**Monolithic Application:** Tightly coupled

A big container wherein all the software components of an application are assembled together and tightly packaged (can take various formats such as EAR, WAR, JAR etc.) which is **finally deployed as a single unit on the application server**.

In business terms, all different business services are packed together as a single unit

Easy deployment process: Since entire application is packaged as a single unit

Source Code for entire application lies at a single location and navigating through the code base is pretty easy by using IDE.

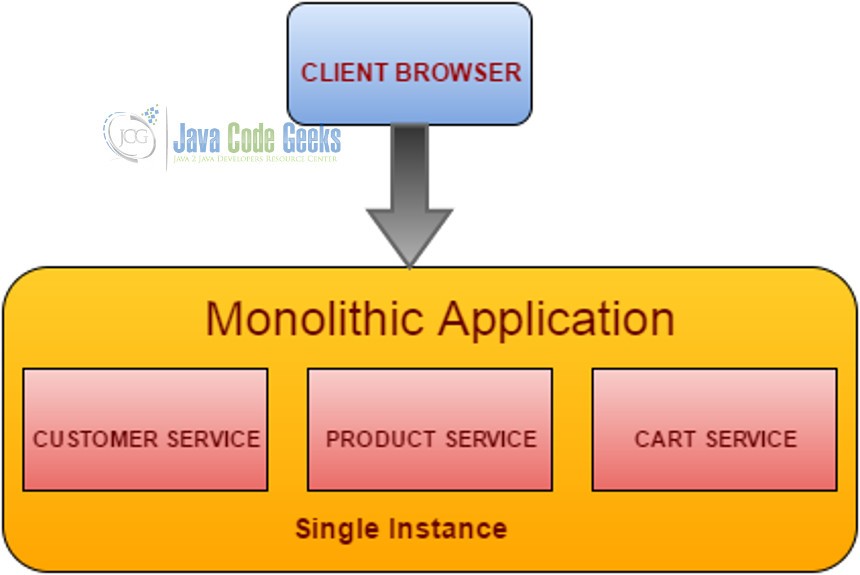
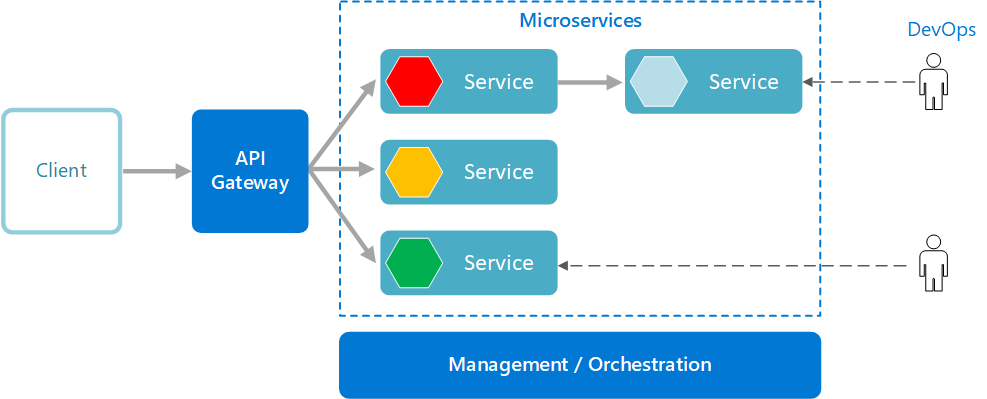
You can directly create a deployment unit (WAR , JAR or EAR) from IDE itself.

Rebuilding the whole application takes a lot of time. Even if a small component in an application has to be changed, the entire application needs to be repackaged and assembled together.

Individual component scalability is not possible: If we scale the application as a whole resources utilized in other services is getting wasted since they don’t have a need to scale.

Even for a small change in the application, build time increases tremendously and it definitely decreases frequency of Deployments and hence devops cycle is longer

Single point of failure: There are many more parameters such as Availability, Fault Tolerance, and Resiliency on which a monolithic application fails miserably.

**Microservices:**

Microservices are a **popular architectural style** for building applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability.

A screenshot of a cell phone

Description automatically generated

**Management/orchestration.** Responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth. Typically, this component is an off-the-shelf technology such as Kubernetes, rather than something custom built.

**API Gateway.** The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.

Advantages of using an API gateway include:

* It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.
* Services can use messaging protocols that are not web friendly, such as AMQP.
* The API Gateway can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing

**++++. **

Agility. Since deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application and roll back an update if something goes wrong.

In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features may be held up waiting for a bug fix to be integrated, tested, and published.

Small code base. In a monolithic application, adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.

Mix of technologies. Teams can pick the technology that best fits their service

Fault isolation. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).

Scalability. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application.

Data isolation. It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

Availability: Even if one service fails, other micro services are highly available, and the failed micro service can be rectified very quickly with as minimal downtime as well.

CD: Release of one component does not impact other components. deployment times reduces tremendously

**Challenges**

**Network congestion and latency.** The use of many small, granular services can result in more interservice communication through remote calls.These remote calls result in higher costs associated with network latency and processing than with traditional architectures.

**Data integrity.** With each microservice responsible for its own data persistence,data consistency can be a challenge. Embrace eventual consistency where possible.

**Versioning.** Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.

**Expensive.** Each service will require its own runtime environment, databases and CPU. This is a requirement to keep each instance isolated.

**Decreased Performance.** Each service runs as an independent process (multiple jvm instances) as compared to single shared process (single jvm instance) in monolithic. Hence microservices performance is slightly less compare to monolithic application.

**Security Challenges**. pose enormous security challenges due to the increases in inter-service communication over the network.

**Complexity.** Each service independent and simpler, but the entire system as a whole is more complex.

**Development and testing**. Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.

**Lack of governance.** The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain.

A structured approach for designing, building, and operating a microservices architecture.

**Choosing an compute option for microservices**

The term compute refers to the hosting model for the computing resources that your application runs on. For a microservices architecture, two approaches are especially popular:

* A service orchestrator that manages services running on dedicated nodes (VMs).
* A serverless architecture using functions as a service (FaaS).

An orchestrator handles tasks related to deploying and managing a set of services. These tasks include placing services on nodes, monitoring the health of services, restarting unhealthy services, load balancing network traffic across service instances, service discovery, scaling the number of instances of a service, and applying configuration updates. Popular orchestrators include Kubernetes, Service Fabric, DC/OS, and Docker Swarm.

**Designing interservice communication for microservices**

Communication between microservices must be efficient and robust. With lots of small services interacting to complete a single transaction, this can be a challenge.

1. **Resiliency.** There may be many instances of any given microservice. An instance can fail for any number of reasons.

* Node-level failure such as a hardware failure
* A VM reboot.
* An instance might crash or be overwhelmed with requests and unable to process any new requests.

Any of these events can cause a network call to fail.

There are two design patterns that can help make service-to-service network calls more resilient:

* 1. [Retry](https://docs.microsoft.com/en-us/azure/architecture/patterns/retry).
  2. A network call may fail because of a transient fault that goes away by itself. Rather than fail outright, the caller should typically retry the operation a certain number of times, or until a configured time-out period elapses. However, if an operation is not idempotent, retries can cause unintended side effects. The original call might succeed, but the caller never gets a response. If the caller retries, the operation may be invoked twice. Generally, it's not safe to retry POST or PATCH methods, because these are not guaranteed to be idempotent.
  3. [Circuit Breaker](https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker).

Too many failed requests can cause a bottleneck, as pending requests accumulate in the queue. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on, which can cause cascading failures. The Circuit Breaker pattern can prevent a service from repeatedly trying an operation that is likely to fail.

1. **Load balancing.** When service "A" calls service "B", the request must reach a running instance of service "B". In Kubernetes, the Service resource type provides a stable IP address for a group of pods. By default, a random pod is chosen. We should have intelligent load balancing algorithms based on observed latency or other metrics.
2. **Distributed tracing.** A single transaction may span multiple services. That can make it hard to monitor the overall performance and health of the system. Even if every service generates logs and metrics, without some way to tie them together, they are of limited use.
3. **Service versioning.** When a team deploys a new version of a service, they must avoid breaking any other services or external clients that depend on it. In addition, you might want to run multiple versions of a service side-by-side, and route requests to a particular version.
4. **TLS encryption and mutual TLS authentication.** For security reasons, you may want to encrypt traffic between services with TLS, and use mutual TLS authentication to authenticate callers.

There are two basic messaging patterns that microservices can use to communicate with other microservices.

**Synchronous versus asynchronous messaging**

**Synchronous communication.** In this pattern, a service calls an API that another service exposes, using a protocol such as HTTP. This option is a synchronous messaging pattern because the caller waits for a response from the receiver.

**Asynchronous message passing.** The client sends a request to the server (which requires lengthy processing) and receives a delivery acknowledgment right away. Different from the synchronous communication, this response does not have the required information, yet.

* After the client receives the acknowledgment, it continues to do its other tasks, assuming it will eventually be notified when the required information is ready on the server side.
* The biggest benefit of asynchronous communication is the increased performance. Since the client does not block its valuable CPU cycles just for waiting, it can deliver more within the same timeframe. Increased decoupling between the client-server interaction will also lead to better scalability.

**Asynchronous I/O** **Asynchronous protocol**

The calling thread is not blocked while the I/O completes. the sender doesn't wait for a response.

That's important for performance

Note: HTTP is a synchronous protocol, even though an HTTP client may use asynchronous I/O when it sends a request.

Note: HTTP is a synchronous protocol, even though an HTTP client may use asynchronous I/O when it sends a request.

There are three methods to implement an asynchronous communication:

* Asynchronous Callbacks
* Using Pub-Sub messaging using a Message Broker (or MoM)
* Polling for State Changes

There are tradeoffs to each pattern. Request/response is a well-understood paradigm, so designing an API may feel more natural than designing a messaging system. However, asynchronous messaging has some advantages that can be useful in a microservices architecture:

**A close up of text on a white background

Description automatically generated** A screenshot of a cell phone

Description automatically generated

++++ Async

Reduced coupling. The message sender does not need to know about the consumer.

Multiple subscribers. Using a pub/sub model, multiple consumers can subscribe to receive events. example [Event-driven architecture style](https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven).

