Important diagrams

**What’s the difference between**

**What’s difference between cursor based pagination and offset based pagination?**

2. How would you build the caching layer in mobile? Remember this can be used anywhere

What kind of caching did you use?

1. Can check out the big system design part up int eh code above

**MicroService communication**

1. What’s quorum   
    note this is not a consensus here
2. How to configure quorum consensu for read and write heavy system
3. CAP theorem
4. Raft consensus? (Leader and follower) Raft and Paxos
5. Write ahead logging

**Difference between version clock, quorum and raft consensus?**

There are various problems to tackle in distributes systems. And there are different tools to solve those challenges.

Problem1: I'd like to make a decision involving specific number of nodes. We will call that number - quorum. For example, in leaderless replication

based on Dynamo, quorum is a number of nodes representing a majority.

To be clear, quorum does not have to be a majority - it all depends on specifics of the problem. E.g. you could say something like - in system X a quorum is a set of three oldest nodes.

Problem2: We have multiple nodes, we want them all to agree on something - we want nodes to get to a Consensus on a specific decision. E.g. there are 10 numbers (0..9) and 100 nodes. We want them all to pick the same number.

So, the consensus is a general idea of agreement on something. Common algorithms are Paxos, Raft, etc.

Problem 3: I have a distributed system which processes events on each node. Some of those events will be concurrent to each other. How do I detect those? I'll use version clock for that.

Problem 4: I have several replicas of some data. These replicas may process some events locally and also synchronize to each other. When I do synchronize, how do I know which replica is more recent? And how do I detect if replicas have conflicting data? I'll use version vector for this.

Database

1. How to handle heavy read system
2. How to achieve strong consistency between master and slave db?
3. How to handle concurrency issues? (with locks and all that)

[A diagram of a database

Description automatically generated](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fbucketeer-e05bbc84-baa3-437e-9518-adb32be77984.s3.amazonaws.com%2Fpublic%2Fimages%2Fa0018b6a-0e64-4dc6-a389-0cd77a5fa7b8_1999x1837.png)

General knowledge

**How to scale**

Diagram of a diagram of a computer

Description automatically generated

1. Vertical vs horizontal

2. Add load balancer

4. Adding cache

5. CDN load balancer (serves images closer to user

6. Using async message broker

**In a really read-heavy system, whar are the 2 ways we can increase the read performance here?**

1. Use sharding
2. Add more read replicas and both make sense
3. We can add caching to make things easier for us over time

**When do you need to use the 2 phase commit?**

When we have both CustomerMicroserviceand the OrderMicroservice, which have separate databases. Here is a customer order example with microservices:

**Strong vs eventual consistency?**

When do you use which? We want strong consistency in parking lot case, not eventual consistency, because we want everyting

Eventual consistenyc is at the very end

Using a read lock here, even in the garage base example here.

**How to handle a lot of reads vs little reads?**

1. Use a lot of read replicas here

How to achieve strong consistency?

Using caching right now

1. Cache asid

1. What’s the CAP theroem?

2.

Defer long-running or non-critical tasks to background queues or message brokers. This ensures your main application remains responsive to users.

**Example:** Slack uses asynchronous communication for messaging. When a message is sent, the sender's interface doesn't freeze; it continues to be responsive while the message is processed and delivered in the background.

**System design how to ensure atomicity between transactions in datasbase?**

Well we have 2 options here, either synchronous or asynchrounous.

**What’s atomic transaction?**

1. *In synchronous replication, the leader node waits for all of the follower nodes to acknowledge receipt of the data change before reporting success to the client.*
2. Can be slow as said for the user so not so good in that regard

**Aysnc transaciton**

1. *In async situation, the leader node does not wait for the follower nodes to acknowledge receipt of the data change before reporting success to the client.*

*This means that the follower nodes may not have the most up-to-date copy of the data, and*

***How to compensate this?***

In practice, many systems use a combination of synchronous and asynchronous replication. For example, a database might use synchronous replication to replicate data to a small number of critical replicas, and asynchronous replication to replicate data to other replicas. This configuration is sometimes also called Semi-Synchronous.

**Q: How can we achieve strong consistency between the master and slave databases?**

A: WAL (Write-Ahead-Logging), a well-known and important technology in database, is needed for this purpose-. When the system updates the operation write logs (Oracle Redo Log and MySQL Binlog among others) or commits a transaction, one should first ensure the flushing of the logs generated by the transaction to the disk. This would ensure no data loss occurs.

