Managing Concurrency in Web Applications

Concurrent Access to Web Applications
- A fundamental property of Web sites: Concurrent accesses by multiple users
- Concurrent accesses intersect at:
  - Database accesses (read / write)
  - Execution of application code (Servlets / JSP)
- Users activate the same application objects:
  - Database
    - Shared connection / connection pool
  - Servlet / JSP object
    - For each Servlet / JSP, only one object is created

Concurrency Hazards
- What are the potential problems?
- Inconsistency due to concurrent accesses
  - For example, generating IDs, verifying uniqueness of names, presenting information while being updated
- What are possible solutions?
  - Transactions: to consider a sequence of operations as an atomic action
    - Until commitment, updates are partial (inconsistent)
    - On failure, the origin state is restored (rollback)
  - How do we disable the server from relying on partial information or such that may later be cancelled due to rollback?

Handling Concurrency
- For managing concurrency on the level of Java code, use known Java tools
  - Synchronization, wait/notify, etc.
- For handling concurrency in database accesses, transactions are used
- Concurrent transactions use distinct connections
- But connections (and their generation) are expensive!
  - Use thread pools
  - Share connections whenever possible (e.g., simple reads)

Different Applications Have Different Requirements
- Consider the following three examples:
  - **britanica.com**: many users view it but only a few users change it
  - **wikipedia.com**: many users view it and many users change it
  - A page showing access trends (statistics) for a popular web site—many users change the data but only a few view it
- How do we guarantee correctness and efficiency in each scenario?

Thumbnail Rules
- Some concurrency problems can be solved at both the code and database levels (what about client-side programming?)
- In principle, prefer letting the database handle concurrency
  - Database researchers and vendors have put a lot of effort on handling concurrent transactions
- But you need to understand the exact model of transaction management that is implemented by your specific database engine
  - We will examine transaction management in MySQL
ACID

**Atomicity:** transactions are atomic and indivisible
**Consistency:** operations transform the database from one valid state to another valid state
**Isolation:** transactions do not affect each other while they are running
**Durability:** the changes, to the database, of a transaction that has committed are permanent

---

**Transaction Management via JDBC**

**Transaction:** A sequence of statements that must all succeed (or all fail) together
- e.g., updating several tables due to customer purchase
- Failure: System must reverse all previous actions
- Cannot leave DB in inconsistent state halfway through a transaction
- COMMIT = complete transaction
- ROLLBACK = cancel all actions

---

**Example**

Suppose that we want to transfer money from bank account 13 to account 72:

```java
PreparedStatement pstmt = con.prepareStatement("update BankAccount set amount = amount + ? where accountId = ?");
pstmt.setInt(1, -100);
pstmt.setInt(2, 13);
pstmt.executeUpdate();
pstmt.setInt(1, 100);
pstmt.setInt(2, 72);
pstmt.executeUpdate();
```

What happens if this update fails?

---

**Transaction Life Cycle**

- Through JDBC, transactions are *not* opened and closed explicitly
- A transaction *starts* on 1st (successful) command
  - After a connection is established
  - After the previous transaction ends
- A transaction *ends* when COMMIT or ROLLBACK are applied
  - Either explicitly or implicitly (next slides)
- Multiple transactions are obtained by using multiple connections to the database
Committing a Transaction

How do we commit?

- Explicitly invoking `Connection.commit()`
- Implicitly
  - After every query execution, if `AutoCommit` is true
  - When the user normally disconnects (i.e., appropriately closes the connection)
  - In some DBs: After invoking a DDL command (CREATE, DROP, RENAME, ALTER, ...)

