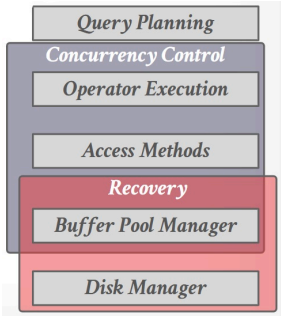
Lecture #17: Concurrency Control Theory

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo



1 Motivation

- **Lost Update Problem (Concurrency Control):** How do we handle two or more transactions trying to update the same data at the same time?
- **Durability Problem (Recovery):** How can we ensure the correct state in case of a power failure?

2 Transactions

A *transaction* is the execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher level function. They are the basic unit of change in a DBMS. Partial transactions are not allowed (i.e. transactions must be atomic).

Example: Pay $25 from Andy's bank account with balance $100 to a concert promoter.

1. **Read** Andy's account balance.
2. **Check** if balance $>$ $25.
3. **Deduct** $25 from his account.
4. **Write** Andy's new balance $75 back to the database.

Either all of the steps need to be completed or none of them should be completed.

The Strawman System

A simple system for handling transactions is to execute one transaction at a time using a single worker (i.e. serial order). Thus, only one transaction can be running at a time in the DBMS. before a transaction starts, the DBMS copies the entire database to a new file and makes all changes to that file.

- If the transaction succeeds, the new file becomes the current database file.
- If the transaction fails, the DBMS discards the dirty copy and none of the transaction's changes would have been saved.

This method (also known as shadow paging) is slow as it does not allow for concurrent transactions and throws away any parallelism, while also requires copying the whole database file for every transaction.

A (potentially) better approach is to allow concurrent execution of independent transactions while also maintaining correctness and fairness (as in all transactions are treated with equal priority and don't get "starved" by never being executed). Taking advantage of all the additional parallelism available in modern hardware (multiple cores, multiple disks, etc.) can significantly improve throughput and latency of transactions. But executing concurrent transactions in a DBMS is challenging. It is difficult to ensure correctness (for example, if Andy only has $100 and tries to pay off two promoters at once, who should get paid?) while also executing transactions quickly (our strawman example guarantees sequential correctness, but at the cost of parallelism).

Arbitrary interleaving of operations can lead to:

- **Temporary Inconsistency:** Unavoidable, but not an issue.
- **Permanent Inconsistency:** Unacceptable, cause problems with correctness and integrity of data.

The DBMS is only concerned about what data is read/written from/to the database. Changes to the outside world are beyond the scope of the DBMS. For example, if a transaction causes an email to be sent, this cannot be rolled back by the DBMS if the transaction is aborted. We want formal correctness criteria to determine whether an interleaving is valid.

3 Definitions

Formally, a *database* can be represented as a fixed set of named data objects ($A, B, C, \ldots$). These objects can be attributes, tuples, pages, tables, or even databases. The algorithms that we will discuss work on any type of object but all objects must be of the same type.

A *transaction* is a sequence of read and write operations (e.g., $R(A), W(B)$) on those objects, which is DBMS's abstract view of a user program.

The boundaries of transactions are defined by the client. In SQL, a new transaction starts with the `BEGIN` command. Then the transaction stops with either `COMMIT` or `ABORT`. For `COMMIT`, either all of the transaction's modifications are saved to the database, or the DBMS overrides this and aborts instead. For `ABORT`, all of the transaction's changes are undone so that it is like the transaction never happened. Aborts can be either self-inflicted or caused by the DBMS. ROLLBACK

The criteria used to ensure the correctness of a database is given by the acronym **ACID**.

- **A**tomicity: Atomicity ensures that either all actions in the transaction happen, or none happen.
- **C**onsistency: If each transaction is consistent and the database is consistent at the beginning of the transaction, then the database is guaranteed to be consistent when the transaction completes. Data is consistent if it satisfies all validation rules such as constraints, cascades and triggers.
- **I**solation: Isolation means that when a transaction executes, it should have the illusion that it is isolated from other transactions. Isolation ensures that concurrent execution of transactions should have the same resulting database state as a sequential execution of the transactions.
- **D**urability: If a transaction commits, then its effects on the database should persist no matter what happens (e.g. power failure, OS crash).

4 ACID: Atomicity

The DBMS guarantees that transactions are **atomic**. From application's point of view: the transaction either executes all its actions or none of them. Two possible outcomes of executing a transaction:

- **Commit:** Commit after compelting all its actions
- **Abort:** Abort (or be aborted by the DBMS) after executing some actions

There are two approaches to ensure atomicity:

Approach #1: Logging ← Most Common (runtime performance ✓ recovery performance ✗)
DBMS logs all actions in an ordered ledger so that it can undo the actions in case of an aborted transaction. It maintains undo records both in memory and on disk. The logs will be replayed after crash to put database back in correct state. Logging is used by almost all modern systems for audit and efficiency reasons.

Approach #2: Shadow Paging ← Don't do this ! (runtime performance ✗ recovery performance ✓)
The DBMS makes copies of pages modified by the transactions and transactions make changes to those copies. Only when the transaction succesfully commits is the page made visible to other transactions.

This approach is typically slower at runtime than a logging-based DBMS. However, one benefit is, if you are only single threaded, there is no need for logging, so there are less writes to disk when transactions modify the database. This also makes recovery instant and simple, as all you need to do is delete all pages from uncommitted transactions. In general, though, better runtime performance is preferred over better recovery performance, so this is rarely used in practice.

