# Introduction to NoSQL

Week-1

Part 1: Basics of NoSQL
Part 2: Working with Distributed Data

## Overview of NoSQL

### What is NoSQL?

- The NoSQL name was introduced during an open-source event on distributed databases
- It doesn't mean 'No SQL' – it means 'Not only SQL'

NoSQL

↓

'Not only SQL'

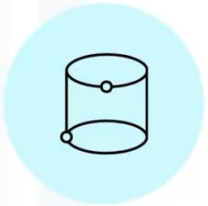
### What is NoSQL?

- Refers to family of databases that vary widely in style and technology
- However, they share a common trait
  - Non-relational
  - Not standard row and column type RDBMS
- Could be referred to as 'Non-relational databases'

### What is NoSQL?

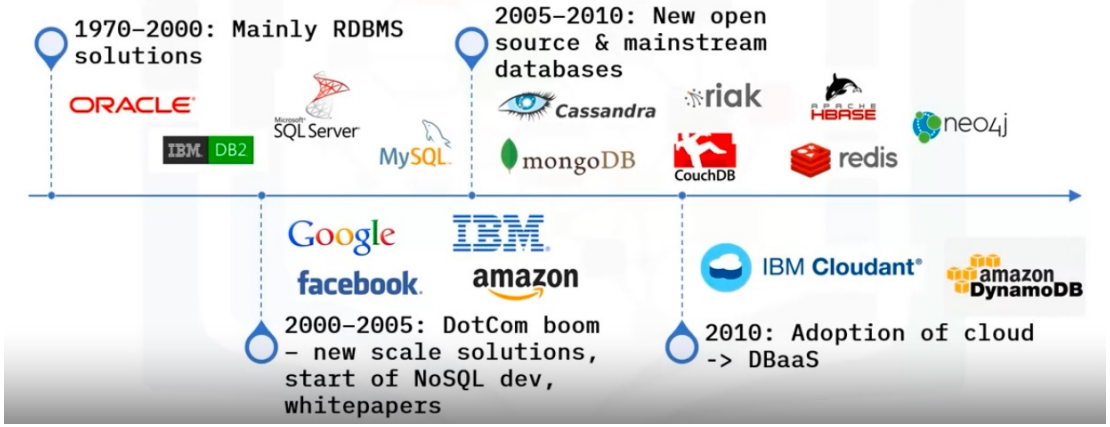
NoSQL databases:

- Provide new ways of storing and querying data
  - Address several issues for modern apps
- Are designed to handle a different breed of scale – 'Big Data'
- Typically address more specialized use cases
- Simpler to develop app functionality for than RDBMS



NoSQL Databases

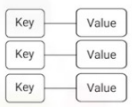
### History of NoSQL



## Characteristics of NoSQL Databases

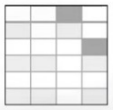
### NoSQL Database Categories

General consensus is...
...NoSQL databases fit into four categories



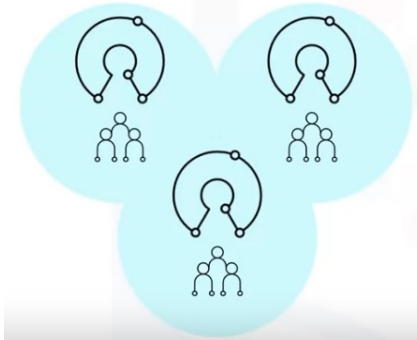
Key-Value


Document


Column


Graph

### NoSQL Database Characteristics


Open Source Communities

But what ties NoSQL databases together?

- Majority have their roots in the open source community
- Many have been used and leveraged in an open source manner
- Open source community support is fundamental to their industry growth

### NoSQL Database Characteristics

- Companies often develop a commercial version of the database alongside the open source version
- Examples include:

DATASTAX

IBM Cloudant®

mongoDB

- **commercial version** of the database, and services and support of the technology, at the same time providing sponsorship of the open source counterpart.
- Examples of this include
- IBM Cloudant for CouchDB,
- Datastax for Apache Cassandra
- Mongo has their own open source version of the Mongo database too.

## NoSQL Database Characteristics

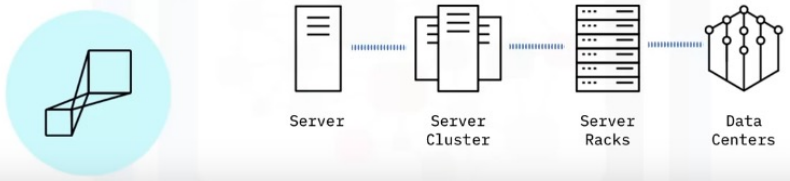
- They all differ technically speaking, but have a few commonalities
- Most NoSQL databases:
  - Are built to scale horizontally
  - Share data more easily than RDBMS
  - Use a global unique key to simplify data sharding
  - Are more use case specific than RDBMS
  - Are more developer friendly than RDBMS
  - Allow more agile development via flexible schemas

**Benefits of NoSQL Database**

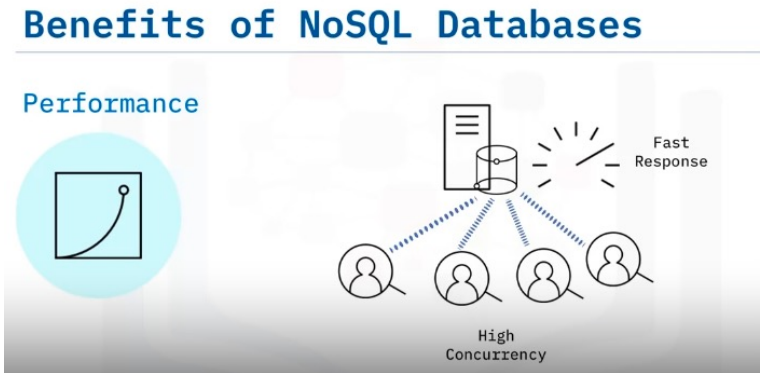
- Scalability
- Performance
- Cloud Architect
- High Availability
- Cloud Architecture
- Cost
- Flexible schema
- Varied Data Structures
- Specialized Capabilities

## Benefits of NoSQL Databases

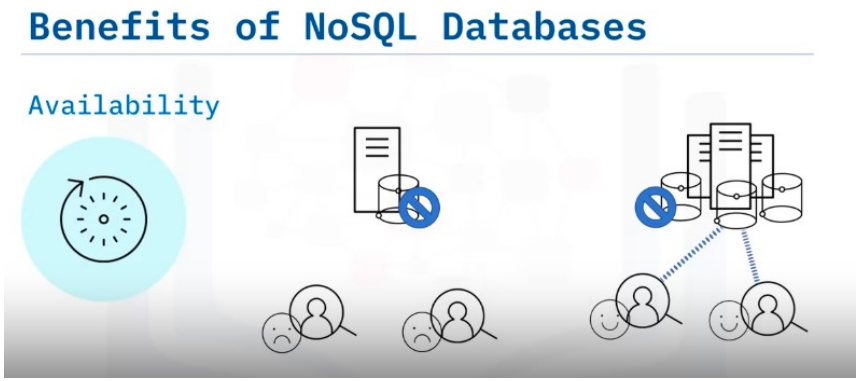
Scalability




- First, the most common reason to employ a NoSQL database is for scalability, particularly the ability to horizontally scale across clusters of servers, racks, and possibly even data centers.
- The elasticity of scaling both up and down to meet the varying demands of applications is key.
- NoSQL databases are well suited to meet the large data size and huge number of concurrent users that “Big Data” applications exhibit.

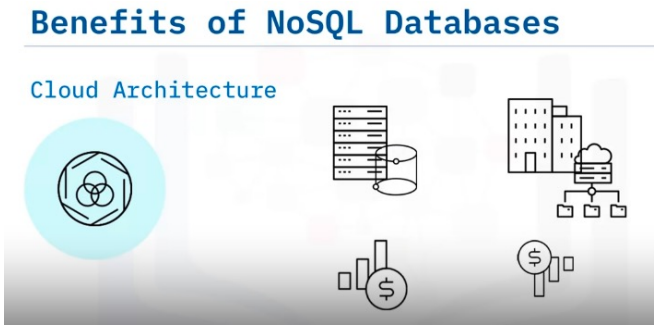



- The need to deliver fast response times even with large data sets and high concurrency is a must for modern applications, and the ability of NoSQL databases to leverage the resources of large clusters of servers makes them ideal for fast performance in these circumstances.

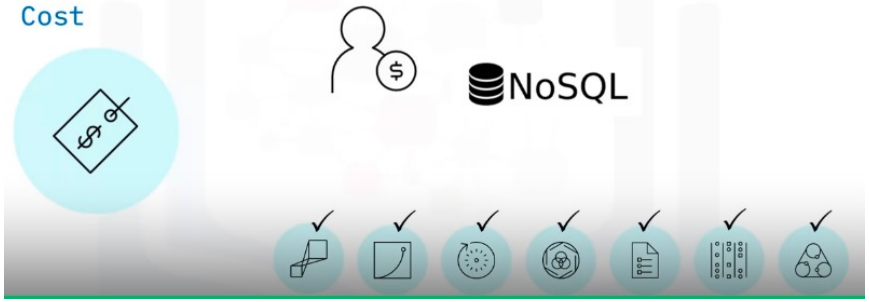



- High Availability is an obvious requirement for a database, and having a database run on a cluster of servers with multiple copies of the data makes for a more resilient solution than a single server solution.

- Historically, large databases have run on expensive machines or mainframes. Modern enterprises are employing cloud architectures to support their applications, and the distributed data nature of NoSQL databases eans that they can be deployed and operated on clusters of servers in cloud architectures, thereby massively reducing cost.
- Cost is important for any technology venture, and it is common to hear of NoSQL adopters cutting significant costs vs. their existing databases... and still be able to get the same or better performance and functionality.