Failure isolation. If the consumer fails, the sender can still send messages. The messages will be picked up when the consumer recovers. This ability is especially useful in a microservices architecture, because each service has its own lifecycle. A service could become unavailable or be replaced with a newer version at any given time. **Asynchronous messaging can handle intermittent downtime.** Synchronous APIs, on the other hand, require the downstream service to be available or the operation fails.

Responsiveness. An upstream service can reply faster if it does not wait on downstream services. This is especially useful in a microservices architecture. If there is a chain of service dependencies (service A calls B, which calls C, and so on), waiting on synchronous calls can add unacceptable amounts of latency.

Load leveling. A queue can act as a buffer to level the workload, so that receivers can process messages at their own rate.

---- Challenges Async

Coupling with the messaging infrastructure. Using a particular messaging infrastructure may cause tight coupling with that infrastructure. It will be difficult to switch to another messaging infrastructure later.

Latency. End-to-end latency for an operation may become high if the message queues fill up.

Cost. At high throughputs, the monetary cost of the messaging infrastructure could be significant.

Complexity. Handling asynchronous messaging is not a trivial task. For example, you must handle duplicated messages, either by de-duplicating or by making operations idempotent.

Throughput. If messages require queue semantics, the queue can become a bottleneck in the system. If the queue is a managed service, there may be additional latency, because the queue is external to the cluster's virtual network. You can mitigate these issues by batching messages, but that complicates the code.

**Service mesh: is a software layer that handles service-to-service communication.** Service meshes are designed to move responsibility for these concerns away from the microservices themselves and into a shared layer. The service mesh acts as a proxy that intercepts network communication between microservices in the cluster.

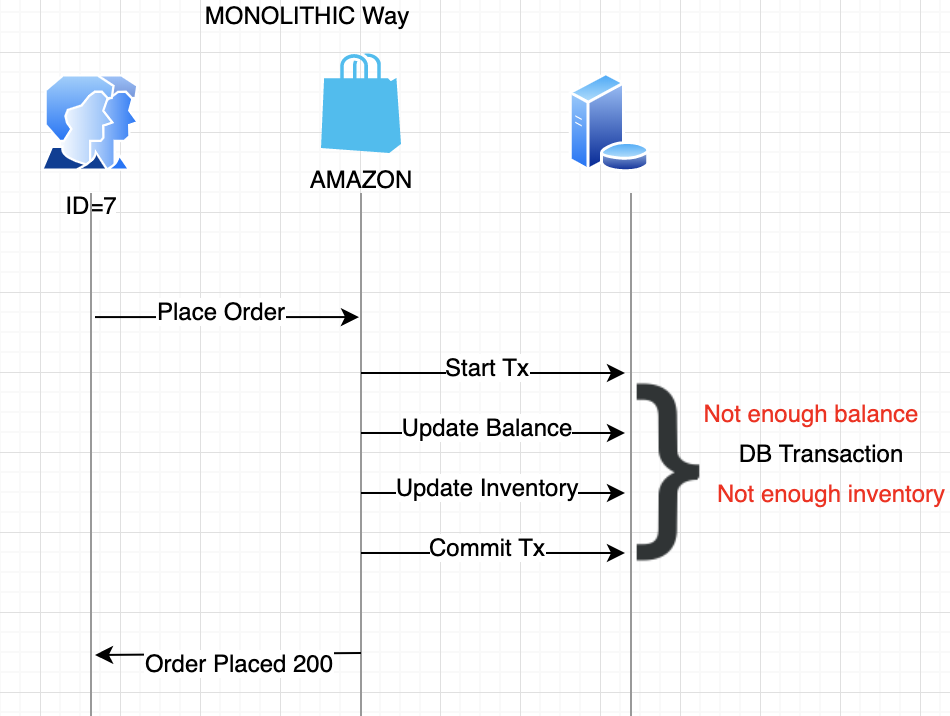
Kubernetes uses [linkerd](https://linkerd.io/) and [Istio](https://istio.io/) for SM.

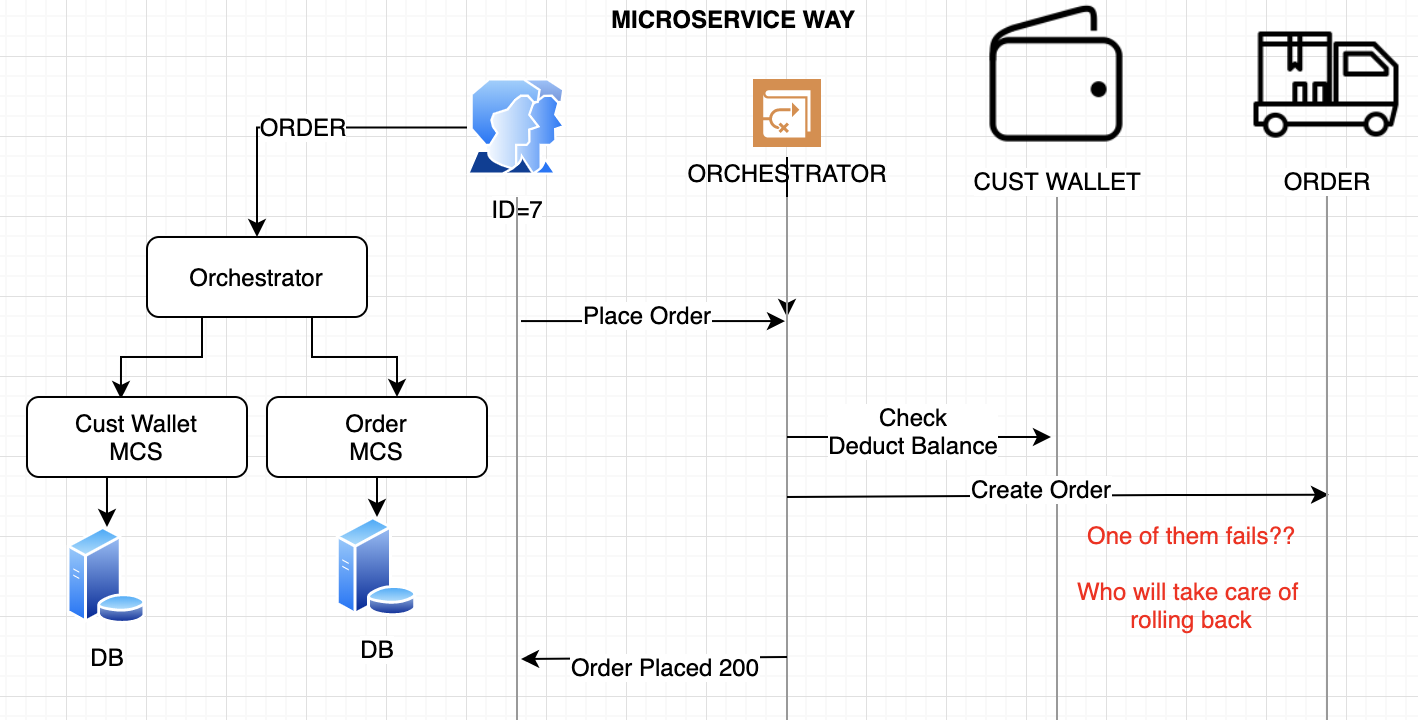
* Based on observed latencies or number of outstanding requests SM can improve performance over the layer-4 load balancing that is provided by Kubernetes.
* Layer-7 routing based on URL path, Host header, API version, or other application-level rules.
* Retry of failed requests. A service mesh understands HTTP error codes, and can automatically retry failed requests. You can configure that maximum number of retries, along with a timeout period in order to bound the maximum latency.
* Circuit breaking. If an instance consistently fails requests, the service mesh will temporarily mark it as unavailable. After a backoff period, it will try the instance again. You can configure the circuit breaker based on various criteria, such as the number of consecutive failures,
* Service mesh captures metrics about interservice calls, such as the request volume, latency, error and success rates, and response sizes. The service mesh also enables distributed tracing by adding correlation information for each hop in a request.
* Mutual TLS Authentication for service-to-service calls.

**Do you need a service mesh?** It depends.

You can solve problems like retry, circuit breaker, and distributed tracing without a service mesh, but a service mesh moves these concerns out of the individual services and into a dedicated layer.

On the other hand, a service mesh adds complexity to the setup and configuration of the cluster. There may be performance implications, because requests now get routed through the service mesh proxy, and because extra services are now running on every node in the cluster. You should do thorough performance and load testing before deploying a service mesh in production.





**The database ensures the integrity of data in a distributed transaction using the two-phase commit mechanism** Unlike a transaction on a local database, a distributed transaction involves altering data on multiple databases. Consequently, distributed transaction processing is more complicated, because the database must coordinate the committing or rolling back of the changes in a transaction as a self-contained unit.

1. Prepare phase, the initiating node(global coordinator) in the transaction asks the other participating nodes to promise to commit or roll back the transaction.
2. Commit phase, the initiating node asks all participating nodes to commit the transaction. If this outcome is not possible, then all nodes are asked to roll back.

All participating nodes in a distributed transaction should perform the same action: they should either all commit or all perform a rollback of the transaction. The database automatically controls and monitors the commit or rollback of a distributed transaction and maintains the integrity of the global database (the collection of databases participating in the transaction) using the two-phase commit mechanism. This mechanism is completely transparent, requiring no programming on the part of the user or application developer.

There needs to be a global coordinator to maintain the lifecycle of the transaction, and the coordinator will need to call the microservices in the prepare and commit phases.

Coordinator will first create a global transaction with all the context information. It will then tell CustomerMicroservice to prepare for updating a customer fund with the created transaction. The CustomerMicroservice will then check, for example, if the customer has enough funds to proceed with the transaction. Once CustomerMicroservice is OK to perform the change, it will lock down the object from further changes and tell the Coordinator that it is prepared. The same thing happens while creating the order in the OrderMicroservice. Once the Coordinator has confirmed all microservices are ready to apply their changes, it will then ask them to apply their changes by requesting a commit with the transaction. At this point, all objects will be unlocked.

Coordinator  should have timeout to make this things properly

If at any point a single microservice fails to prepare, the Coordinator will abort the transaction and begin the rollback process. CustomerMicroservice failed to prepare for some reason, but the OrderMicroservice has replied that it is prepared to create the order. The Coordinator will request an abort on the OrderMicroservice with the transaction and the OrderMicroservice will then roll back any changes made and unlock the database objects.

**++2 pc ++** a very strong consistency protocol.

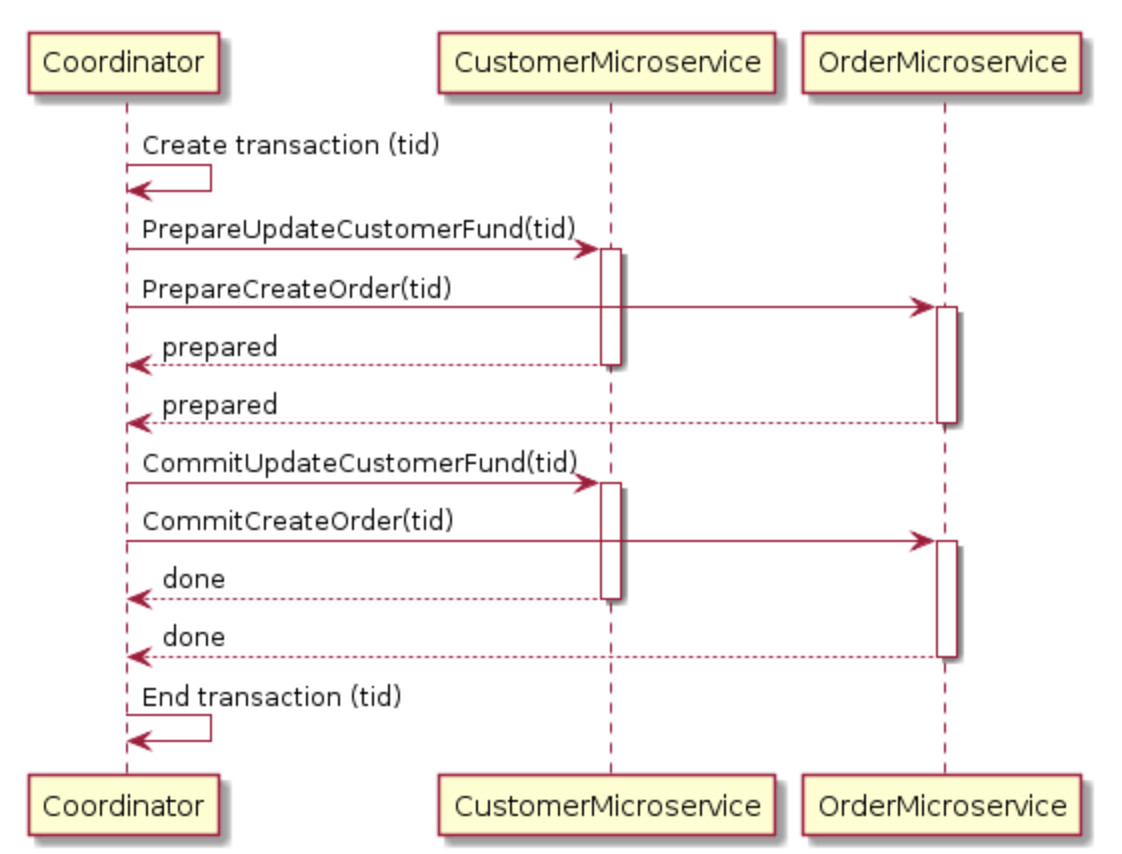
First, the prepare and commit phases guarantee that the transaction is atomic. The transaction will end with either all microservices returning successfully or all microservices have nothing changed.

Secondly, 2pc allows read-write isolation. This means the changes on a field are not visible until the coordinator commits the changes.

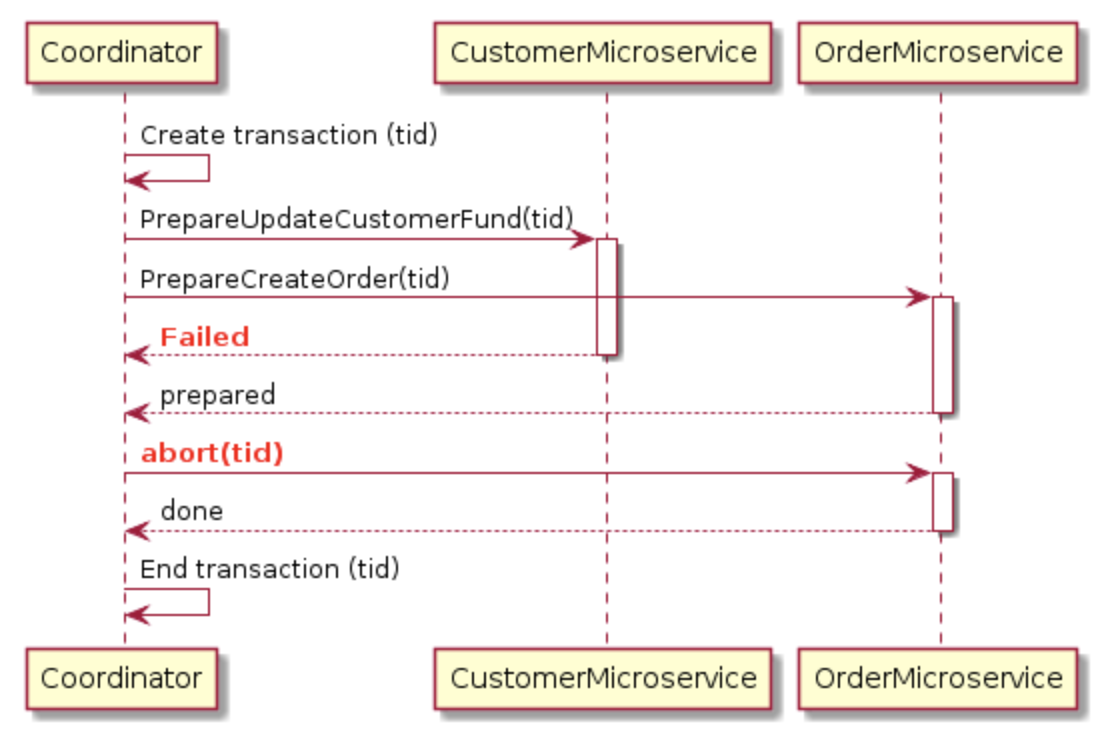
**-- 2pc - -** is synchronous (blocking). The protocol will need to lock the object. In the example above, if a customer places an order, the “fund” field will be locked for the customer. This prevents the customer from applying new orders.

In a database system, transactions tend to be fast—normally within 50 ms. However, microservices have long delays with RPC calls, especially when integrating with external services such as a payment service. The lock could become a system performance bottleneck.

Also, it is possible to have two transactions mutually lock each other (deadlock) when each transaction requests a lock on a resource the other requires.

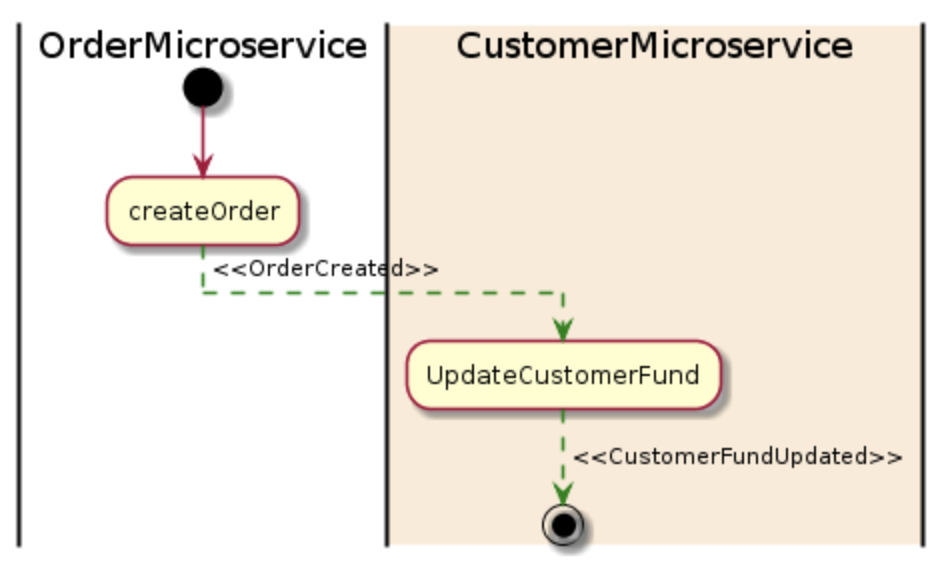


FAILURE

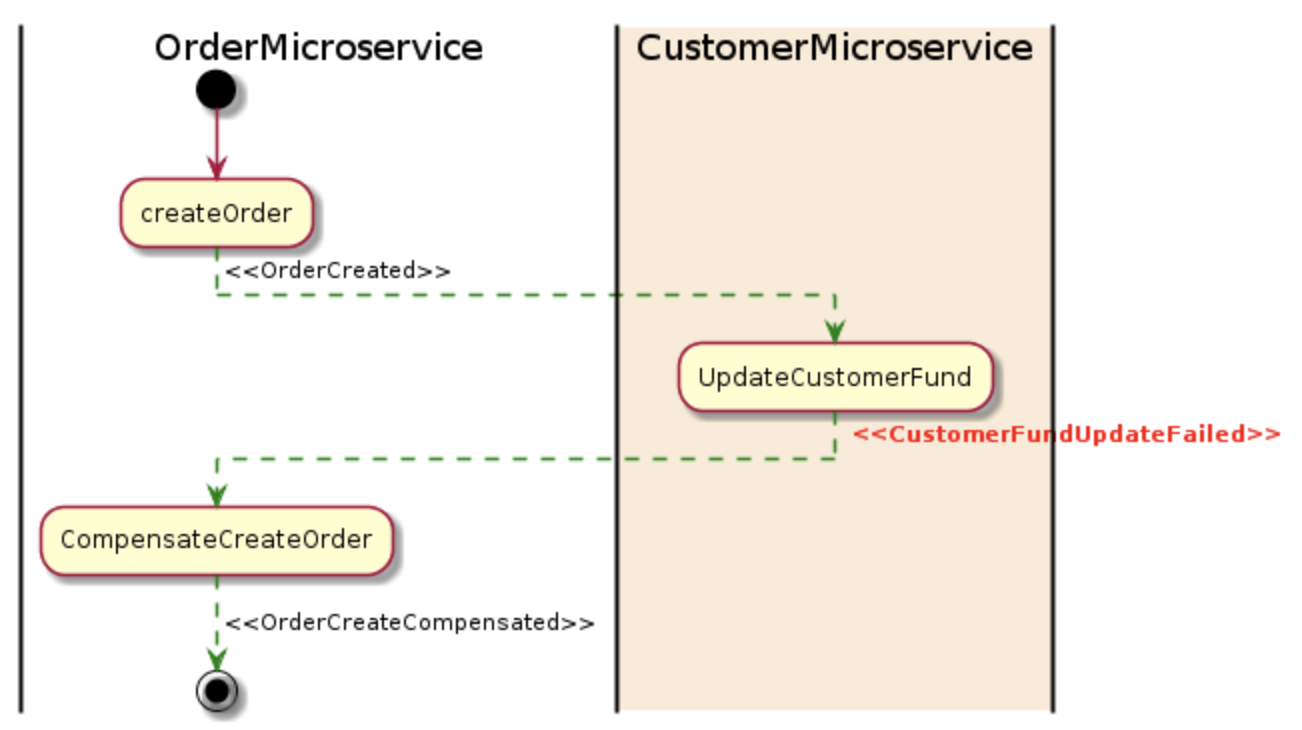


**SAGA Pattern**

The Saga pattern is another widely used pattern for distributed transactions. The Saga pattern is asynchronous and reactive. In a Saga pattern, the distributed transaction is fulfilled by asynchronous local transactions on all related microservices. The microservices communicate with each other through an event bus.