5 ACID: Consistency

At a high level, consistency means the "world" represented by the database is **logically** correct. SQL has methods to specify integrity constraints (e.g., key definitions, `CHECK`, `ADD CONSTRAINT`) and the DBMS will enforce them. Then it is up to the application to define these constraints. The DBMS is responsible for ensuring that all integrity constraints are true before and after the transaction ends.

Eventual Consistency: A commited transaction may see inconsistent results (e.g. may not see the updates of an older commited transaction immediately). Then it becomes difficult for developers to reason about such semantics, so the trend is to move away from eventual consistency and towards stronger consistency models.

***Lec#23** - Distributed DBMSs*

6 ACID: Isolation

The DBMS provides transactions the illusion that they are running alone in the system. They do not see the effects of concurrent transactions. This is equivalent to a system where transactions are executed in serial order (i.e., one at a time). But to achieve better performance, the DBMS has to interleave the operations of concurrent transactions while maintaining the illusion of isolation.

Concurrency Control

A *concurrency control protocol* is how the DBMS decides the proper interleaving of operations from multiple transactions at runtime.

There are two categories of concurrency control protocols:

1. **Pessimistic:** The DBMS assumes that transactions will conflict, so it doesn't let problems arise in the first place.
2. **Optimistic:** The DBMS assumes that conflicts between transactions are rare, so it chooses to deal with conflicts after they happen.

Example: assuming A and B both start with $ 1000

There are many possible outcomes of running T_1 and T_2 concurrently. But the ground rule is $A+B$ should be $2000 * 1.06 = 2120$. There is no guarantee on the order of transaction execution, but the outcome of the database must be equivalent to some serial execution of the transactions.

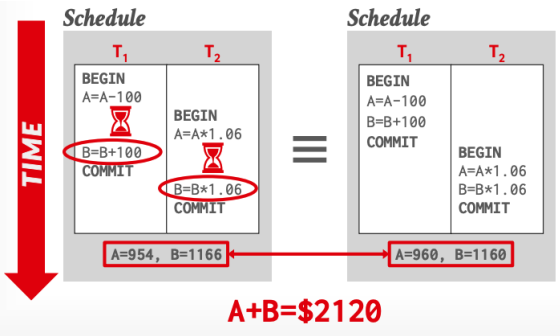


Figure 1: Good interleaving schedule

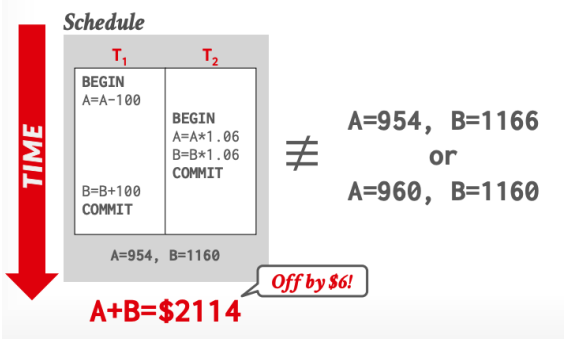


Figure 2: Bad interleaving schedule

The order in which the DBMS executes operations is called an *execution schedule*. We want to interleave transactions to maximize concurrency while ensuring that the output is "correct". The goal of a concurrency control protocol is to generate an execution schedule that is is equivalent to some serial execution:

- **Serial Schedule:** Schedule that does not interleave the actions of different transactions.
- **Equivalent Schedules:** For any database state, if the effect of executing the first schedule is identical to the effect of executing the second schedule, the two schedules are equivalent.
- **Serializable Schedule**: A serializable schedule is a schedule that is equivalent to some serial execution of the transactions. Different serial executions can produce different results, but all are considered "correct". Note that if each transaction preserves the consistency, every serializable schedule also preserves the consistency.

A *conflict* between two operations occurs if the operations are for different transactions, they are performed on the same object, and at least one of the operations is a write. There are three variations of conflicts (note that there are also Phantom Reads and Write-Skew to be covered later): *Lec#18+20*

- **Read-Write Conflicts ("Unrepeatable Reads"):** A transaction is not able to get the same value when reading the same object multiple times.
- **Write-Read Conflicts ("Dirty Reads"):** A transaction sees the write effects of a different transaction before that transaction commits its changes.
- **Write-Write Conflicts ("Lost Updates"):** One transaction overwrites the uncommitted data of another uncommitted transaction.

There are two types for serializability: (1) *conflict serializability* and (2) *view serializability*. Neither definition allows all schedules that one would consider serializable. In practice, DBMSs support conflict serializability because it can be enforced efficiently.

Conflict Serializability

Two schedules are *conflict equivalent* if and only if they involve the same operations of the same transactions and every pair of conflicting operations is ordered in the same way in both schedules. A schedule S is *conflict serializable* if it is conflict equivalent to some serial schedule.

One can verify that a schedule is conflict serializable by swapping consecutive non-conflicting operations of different transactions until a serial schedule is formed. For schedules with many transactions, this becomes too expensive. A better way to verify schedules is to use a *dependency graph* (precedence graph).

In a dependency graph, each transaction is a node in the graph. There exists a directed edge from node T_i to T_j iff an operation O_i from T_i conflicts with an operation O_j from T_j and O_i occurs before O_j in the schedule. Then, a schedule is conflict serializable iff the dependency graph is acyclic.