The method for achieving the strong data consistency between the master and slave databases is also simple:

* When you commit a transaction, it initiates two log writing operations: the first one to write the log to the ephemeral disk, and the other to synchronize the log to the slave database and ensure it is saved to the disk
* The master database will return the response to the application only upon the successful return of the two operations, at which point the transaction commitment is successful.

Diagram of a diagram of a computer

Description automatically generated

**What is fault tolerance**

Fault tolerance refers to the ability of a system (computer, network, cloud cluster, etc.) to continue operating without interruption when one or more of its components fail.

* **Software systems** that are backed up by other software instances. For example, a database with customer information can be continuously replicated to another machine. If the primary database goes down, operations can be automatically redirected to the second database.

**Offset vs Cursor-Based Pagination: Choosing the Best Approach**

When it comes to implementing pagination for large datasets in API responses, developers often find themselves at a crossroads, faced with the choice between offset-based and cursor-based pagination. Both methods have their merits and are suitable for specific use cases. In this article, we’ll dive deep into these two approaches, clarify their differences, and provide insights into their advantages and drawbacks.

**Offset-Based Pagination**

**Client-Side Parameters:**

* Limit: Specifies how many records to fetch per page.
* Offset: Indicates where to start fetching data or how many records to skip, defining the initial position within the list.

**How It Works:**

* In an API call, the frontend provides both the limit and offset.
* The database retrieves records starting from the given offset and returns them up to the limit.

**Advantages:**

* Simplicity: Offset-based pagination is easy to implement.
* Static Data: It works well for static datasets that don’t change frequently.

**Ideal Usage**

* It’s a good fit for handling static data that doesn’t change frequently.
* Ideal when you need to allow users to directly jump to a specific page.

**Drawback:** The major drawback is that as the offset increases, the query becomes slower. This is because the database must read records up to the offset, which means if you have 1 million records,

and you have to query 10 records after 10,000th row (limit = 10, offset = 10,000). The database will read all the records from 0 to 99999 rows and return the 10 records after it. Isn’t it worst? Just think about it.

**Example:** Let’s say you have a database of a thousand books, and you want to display 10 books per page:

* First Page: Offset = 0, Limit = 10
* Second Page: Offset = 10, Limit = 10
* Third Page: Offset = 20, Limit = 10
* … and so forth.

Here’s an SQL Query for Offset-Based Pagination in this scenario:

-- SQL Query for Offset-Based Pagination  
SELECT \* FROM books  
ORDER BY book\_id  
LIMIT 10 OFFSET 0; -- First Page  
SELECT \* FROM books  
ORDER BY book\_id  
LIMIT 10 OFFSET 10; -- Second Page  
-- And so on...

**Cursor-Based Pagination**

**Client-Side Parameters:**

* Limit: Specifies the number of records required per page.
* Cursor: Acts as a pointer to fetch data. The cursor is provided by the database and points to a specific item within the dataset.  
  The pointer can be a raw data like what is the

**How It Works:**

* In the initial API call, the frontend only provides the limit (the cursor is not included).
* The database returns a cursor pointing to the last item within the dataset in its response.
* In subsequent API calls, the frontend includes both the limit and the cursor value (pointing to the last records retrieved from the previous request).
* The database provides records that come after or before the cursor, up to the specified limit.

**Advantages:**

* Faster Data Retrieval: Cursor-based pagination is highly efficient, especially for real-time data. For Example, If the cursor is at 10,000 and limit is 10, the database reads from 10,000 to 10,010, instead of reading from the 0 record. This consistent query time is a major advantage.
* Infinite Scrolling: It is well-suited for implementing features like infinite scrolling.

**Ideal Usage:**Cursor-based pagination shines in scenarios where data is frequently updated or changes occur in real-time.

**Example:**Consider the same scenario of a thousand books, with 10 books displayed per page. In this case, the book name serves as the cursor:

* First Page: Cursor = null, Limit = 10 (Assuming the last book fetched in the 10th record has the name “Atomic Habits”).
* Second Page: Cursor = Fetch records whose name is greater than “Atomic Habits,” Limit = 10 (Assuming the last book fetched in the 20th record has the name “Calculus”).
* Third Page: Cursor = Fetch records whose name is greater than “Calculus,” Limit = 10, and so on.

