# Lecture #02: Transaction Models & Concurrency Control

**15-721 Advanced Database Systems (Spring 2019)**
`https://15721.courses.cs.cmu.edu/spring2019/`
Carnegie Mellon University
Prof. Andy Pavlo

## 1 Background

### Database Workloads

- **On-Line Transaction Processing (OLTP):** Fast operations that only read/update a small amount of data each time.
- **On-Line Analytical Processing (OLAP):** Complex queries that read a lot of data to compute aggregates.
- **Hybrid Transaction + Analytical Processing (HTAP):** OLTP + OLAP together on the same database instance.

### Transactions

A transaction is defined as a sequence of *actions* that are executed on a shared database to perform some higher-level function. It is a basic unit of change in the DBMS. No partial transactions are allowed.

There are three categories of actions that the DBMS can execute.

- **Unprotected Actions:** Low-level operations on physical resources (e.g., disk, memory). These lack all of the ACID properties except for consistency. Their effects cannot be depended upon.
- **Protected Actions:** These are the high-level changes that the application wants to perform on the database. The DBMS does not externalize their results before they are completely done. Fully ACID.
- **Real Actions:** These affect the physical world in a way that is hard or impossible to reverse. For example, if the application sends out an email, then the DBMS cannot retract it.

## 2 Transaction Models

A *transaction model* specifies the execution semantics of protected actions.

### Flat Transactions

Standard transaction model that starts with `BEGIN`, followed by one or more actions, and then completed with either `COMMIT` or `ROLLBACK`. This is what most people think of when discussing transaction support in a DBMS.

There are several limitations to flat transactions that motivate us to consider other models. Foremost is that the application can only rollback the entire transaction (i.e., no partial rollbacks). All of a transaction's work is lost is the DBMS fails before that transaction finishes. Each transaction takes place at a single point in time.

### Transaction Savepoints

Save the current state of processing for the transaction and provide a handle for the application to refer to that savepoint.

The application can control the state of the transaction through these savepoints. The application can create a handle with the `SAVEPOINT` command during a transaction. It can use `ROLLBACK` to revert all changes back to the state of the database at a given savepoint. It can also use `RELEASE` to destroy a savepoint previously defined in the transaction.

### Nested Transactions

The invocation of a transaction during the execution of another transaction. The nested transactions form a hierarchy of work. The outcome of a child transaction depends on the outcome of its parent transaction.

### Chained Transactions

The ability to link multiple transactions one after each other. The combined `COMMIT` and `BEGIN` operations between two transactions is atomic. This means that no other transaction can change the state of the database as seen by the second transaction from the time that the first transaction commits and the second transaction begins.

### Compensating Transactions

A special type of transaction that is designed to semantically *reverse* the effects of another already committed transaction. Such a reversal has to be logical instead of physical.

### Saga Transactions

A sequence of chained transactions $T_1$-$T_n$ and compensating transactions $C_1$-$C_{n-1}$ where one of the following is guaranteed: The transactions will commit in the order $T_1$,...$T_j$,$C_j$...$C_1$ (where $j < n$).

## 3 Concurrency Control

A DBMS's *concurrency control protocol* to allow transactions to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system. The goal is to have the effect of a group of transactions on the database's state is equivalent to any serial execution of all transactions.

Concurrency Control Schemes

1. **Two-Phase Locking (Pessimistic):** Assume transactions will conflict so they must acquire locks on database objects before they are allowed to access them.
2. **Timestamp Ordering (Optimistic):** Assume that conflicts are rare so transactions do not need to first acquire locks on database objects and instead check for conflicts at commit time.

## 4 Two-Phase Locking

There are two ways to deal with deadlocks in a two-phase locking (2PL) concurrency control protocol:

- **Deadlock Detection:** If deadlock is found, use a heuristic to decide what transaction to kill in order to break deadlock.
- **Deadlock Prevention:** If lock is not available, then make a decision about how to proceed.

## 5 Timestamp Ordering Concurrency Control

Use timestamps to determine the order of transactions.

### Basic T/O Protocol

- Every transaction is assigned a unique timestamp when they arrive in the system.
- The DBMS maintains separate timestamps in each tuple's header of the last transaction that read that tuple or wrote to it.
- Each transaction check for conflicts on each read/write by comparing their timestamp with the timestamp of the tuple they are accessing.
- The DBMS needs copy a tuple into the transaction's private workspace when reading a tuple to ensure repeatable reads.

