Module 1

Types NoSQL Database:
Key Value
Document based
Column based
Graph based

- **Key/Value stores**
  - Data is stored in the form of key/value
  - Each key is unique and accepts only String, JSON, XML
  - Hence it is capable to store massive load of data
  - It consists of index key and a value
  - Has fast query performance
  - Suitable for content caching where content from single server can be copied and distributed across various centers and clouds e.g: gaming website which constantly updates top 10 scores and players
- **Document stores**
  - Extension of key-value stores
  - Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document.
  - Values are stored in structured documents like JSON/XML
  - It is schema free
  - Do not support relations
  - Documents are stored in collections
  - Unlike key stores it can query attributes within a document e.g A product review website where zero or many users can review each product and each review can be commented on by other users and can be liked or disliked by zero to many users.

Column based stores
- Stores data tables as sections of columns of data rather than as rows of data.
- Works on the concept of keyspace
- Keyspaces contains column family, which contains rows and those rows then contains columns

Graph Based
- A graph database stores nodes and relationships instead of tables, or documents.
- Data is stored just like you might sketch ideas on a whiteboard.
- A graph type database stores entities as well the relations amongst those entities.
- The entity is stored as a node with the relationship as edges.
- An edge gives a relationship between nodes. Every node and edge has a unique identifier.

ACID Vs Base

Criteria	**ACID**	**BASE**
Simplicity	Simple	Complex
Focus	Commits	Best attempt

Maintenance	High	Low
Consistency Of Data	Strong	Weak/Loose
Concurrency scheme	Nested Transactions	Close to answer
Scaling	Vertical	Horizontal
Implementation	Easy to implement	Difficult to implement
Upgrade	Harder to upgrade	Easy to upgrade
Type of database	Robust	Simple

When to Use NoSQL database

NoSQL database technology is usually adopted for one or more of the following reasons:
The pace of development with NoSQL databases can be much faster than with a SQL database.
Because NoSQL databases often allow developers to be in control of the structure of the data, they are a good fit with modern Agile development practices based on sprints, quick iterations, and frequent code pushes. When a developer must ask a SQL database administrator to change the structure of a database and then unload and reload the data, it can slow development down.
The structure of many different forms of data is more easily handled and evolved with a NoSQL database.NoSQL databases are often better suited to storing and modeling structured, semi-structured, and unstructured data in one database.NoSQL databases often store data in a form that is similar to the objects used in applications, reducing the need for translation from the form the data is stored in to the form the data takes in the code.The amount of data in many applications cannot be served affordably by a SQL database.NoSQL databases were created to handle big data as part of their fundamental architecture. Additional engineering is not required as it is when SQL databases are used to handle web-scale applications. The path to data scalability is straightforward and well understood.
NoSQL databases are often based on a scale-out strategy, which makes scaling to large data volumes much cheaper than when using the scale-up approach the SQL databases take.
The scale of traffic and need for zero downtime cannot be handled by SQL.The scale-out strategy used by most NoSQL databases provides a clear path to scaling the amount of traffic a database can handle.
Scale-out architectures also provide benefits such as being able to upgrade a database or change its structure with zero downtime.
The scale-out architecture is one of the most affordable ways to handle large volumes of traffic.
New application paradigms can be more easily supported.
The scalability of NoSQL databases allows one database to serve both transactional and

analytical workloads from the same database. In SQL databases, usually, a separate data warehouse is used to support analytics.
NoSQL databases were created during the cloud era and have adapted quickly to the automation that is part of the cloud. Deploying databases at scale in a way that supports microservices is often easier with NoSQL databases.
NoSQL databases often have superior integration with real-time streaming technologies.
NoSQL databases support polyglot persistence, the practice of mixing various types of NoSQL databases depending on the needs of particular segments of an application. For example, some applications store most of their data in a document database like MongoDB, but supplement that with a graph database to capture inherent connections between people or products.