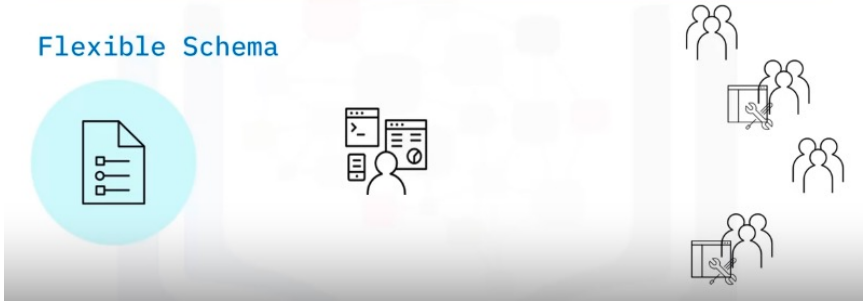


## Benefits of NoSQL Databases



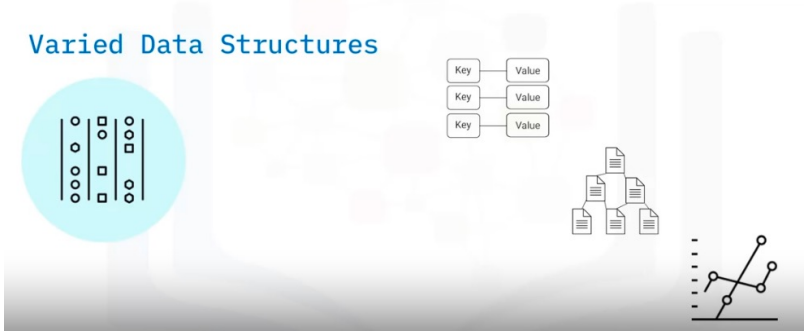

- Cost is important for any technology venture, and it is common to hear of NoSQL adopters cutting significant costs vs. their existing databases... and still be able to get the same or better performance and functionality.

## Benefits of NoSQL Databases



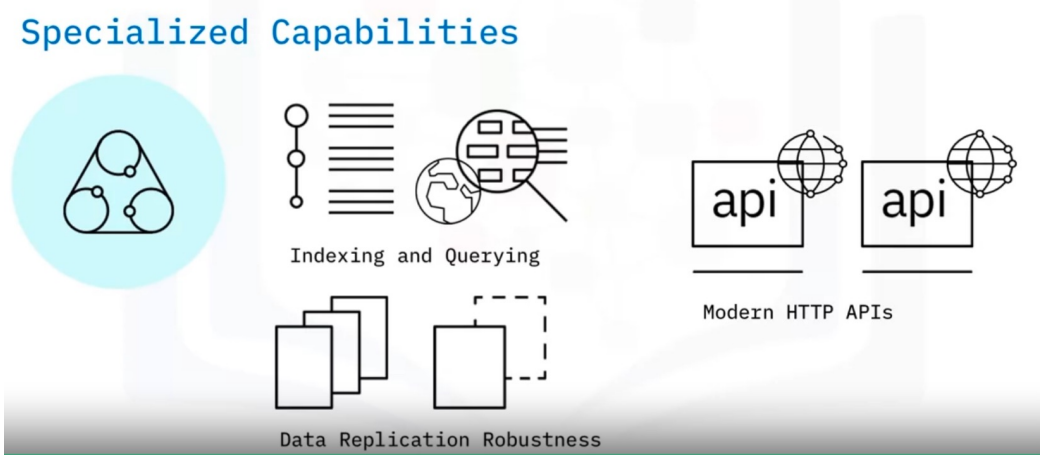

- Flexible schema and intuitive data structures are key features that developers love when wanting to build applications efficiently.
- Most NoSQL databases allow for having flexible schemas, which means that one can build new features into applications quickly and without any database locking or downtime.

## Benefits of NoSQL Databases




- NoSQL databases also have varied data structures which often are more eloquent for solving development needs than the rows and columns of relational datastores.
- Examples include key- value stores for quick lookup, document stores for storing de-normalized intuitive information, and graph databases for associative data sets.

## Benefits of NoSQL Databases




- There are also various specialized capabilities that certain NoSQL providers offer that attract end users.
- Examples include specific indexing and querying capabilities such as geospatial search, data replication robustness, and modern HTTP API's.
- With all these benefits you might well ask why anyone would ever use anything but a NoSQL database.
- Well, you could say this is true for most cases these days, but there are definitely still many requirements which are best met with an RDBMS.

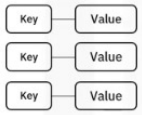
## Summary

In this video, you learned that:

- NoSQL databases are non-relational
- There are four categories of NoSQL database
- NoSQL databases have their roots in the open-source community
- NoSQL database implementations are technically different
- There are several benefits to adopting NoSQL databases

# NoSQL Database Categories - Key-Value

## NoSQL Database Categories



Key-Value


Document

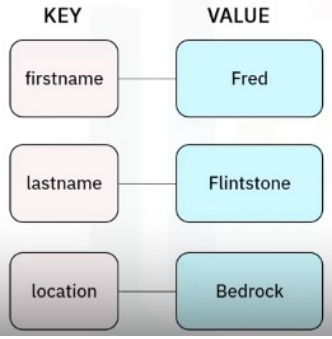

Column


Graph

Each category has:

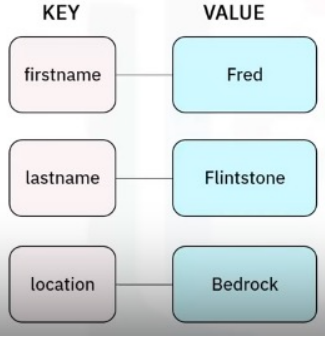
- Unique characteristics
- Architecture
- Use cases

## Key-Value NoSQL Database Architecture



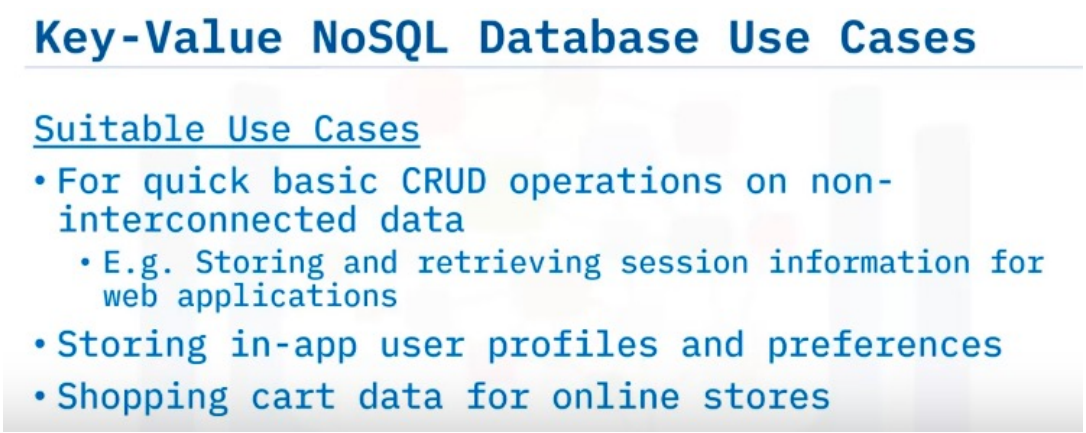

- Least complex
- Represented as hashmap
- Ideal for basic CRUD operations
- Scale well
- Shard easily

## Key-Value NoSQL Database Architecture



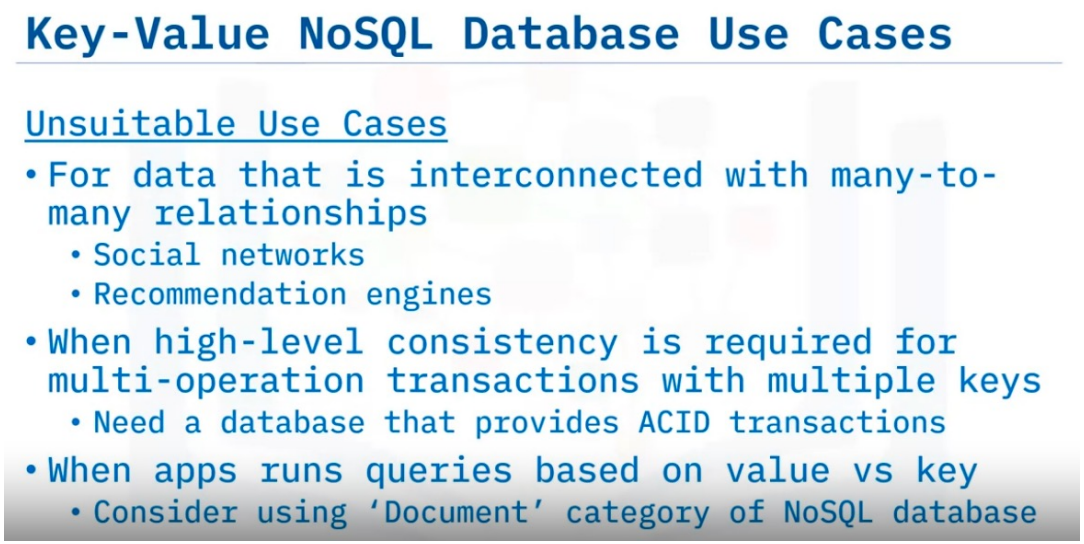

- Not intended for complex queries
- Atomic for single key operations only
- Value blobs are opaque to database
  - Less flexible data indexing and querying

## Key-Value NoSQL: Suitable Usecases




- Well, anytime you need quick performance for basic CRUD operations and your data is not interconnected.
- For example, storing and retrieving session information for a Web application.
- Each user session would receive some sort **of unique key** and all **data** would be stored together in **the opaque value blob**.
- Plus, there would be no need to query based on the information in the value blob.
- All transactions would be based on the unique key.
- Similar use cases would be for **storing user profiles and preferences** within an application or **even storing shopping cart** data for online stores or marketplaces.
- In these cases, complex queries or handling relationships between different keys would be few and far between.

