# Lecture #19: Two-Phase Locking Concurrency Control

**15-445/645 Database Systems (Fall 2026)**
https://15445.courses.cs.cmu.edu/fall2026/
Carnegie Mellon University
Jignesh Patel

## 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 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* holds all exclusive locks until commit/abort, but allows shared locks to be released during the shrinking phase. Like Strong Strict 2PL, it produces recoverable schedules that avoid cascading aborts and are strict; Strong Strict 2PL additionally holds shared locks until commit/abort, producing rigorous schedules.

### Universe of Schedules (slide 26)

`SerialSchedules` $\subset$ `StrongStrict2PL` $\subset$ `ConflictSerializableSchedules`

$\subset$ `ViewSerializableSchedules` $\subset$ `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)

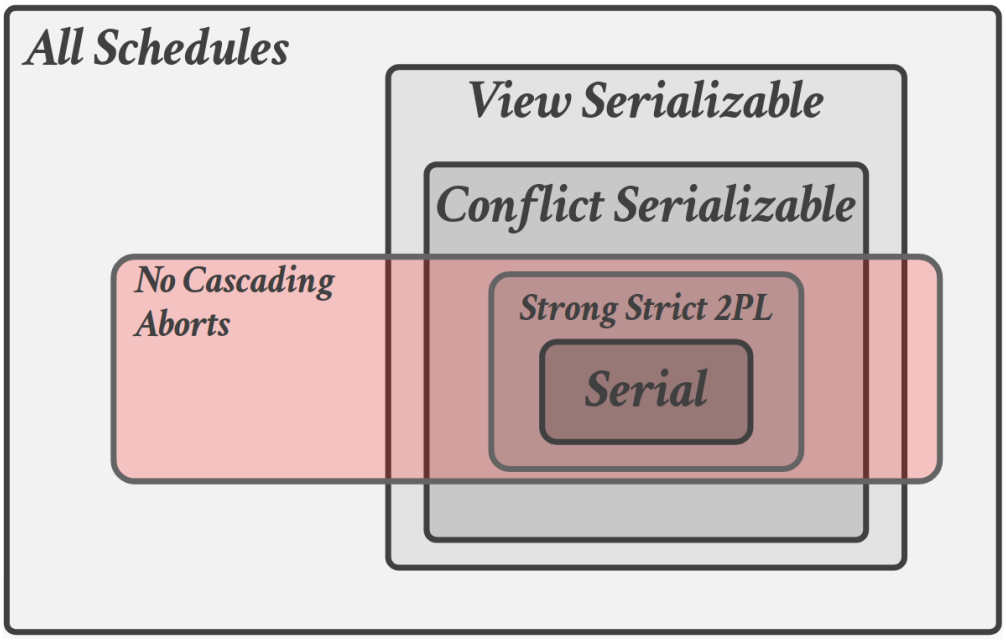



**Figure 1:** Relationship between schedules, including Serial, Strong Strict 2PL, Conflict Serializable, View Serializable, and schedules without cascading aborts

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 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:

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).

### 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.

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.