Module 2

Advantages of Document Store:

1.Probably the biggest benefit of document-store is that everything is available in a single database, rather than having information spread across several linked databases.
2.Unlike in conventional databases where a field exists for each piece of information, even if there's nothing in it, a document-store is more flexible
3.Similarly, since it’s more flexible, integrating new data isn’t problematic at all.
4.Compared to a relational database where any new type of information must be added to all datasets, document-store only requires you to do it in a few.
5.More specifically, because schema can be modified without any downtime, or due to the fact that you may not know user needs in the future, document stores are great for these applications:

Large eCommerce platforms (Like Amazon)
Blogging sites (such as Twitter)
Content management systems (WordPress, windows registry)
Analytical platforms

Capped Collection:

Fixed-size collections are called capped collections in MongoDB. While creating a collection, the user must specify the collection’s maximum size in bytes and the maximum number of documents that it would store. If more documents are added than the specified capacity, the existing ones are overwritten.
Capped collection in MongoDB supports high-throughput operations used for insertion and retrieval of documents based on the order of insertion. Capped collection working is similar to circular buffers, i.e., there is a fixed space allocated for the capped collection. The oldest documents are overwritten to make space for the new documents in the collection once the fixed size gets exhausted.

CRUD Operations:

Implement CRUD operation in MongoDB.
CRUD operations create, read, update, and delete documents.

Create Operations
Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.
MongoDB provides the following methods to insert documents into a collection:

db.collection.insertOne()
db.collection.insertMany()

In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
Read Operations
Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

db.collection.find()
Update Operations
Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:

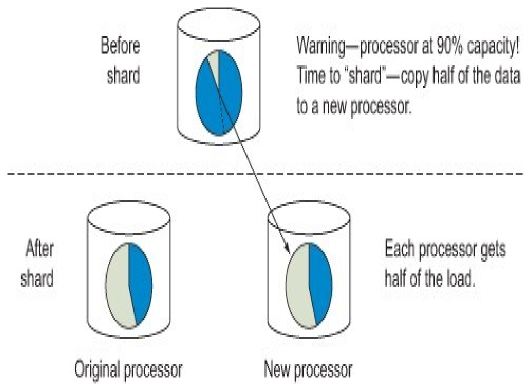
db.collection.updateOne()
db.collection.updateMany()
db.collection.replaceOne()
In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
Delete Operations
Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

db.collection.deleteOne()
db.collection.deleteMany()
In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

Sharding

Sharding : Breaking a database into chunks called *shards* and spreading the chunks across a number of distributed servers

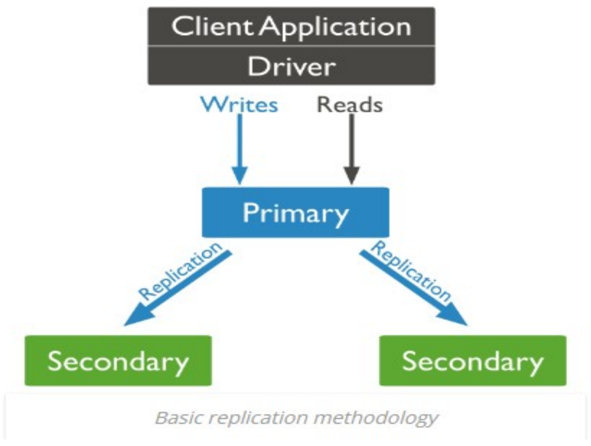
- Working: Sharding is performed when a single processor can’t handle the throughput requirements of a system.
- When this happens you’ll want to move the data onto two systems that each take half the work.
- Many NoSQL systems have automatic sharding built in so that you only need to add a new server to a pool of working nodes and the database management system automatically moves data to the new node.
- But RDBMSs don’t support automatic sharding**.**



Replication

- Creates a copy of the same data set in more than one MongoDB server.
- This can be achieved by using a Replica Set.
- A replica set consists of multiple MongoDB nodes that are grouped together as a unit.
- Redundancy and Failover

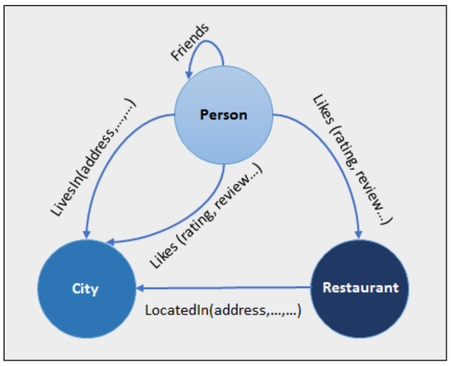
How replication works:



Graph Based Database:

- A graph database stores nodes and relationships instead of tables, or documents.
- Data is stored just like you might sketch ideas on a whiteboard.
- A graph type database stores entities as well the relations amongst those entities.
- The entity is stored as a node with the relationship as edges.
- An edge gives a relationship between nodes. Every node and edge has a unique identifier.
  Example: Neo4J
- Neo4j Graph Database has the following building blocks:
  - Nodes
  - Properties
  - Relationships
  - Labels
  - Data Browser
- It contains nodes and relationships.
- Nodes contains properties (Key Value Pair).
- Nodes can be labelled with one or mode labels.