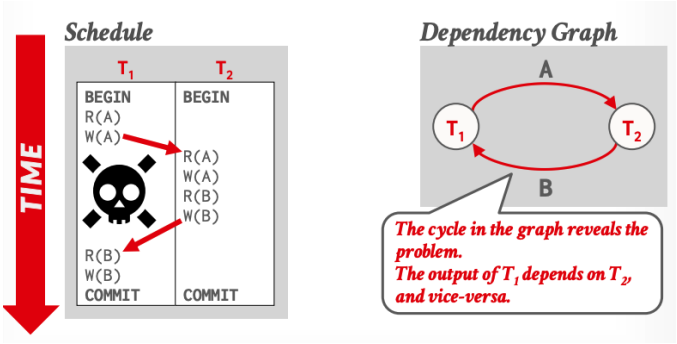


Figure 3: Dependency Graph Example

View Serializability

View serializability is a weaker notion of serializibility that allows for all schedules that are conflict serializable and "blind writes" (i.e. performing writes regardless of its original value). Thus, it allows for more schedules than conflict serializability, but is difficult to enforce efficiently. This is because the DBMS does not know how the application will "interpret" values. As such, view serializability is not used in practice.

Formally, two schedules S_1 and S_2 are *view equivalent* if

- If T_1 reads initial value of A in S_1, then T_1 also reads initial value of A in S_2 .
- If T_1 reads value of A written by T_2 in S_1 , then T_1 also reads value of A written by T_2 in S_2 .
- If T_1 writes final value of A in S_1 , then T_1 also writes final value of A in S_2

About *"blind writes"*: From a database perspective, all that matters is whether the database state is equivalent to some sort of serial execution, so blind writes can still make a set of transactions view serializable even if its dependency graph has cycles.

Universe of Schedules

SerialSchedules $\subset$ ConflictSerializableSchedules $\subset$ ViewSerializableSchedules $\subset$ AllSchedules

7 ACID: Durability

All of the changes of committed transactions must be **durable** (i.e., persistent) after a crash or restart, so there will be no torn updates nor changes from failed transactions. The DBMS can either use logging or shadow paging to ensure that all changes are durable. This usually requires that committed transactions are stored in non-volatile memory.

Lecture #18: Two-Phase Locking

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Transaction Locks

Our notions of serializability thus far have assumed we know all reads and writes while constructing a schedule, but in practice, we need a way to guarantee correctness on the fly. A DBMS uses *locks* to dynamically generate an execution schedule for transactions that is serializable. These locks protect database objects during concurrent access when there are multiple transactions trying to read/write. The DBMS contains a centralized *lock manager* that decides whether a transaction can acquire a lock on each particular object. This is how we avoid deadlocks (rather than being disciplined about their usage like with latches as we don't have control over application logic).

Importantly, locks are different from *latches* such as those used in the B+ tree crabbing algorithm (slide 6). Latches protect the DBMS's internal data structures from concurrent threads whereas locks protect values in the database from concurrent transactions. For example, in a B+ tree, you only hold latches over individual leaf nodes in a scan because that's all you need to do to ensure correctness, but if one transaction attempts a leaf scan while another attempts to write to two arbitrary values, the leaf scan needs to lock the whole table, not just the current leaf, to avoid seeing only one of the two writes. For this reason, locks are usually held for entire transactions while latches are held for short periods to facilitate thread-safe operations on DBMS data structures.

There are two basic lock types (slide 9):

- **Shared Lock (`S-LOCK`):** A shared lock allows multiple transactions to read the same object at the same time. If one transaction holds a shared lock, then another transaction can also acquire that same shared lock.
- **Exclusive Lock (`X-LOCK`):** An exclusive lock allows a transaction to modify an object. This lock prevents other transactions from taking any other lock (`S-LOCK` or `X-LOCK`) on the object. Only one transaction can hold an exclusive lock at a time.

Most real world DBMSs have many more types of locks that build on these and the concepts discussed in this lecture.

Transactions must request locks (or upgrades) from the lock manager. The lock manager grants or blocks requests based on what locks are currently held by other transactions. Transactions must release locks when they no longer need them to free up the object. The lock manager updates its internal lock-table with information about which transactions hold which locks and which transactions are waiting to acquire locks.

The DBMS's lock-table does not need to be durable since any transaction that is active (i.e., still running) when the DBMS crashes is automatically aborted.

However, locks alone are not enough to achieve serializability (slide 11 example). Locks need to be complemented by a concurrency control protocol that ensures locks are used in a way that satisfy correctness guarantees.

2 Two-Phase Locking

Two-Phase locking (2PL) is a pessimistic concurrency control protocol that uses locks to determine whether a transaction is allowed to access an object in the database on the fly. The protocol does not need to know all of the queries that a transaction will execute ahead of time.

Phase #1– Growing: In the growing phase, each transaction requests the locks that it needs from the DBMS's lock manager. The lock manager grants/denies these lock requests.

Phase #2– Shrinking: Transactions enter the shrinking phase immediately after they release their first lock. In the shrinking phase, transactions are only allowed to release locks. They are not allowed to acquire new ones.

On its own, 2PL is sufficient to guarantee conflict serializability. It generates schedules whose precedence graph is acyclic. But it is susceptible to *cascading aborts*, a performance issue where a transaction aborts and then another transaction must be rolled back. This results in wasted work (slide 18), since any computation performed must be rolled back.

2PL can still have dirty reads (W-R Conflict) and it can also lead to deadlocks. There are also potential schedules that are serializable but would not be allowed by 2PL (locking can limit concurrency).

