LECTURE 12: TRANSACTIONS AND LOCKING

- 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 > 1M;

  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 those queries. Depending on the order (interleaving) of the actions we may end up with wrong results and leave the DB in a bad state.

Query 1	Query 2
R(A)	
	R(A)
W(A)	
	W(A)

Query 1	Query 2
R(A)	
W(A)	
	R(A)
	W(A)

- Remove item from inventory, only if |inventory| > 0.
- Queries are users buying items
- R ≡ read, W ≡ write. A ≡ some logic object in DB.

- It is very hard and inconvenient for users / developers 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 succesfully
  - ↳ ABORT: Terminate and undo all the actions carried out.

- A schedule is a ~~set~~ list of actions from a set of transactions, and the order in which two actions from T_i appear in T_i 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.

Serial order

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
commit	R(A)
	W(A)
	R(B)
	W(B)
	commit

Serializable

T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
	R(B)
commit	
	W(B)
	commit

• Executing txs in different orders may lead to different results. All presumed to be correct.

Non Serializable

T1	T2	
R(A)		
	R(A)	
	W(A)	→ lost update
W(A)		
R(B)		
W(B)		
	R(B)	→ this is visibl
	W(B)	

- Now let's consider the problems ('conflicts') that may arise without using serializable schedules. There's a conflict when there are 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. Unrepeatable reads. T1 R(A), T2 W(A), T1 has not commit 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:

~~Find conflicting actions. [For 2 ops, swapping their order leads to different results]~~
~~↳ Based on RW, WR, WW.~~

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.

1. Two schedules are conflict equivalent means they have same effect in the database.

3. 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.

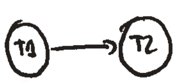
hm. A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

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

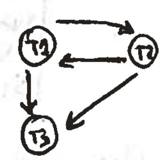
Examples:

T1	T2
R(A)	
W(A)	
R(B)	R(A)
W(B)	W(A)
	R(B)
	W(B)



T1	T2
R(A)	
	R(A)
	R(B)
	W(B)
R(C)	
W(C)	

T1	T2	T3
R(A)		
	W(A)	
W(A)		
		W(A)



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.

T1	T2
R(A)	
W(A)	
	R(A)
	commit
R(B)	
abort	

+ This schedule is not 'recoverable'. T2 is going to commit based on uncommi- data.

T1	T2	T3
R(A)		
W(B)		
W(A)		
	R(A)	
	W(A)	
		R(A)
abort		

+ Possible to deal with it by cascading the rollbacks. However if either T2 or T3 would have committed this would be unrecoverable!

- A 'cascadeless' schedule is one in which, if T2 reads something written by T1, T1 has commited before T2 performs that read.

• We enforce serializable and cascadeless schedules by using concurrency control mechs.

• 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.

T1	T2
Slock(A)	
R(A)	
Xlock(A)	
W(A)	
□	
Xlock(B)	
release(A)	
	Slock(A)
	R(A)
	Xlock(A)
	W(A)
	Slock(B) □

Can we release lock(A) here?
↳ No. If T1 releases A here, T2 could W(A) and W(B) before T1. That would result in a non-serial schedule.

What happens here?
↳ T2 waits for T1.

• The end of the 'growing phase' is called lock point. Serial order between conflicting txs is determined by lock points.

T1	T2
Acq A	
R(A)	
W(A)	
Rel(A)	

T1	T2	T1 lock	T l
R(A)		SA	
W(A)		XA	
W(B)		XB	
	R(A)	Rel A	S
	W(A)		X
R(B)			
W(B)			
	R(B)	Rel B	S
	W(B)		X

S ≡ Shared
X ≡ Exclusi
Rel ≡ releases

• In the ~~[illegible]~~ 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.

In practice, we use rigorous two-phase commit because we don't know when all locks have been acquired.

• 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.

Locking brings a few additional challenges one has to address in practice:
- Deadlocks
- Performance. Lock granularity.

Deadlocks:

- Consider this schedule:

T1	T2	Lock Manager
Slock(A)		Grant(A)
R(A)		
	Slock(B)	Grant(B)
	R(B)	
Xlock(B)		Wait(T2)
	Xlock(A)	Wait(T1)

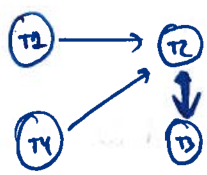
} Deadlock!