## Key-Value NoSQL: Unsuitable Usecases




- Key-Value type NoSQL databases would not be suitable for use cases that require just the opposite.
- When your data is interconnected with a number of many-to-many relationships in the data, such as social networking or recommendation engine scenarios, a Key-Value NoSQL database is likely to exhibit poor performance.
- Also, if your use case requires a high level of consistency for multi-operation transactions, involving multiple keys, you may want to look more towards databases that provide Atomicity, Consistency, Isolation, and Durability, (or ACID), transactions.
- Lastly, if you expect the need to query based on value versus key, it may be wise to consider the 'Document' category of NoSQL databases, which we will cover shortly.

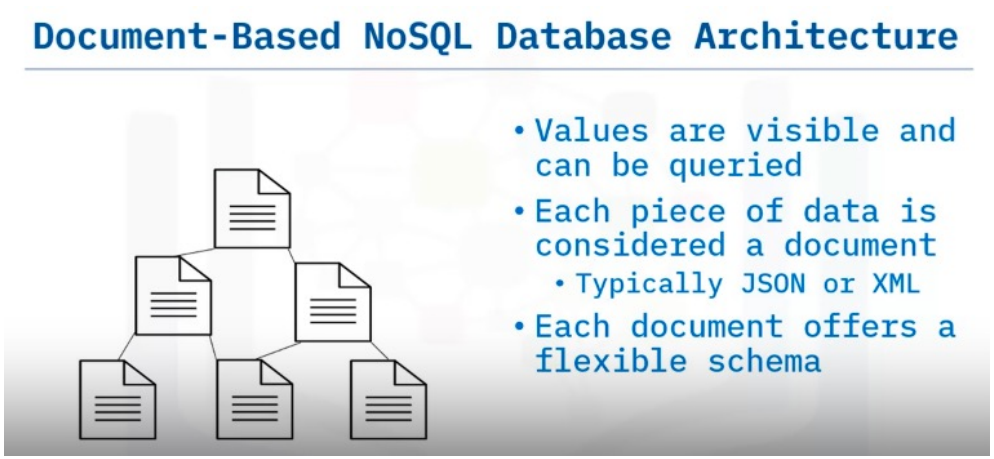
# Key-Value NoSQL Database Examples

# Summary

In this video, you learned that:

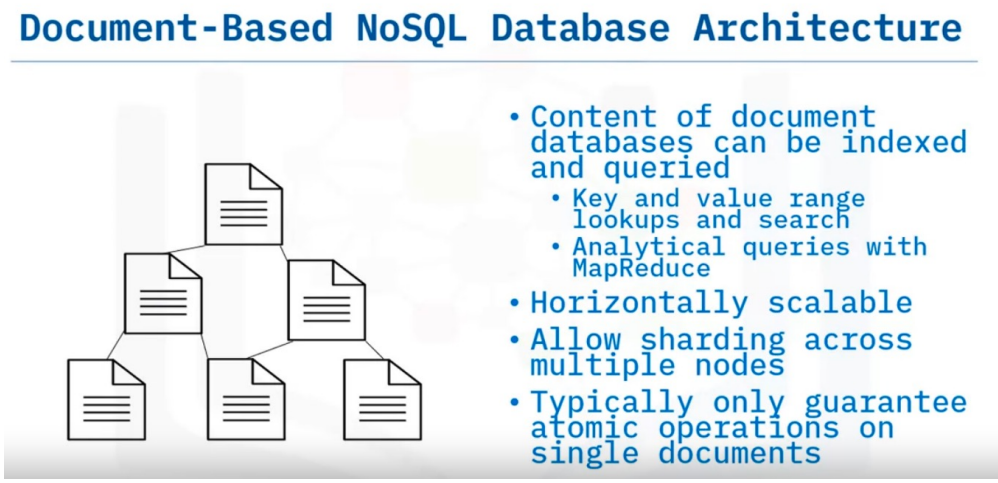
- The four main categories of NoSQL database are Key-Value, Document, Column, and Graph
- The Key-Value database architecture is the least complex; data is stored with a key and corresponding value blob and is represented by a hashmap
- The primary use cases for Key-Value databases are for quick CRUD operations; for example, storing and retrieving session information, storing in-app user profiles, and storing shopping cart data

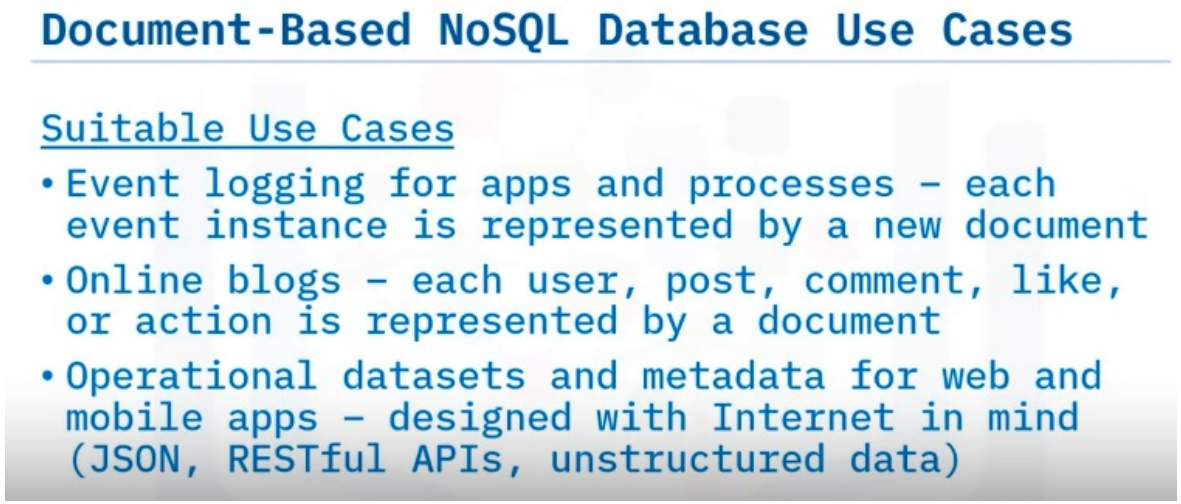
## NoSQL Database Categories - Document




- Document databases build off the Key-Value model by making the value visible and able to be queried.
- Each piece of data is considered a document and typically stored in either **JSON or XML** format.
- One of the benefits of document databases is that each document truly offers a **flexible schema**, where no two documents need to be the same or contain the same information

- Document databases typically offer the **ability to index and query** the contents of the documents, offering key and value range lookups and search ability, or perhaps analytical queries via paradigms like MapReduce.
- Document databases **are horizontally scalable** and allow for **sharding across multiple nodes**, typically sharded by some unique key in the document.
- Document stores also **typically only guarantee atomic transactions** on single document operations.






- The first example would be for event logging for an application or process.
- Each instance would constitute a new document or aggregate, containing all of the information corresponding to the event.
- Another use case would be online blogging.
- Each user would be represented as a document; each post a document; and each comment, like, or action would be a document.
- All documents would contain information pertaining to the type of data, such as username, post content, or timestamp when the document was created.
- More generally speaking, document stores work well with operational datasets for Web and mobile applications
- They were designed with the internet in mind – thinking JSON, RESTful API, and unstructured data.

## Document-Based NoSQL Database Use Cases

Unsuitable Use Cases

- When you require ACID transactions
  - Document databases can't handle transactions that operate over multiple documents
  - Relational database would be a better choice
- If your data is in an aggregate-oriented design
  - If data naturally falls into a normalized tabular model

- Document type NoSQL databases would not be suitable for use cases that require ACID transactions.
- It's not possible for a document store to handle a transaction that operates over multiple documents and a relational database may be a better choice in this instance.
- Secondly, document databases may not be the right choice if you find yourself forcing your data into an aggregate-oriented design.
- If it naturally falls into a normalized/tabular model, this would be another time to research relational databases instead.

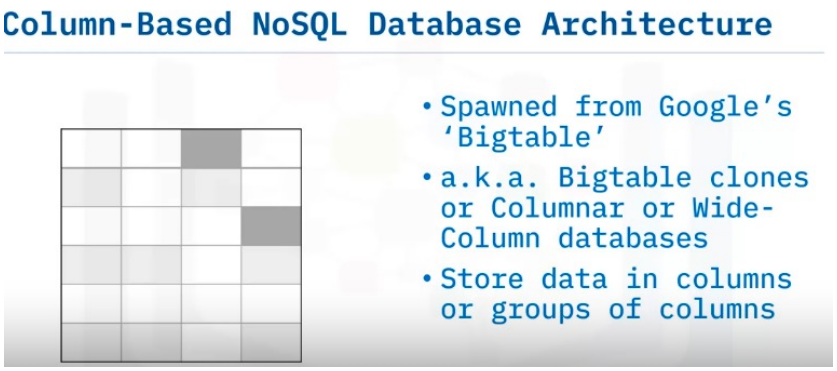
## Document-Based NoSQL Database Examples


## Summary

In this video, you learned that:

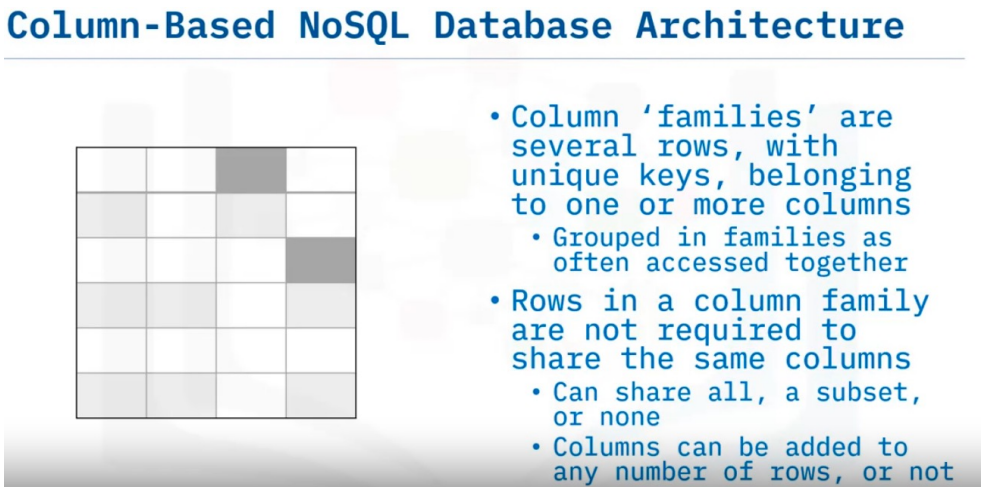
- Document-based NoSQL databases use documents to make values visible and able to be queried
- Each piece of data is considered a document (JSON/XML)
- Each document offers a flexible schema
- The primary use cases for document-based NoSQL databases are event logging for apps/processes, online blogging, operational datasets or metadata for web and mobile apps