Strong Strict Two-Phase Locking

A schedule is *strict* if any value written by a transaction is never read or overwritten by another transaction until the first transaction commits. *Strong Strict 2PL* (also known as *Rigorous 2PL*) is a variant of 2PL where the transactions only release locks when they commit (slide 19).

The advantage of this approach is that the DBMS does not incur cascading aborts. The DBMS can also reverse the changes of an aborted transaction by restoring the original values of modified tuples. However, Strong Strict 2PL generates more cautious/pessimistic schedules that limit concurrency.

Strict 2PL is another variant that makes slightly weaker guarantees (reintroducing the dirty read / cascading aborts problems) by only releasing shared locks during the shrinking phase.

Universe of Schedules (slide 26)

`SerialSchedules` ⊂ `StrongStrict2PL` ⊂ `ConflictSerializableSchedules` ⊂ `ViewSerializableSchedules` ⊂ `AllSchedules`

3 Deadlock Handling

A *deadlock* is a cycle of transactions waiting for locks to be released by each other. A classic example of a deadlock occurs when Transaction 1 holds a lock on resource A and needs resource B, while Transaction 2 holds resource B and needs resource A. There are two approaches to handling deadlocks in 2PL: detection and prevention.

Approach #1: Deadlock Detection

To detect deadlocks, the DBMS creates a *waits-for* graph where transactions are nodes (slides 30, 31), and there exists a directed edge from T_i to T_j if transaction T_i is waiting for transaction T_j to release a lock. The system will periodically check for cycles in the waits-for graph (usually with a background thread) and then make a decision on how to break it. Latches are not needed when constructing the graph since if the DBMS misses a deadlock in one pass, it will find it in the subsequent passes. Note that there is a

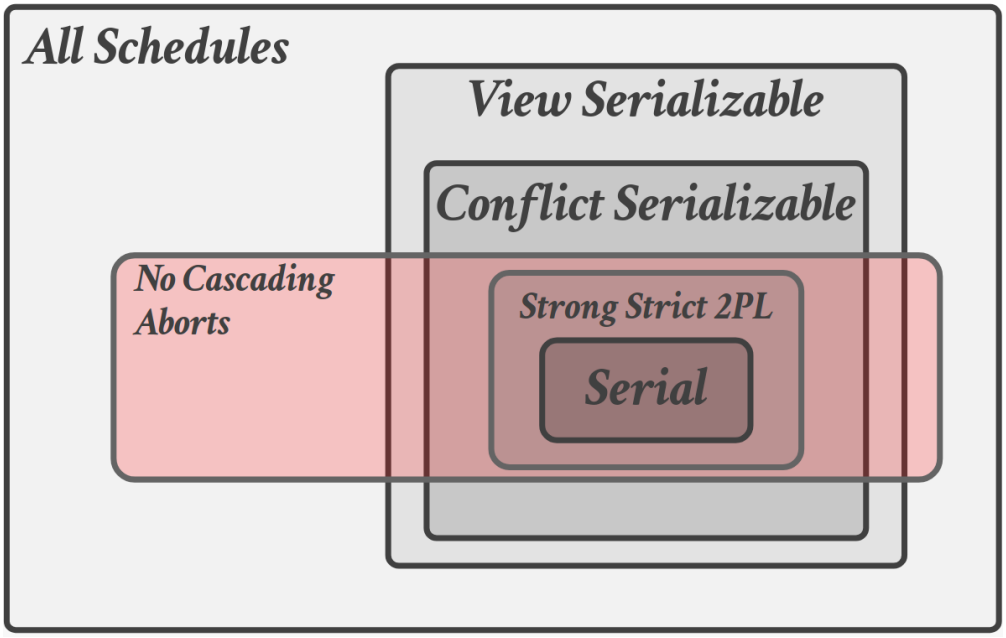


Figure 1: Relationship between schedules, including Serial, Strong Strict 2PL, Conflict Serializable, View Serializable, and schedules without cascading aborts

trade-off between the frequency of deadlock checks (uses CPU cycles) and the wait time until a deadlock is broken. A common heuristic may be to quickly search for any 2-cycles and then fall back to a full cycle detection algorithm if none are found or breaking the 2-cycle isn't sufficient.

When the DBMS detects a deadlock, it will select a "victim" transaction to abort to break the cycle. The victim transaction will either restart or abort depending on how the application invoked it. The DBMS can consider multiple transaction properties when selecting a victim to break the deadlock:

Select?

1. By age (newest or oldest timestamp).
2. By progress (least/most queries executed).
3. By the # of items already locked.
4. By the # of transactions needed to rollback with it.
5. # of times a transaction has been restarted in the past (to avoid starvation).

There is no one choice that is better than others. Many systems use a combination of these factors.

After selecting a victim transaction to abort, the DBMS can also decide on how far to rollback the transaction's changes. It can either rollback the entire transaction or just enough queries to break the deadlock (some DBMSs will establish "savepoints" throughout a transaction that can be rolled back to).

Rollback Length?

Approach #2: Deadlock Prevention

Instead of letting transactions try to acquire any lock they need and then deal with deadlocks afterwards, deadlock prevention 2PL stops transactions from causing deadlocks before they occur. When a transaction tries to acquire a lock held by another transaction (which could cause a deadlock), the DBMS can kill one of them. To implement this, transactions are assigned priorities (potentially based on timestamps with older transactions have higher priority). These schemes guarantee no deadlocks because only one type of direction is allowed when waiting for a lock. When a transaction restarts, the DBMS reuses the same timestamp to prevent it from getting starved for resources.