- Relationships are named and directed and always have start and end node.
- Relationships can also have properties.



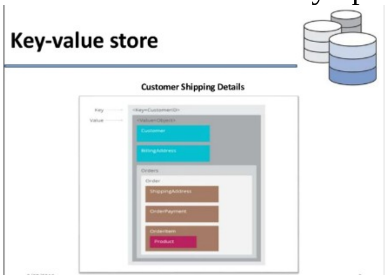
- Features of graph databases
- Flexibility
- Agility
- Improved performance, even with huge volumes of data.
- Typical use cases for graph databases would include social networking site, recommendation engine.
- **Pros:**
- – Extremely powerful
- – Connected data is locally indexed
- – Can provide ACID
- – Results in real-time
- – Agile Structure
- **Cons:**
- – Difficult to scale since it has one tier architecture
- – No Uniform query language

Typical use cases for graph databases would include:

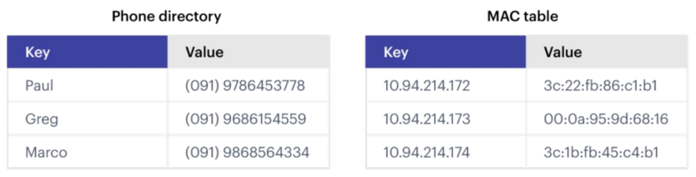
- Social networking site
- Recommendation engine
- Risk assessment
- Fraud detection

Key/Value stores

– Data is stored in the form of key/value
– Each key is unique and accepts only String, JSON, XML
– Hence it is capable to store massive load of data
– It consists of index key and a value
– Has fast query performance
– Suitable for content caching where content from single server can be copied and distributed across various centers and clouds e.g: gaming website which constantly updates top 10 scores and players



A key-value database (sometimes called a key-value store) uses a simple key-value method to store data. These databases contain a simple string (the key) that is always unique and an arbitrary large data field (the value). They are easy to design and implement.



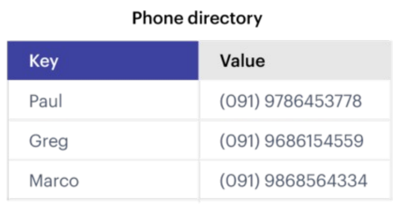
Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

MAC table

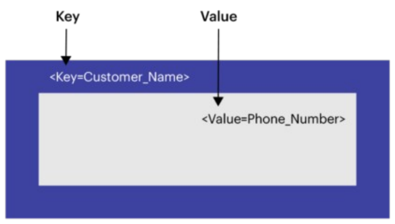
Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1

As the name suggests, this type of NoSQL database implements a hash table to store unique keys along with the pointers to the corresponding data values. The values can be of scalar data types such as integers or complex structures such as JSON, lists, BLOB, and so on. A value can be stored as an integer, a string, JSON, or an array—with a key used to reference that value. It typically offers excellent performance and can be optimized to fit an organization’s needs. Key-value stores have no query language, but they do provide a way to add and remove key-value pairs. Values cannot be queried or searched upon. Only the key can be queried.



Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

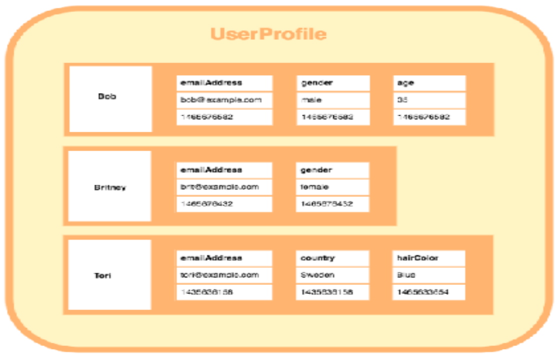


- PROS
  - SIMPLE DATA MODEL
  - SCALABLE
  - VALUE CAN INCLUDE JSON, XML, FLEXIBLE SCHEMA
  - EXTREMELY FAST OWING TO ITS SIMPLICITY
  - BEST FITS WHERE DATA IS NOT HIGHLY RELATED
- CONS
  - Not suitable for complex data
  - Not ideal for operations rather than CRUD