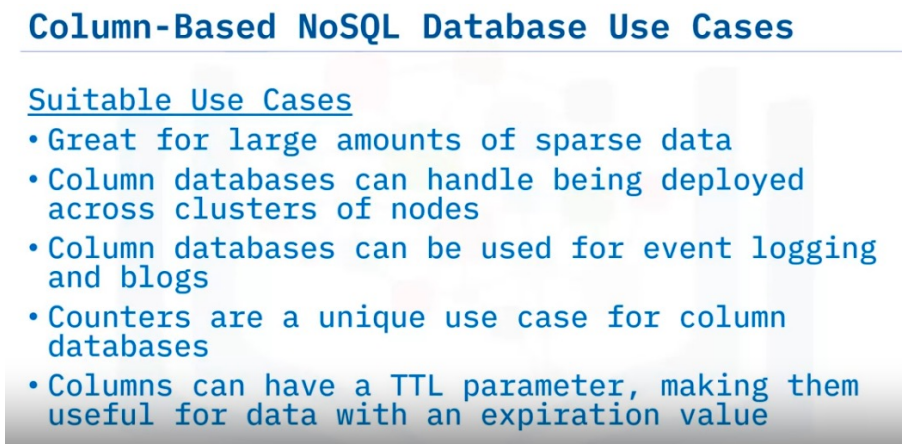
## NoSQL Database Categories - Column




- Column-based databases spawned from an architecture that Google created called Bigtable.
- These databases are therefore also commonly referred to as Bigtable clones, or Columnar databases, or Wide-Column databases.
- As you can tell from the name, these databases focus on columns and groups of columns when storing and accessing data.

- Column 'Families' are several rows, each with a unique key or identifier, that belong to one or more columns.
- These columns are **grouped together** in families because they are often **accessed together.**
- It's also important to point out that rows in a column family are **not required to share any of the same columns.**
- They can share all, a subset, or none of the columns and columns can be added to any number of rows and not to others






- Column-based databases are great for when you're dealing with **large amounts of sparse data.**
- When compared to row-oriented databases, Column-based databases **can better compress data and save storage space**.
- these databases continue the trend of **horizontal scalability**. [ Horizontal Scalability : same size nodes grouped together having same configurations like CPU,RAM,Storage]
- As with Key-Value and Document databases, Column-based databases can **handle being deployed across clusters of nodes.**
- Similar to document databases, a Column-based No SQL database could be used for event logging and blogs, but the data would be stored in a different fashion.
- For enterprise logging, every application can write to its own set of columns and have each row key formatted in such a way to promote easy lookup based on application and timestamp.
- **Counters** are a unique use case for Column-based databases. You may come across applications that need an easy way to count or increment as events occur.
- Some Column-based databases, like Cassandra, have special column types that allow for simple counters.
- In addition, columns can have **a time-to-live parameter**, making them useful for **data with an expiration date or time**, like trial periods or ad timing.

## Column-Based NoSQL Database Use Cases

Unsuitable Use Cases

- For traditional ACID transactions
  - Reads and writes are only atomic at the row level
- In early development, query patterns may change and require numerous changes to column-based databases
  - Can be costly and can slow down the production timeline

- Column-based databases should be avoided if you require traditional ACID transactions provided by relational databases.
- Reads and writes are only atomic at the row level.
- And, early into development, query patterns may change and require numerous changes to the column-based designs.
- This can be costly and slow down the production timeline.

## Column-Based NoSQL Database Examples

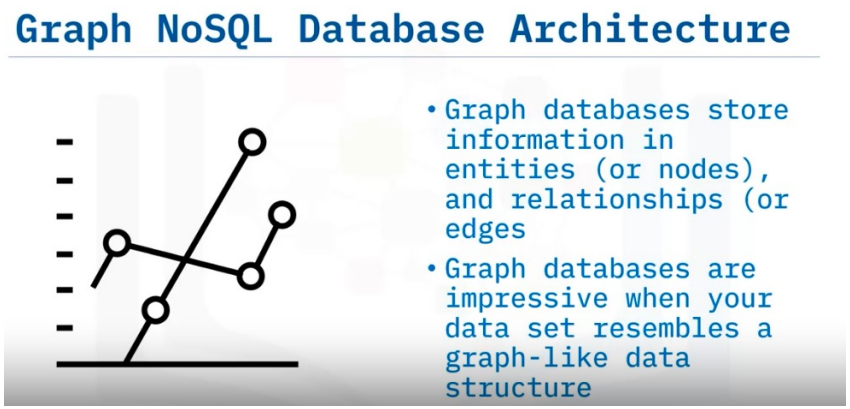

examples of the more popular implementations of Column-based NoSQL databases are: Cassandra, HBASE, Hypertable, and accumulo.

## Summary

In this video, you learned that:

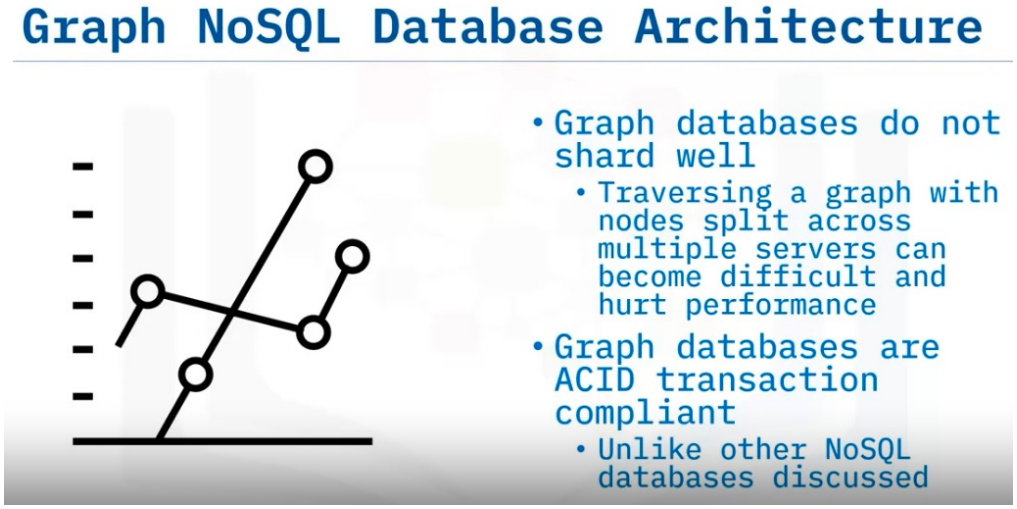
- Column-based databases were spawned from the architecture of Google's Bigtable storage system
- Column-based databases store data in columns or groups of columns
- Column 'families' are several rows, with unique keys, belonging to one or more columns
- The primary use cases for Column-based databases are event logging, blogs, counters, and data with expiration values

## NoSQL Database Categories - Graph



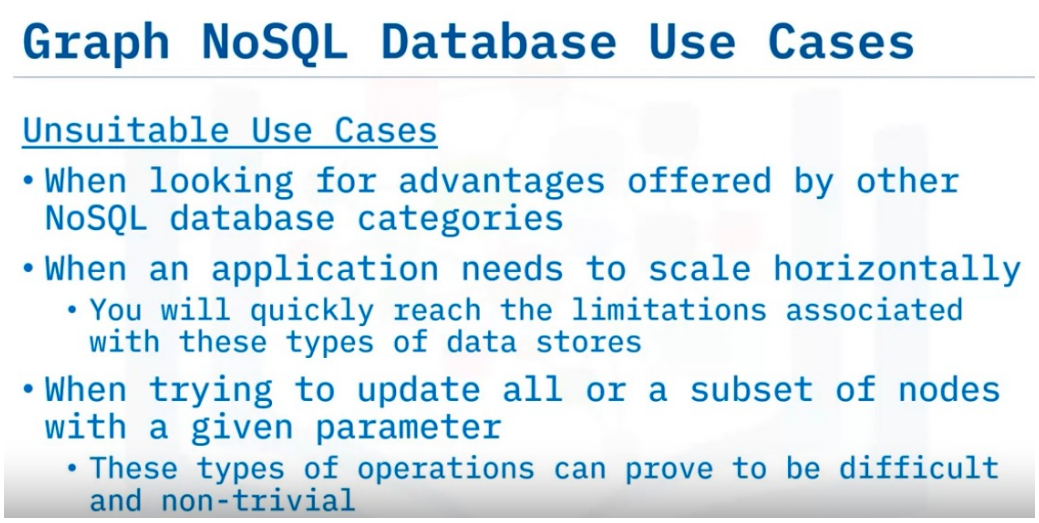

- This database category stands apart from the previous three types covered because it doesn't follow a few of the common traits previously seen.
- From a high level, graph databases store information in entities (or nodes), and relationships (or edges).
- Graph databases are impressive when your data set resembles a graph-like data structure.
- Traversing all of the relationships is quick and efficient, but these databases tend not to scale as well horizontally

- **Sharding a graph database is not recommended** since traversing a graph with nodes split across multiple servers can become difficult and hurt performance.
- Graph databases are also **ACID transaction compliant,** very much unlike the other NoSQL databases previously discussed.
- This **prevents any dangling relationships** between nodes that don't exist.




