medium.com

Distributed Transactions & Two-phase Commit - Geek Culture - Medium

Animesh Gaitonde

8-10 minutes

Anatomy of Two-phase commit

Disk

Introduction

In today’s world, data is growing at an enormous rate. Enterprises have built innovative solutions to handle a humongous amount of data. It’s not uncommon to see data distributed across many machines or databases. This technique is known as ‘***Sharding***’, helps in building scalable & reliable systems.

Tech companies also are adopting a microservices architecture. In this type of architecture, every microservice manages its own database. To add or modify data in a different database, it calls the responsible microservice.

Distributing data across many machines comes up with its own set of challenges. Data management for a monolithic non-sharded system is straightforward. Relational Databases such as Postgres, MySQL, etc. offer A.C.I.D properties out of the box.

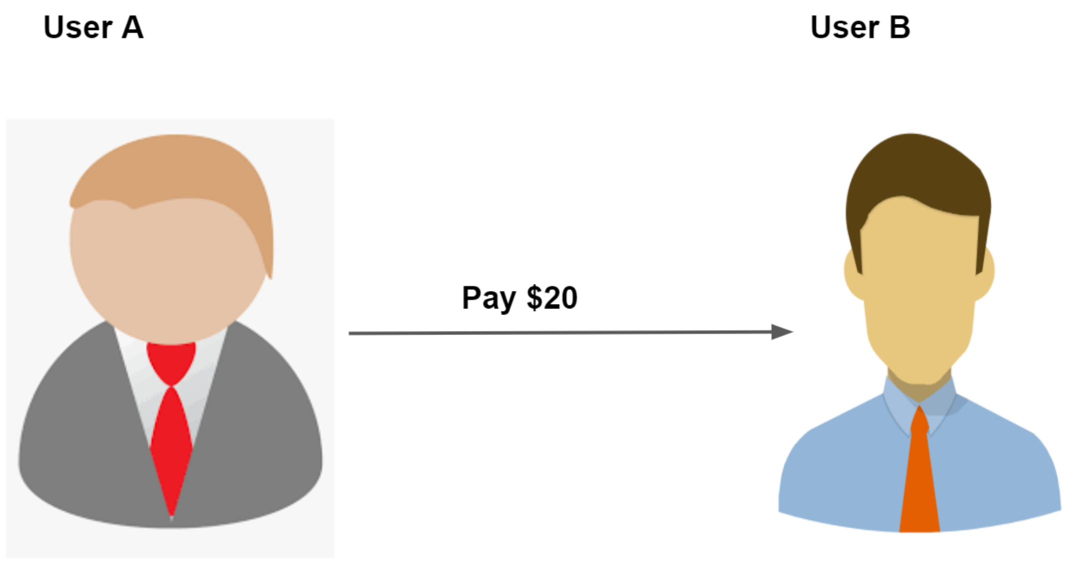
The same isn’t applicable for a sharded database or data distributed across microservices. In this article, we will see understand atomicity while handling distributed transactions. We will take a look at a protocol called 2-Phase Commit, that helps us achieve the same. So, let’s get started.

Monolithic system & non-sharded database

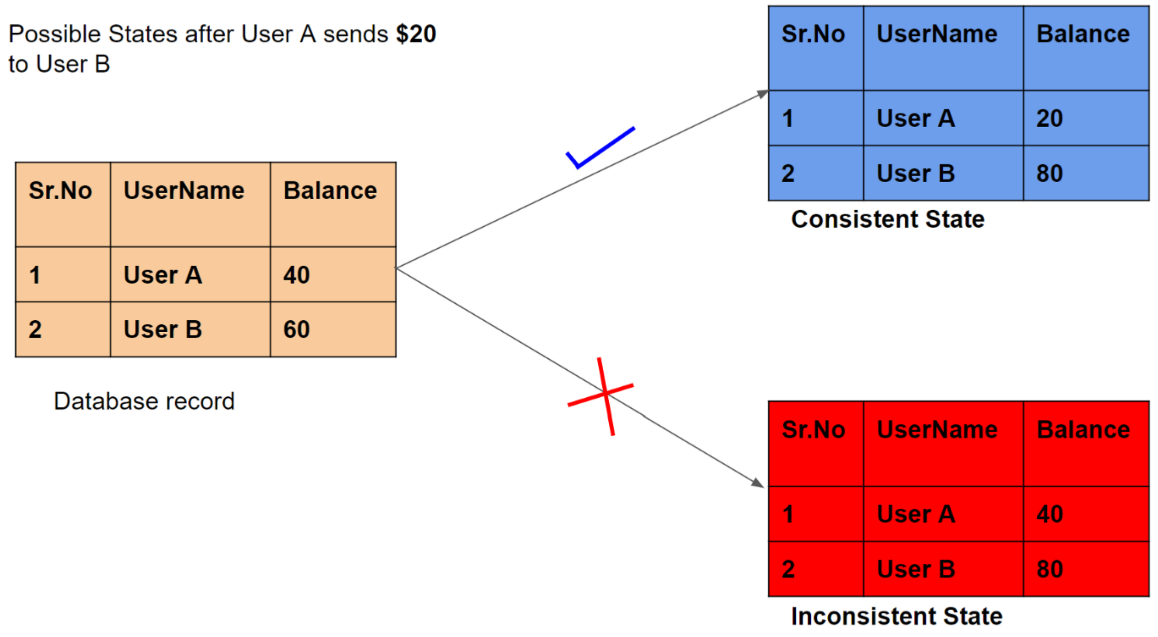
Let’s take a simple example of a monolithic banking application. This system interacts with a single database server managing multiple tables. Assume that the database is managing user’s balances. The application is responsible for handling user’s bank transactions.

