Lecture #8: Granularities of Locks and Degrees of Consistency

15-721 Advanced Database Systems (Fall 2024)
https://www.cs.cmu.edu/~15721-f24//
Carnegie Mellon University
Author: Daniel Cai (Andrew ID: dcai)

1 Introduction

1.1 Background

In some applications like bank transfer, we want a group of database operations to be executed as a whole: either all of them succeed, or all of them fail. This feature is called **transaction**, which is a sequence of reads and writes we hope to execute atomically. Transaction guarantees the **ACID** property: which stands for atomicity, consistency, isolation and durability.

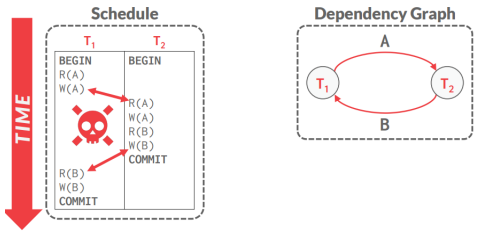

Figure 1: Dependency Graphs can be used to check if the Schedule is serial

This topic of transaction is often discussed together with **concurrency control**: how to enable multiple users to read and modify the database at the same time? To study these topics, the concept of a **schedule** is introduced, which shows the order of operations each transaction executes. Based on schedules, we can introduce **serializability**: if a schedule leaves the database in a state that can be achieved by running the transaction one by one, we say the schedule is serializable. Note that serializablility does not specify any ordering of the transactions.

As shown in figure 1, with the dependency graph, we can check if there are conflicts in a schedule and determine if it is serializable.

1.2 Tradeoffs in Transaction Management

The paper discussed two aspects of transaction management: granularity of lock and transaction isolation levels. Behind these two topics are two tradeoffs: **concurrency** and locking overhead; **consistency** and locking overhead.

2 Granularity of Lock in Database

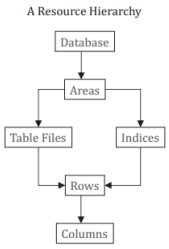

Figure 2: Hierarchy of Resources in a Database

As shown by the image, in a database system, there might be resources of different granularity, from entire database, to tables, to rows and finally each field of a row, forming a tree-like hierarchy. Here, we face the tradeoff between concurrency and locking overhead: if we only allow coarse-grained locks like locks on the entire database, the concurrency will be low. On the other end of the spectrum, if we allow only column level locks, there will be a high overhead maintaining the locks. Therefore, we must allow locks of different granularity to coexist. This is further complicated by the existence of indices, which make the resource hierarchy a DAG. Given a tree of resources, how to properly acquire locks at different level?

2.1 Hierarchical Locking

The key observation here is this: if a transaction locks a node of the tree (such as a table), then that implies all the decedents of that nodes are also locked by that transaction (tuples and fields). Therefore, when locking a resources, we must make sure no incompatible locks are acquired at high levels. To solve this problem, the author introduced the concept of **intention locks**.

2.1.1 Intention Locks: What are the locks

In addition to the shared(S) and exclusive lock(X), the author introduce three intention locks:

1. **Intention-Shared** (IS) locks, which indicates a S lock is acquired at lower level
2. **Intention-Exclusive** (IX) locks, which indicates a X lock is acquired at lower level
3. **Shared+Intention-Exclusive** (SIX) lock: SIX lock is the combination between S and IX lock: The sub tree at that node is locked in S mode and a X lock is acquired at lower level. For instance, one example that requires this lock is a query that sets one value to the average value of the table.

The compatibility matrix of locks is given below. The non-trivial thing to notice here is that IX and IX, IX and IS are actually compatible on a node, because IX only indicates a descendent of that node is locked in exclusive mode, and two transactions might be holding locks on different descendents.

Figure 3: Lock Compatibility Matrix

2.1.2 Locking Protocol: What locks to acquire?

With intention lock, let's see how locks are acquired and released for hierarchical resources. Suppose a transaction want to get a X (S) lock, we start from the root node, and apply a IX (IS) locks for each

node in a top down manner. When releasing the locks, it is done reversely, starting bottom up. With this protocol, the locks on leaf nodes must be none-intention lock. Intuitively, when locking, the resource hierarchy is traversed top-down, and the corresponding intention locks is locked at each level. In this way, if a transaction has an intention lock, every other transaction will be able to know that a S or X lock is acquired at a lower level in the resource hierarchy, and will not mistakenly put an incompatible lock on that resource.