- Graph databases can be very **powerful** when **your data** is **highly connected** and related in some way.
- Social networking sites can benefit by quickly locating friends, friends of friends, likes, and so on.
- And routing, spatial, and map applications may use graphs to easily model their data for finding close locations or building shortest routes for directions.
- Lastly, **recommendation engines** can leverage the close relationships and links between products to easily provide other options to their customers.

- As far as unsuitable use cases for a Graph database is concerned.
- Graph databases are not a good fit when you're looking for some of the advantages offered by the other NoSQL database categories.
- When an application needs to scale horizontally, you're going to quickly reach the limitations associated with these types of data stores.
- Another general negative surfaces when trying to update all or a subset of nodes with a given parameter.
- These types of operations can prove to be difficult and non-trivial



## Graph NoSQL Database Examples


- some examples of the more popular implementations of Graph NoSQL databases are:
- Neo4j, OrientDB, ArangoDB, Amazon Neptune (part of Amazon Web Services), Apache Giraph, and JanusGraph

## Summary

In this video, you learned that:

- Graph databases store information in entities and relationships
- Graph databases are impressive when your data set resembles a graph-like data structure
- Graph databases don't shard well but are ACID transaction compliant
- The primary use cases for Graph databases are for highly connected and related data, for social networking sites, for routing, spatial and map applications, and for recommendation engines

**Summary and Highlights**

Congratulations! You have completed this lesson. At this point in the course, you know:

•NoSQL means Not only SQL.

•NoSQL databases have their roots in the open source community.

•NoSQL database implementations are technically different from each other.

•There are several benefits of adopting NoSQL databases including storing and retrieving session information, and event logging for apps.

•The four main categories of NoSQL database are Key-Value, Document, Wide Column, and Graph.

•Key-Value NoSQL databases are the least complex architecturally.

•Document-based NoSQL databases use documents to make values visible for queries.

•In document-based NoSQL databases, each piece of data is considered a document, which is typically stored in either JSON or XML format.

•Column-based databases spawned from the architecture of Google's Bigtable storage system.

•The primary use cases for column-based NoSQL databases are event logging and blogs, counters, and data with expiration values.

•Graph databases store information in entities (or nodes) and relationships (or edges).

# Part 2 : Working With Distributed Data

## ACID vs. BASE

- Define what ACID and BASE operations stand for
- Identify the use cases for ACID- and BASE-modelled systems
- Describe the difference between ACID and BASE One of the biggest and most striking differences

- differences between RDBMS and NoSQL databases is their data consistency models.
- The two most common consistency models that are known and in use nowadays are: ACID and BASE.
- Relational databases use the ACID model , and NoSQL databases use the BASE model.
- Although ACID and BASE models are often seen as enemies or one versus the other, the truth is that both come with their advantages and disadvantages. So, let's take a closer look at what both terms mean.

## ACID definition

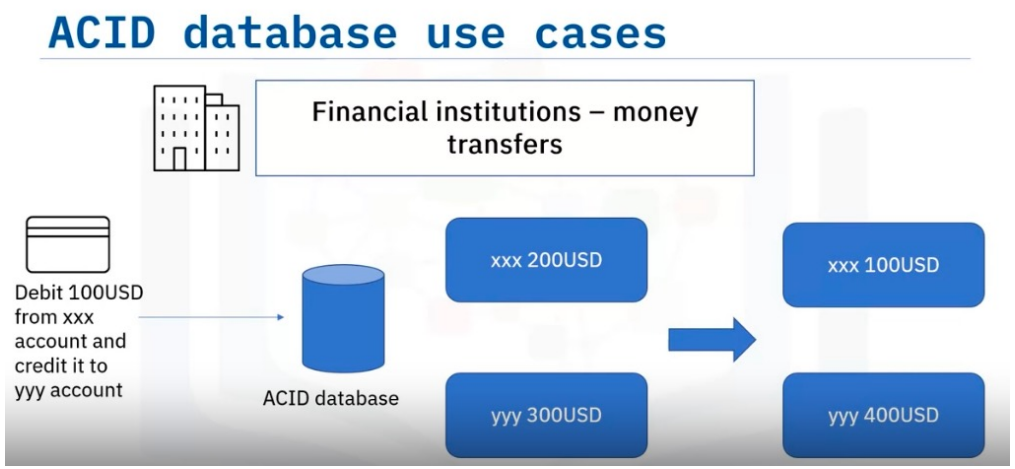

The ACID acronym stands for:

- Atomic – All operations in a transaction succeed or every operation is rolled back.
- Consistent – On the completion of a transaction, the structural integrity of the data in the database is not compromised.
- Isolated – Transactions cannot compromise the integrity of other transactions by interacting with them while they are still in progress.
- Durable – The data related to the completed transaction will persist even in the case of network or power outages. If a transaction fails, it will not impact the already changed data.

## ACID consistency model

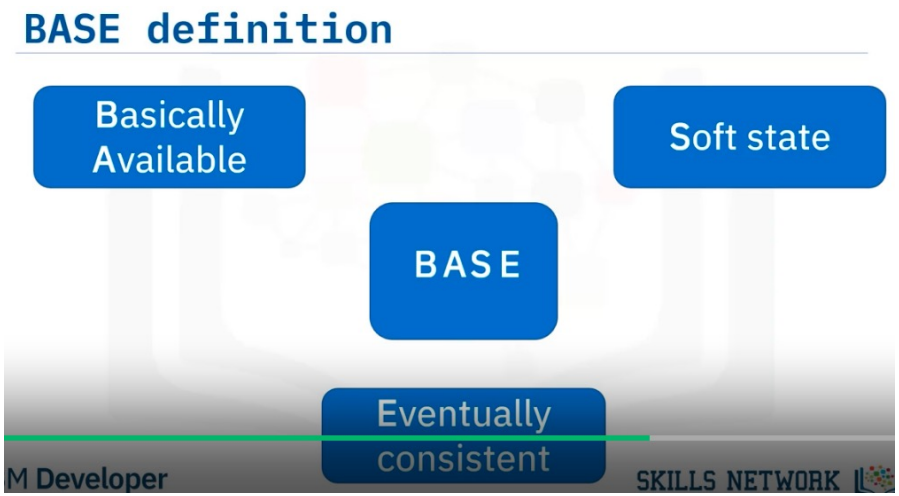
- Used by relational databases
- Ensures a performed transaction is always consistent
- Used by
  - Financial institutions
  - Data warehousing
- Databases that can handle many small simultaneous transactions => relational databases
- Fully consistent system

- Many developers are familiar with ACID transactions from working with relational databases.
- As such, the ACID consistency model has been the norm for some time.
- The ACID consistency model ensures that a performed transaction is always consistent.
- This makes it a good fit for businesses that deal with online transaction processing, such as financial institutions, or data warehousing types of applications.
- These organizations need database systems that can handle many small simultaneous transactions, like relational database can.
- An ACID system provides a consistent model you can count on for the structural integrity of your data




- While there are many possible use cases for ACID databases, one stands out: **financial institutions** will almost exclusively **use ACID databases for their money transfers,** since these operations depend on the atomic nature of ACID transactions.
- An **interrupted transaction** that is not immediately removed from the database can cause a lot of issues.
- Money could be debited from one account and, **due to an error**, **never credited to the other**, or not credited at all.

- The BASE acronym stands for:
- **Basically Available** – Rather than enforcing immediate consistency, BASE-modelled NoSQL databases will ensure availability of data by spreading and replicating it across the nodes of the database cluster.
- **Soft state** – Due to the lack of immediate consistency, data values may change over time. In the BASE model data, stores don't have to be write-consistent, **nor do different replicas have to be mutually consistent all the time.**
- **Eventually consistent** – The fact that the BASE model does not enforce immediate consistency does not mean that it never achieves it.
- However, until it does, data reads might be inconsistent.



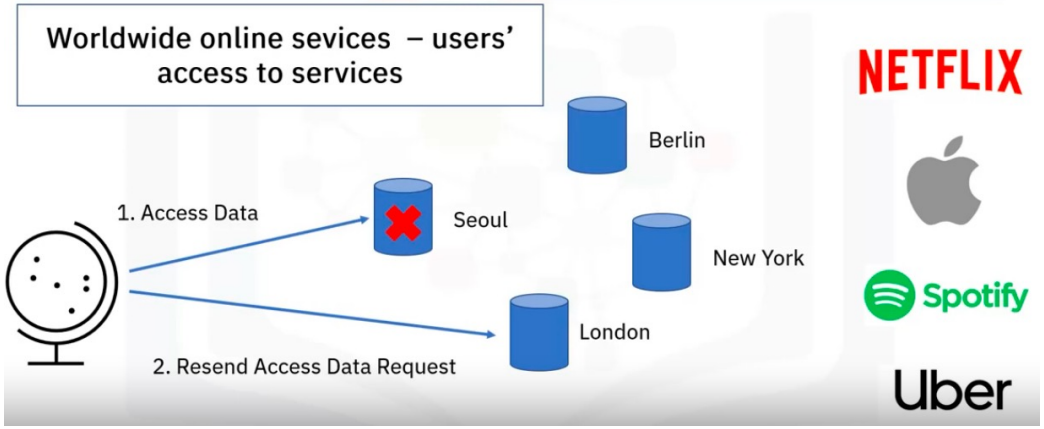
## BASE consistency model

- NoSQL - few requirements for immediate consistency, data freshness, and accuracy
- NoSQL benefits: availability, scale, and resilience
- Used by:
  - Marketing and customer service companies
  - Social media apps
  - Worldwide available online services
- Favors availability over consistency of data
- NoSQL databases use BASE consistency model

