For the most part of the course we've focused on understanding how a SQL query gets executed. We have focused mostly on read-only queries, assuming that data was sitting, static, in the database.

Now we turn to write queries as well. Most applications have both read/write queries. Consider a bank, online or traditional commerce, social networks. Pretty much any application that has some state that must be maintained.

Also most queries both read and write the database:

```
UPDATE balance SET balance = balance + 1 WHERE balance > 10;
```

```
UPDATE inventory SET inventory = inventory - 1 WHERE product_id = 33;
```

How many of these queries can we run concurrently in the database?

Why would we want to run them concurrently?

We want to run them concurrently because that's the only way to get performance out of the database when we have applications with multiple concurrent users.

Think of the alternative of 'serializing' (put in a queue) all requests.

When running RW queries concurrently we may run into 'conflicts' over tuples that are affected by these queries. Depending on the order (interleaving) of the actions we may end up with wrong results and leave the DB in a bad state.

<table>
<thead>
<tr>
<th>Query 1</th>
<th>Query 2</th>
<th>Query 1</th>
<th>Query 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
</tbody>
</table>

- It is very hard and inconvenient for users to think about concurrency.
- Transactions permit packing a set of logical actions (e.g. a logic query) into a unit so that users of the system do not need to worry about conflicts and so that the system knows how to execute multiple units concurrently.
- A transaction is atomic and recoverable. It has 4 properties:
  - Atomicity: Either all operations in the transaction succeed, or none does.
  - Consistency: Each transaction leaves the database in a consistent state.
  - Isolation: Concurrently running transactions cannot see each other's partial results.
  - Durability: Transaction effects should persist even after crashes.
- ACID properties of transactions.
Transactions from DB view:

- Sequence of R, W actions followed by COMMIT or ABORT
  - COMMIT: Complete successfully
  - ABORT: Terminate and undo all the actions carried out.

- A schedule is a set of actions from a set of transactions, and the order in which two actions from T1 appear in T2 is the same order in the schedule. A schedule represents an actual or potential execution sequence.

- Example:
  - There are multiple possible schedules for any set of transactions, but only some are 'correct'. We focus on one type of schedules: Serializable schedules. Serializability means that although transactions are interleaved, the end result is as though those concurrent actions had run in some serial order.

<table>
<thead>
<tr>
<th>Serial order</th>
<th>Serializable</th>
<th>Non Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
<td>W(B)</td>
</tr>
</tbody>
</table>

- Now let's consider the problem ('conflicts') that may arise without using serializable schedules. There's a conflict when more than 1 op. on the same object and 1 is a write.

  - WR Conflict: Reading uncommitted data. T1 W(A), T2 R(A) before T1 commits. That's a 'dirty read'.

  - RW Conflict: Unrepeateable reads, T1 R(A), T2 W(A). T1 has not committed yet. If T1 tries to read A again, it will see a different result.

  - WW Conflict: Overwriting Uncommitted data. T1 W(A), T2 W(A), T1 has not committed yet.

- To avoid all the previous conflicts we need to make sure the DB only executes serializable schedules. So in the remaining of the class we address 2 key questions:
  - How do we know if a schedule is serializable?
  - How do we implement tx in the DB to enforce a serializable execution?

  > Concurrency control mechanisms.
There are 2 methods to verify a schedule is serializable:

- View Serializability. Useful to understand serializability, but not practical.
- Conflict Serializability.

**Conflict Serializability:**

1. Find conflicting actions. For 2 ops, swapping their order leads to different results.
2. Two schedules are conflict equivalent if they involve the same actions of the same txs and they order every pair of conflicting actions (rw, wr, ww) of 2 committed txs in the same way.
3. Two schedules are conflict equivalent means they have same effect in the database.
4. The outcome of a schedule depends only on the order of conflicting operations. Exchanging order of non-conflicting ops should not affect the final DB state.

A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

To determine if a schedule is "conflict serializable":

- Build a precedence graph with edges from $T_i \rightarrow T_j$ if:
  - $T_i$ reads/writes $A$ before $T_j$ writes $A$
  - $T_i$ writes some $A$ before $T_j$ reads $A$

**Examples:**

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(A)$</td>
<td>$R(B)$</td>
</tr>
<tr>
<td>$R(A)$</td>
<td>$R(B)$</td>
</tr>
<tr>
<td>$R(B)$</td>
<td>$W(B)$</td>
</tr>
<tr>
<td>$W(A)$</td>
<td>$W(B)$</td>
</tr>
<tr>
<td>$W(B)$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(A)$</td>
<td>$R(B)$</td>
<td>$W(A)$</td>
</tr>
<tr>
<td>$R(B)$</td>
<td>$W(B)$</td>
<td>$W(A)$</td>
</tr>
<tr>
<td>$W(B)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A schedule is "conflict serializable" if the precedence graph is cycle-free.