When a user `A` transfers money to the user `B`, we need to ensure the following things:-

1. If the transactions succeeds, the system must credit the user `B's` account & debit user `A's` account
2. The database server may crash after the completion of the transaction. However, it must go back to its state before the crash
3. The transaction may fail due to multiple reasons. For eg:- the user `A` may not have sufficient balance. In this case, the accounts of both the users shouldn’t be updated
4. The database needs to be in a consistent state after the transaction completion. For eg:- user `B` shouldn’t receive the credit without the user `A` getting the debit



Transaction of 20$



Sr.No	UserName	Balance
1	User A	40
2	User B	60

Sr.No	UserName	Balance
1	User A	20
2	User B	80

Sr.No	UserName	Balance
1	User A	40
2	User B	80

Possible states after the transaction

If you use a relational database, it will guarantee all the above four points. Relational databases use transactions to achieve the same. The transaction is an abstraction & it encapsulates a unit of work. Transactions guarantee atomicity in a database. So, either all operations complete successfully or none of them execute.

In simple words, a transaction is a set of SQL statements, that a database can execute. Database executes every SQL statement. In case there is a failure, it will abort the transaction. When the transaction is aborted, no change is done on the underlying data. From the state perspective, it’s equivalent to not executing any statement.

If all the statements execute, the transaction is committed. Once a transaction is committed, the underlying data is modified & persisted.

For our above example, the database transaction will consist of the following statements :

Beginupdate set balance = balance + 20 where user = ‘B’;update set balance = balance - 20 where user = ‘A’;Commit

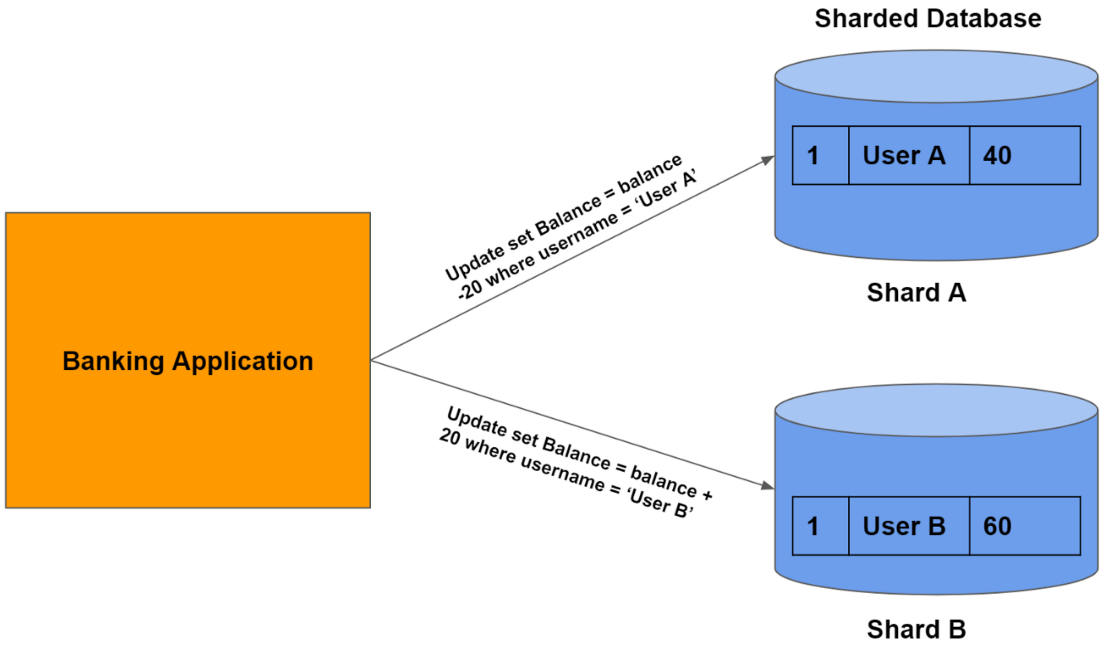
Assume that the initial balance of user `A` and `B` are 40 $ & 60 respectively. Following are the possibilities while executing the above transaction:-

- **Success** - In this case, the transaction will be committed. User `A's` balance will be 20 $ & user `B's` balance will be 80$. If the database crashes after this, it will come back to this same state after recovery.

