**Concurrency Control**
Optimistic vs Pessimistic
Optimistic Locking

- is a concurrency control
    - begin: record a timestamp of transaction
    - modify: read data, tentatively write changes
    - Validate: Check whether other transactions have modified data that this transaction has used (read or written). This includes transactions that completed after this transaction's start time, and optionally, transactions that are still active at validation time.
    - Commit/Rollback: If there is no conflict, make all changes take effect. If there is a conflict, resolve it, typically by aborting the transaction

Pessimistic Locking

- a resource is locked from the time it is first accessed in a transaction until the transaction is finished, that resource is inaccessible for other transactions during that time.
    - disadvantages: may cause deadlock situations
        - deadlock: all resources are waiting for each other.

guidelines for locking.

- Try not to hold locks across long operations like I/O where performance can be adversely affected.
- Don't hold locks when calling a function that is outside the module and that might reenter the module.
- In general, start with a coarse-grained approach, identify bottlenecks, and add finer-grained locking where necessary to alleviate the bottlenecks. Most locks are held for short amounts of time and contention is rare, so fix only those locks that have measured contention.
- When using multiple locks, avoid deadlocks by making sure that all threads acquire the locks in the same order.

**Summarize Transactions in database**

Transactions mean a series of SQL operations in the database that follow the ACID properties

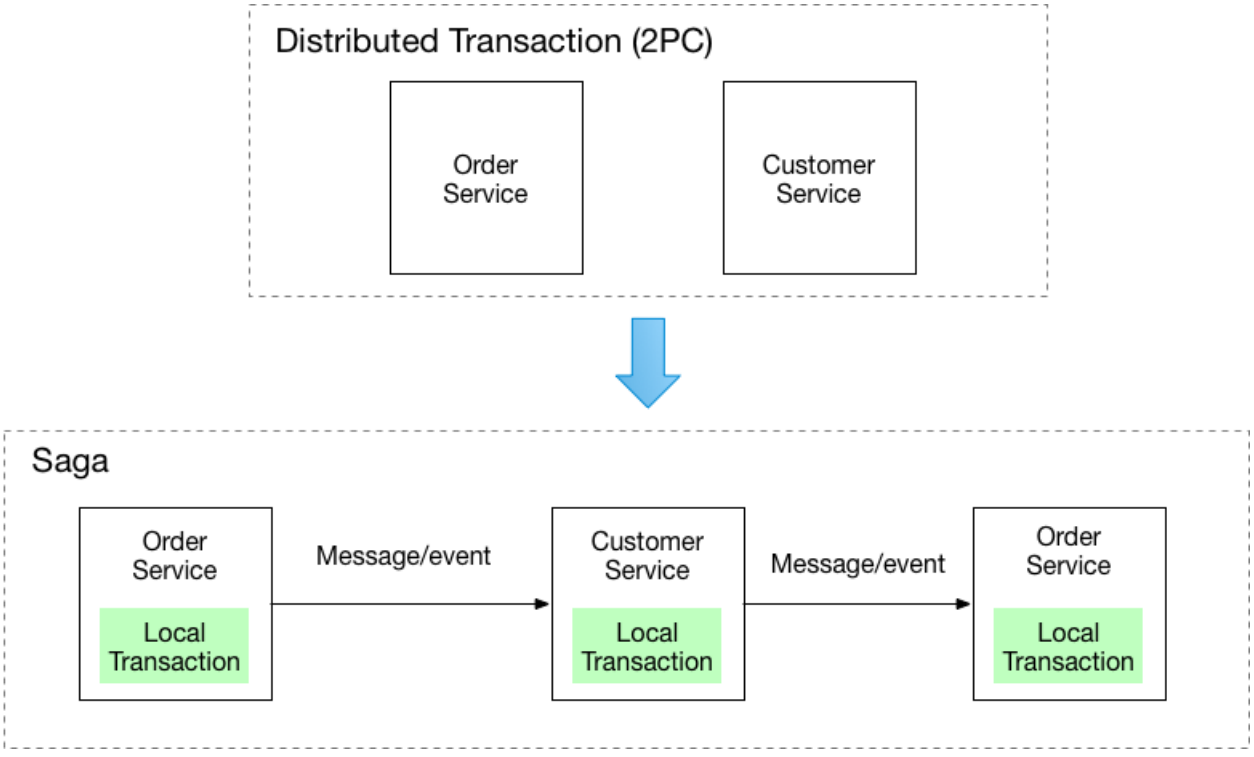
ACID

- Atomicity − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

- Consistency − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- Durability − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- Isolation − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

**SAGA Design Pattern**

Saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios.  A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



There are two ways of coordination sagas:

- Choreography - each local transaction publishes domain events that trigger local transactions in other services
- Orchestration - an orchestrator (object) tells the participants what local transactions to execute

This pattern has the following benefits:

- It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

- The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

**Two-Phase Commit**

The two-phase commit protocol breaks a database commit into two phases to ensure correctness and fault tolerance in a distributed database system.

Prepare phase

- After each database store (slave) has locally completed its transaction, it sends a “DONE” message to the transaction coordinator. Once the coordinator receives this message from all the slaves, it sends them a “PREPARE” message.
- Each slave responds to the “PREPARE” message by sending a “READY” message back to the coordinator.
- If a slave responds with a “NOT READY” message or does not respond at all, then the coordinator sends a global “ABORT” message to all the other slaves. Only upon receiving an acknowledgment from all the slaves that the transaction has been aborted does the coordinator consider the entire transaction aborted.

Commit phase

- Once the transaction coordinator has received the “READY” message from all the slaves, it sends a “COMMIT” message to all of them, which contains the details of the transaction that needs to be stored in the databases.
- Each slave applies the transaction and returns a “DONE” acknowledgment message back to the coordinator.
- The coordinator considers the entire transaction to be completed once it receives a “DONE” message from all the slaves.

Advantage

The protocol makes the data consistent and available, either all the databases get an update or none do. This protocol ensures that the databases are always synchronized.

Disadvantage

The two-phase commit is a blocking protocol; the failure of a single node blocks progress until the node recovers. Moreover, if the transaction coordinator fails, then the database is left in an inconsistent state and only recovers once the coordinator recovers. This leads to another drawback as the protocol's latency depends on the slowest node. Since it waits for all the nodes to send acknowledgment messages, a single slow node will slow down the entire transaction.