Lecture #17: Concurrency Control Theory

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

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.

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
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
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.

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.

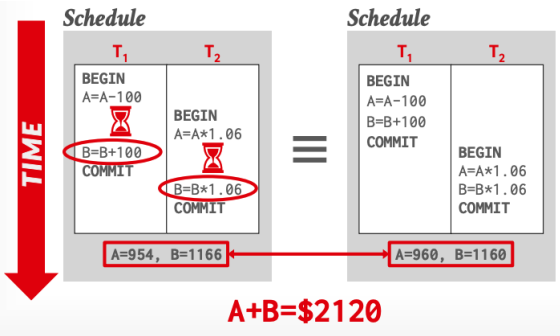


Figure 1: Good interleaving schedule

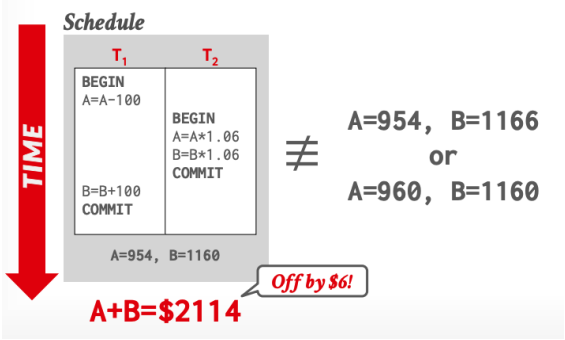


Figure 2: Bad interleaving schedule

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):

- **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.

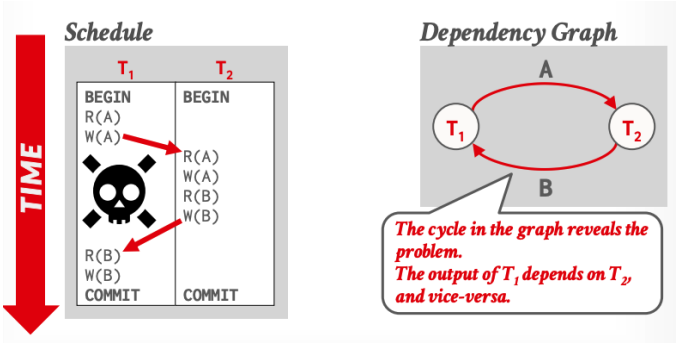


Figure 3: Dependency Graph Example

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.