NoSQL - Scaling Out / Up

Welcome to the session on NoSQL

In this session, we will learn What, why and how about NoSQL.

What does it mean by scale out and scale up?

Say, we have a cart with a single horse and we need to carry more than a horse can handle.

We can replace the horse with an elephant and carry a higher load. This approach is called scaling up or vertical scaling. We may not be able to carry load higher than an elephant can carry.

If we add more horses to the cart, it is called horizontal scaling or scaling out.

Basically, if we have to handle higher computing load and we choose to upgrade the hardware, we call it vertical scaling or scaling up.

But if we choose to move to a distributed architecture and add more computers to solve our problem we call it horizontal scaling or scaling out.

NoSQL databases are needed when we want to horizontally scale.

NoSQL - ACID Properties and RDBMS Story

NoSQL - ACID Properties and RDBMS Story

ACID - Another term that we frequently use while talking about relational databases is ACID properties of the database.

ACID is an acronym of Atomicity, Consistency, Isolation and Durablity.

Atomicity means transaction either completes or fails in entirety. There is no state in between. No body sees a partial completion of a transaction.

Consistency means the transaction leaves the database in the valid state.

Isolation means no two transactions mingle or interfere with each other. The result of two transactions executed in parallel would be same as sequential execution.

Durability means the changes of the transaction are saved. It remains there even if power is turned off.

Every relational database such as MySQL, postgresql, oracle and microsoft sql guarantees ACID properties of transaction.

Lets me take you through a typical story before NoSQL.

On the initial public launch, we would move a database from local workstation to shared, remotely hosted MySQL instance with a well-defined schema. This is our beginning.

Soon, enough our service becomes popular and we have a problem: there are just too many reads hitting the database.

This is quite usual with any server. And the solution is pretty simple - start caching frequently executed queries. We generally use memcached for caching. But note that the reads are no longed ACIDic. The moment we have two places for data, it generally becomes inconsistent. It would sometimes update the cache and till then cache would be serving older data.

Still, if service continues to grow in popularity. Then too many writes would hit the database. To solve this, if we have money, we would vertically scale our hardware to say 16 cores processor, 128 GB of RAM and banks of really fast hard drives.

Soon enough, new features would increase query complexity. We will have too many joins. If you take a look at Amazon details page, there are more than 100 features on the page. If each feature was in its own table, to prepare such page, 100s of joins would be required.

So, you start denormalizing to avoid joins between tables. Yes, denormalizing. If you are from DBA background, this is going to be a very hard moment.

If your service further goes popular, it would swamp the server. Things would become too slow. So, we should stop doing server side computations such as stored procedures and move those to the client side. For example, date time computations.

Ever after this, there would be some queries that are still slow. So, we periodically prematerialize the most complex queries and try to stop joining in most cases. Pre-materializing means keeping the results of the queries that are often required ready.

Now, the reads might be okay but writes are getting slower and slower. So, you drop secondary indexes and triggers.

Now, at this point you will realize that you db is left with: 1. No ACID properties due to caching 2. No Normalized schema 2. No stored proceduces, triggers and secondary indexes

Now the question is "why do we need relational database then?"

Thats why NoSQL databases originated.

NoSQL - Types of NoSQL Stores

NoSQL Stores

What Is a NoSQL Datastore?

It is a datastore that store and handle really big data and it provides High availability - which means serving to many concurrent users.

NoSQL achieves this with scale out architectures which means we can have many machines and these machines can be commodity hardware. Every NoSQL supports the addition of hardware whenever needed.

[other]

The other properties of NoSQL data store is that they are non-relational meaning they don't guarantee ACID properties and don't to adhere to a fixed schema.

All of the NoSQL datastores available in market at open source? Why? To avoid vendor lockins, the users prefer open source data store.

[types] Every NoSql stores records. Based on the type of record a NoSQL can store, it is classified into four categories: First one is Column Oriented or Wide Column

These are very close to relational databases. They have tabular structure. Examples of such databases are HBASE,Cassandra and accumulo.

The second kind is Document oriented NoSQL datastores. In such data stores, we can store complex objects. For example, we can store TVShows into such datastores where each record would represent a tv show. The TV show is a complex object or we can say document. TV Show has many seasons, each search has many episodes. Every episode has many reviews and cast members.

Examples of Document oriented NoSQL datastores are MongoDB, couchbase, clusterpoint and marklogic.

The Third kind of NoSQL stores are key-value store. Such NoSQL store reach record in the form of key-value. Key and Value can be primitives such as string, byte array, number or boolean. Key-Value stores are most primitive and the first datastores to be invented.

The examples of key-value stores are: Amazon dynamo, memcachedb, voldemort, redis and riak.

The fourth category of NoSQL datastores is Graph Oriented data stores. These are the most recent kind of datastores.

Such NoSQLs are used for storing graphs. A graph is made up complex relations between objects for example on Facebook relations between people.

Allegro, neo4j, orient db, virtuoso and giraph are examples of NoSQLs.

