, w, y, u_l, x, u_l, y, s_l, r, y \]

Two-Phase Locking Protocol (2PL)

- A shared lock must be acquired before reading.
- An exclusive lock must be acquired before writing.
- In each transaction, all lock requests proceed all unlock requests.

Example: 2PL

- Why the following schedule is not possible under 2PL??
\[ s_l, r, x, u_l, x, w, x, l, y, w, y, u_l, x, u_l, y, x_l, y, w_1, y, u_l, y \]

2PL Schedules

The Recoverability Problem

- Serializability problem:
  - Ensure correct execution of \( T_1, \ldots, T_k \) when all transactions successfully commit.

- Recoverability problem:
  - Ensure correct execution of \( T_1, \ldots, T_k \) when some of the transactions abort.
Example: Unrecoverable Schedule
◆ Is the following schedule serializable??
◆ Is the following schedule 2PL??

\[ w_1(x), r_2(x), w_2(x), c_2, a_1 \]

Recoverable Schedule
◆ In a recoverable schedule, each transaction commits only after each transaction from which it has read committed

Serializable and Recoverable (I)

Serializability and Recoverable (II)

ACR Schedules
◆ Cascading rollback
  • \( w_1(x), w_2(y), w_2(x), r_2(y), a_1 \)
◆ A schedule avoids cascading rollback (ACR) if transactions only read values written by committed transactions

Strict 2PL
◆ 2PL
◆ A transaction releases all write-related locks (e.g. exclusive locks) after the transaction is completed
  • After <COMMIT,T> or <ABORT,T> is flushed to disk
  • After <COMMIT,T> or <ABORT,T> is created in memory (would this work??)
Example: Strict 2PL

Why the following schedule is not possible under Strict 2PL??

\[ w_1(x), r_2(x), w_2(x), c_3, c_1 \]

Deadlock

\( T_1: w_1(x), w_1(y) \)
\( T_2: w_2(x), w_2(y) \)

\[ x_1(x), w_1(x), x_2(y), w_2(y), \ldots \]

Necessary Conditions for Deadlock

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Handling Deadlocks

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection

Resource Numbering

- Impose a total ordering of all shared resources
- A process can only request locks in increasing order
- "Why the deadlock example shown before can no longer happen??"
About Resource Numbering
- A deadlock prevention strategy
- Not suitable for databases

Wait-Die
- Suppose $T_1$ requests a lock that conflicts with a lock held by $T_2$
  - If $T_1$ is older than $T_2$, then $T_1$ waits for the lock
  - If $T_1$ is newer than $T_2$, $T_1$ aborts (i.e. "dies")
- Why does this strategy work??

About Wait-Die
- A deadlock avoidance strategy (not deadlock detection as the textbook says)
- Transactions may be aborted to avoid deadlocks

Wait-For Graph
- Each transaction is a node in the graph
- An edge from $T_1$ to $T_2$ if $T_1$ is waiting for a lock that $T_2$ holds
- A cycle in the graph indicates a deadlock situation

About Wait-for Graph
- A deadlock detection strategy
- Transactions can be aborted to break a cycle in the graph
- Difficult to implement in databases because transaction also wait for buffers
  - For example, assume there are only two buffer pages
    - $T_1$: $x_i(y)$, $pin(b_i)$
    - $T_2$: $pin(b_i)$, $pin(b_j)$, $x_j(x)$

Problem of Phantoms
- We can regulate the access of existing resources with locks, but how about new resources (e.g. created by appending new file blocks or inserting new records)??
Handle Phantoms

- Lock “end of file/table”

Multiversion Locking

- Each version of a block is time-stamped with the commit time of the transaction that wrote it.
- When a read-only transaction requests a value from a block, it reads from the block that was most recently committed at the time when this transaction began.

How Multiversion Locking Works

1. \( T_1: w_i(b_1), w_i(b_2) \)
2. \( T_2: w_j(b_1), w_j(b_2) \)
3. \( T_3: r_j(b_1), r_j(b_2) \)
4. \( T_4: w_k(b_2) \)

\( w_i(b_1), w_i(b_2), c_1, w_j(b_1), r_j(b_2), w_j(b_2), c_p, r_j(b_2), c_3, w_2(b_1), c_2 \)

- Which version of \( b_1 \) and \( b_2 \) does \( T_3 \) read?

About Multiversion Locking

- Read-only transactions do not need to obtain any lock, i.e. never wait.
- Implementation: use log to revert the current version of a block to a previous version.

SQL Isolation Levels

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Lock Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serializable</td>
<td>Slocks held to completion; no slock on eof</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>Slocks held to completion; no slock on eof</td>
</tr>
<tr>
<td>Read committed</td>
<td>Slocks released early; no slock on eof</td>
</tr>
<tr>
<td>Read uncommitted</td>
<td>No slock</td>
</tr>
</tbody>
</table>
Concurency Control in SimpleDB

- Transactions
  - `simpledb.tx`
- Concurrency Manager
  - `simpledb.tx.concurrency`

SimpleDB Transaction

- Keep track of the buffers it uses in `BufferList`
- Block-level locking
  - Acquire lock before reading
  - Acquire lock before writing
  - Dummy block for EOF

Transaction Commit

- Flush buffers and log records
- Release all locks
- Unpin all buffers

Concurrency Manager

- Each transaction has its own concurrency manager
- Concurrency manager keeps tracks of the locks held by the transaction
- A `lock table` is shared by all concurrency managers

Lock Table

- Keeps lock in a Map
  - Key: block
  - Value: -1 (lock), 0 (no lock), >0 (slock)
- `Lock()` and `unlock()` are synchronized methods so only one transaction can modify the lock map at a time
- Transaction aborts if it waits for a lock for too long, i.e. avoid deadlock

Life Cycle of a Java Thread

- `start`
- `yield`
- `run method terminates`
- `Dead`
- `Not Runnable`
- `Running (Runnable)`
Wait() and Notify()

- Methods of the Object class
- `wait()` and `wait(long timeout)`
  - Thread becomes *not runnable*
  - Thread is placed in the *wait set* of the object
- `notify()` and `notifyAll()`
  - Awake one or all threads in the wait set, i.e. make them *runnable* again

Readings

- Textbook Chapter 14.4-14.6
- SimpleDB source code
  - simpledb.tx
  - simpledb.tx.concurrency