What are Transactions?

- So far, we looked at individual queries; in practice, a task consists of a sequence of actions.
- E.g., “Transfer $1000 from account A to account B”
  - Subtract $1000 from account A
  - Subtract transfer fee from account A
  - Credit $1000 to account B
- A transaction is the DBMS’s view of a user program:
  - Must be interpreted as “unit of work”: either entire transaction executes, or no part of it executes/has any effect on DBMS.
  - Two special final actions: COMMIT or ABORT.

Concurrent Execution

- DBMS receives large numbers of concurrent requests.
  - Concurrent (or parallel) execution improves performance.
  - Two transactions are concurrent if they overlap in time.
  - Disk accesses are frequent, and relatively slow; CPU can do a lot of work while waiting for the disk, or even SSD.
  - Goal is to increase/maximize system throughput.
- Concurrency control
  - Protocols that ensure things execute correctly in parallel.
  - Broad and difficult challenge that goes beyond DBMS realm.
  - OS, Distributed Programming, hardware scheduling (CPU registers), etc.
  - Our focus is DBMS, but some principles span beyond DBMS.

Web app Transactions

- Each application action turns into a database transaction.
- A well-designed app has a “service API” describing those actions.
- A request execution calls the service API one or more times.
- Each service call represents an application action and contains a transaction.
- Thus transactions are contained in request-response cycles.
- This ensures that transactions are short-lived, good for performance.
- But they still can run concurrently under high-enough load.

ACID Properties

Transaction Management must fulfill four requirements:

1. **Atomicity**: either all actions within a transaction are carried out, or none is.
   - Only actions of committed transactions must be visible.
2. **Consistency**: concurrent execution must leave DBMS in a consistent state.
3. **Isolation**: each transaction is protected from effects of other concurrent transactions.
   - Net effect is that of some sequential execution.
4. **Durability**: once a transaction commits, DBMS changes will persist.
   - Conversely, if a transaction aborts/is aborted, there are no effects.

Major Example: the web app

- Concurrent web requests from users
- App server(s)
- Multi-threaded Object layer
- JDBC
- Database server
- Other apps
- Web layer

The diagram shows a multi-threaded object layer receiving concurrent web requests from users, which are handled by the app server(s) using JDBC to interact with the database server. Other apps may also be connected to the database.
Roles of Transaction Manager

- **Concurrency Control**
  - Ensuring correct execution in the presence of multiple transactions running in parallel

- **Crash recovery**
  - Ensure that atomicity is preserved if the system crashes while one or more transactions are still incomplete
  - Main idea is to keep a log of operations; every action is logged before execution (Write-Ahead Log or WAL)
  - Details are covered in CS634

Transactions

The user/programmer can group a sequence of commands so that they are executed atomically and in a serializable fashion:

- **Transaction commit**: all the operations should be done and recorded.
- **Transaction abort**: none of the operations should be done.

Compare to file storage: abort commonly leaves broken files

Modeling Transactions

- User programs may carry out many operations …
  - Data-related computations
  - Prompting user for input, handling web requests
- … but the DBMS is only concerned about what data is read/written from/to the database
- A **transaction** is abstracted by a sequence of time-ordered read and write actions
  - e.g., R(X), R(Y), W(X), W(Y)
  - R=read, W=write, data element in parentheses
  - Each individual action is indivisible, or atomic
  - SQL UPDATE = R(X) W(X)

Important dataflow assumptions

- Transactions interact with one another as they run only via database read and write operations.
- No messages exchanged between running transactions
- No use of shared memory between running transactions
- Oracle, other DBs, enforce this
- Transactions may accept information from the environment when they start and return information to the environment when they finish by committing.
  - The agent that starts a transaction will come to know whether it committed or aborted, and can act on that information, plus any results (data) returned from the transaction.
  - Thus it is possible for data to go from one transaction to the environment and then to another starting transaction, but note that these transactions are not concurrent.

Scheduling Transactions

- **Serial schedule**: no interleaving of transactions
  - Safe, but poor performance!
- **Schedule equivalence**: two schedules are equivalent if they lead to the same state of the DBMS (see footnote on pg. 525 that includes values returned to user in relevant “state”)
- **Serializable schedule**: schedule that is equivalent to some serial execution of transactions
  - But still allows interleaving/concurrency!

Serializable schedule example

<table>
<thead>
<tr>
<th>Schedule</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td>A=A+100,</td>
<td>B=B-100</td>
</tr>
<tr>
<td>T2:</td>
<td>A=1.06*A,</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

- Same effect as executing T1 completely, then T2
If execution is not serializable...