USE CASE:
Storing session information: Offers to save and restore sessions
User preferences: Specific data for user
Shopping carts: Easily handle the loss of storage nodes and quickly scale data during a holiday on an e-commerce application
Product recommendations: Offering recommendations based on the persons data
Example: REDIS AEROSPIE AMAZON DYNAMODB

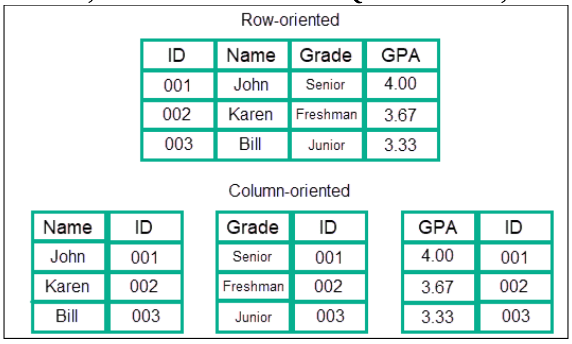
Column based stores

- Stores data tables as sections of columns of data rather than as rows of data.
- Works on the concept of keyspace
- Keyspaces contains column family, which contains rows, and those rows then contains columns



How Does a Column Database Work?

At a very surface level, column-store databases do exactly what is advertised on the tin: namely, that instead of organizing information into rows, it does so in columns. This essentially makes them function the same way that tables work in relational databases. Of course, since this is a NoSQL database, this data model makes them much more flexible.



Row-oriented

ID	Name	Grade	GPA
001	John	Senior	4.00
002	Karen	Freshman	3.67
003	Bill	Junior	3.33

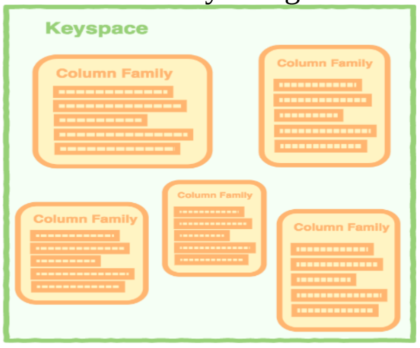
Column-oriented

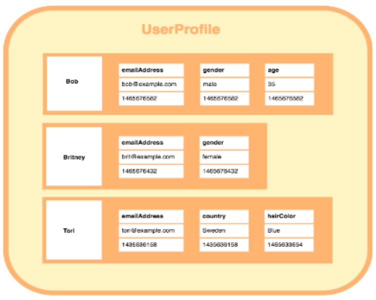
Name	ID
John	001
Karen	002
Bill	003

Grade	ID
Senior	001
Freshman	002
Junior	003

GPA	ID
4.00	001
3.67	002
3.33	003

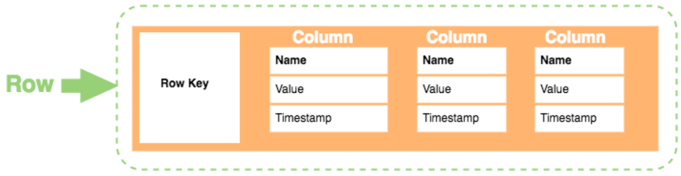
More specifically, column databases use the concept of keyspace, which is sort of like a schema in relational models. This keyspace contains all the column families, which then contain rows, which then contain columns. It's a bit tricky to wrap your head around at first but it's relatively straightforward.





we can see that a column family has several rows. Within each row, there can be several different columns, with different names, links, and even sizes (meaning they don't need to adhere to a standard). Furthermore, these columns only exist within their own row and can contain a value pair, name, and a timestamp.

If we take a specific row as an example:



The Row Key is exactly that: the specific identifier of that row and is always unique. The column contains the name, value, and timestamp, so that's straightforward. The name/value pair is also straight forward, and the timestamp is the date and time the data was entered into the database.
Some examples of column-store databases include Casandra, CosmoDB, Bigtable, and HBase.

MongoDB for CMS

MongoDB helps in CMS.