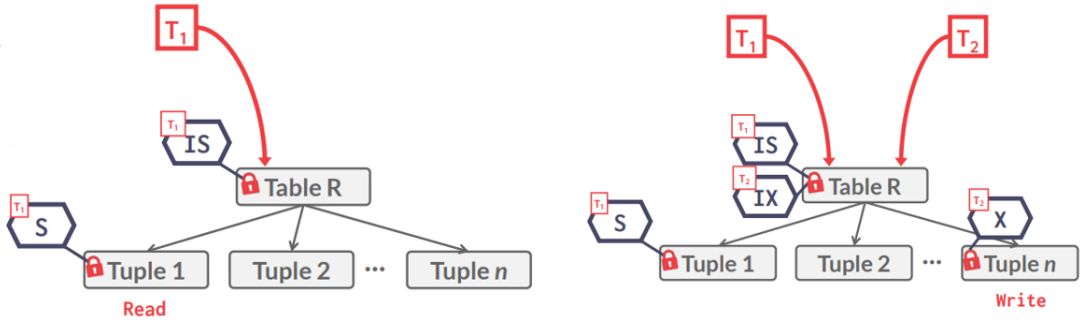

Figure 4: Example

To better understanding this protocol, consider the example shown in figure 4: Suppose the transaction T_1 wants to acquire an S lock on Tuple 1. To do so, it starts from the top of the resource hierarchy, first placing an intention-shared lock on Table R, then places the S lock on tuple 1. The second transaction T_2 wants to acquire a write lock on Tuple n. It first check Table R: since the existing IS lock of T_1 is compatible with IX lock, T_2 places an IX lock. Then, it descends to Tuple n and place an X lock. When unlocking, both transaction release locks from bottom up.

2.2 Locking Schedule and Update

Another aspect of this protocol is locking schedules and updates: suppose a particular resource has already been locked by several transactions and there are some other transactions are pending on the resource. Which transaction should be allowed next? One simple solution is to use a first-come-first-serve policy: granting lock to the first transactions whose lock is compatible with existing locks. Here, the author used **lock upgrade** instead: giving priority to a transaction if it is already part of the granted group. In this way, the transaction is able to finish quicker and free up the resource sooner.

2.3 Deadlocks

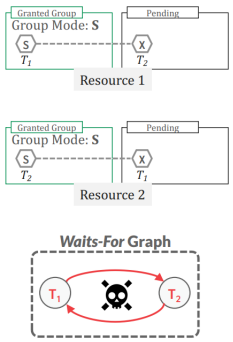

Figure 5: Two transaction waiting for each other, causing deadlock

The problem of **deadlock** (conflicting locks) is almost unavoidable in any lock-based concurrency control protocols. Consider the figure above, with two resources and two transactions. T_1 has acquired shared lock on R_1 and is pending for R_2. T_2 has acquired shared lock for R_2 is requesting exclusive lock on R_1.

Essentially, each transaction is waiting for the other to release the resource and neither can move forward, forming a cycle in the wait-for graph.

Two approaches based on the wait-for graph can be used to solve the problem of deadlock: Deadlock prevention aborts the conflicting transaction (the one causing cycle in wait-for graph) immediately and restarts it. Deadlock prevention constructs and periodically examines cycles in the wait for graph, picking one transaction to abort when a cycle is formed.

2.4 Two Phase Locking

Up to this points, it is fairly clear *how* to acquire and release locks in a database system with resources of different granularity. However, the *how*s are not sufficient to get conflict serializability guarantee in this system and the problem of *when* to acquire and release locks is equally important, which leads us to Two Phase Locking.

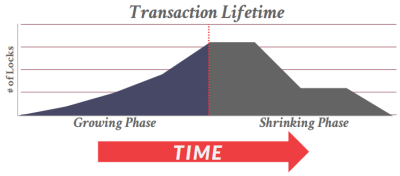

Figure 6: Lock count in transaction lifetime with 2 Phase Locking

As shown in the diagram above, with **two-phase locking** (2PL), for each transaction, its locking behaviors are divided into two phrases, the growing phase when it acquires locks, and the shrinking phase when it releases lock. In other words, when the transaction finished a read or write operation on a record, it does not release the lock immediately. Instead, it wait until the moment when all operations requiring lock have finished. Only until then does it start releasing locks. With two phase locking protocol, the transactions are conflict-serializable.