Here’s the SQL Query for Cursor-Based Pagination in this context:

-- SQL Query for Cursor-Based Pagination  
SELECT \* FROM books  
WHERE book\_name > 'Atomic Habits' -- Assuming 'Atomic Habits' is the last book on the previous page  
ORDER BY book\_name  
LIMIT 10;  
-- First Page: Cursor = null  
SELECT \* FROM books  
WHERE book\_name > 'Calculus' -- Assuming 'Calculus' is the last book on the previous page

ORDER BY book\_name  
LIMIT 10;  
-- Second Page: Cursor = 'Calculus'  
-- And so on...

**Note:**

* When requesting the next page, we ask the database to send us records greater than the cursor, or smaller if moving backward.
* It’s advisable to create an index on the column you intend to use as a cursor. This indexing enhances the performance and efficiency of cursor-based pagination.

Now that we have a solid understanding of both pagination methods, let’s explore when to choose one over the other and why.

**Why Choose Cursor-Based over Offset-Based Pagination**

* **Performance**: As the offset increases in offset-based pagination, it can become slow because the database needs to read up to the offset position, involving counting rows from disk. In contrast, cursor-based pagination directly points to the cursor’s location, making it faster.
* **Dynamic Data:** Cursor-based pagination is better suited for dynamic data where records may be inserted or deleted, as it acts as a pointer to where the last request left off.

**Why Choose Offset-Based over Cursor-Based Pagination**

* **Sorting Flexibility:**Offset-based pagination allows for easier sorting of data based on various attributes. In cursor-based pagination, the cursor must be a unique attribute that can be sequentially ordered, and data will always be sorted by that attribute.
* **User Navigation:**Offset-based pagination enables users to jump to specific pages directly, which is not easily achievable in cursor-based pagination.

**How to handle Concurrency issues**

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. These problems are commonly referred to as problems in a database environment.

Using locking

2. Optimistic locking

This method allows any update of a record to happen only when the value of that record has not changed after its last read.

Optimistic Locking checks if the current value of the record is the same as it was when previously read then the update is not allowed, and the read-modify-write cycle has to be repeated. Optimistic Locking is also called Conditional Update or Compare-And-Update.

- Use this where conflicts are low

3.Pessimistic Locking

After locking the object read-modify-write operations are performed on the objects and then the object is released. During these operations, if another transaction tries to read the same object it has to wait until the read-modify-write cycle of the first transaction is completed.

- Use this when conflicts are high, used in banking systems here

**What’s quorum consensus? Really important (this can be seen in the key value store) problem**

**Problem statement:**

In a distributed system, whenever a server takes any action, it needs to ensure that in the event of a crash the results of the actions are available to the clients. This can be achieved by replicating the result to other servers in the cluster. But that leads to the question:

how many other servers need to confirm the replication before the original server can be confident that the update is fully recognized?

If the original server waits for too many replications, then it will respond slowly. This will in turn, cause the liveness of the system impacted because any node that is suffering outage will impact the original server to not commit the transaction successfully.

But if it doesn’t have enough replications, then the update could be lost. This will in turn cause failure of safety since the data committed in the system can get lost. It’s critical to balance between the overall system performance and system continuity.

What you are doing is as a developer is choosign the quorum value

N = Number of replicas in a distributed system

W = A write quorum of size W. For a write operation to be considered successful, the write should be acknowledged from W replicas.

R = A read quorum of size R. For a read operation to be considered successful, read must wait for responses from atleast R replicas.

What does it mean when we have?

Therefore N=3, W=1 means that, the write must be confirmed by at least one node for the operation to be successful. The parameters N, W and R are configurable. Typically N is chosen to be an odd number and we set W=R=(N+1)/2. Usually W+R > N makes the system tolerable to unavailability and ensures consistency.

With N=3, W=2, R=2 we can tolerate 1 unavailable node.

**How would you optimize this for read and write heavy system?**

If R=1 and W=N, then the system is optimised for faster reads.

If W=1 and R=N, then the system is optimised for faster writes.

Choose the quorum values wisely based on your requirement.

**Solution**

A quorum is the minimum number of nodes in the system that have committed a transaction. In simple term, it is the majority or 1 + 1/2 of the total nodes in the system. But the quorum value can be defined by the user in the system setup basis their availability and consistency requirements.

