Leqian Cai
emmacai507@gmail.com

Assignment 3

3.1 Optimistic lock and pessimistic lock

The optimistic locking model, also referred to as optimistic concurrency control, is a concurrency control method used in relational databases that does not use record locking. Optimistic locking allows multiple users to attempt to update the same record without informing the users that others are also attempting to update the record. The record changes are validated only when the record is committed. If one user successfully updates the record, the other users attempting to commit their concurrent updates are informed that a conflict exists.
data conflicts must be saved and manually merged.

Pessimistic Locking is when you **lock the record for your exclusive use** until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks.

In conclusion, Optimistic locking, where a record is locked only when changes are committed to the database. Optimistic assumes that nothing's going to change while you're reading it. Pessimistic locking, where a record is locked while it is edited.

In both data-locking models, the lock is released after the changes are committed to the database.

3.2 How to solve the deadlock?

One strategy is **timeout.** If the scheduler finds that a transaction has been waiting too long for a lock, then it simply guesses that there may be a deadlock involving this transaction and therefore aborts it. Since the scheduler is only guessing that a transaction may be involved in a deadlock, it may be making a mistake. It may abort a transaction that isn't really part of a deadlock but is just waiting for a lock owned by another transaction that is taking a long time to finish. There's no harm done by making such an incorrect guess, insofar as correctness is concerned. There is certainly a performance penalty to the transaction that was unfairly aborted.

Another approach to deadlocks is to **detect them precisely**. To do this, the scheduler maintains a directed graph called a waits-for graph (WFG). When the scheduler discovers a deadlock, it must break the deadlock by aborting a transaction. The Abort will in turn delete the transaction's node from the WFG. The transaction that it chooses to abort is called the **victim**. Among the transactions involved in a deadlock cycle in WFG, the scheduler should select a victim whose abortion costs the least.

It is possible to construct a 2PL scheduler that never aborts transactions. This technique is known as **Conservative 2PL** or **Static 2X**. As we have seen, 2PL causes abortions because of deadlocks. Conservative 2PL avoids deadlocks by requiring each transaction to obtain all its locks before any of its operations are submitted to the DM.

Leqian Cai
emmacai507@gmail.com

Wait-Die Scheme vs. Wound-Wait Scheme

In Wait-Die scheme, if a transaction requests a resource that is locked by another transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.
This scheme allows the older transaction to "wait" but kills the younger one ("die").

Wound-Wait scheme allows the younger transaction requesting a lock to "wait" if the older transaction already holds a lock but **forces the younger one to be suspended ("wound")** if the older transaction requests a lock on an item already held by the younger one.

In both the cases, only the transaction that enters the system at a later timestamp (i.e. **the younger transaction**) might be killed and restarted.

3.3 Saga design pattern

The Saga design pattern is a way to **manage data consistency** across microservices in distributed transaction scenarios. A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, the saga executes **compensating transactions** that counteract the preceding transactions.

The Saga pattern may initially be challenging, as it requires a new way of thinking on how to coordinate a transaction and maintain data consistency for a business process spanning multiple microservices.

The Saga pattern is particularly **hard to debug**, and the complexity grows as participants increase.

Data can't be rolled back because saga participants commit changes to their local databases.

The implementation must be capable of handling a set of potential transient failures and provide idempotence for reducing side-effects and ensuring data consistency. **Idempotence** means that the same operation can be repeated multiple times without changing the initial result.

It's best to implement observability to monitor and track the saga workflow.

The lack of participant data isolation **imposes durability challenges**. The saga implementation must include countermeasures to reduce anomalies.

Leqian Cai
emmacai507@gmail.com

References

https://www.ibm.com/docs/en/rational-clearquest/7.1.0?topic=clearquest-optimistic-pessimistic-record-locking

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/05/chapter3.pdf

https://stackoverflow.com/questions/32794142/what-is-the-difference-between-wait-die-and-wound-wait-deadlock-prevention-a

https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga