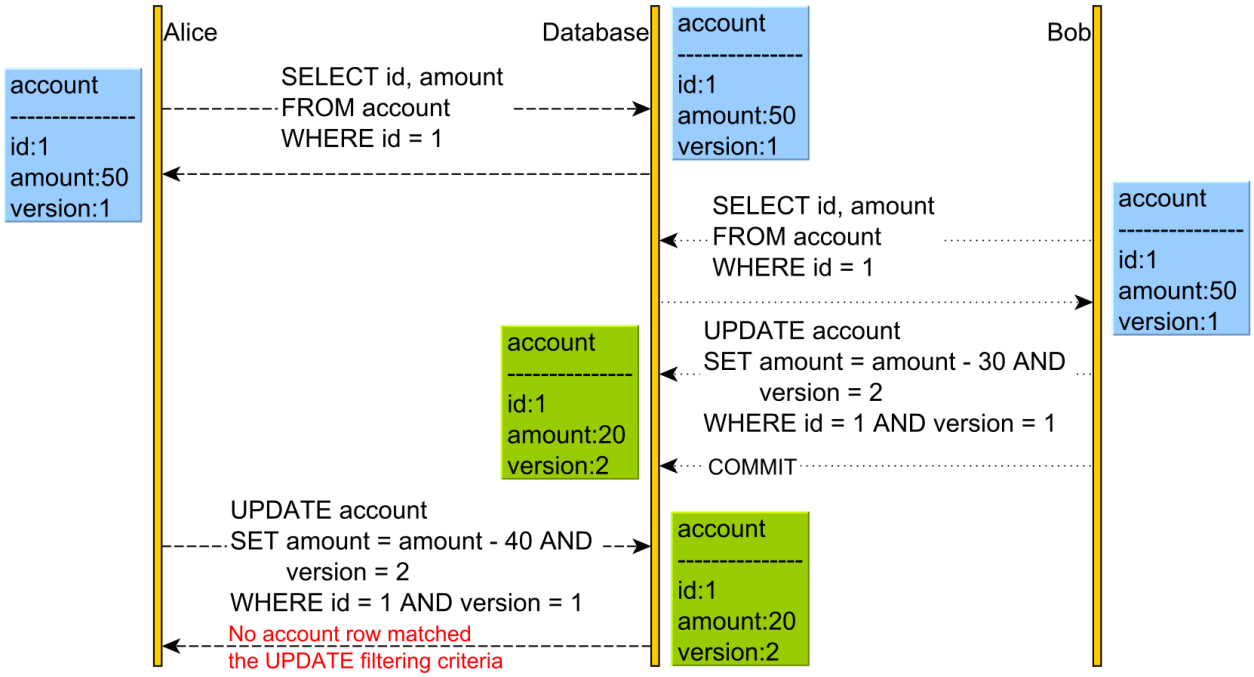
Homework 3.1

Optimistic lock and Pessimistic lock

Optimistic lock and Pessimistic lock are both concurrency control strategies for the database. To solve the Read Committed isolated level (cause conflict), Optimistic lock allows the conflict, but it needs to detect it at write time with either a physical or logical clock. Since logical clock is superior to physical clock when implementing a concurrency control mechanism, we can add a column named version(Other methods to do this involve dates, timestamps or checksum/hashers) to capture the read-time row snapshot information. The version column is going to incremented everything when UPDATE or DELETE statement is executed while also being used for matching the expected id and version int that row in the WHERE clause.



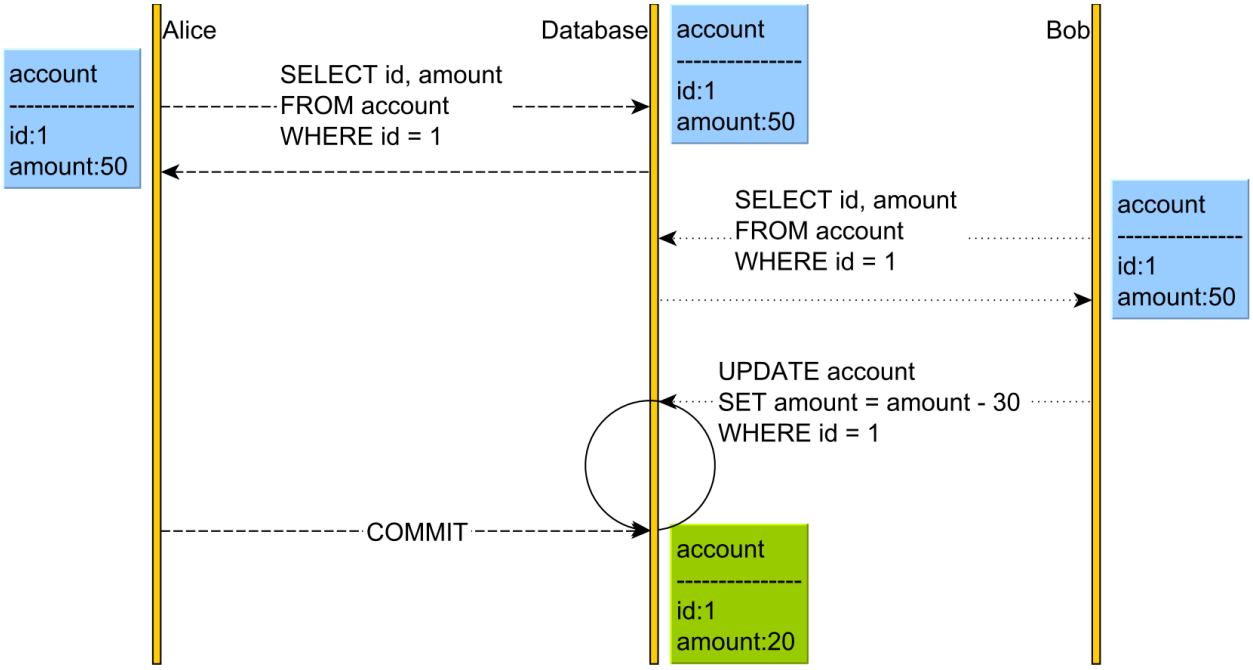
Alice reading the database then bob started reading too, the version wouldn't change, since only UPDATE or DELETE will incremented the version number, so far, both version numbers are 1.

