**Data 101: Data Engineering** Note 10 – Nov 1, 2022

# SQL Modifications, Transactions and Recovery

*Lecturer: Lakshya Jain* *Scribe: Evelyn Liu*

## 1 Transactions

### 1.1 Introduction

In most situations, many users can query & update a database simultaneously, causing concurrency issues. One user can write to the database while another user reads from the same resource. Or both users may try to write to the same resource. We use transactions to address these problems. A transaction is a sequence of multiple actions to be executed as a single, logical, atomic unit. From SQL view, a transaction is in the form of:

- Begin transaction
- Sequence of SQL statements
- End transaction

### 1.2 The Basic Guarantees: ACID

Transactions guarantee the ACID properties to avoid the concurrency problems discussed above:

- Atomicity: A transaction ends in two ways: it either commits or aborts. Atomicity means that either all actions in the Xact happen, or none happen
- Consistency: If the DB starts out consistent, it ends up consistent at the end of the Xact
- Isolation: Execution of each Xact is isolated from that of others. In reality, the DBMS will interleave actions of many Xacts and not execute each in the order of one after the other. The DBMS will ensure that each Xact executes as if it ran by itself.
- Durability: If the COMMIT succeeds, all changes from the transaction persist until overwritten by a later transaction

## 2 Serialization

### 2.1 Serial Schedules

- Transaction schedule: a sequence of reads and writes by named transactions on named objects.
- Serial Schedules: transactions that run from start to commit without interleaved actions from any other.
- Serializable Schedules: a schedule that has results equivalent to a serial schedule.

### 2.2 Conflict Dependency Graph

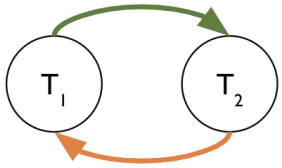
How do we know if a schedule is serializable? We define a notion of conflicting actions, and two actions conflict if:

1. They are from two different, concurrent transactions
2. They reference the same object
3. At least one is write

One way to check if a schedule is serializable is to build a conflict graph. Conflict graphs have the following structure:

1. One node per Xact
2. Edge from Ti to Tj if:
    - An operation Oi of Ti conflicts with an operation Oj of Tj
    - Oi appears earlier in the schedule than Oj

A cycle corresponds to a schedule that is not conflict serializable. A schedule is conflict serializable if and only if its dependency graph is acyclic. Every conflict serializable schedule is serializable. An example of a schedule that is not conflict serializable is shown below.




### 2.3 Strict 2PL

Databases need to be able to interleave the actions of many transactions, while guaranteeing isolation. To ensure 2 conflicting actions happen in order, we need the first conflicting action that arrives "lock" the shared object, then the second action waits until the first action's transaction completes. Strict two-phase locking (Strict 2PL) is a scheme to ensure the database use conflict serializable schedules and therefore only allows serializable schedules. The two rules for 2PL are:

1. Phase 1: Before read, a transaction must acquire a shared (S) lock on the resource. Before write, a transaction must acquire an exclusive (X) lock on the resource. If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
2. Phase 2: At end of a transaction, drop all locks at once.

## 3 Isolation

Serializable transactions ensure the ACID properties, and strict 2PL is a common implementation of serializability. However, sometimes we want to trade correctness for performance. We can use weak isolation to allow a transaction to be a little "sloppy" as long as it doesn't mess up other transactions' choices. Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena:

- Dirty read: the situation where a transaction reads data that has not yet been committed.
- Non-repeatable reads: a transaction reads the same tuple twice and gets a different value each time because another transaction runs between the two reads and updates the record.

- Phantoms: two same queries are executed, but the rows retrieved by the two are different. E.g. "find all students with an A grade". If you run it again, some brand-new tuples (phantoms) appear.

### 3.1 Isolation Level

Based on these phenomena, we define three types of weak isolation modes:

- Read committed isolation
  - We dropped each Shared lock right after reading but kept our eXclusive locks until COMMIT/ROLLBACK.
  - Prevents "dirty" (uncommitted) reads from other transactions.
- Repeatable read isolation
  - Prevents dirty reads and non-repeatable reads.
  - Phantoms are still possible.
  - The transaction holds a read or write lock on the tuple until COMMIT/ROLLBACK, and thus prevents other transactions from reading, updating, or deleting it.
- Snapshot isolation
  - In a snapshot isolated system, each transaction appears to operate on an independent, consistent snapshot of the database.
  - All the reads of a transaction are from the same snapshot.
  - A transaction can commit if the values updated by this transaction have not been changed externally since the snapshot was taken.
  - If transaction T1 has modified an object x, and another transaction T2 committed a write to x after T1's snapshot began, and before T1's commit, then T1 must abort.

## 4 Lock Management

### 4.1 Protocols for Writes (X Locks)

We can use lock tables to manage lock acquisition and release. A lock table usually looks like this

| Item | Lock Mode | Granted | Wait Queue |
|---|---|---|---|
| O | S | T2, T3 | |
| | | | |
| | | | |

When T1 tries to acquire X lock on resource O, check: is O in the lock table?

- If no, add an entry (O (resource), X (lock mode), T1 (Xact), NULL (wait queue)) to the lock table. Allow T1 to write O and proceed.
- If yes, then there is another Xact holding the X lock on resource O. We add T1 to the wait queue for O.

When T1 releases an X lock on resource O, check: is the wait queue for O empty?

- If yes: delete O's entry from the lock table.
- If no: remove all mutually compatible items from the head of the wait queue (multiple S locks or a single X lock). Set the lock mode for O to the mode of those items and let those transactions proceed.

## 4.2 Protocols for Reads (S Locks)

When T1 tries to acquire S lock on resource O, check: is O in the lock table?

- If no: add an entry (O, S, T1, NULL) to the lock table. Allow T1 to read O and proceed.
- if yes: add T1 to the granted list for O if the current lock mode is S. Add T1 to the wait queue for O if the current lock mode is X.

## 4.3 Deadlock

- A deadlock occurs when a bunch of transactions is waiting on each other in a cycle. For example, T1 waits for T2 to release a lock, but T2 also waits for T1 to release a lock.
- The solution to this problem is that system periodically detects deadlock cycles and rolls back one transaction on the cycle.