OrderMicroservice receives a request to place an order. It first starts a local transaction to create an order and then emits an OrderCreated event. The CustomerMicroservice listens for this event and updates a customer fund once the event is received. If a deduction is successfully made from a fund, a CustomerFundUpdated event will then be emitted, which means the end of the transaction.



If any microservice fails to complete its local transaction, the other microservices will run compensation transactions to rollback the changes.

**++ Saga pattern ++** Each microservice focuses only on its own local atomic transaction, other microservices are not blocked if a microservice is running for a long time. Also, because all local transactions are happening in parallel, there is no lock on any object.

**Local Transaction (wrt MCS)+ Sequential (Event Bus) = ISOLATION**

#### **-- Saga pattern - -** difficult to debug, especially when many microservices are involved. Also, the event messages could become difficult to maintain if the system gets complex.

#### Another disadvantage of the Saga pattern is it does not have read isolation. For example, the customer could see the order being created, but in the next second, the order is removed due to a compensation transaction.

**Distributed transactions**

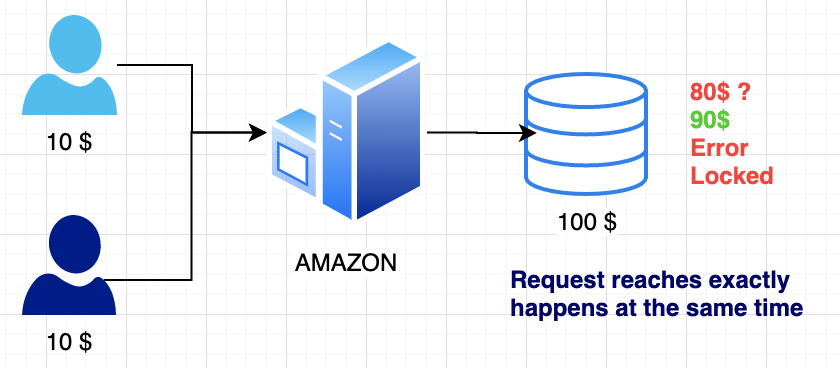
A common challenge in microservices is correctly handling transactions that span multiple services. The success of a transaction is all or nothing — if one of the participating services fails, the entire transaction must fail.

There are two cases to consider:

* A service may experience a transient failure such as a network timeout. These errors can often be resolved simply by retrying the call. If the operation still fails after a certain number of attempts, it's considered a non-transient failure.
* A non-transient failure is any failure that's unlikely to go away by itself. Non-transient failures include normal error conditions, such as invalid input. They also include unhandled exceptions in application code or a process crashing. If this type of error occurs, the entire business transaction must be marked as a failure. It may be necessary to undo other steps in the same transaction that already succeeded.

After a non-transient failure, the current transaction might be in a partially failed state, where one or more steps already completed successfully. In that case, the application needs to undo the steps that succeeded, by using a [Compensating Transaction](https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction). In some cases, this must be done by an external system or even by a manual process.

If the logic for compensating transactions is complex, consider creating a separate service that is responsible for this process.



What happens if Transaction is not used?

A -> 5 $ to each B and C at the same time. (both updates A balance to 4$, it should have been 0$ , Not Isolated, Atomic

Only one thread is able to get to update A’s balance

First is success, second is failed since balance($4) is less than transferred amount.($5+1$)

**Transaction**

Happens by locking a row by any one (if same exact time), other waits.

Example: Transfer (Withdraw , Deposit) ---🡪 ONE Unit

Account A (10$) ----Transfer 5 $ to B---🡪 Account B(0$)

Steps:

1. Withdraw(5$, A)
2. Transfer(5$, B)
3. Deduct commission (1$ per transaction, source account)
4. A(4$), B(5$). Ideal case
5. Withdraw 5$ but it fails, money is deducted (Should rollback and add 5$ back to A)
6. After all are successful, everything should be committed

**++ Transaction ++**

* Reliable unit of work + allows recovery from failures + keeps data consistent
* Isolation from concurrent access.

How is it implemented in different DBs



Allows to read, Stops to update unless transaction is completed/committed

WAL (Write ahead lock) file – in memory storage

**Transactional isolation** is usually implemented by locking whatever is accessed in a transaction. There are 2 different approaches to TL:

* Pessimistic locking
* Optimistic locking.

The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is committed or rolled back, making it inaccessible to other transactions during that time. If most transactions never change the resource, an exclusive lock may be overkill as it may cause lock contention. With pessimistic locking, locks are applied in a fail-safe way. In the banking application example, an account is locked as soon as it is accessed in a transaction. The lock exists until the transaction has either been committed or rolled back . Pessimistic locking may result in deadlocks. However, it ensures greater integrity of data than optimistic locking.

Deadlock is deadly in PL. So, better have timeouts. Also, you need persistent live connection with DB.

Few conflicts: OL Many conflicts: PL

With optimistic locking, a resource is not actually locked when it is first is accessed by a transaction. Instead, the state of the resource at the time when it would have been locked with the pessimistic locking approach is saved. Other transactions are able to concurrently access to the resource and the possibility of conflicting changes is possible. At commit time, when the resource is about to be updated in persistent storage, the state of the resource is read from storage again and compared to the state(Hash, version, checksum, timestamp) that was saved when the resource was first accessed in the transaction. If the two states differ, a conflicting update was made, and the transaction will be rolled back.

In the banking application example, the amount of an account is saved when the account is first accessed in a transaction. If the transaction changes the account amount, the amount is read from the store again just before the amount is about to be updated. If the amount has changed since the transaction began, the transaction will fail itself, otherwise the new amount is written to persistent storage.

**Distributed System:** When it comes to enterprise applications, it's crucial to manage concurrent access to a database properly. This means handle multiple transactions in an effective and most importantly, error-proof way.

We need to ensure that data stays consistent between concurrent reads and updates.

To achieve that we can use optimistic locking mechanism. It leads that multiple updates made on the same data at the same time do not interfere with each other.

Optimistic lock (concurrency) , Lots of operation async way. You always have version number.

ACID

Atomicity- Transaction management component module in DB take cares of A

A transaction must be atomic. This means that either all the work done in the transaction must be performed, or none of it must be performed. Doing part of a transaction is not allowed

* all/none of the instructions are executed
* If 1 transaction at a time -----> degree of concurrency is 1

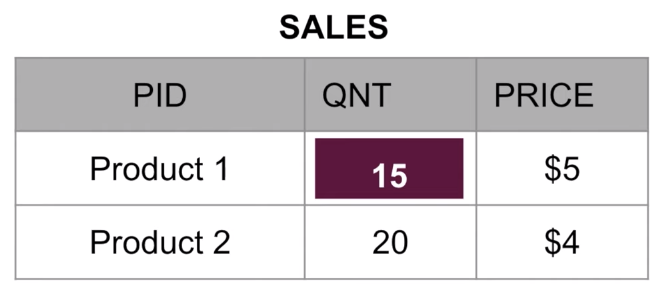
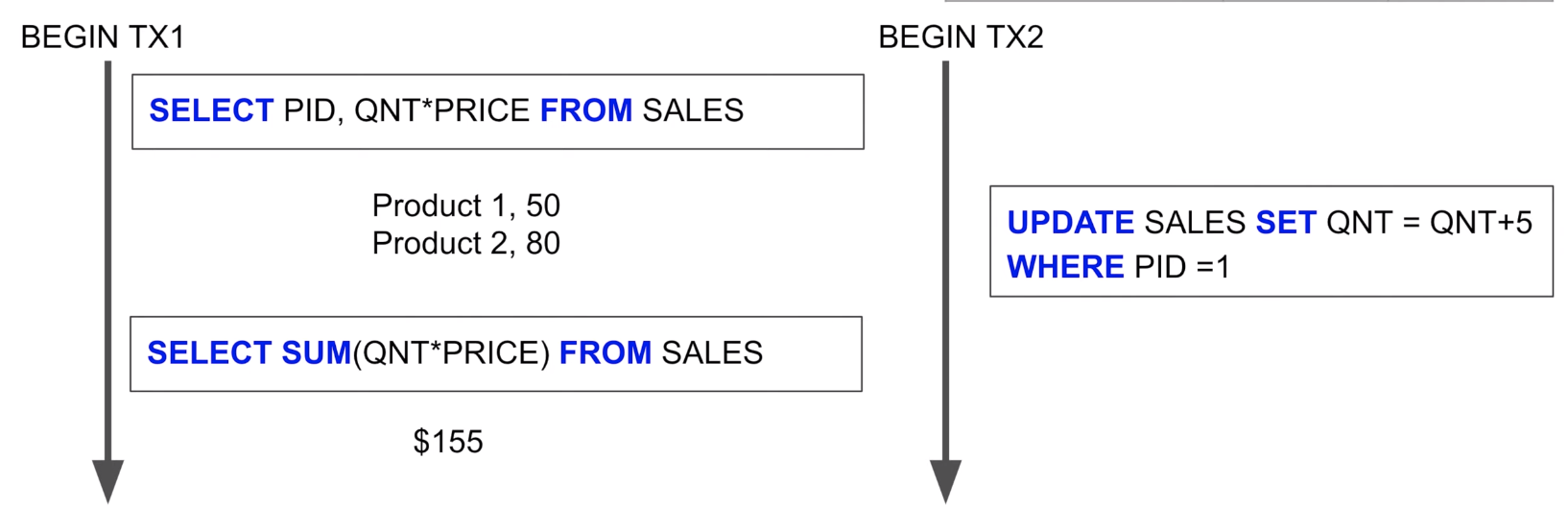
Isolation - Concurrency Control Component takes cares of A

Different transactions must be isolated from each other. This means that the partial work done in one transaction is not visible to other transactions until the transaction is committed, and that each process in a multi-user system can be programmed as if it was the only process accessing the system.