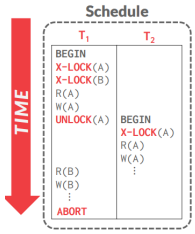

Figure 7: Example Cascading Abort Schedule

Nevertheless, this method still suffers from the problem of **cascading abort**. Consider the schedule above in figure above: The unlocking of A in T_1 caused T_2 to read the modified version of A. As a result, when T_1 aborts, since T_2 has read the value modified by A, T_2 also has to abort. If some transaction T_3 also read the A modified by T_2, it will also need to abort, causing a cascade of failed transactions. To solve this problem, **strict two-phase commit** is proposed: only releasing locks when at the end of a transaction (commit or abort).

3 Isolation Level

Just like the trade of between locking cost and concurrency, which motivates the study of the granularity of locks, there is also the tradeoff between locking cost and consistency, leading us to the topic of **isolation levels**

Isolation levels are different degree of consistency in a system, which is the result of different lock techniques. For some workload, it might be acceptable with lower level of consistency. Consider the example of online

shopping cart update versus bank transfers. For the former, it is acceptable to exchange consistency for better performance.

3.1 Four Isolation Levels

The author proposes four isolation levels. Below, we discuss their characteristic and locking patterns.

1. **Read Uncommitted:** The is the lowest isolation level, where a transaction can read uncommitted writes of another transaction. In this mode, no read locks are used (only write locks).
2. **Read Committed:** A transaction can read the committed writes of another transaction (which also violates Isolation in ACID). In this mode, S and X locks are used are released immediately (no two phase locking).
3. **Repeatable Reads:** In a transaction, all the reads on the same values will return same value, meaning that the reads are no longer interfered by writes of another transaction, despite committed or not. In this mode, two phase locking is used. However, it is still susceptible to **Phantoms** (reading records created by other transactions), which will be discussed in the next section.
4. **Serializable** This is the highest isolation level, only allow serializable schedules. This is the result of predicating locking (rule out Phantoms) and strict two phase locking.

Support for different isolation levels in each transaction has already been in the SQL standard: with "BEGIN TRANSACTION ISOLATION LEVEL" statement, one can specify the isolation level that a transaction runs in, giving great flexibility for handling the tradeoff between consistency and performance. It is also worth noting that different databases have different default isolation level: for instance, the default isolation level in MySQL is Repeatable Read and the maximum it can support is serializable.

3.2 Phantoms

Phantom is a type of violation of isolation caused by reading a record created by another transaction.

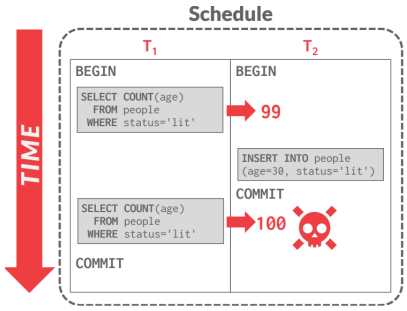

Figure 8: Example Cascading Abort Schedule

As shown in the example schedule, due to the newly inserted record by T_2, the count statement now return 100 instead of 99. Solving this problem requires **predicate locking**: checking if the newly inserted record falls into the predicate specified in the read statements. In systems with B-tree index, this can be done using the index: For instance, if there is a statement in the transaction counting the number of records larger than 5, then inserting a record in this range will not be allowed.

4 Conclusion

This lecture focuses on two topics: granularity of locks and different isolation levels. These topics arise for tradeoffs that transaction management has to handle, namely the tradeoff between concurrency and locking overhead, and between consistency and locking overhead.

Nowadays, aside from lock-based concurrency control, other techniques without locks have been developed, such as optimistic concurrency control. For isolation levels, more finer-grained isolation levels have also been proposed.

5 Additional Materials

There are unofficial materials (not required by the course) that I think might be relevant. For many concepts covered in undergraduate database courses, I do not go into too much detail in this note and these two courses might be helpful

1. CMU Database Course: https://15445.courses.cs.cmu.edu/fall2022/schedule.html
2. Berkeley Database Course: https://cs186berkeley.net/

In addition, another good material is the documentation of MySQL on transaction isolation levels. It covers the characteristics and locking techniques (combination of locking and MVCC) used to achieve these levels of consistency and provides insight into how these features are implemented in real-life databases.