- In the NoSQL database world, ACID transactions are less fashionable because some databases have loosened the requirements for immediate consistency, data freshness, and accuracy.
- They do this to gain other benefits, such as availability, scale, and resilience.
- The BASE consistency model is used by: Marketing and customer service companies that deal with sentiment analysis and social network research.
- Social media applications that contain huge amounts of data and need to be available at all times.
- And, worldwide available online services like Netflix, Spotify, and Uber.
- A BASE data store **values availability over consistency**, but it doesn't offer guaranteed consistency of replicated data at write time.
- **NoSQL** databases use the **BASE** consistency model.
- Essentially, the BASE consistency model provides **high availability.**

- BASE data stores can be, and are, used in a lot of specific use cases.
- The fame of BASE databases increased because well-known, worldwide online services such as Netflix, Apple, Spotify, and Uber use them in applications like **user profile data storage.**
- A common characteristic of these services is that they need to be accessible at all times, no matter which part of the globe you are in.
- So, if a part of their database cluster becomes unavailable, the system needs to be able to serve the user requests seamlessly

## BASE database use cases



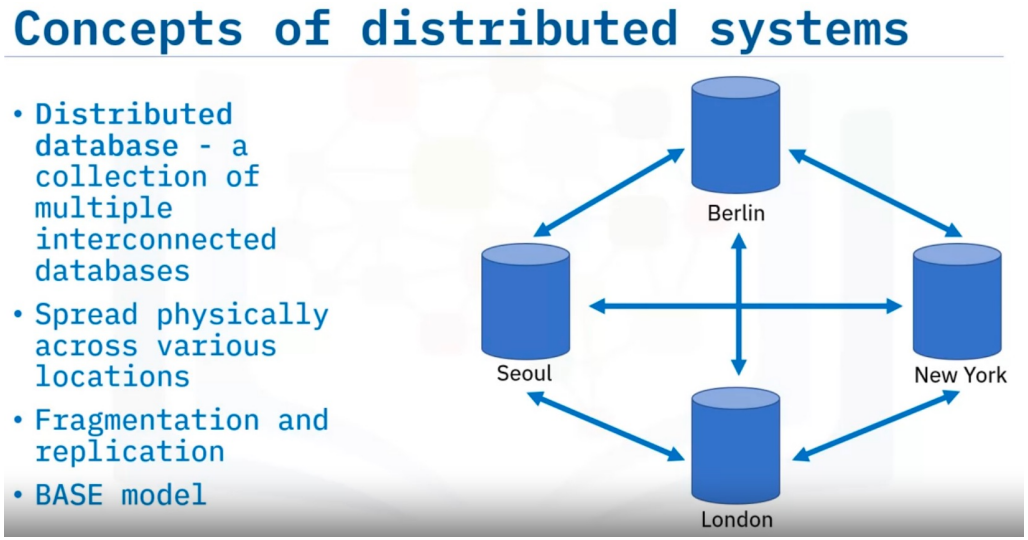
# Summary

In this video you have learned that:

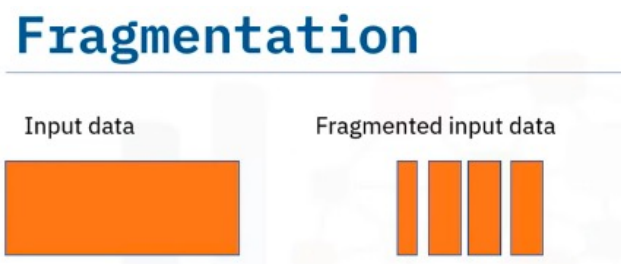
- ACID and BASE are database consistency models
- ACID = Atomicity, Consistency, Isolated, Durable
- BASE = Basically Available, Soft state, Eventually consistent
- ACID (RDBMS) is focused on consistency
- BASE (NoSQL) is focused on availability
- Usage selected by case-by-case basis selection

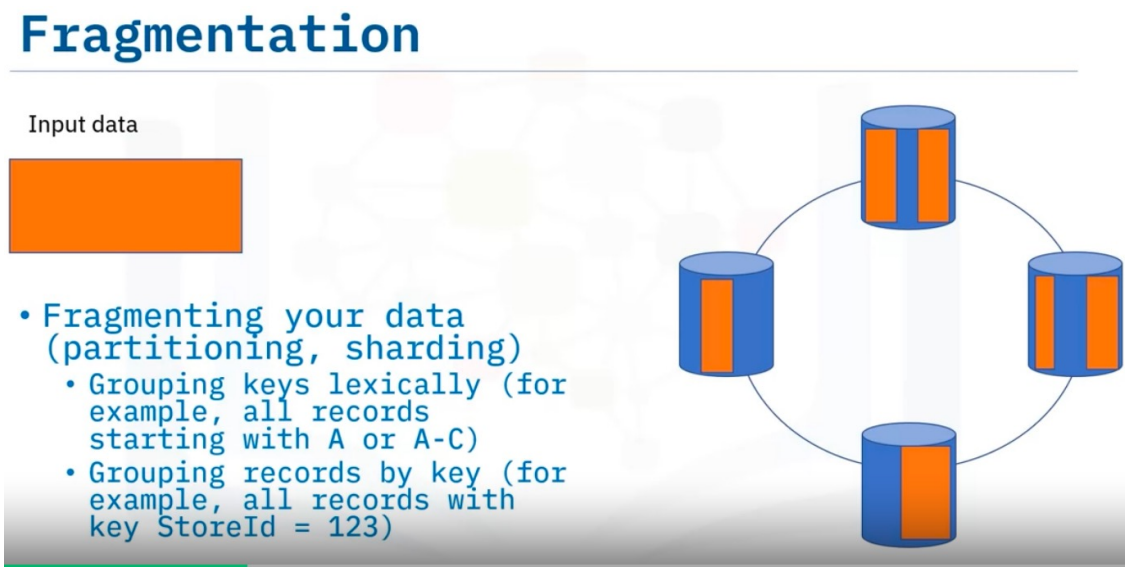
# Distributed Databases

- Describe the concepts of distributed systems,
- Define fragmentation and replication of data, and
- Describe the advantages and challenges of distributed systems.

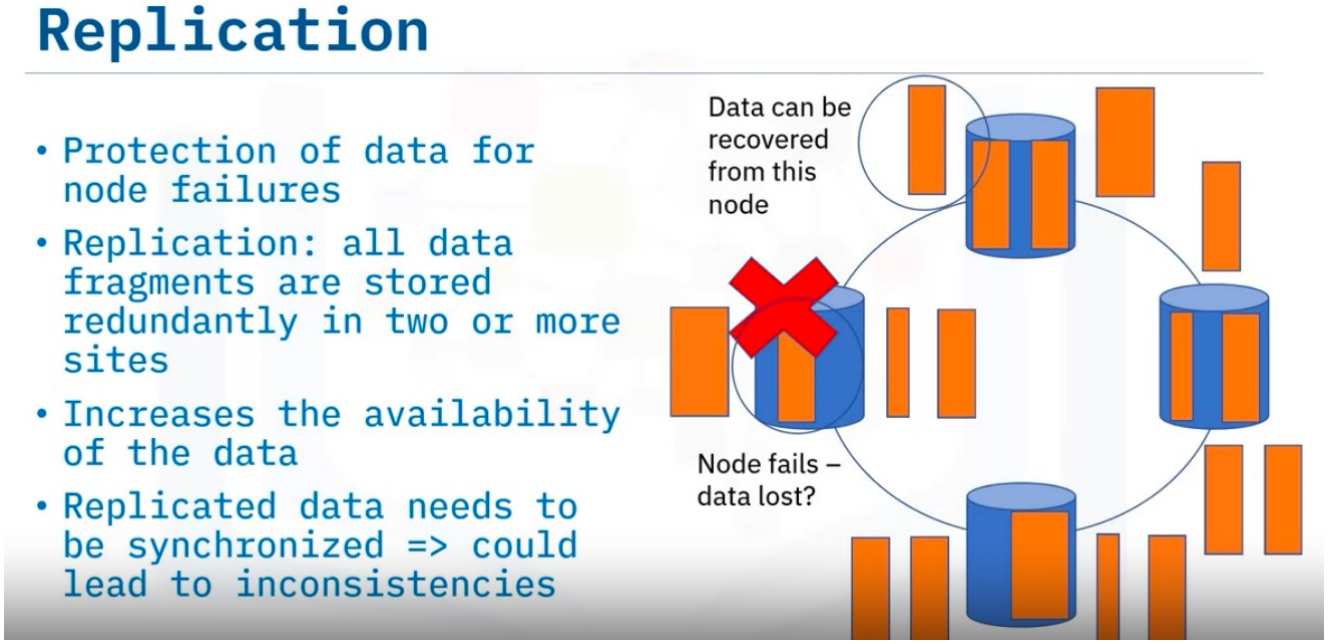



- A distributed database is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.
- A distributed database is physically distributed across the data sites by **fragmenting and replicating the data**.
- A distributed database **follows the BASE consistency model** To store a large piece of data on all the servers of a distributed system, you need to break your data into smaller pieces