NoSQL - CAP Theorem

CAP theorem

It is very important to understand the limitations of NoSQL database. NoSQL can not provide consistency and high availability together. This was first expressed by Eric Brewer in CAP Theorem.

CAP theorem or Eric Brewers theorem states that we can only achieve at most two out of three guarantees for a database: Consistency, Availability and Partition Tolerance.

Here Consistency means that all nodes in the network see the same data at the same data. Or a reader gets most recently written data. Availability is a guarantee that every request receives a response about whether it was successful or failed. The more number of users a system can cater to better is the availability.

Partition Tolerance is a guarantee that the system continues to operate despite arbitrary message loss or failure of part of the system. In other words, even if there is a network outage in the data center and some of the computers are unreachable, still the system continues to perform.

Out of these three guarantees, no system can provide more than 2 guarantees. Since in the case of a distributed systems, the partitioning of the network is must, the tradeoff is always between consistency and availability.

As depicted in the Venn diagram, RDBMS can provide only consistency but not partition tolerance. While MongoDB, HBASE and Redis can provide Consistency and Partition tolerance. And CouchDB, Cassandra and Dynamo guarantee only availability but no consistency. Such databases generally settle down for eventual consistency meaning that after a while the system is going to be ok.

Let us take a look at various scenarios or architectures of systems to better understand the CAP theorem.

The first one is RDBMs where Reading and writing of data happens on the same machine. Such systems are consistent but not partition tolerant because if this machine goes down, there is no backup. Also, if one user is modifying the record, others would have to wait thus compromising the high availability.

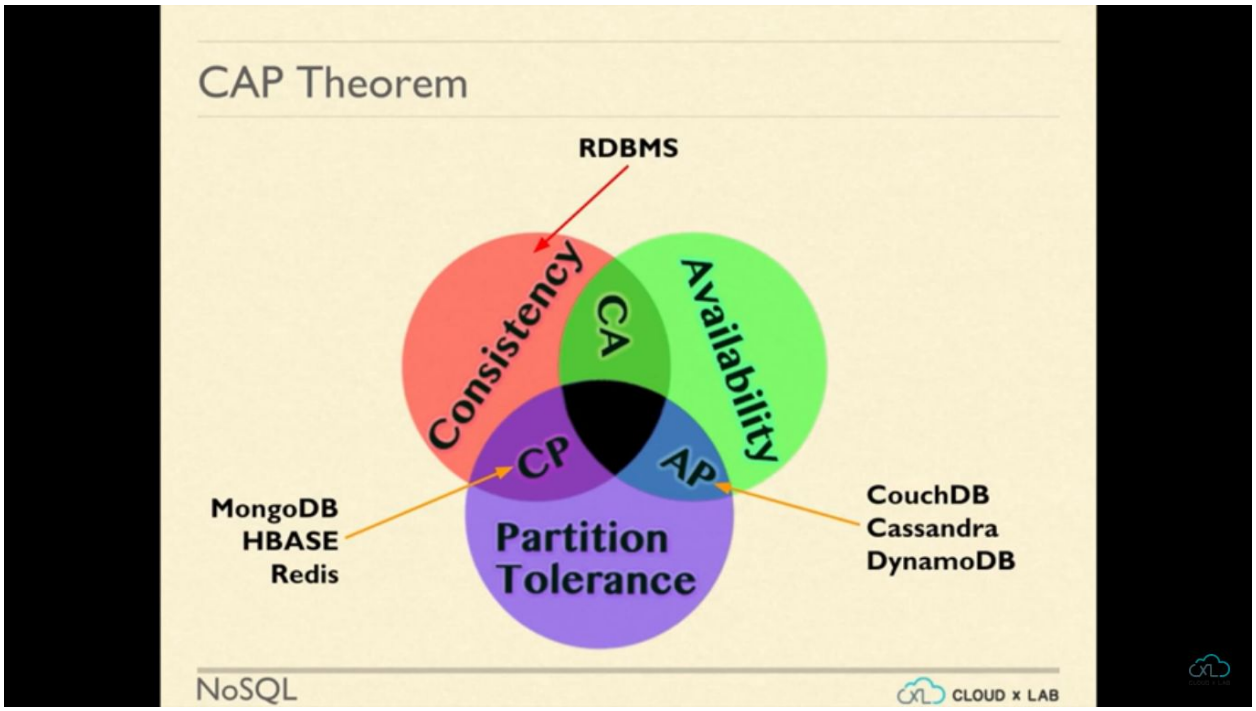
The second diagram is of a system which has two machines. Only one machine can accept modifications while the reads can be done from all machines. In such systems, the modifications flow from that one machine to the rest.

