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>items</th>
<th>id</th>
<th>name</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>milk</td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<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;
```

r₁(x), r₁(x), w₁(x), r₁(y), w₁(y)

Schedule

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

  Two transactions:
  T₁: r₁(x), w₁(x), r₁(y), w₁(y)
  T₂: r₂(y), w₂(y), w₂(x)

  One possible schedule:
  r₁(x), w₁(x), r₁(y), w₁(y), r₂(y), w₁(y), w₂(x)

Serial Schedule

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

  r₁(x), w₁(x), r₁(y), w₁(y), r₁(y), w₁(y), w₂(x)

  and

  r₂(y), w₂(y), r₂(x), r₁(x), w₁(x), r₁(y), w₁(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??

  r₁(x), w₁(x), r₁(y), w₁(y), r₁(y), w₁(y), w₂(x)

  r₁(x), w₁(x), r₁(y), r₁(y), w₁(y), w₁(y), w₂(x)

  r₁(x), w₁(x), r₁(y), w₁(y), r₁(y), w₁(y), w₂(x)

Conflicting Operations

- Two operations conflict if the order in which they are executed can produce different results.

  - Write-write conflict, e.g. w₁(x) and w₂(x)

  - Read-write (or write-read) conflict, e.g. r₁(y) and w₂(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 $S$

Determine Serializability

- A schedule is serializable if its precedence graph is acyclic

Scheduling

- 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

Synchronization Using Locks

- Lock lock;
- lock.lock();
- // execute
- // some txn
- ....
- lock.unlock();

Thread 1

- lock.lock();
- lock.lock();
- // execute
- // another txn
- ....
- lock.unlock();
- lock.unlock();
Simple Lock Implementation in Java

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

Is there anything wrong with this implementation??

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

Basic Locking Scheme

- A transaction must acquire a lock on some data before performing any operation on it
  - E.g. `l1(x), r1(x), ul1(x), sl1(x), l2(x), l1(y), w2(x), ul2(x), ul2(y), x1(y), w1(y), ul1(y), sl2(y), r2(y)`
- 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:
  - `sl1(x), r1(x), sl1(y), w1(y), sl2(x), r2(x), ul1(y), sl2(y), r2(y)`

Example: Releasing Locks Too Early

- Is the following schedule serializable??
  ```
  sl1(x), r1(x), ul1(x), sl1(x), w2(x), l2(x), l1(y), w2(y), ul2(x), ul2(y), x1(y), w1(y), ul1(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??
  \[ s(x), r(x), u(x), x(x), w(x), x(y), w(y), u(x), u(y), x(y), w(y), u(y) \]

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??

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

... Example: Unrecoverable Schedule

- But what if T2 commits but T1 aborts?

\[ w_1(x), r_2(x), w_2(x), c, a_1 \]
In a recoverable schedule, each transaction commits only after each transaction from which it has read committed.

A schedule avoids cascading rollback (ACR) if transactions only read values written by committed transactions.

A transaction releases all write-related locks (i.e. exclusive locks) after the transaction is completed.

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
- 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>hold to completion;</td>
</tr>
<tr>
<td></td>
<td>lock on eof</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>hold to completion;</td>
</tr>
<tr>
<td></td>
<td>no lock on eof</td>
</tr>
<tr>
<td>Read committed</td>
<td>released early;</td>
</tr>
<tr>
<td></td>
<td>no lock on eof</td>
</tr>
<tr>
<td>Read uncommitted</td>
<td>No lock</td>
</tr>
</tbody>
</table>
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

- \( T_1: w_1(b_1), w_2(b_2) \)
- \( T_2: w_3(b_1), w_4(b_3) \)
- \( T_3: r_1(b_1), r_3(b_3) \)
- \( T_4: w_1(b_2) \)

\( \{ w_1(b_1), w_2(b_2), c_1, w_3(b_1), r_1(b_1), w_4(b_2), c_4, r_3(b_3), c_5, w_2(b_3), c_2 \} \)

- 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

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

\( \{ x_1(x), w_1(x), x_1(y), w_2(y), \ldots \} \)

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
    - $T_1$: $x_1(x)$, $pin(b_1)$
    - $T_2$: $pin(b_1)$, $pin(b_2)$, $x_2(x)$
Readings

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