- What is CMS: Software that helps users create, manage, and modify content on a website without the need for specialized technical knowledge like building a website without needing to write all the code.
- Let us store any content , retrieve it and change it.
- MongoDB is particularly well-suited to support your CMS efforts because the software offers:
  – A flexible data model
  – Scalable to millions of users
  – Much lower cost

CMS application overview

- Business new services
- Hundreds of stories per day
- Millions of websites visitors per month
- Comments
- Tags
- Company profile

Horizontal Scaling Vs Vertical Scaling

Horizontal Scaling	Vertical Scaling
When new server racks are added in the existing system to meet the higher expectation, it is known as horizontal scaling.	When new resources are added in the existing system to meet the expectation, it is known as vertical scaling

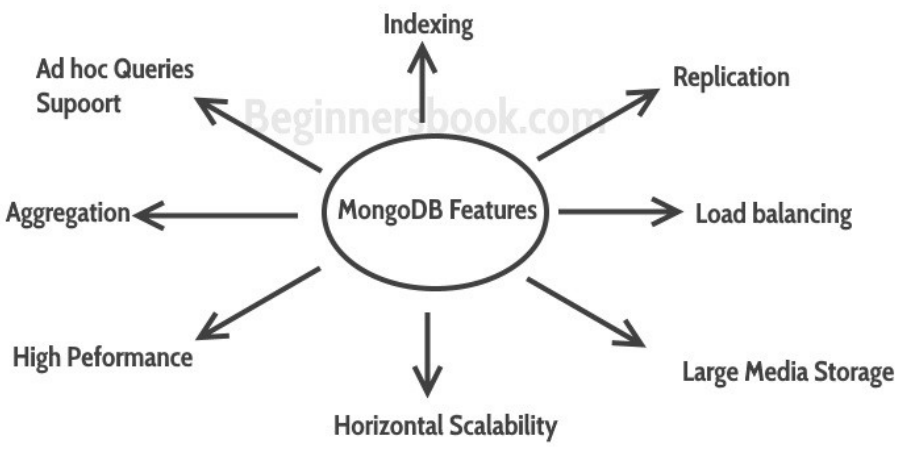
It expands the size of the existing system horizontally.	It expands the size of the existing system vertically.
It is easier to upgrade.	It is harder to upgrade and may involve downtime.
It is difficult to implement	It is easy to implement
It is costlier, as new server racks comprises of a lot of resources	It is cheaper as we need to just add new resources
It takes more time to be done	It takes less time to be done

Features/Advantages/Disadvantages of MongoDB

Features:

- **Schema-less Database:** It is the great feature provided by the MongoDB. A Schema-less database means one collection can hold different types of documents in it. Or in other words, in the MongoDB database, a single collection can hold multiple documents and these documents may consist of the different numbers of fields, content, and size. It is not necessary that the one document is similar to another document like in the relational databases. Due to this cool feature, MongoDB provides great flexibility to databases.
- **Document Oriented:** In MongoDB, all the data stored in the documents instead of tables like in RDBMS. In these documents, the data is stored in fields(key-value pair) instead of rows and columns which make the data much more flexible in comparison to RDBMS. And each document contains its unique object id.
- **Indexing:** In MongoDB database, every field in the documents is indexed with primary and secondary indices this makes easier and takes less time to get or search data from the pool of the data. If the data is not indexed, then database search each document with the specified query which takes lots of time and not so efficient.
- **Scalability:** MongoDB provides horizontal scalability with the help of sharding. Sharding means to distribute data on multiple servers, here a large amount of data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that reside across many physical servers. It will also add new machines to a running database.
- **Replication:** MongoDB provides high availability and redundancy with the help of replication, it creates multiple copies of the data and sends these copies to a different server so that if one server fails, then the data is retrieved from another server.
- **Aggregation:** It allows to perform operations on the grouped data and get a single result or computed result. It is similar to the SQL GROUPBY clause. It provides three different aggregations i.e, aggregation pipeline, map-reduce function, and single-purpose aggregation methods

- **High Performance:** The performance of MongoDB is very high and data persistence as compared to another database due to its features like scalability, indexing, replication, etc.
- **Load Balancing:** At the end of the day, optimal load balancing remains one of the holy grails of large-scale database management for growing enterprise applications. Properly distributing millions of client requests to hundreds or thousands of servers can lead to a noticeable (and much appreciated) difference in performance.
- **Ad-hoc queries for optimized, real-time analytics**
  When designing the schema of a database, it is impossible to know in advance all the queries that will be performed by end users. An ad hoc query is a short-lived command whose value depends on a variable. Each time an ad hoc query is executed, the result may be different, depending on the variables in question.



Advantages of MongoDB:

- It is a schema-less NoSQL database.
- It does not support join operation.
- It provides great flexibility to the fields in the documents.
- It contains heterogeneous data.
- It provides high performance, availability, scalability.
- It supports Geospatial efficiently.
- It is a document-oriented database, and the data is stored in BSON documents.