- **Failure** - If there is a failure while updating the user `A's` balance, the database will abort the transaction. And it will rollback all the changes. The user's balance won't be affected.

Sharding the Banking database

We 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.


Sharded Database
1 | User A | 40
Shard A
Banking Application
Update set Balance = balance -20 where username = 'User A'
Update set Balance = balance + 20 where username = 'User B'
1 | User B | 60
Shard B

Sharded Database

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.

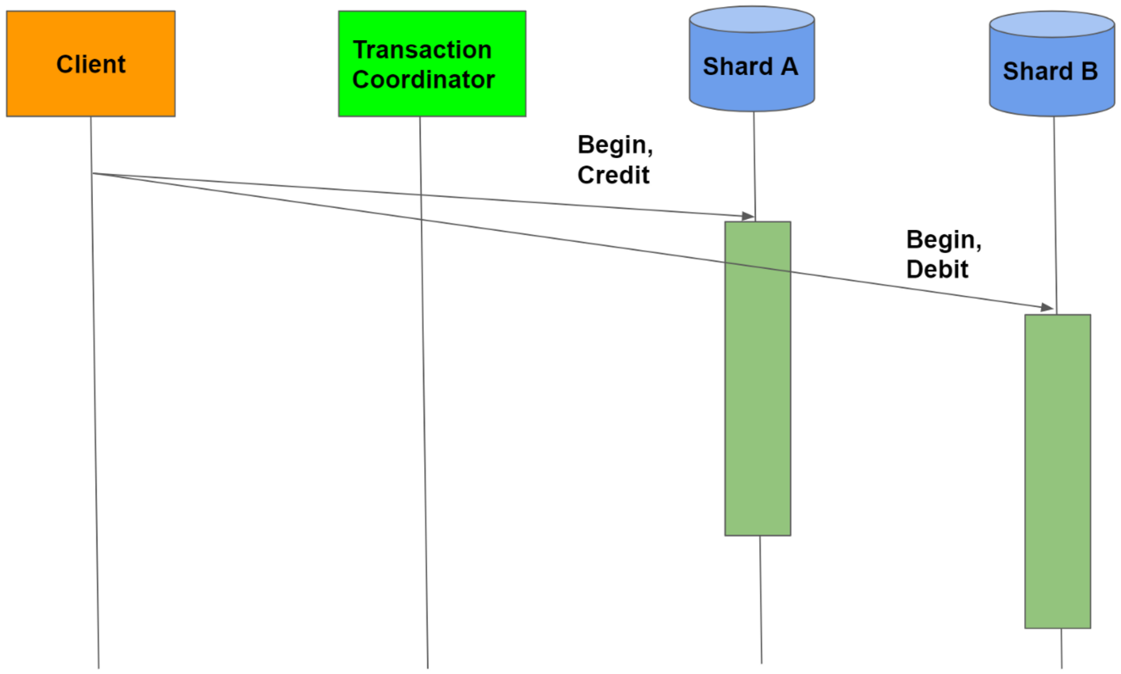
We have to ensure that either the transaction completes successfully or fails. We don't want to leave the transaction midway in an inconsistent state. 2-Phase Commit makes distributed transactions atomic in nature.

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.