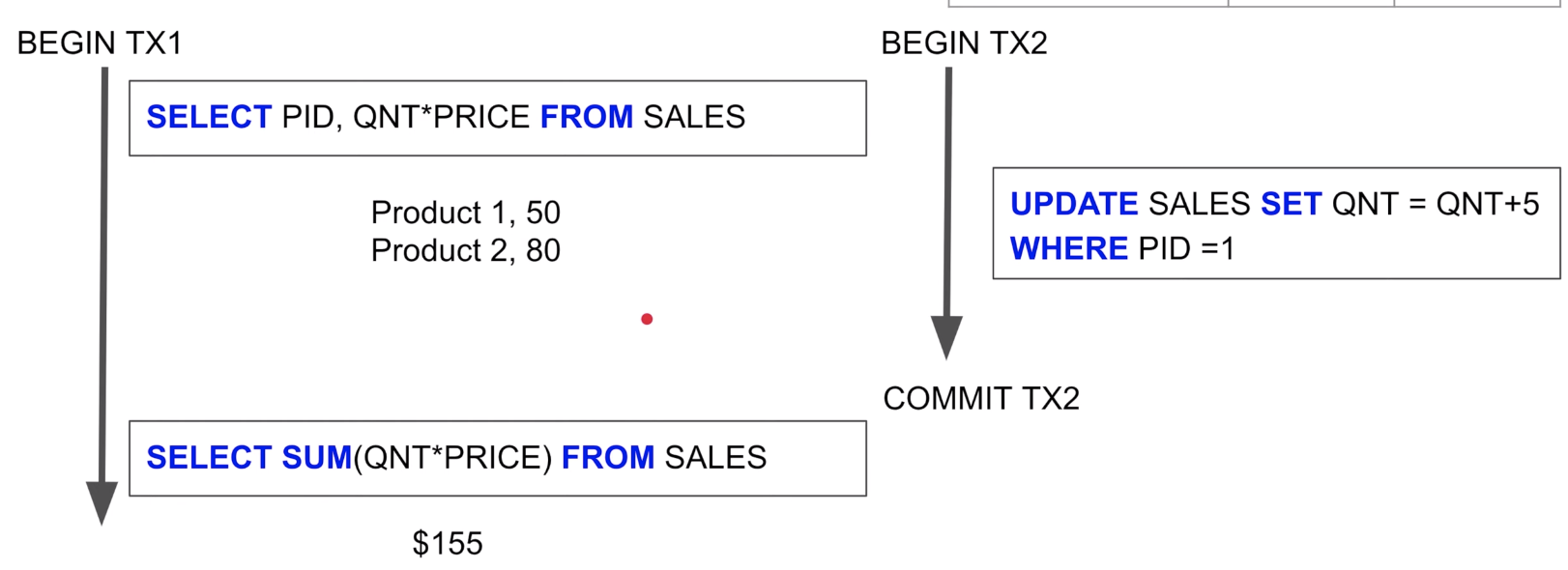
* Can my inflight transaction see changes made by other transactions? {Logical Isolation is needed, two transactions are executed in isolation, no-one affects others}
* As a lack of Isolation, you get -> Read phenomena
* DB can implement isolation Levels to get rid of RP

RP - Read Phenomena

1. Dirty Reads (aka uncommitted dependency) – I am in a transaction and some other transaction made the change and that transaction did not commit that change, but I read it (dirty value)

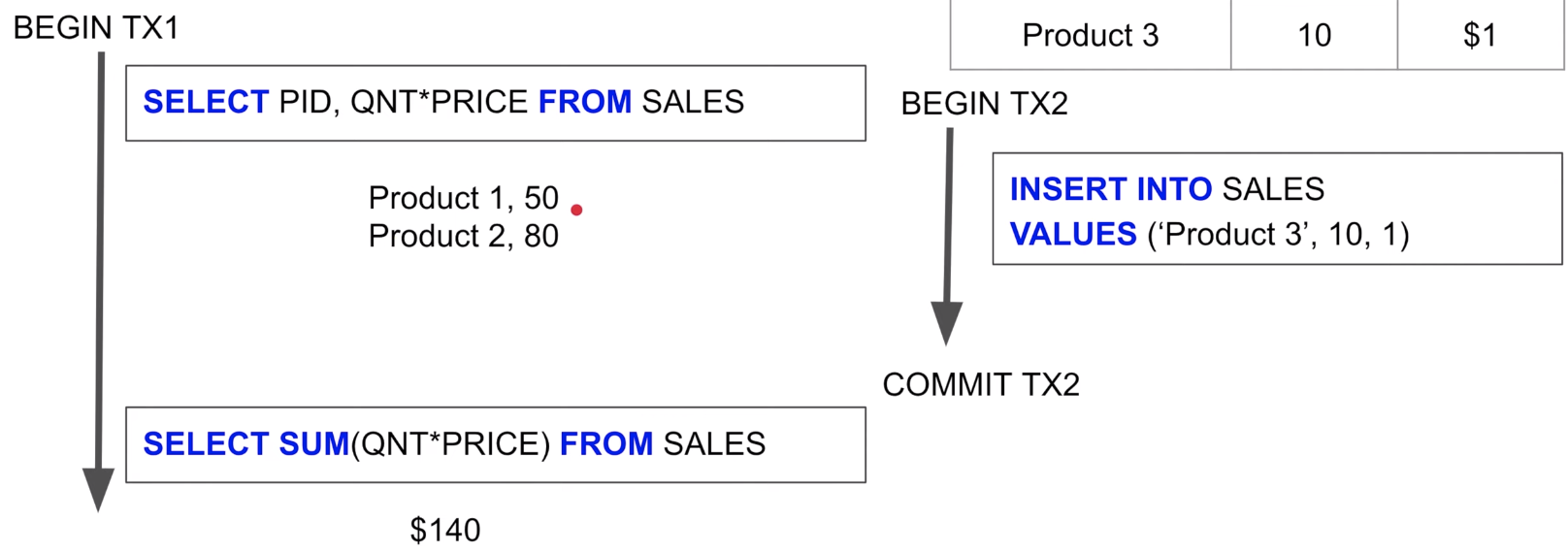


1. Non Repeatable Reads – Between two reads, the value is changed.



1. Phantom Reads –

* Between 2 read the set of rows is changed.
* Happens in Range Queries
* Harder to fix and avoid, previous 2 can be fixed by lock or version controls. How can you lock new value? Fixed by serialization isolation level.



1. (Lost Updates) - Some other transaction overwrote your value before your commit. So, you lost your value.

Isolation Levels

* **Read uncommitted** – Offers no isolation at all, any changes to the DB we are gonna see it. All types of RP.
* **Read committed** – Little bit of isolation, Most of the time it is the best transaction isolation level since you are happy to read the committed stuff. In our case it is not(sales report)

Each query in a transaction only sees committed stuff **at the time of the query.**

* **Repeatable read** – Each query in a transaction only sees committed updates at the beginning of the transaction. (You will see only the changes that have been committed before you started the transaction) – Really powerful, Nice consistent view, reading the same version.

**Some databases implement it using versioning and some in form of locks(expensive). Shared lock on DB rows, You block others changing that data. Exclusive lock needed,**

* **Serializable –** Each transaction has to be serialized

**The performance goes down from top to down.**

****

**Read committed** -> No Dirty Reads, since you are reading only committed values. But N-R reads, since each query is getting new committed stuff. Between two querues someone changed and **committed** the value. You are reading committed value but that changed

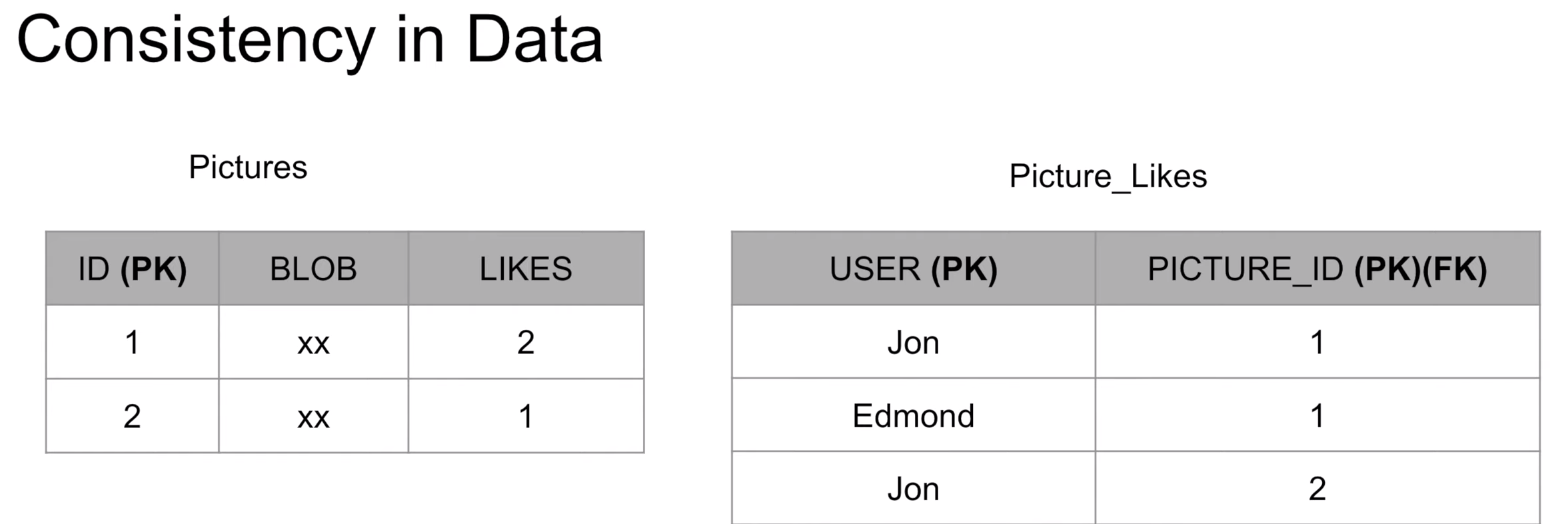
**Repeatable read ->** Only reading committed stuff at the beginning of the transaction, using same version of the db , or db will force you to place lock and read the stuff at the beginning, You can only control the rows you can lock, so Phantoms still can happen

Consistency - Atomicity and Isolation leads to C.