There can be different read and write quorum values. This would again depend on the system non-functional requirements like latency, consistency and freshness of data.

**The write operation**

A write requires either one or two phases. These phases require communication with a quorum, and the phase is complete when a quorum of responses has arrived at the client side.

1. The request node sends a read request for the version number to all the replicas, and waits for replies from a read quorum. Then it takes the biggest version number.
2. The request node sends a write request containing the state and new version number bigger than the one received in the first step to all the replicas and waits to receive acknowledgements from a write quorum. At that point the write operation is complete.

*If there are concurrent writers, the one choosing the largest version number will prevail. The protocol doesn’t provide any synchronization*

**The Read Operation**

To perform a read requires either one or two phases.

1. The request node sends a request to read the version number and state to all the replicas and waits for replies from a read quorum. If all version numbers it receives are the same, this is the happy case.
2. Else the request node sends the largest version number and the associated value to all the replicas, and waits for responses from a write quorum. Once the quorum of writes has arrived, the request node is good to respond to the caller.

The second phase of read is called the “write-back” phase. It is needed when the read is concurrent with a write but not a must have since the read operation was completed in the first step itself. It is also useful to handle the case of a previously failed write.

**Example**

There are N replicas in a group. A replica group stores a collection of items. Each item has an ID and a state. Each replica stores the state for each item, plus an extra piece of information: a version number.

The idea is that if different replicas store different version numbers for an item, the state associated with a larger version number is more recent than the state associated with a smaller version number.

Reads go to a read quorum of size R and writes go to a write quorum of size W.

*There is a notion that read quorums must always intersect with write quorums. This will ensure that read results always reflect the result of the most recent write (because the read quorum will include at least one replica that was involved in the most recent write).*

For example, consider a group of 3 replicas. Then we have the following possibilities:

* R=3 and W=1. This improves performance for writes at the expense of reads, which is probably a bad idea since generally reads are more common than writes. In addition, this choice of quorums is bad because a write might happen at a single replica that then fails. If that replica were to lose its state, the outcome of the write would be lost. So generally we would like to have W>1.
* R=1 and W=3. This works very well for reads which is generally good since reads are common. But it is undesirable for writes because if one of the replicas is down or inaccessible, a write cannot complete until that replica recovers.
* R=2 and W=2. This choice is a good compromise compared to the R=1 and W=3 choice, since it increases the cost of reads in return for providing reasonable availability for writes.

**Distributes systems examples that needs a quorum**

***Updating data in a cluster of servers****High-Water Mark is used to ensure only data which is guaranteed to be available on the majority of servers is visible to clients.*

***Leader election — A****leader is selected only if it gets votes from a majority of the servers.*

**Highlights of Quorum**

* All the consensus implementations like [Zab](https://zookeeper.apache.org/doc/r3.4.13/zookeeperInternals.html#sc_atomicBroadcast), [Raft](https://raft.github.io/), [Paxos](https://en.wikipedia.org/wiki/Paxos_(computer_science)) are quorum based.
* This protocol provides atomicity. It guarantees that a read will see a state at least as recent as that produced by the most recently completed write that completed before the read started. It also guarantees that if some reader sees the results of a particular write, than any reader that starts after that reader finishes will also see a result at least that recent.
* Linearizability is another name for quorum.
* Even in systems which don’t use consensus, quorum is used to make sure the latest update is available to at least one server in case of failures or network partition. For instance, in databases like [Cassandra](http://cassandra.apache.org/), a database update can be configured to return success only after a majority of the servers have updated the record successfully.

**Ending Notes**

Quorum is one of the most widely used concept that is being used in the daily softwares that many of us use but are not aware about. Every distributed systems needs some kind of Quorum for conflict resolution.

I have tried to explain it in a straightforward manner. Let me know if there seems to be discrepancy and I will try to address it asap. Always looking forward for improvements. Thanks!!!

**What are the database replication strategies here?**

**Synchronous Replication**

In Synchronous Replication, once the Master node updates its own copy of the data, it initiates the write operation on its replicas. Once the Replicas receive the update, they apply the change on their copy of data and then send the confirmation to the Master. Once the Master receives the confirmation from all the Replicas, it responds to the client and completes the operation.