Automatic Commitment

- A `Connection` object has a boolean `AutoCommit`
- If `AutoCommit` is `true` (default), then every statement is automatically committed
- If `AutoCommit` is `false`, then each statement is added to an ongoing transaction
- Change using `setAutoCommit(boolean)`
- If `AutoCommit` is `false`, need to explicitly commit or rollback the transaction using `Connection.commit()` and `Connection.rollback()`

Rollback

- Rolling Back: Undoing any change to data within the current transaction
- The ROLLBACK command explicitly rolls back (and ends) the current transaction
- ROLLBACK is implicitly applied when the user abnormally disconnects (i.e., without appropriately closing the connection)

Improved Example

```java
con.setAutoCommit(false);
try {
    PreparedStatement pstmt =
        con.prepareStatement("update BankAccount
                        set amount = amount + ?
                        where accountId = ?");
    pstmt.setInt(1, -100); pstmt.setInt(2, 13);
    pstmt.executeUpdate();
    pstmt.setInt(1, 100); pstmt.setInt(2, 72);
    pstmt.executeUpdate();
    con.commit();
} catch (SQLException e) { con.rollback(); ...; }
```

Storage Engines

- In MySQL, different storage engines can be used
  - MyISAM
    - Efficient
    - Has built-in full-text search
    - Does not support transactions and uses table-level locking
  - InnoDB
    - Supports transactions (ACID compliant)
    - Other advanced features (e.g., referential integrity)
MyISAM Indexes

- PRIMARY KEY INDEX

- .MYD
- .MYI

MyISAM Indexes

- PRIMARY KEY INDEX

- .MYD
- .MYI

MyISAM Indexes

- PRIMARY KEY INDEX

- .MYD
- .MYI

InnoDB Indexes

- PRIMARY KEY INDEX

- PKV = Primary Key Value

MyISAM vs. InnoDB

- MyISAM uses index file (.MYI) and data file (.MYD)
  - All indexes are "created equal"
  - Lookup using KEY vs. PRIMARY KEY has same efficiency
  - Secondary keys contain pointers to the data file

- InnoDB uses PRIMARY KEY "clustered index"
  - The data rows are stored in the index with the PRIMARY KEY
  - Secondary keys contain PRIMARY KEY values (PKVs) to reference the row

- InnoDB can be faster, for simple PK lookups!
- MyISAM is usually faster for sequential accesses, updates, index lookups
Setting the Storage Engine

CREATE TABLE t (i INT) ENGINE = INNODB;

SET storage_engine=MYISAM;

ALTER TABLE t ENGINE = MYISAM;

InnoDB Model

- InnoDB combines multi versioning with 2-phase locking (2PL) to achieve atomicity and isolation
- The goal: serializability – transactions are executed in parallel, however, their effect, on the database and one on the other, is as if they were executed one after the other

Schedule

- A schedule is a list of actions (read/write tuples) of different transactions
- A serial schedule is a schedule where for each two transactions, all the actions of one transaction appear before all the actions of the other transaction (non-interleaved)

View Equivalence

- Two schedules S1 and S2 are equivalent if the following hold:
  - If in one schedule a transaction T1 reads an initial value for an object X, then T1 reads the same value for X in the other schedule
  - If in one schedule the transaction T1 reads for an object X the value written by the transaction T2, then in the other schedule T1 reads for X the value written by T2
  - If in one schedule the transaction T1 is the final transaction to write the value for an object X, then it is the final transaction to write the value for X in the other transaction T1

View Serializable

- A schedule is view serializable if it is equivalent to a serial schedule
- Deciding whether a schedule is view serializable is hard (the time complexity of the test is NP-hard)
- Instead, we apply a simpler test of detecting conflicting operations
Conflicting Operations

- Two schedules may not be view equivalent because of conflicting actions:
  - The actions are being done by two different transactions
  - They are done on the same tuple
  - At least one of the actions is a write operation
  - In other words,
    - Two different transactions $T_1$ and $T_2$ write the same tuple $t$
    - A transaction $T_1$ write a tuple $t$ and a different transaction $T_2$ reads the same tuple $t$, or vice versa

Conflict Serializable

- Two schedules are conflict equivalent if for every two conflicting actions, their order in one schedule is equal to their order in the other schedule
- A schedule is conflict serializable if it is conflict equivalent to a serial schedule