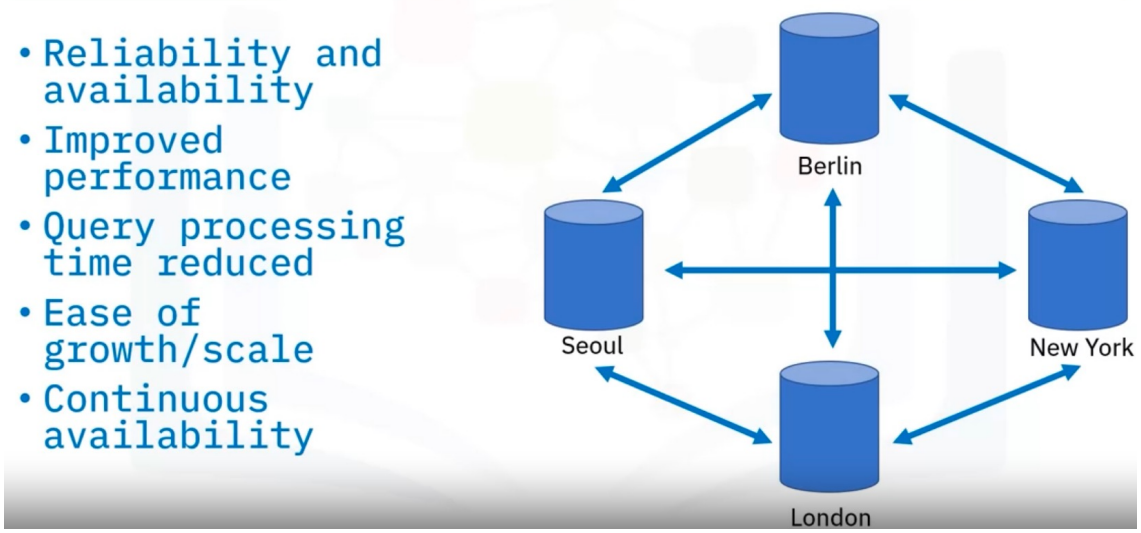




- A distributed database follows the BASE consistency model To store a large piece of data on all the servers of a distributed system, you need to break your data into smaller pieces. This process, known **as fragmentation of data,** is also called **partitioning of data,** or **sharding of data**, by some NoSQL databases.
- No matter the name, all distributed systems need to expose a way to store a large chunk of data.
- This process is usually done by the key of **the 'key-value'** record in two ways: By either grouping all keys lexically.
- For example, all keys that start with A or between A and C can be found on a specific server, or by grouping all records that have the same key, and placing them on the same server.
- For example, all transactions from a store, where 'StoreID' is the key of the records. In this way, with a query like "give me all sales from a store," all records will be on a single server.

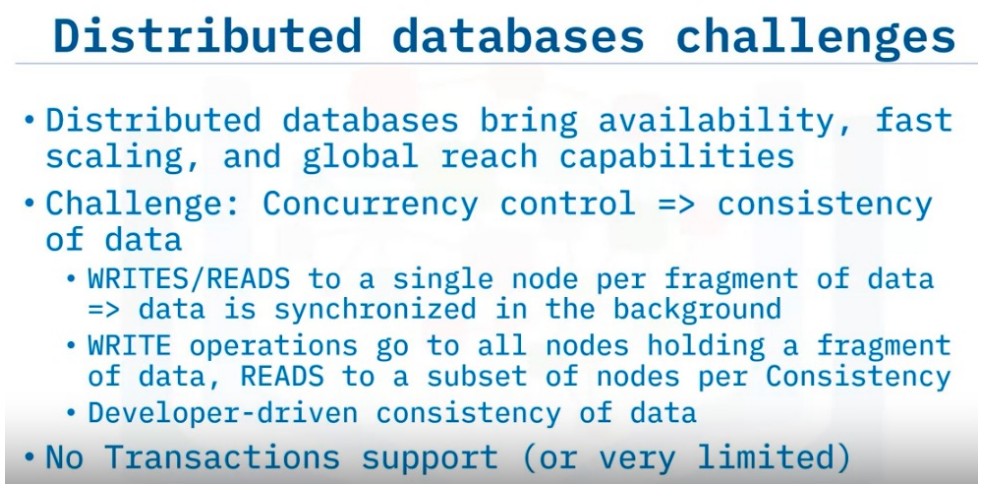



- Data is now **distributed to all the cluster's nodes**, but how do we make sure that if a node fails, we don't lose all the data in that node? This is done through replication, where all fragments (or partitions, or shards) of your **data are stored redundantly in two or more sites**.
- Hence, in replication, systems maintain copies of data.
- Replication **increases the availability** of data in different sites.
- If one node fails, that piece of data can be retrieved from another node.
- However, it has certain disadvantages as well. **Data needs to be constantly synchronized**.
- Any change made at one site needs to be replicated to every site where that related data is stored, or else it will lead to inconsistency.

## Advantages of distributed systems




- Distributed systems have numerous advantages:
- They allow **more reliability and availability**. The data is replicated at multiple sites.
- If the local server is unavailable, the data can be **retrieved from another** available **server.**
- **improved performance**, especially for high volumes of data.
- **Query processing time is reduced**, which also helps improve performance.
- You can easily grow (**or scale) to increase your system capacity**, just by adding new servers to the cluster.
- Distributed systems also provide **continuous operation** with no more reliance on the central site.




- Distributed Databases follow the BASE model: always available systems, data might be inconsistent at times, but eventually consistency can be achieved.

- Distributed databases solve a lot of the technological issues for today's application services, like availability, fast scaling, and global reach, but their architecture also introduces some challenges.
- One is **concurrency control.** Because the same piece of data is stored in multiple locations,
- if you modify (update or delete) your data, how can data synchronization be secured?
- To solve this issue, some distributed databases **direct operations for a certain fragment of data to only one node**, leaving the cluster to synchronize with the other nodes.
- Others **WRITE to all nodes** holding that particular fragment of data and **read from as many** nodes as required **per CONSISTENCY**.
- In both cases the developer can control the consistency of the operation, or how many nodes need to answer for a certain operation to be considered successful.
- Due to concurrency control issues, distributed databases, by design, **don't really support Transactions** (or provide a very limited version of them)

# Summary

In this video you learned that:

- Distributed databases are physically distributed across sites by fragmenting and replicating data
- Fragmentation enables storage of large pieces of data by breaking data into smaller pieces
- Replication means all data partitions stored redundantly in two or more sites
- If one node fails, data can be retrieved from another node
- Distributed databases provide several advantages but also have their disadvantages
- Distributed databases follow the BASE consistency model

# The CAP Theorem

## Objectives

After watching this video, you will be able to:

- Define the CAP theorem
- Describe the characteristics of CAP theorem
- Describe the history and the relevance of CAP theorem

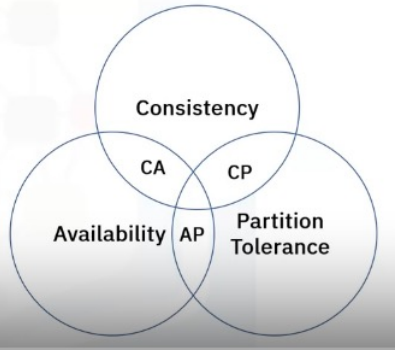
## Consistency vs. Availability

- Early 2000s: emergence of Hadoop, the first open big data architecture that allowed distributed storage and processing of large amounts of data
- Services emerging in 2000s required distributed databases
  - Active and accessible worldwide
  - Always available
- Relational databases: ACID-based, relied on data consistency
- Availability and consistency seemed impossible

- In the early 2000s big data Hadoop architecture was being created as the first open source, distributed architecture that can store and process high volumes of data.
- At this time, more and more services were developed that required databases to be distributed as well.
- Actually, these businesses required not only that their services were active and accessible in most parts of the world, but also that their services were always available.
- For relational databases that rely so much on the consistency of data, the new concept of availability while having a distributed system seemed impossible, and this was proven by the CAP Theorem.

## CAP Theorem definition and history

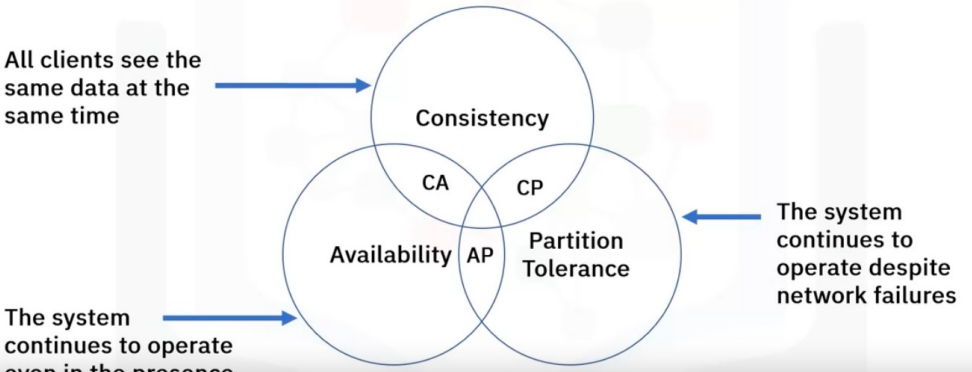
- CAP Theorem (Brewer's theorem)
- Evolved by MIT professors Seth Gilbert and Nancy Lynch
- Consistency, Availability and Partition Tolerance (CAP)

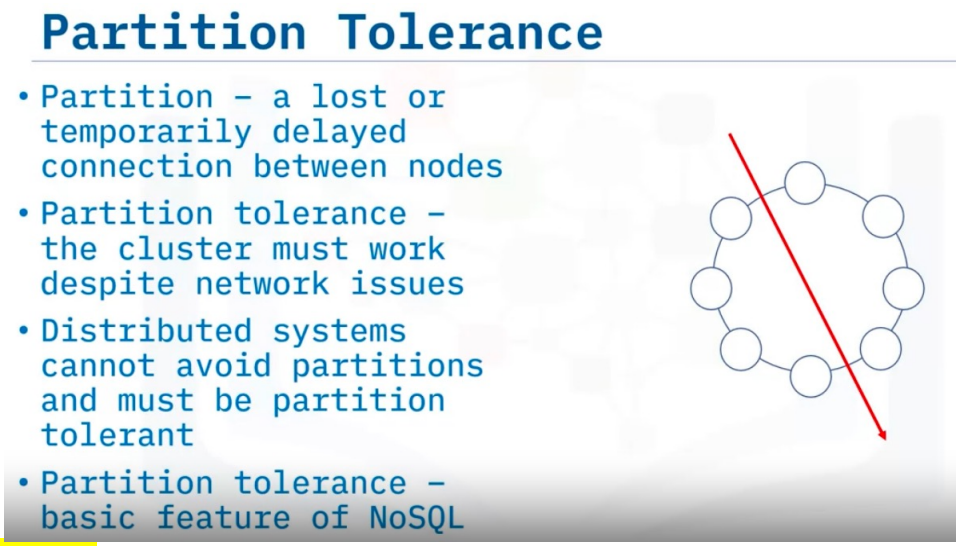



- The CAP Theorem is also called Brewer's Theorem, because it was first advanced by Professor Eric A. Brewer during a talk he gave on distributed computing in the year 2000.
- Two years later it was revised by MIT professors Seth Gilbert and Nancy Lynch, and there have been many other contributors since.
- The theorem states that there are three essential system requirements necessary for the successful design, implementation, and deployment of applications in distributed systems.
- These are Consistency, Availability, and Partition Tolerance, or CAP.
- A **distributed system** can **guarantee** delivery of only two of **these three desired characteristics.**

- Consistency refers to whether a system operates fully or not.
- Do all nodes within a cluster see all the data they are supposed to? Availability means just as it sounds.
- Does each request get a response outside of failure or success?
- Partition Tolerance represents the fact that a given system continues to operate even under circumstances of data loss or network failure.