There are two ways to kill transactions under deadlock prevention (slide 36):

- **Wait-Die ("Old Waits for Young"):** If the requesting transaction has a higher priority than the holding transaction, it waits. Otherwise, it aborts (dies).
- **Wound-Wait ("Young Waits for Old"):** If the requesting transaction has a higher priority than the holding transaction, the holding transaction aborts (gets wounded) and releases the lock. Otherwise, the requesting transaction waits.

To remember these - if the protocol is X-Y, then if the requesting transaction has a higher priority, it will X, and if the requesting transaction has a lower priority, it will Y.

4 Lock Granularities

If a transaction wants to update one billion tuples, it has to ask the DBMS's lock manager for a billion locks. This will be slow because the transaction has to take latches in the lock manager's internal lock table data structure as it acquires/releases locks.

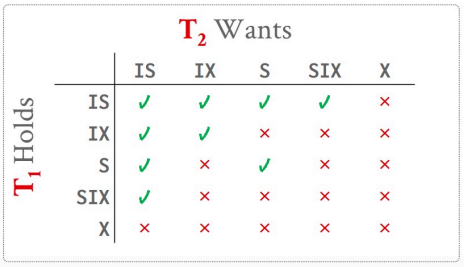
overhead v.s. parallelism

Alternatively, if a transaction locks the entire table when it only needs to read one value, there are less opportunities for parallelism. To handle this trade-off, the DBMS uses a *lock hierarchy* to simultaneously handle locks at different granularity levels. For example, it could acquire a single lock on the table with one billion tuples instead of one billion separate locks.

When a transaction acquires a lock for an object in this hierarchy, it implicitly acquires the locks for all its children objects, so the one-write lock couldn't grab any of the tuple locks. However, if no lock is held on the table, multiple tuple-level locks are allowed on different tuples, allowing parallelism.

Database Lock Hierarchy:

1. Database level (Slightly Rare)
2. Table level (Very Common)
3. Page level (Common)
4. Tuple level (Very Common)
5. Attribute level (Rare)



Importantly, if a transaction is using tuple-level locks, it needs to communicate that no other transaction can grab a page-level lock (or anything higher) since that would conflict. To facilitate this, **intention locks** are implicit locks that signal that there are explicit locks held at lower levels.

- **Intention-Shared (IS):** Indicates explicit locking at a lower level with shared locks.
- **Intention-Exclusive (IX):** Indicates explicit locking at a lower level with exclusive or shared locks.
- **Shared+Intention-Exclusive (SIX):** The sub-tree rooted at that node is locked explicitly in shared mode, and explicit locking is being done at a lower level with exclusive-mode locks.

With hierarchical locking, the DBMS can automatically switch to coarser-grained locks when a transaction acquires many lower level locks. This reduces the number of requests the Lock Manager has to handle.

5 Locking in Practice

In practice, applications don't usually explicitly take locks and the DBMS infers which locks are necessary for a query. Sometimes explicit locking is necessary to improve performance: some databases support a `FOR UPDATE/FOR SHARE` syntax that can go at the end of a `SELECT` query to indicate that a DBMS should take a particular kind of lock. This is particularly useful for read-modify-write workloads: without it, it's possible getting the first shared lock for the read query would succeed and the exclusive lock for the write

would fail, resulting in a rollback and wasted work. With it, you can directly try and acquire an exclusive lock for the read (knowing it will be immediately followed by a write).

Some systems also support a `SKIP LOCKED` suffix that skips any tuples that the DBMS can't acquire a lock for. This is handy for accessing queues inside a DBMS.

Lecture #19: Timestamp Ordering Concurrency Control

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Timestamp Ordering Concurrency Control

Timestamp ordering (T/O) is an optimistic class of concurrency control protocols where the DBMS assumes that transaction conflicts are rare. Instead of requiring transactions to acquire locks before they are allowed to read/write to a database object, the DBMS instead uses timestamps to determine the serializability order of transactions.

Each transaction T_i is assigned a unique fixed timestamp $\text{TS}(T_i)$ that is monotonically increasing. Different schemes assign timestamps at different times during the transaction. Some advanced schemes even assign multiple timestamps per transaction.

If $\text{TS}(T_i) < \text{TS}(T_j)$, then the DBMS must ensure that the execution schedule is equivalent to the serial schedule where T_i appears before T_j.

There are multiple timestamp allocation implementation strategies. The DBMS can use the system clock as a timestamp, typically using UTC to avoid issues from daylight saving. Some DBMSs can get accurate clock time from satellites. Another option is to use a logical counter. However, this has issues with overflow and with maintaining the counter across a distributed system with multiple machines. There are also hybrid approaches that use a combination of both methods.

2 Optimistic Concurrency Control (OCC)

Optimistic concurrency control (OCC) is an optimistic concurrency control protocol which uses timestamps to validate transactions. OCC works best when the number of conflicts is low. This is when either all of the transactions are read-only or when transactions access disjoint subsets of data. If the database is large and the workload is not skewed, then there is a low probability of conflict, making OCC a good choice.

① R/W

In OCC, the DBMS creates a *private workspace* for each transaction. All modifications of the transaction are applied to this workspace. Any object written is copied to the workspace and modified there, any object read is copied to the workspace only if the isolation level guarantees repeatable read. No other transaction can read the changes made by another transaction in its private workspace.

② Validation

When a transaction commits, the DBMS compares the transaction's workspace *write set* to see whether it conflicts with other transactions. If there are no conflicts, the write set is installed into the "global" database.