### Optimistic Concurrency Control (OCC)

Store all changes in private workspace. Check for conflicts at commit time and then merge. First proposed in 1981 at CMU by H. T. Kung [3].

The protocol puts transactions through three phases during its execution:

1. **Read Phase:** Transaction's copy tuples accessed to private work space to ensure repeatable reads, and keep track of read/write sets.
2. **Validation Phase:** When the transaction invokes `COMMIT`, the DBMS checks if it conflicts with other transactions. Parallel validation means that each transaction must check the read/write set of other transactions that are trying to validate at the same time. Each transaction has to acquire locks for its write set records in some global order. Original OCC uses serial validation.
   The DBMS can proceed with the validation in two directions:
   - Backward Validation: Check whether the committing transaction intersects its read/write sets with those of any transactions that have <u>already</u> committed.
   - Forward Validation: Check whether the committing transaction intersects its read/write sets with any active transactions that have <u>not</u> yet committed.
3. **Write Phase:** The DBMS propagates the changes in the transactions write set to the database and makes them visible to other transactions' items. As each record is updated, the transaction releases the lock acquired during the Validation Phase

### Timestamp Allocation

There are different ways for the DBMS to allocate timestamps for transactions [4]. Each have their own performance trade-offs.

- **Mutex:** This is the worst option. Mutexes are <u>always</u> a terrible idea.
- **Atomic Addition:** Use compare-and-swap to increment a single global counter. Requires cache invalidation on write.
- **Batched Atomic Addition:** Use compare-and-swap to increment a single global counter in batches. Needs a back-off mechanism to prevent fast burn.
- **Hardware Clock:** The CPU maintains an internal clock (not wall clock) that is synchronized across all cores. Intel only. Not sure if it will exist in future CPUs.
- **Hardware Counter:** Single global counter maintained in hardware. Not implemented in any existing CPUs.

## 6 Performance Bottlenecks

All concurrency control protocols have performance and scalability problems when there are a large number of concurrent threads and large amount of contention (i.e., the transactions are all trying to read/write to the same set of tuples).

**Lock Thrashing:**

- Each transaction waits longer to acquire locks, causing other transaction to wait longer to acquire locks.
- Can measure this phenomenon by removing deadlock detection/prevention overhead.

**Memory Allocation**

- Copying data on every read/write access slows down the DBMS because of contention on the memory controller.
- Default libc malloc is slow. Never use it.

## 7 Isolation Levels

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance. Thus, an application 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.

The SQL standard definitions isolation levels in terms of what anomalies a transaction could be exposed to during execution:

- **Dirty Read:** Reading uncommitted data.
- **Unrepeatable Reads:** Redoing a read results in a different result.
- **Phantom Reads:** Insertion or deletions result in different results for the same range scan queries.

Isolation levels (strongest to weakest) [1]:

1. `SERIALIZABLE`: No Phantoms, all reads repeatable, and no dirty reads.
2. `REPEATABLE READS`: Phantoms may happen.
3. `READ COMMITTED`: Phantoms and unrepeatable reads may happen.
4. `READ UNCOMMITTED`: All anomalies may happen.

The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS [2]. There are two additional isolation levels:

1. `CURSOR STABILITY`
   - Between repeatable reads and read committed
   - Prevents "Lost Update" Anomaly.
   - Default isolation level in **IBM DB2**.
2. `SNAPSHOT ISOLATION`
   - Guarantees that all reads made in a transaction see a consistent snapshot of the database that existed at the time the transaction started.
   - A transaction will commit only if its writes do not conflict with any concurrent updates made since that snapshot.
   - Susceptible to write skew anomaly.

## References

[1] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, Feb 2000. URL `https://doi.org/10.1109/ICDE.2000.839388`.

[2] H. Berenson, P. Bernstein, J. Gray, M. Jim, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95 Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10. URL `https://dl.acm.org/citation.cfm?id=223785`.

[3] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Datab. Syst*, 6(2), June 1981. URL `https://dl.acm.org/citation.cfm?id=319567`.

[4] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: an evaluation of concurrency control with one thousand cores. In *VLDB '14: Proceedings of the VLDB Endowment*, volume 8, pages 209–220, November 2014. URL `https://dl.acm.org/citation.cfm?id=2735511`.