## CAP Theorem

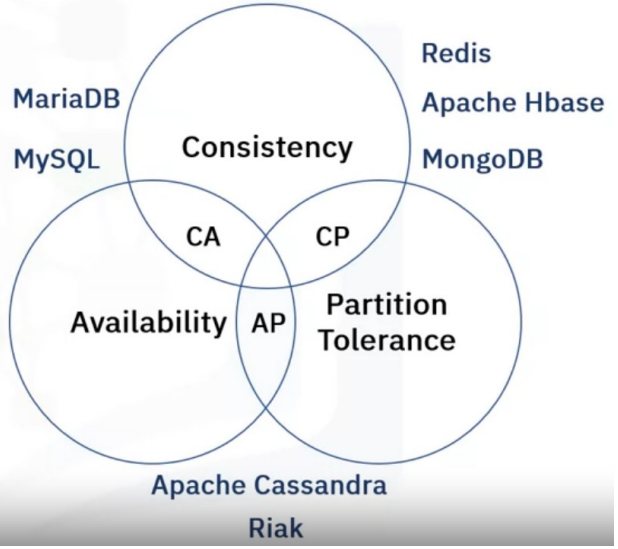





- A partition is a communications break within a distributed system—a lost or temporarily delayed connection between nodes.
- Partition tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.
- In distributed systems, partitions can't be avoided. Therefore, partition tolerance becomes a basic feature of native distributed systems such as NoSQL.
- In a cluster with eight distributed nodes, a network partition could occur, and communication will be broken between all the nodes.
- In our case, instead of one 8-node cluster we will have two smaller 4-node clusters available.
- Consistency between the two clusters will be achieved when network communication is re-established.
- Partition tolerance has become more of a necessity than an option in distributed systems.
- It is made possible by sufficiently replicating records across combinations of nodes and networks.

## NoSQL: CP or AP

- NoSQL – a choice between consistency and availability
- MongoDB – consistency
- Apache Cassandra – availability
- Tunable systems



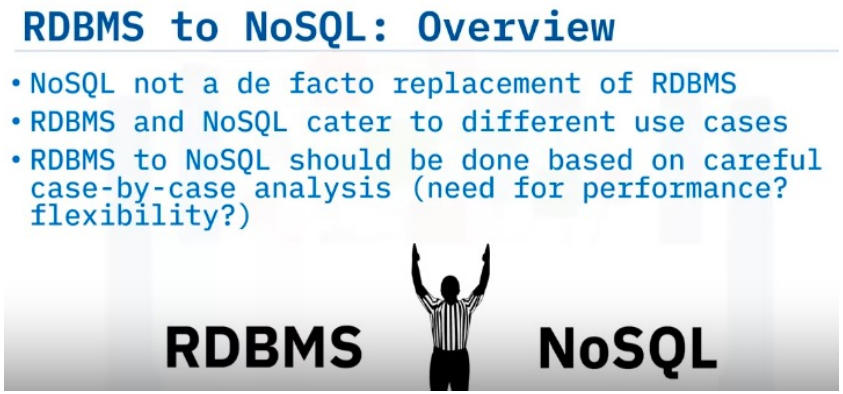

- For such systems as NoSQL, since partition tolerance is mandatory, a system can be either Consistent and Partition Tolerant (CP) or Available and Partition Tolerant (AP).
- Existing NoSQL systems, like MongoDB or Cassandra, can be classified using CAP Theorem.
- For example, MongoDB chooses consistency as the primary design driver of the solution, and Apache Cassandra chooses availability.
- This doesn't mean that MongoDB cannot be available, or that Cassandra cannot become fully consistent.
- It means that these solutions first ensure that they are consistent (in the case of MongoDB) or available (in the case of Cassandra) and the rest is tunable.

# Summary

In this video you have learned that:

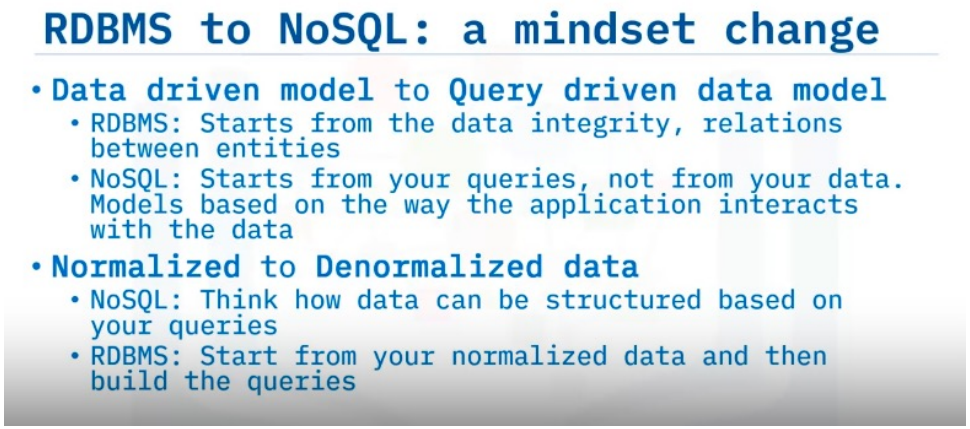
- CAP theorem can be used to classify NoSQL databases
- NoSQL databases choose between availability and consistency
- Partition Tolerance is a basic feature of NoSQL databases

## Challenges in Migrating from RDBMS to NoSQL Databases




- There is a misconception sometimes that either NoSQL is the replacement of RDBMS, or that you need to choose between relational and NoSQL databases.
- In reality, RDBMS and NoSQL database are not competing because they cater to quite different requirements and use cases.
- While it is possible to migrate from an RDMBS system to a NoSQL system, this should be done based on the characteristics required by your end solution.
- The change might be driven by additional performance or flexibility.
- Whatever the reasons are, you need to analyze them against what NoSQL can do for you.

- If you need full consistency for your data, if your data is structured, if you need multi document transactions and complicated joins, in this case a **relational database** will make more sense.
- If you have loads of data and need a high-performance system, if your data is unstructured and could benefit from a flexible schema, if your system needs to be available and scalable, then in this case a **NoSQL database** will make more sense.
- There might be cases in which both relational and NoSQL databases will be needed.
- If you have **too much data** and **need performance**, and **need to scale** fast, But at the same time, you also **need transactions support**, and **complex joins** on your data, then you might think of a **combined solution.**




When embracing the NoSQL world and coming from a relational one, there are a few things you should pay attention to:

- Like the fact that in NoSQL your data model is driven by your queries, not the data itself.
- In a relational world the solution design starts from the data, the entities, and their relationship.
- In NoSQL it's not the data that drives your data model or schema. It's the way your application accesses the data and the queries you are going to make.
- In NoSQL, models should be based on how the app interacts with the data, rather than how the model can be stored as rows in one or more tables.
- Another factor to pay attention to is the fact that in RDBMS data is normalized, while in NoSQL it is denormalized.
- With NoSQL, starting from your query means that you will structure your data on disk accordingly.
- Thus, you may need to store the same data in different models just to answer the question.
- This will lead to data denormalization. While data in RDBMS is normalized, you need to be open to the situation in which you start with your queries instead of data.

## RDBMS to NoSQL: a mindset change

- From ACID to BASE model
  - Availability vs. Consistency
  - CAP Theorem – choose between availability and consistency
  - Availability, performance, geographical presence, high data volumes
- NoSQL systems, by design, do not support transactions and joins (except in limited cases)

- When migrating from relational to NoSQL databases, you need to understand that sometimes services require availability more than consistency.
- When both availability and performance are needed (thus distributed systems), consistency cannot be ensured.
- Remember CAP Theorem? Many of today's online services value availability more than consistency. Because of this, they look for systems that can provide it, taking into consideration the amount of data they are dealing with, and their geographical presence.
- last thing to know when dealing with NoSQL databases: These are not designed to support transactions, or joins, or complex processing, except in limited cases. You need to consider this when moving from RDBMS to NoSQL.

## Summary

In this video you have learned that:

- NoSQL systems are not a de facto replacement of RDBMS
- RDBMS and NoSQL cater to different use cases
- You could use both RDBMS and NoSQL
- A migration from RDBMS to NoSQL could be triggered by performance or flexibility
- Migrating from RDBMS to NoSQL requires adoption of NoSQL concepts

**Summary and Highlights**

Congratulations! You have completed this lesson. At this point in the course, you know:

•ACID stands for Atomicity, Consistency, Isolated, Durable.

•BASE stands for Basic Availability, Soft-state, Eventual Consistency.

•ACID and BASE are the consistency models used in relational and NoSQL databases.

•Distributed databases are physically distributed across data sites by fragmenting and replicating the data.

•Fragmentation enables an organization to store a large piece of data across all the servers of a distributed system by breaking the data into smaller pieces.

•You can use the CAP Theorem to classify NoSQL databases.

•Partition Tolerance is a basic feature of NoSQL databases.

•NoSQL systems are not a de facto replacement of RDBMS.

•RDBMS and NoSQL cater to different use cases, which means that your solution could use both RDBMS and NoSQL.

•Python and MongoDB: Connecting to NoSQL Databases – Real Python