③ Write

OCC consists of three phases:

1. **Read Phase:** Here, the DBMS tracks the read/write sets of transactions and stores their writes in a private workspace. The DBMS copies every tuple accessed to its private workspace to ensure repeatable reads.

2. **Validation Phase:** When a transaction commits, it is assigned a unique timestamp and the DBMS checks whether it conflicts with other transactions.
3. **Write Phase:** If validation succeeds, the write timestamp is assigned to all modified objects in the private workspace, and the DBMS installs the private workspace's changes to the global database atomically. Otherwise, it aborts and restarts the transaction.

The read phase is where the transaction performs all the works, the validation and the write phase consists the commit protocol.

Dirty Reads
Unrepeatable Reads
Lost Updates

Validation Phase

The DBMS assigns transactions timestamps when they enter the validation phase. To ensure only serializable schedules are permitted, the DBMS checks T_i against other transactions for RW and WW conflicts and makes sure that all conflicts go one way.

- **Approach 1:** Forward validation (from older transactions to younger transactions)
- **Approach 2:** Backward validation (from younger transactions to older transactions)

In forward validation, the DBMS checks the timestamp ordering of the committing transaction with all other running transactions that have not yet committed. Transactions that have not yet entered the validation phase are assigned a timestamp of ∞. [Protect future txn, sacrifice current txn]

If $\text{TS}(T_i) < \text{TS}(T_j)$, then one of the following three conditions must hold:

1. T_i completes its Write phase before T_j begins its Read phase (serial ordering) (slide 16).
2. T_i completes its Write phase before T_j starts its Write phase, and T_i does not write to any object read by T_j (slide 17).
   - $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \varnothing$.
3. T_i completes its Read phase before T_j completes its Read phase, and T_i does not write to any object that is either read or written by T_j (slide 20).
   - $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \varnothing$, and $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j) = \varnothing$.

Common → In backward validation, the DBMS checks the timestamp ordering of the committing transaction with the Read/Write sets of transactions that have already committed since the current transaction started, using the same three conditions as above. This is the more common approach compared to forward validation.

Ideal scenario for OCC: OCC works well when the number of conflicts is low, ideally when all transactions are read-only, or when transactions access disjoint subsets of data.

Potential Issues:

- High overhead for copying data locally into the transaction's private workspace.
- Validation/Write phase bottlenecks.
- Aborts are potentially more wasteful than in other protocols because they only occur after a transaction has already executed.
- Suffers from timestamp allocation bottleneck.

3 Dynamic Databases and The Phantom Problem

In our previous discussions, we have considered transactions that operate on a static set of objects within the database. However, when transactions perform insertions, updates, and deletions, we encounter a new set of complications.

When a transaction scans a range more than once and another transaction inserts or removes tuples that fall within the range between the scans, the scanning transaction can get different results across multitple

scans. This is referred to as phantom read.

The phantom problem arises when transactions only lock existing records, neglecting those that are in the process of being created. Conflict serializability on reads and writes of individual items guarantees serializability **only** if the database's set of objects is fixed.

Approaches to Address the Phantom Problem:

Less Common →

1. **Lock Everything:**
   Transactions may take the lock of the entire table or every page, preventing insertions and deletions from other transactions during its execution. This approach is expensive as it locks more than the transaction needs.
2. **Re-Execute Scans:**
   Transactions may re-run queries at commit time to check for different results, indicating missed changes due to new or deleted records. The DBMS keeps track of the WHERE clauses for all queries executed by the transaction. At commit time, it re-executes the scans to ensure that the results remain consistent.
3. **Predicate Locking:** Check overlap in high dimensional space
   This involves acquiring locks based on the predicates of the queries, ensuring that any data that satisfies the predicate cannot be modified by other transactions. Originally proposed in System R, this scheme is not widely implemented because of its complexity. However, systems like HyPer utilize a form of precision locking that is akin to predicate locking.

Common → ★

4. **Index Locking:**
   Utilizing index keys to protect ranges of data, preventing phantoms by ensuring that no new data can fall within the locked ranges. Different schemes are employed to prevent phantoms using index locking:
   - **Key-Value Locks:** Locks on individual key-values in an index, including virtual keys for non-existent values.
   - **Gap Locks:** Locks on the gap following a key-value, preventing insertion in these gaps.
   - **Key-Range Locks:** Locks on a range of keys, from one existing key to the next.
   - **Hierarchical Locking:** Allows transactions to hold broader key-range locks with different modes, reducing lock manager overhead.

   Lock everything !
   In the absence of a suitable index, transactions must fall back to the first approach to lock every page in the table or the entire table itself to prevent changes that could lead to phantoms. The Index Locking apporach is the most common among the four.

4 Isolation Levels

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance. We may want to use a weaker level of consistency to improve scalability.

Isolation levels control the extent that a transaction is exposed to the actions of other concurrent transactions.

Anomalies: *Lec#17*

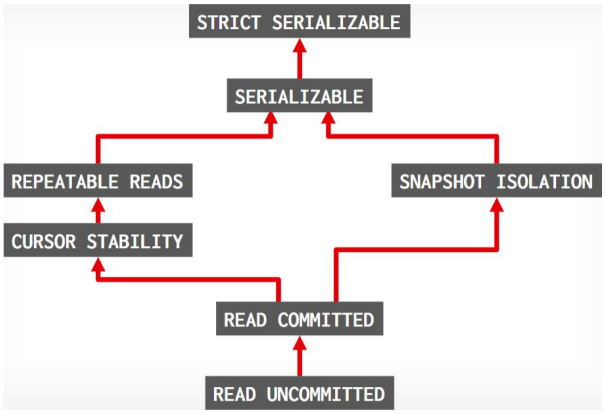
- **Dirty Read:** Reading uncommitted data. W-R Conflict
- **Unrepeatable Reads:** Redoing a read retrieves a different result. R-W Conflict
- **Lost Updates:** Transaction overwrites data of another concurrent transaction. W-W Conflict
- **Phantom Reads:** Insertion or deletions result in different results for the same range scan queries.