While testing for view serializability is NP complete, conflict serializability is not and in addition, it can be enforced as txs run. This is what concurrency control mechanisms enforce. At least most of them.

So far, we have only dealt with the issues that arise from concurrent execution of transactions. We have not talked about what happens if a tx aborts.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(A)$</td>
<td>$W(A)$</td>
<td>$R(B)$</td>
</tr>
<tr>
<td>$R(B)$</td>
<td>$W(B)$</td>
<td>$R(A)$</td>
</tr>
<tr>
<td>$W(B)$</td>
<td>$W(A)$</td>
<td>$W(B)$</td>
</tr>
<tr>
<td>commit</td>
<td>abort</td>
<td>commit</td>
</tr>
<tr>
<td>abort</td>
<td>abort</td>
<td>abort</td>
</tr>
</tbody>
</table>

+ This schedule is not recoverable. $T_2$ is going to commit based on uncommitted data.
+ Possible to deal with it by cascading the rollbacks. However, if either $T_2$ or $T_3$ would have committed this would be irrecoverable.
- A ‘cascadeless’ schedule is one in which, if T2 reads something written by T1, T1 has committed before T2 performs that read.
- We enforce serializable and cascadeless schedules by using concurrency control mechanisms.
- How do we implement concurrency control?

**Locking:**

- The idea is that a tx acquires a (shared) read lock before reading an item, and acquires an (exclusive) write lock before writing an item.
  - Only ‘shared’ locks can be shared.

**Two-Phase Locking:**

- A tx cannot acquire locks after it has released the first.
  - ‘Growing’ phase: acquiring locks.
  - ‘Shrinking’ phase: releasing locks.

- The end of the ‘growing phase’ is called lock point. Serial order between conflicting txs is determined by lock points.
- In the example, what if T1 aborts after releasing lock?
  - 2PL is not cascadeless.

**Strict Two-Phase Locking:**

- Don’t release exclusive (write) locks until the tx commits. Some people call these ‘long duration locks’.
- Another version ‘rigorous two-phase locking’ does not release any lock before commit. This allows to order transactions by commit points.
- A simple implementation protocol for the last is:
  - Acquire shared lock before read.
  - Acquire exclusive lock (or upgrade) before write.
  - Release all locks after commit.

In practice, we use rigorous two-phase commit because we don’t know when all locks have been acquired.
Locking brings a few additional challenges one has to address in practice:
- Deadlocks
- Performance: Lock granularity.

**Deadlocks:**

- Consider this schedule:

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>Lock Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock(A)</td>
<td>R(A)</td>
<td>Grant(A)</td>
<td></td>
</tr>
<tr>
<td>Lock(B)</td>
<td>R(B)</td>
<td>Grant(B)</td>
<td></td>
</tr>
<tr>
<td>Lock(C)</td>
<td>R(C)</td>
<td>Grant(C)</td>
<td></td>
</tr>
<tr>
<td>Lock(D)</td>
<td>R(D)</td>
<td>Grant(D)</td>
<td></td>
</tr>
<tr>
<td>Lock(E)</td>
<td>R(E)</td>
<td>Grant(E)</td>
<td></td>
</tr>
</tbody>
</table>

- Deadlock avoidance → require locks to be acquired in order. Conservative Two Phase locking
  - This reduces concurrency.
- Deadlock detection → looking for cycles in a waits-for graph.
  - When a deadlock is detected, it is resolved by aborting 1 tx.

- Waits-for graph. Nodes are currently running txs. There's an edge from Ti to Tj if Tj is waiting for Ti to release a lock.

- When cycle is detected: Which transaction to abort? Different criteria:
  - The one with the fewest locks.
  - The one which did the least work.
  - The one which is farthest to complete.

**Lock Granularity:**

- We've been abstractly discussing locking on objects, without defining what those objects are.
- Lo could be tables, pages, tuples.
- Each granularity has a tradeoff between concurrency and overhead.
- Hierarchical locking: To lock all objects under current element.
- Lo lock on page locks all tuples in that page.
- Lock escalation technique: Lock fine-grained. If many such locks are requested, then grant the lock on the container (higher level in hierarchy).

- Patterns of lock request are not known until runtime. Why?
- This helps achieve a good tradeoff at runtime between concurrency/overhead.

A few final notes:

+ Consider T1 doing something like:
  
  UPDATE status SET status = 'retired' if age > 65;
+ T2 concurrently inserts a new row with status = 'default' and age = 104. Then T2 commits before T1.
- This is called 'phantom problem'.
  
