CMPSCI 677 Distributed and Operating Systems **Spring 2019**

Lecture 15: March 28

Lecturer: Prashant Shenoy

15.1 Overview

This section covers the following topics:

Distributed Transactions: ACID Properties, Concurrency Control, Serializability, Two-Phase Locking

15.2 Transactions

Transactions are used widely in databases as they provide atomicity (all or nothing property). Atomicity is the property in which operations inside a transaction all execute successfully or none of the operations execute. Thus, if a process crashes during a transaction, all operations that happened during that transaction must be undone.

Transactions are usually implemented using locks. Without locks there would be interleaving of operations from two transactions, causing overwrites and inconsistencies. It has to look like all instructions/operations of one transaction are executed at once, while ensuring other external instructions/operations are not executing at the same time. This is especially important in real-world applications such as banking applications.

15.2.1 ACID: Most transaction systems will provide the following properties

- Atomicity: All or nothing (all instructions execute or none; cannot have some instructions execute and some do not)
- Consistency: Transaction takes system from one consistent (valid) state to another
- Isolated: Transaction executes as if it is the only transaction in the system. Ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed sequentially (serializability). In reality, there still will be interleaving of instructions but inconsistent interleaving is prevented.
- Durability: Changes are permanent once transaction completes (commits to disk)

15.2.2 Transaction Primitives

BEGIN_TRANSACTION: Mark the start of a transaction
END_TRANSACTION: Terminate the transaction and try to commit
ABORT_TRANSACTION: Kill the transaction and restore the old values (undo all the effects of the transaction)
READ: Read data from a file, a table, or otherwise

WRITE: Write data to a file, a table, or otherwise

Ex: Airline Reservation

```
BEGIN_TXN
if(reserve(NY, Paris) == full) abort_txn
if(reserve(Paris,Athens) == full) abort_txn
...
END_TXN
```

15.3 Distributed Transactions

Nested Transactions

You can nest multiple smaller child transactions inside a bigger parent transaction. These smaller subtransactions all execute atomically. Each subtransaction is operated on two entirely different (independent) databases.

Distributed Transactions

Database itself is distributed across multiple machines (two physically separated parts of the same database). You execute transactions on this distributed database. Internally, when executing a distributed transaction, the system will have subtransactions that will be executed on physically separate parts of the same database.

15.3.1 Implementation: Private Workspace

Whenever a transaction starts, you make a private copy of the entire database and you will operate on your local copy.

- If you need to abort, just delete the private copy.
- If you want to commit a transaction, you see what changes were made in the private copy and copy the changes to the original database when transaction completes. Changes become visible to everyone and you cannot undo the changes.

In reality, you don't create a full copy of the database every time. You use copy-on-write (lazy) technique.

- Maintain a set of pointers to the same set of underlying data
- When reading data, nothing changes
- When original data or your local copy changes, you start actual copying
- Defer making a copy until the data starts diverging between copies
- You only copy by blocks or record by record

Question: Should you defragment data because you are moving it over?
Answer: If you have random access, there is no fragmentation since you are accessing fixed size blocks, which will not cause a performance overhead.

Question: Is the data in memory or disk?
Answer: All data is reflected (write through) onto disk since it is a database and you want to ensure ACID properties.

Question: Will you see consistency issues if you are making changes to private workspaces and the process crashes during write?
Answer: If you have a problem while you are making changes to private workspace, you simple abort transaction and the original data is not changed.

Question: Can you make private workspace the original?
Answer: You can but it depends on whether or not another transaction has committed recently that would change the original.

Question: If you have a nested transaction, how will you ensure that if you commit a sub transaction, you don't roll it back?
Answer: The commit of the subtransaction and the main transaction all have to happen at the same time. If you commit a subtransaction while other subtransactions are going to execute, you can't undo it so you have to commit all the subtransactions when the main transaction commits.

15.3.2 Implementation: Write-ahead Log

No actual full copy of database is made as changes are made directly to the main original database.

- There is a main database and a log file. The log file tracks every change you make.
- In-place updates: transaction makes changes directly to all original files/objects
- Write-ahead log: prior to making a change, transaction writes to log on stable storage
- Use logs to undo changes or rerun transactions
- If you want to commit, you just write commit entry to log stating that this transaction is committed.

15.4 Concurrency Control

Both write-ahead log and private workspace can be used in distributed transactions but the two by themselves are not enough to ensure ACID properties.

You want high concurrency in your database by having multiple transactions execute concurrently. However, you don't want to have a global lock of the entire database since queries will be queued and throughput will be low. Locking should only happen if two transactions are accessing the same data. One can achieve atomicity, consistency, and correctness by ensuring data items are accessed in a specific order.