- Non-serializable concurrent executions can show anomalies, i.e., clearly bad behavior
- Let's look at some examples

Concurrency: lost update anomaly

- Consider two transactions (in a really bad DB) where $A = 100$

<table>
<thead>
<tr>
<th>T1</th>
<th>A = A + 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>A = A + 100</td>
</tr>
</tbody>
</table>

- $T_1$ & $T_2$ are concurrent, running same transaction program
- $T_1$ & $T_2$ both read old value, 100, add 100, store 200
- One of the updates has been lost!
- **Consistency requirement:** after execution, $A$ should reflect all deposits (Money should not be created or destroyed)
- No guarantee that $T_1$ will execute before $T_2$ or vice-versa…
- … but the net effect must be equivalent to these two transactions running one-after-the-other in some order

Lost Update with flight reservations

Customer 1 - finds a seat empty
Customer 2 - finds the same seat empty
Customer 1 - reserves the seat.
Customer 2 - reserves the seat.

Customer 1 will not be happy.

Dirty Reads

- **Reading Uncommitted Data (Dirty Reads)**

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A), W(A), R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

- Value of $A$ written by $T_1$ is read by $T_2$ before $T_1$ completed all its changes
- But $T_1$ could abort…

Other anomalies due to interleaved execution

- **Unrepeatable Read:** a transaction reads $A$, then does other things, then reads $A$ again and finds another value.
- **Phantom:** One transaction inserts a row, another transaction sees it, but then the first transaction deletes the row or aborts, making the data invalid.
- Or a transaction deletes a row, another transaction sees the smaller set of rows, then the first transaction aborts.
- Neither of these are allowed to happen with serializable transactions.

Why is serializability important?

- If each transaction preserves consistency, every serializable schedule preserves consistency
- For example, transactions that move money around should always preserve the total amount of money.
- If running with serializable transactions, we only need to check that each transaction program has this property, and we know that the system does.
- **How to ensure serializable schedules?**
  - Use **locking** protocols (ensuring conflict serializability)
  - DBMS inserts proper locking actions, user is oblivious to locking (except through its effect on performance, and deadlocks)
  - Covered in cs634
Transaction Support in SQL

- A transaction is automatically started when user executes a statement or accesses the catalogs.
- Transaction is either committed (COMMIT) or aborted (ROLLBACK).
- New in SQL-99: SAVEPOINT feature
  - SAVEPOINT <savepoint name>
  - Actions ...
  - ROLLBACK TO SAVEPOINT <savepoint name>
- SAVEPOINT advantage vs. sequence of transactions
  - Can roll back over multiple savepoints (only the target one remains)
  - Lower overhead: no new transaction initiated (book, pg. 536)
  - But transaction initiation is not an expensive action. Locks are still held on changes done before savepoint, when rollback to savepoint done. Locks would be released if a real commit is done.

Setting Transaction Properties in SQL

- Access Mode
  - READ ONLY vs READ WRITE
- Isolation Level (decreasing level of concurrency)

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Possible</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Isolation Levels in Practice

- Databases default to RC, read-committed, so many apps run that way, can have their read-accessed data changed, and phantoms
- Web apps (JEE, anyway) have a hard time overriding RC, so most are running at RC

Read Committed (RC) Isolation

- 2PL can be modified for RC: take long-term write locks but not long term read locks
- Reads are atomic as operations, but that’s it
- Lost updates can happen in RC: system takes locks only for the write operations:
  - R(A)R(A)W(B)C(B)W(B)C(B)W(B)C(A) (RC isolation)
- Update statements are atomic, so that case of read-then-write is safe even at RC
- Update T set A = A + 100 (safe at RC isolation)
- Remember to use update when possible!

Importance of Short Transactions

- Transactions hold locks in the database, preventing other transactions from finishing.
- The slowest activity involved in program execution is human response time, i.e. UI (user interaction) time.
- Thus there is a simple rule: no UI inside a transaction.
- Example:
  - Show user the product, its price (one transaction)
  - Accept order
  - Run a new transaction to record the sale, rollback if the object was bought meanwhile by someone else

Syntax for SQL

- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE
In Java, from jdbc/Transfer.java

```java
try {
    in = new Scanner(System.in);
    conn = getConnected(in);
    conn.setAutoCommit(false);
    conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
    doCustomerTransfers(conn, in);
    System.out.println("Finished, exiting");
} catch (SQLException e) { … }
```

Transfer Failure cases

- **Cases:**
  - **Deadlock:** transfers from A to B, B to A concurrently yields "Deadly embrace"
  - In Oracle at serializable isolation: "Can’t serialize access for this operation" due to Oracle’s concurrency control system
  - In both these cases, a retry may work, so the best code tries the transfer method several times, then gives up
  - See doTransfer method of Transfer.java, with a loop of retries calling the transfer method.