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);

Consider the interleaving of T1 and T2:

MIN, DELETE, INSERT, MAX

Concurrency Control

- Ensure the correct execution of concurrent transactions
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;
```

$r_1(x), r_1(y), w_1(x), r_1(y), w_1(y)$

Schedule

- A schedule is the interleaving of the transactions as executed by the DBMS
- Example:
  - Two transactions
    - $T_1: r_1(x), w_1(x), r_1(y), w_1(y)$
    - $T_2: r_2(y), w_2(y), r_2(x)$
  - One possible schedule:
    - $r_1(x), w_1(x), r_1(y), w_1(y), r_1(y), w_1(y)$

Serial Schedule

- A serial schedule is a schedule in which the transactions are not interleaved
- Example:
  - $r_1(x), w_1(x), r_1(y), w_1(y), r_2(y), w_2(y), w_2(x)$
  - And
  - $r_2(y), w_2(y), w_2(x), r_1(x), w_1(x), r_1(y), w_1(y)$

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??
  - $r_1(x), w_1(x), r_2(y), w_2(y), r_1(y), w_1(y), w_2(x)$
  - $r_1(x), w_1(x), r_2(y), r_2(y), w_2(y), w_1(y), w_2(x)$
  - $r_1(x), w_1(x), r_1(y), w_1(y), r_1(y), w_1(y), w_2(x)$

How do we check if two schedules produce the same results?

Conflicting 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 (or write-read) 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 $a_i$ precedes $a_j$ in $S$.

Example: Precedence Graph

Determine Serializability

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

Scheduling

Synchronization Using Locks

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.

Lock lock;
lock.lock();
// execute
// some bn
...
lock.unlock();
Thread 1

lock.lock();
// execute
// another bn
...
lock.unlock();
Thread 2
Simple Lock Implementation in Java

```java
public class Lock {
    private long value = -1;
    public void lock() {
        long threadId = Thread.currentThread().getId();
        while (value != -1) wait(5000);
        lock = threadId;
    }
    public void unlock() { value = -1; }
}
```

Is there anything wrong with this implementation??

Life Cycle of a Java Thread

Basic Locking Scheme

- 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

Basic Locking Scheme

- A transaction must acquire a lock on some data before performing any operation on it
  - E.g. \( l_1(x), r_1(x), u_1(x), l_2(x), w_2(x), u_2(x) \)
- 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:
  \[ s_1(x), r_1(x), x_1(y), w_1(y), s_2(x), r_2(x), u_1(y), s_3(y), r_3(y) \]

About Locking and Schedule

- \( T_1: r_1(x), w_2(x) \)
- \( T_2: r_3(x), w_3(x) \)

Invalid Schedule:

\[ s_1(x), r_1(x), s_2(x), r_2(x), x_1(y), w_1(y), \ldots \]

Lock upgrade:

\[ s_1(x), r_1(x), s_2(x), r_2(x), u_2(x), x_1(y), w_1(y), \ldots \]
Example: Releasing Locks Too Early

- Is the following schedule serializable??
  \[ s_l(x), r_l(x), u_l(x), x_l(x), w_s(x), x_l(y), w_s(y), u_l(x), u_l(y), x_l(y), w_s(y), u_l(y) \]

Two-Phase Locking Protocol (2PL)

- A shared lock must be acquired before reading
- A 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??
  \[ s_l(x), r_l(x), u_l(x), x_l(x), w_s(x), x_l(y), w_s(y), u_l(x), u_l(y), x_l(y), w_s(y), u_l(y) \]

2PL Schedules

- Show a schedule that is 2PL but not serial
- Show a schedule that is serializable but not 2PL

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??

  \[ w_s(x), r_s(x), w_s(x) \]
... Example: Unrecoverable Schedule

- But what if T2 commits but T1 aborts? $w_1(x), r_2(x), w_2(x), c_2, a_1$

Recoverable Schedule

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

Serializable and Recoverable (I)

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

Serializable and Recoverable (II)

- Strict 2PL
  - 2PL
    - A transaction releases all write-related locks (i.e. 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??

\[ w_1(x), r_2(x), w_2(x), c_2, c_1 \]

Serializable and Recoverable (III)

Other Lock Related Issues

- Phantoms
- Lock granularity
- Lock and SQL Isolations Levels

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”

Lock Granularity

- Fewer locks but less concurrency
- Record block table
- More locks but better concurrency
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; 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>

xlocks are always held to completion

Alternative Locking Scheme – 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

- \( T_1: w_1(b_1), w_1(b_2) \)
- \( T_2: w_2(b_1), w_2(b_2) \)
- \( T_3: r_3(b_1), r_3(b_2) \)
- \( T_4: w_4(b_2) \)

- \( w_1(b_1), w_1(b_2), c_1, w_2(b_1), r_3(b_1), w_4(b_2), c_4, r_3(b_2), c_3, w_2(b_2), c_2 \)

- 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.

Deadlock

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

- \( x_1(x), w_1(x), x_2(y), w_2(y), ... \)

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
  - $T_1: x_1(x), \text{pin}(b_1)$
  - $T_2: \text{pin}(b_2), \text{pin}(b_3), x_3(x)$

Readings

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