Bob start UPDATE the account and SET the amount and version number to 2. After Alice want to UPDATE the account balance, however the version number doesn't match when Alice still try to access the version 1.

Therefor, the executeUpdate method of the UPDATE PreparedStatement is going to return value of 0 (no record changed), and underlying data access framework will throw an OptimisticlockException that will cause Alice's transaction to roll back.

Nowadays, many relational database systems use optimistic locking to provide ACID guarantees. Oracle, PostgreSQL, and the InnoDB MySQL engine use MVCC (Multi-Version Concurrency Control), which is based on optimistic locking.

For Pessimistic locking aims to avoid conflicts by using locking.



In the diagram above, both Alice and Bob will acquire a read (shared) lock on the account. Both Alice and Bob are holding the read lock on the account record with id of 1. Neither of them can change until one release the read lock they acquired. This is because a write operation required a write (exclusive) lock acquisition and read (shared) locks prevent write (exclusive) locks.

For this reason, Bob's UPDATE blocks until Alice releases the shared lock she has acquired previously.

When using SQL Server, the database acquires the shared locks automatically when reading a record under Repeatable Read or Serializable isolation level because SQL uses the 2PL (two-Phase Locking) algorithm by default.

MySQL also uses pessimistic locking by default when using the serializable isolation level and optimistic locking for the other less-strict isolation levels.

Summary:

Optimistic locking, where a record is locked only when changes are committed to the database, and Pessimistic locking, where a record is locked while it is edited.

An advantage of OL is that it avoids the overhead of locking a record for the duration of the action. If there are no concurrency, then this model provides fast updates. Most applicable for high-volume systems and three-tier architectures where you do not necessarily maintain a connection the database for your session. In this situation the client cannot maintain the database locks as the connections are taken from a pool and may not be using the same connection from one access to the next.

Pessimistic locking might be more suitable when conflicts happen frequently, as it reduces the chance of rolling back transactions.

For PL need to avoid the Deadlocks. To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.

Source:

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

https://vladmihalcea.com/optimistic-vs-pessimistic-locking/

Homework 3.2

How to solve the deadlock

The databases engine runs a separate process that scans the current conflict graph for lock-wait cycles (which are caused by deadlocks). When a cycle is detected, the engine picks one transaction and aborts it, and causing its locks to be released, so that other transaction can make progress.

Unlike the JVM, a database transaction is designed as an atomic unit of work.

We can try to prevent a deadlock.

Wait-Die Scheme-

DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution. DMBS will first check the timestamp. If a younger transaction has locked some resource and an older transaction is waiting for it, then an older transaction is allowed to wait for it till it is available. If the older transaction has held some resource and younger transaction waits for the resource, the the younger transaction is killed and restarted with a very minute delay with same timestamp.

Wound-wait Scheme-

If an older transaction requests for a resource held by a younger transaction, then an older transaction forces a younger transaction to kill the transaction and release the resource. The younger transaction is restarted with a minute delay but with the same timestamp. If the younger transaction is requestion a resource that is held by an older one, then the younger transaction is asked to wait till the order releases it.