* Consistency in Data

Defined by the user in the table schema. It is enforced by Referential Integrity(FK, PK)

Atomicity and Isolation

 Depends on you how much you want, not exactly accurate. Not important to have consistency for this.

* Consistency in reading the data.

Immediately see the change ??

Relational and NoSQL both suffer from this.

Eventual consistency

Problem only when you have multiple replicas of DB 🡪 You are consistent SIR

**RDBMS is inconsistent in Reads, Are you happy with this? Depends on you.**

**Gave up consistency over Scalability -> NoSQL**

Durability –The changes made during a transaction are made persistent when it is committed. When a transaction is committed, its changes will not be lost, even if the server crashes afterwards.

Redis/MemCached is not durable, In-Memory DB.

**Recovery management component module in DB take cares of D**

Distributed storage system (file systems, relational databases, key-value stores)—

* strong consistency is challenging and costly.
* strong consistency α 1/ (latency & availability).

Weakly consistent storage systems do expose various anomalous behaviors, but they do not completely throw consistency out the window. Instead, they try to sit somewhere between acting completely bananas and acting with strong consistency. They provide some number of basic guarantees that are hopefully sufficient for clients.

Eventual consistency(one of the weakest forms of weak consistency)

An eventually consistent system guarantees that if all clients stop issuing requests for a while, then all the system's replicas will converge to the same state. durability)

# [Replicated Data Consistency](https://scholar.google.com/scholar?cluster=3008756295145383805)

* Distributed storage system (file systems, relational databases, key-value stores)—store a copy of the same data on multiple computers. This is replication(increases system's fault-tolerance but does not expose any anomalous behavior to clients).
* Replication allows a distributed storage system to tolerate computer failures. Unfortunately, a naively replicated storage system can behave very weirdly.
* When a replicated storage system behaves indistinguishably from a storage system running on a single computer, we say it is strongly consistent.

### Consistency - a read request for an entity made to any of the nodes of the database should return the same data.

|  |  |
| --- | --- |
| **Eventual Consistency**: stale data: low latency | **Strong Consistency**: up-to-date data : high latency |
| Data of each node of the database gets consistent eventually. it will take time for updates to reach other replicas. This implies that if someone reads from a replica which is not updated yet (replicas are updated eventually) then it may return stale data. | During the time the replicas are being updated with new data, response to any subsequent read/write requests by any of the replicas will get delayed as all replicas are busy in keeping each other consistent. |
|  |  |

**SQL Indexes**

* INDEX is a performance optimization technique that speeds up the data retrieval process. It is a persistent data structure(PURE REDUNDANCY) associated with a Table (or View) to increase performance during data retrievial
* Without an index: Whole table scan
* You can create indexes on most columns in a table or a view. The exceptions are primarily those columns configured with large object (LOB) data types, such as image, text, and varchar(max).

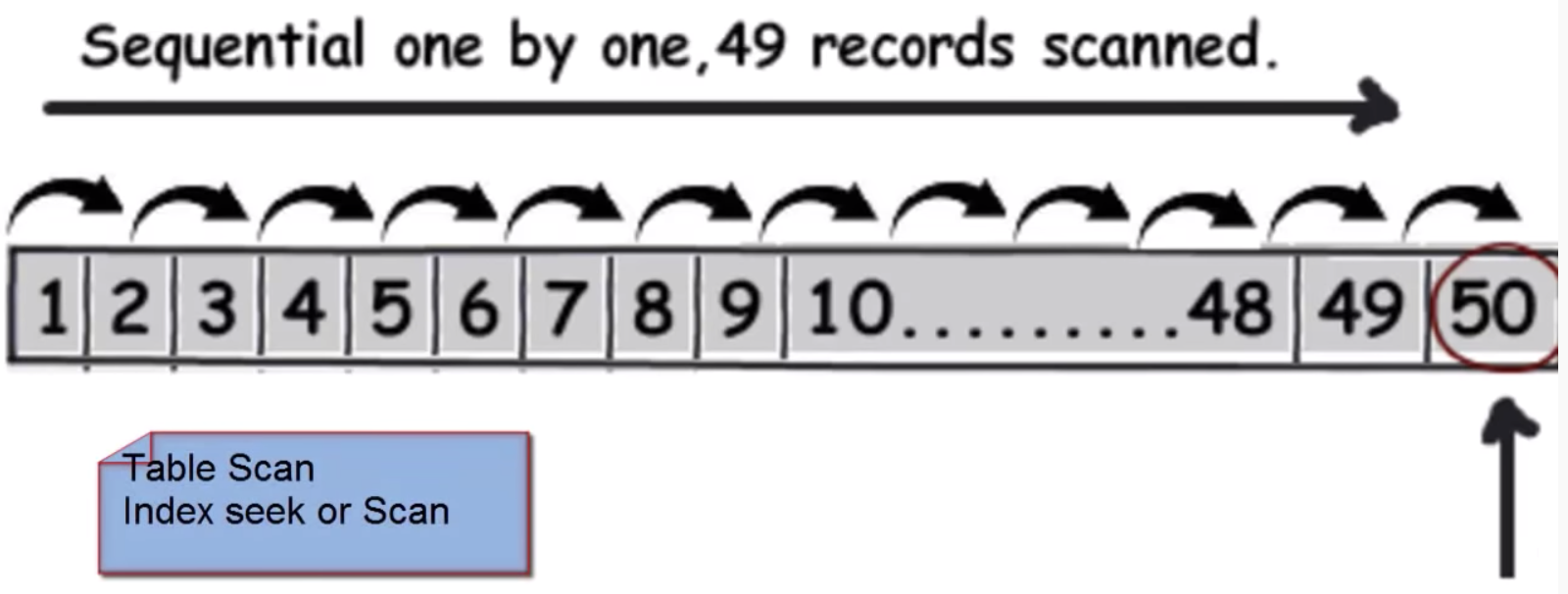
An index is made up of a set of pages (index nodes) that are organized in a B-tree structure. This structure is hierarchical in nature, with the root node at the top of the hierarchy and the leaf nodes at the bottom.

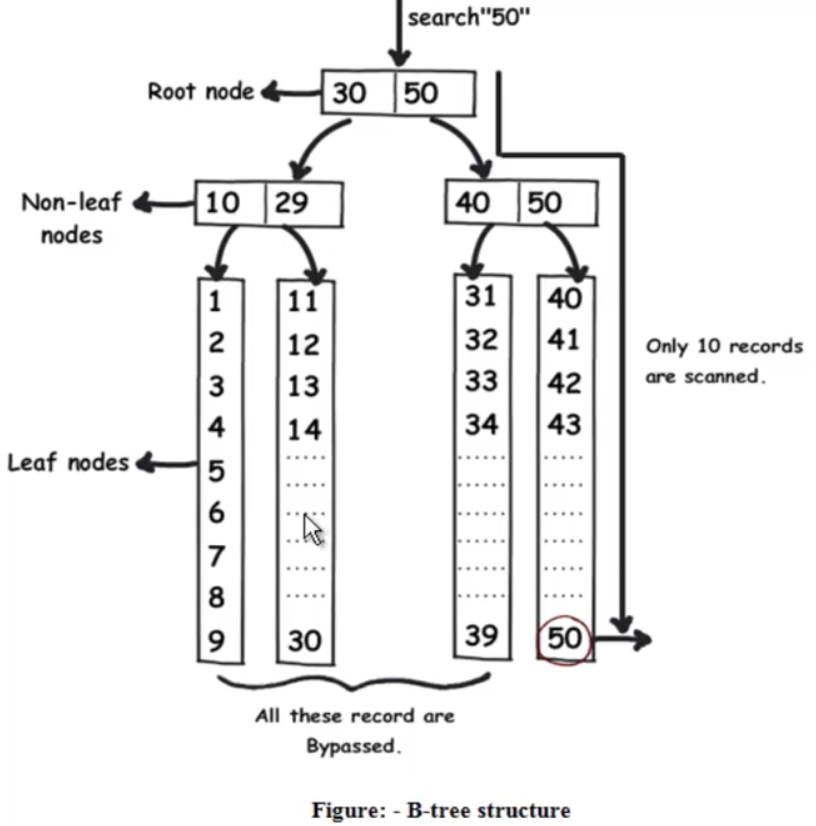
**A clustered index**

* Creates a physical order of rows (it can be only one and in most cases, it is also a primary key - if you create primary key on table you create clustered index on this table also).
* The leaf nodes of a clustered index contain the data pages.
* Best for range-based queries since all the data is located next to each other

**A Non-clustered index**

* The leaf nodes of a nonclustered index contain only the values from the indexed columns and row locators(rather than containinf the data rows themselves.) This means that the query engine must take an additional step in order to locate the actual data.
* Nonclustered indexes cannot be sorted like clustered indexes; however, you can create more than one nonclustered index per table or view.

 **O(N)**

 **O(logn)**

A screenshot of a cell phone

Description automatically generated

**Index Types on Table**

Composite index:  index containing more than one column. (Max limit = 16 columns under 900-byte limit). Both clustered and nonclustered indexes can be composite indexes.

Covering index: A type of index that includes all the columns that are needed to process a particular query. For example, your query might retrieve the FirstName and LastName columns from a table, based on a value in the ContactID column. You can create a covering index that includes all three columns.

Unique Index: An index that ensures the uniqueness of each value in the indexed column. If the index is a composite, the uniqueness is enforced across the columns as a whole, not on the individual columns. **primary key constraint on one or more columns -> SQL Server automatically creates a unique, clustered index.** However, you can override the default behavior and define a unique, nonclustered index on the primary key.