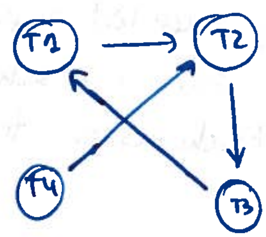
- There are two general strategies we could implement to deal with deadlocks:
  - Deadlock avoidance / ~~prevention~~.
  - Deadlock detection.
- In practice ~~they~~ deadlocks are rare, so typically the second option is implemented.

- Deadlock avoidance → require locks to be acquired in order. Conservative Two Phase Locking
  - ↳ all locks acquired at the beginning of the tx.
  - ↳ 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 T_i to T_j if T_i is waiting for T_j to release a lock.

T1	T2	T3	T4
Slock(A)			
R(A)			
	Xlock(B)		
	W(B)		
Slock(B)			
		Slock(C)	
		R(C)	
	Xlock(C)		
			Xlock(B)
		Xlock(A)	





Cycle indicates deadlock.

- 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.

} At the same time try to avoid aborting always the same tx.

Lock Granularity:

- We've been abstractly discussing locking on objects, without defining what those objects are.
  - ↳ could be tables, pages, tuples.
  - ↳ each granularity has a tradeoff between concurrency and overhead.
- Hierarchical locking: To lock all objects under current element.
  - ↳ 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: → Note that even if this tx is read-only the same problem occurs.

  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.
  - ↳ In other words, without enforcing a serializable schedule we may end up with one!
- Optimistic Concurrency Control tries to avoid this 'issue' with locking by checking if 2 txs conflicted at the very end of their execution.
- OCC never has to wait for locks. (+)
- OCC does not suffer from deadlocks. (+)
- Conflicting txs have to be restarted (−)
- Txs can 'starve', e.g., repeatedly restarted (−)
- Intuitively OCC is a win when:

$$P(\text{conflict}) \times \text{tx\_restart\_cost} < \text{avg}(\text{locking delay per query})$$

↳ Since tx_restart_cost is more or less fixed, the decision boils down to P(conflict), the probability of 2 txs conflicting.

How does OCC work?

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

Read Phase:

- Txs 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 write
- Why does it use local copies?
  - ↳ To not make dirty results visible to other transactions.
  - ↳ To be able to undo.

Validate Phase:

- Verify that the tx being validated did not conflict with other already committed txs.
  - ↳ only overlapping txs.
- To validate the tx 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 txs.

Write Phase:

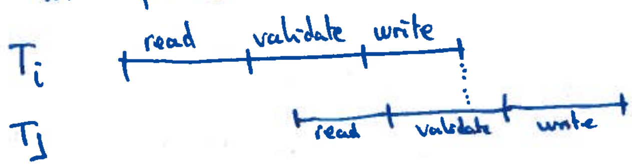
- Copy local copy to DB, making changes visible to other txs.

- 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?
  - ↳ Make sure T_j read all writes (conflicting) of every earlier tx.
  - ↳ 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.

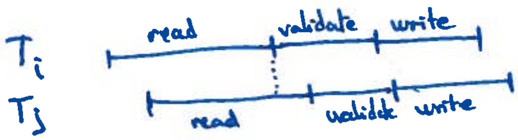
1.- All $T_i < T_j$ completed write phase before T_j started read phase (No overlap)

2.- T_i's write set does not intersect T_j's read set. ~~And 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 is complete and on-disk before T_j is.
* T_j reading something while T_i writes, so it sees some but not all T_i's updates.

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.



Only potential concern is T_j wrote something T_i read. Cannot be a conflict, however, if T_i finishes read phase before T_j starts writing.

→ T_i finishes read phase before T_j because of the way we assigned ids.

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

WRITE PHASE:

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

More on Validation:

+ Serial Validation. Make the phase a critical section and validate txs one by one.
  - ↳ Note only conditions 1 and 2 must be checked, not 3
    - ↳ 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 critical section
- To address the 2nd problem → parallel validation.
  - ↳ Check txs in 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 ~~ﾠ~~ shared object access overhead.

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