Check the system_health session for deadlocks

Create an extended event session to capture the deadlocks

Analyze the deadlock reports and graphs to figure out the problem

If it is possible to make improvements or changing the queries involved in the dead lock

Sources:

https://www.ibm.com/docs/en/db2/11.5?topic=problem-resolving-deadlock-problems

https://www.geeksforgeeks.org/deadlock-in-dbms/

https://www.mssqltips.com/sqlservertip/5100/lesson-on-sql-server-deadlocks-and-how-to-solve/

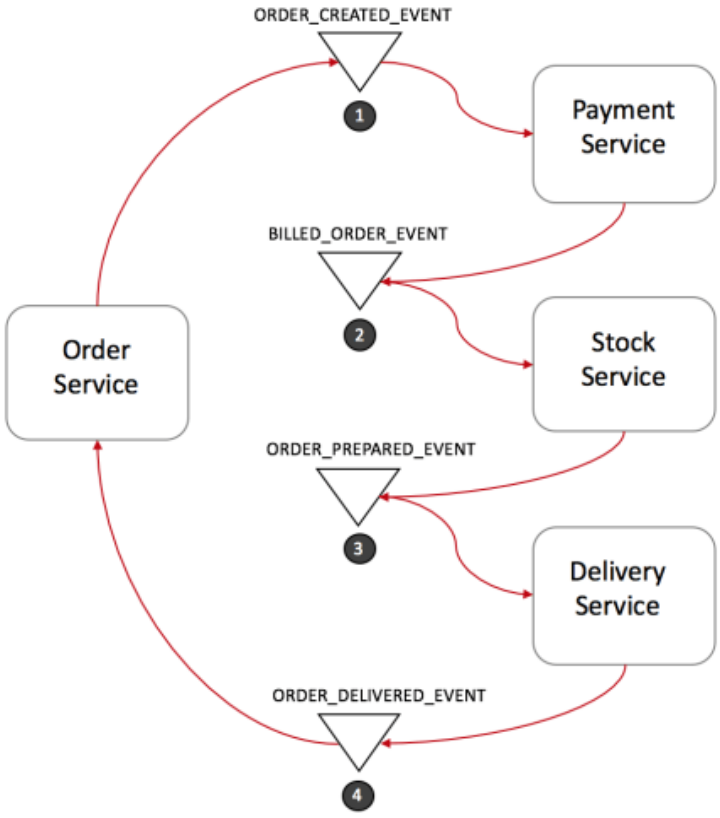
Homework 3.3

Saga

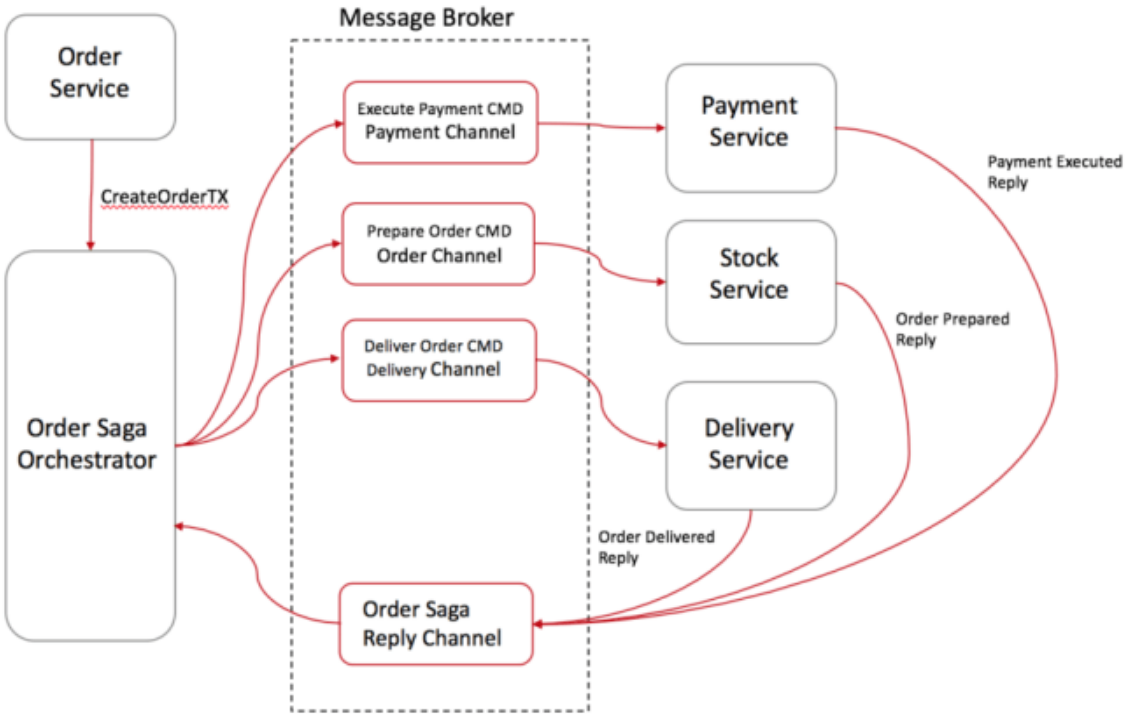
The Saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios. Saga is a sequence of local transactions where each transaction updates data within single service. The first transaction in a saga is initiated by an external request corresponding to the system operation, and then each subsequent step is triggered by the completion of the previous one.

Two most popular to implement a saga transaction are:

Events/Choreography: When there is no central coordination, each service produces and listen to other service's events and decides if an action should be taken or not.



Command/Orchestration: When a coordinator service is responsible for centralizing the saga's decision making and sequencing business logic.



Source:

https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/

https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part-2/

https://github.com/bertilmuth/requirementsascode

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