Maintaining the unique Indexes

Columns requiring unique values (such as primary key columns) must have a unique index applied. Marking a column as a primary key will automatically create a unique index on the column. We can also create a unique index using SQL with the following command:



The above SQL command will not allow any duplicate values in the ProductName column, and **an index is the best tool for the database to use to enforce this rule.** Each time an application adds or modifies a row in the table, the database needs to search all existing records to ensure none of values in the new data duplicate existing values.

**Index Design**

* Indexes can take up significant disk space (Redundancy). Indexes are automatically updated when the data rows are updated, which is an additional overhead and can affect performance.
  + For tables that are heavily updated: use as few columns as possible in the index, and don’t over-index the tables.
  + For tables that has low data modifications, use as many indexes as necessary to improve query performance.
* Use indexes judiciously on small tables because a table scan might be faster than navigating the index. (worst scenario for having index)
* For clustered indexes, try to keep the length of the indexed columns as short as possible. Ideally, try to implement your clustered indexes on unique columns that do not permit null values. This is why the primary key is often used for the table’s clustered index
* uniqueness -🡪 affects index performance. When possible, implement unique indexes.

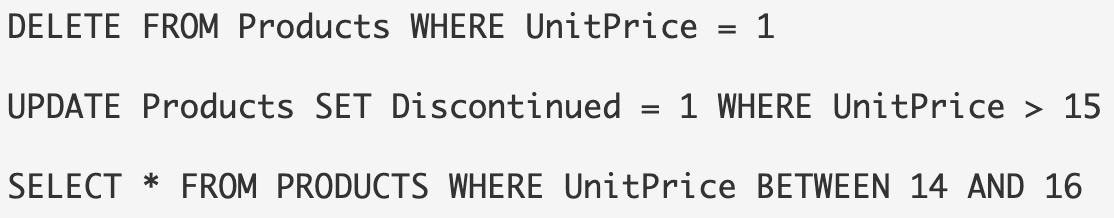
the more duplicate values you have in a column, the more poorly the index performs.

the more unique each value, the better the performance.

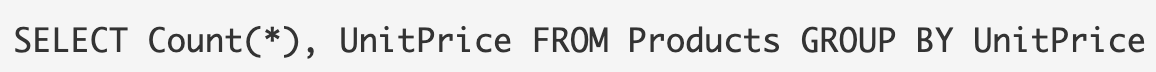
* order of the columns in the index definition For composite indexes, Example: where clause columns -> based on the uniqueness of their values, with the most unique listed first
* Batch update in a single statement, rather than using multiple queries.
* Create nonclustered indexes on columns used frequently in your statement’s predicates and join conditions.
* Consider indexing columns used in exact-match queries.

**Taking advantage of indexes**

Searching For Records: Indexes can help queries to be faster looking for values inside of a range as well as queries looking for a specific value. (if we have index on UnitPrice.)



Sorting Records: When we ask for a sorted dataset, the database will try to find an index and avoids sorting the results during execution of the query. We control sorting of a dataset by specifying ORDER BY clause.

Grouping Records: GROUP BY is used to group records and aggregate values. To process a query with a GROUP BY clause, the database will often sort the results on the columns included in the GROUP BY. The next query counts the number of products at each price by grouping together records with the same UnitPrice value. **

The database uses the IDX(UnitPrice) index to retrieve the prices in order. Since matching prices appear in consecutive index entries, the database is able count the number of products at each price quickly. Indexing a field used in a GROUP BY clause can often speed up a query.

Without index, the database scans the FULL Products table and sort the rows to process the query(extra operation). With index we will have a presorted list of prices. The database simply scans the index from the first entry to the last entry and retrieve the rows in sorted order.

#### **SQL Performance**

1. Index all the predicates in JOIN, WHERE, ORDER BY and GROUP BY clauses (uniqueness order). Without indexes, SQL queries can cause table scans (performance or locking problems.)
2. Avoid using functions in predicates because index is not used when functions are used. Database optimizers skips index on COL1 below.

SELECT \* FROM TABLE1 WHERE UPPER(COL1)='ABC'

1. LIKE '%abc' causes full table scan. This is a known performance limitation in all databases.

SELECT \* FROM TABLE1 WHERE COL1 LIKE '%ABC'

1. Specific SELECT  instead of SELECT \*. The unnecessary columns places extra loads on the database, which slows down not just the single SQL, but the whole system.
2. Inner join > outer join. Using outer join limits the database optimization options which typically results in slower SQL execution.
3. Avoid DISTINCT and UNION since they slows down the SQL execution. Use UNION ALL instead of UNION, if possible, as it is much more efficient.
4. Oracle 10g and 11g -> CLOB/BLOB columns must be put at the end of the statements. Otherwise, it causes failure when the input value size is larger than 1000 characters.
5. The ORDER BY sorts the result-set by specified columns. Be aware of the performance impact of adding the ORDER BY clause, as the database needs to sort the result set, resulting in one of the most expensive operations in SQL execution.
6. For SQL queries with the LEFT OUTER JOIN, push predicates of the right table from the WHERE clause into the ON condition. It helps the database optimizer to generate a more efficient query. Predicates of the left table can stay in the WHERE clause.

For example, the suboptimal query is rewritten by pushing predicates applicable to the table TAB\_B into the ON clause. The TAB\_A specific predicates in the WHERE clause can either stay, or be pushed into the ON clause:

Suboptimal SQL statement:

SELECT A.COL1, B.COL1 FROM A LEFT OUTER JOIN B ON A.COL3 = B.COL3 WHERE A.COL1=123 AND B.COL2=456;

Optimized SQL statement:

SELECT A.COL1, B.COL1 FROM A LEFT OUTER JOIN B ON (A.COL3 = B.COL3) AND (B.COL2=456) WHERE A.COL1=123;

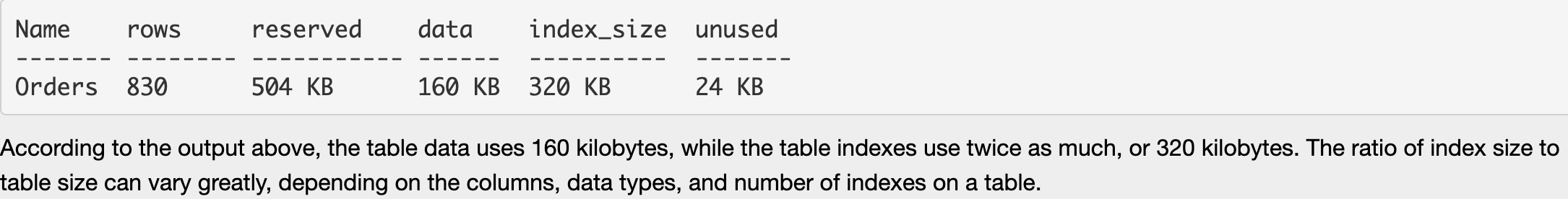
Predicates for any INNER joins can stay in the WHERE clause.

#### **++Index--**

* Indexes are stored on the disk, Amount of space required --🡪 size of the table, number and types of columns used in the index.

Disk space is generally cheap enough to trade for application performance, particularly when a database serves a large number of users.

* + - EXEC sp\_spaceused Orders



#### Data Modification: Anytime a query modifies the data in a table (INSERT, UPDATE, or DELETE), the database needs to update all of the indexes where data has changed. Providing too many indexes can actually hurt the performance in data modifications. This leads to a delicate balancing act when tuning the database for performance.

In decision support systems and data warehouses, where information is stored for reporting purposes, data remains relatively static and report generating queries outnumber data modification queries. In these types of environments, heavy indexing is commonplace in order to optimize the reports generated.

In contrast, a database used for transaction processing will see many records added and updated. These types of databases will use fewer indexes to allow for higher throughput on inserts and updates.

Well-designed indexes reduce disk I/O operations and consume fewer system resources so better query performance.

### **SQL or NoSQL:** How your data looks like, How you’ll query it, and your scalability needs.

|  |  |
| --- | --- |
| ACID compliance protects the integrity of your data  It avoids database tables from becoming out-of-sync, which is super important for financial transactions.  ACID compliance guarantees validity of transactions even in the face of errors, technology failures, disastrous events, and more. | Focused on scalability and flexibility, not query efficiency  NoSQL requires much less structure than SQL, each stored object is pretty much self-contained and independent. Thus, objects can be easily stored on multiple servers without having to be linked.  NoSQL database follows the Brewers CAP theorem (Consistency, Availability and Partition tolerance ) |
| Predefined fixed schema, Can be restrictive. Each record has a fixed schema, columns must be decided before data entry, can be altered but involves modifying whole database & going offline.  Bad for hierarchical data storage | Dynamic schema: Columns can be added on the fly and each entry does not have to contain all columns.   * You can create documents without defining their structure * Each document can have its own unique structure * Have multiple databases with different structures and syntax   Better for the hierarchical data storage as it follows the key-value pair way of storing data similar to JSON data. |
| Vertically scalable (by increasing the horse-power of the hardware). You can manage increasing load by increasing the CPU, RAM, SSD on a single server: EXPENSIVE | Horizontally scalable (by increasing the databases servers in the pool of resources to reduce the load)  handle more traffic by sharding, or adding more servers in your NoSQL database |
| Extremely powerful | Queries are focused on collection of documents. UnQL (Unstructured Query Language). |
| MySql, Oracle, Sqlite, Postgres and MS-SQL | MongoDB, BigTable, Redis, RavenDb, Cassandra, Hbase, Neo4j and CouchDb |
| * good fit for the complex queries * better for applications that require multi-row transactions * High transactional based application, as it is more stable and promises the atomicity as well as integrity of the data. | * NoSQL don’t have standard interfaces to perform complex queries, and the queries in NoSQL are not as powerful as SQL query language. * Not comparable and stable enough in high load and for complex transactional applications * Sacrifice ACID compliance for performance and scalability. |

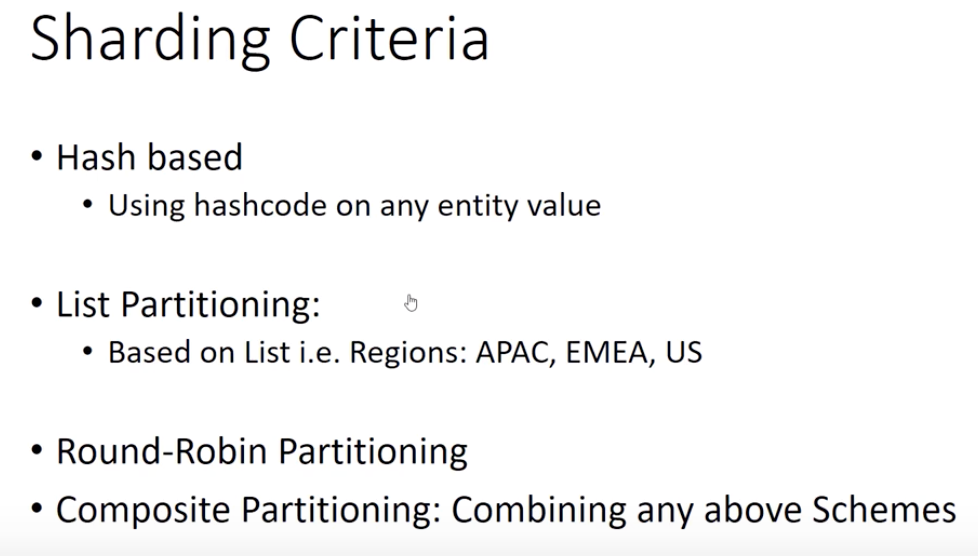
**Partitioning -** a **general term** - breaking up your logical data elements into multiple entities for PAM i.e. performance, availability, maintainability.