Concurrency control can be implemented in a layered fashion.

How does single database concurrency control work internally?
Transaction managers track each transaction in progress. Scheduler grabs fine-grained locks (for individual records or fields) or use timestamps. Data manager will then read and write data either to workspace or actual database after acquiring a lock from scheduler.

How does distributed concurrency control work?
There is still one transaction manager. Each disk for the distributed database will have its own scheduler

that will lock the individual records of the data on the disk. Multiple schedulers and data managers will communicate with each other to figure out the ordering of lock acquisition and read/writes of data.

Question: Is distributed database sharded or replicated?
Answer:
Sharded: Take all the data and partition it into chunks and keep chunks on different disks. Distributed database falls into this category.
Replicated: Keep N copies of the same database

15.4.1 Serializability

There can be multiple scenarios of the interleaving of instructions from different transactions. You have to identify which scenario is valid or invalid.

You know if a scenario is valid if there is at least some sequential execution of transactions that results in one of the known valid states.

You know if a scenario is invalid if there is no sequential execution of transactions that results in the one of the known valid states. Thus, you will reject it as being ILLEGAL or that there is a write-write conflict and the transaction will then have to be aborted.

You want to execute concurrently but prevent ILLEGAL scenarios. Scenarios with arbitrary interleavings will be accepted as being correct if they produce a known valid state.

15.4.2 Optimistic Concurrency Control

The goal of optimistic concurrency control is to maximize the number of transactions executed in parallel to achieve higher throughput but at the same time avoid scenarios that violate serializability.

Implementation:

- Run database without use of locks
- Assume most transactions execute on mutually exclusive parts of the database
- When conflicts arise, abort transaction and redo

This type of concurrency control works well most of the time since conflicts are rare. It IS also easily implemented using private workspaces. However, most relational DBs do not use optimistic concurrency control, but instead strive for correctness rather than performance.

Advantages:

- No deadlocks!
- Maximum parallelism

Disadvantages:

- Rerun transaction if aborts
- Probability of conflict rises at high loads
- Goodput starts to fall due to more aborts

15.4.3 Two-phase Locking (Pessimistic Concurrency Control)

In two-phase locking, the scheduler acquires all necessary locks gradually in a growing phase, and releases locks gradually in shrinking phase. Whenever you want to do an operation on a data item x, you see if x conflicts with existing locks. If so, the transaction is delayed and you must wait for lock to be released. If not, you are granted a lock on x to perform operations on.

The lock is released when the data manager finishes performing all operations on data item x for a transaction. There should be no more instructions that use x in that transaction after releasing the lock. Once a lock is released, no further locks can be granted for x. Cannot acquire and release a lock on a data item multiple times for multiple operations.

Deadlock is possible if you acquire two locks in different order:

TXN 1

---

begin_txn
Lock(x)
Read(x)
Read(y) $\rightarrow Blocked...waiting\, for\, TXN\, 2\, to\, release\, lock\, on\, y$
$Write(x) \rightarrow Another\, operation\, on\, x$
...
$Release(x)$
end_txn

TXN 2

---

begin_txn
Lock(y)
Read(y)
Read(x) $\rightarrow Blocked...waiting\, for\, TXN\, 1\, to\, release\, lock\, on\, x$
$Write(y) \rightarrow Another\, operation\, on\, y$
...
$Release(y)$
end_txn

NOTE: In the context of two-phase locking, TXN 1 and TXN 2 cannot release locks after their first reads because a lock cannot be released until all operations on the data item are performed.

Strict two-phase locking: A variant of two-phase locking where all locks are released at once in the end.

How do you resolve deadlocks?
You can number your data items 1 through N and acquire locks in monotonically increasing order. For example, if you need a lock on records 1 and 2, you first acquire a lock on 1 before you acquire a lock on 2. Therefore, you scan operations in a transaction and number and order all data items used. Then, you acquire locks corresponsing to these data items in increasing order. This will ensure that no deadlock cycles happen while operating on the data items from multiple transactions at the same time.

Question: Is it possible you can break logic by numbering resources?
Answer: If you acquire all locks in the beginning and release them at the end, it will not break the logic.

15.4.4 Timestamp-based Concurrency Control (Pessimistic)

Assume every transaction is an event and associate a logical clock with it. Increment timestamp and whenever multiple transactions read or write data, use logical timestamps to figure out if the conflict is a read-write conflict ora write-write conflict

- Read-write conflict: out of date read value due to previous write
- Write-write conflict: overwriting previous update

How to avoid conflict? For every value in the database, keep the last read and write timestamp. Details of the algorithm are in the slides.