If a schedule is conflict serializable then it is view serializable

Locks

- MySQL controls concurrent access to data by deploying locks
- An operation on the database (select, update, insert, create, drop, etc.) may require some lock to be obtained by the running transaction
  - If a lock is not available, the transaction is blocked
- Locks are applied to either whole tables or to (sets of) specific rows

Lock Types

- MyISAM – Table-level locks
- InnoDB – Row-level locks
- Explicitly locking tables is possible

READ Locks – Shared
WRITE Locks – Exclusive

Two Phase Locking (2PL)

- Each transaction has two phases:
  1. Acquiring locks
  2. Releasing locks
- A schedule in which all the transactions satisfy 2PL is conflict serializable (and hence, also view serializable)
Multi Versioning
- Each transaction has a "snapshot" of the database
  - A time stamp is assigned to transactions and to objects
  - When reading an object, the transaction reads the latest version among the versions that were written by transactions that precede it
  - When writing, a transaction cannot write to an object if the object has already been read by a later transaction (i.e., by a transaction with a greater time stamp)

Transaction Isolation

Isolation Issues
- How do different transactions interact?
- Does a running transaction see uncommitted changes?
- Does it see committed changes?
- Can two transactions update the same row?

Isolation Levels
- The isolation level determines the capabilities of a transaction to read/write data that is accessed by other transactions
  - In MySQL, there are four levels of isolation:
    1. READ UNCOMMITTED
    2. READ COMMITTED
    3. REPEATABLE READ (default)
    4. SERIALIZABLE
- Each transaction determines its isolation level
  - Connection.setTransactionIsolation(int level)

READ COMMITED & SERIALIZABLE
- **SERIALIZABLE**: Every reading of tuples puts shared locks on the scanned tuples and on the gaps between them (locking index entries)
- **REPEATABLE READ**: Each transaction gets to work in an isolated version of the table, where each row remains as it was when the transaction started. During the whole transaction, statements read only the changes that were committed by the time the transaction begun (and the changes made by the transaction itself)

READ COMMITED & SERIALIZABLE
- **READ COMMITED**: A statement reads the data that was committed by the time the statement (not the transaction) begun
- **READ UNCOMMITTED**: It is possible for the transaction to read changes that other transactions have made before the changes have been committed (practically, no isolation for read operations)
Some Definitions

- **Dirty reads**: A transaction reads data that is written by another, uncommitted transaction
- **Non-repeatable reads**: A transaction rereads data it previously read and finds that a committed transaction has modified or deleted that data
- **Phantom reads**: A transaction re-executes a query returning a set of rows satisfying a search condition and finds that a committed transaction inserted additional rows that satisfy the condition

### Comparison

<table>
<thead>
<tr>
<th></th>
<th>READ UNCOMMITTED</th>
<th>READ COMMITED</th>
<th>REPEATABLE READ</th>
<th>SERIALIZABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty Reads</td>
<td>Possible</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Impossible</td>
</tr>
<tr>
<td>Non-repeatable Reads</td>
<td>Possible</td>
<td>Possible</td>
<td>Impossible</td>
<td>Impossible</td>
</tr>
<tr>
<td>Phantom Reads</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Impossible</td>
</tr>
</tbody>
</table>

### What Happens Here (1)?

1. `CREATE TABLE pairs (x INTEGER, y INTEGER);`
2. `select * from pairs`
3. `insert into pairs values(1,1)`
4. `select * from pairs`
5. `insert into pairs values(1,2)`
6. `select * from pairs`
7. `COMMIT`
8. `select * from pairs`
9. `COMMIT`

### What Happens Here (2)?

1. `CREATE TABLE pairs (x INTEGER, y INTEGER);`
2. `select * from pairs`
3. `insert into pairs values(1,1)`
4. `COMMIT`
5. `select * from pairs`
6. `select * from pairs`
7. `insert into pairs values(1,2)`
8. `COMMIT`
9. `select * from pairs`
10. `COMMIT`
What Happens Here (3)?

