Relational Database

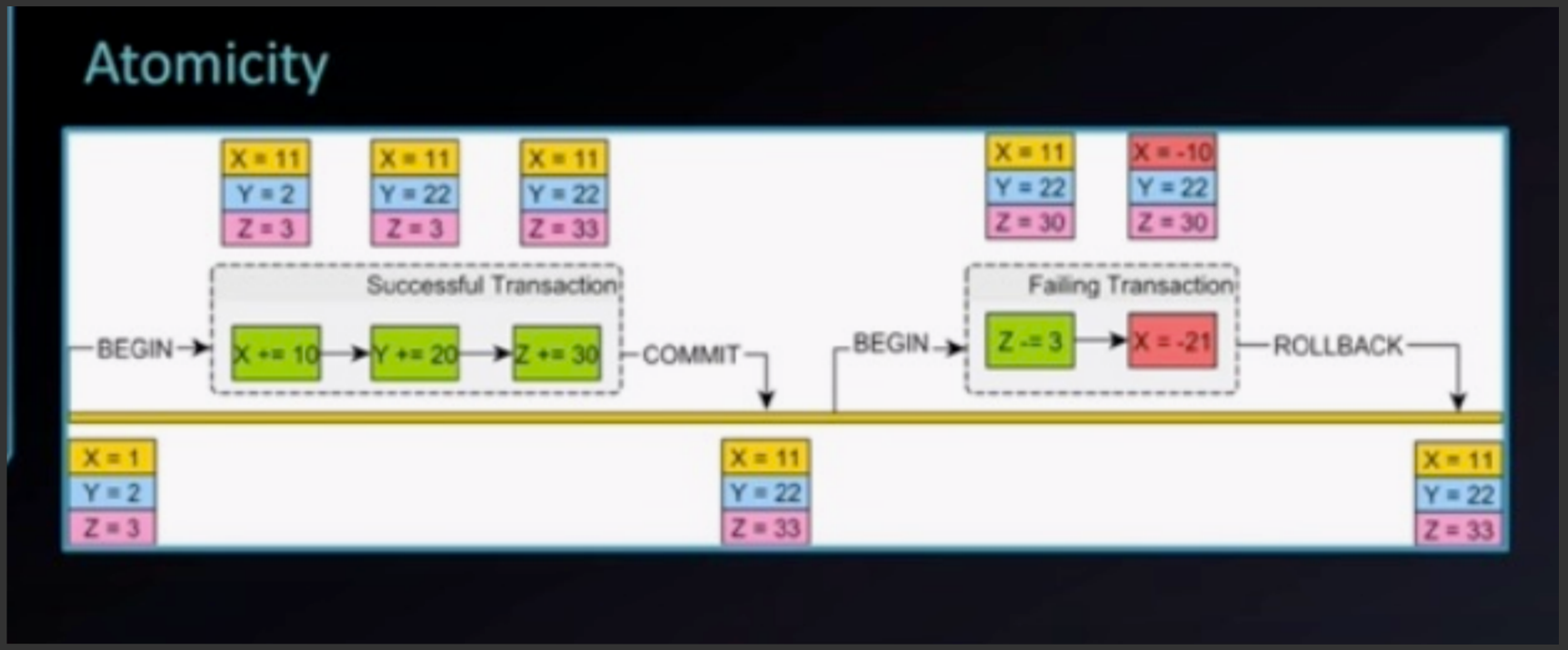
Represent data as rows and columns and also maintain relationship between entities through foreign keys. Based on the concept of Relational Algebra, sets, unions and joins

To operate effectively and accurately, it must support ACID transactions.

All of these have to do with updating the **state of the database**

Atomicity:

- Each transaction that is committed appears to be indivisible
- A transaction is said to be successful if all the steps involved in the transaction complete successfully
- If any part of the transaction fails, the whole transaction is marked as failed and none of the changes are committed to the database.

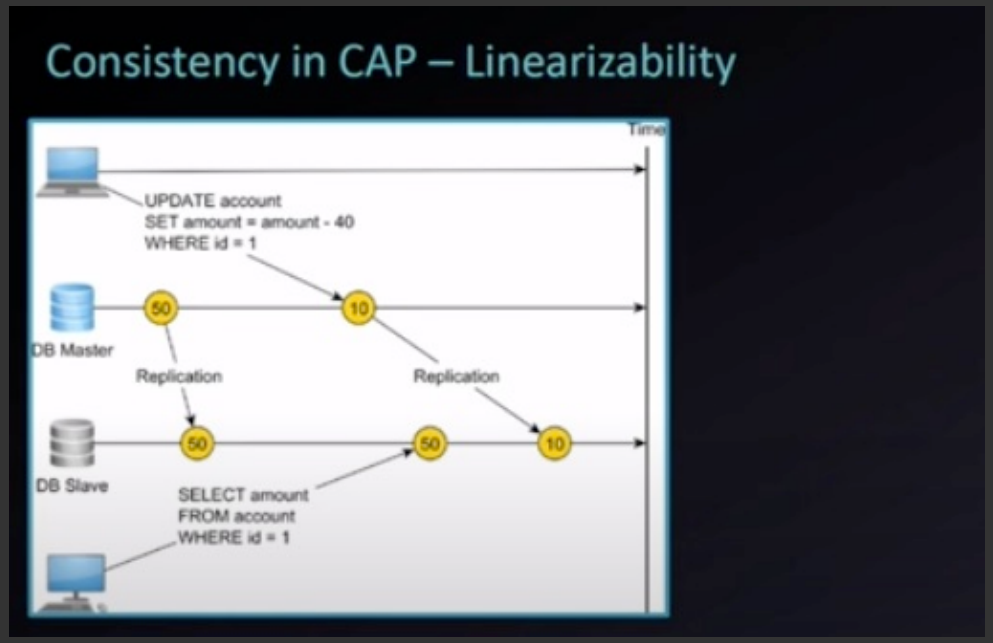


Consistency for RDBMS:

- This is especially important when multiple applications / services are reading from / writing to the same database, state updates created by one client should be accurately reflected for the other clients.
- Consistency can be ensured by checking if every transaction abides by the constraints mentioned for the table / database - column types, null ability, key constraints.. etc..
- Important while handling Write Write Conflicts

Consistency for in CAP theory:

- A better term is leniarizability
- In a master slave system we perform a change in the master. If the system is leniarizable, and if we read values from the slave after a short point in time, we get the updated value, if not we get a stale value.
- This is consistency in CAP theory - slave lags behind master



Durability

- How database system generally works is that, when the database program is running, it creates an **in-memory buffers** of the actual data stored in the disk.
- And whenever changes are made to the database it is first updated in the in-memory buffer and then after a delay it is flushed to the disk.
- Why is there an in memory layer and why transactions are not immediately flushed to the disk ? : flushing creates a lot of Random Access writes to the disk that are very slow. Therefore database applications only flush **in memory data** to the database at every **checkpoint** - which could be from one minute to every couple minutes - depending on the configuration.

What happens if the power outage happens before data is flushed to the DB ?

This is where **read logs** come into the picture, whenever a transaction is completed processing, it is also written to the read log. So thereby whenever the database is brought up, it can synchronize itself by accessing the read logs.

Serializability and Concurrency and Isolation

- Interleaving concurrent transaction statements so that the outcome is equivalent to a serial execution
- Dealing with conflicts
  - Avoid concurrency : example VoltDB - in memory, single thread, guarantees serializability
  - Control it : conflict avoidance, conflict detection - includes locking - row / cell
    But there is a trade off with locking, which is lesser through put

AMAZON RDS

AUTOMATED BACKUPS

- Point in time recovery
- Automated backups enabled by default - take full daily snapshot and also store transaction logs throughout the day
- Terminated when we terminate RDS

MANUAL SNAPSHOTS

Are stored even if we delete the original RDS instance

Restored backups always result in a new instance with a new end point

Multi AZ deployments

- This improves the availability and durability of the database service
- Automatically assigns a **primary instance** - where all writes go into
- Data is synchronized with replicas in the other AZs
- In case of infra failure, RDS automatically performs a **failover to the standby** and operations can resume as soon as this is completed, the connection details do not change so there is no need from the application side to reconfigure database connection in this case

Standbys and Read - Write Replicas

Apart from service as a backup the standbys can be used to serve other purposes

- In case of use cases where there are **substantially more read requests** than write, these standbys can **serve as read replicas** and lesser traffic can be diverted to the write replicas
- If the standby is not lagging behind much from the primary, then the standby instance can be used for **taking automated snapshots** of the database instance, this **reduces lag in the primary** while the snapshot is being taken
- Also in case of database software updates, and if the standby is not lagging much, the **update can be applied to the standby first**, then the standby is promoted to be the primary and then the primary is demoted to be the standby.

Final Notes regarding read replicas and lagging

Read replicas will always be in a lagging state and the db has a track of what is lagging. So when we fetch data, no response is returned if the data in the read replica is stale. This also sort of falls into the eventual consistency model.

If the read replica lags too much behind the primary then we will have to decommission the instance

This is where monitoring systems come in the picture

SPLIT BRAIN PROBLEM IN DISTRIBUTED SYSTEMS

Whenever we have auto incrementing keys for databases, in case the different instances go out of sync, the keys for the same data in both the databases will have different values. This will become an issue - hence it is always better to not use auto incrementing keys as the primary key in databases.

This also sort of feeds into the problem of having distributed RDBMS, the only true way to scale a database instance is vertical scaling, as it is easier to manage data that has a single source of truth

Pros and Cons of RDBMS

Pros	Cons
• Everyone Knows SQL • Legacy users are all on SQL • ACID compliant • Store large amounts of persistent data	• Mediocre horizontal scaling support • Select queries make full table scans • Mismatch on how data is stored and how it is actually consumed in the app layer - overcome by using ORM frameworks

NOSQL Databases

These databases

- Run well on a cluster / horizontally scaled setup
- Schema need not be explicitly defined in the database

Trade offs:

- Additional logic to maintain schema is shifted to the application layer
- Maintaining consistency is now the work of the application

Having read the above, it looks like NoSQL databases provide good availability over consistency given their ability to be horizontally scaled.

Consistency Availability and Partition Tolerance

CAP theorem states that, **in presence of a network partition** it is impossible for a distributed computer system to provide more than two of the following:

- Consistency: Every read receives the most recent write or an error (No wrong output)
- Availability: Ever request either receives a non-error response without guarantee that it will contain the most recent write
- Partition Tolerance: The system continues to operate even if there is an arbitrary number of messages being dropped in transit across the network

JUST BECAUSE SOMETHING FEELS COOL IT ISN'T RIGHT

So in presence of a network partition, one has to choose between consistency or availability. (Most applications that prefer NoSQL databases go towards availability)

Concurrency Control Mechanisms

Pessimistic: Block the operation of a transaction if it may cause a violation, the blocked transaction is not performed until the possibility of a violation disappears. If too many transactions are being aborted, being optimistic is the good way.

Optimistic: Let the transaction proceed and check for possibility of violation at the end, if there is a possibility of a violation, restart and re execute the transaction.

Resume studying from slide 80