  A tx retrieves a collection of tuples twice and sees different results.
  
  If new items are added to the DB, conflict serializability does not guarantee serializability.

Summary:

- Concurrent R/W query execution
- Transaction and ACID properties
- Transaction schedules
- Serializable schedules
- Conflict Serializability, conflict equivalent schedules
- Unrecoverable and cascadless schedules
- Concurrency Control Mechanisms to enforce schedule execution

- Locking
  - Two-Phase Locking
  - Strict Two-Phase Locking
  - Rigorous Two-Phase Locking
- Deadlocks
- Lock Granularity
- Phantom Problem
Lecture 13: Optimistic Concurrency Control and Snapshot Isolation

- Locking is pessimistic in that it acquires locks (it pays the associated cost) on objects that do not actually conflict.
  \( \Rightarrow \) In other words, without enforcing a serializable schedule we may end up with one transaction waiting on another.

- Optimistic Concurrency Control tries to avoid this 'issue' with locking by checking if two transactions conflict at the very end of their execution.

- OCC never has to wait for locks. (+)
- OCC does not suffer from deadlocks. (+)
- Conflicting transactions have to be restarted (-)
- Transactions can 'starve', e.g., repeatedly restarted (-)

- Intuitively OCC is a win when:
  \[ P(\text{conflict}) \times \text{tx-restart-cost} \leq \text{avg (locking delay per query)} \]
  \( \Rightarrow \) Since tx-restart-cost is more or less fixed, the decision boils down to \( P(\text{conflict}) \), the probability of two transactions conflicting.

How does OCC work?

- Transactions have 3 phases: Read, Validate, Write.

**Read Phase:**
- Transactions execute, all updates affect a local copy of the data. Not the DB directly.
- Keep track of read and write sets for each transaction, i.e., objects read and written.
- Why does it use local copies?
  \( \Rightarrow \) To make dirty results visible to other transactions.
  \( \Rightarrow \) To be able to undo.

**Validate Phase:**
- Verify that the transaction being validated did not conflict with other already committed transactions.
  \( \Rightarrow \) Only overlapping transactions.
- To validate the transaction, we must assign an order to the transactions, e.g., \( T_1, T_2, T_3, \ldots \) and then we check for conflicts with each of those transactions.

**Write Phase:**
- Copy local copy to DB, making changes visible to other transactions.
- Ordering is chosen based on the time at which the read phase finishes.
  - In practice, monotonically increasing IDs are used.

- What are the checks we need to perform to validate $T_j$?
  1. Make sure $T_j$ reads all writes (collecting) of every earlier tx.
  2. Make sure $T_j$ completed all of its writes after the writes of every earlier tx.

- That translates into requiring at least one of the following conditions to be true.

1. All $T_i < T_j$ completed writes before $T_j$ started read phase (No overlap)
   
   
   $T_i$ \hspace{1cm} read, validate, write \hspace{1cm} read, validate, write
   \hspace{1cm} $T_j$

2. $T_i$'s write set does not intersect $T_j$'s read/write set. $T_i$ completes read phase before $T_j$ completes its read phase, $T_i$ completes write phase before $T_j$ starts its write phase
   
   $T_i$ \hspace{1cm} read \hspace{1cm} validate \hspace{1cm} write
   \hspace{1cm} $T_j$

3. $T_i$'s write set does not intersect $T_j$'s read/write set. $T_i$ completes read phase before $T_j$ completes its read phase.
   
   $T_i$ \hspace{1cm} read \hspace{1cm} validate \hspace{1cm} write
   \hspace{1cm} $T_j$

- If none of these 3 conditions is true, abort tx.

**WRITE PHASE:**

- Copy local copy to the database, making the updates visible to other txs.

**Note on Validation:**

+ Serial Validation. Make the phase a critical section and validate txs one by one.
  1. Note only conditions 1 and 2 must be checked, not 3
  2. If read phase finished first then also validate and write phase!

- Critical section is large (-)
- Only 1 tx can write at a time (-)
To address the 1st problem we check as many txs as possible before entering the critical section.

To address the 2nd problem → parallel validation.
   - Check, validate/ write concurrently.
   - More opportunities for txs to abort. E.g., 2 txs that wrote same value and entered validation now need to abort. With serial validation only one.

What about read-only txs?
   - No need for critical section.
   - Still need to check if read set intersects write set of other tx and abort if so.

Why is OCC interesting today?

→ Locking approaches require a centralized shared structure, the locking table.
→ That may become a bottleneck in in-mem DBs.
   - Accessing the lock table, that is.
→ In OCC read and write sets are local to the tx thread.
→ Multicore aggravates the shared object access overhead.

As usual, no clear winner in every case. Analyze the situation, state your assumptions.