1. CREATE TABLE pairs (x INTEGER, y INTEGER);
2. insert into pairs values(1,1)
3. select * from pairs
4. COMMIT
5. select * from pairs
6. select * from pairs
7. insert into pairs values(1,2)
8. COMMIT
9. select * from pairs
10. COMMIT

T1: SERIALizable

Is it equivalent to any truly serial execution of the transactions?

What Happens Here (4)?

1. CREATE TABLE pairs (x INTEGER, y INTEGER);
2. insert into pairs values(1,1)
3. select * from pairs
4. COMMIT
5. select * from pairs
6. select * from pairs
7. insert into pairs values(1,2)
8. COMMIT
9. select * from pairs
10. COMMIT

T1: REPEATABLE READ

Only after the commit, T2 sees (1,1)

What Happens Here (5)?

1. CREATE TABLE pairs (x INTEGER, y INTEGER);
2. insert into pairs values(1,1)
3. select * from pairs
4. COMMIT
5. select * from pairs
6. select * from pairs
7. insert into pairs values(1,2)
8. COMMIT
9. select * from pairs
10. COMMIT

T1: READ COMMITTED

After the commit, T2 sees (1,1)

Table-Level Locks

Locking a Table

- To lock the table mytbl, use the command
  `LOCK TABLES tbl-name AS alias WRITE`
- Acquires the write (exclusive) lock
  - Thus, no transaction can read or update the table
  - This lock is also needed in ALTER TABLE & DROP TABLE
  - This operation blocks until all locks on the table are released

Table Locking in MySQL

- WRITE locks are exclusive
- READ locks are shared
- Writes (table updates) are given higher priority than reads (table retrievals) in order to avoid starvation
Comparison

Advantages of row-level locking:
- Fewer conflicts between transactions
- Smaller chances for rollbacks
- Possible to lock a single row for a long time

Disadvantages of row-level locking:
- Requires more memory than table-level locks
- Slower when used on a large part of the table because it must acquire many more locks
  e.g., usually slower for GROUP BY operations and scans of the entire table

Lock Types

- There are four types of locks
  - X – exclusive lock (for write actions)
  - S – shared lock (for read actions)
  - IX – an intension to acquire an exclusive lock
  - IS – an intension to acquire a shared lock

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

Write Locks in Updates

- In all four isolation levels, an exclusive (write) lock should be acquired in the following cases:
  - On a row that should be deleted, prior to the deletion
  - On an inserted row, immediately after the insertion
  - On an updated row, prior to the update

Updates by Concurrent Transactions

Concurrent Updates

- MySQL prevents updating a row that is being updated by an uncommitted transaction
  - Locks are held until the transaction ends
  - The second updating transaction is blocked until the lock is released (first one commits or rolls back)

What Happens Here (6)?

1. CREATE TABLE pairs (x INTEGER, y INTEGER);
2. insert into pairs values(1,1)
3. COMMIT
4. update pairs set y=2 where x=1
5. update pairs set y=3 where x=1
6. select * from pairs
7. COMMIT
8. select * from pairs
9. COMMIT
What Happens Here (7)?
1. CREATE TABLE pairs (x INTEGER, y INTEGER);
2. insert into pairs values(1,1)
3. COMMIT
4. update pairs set y=2
   where x=1
5. update pairs set y=3
   where x=1
   “a blind write”
6. select * from pairs
7. COMMIT
T.1: SERIALIZABLE
T.2: SERIALIZABLE

What Happens Here (8)?
1. CREATE TABLE pairs (x INTEGER, y INTEGER);
2. insert into pairs values(1,1)
3. COMMIT
4. select * from pairs
5. update pairs set y=3
   where x=1
6. update pairs set y=2
   where x=1
7. COMMIT
T.1: SERIALIZABLE
T.2: SERIALIZABLE

Deadlocks
- A deadlock prevention mechanism
  - Checks each operation and allows only operations that do not cause a deadlock (looks for cycles in the dependency graph)
  - In case of a deadlock, terminates the “smallest” transaction (heuristically decides which one is the smallest)
  - Prevents starvation (livelock) by terminating transactions that wait for a lock for more than some predefined waiting time

Links