Isolation Levels (Strongest to Weakest):

1. SERIALIZABLE: No Phantoms, all reads repeatable, and no dirty reads. (0)
   - Possible implementation: Strict 2PL + Phantom Protection (e.g., index locks).
2. REPEATABLE READS: Phantoms may happen. (1) ← Common
   - Possible implementation: Strict 2PL.
3. READ-COMMITTED: Phantoms, unrepeatable reads, and lost updates may happen. (3)
   - Possible implementation: Strict 2PL for exclusive locks, immediate release of the shared lock after a read.
4. READ-UNCOMMITTED: All anomalies may happen. Dirty Reads (4)
   - Possible implementation: Strict 2PL for exclusive locks, no shared locks for reads.

The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS. An application sets a per-transaction isolation level *before* it starts executing queries.

	Dirty Read	*Unrepeatable Read*	*Lost Updates*	*Phantom*
SERIALIZABLE	No	No	No	No
REPEATABLE READ	No	No	No	Maybe
READ COMMITTED	No	Maybe	Maybe	Maybe
READ UNCOMMITTED	Maybe	Maybe	Maybe	Maybe



	Default	*Maximum*
Actian Ingres	SERIALIZABLE	SERIALIZABLE
IBM DB2	*CURSOR STABILITY	SERIALIZABLE
CockroachDB	SERIALIZABLE	SERIALIZABLE
Google Spanner	STRICT SERIALIZABLE	STRICT SERIALIZABLE
MSFT SQL Server	READ COMMITTED	SERIALIZABLE
MySQL	REPEATABLE READS	SERIALIZABLE
Oracle	READ COMMITTED	*SNAPSHOT ISOLATION
PostgreSQL	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
VoltDB	SERIALIZABLE	SERIALIZABLE
YugaByte	SNAPSHOT ISOLATION	SERIALIZABLE

Lecture #20: Multi-Version Concurrency Control

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) is a larger concept than just a concurrency control protocol. It involves all aspects of the DBMS's design and implementation. MVCC is the most widely used scheme in DBMSs. It is now used in almost every new DBMS implemented in last 10 years. Even some systems (e.g., NoSQL) that do not support multi-statement transactions use it.

With MVCC, the DBMS maintains multiple physical versions of a single logical object in the database. When a transaction writes to an object, the DBMS creates a new version of that object. When a transaction reads an object, it reads the newest version that existed when the transaction started.

The fundamental concept/benefit of MVCC is that writers do not block readers and readers do not block writers. This means that one transaction can modify an object while other transactions read old versions. Writers may still block other writers if they are writing the same object, since there is still a lock on the versions related to the database object. See slide 8 for an example.

One advantage of using MVCC is that read-only transactions can read a consistent **snapshot** of the database without using locks of any kind, and it naturally supports Snapshot Isolation (SI). In MVCC, timestamps are used to determine visibility of versions to transactions. Additionally, MVCC without garbage collection allows the DBMS to support *time-travel queries*, which are queries based on the state of the database at some other point in time (e.g. performing a query on the database as it was 3 hours ago).

A typical MVCC-based database design will:

1. Have a versioned storage which stores different versions of the same logical object. (*Note: Do not do this!*)
2. Takes a snapshot of the database (by copying the transaction status table) when a transaction starts.
3. Use the snapshot to determine which versions of objects are visible to the transaction.

Snapshot Isolation

Snapshot Isolation involves providing a transaction with a consistent snapshot of the database when the transaction started. Data values from a snapshot consist of only values from committed transactions, and the transaction operates in complete isolation from other transactions until it finishes. This is ideal for read-only transactions since they do not need to wait for writes from other transactions. Writes are maintained in a transaction's private workspace or written to the storage with transaction metadata and only become visible to the database once the transaction successfully commits.

Write Conflicts If two transactions update the same object, the first writer wins.

Write Skew Anomaly can occur in Snapshot Isolation when two concurrent transactions modify different objects resulting in non-serializable schedules. For example, if one transaction changes all white marbles to

black and the other changes all black marbles to white, the outcome may not correspond to any serializable schedule. See slides 28 to 31 for details.

There are five important MVCC design considerations:

1. Concurrency Control Protocol
2. Version Storage
3. Garbage Collection
4. Index Management
5. Deletes

Lec#18+19

The choice of concurrency protocol is between the approaches discussed in previous lectures (two-phase locking, timestamp ordering, optimistic concurrency control).

2 Design consideration: Version Storage

This is how the DBMS will store the different physical versions of a logical object and how transactions find the newest version visible to them.

The DBMS uses the tuple's pointer field to create a **version chain** per logical tuple, which is essentially a linked list of versions sorted by timestamp. This allows the DBMS to find the version that is visible to a particular transaction at runtime. Indexes always point to the "head" of the chain, which is either the newest or oldest version depending on implementation. A thread traverses chain until it finds the correct version. Different storage schemes determine where/what to store for each version.