Disadvantages of MongoDB:

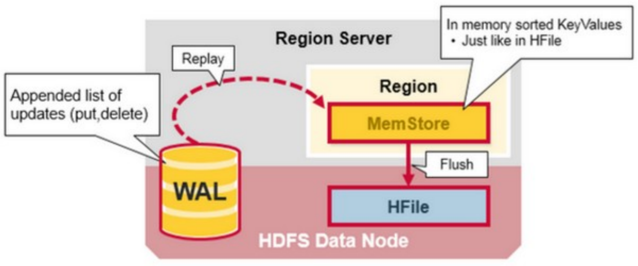
- It uses high memory for data storage.
- You are not allowed to store more than 16MB data in the documents.
- The nesting of data in BSON is also limited you are not allowed to nest data more than 100 levels.

Module 3

HDFS	HBase
HDFS is a distributed file system suitable for storing large files.	HBase is a database built on top of the HDFS.
HDFS does not support fast individual record lookups.	HBase provides fast lookups for larger tables.
It provides high latency batch processing.	It provides low latency access to single rows from billions of records (Random access).
It provides only sequential access of data.	HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups

HBase Crash and Data Recovery.

- Whenever a Region Server fails, ZooKeeper notifies to the HMaster about the failure.
- Then HMaster distributes and allocates the regions of crashed Region Server to many active Region Servers. To recover the data of the MemStore of the failed Region Server, the HMaster distributes the WAL to all the Region Servers.
- Each Region Server re-executes the WAL to build the MemStore for that failed region’s column family.
- The data is written in chronological order (in a timely order) in WAL. Therefore, Re-executing that WAL means making all the change that were made and stored in the MemStore file.
- So, after all the Region Servers executes the WAL, the MemStore data for all column family is recovered.



Cassandra

Cassandra can achieve both availability and scalability using a data structure that allows any node in the system to easily determine the location of a particular key in the cluster.

- **Node** − It is the place where data is stored.
- **Data center** − It is a collection of related nodes.
- **Cluster** − A cluster is a component that contains one or more data centers.
- **Commit log** − The commit log is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.
- **Mem-table** − A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.
- **SSTable** − It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- **Bloom filter** − These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

Replicas ensure data availability

- When Cassandra writes data it typically writes multiple copies (usually three) to different cluster nodes.
- This ensures that data isn’t lost if a node goes down or becomes unavailable.
- A replication factor specified when a database is created controls how many copies of data are written.
- When data is written, it takes time for updates to propagate across networks to remote hosts.
- Sometimes hosts will be temporarily down or unreachable.
- Cassandra is described as “eventually consistent” because it doesn’t guarantee that all replicas will always have the same data.
- This means there is no guarantee that the data you read is up to date.
- For example, if a data value is updated, and another user queries a replica to read the same data a few milliseconds later, the reader may end up with an older version of the data.

Consistency

- When performing a read or write operation a database client can specify a **consistency level**. The consistency level refers to the number of replicas that need to respond for a read or write operation to be considered complete.
- **Cassandra prefers availability over consistency**. It doesn't optimize for consistency. Instead, it gives you the flexibility to tune the consistency depending on your use case

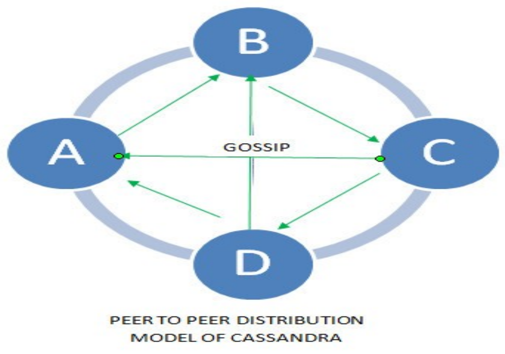
The Cassandra consistency level is defined as the minimum number of Cassandra nodes that must acknowledge a read or write operation before the operation can be considered successful. Different consistency levels can be assigned to different Edge keyspaces.
When connecting to Cassandra for read and write operations, Message Processor and Management Server nodes typically use the Cassandra value of LOCAL_QUORUM to specify the consistency level for a keyspace. However, some keyspaces are defined to use a consistency level of one.