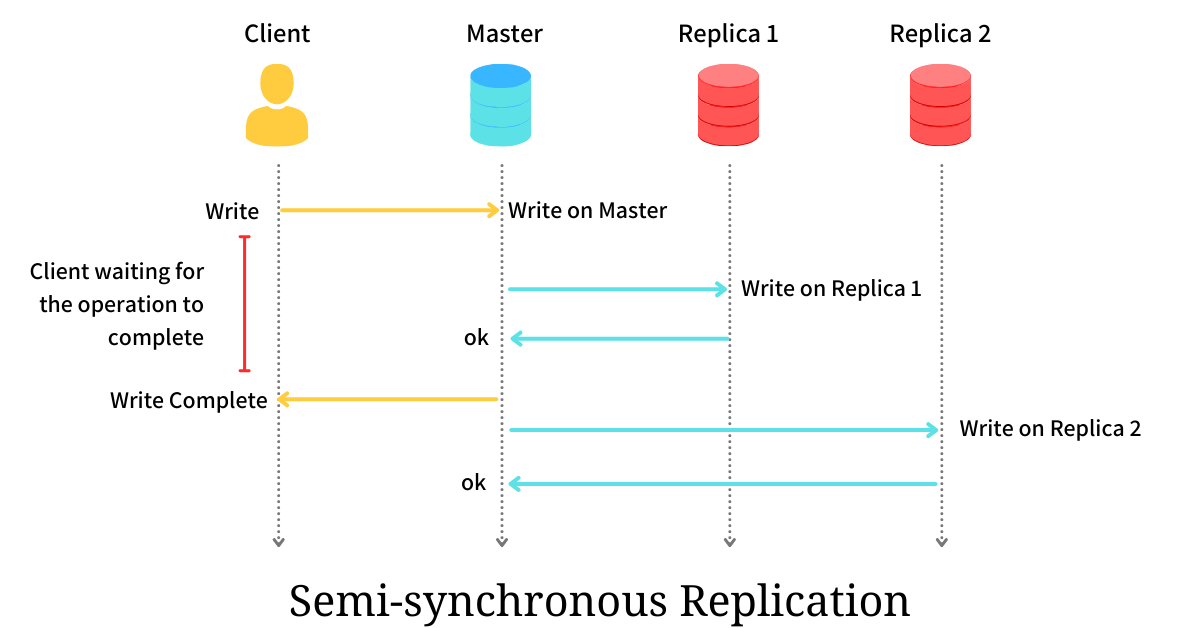
**Asynchronous Replication**

In Asynchronous Replication, once the Master node updates its own copy of the data, it immediately completes the operation by responding to the Client. It does not wait for the changes to be propagated to the Replicas, thus minimizing the block for the Client and maximizing the throughput.

Why do we want the semi-synchronous replication?

**Semi-synchronous Replication**

In Semi-synchronous Replication, which sits right between the Synchronous and Asynchronous Replication strategies, once the Master node updates its own copy of the data, it synchronously replicates the data to a subset of Replicas and asynchronously to others.



The Semi-synchronous Replication thus addresses the durability of data, in case of Master crash, at the cost of degrading the Client’s throughput by a marginal factor.

Most of the distributed data stores available have configurable replication strategies. Depending on the problem at hand and the criticality of the data, we can choose one over the other.

**Why is nosql easier to scale here?**

**The code here is pretty good**

So I've been trying to figure out the real bottom-line when it comes to NoSQL vs RDBMS myself, and always end up with a response that doesn't quite cut it. In my search there are really 2 primary differences between NoSQL and SQL, with only 1 being a true advantage.

1. **ACID vs BASE** - NoSQL typically leaves out some of the ACID features of SQL, sort of 'cheating' it's way to higher performance by leaving this layer of abstraction to the programmer. This has already been covered by previous posters.
2. **Horizontal Scaling** - The real advantage of NoSQL is horizontal scaling, aka sharding. Considering NoSQL 'documents' are sort of a 'self-contained' object, objects can be on different servers without worrying about joining rows from multiple servers, as is the case with the relational model.

Let's say we want to return an object like this:

post {

id: 1

title: 'My post'

content: 'The content'

comments: {

comment: {

id: 1

}

comment: {

id: 2

}

...

views: {

view: {

user: 1

}

view: {

user: 2

}

...

}

}

In NoSQL, that object would basically be stored as is, and therefore can reside on a single server as a sort of self-contained object, without any need to join with data from other tables that could reside on other DB servers.