Don't (PostgreSQL)

Approach #1: Append-Only Storage

All physical versions of a logical tuple are stored in the same table space. Versions are mixed together in the table and each update just appends a new version of the tuple into the table and updates the version chain. The chain can either be sorted *oldest-to-newest* (O2N) which requires chain traversal on look-ups, or *newest-to-oldest* (N2O), which requires updating index pointers for every new version (i.e. no need to traverse the chain. This is typically the better approach because most txns only care about the newest version). See slides 37 to 40 for an example.

Approach #2: Time-Travel Storage

The DBMS maintains a separate table called the time-travel table which stores older versions of tuples. On every update, the DBMS copies the old version of the tuple to the time-travel table and overwrites the tuple in the main table with the new data. Pointers of tuples in the main table point to past versions in the time-travel table. See slides 42 to 46 for an example.

Common

Approach #3: Delta Storage

Like time-travel storage, but instead of the entire past tuples, the DBMS only stores the deltas, or changes between tuples in what is known as the delta storage segment. Transactions can then recreate older versions by iterating through the deltas in reverse order and applying them. This results in faster writes than time-travel storage but slower reads. See slides 47 to 51 for an example.

3 Design consideration: Garbage Collection

The DBMS needs to remove *reclaimable* physical versions from the database over time. A version is reclaimable if no active transaction can "see" that version or if it was created by a transaction that was aborted. We can either perform tuple level, or transaction level garbage collection.

Approach #1: Tuple-level GC

With tuple-level garbage collection, the DBMS finds old versions by examining tuples directly. There are two approaches to achieve this:

- **Background Vacuuming:** Separate threads periodically scan the table and look for reclaimable versions. This works with any version storage scheme. A simple optimization is to maintain a "dirty page bitmap," which keeps track of which pages have been modified since the last scan. This allows the threads to skip pages which have not changed. See slides 54 to 60 for an example.
- **Cooperative Cleaning:** Worker threads identify reclaimable versions as they traverse version chain. This only works with O2N chains. The data will never be cleaned if it is not accessed.

Approach #2: Transaction-level GC

Under transaction-level garbage collection, each transaction is responsible for keeping track of their own old versions so the DBMS does not have to scan tuples. Each transaction maintains its own read/write set. When a transaction completes, the garbage collector can use that to identify which tuples to reclaim. The DBMS determines when all versions created by a finished transaction are no longer visible. See slides 66 to 73 for an example.

4 Design consideration: Index Management

聚簇索引(主键) & 二级索引(辅助)

All primary key (pkey) indexes always point to version chain head. How often the DBMS has to update the pkey index depends on whether the system creates new versions when a tuple is updated. If a transaction updates a pkey attribute(s), then this is treated as a `DELETE` followed by an `INSERT`.

Lec#8

Managing secondary indexes is more complicated. There are two approaches to handling them.

Approach #1: Logical Pointers

The DBMS uses a fixed identifier per tuple that does not change. This requires an extra indirection layer that maps the logical id to the physical location of the tuple. Then, updates to tuples can just update the mapping in the indirection layer.

Approach #2: Physical Pointers

The DBMS uses the physical address to the version chain head. This requires updating every index when the version chain head is updated, which can be very expensive.

Imaging a table with a primary index, and multiple secondary indices, using physical pointers requires each secondary index to point to the tuple version chain's physical address, and each update to the tuple requires and update to all the secondary indices. With a logical pointer approach, the secondary indices point to the location in the primary index corresponding to the tuple, thus requiring less overhead when the tuple is updated.

↓
rollback (回表 bottleneck)

MVCC duplicate key problem

Exception: Index-Organized Table (MySQL-InnoDB)

MVCC DBMS indexes (usually) do not store version information about tuples with their keys. Instead, every index must support duplicate keys from different snapshots, since the same key may point to different logical tuples in different snapshots.

MVCC Duplicate Key Problem describes this need to support duplicate keys in MVCC DBMS indexes when multiple transactions need multiple versions of the same logical tuple. For example, say TXN 1 points to version 0 of a logical tuple, and TXN 2 creates a version 1 of that same logical tuple. Our index would need to point to both of these versions for both TXNs so the proper version is accessible to each transaction at all times.

INSERT ... ON DUPLICATE KEY UPDATE

Workers may get back multiple entries for a single fetch, and they then must follow the pointers to find their proper physical version.

5 Design consideration: Deletes

The DBMS physically deletes a tuple from the database only when all versions of a logically deleted tuple are not visible. If a tuple is a deleted, then there cannot be a new version of that tuple after the newest version. This means no write-write conflicts, and the first-writer wins.

We need a way to denote that a tuple has been logically deleted at some point in time. There are two approaches to this.

Approach #1: Deleted Flag

Maintain a flag to indicate that the logical tuple has been deleted after the newest physical version. This can either be in the tuple header or a separate column.

MVCC metadata: txn_id, start_ts, end_ts

Approach #2: Tombstone Tuple

Create an empty physical version to indicate that a logical tuple is deleted. Use a separate pool for tombstone tuples with only a special bit pattern in version chain pointer to reduce storage overhead.

	Protocol	*Version Storage*	*Garbage Collection*	*Indexes*
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-TO	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
MSSQL Hekaton	MV-OCC	Append-Only	Cooperative	Physical
SingleStore	MV-OCC	Delta	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
DuckDB	MV-OCC	Delta	Txn-Level	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical
CockroachDB	MV-2PL	Delta (LSM)	Compaction	Logical