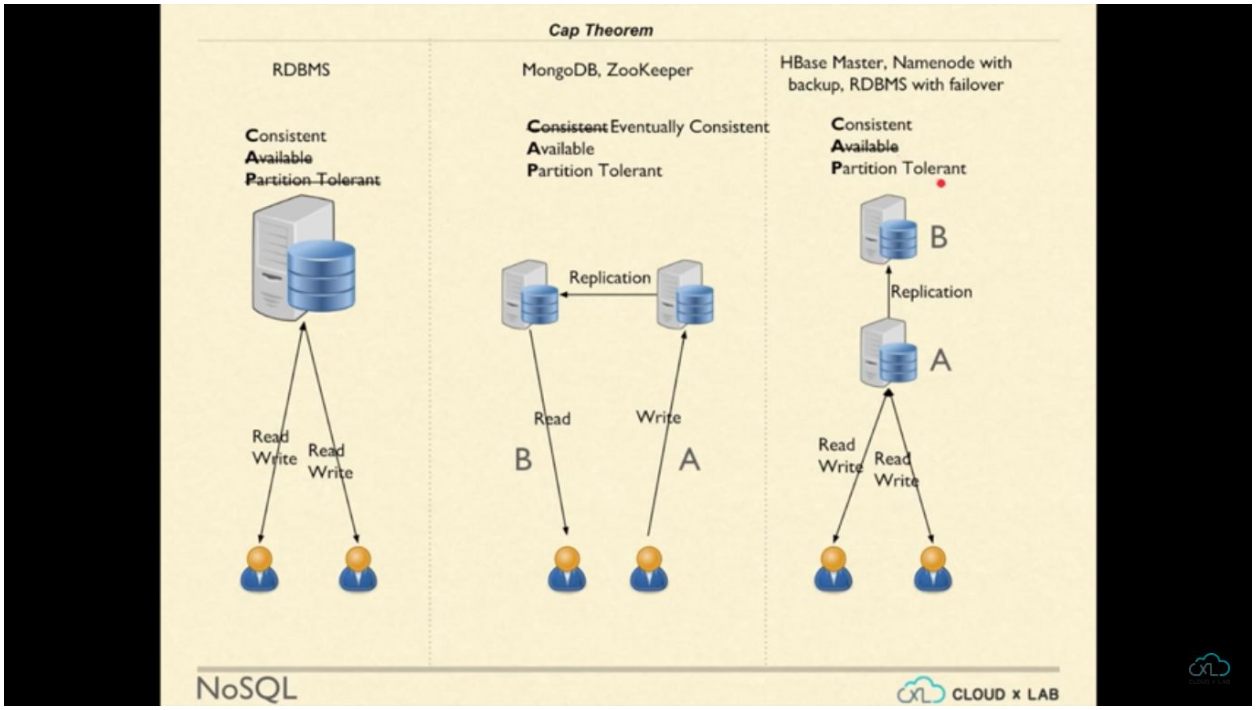
Such systems are highly available as there are multiple machines to serve. Also, such systems are partition tolerant because if one machine goes down, there are other machines available to take up that responsibility. Since it takes time for the data to reach other

machines from the node A, the other machine would be serving older data. This causes inconsistency. Though the data is eventually going to reach all machine and after a while, things are going to okay. There we call such systems eventually consistent instead of strongly consistent. This kind of architecture is found in Zookeeper and MongoDB.

In the third design of any storage system, we have one machine similar to our first diagram along with its backup. Every new change or modification at A in the diagram is propagated to the backup machine B. There is only one machine which is interacting with the readers and writers. So, It is consistent but not highly available. If A goes down, B can take A's place. Therefore this system is partition tolerant.

Examples of such system we are HDFS having secondary Namenode and even relational databases having a regular backup.





NoSQL - Column Oriented Databases

NoSQL - Column Oriented Databases

What do we mean by serialization in computing? Serialization is a process of converting objects into an array of bytes.

As you can see in the diagram, the object on the left side has name, company and gender gets converted into an array of characters.

Once an object is serialized, the bytes can be saved or transferred and then the object can be reconstructed later.

The process of constructing objects from a sequence of bytes is called de-serialization.

While talking about NoSQL databases, a term we would often use is Column oriented data formats or data stores. While converting a tabular data into a sequence of bytes, we can go column wise or row wise.

For example, we could convert this tabular data to 10 Joe 12 Mary 11 Cathy. This is called row oriented data format and the datastores that save the tabular data in this format are called row oriented data store. This is the traditional way of storing data.

If we store the example table as 10 12 11 Joe mary catchy, it is called column oriented data store or format. In column oriented data store, we first store the first column and then second column and so on.

Since the similar data comes together, the column oriented data formats generally offer better compression.

Certain data stores have come up with kind of hybrid storage called Column Family oriented data stores. In such stores, we group columns into column families. The data is stored column family wise in such data store. Though, the values under a column family are stored row wise.

For given example, CF1 will be stored first and then CF2. For CF1 and CF2, the data will be stored row-wise. So, the result will be 10 Joe 12 Mary 11 Cathy 23 33 45

Column family oriented data a very clever design because with column family oriented way, you can model your data either in row oriented, column oriented or hybrid formats. This provides a greater flexibility.

In first example, we have one column family per column and in the second example, we have a single column family for all columns.

You can observe that first design behaves like a column oriented data store and the second one behaves as a row oriented data store.