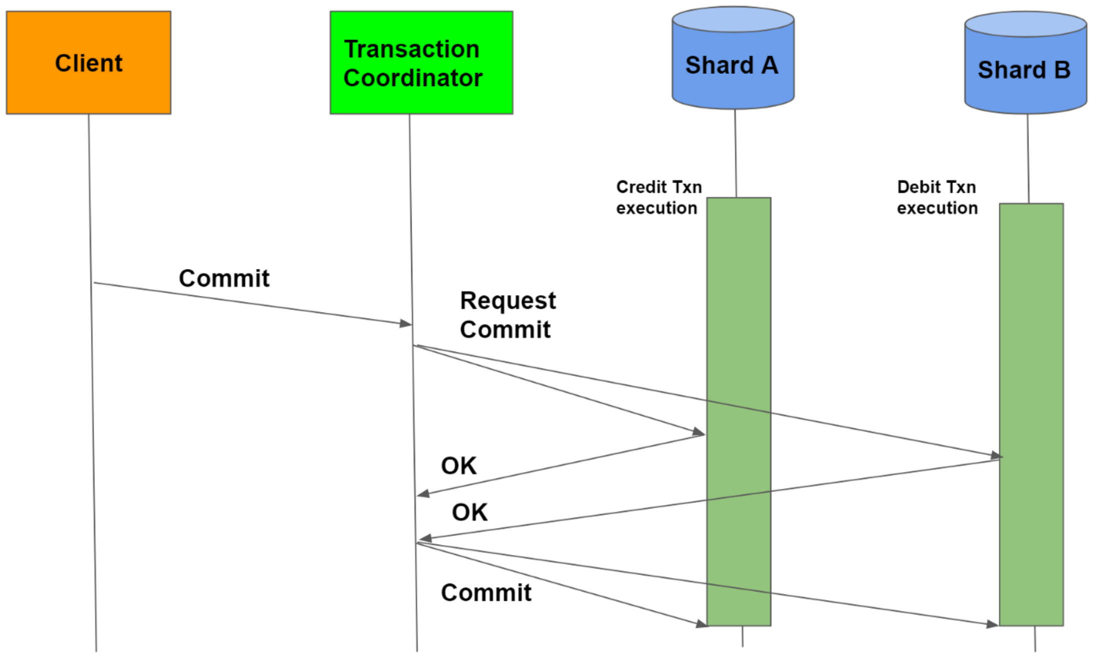
Client submits both the transactions

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 `OK`or `FAIL` message. The server replies with an `OK`if 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.

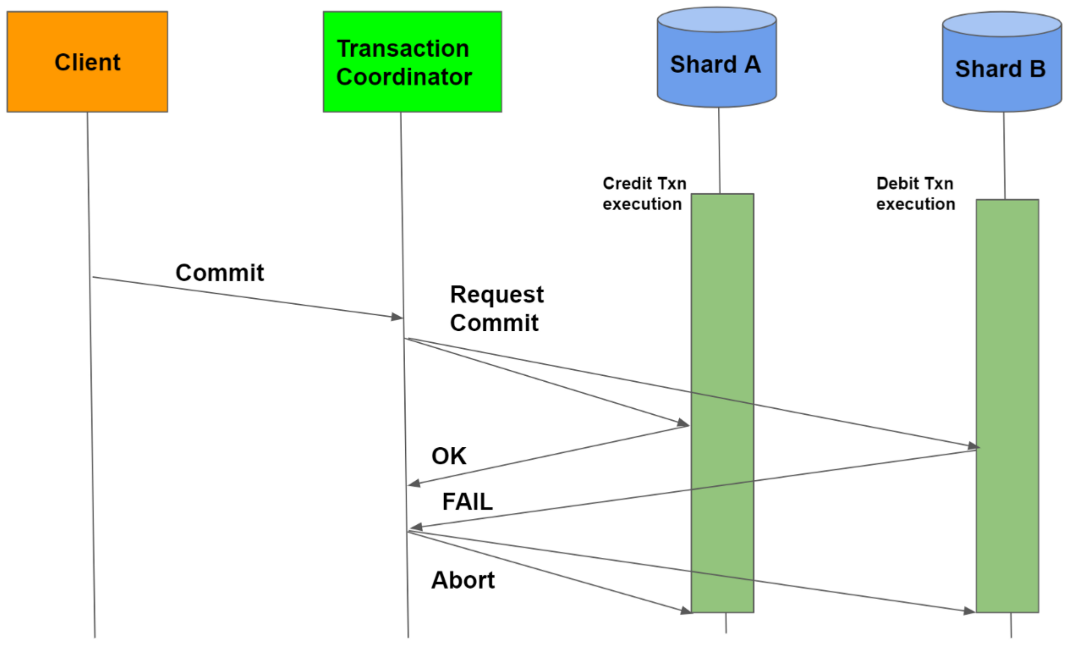
The `Transaction Coordinator` waits for a response from all the servers. Once it receives a response, it will decide to either Commit or Abort the transaction. This becomes the second phase of the commit. The transaction will be committed only if every server replies with a `OK` message. If at least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



Commit Txn after receiving OK from both the servers

In the case of `FAIL`message, the `Transaction Coordinator`sends an abort message to all the participants. As a result, the individual transactions are rolled back by the participants.



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.

All the above shortcomings of 2PC are over-come using the Saga pattern. Saga pattern relies on eventual consistency to handle distributed transactions. We will look at this pattern in another blog post.

References

- Handling Distributed Transactions in Microservices
- Distributed Transactions
- MIT Lecture — Distributed Transactions
- Cover Image
- Saga & Microservices