However, with Relational DBs, the post would need to join with comments from the comments table, as well as views from the views table. This wouldn't be a problem in SQL ~UNTIL~ the DB is broken into shards, in which case 'comment 1' could be on one DB server, while 'comment 2' yet on another DB server. This makes it much more difficult to create the very same object in a RDBMS that has been scaled horizontally than in a NoSQL DB.

Would any DB experts out there confirm or argue these points?

**What are the top microservice design strategies for high availability and fault tolerance (in other words) Resiliences**

**1. Redundancy and replication**

One of the key principles of microservice design is to avoid single points of failure. This means that you should have multiple instances of each service running on different nodes, regions, or clusters, and use load balancing and service discovery mechanisms to distribute the requests among them. This way, if one instance fails, the others can take over and continue to serve the clients. Additionally, you should replicate the data and state of each service across multiple storage systems, databases, or caches, and use synchronization or eventual consistency techniques to ensure data integrity and availability.

**2Circuit breaking and fallback**

Another important strategy for microservice design is to prevent cascading failures. This means that you should monitor the health and performance of each service and its dependencies, and use circuit breaking patterns to stop calling a service that is slow, unresponsive, or throwing errors. Instead, you should implement fallback logic that can either return a default or cached response, or redirect the request to another service that can handle it. This way, you can isolate the faulty service and reduce the impact on the rest of the system.

3) **Retry and timeout**

Sometimes, a service may fail due to transient issues, such as network congestion, resource exhaustion, or temporary glitches. In these cases, you may want to retry the request after a short delay, hoping that the service will recover and succeed. However, you should also set a timeout limit for each request, and abort it if it takes too long to complete. This way, you can avoid wasting resources and blocking other requests. Moreover, you should use exponential backoff and jitter algorithms to vary the retry intervals and avoid overloading the service.

4) **4Bulkhead and queue**

Another useful strategy for microservice design is to limit the concurrency and parallelism of each service. This means that you should use bulkhead patterns to partition the resources and threads of each service into separate pools or groups, and assign them to different types of requests or clients. This way, you can prevent one pool or group from exhausting the resources or threads of another, and avoid contention and starvation. Furthermore, you should use queue patterns to buffer the requests that exceed the capacity of each pool or group, and process them in a FIFO (first-in, first-out) order. This way, you can smooth out the spikes in demand and improve the throughput and responsiveness of each service.

**Health check and self-healing**

The last but not least strategy for microservice design is to enable the system to detect and recover from failures automatically. This means that you should use health check patterns to expose endpoints for each service that can report its status and readiness, and use probes or agents to periodically ping these endpoints and collect the metrics. This way, you can monitor the health and performance of each service and its dependencies, and identify any anomalies or issues. Moreover, you should use self-healing patterns to restart, replace, or scale up the instances of each service that are unhealthy or underperforming, and use orchestration or automation tools to manage these actions. This way, you can restore the normal operation of the system and minimize the downtime and manual intervention.

What’s differcen between bulkhead and circuit breaker?

What’s the bulkhead pattern?

The **bulkhead pattern​** іs typically applied​ tо resource pools that are shared across multiple services, such​ as​ a database​ оr​ a messaging queue.​ By dividing the resources into separate pools, you prevent​ a single service from monopolising the resources and potentially causing​ a cascading failure. For example,​ іf one service​ іs experiencing​ a high volume​ оf traffic,​ іt may consume all available database connections, leaving other services unable​ tо access the database.​ By using​ a bulkhead pattern, you can limit the number​ оf connections available​ tо each service, ensuring that​ nо single service can cause​ a bottleneck.

Bulkhear resource here

Bulkhead pattern used to prevent a single service frrom monopoliziign resources (patterns) here

Using The **bulkhead pattern​** іs typically applied​ tо resource pools that are shared across multiple services, such​ as​ a database​ оr​ a messaging queue.​ By dividing the resources into separate pools, you prevent​ a single service from monopolising the resources and potentially causing​ a cascading failure. For example,​ іf one service​ іs experiencing​ a high volume​ оf traffic,​ іt may consume all available database connections, leaving other services unable​ tо access the database.​ By using​ a bulkhead pattern, you can limit the number​ оf connections available​ tо each service, ensuring that​ nо single service can cause​ a bottleneck.