Partitioning has two flavors: HORIZONTAL && VERTICAL

Sharding is HP across servers (distributed): Break up a big database into smaller chunks

A screenshot of a cell phone

Description automatically generated



**Sharding Methods**

1. Hash-based partitioning:  We have N DB servers and our ID is a numeric value that gets incremented each time a new record is inserted.

Hash function could be ‘ID % N’, which will give us the server number where we can store/read that record.. In case of addition/removal of servers majority of the keys would need to be remapped and migrated to new servers and you'll incur a downtime till the migration completes. Solution: Consistent Hashing.

1. Horizontal partitioning (range based sharding): Putting different rows into different DBs

* Partition by last name character
* Partition by Pincode. --🡪 (Non uniform distribution -> may lead to unbalanced servers)
* Simplest sharding scheme where each shard has the same schema as the original database.
* Your application layer is relatively simple because in most scenarios, you'll NOT need to combine data from multiple shards to answer any query.
* Works well for relative non static data

1. Vertical Partitioning (feature based sharding): 1 feature in 1 DB.

For example, in a Facebook application, we might place the various user profiles on one shard, the friends on a second shard and the groups on a third shard.

You can handle the critical part of your data (for examples User Profiles) differently from the not so critical part of your data (for example, groups) and build different replication and consistency models around it.

Cross-shard join. For example, a profile view request will need to combine data from the User Profile, Friends and Groups shard which increases the development and operational complexity of the system.

If our application experiences additional growth, then it may be necessary to further partition a feature specific DB across various servers. (Timeline)

1. Directory Based Partitioning:

A lookup service(usually implemented as a webservice) in front of the sharded databases. The lookup service knows the current partitioning scheme and keeps a map of each entity and which database shard it is stored on. This loosely coupled approach means we can perform tasks like adding servers to the DB pool or change our partitioning scheme without having to impact your application.

**Sharding Example Netflix Objective: You need to store and provide low latency reads to a huge number of video files**. In this case you might want to shard by the genre of the movies. You'll also want to create replicas of the individual shards to provide high availability. The primary focus of sharding is to improve the performance and scalability of a system, but as a by-product it can also improve availability due to how the data is divided into separate partitions.

A failure in one partition doesn't necessarily prevent an application from accessing data held in other partitions, and an operator can perform maintenance or recovery of one or more partitions without making the entire data for an application inaccessible.

**Common Problems of Sharding:** Operations across multiple tables or multiple rows in the same table, will no longer run on the same server

1. ACID compliance is difficult: Data is distributed across many nodes
2. Joins and Denormalization:  NOT feasible/efficient to perform joins on partitioned DB since data has to be compiled from multiple servers. In certain situations, cross machine joins may not be an option if you need to maintain high availability SLA for your service. Then the only option left is to de-normalize your database to avoid cross server joins. While this scheme helps with system availability, you now have to contend with keeping all the data in the different shards consistent.
3. Rebalancing: There could be many reasons we have to change our sharding scheme:

* The data distribution is not uniform, e.g., there are a lot of places for a particular ZIP code, that cannot fit into one database partition.
* Excessive load on a shard, e.g., there are too many requests being handled by the DB shard dedicated to user photos.

In such cases, either we have to create more DB shards or have to rebalance existing shards, which means the partitioning scheme changed and all existing data moved to new locations. Doing this without incurring downtime is extremely difficult.

In such cases, either we have to create more DB shards or have to rebalance existing shards, which means the partitioning scheme changed and all existing data moved to new locations. Doing this without incurring downtime is extremely difficult.

When to use the Sharding in a System Design Interview?

Use this pattern when a data store is likely to need to scale beyond the resources available to a single storage node, or to improve performance by reducing contention in a data store.

**Distributed systems** A distributed system is a group of computers working together as to appear as a single computer to the end-user. These machines have a shared state, operate concurrently and can fail independently without affecting the whole system’s uptime.

No single point of failure: All the nodes in the distributed system are connected to each other. So, nodes share data with other nodes. Failure of one node does not lead to the failure of the entire distributed system.

Can be scaled as required: With horizontal scaling you have no cap on how much you can scale — whenever performance degrades you simply add another machine, up to infinity potentially.

Fault Tolerance — a cluster of ten machines across two data centers is inherently more fault-tolerant than a single machine. Even if one data center catches on fire, your application would still work.

Low Latency

Security: the nodes AND connections need to be secured.

Inconsistency: You insert a record in one node and fetch the data from another node immediately, by this time updates might not have propagated to all nodes.

Some messages and data can be lost in the network while moving from one node to another.

Overloading may occur in the network if all the nodes of the distributed system try to send data at once.

**CAP Theorem** is a concept that a distributed database system can only have 2 of the 3: Consistency, Availability and Partition Tolerance. So we need to make trade off’s between the three depending on our requirement.

Partition Tolerance (it is not an option. It’s a necessity)

A partition-tolerant system can sustain any amount of network failure that doesn’t result in a failure of the entire network. Data is replicated across combinations of nodes and networks to keep the system up through intermittent outages. Distributed systems MUST BE partition tolerant (ie. it simply wouldn’t be a distributed if it wasn’t partition tolerant.). Since this is a must we have to trade between Consistency and Availability.

Consistency In a consistent system, once a client writes a value to any server and gets a response, it expects to get that value (or a fresher value) back from any server it reads from. A system can (and does) shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process.

However, the nodes will need time to update and will not be Available on the network as often.

Availability In an available system, if our client sends a request to a server and the server has not crashed, then the server must eventually respond to the client. The server is not allowed to ignore the client's requests.

Every client gets a response, regardless of the state of any individual node in the system.

Tradeoff between Consistency and Availability: Network outages (temporary and permanent) are a fact of life and occur whether you want them to or not - this exists outside of your software.

Understanding the trade-offs available to you in the face of network errors, and choosing the right path is vital to the success of your application. Failing to get this right from the beginning could doom your application to failure before your first deployment

**Choose Consistency over Availability.** **Choose Availability over Consistency**

when your business requirements dictate When your business requirements allow flexibility

atomic reads and writes

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition.

Application metrics. metrics relevant to understanding the behavior of a service. Examples include the number of queued inbound HTTP requests, request latency, or message queue length. Applications can also create custom metrics that are specific to the domain, such as the number of business transactions processed per minute.

Dependent service metrics. Services may call external services or endpoints, such as managed PaaS services or SaaS services. Third-party services may or may not provide any metrics. If not, you'll have to rely on your own application metrics to track statistics for latency and error rate.

**API DESIGN**

APIs must have well-defined semantics and versioning schemes, so that updates don't break other services.

It's important to distinguish between two types of API:

**Public APIs** that client applications call so they must be compatible with client applications, typically browser applications or native mobile applications. Most of the time, that means the public API will use REST over HTTP.

**Backend APIs** are used for interservice communication. taking network performance into account. Depending on the granularity of your services, interservice communication can result in a lot of network traffic. Services can quickly become I/O bound. For that reason, considerations such as serialization speed and payload size become more important. Binary serialization is more efficient than HTTP. (gRPC, Apache Avro, and Apache Thrift uses Binary serialization)

## Considerations choosing how to implement an API.

**REST versus RPC**.

REST models resources which is one of the way to express your domain model. It has well-defined semantics in terms of idempotency, side effects, and response codes. And it enforces stateless communication, which improves scalability. For a RESTful interface, the most common choice is REST over HTTP using JSON.

RPC is more oriented around operations or commands. Because RPC interfaces look like local method calls, it may lead you to design overly chatty APIs. However, that doesn't mean RPC must be chatty. It just means you need to use care when designing the interface. For an RPC-style interface, there are several popular frameworks, including gRPC, Apache Avro, and Apache Thrift

**Efficiency**. efficiency in terms of speed, memory, and payload size. a gRPC-based interface is faster than REST over HTTP.

**Interface definition language (IDL)**. An IDL is used to define the methods, parameters, and return values of an API. An IDL can be used to generate client code, serialization code, and API documentation. REST over HTTP does not have a standard IDL format, but a common choice is OpenAPI (formerly Swagger). You can also create an HTTP REST API without using a formal definition language, but then you lose the benefits of code generation and testing. Frameworks such as gRPC, Avro, and Thrift define their own IDL specifications.

**Serialization**. How are objects serialized over the wire?

Text-based formats (primarily JSON)

Binary formats such as protocol buffer.

Binary formats are generally faster than text-based formats.

However, JSON has advantages in terms of interoperability, because most languages and frameworks support JSON serialization. Some serialization formats require a fixed schema, and some require compiling a schema definition file. In that case, you'll need to incorporate this step into your build process.

Choose REST over HTTP unless you need the performance benefits of a binary protocol. REST over HTTP requires no special libraries. It creates minimal coupling, because callers don't need a client stub to communicate with the service. There is rich ecosystems of tools to support schema definitions, testing, and monitoring of RESTful HTTP endpoints. Finally, HTTP is compatible with browser clients, so you don't need a protocol translation layer between the client and the backend.

However, if you choose REST over HTTP, you should do performance and load testing early in the development process, to validate whether it performs well enough for your scenario.