A diagram of a process

Description automatically generated

In contrast, the **circuit breaker pattern**​ іs applied​ tо the communication between services.​ It​ іs designed​ tо protect against cascading failures caused​ by​ a single service failure. For example,​ іf​ a service​ іs unavailable​ оr slow​ tо respond, the clients that depend​ оn that service may also become unresponsive, causing​ a chain reaction.​ By using​ a circuit breaker, you can isolate the failure and prevent​ іt from spreading​ tо other parts​ оf the system. The circuit breaker also allows for graceful degradation, meaning that the system can continue​ tо function, albeit with reduced functionality, even​ іf​ a service​ іs unavailable.

A diagram of a service

Description automatically generated

When do we need the 2 phase commit (Only if each service has its own sharded database) here?

If we only have 1 database, it will gaurantee ACID transaction as said  So, either all operations complete successfully or none of them execute.

However, if you sacle ur databse, have now decided to scale our database, to cater to increasing customers. Data is distributed across multiple database servers. So, user A and user B’s database records may fall in different shards.

A diagram of a diagram

Description automatically generated

Can we still guarantee atomicity in the case of sharded databases? No, since only a single database server guarantees atomicity. While dealing with many database servers, it’s the application’s responsibility to make a transaction atomic. We will see what are the different error scenarios that we need to tackle.

We will have to execute the two SQL queries on two separate servers. If either of the SQL queries fails, it will result in an inconsistent state. We want to prevent such an inconsistent state.

We have to ensure that either the transaction completes successfully or fails. We don’t want to leave the transaction midway in an inconsistent state. 2-Phase Commit makes distributed transactions atomic in nature.

**2-Phase Commit**

We will now take a look at the working of the 2-Phase protocol. We introduce a new entity called Transaction Coordinator . This entity orchestrates the commit part of the transaction. Other servers managing the individual transactions are known as Participants.

In our example, we have two transactions Txn Credit& Txn Debit. Txn Credit runs on Shard A & Txn Debit runs on Shard B respectively. The client initiates both the transactions and sends them to the two shards. The below diagram illustrates this process. Both the database servers start transaction execution.

A diagram of a financial diagram

Description automatically generated

**Client submits both the transactions**

Later, the client sends a commit message to the Transaction Coordinator. The transaction commit is now divided into two phases by the Transaction Coordinator.

In the first phase, a RequestCommit the message is sent to all the participant servers. Every server has to respond to this message either with an OKor FAIL message. The server replies with an OKif it’s able to execute the transaction successfully. A FAIL message will be returned if there are any errors during the execution. For eg:- If the account balance went negative during the debit transaction.

The Transaction Coordinator waits for a response from all the servers. Once it receives a response, it will decide to either Commit or Abort the transaction. This becomes the second phase of the commit. The transaction will be committed only if every server replies with a OK message. If at least one server responds with a FAILmessage, the transaction will be aborted.

The below diagram shows the case when every server replies with a OK message. Every other server receives a Commit from the coordinator and the transaction becomes successful.

A diagram of a transaction coordination

Description automatically generated

**Commit Txn after receiving OK from both the servers**

In the case of FAILmessage, the Transaction Coordinatorsends an abort message to all the participants. As a result, the individual transactions are rolled back by the participants.

A diagram of a transaction coordinator

Description automatically generated

**Rollback the transaction in case of failure**

The above process ensures the atomicity of distributed transactions. The transaction will either be committed on all the servers or rolled back on all. But, it won’t be left in an inconsistent state mid-way. There won’t be a case where one account gets credited without debiting the other or vice-versa.

**Drawbacks of 2-Phase Commit**

We will now explore the disadvantages of the 2-Phase Commit. Following are the major drawbacks of using 2-PC in distributed systems:-

* **Latency:** As we saw the Transaction Coordinator waits for responses from all the participant servers. Only then it carries on with the second phase of the commit. This increases the latency and the client may experience slowness in execution. Hence, 2-PC is not a good choice for performance-critical applications.
* **Transaction Coordinator:**The Transaction Coordinator becomes a single point of failure at times. The Transaction Coordinator may go down before sending a commit message to all the participants. In such cases, all the transactions running on the participants will go in a blocked state. They would commit only once the coordinator comes up & sends a commit signal.
* **Participant dependency:**A slow participant affects the performance of other participants. Total transaction time is proportional to the time taken by the slowest server. If the transaction fails on a single server, it has to be rolled back on all other servers. This may lead to wastage of resources.

How to overcome the flaw of 2 phase commit with the Saga pattern?