Gossip Protocol

- Cassandra uses a gossip protocol to communicate with nodes in a cluster.
- Cassandra uses the gossip protocol to discover the location of other nodes in the cluster and get state information of other nodes in the cluster.
- The gossip process runs periodically on each node and exchanges state information

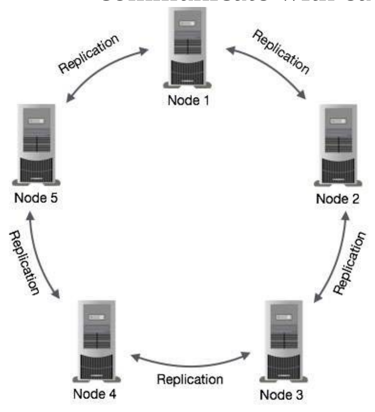
with three other nodes in the cluster.
- Eventually, information is propagated to all cluster nodes.

Even if there are 1000 nodes, information is propagated to all the nodes within a few seconds



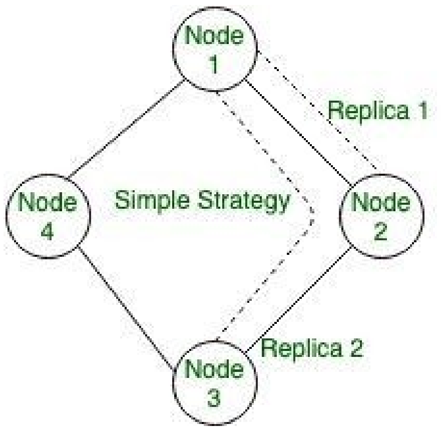
Data replication strategy in Cassandra.

- One or more of the nodes in a cluster act as replicas for a given piece of data.
- If it is detected that some of the nodes responded with an out-of-date value, Cassandra will return the most recent value to the client.
- After returning the most recent value, Cassandra performs a **read repair** in the background to update the stale values
- Cassandra uses the **Gossip Protocol** in the background to allow the nodes to communicate with each other and detect any faulty nodes in the cluster.

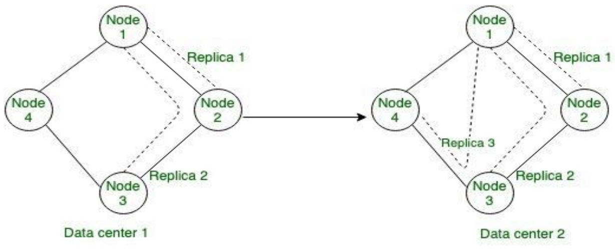


- There are two type of replication Strategy: Simple Strategy, and Network Topology Strategy.
- **1. Simple Strategy:**
  In this Strategy it allows a single integer RF (replication_factor) to be defined. It determines the number of nodes that should contain a copy of each row. For example, if replication_factor is 2, then two different nodes should store a copy of each row. It treats all nodes identically, ignoring any configured datacenters or racks

For E.g CREATE KEYSPACE User_data WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 2};

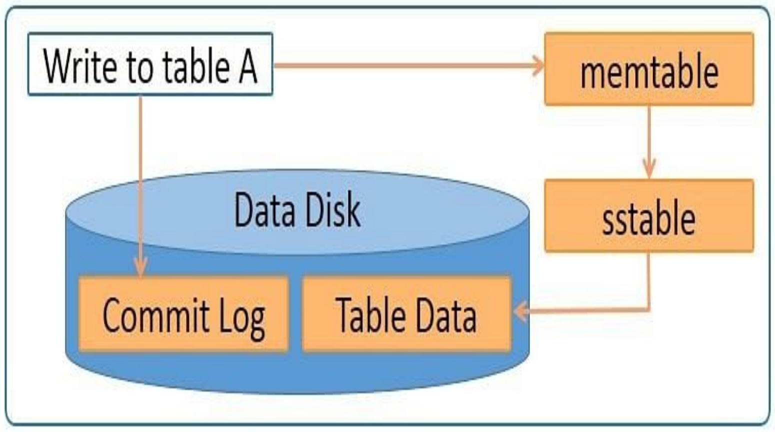


- **Network Topology Strategy:**
  In this strategy it allows a replication factor to be specified for each datacenter in the cluster. Even if your cluster only uses a single datacenter.
- CREATE KEYSPACE User_data WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 2, 'DC2' : 3} AND durable_writes =True;



Read/Write Operation in Cassandra:

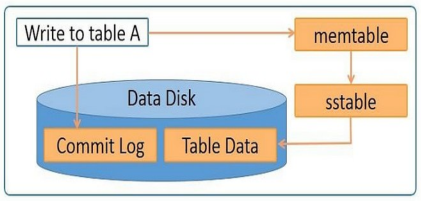
- Data is written to a commitlog on disk.
- The data is sent to a responsible node of Cassandra.
- Nodes write data to an in-memory table called memtable.
- From the memtable, data is written to sstable in memory. Sstable stands for Sorted String table. This has a consolidated data of all the updates to the table.
- From the sstable, data is updated to the actual table.
- If the responsible node is down, data will be written to another node identified as tempnode. The tempnode will hold the data temporarily till the responsible node comes alive.



- During read operations, Cassandra gets values from the mem-table
- Checks the bloom filter to find the appropriate SSTable that holds the required data.
- Fetches the data from the SSTable on disk

Cassandra
Write Mechanism

- Data is written to a commitlog on disk.
- The data is sent to a responsible node of Cassandra.
- Nodes write data to an in-memory table called memtable.
- From the memtable, data is written to sstable in memory. Sstable stands for Sorted String table. This has a consolidated data of all the updates to the table.
- From the sstable, data is updated to the actual table.
- If the responsible node is down, data will be written to another node identified as tempnode. The tempnode will hold the data temporarily till the responsible node comes alive.



Read Mechanism:

- During read operations, Cassandra gets values from the mem-table
- Checks the bloom filter to find the appropriate SSTable that holds the required data.
- Fetches the data from the SSTable on dis

HBase Vs Cassandra

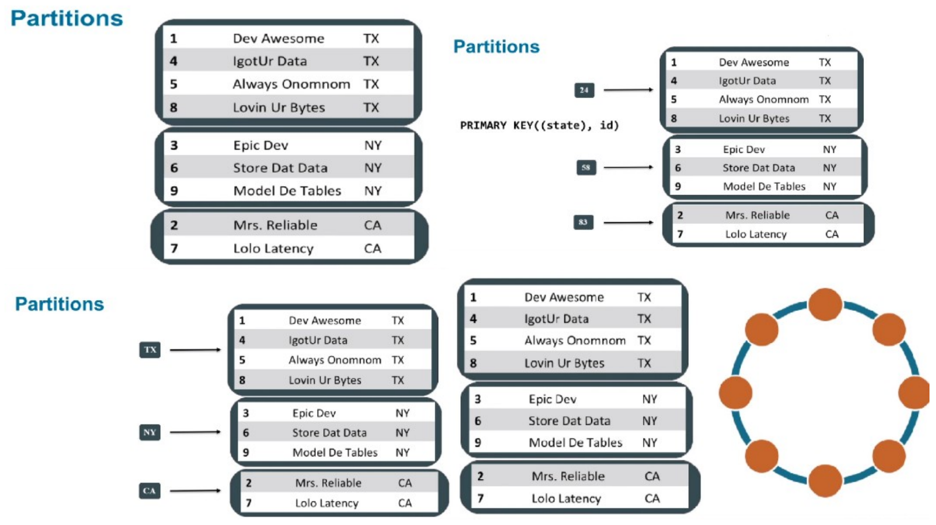
Parameters	**HBase**	**Cassandra**
Infrastructure	It uses Hadoop infrastructure.	Cassandra differs from Hadoop in terms of infrastructure and operation. It employs a variety of DBMS and infrastructure for a variety of applications.
Architecture Model	It is based on Master-Slave Architecture Model.	It is based on Active-Active Node Architecture Model.
Base of Database	HBase is based on Google BigTable.	Cassandra is based on Amazon DynamoDB.
Ordered Partitioning	HBase does not support ordered partitioning.	Cassandra allows for ordered partitioning. Because of this ordered division, Cassandra's row sizes can reach tens of megabytes.
Single Point of Failure (SPoF)	The cluster's accessibility depends on the availability of the Master node.	All nodes are equal so no such SPoF exists.
Consistency	HBase provides more consistency.	It does not provide as much consistency as HBase provides.
Coprocessor	HBase has the ability to use a Coprocessor.	Cassandra is not capable to support Coprocessor functionality.
Triggers	Triggers are supported because of Coprocessor capability.	Triggers are not supported.
Inter-communication	For internal node communication, HBase uses the Zookeeper protocol. Here, one node act as a master through which data is received by all other modes.	For internal node communication, Cassandra uses the Gossip protocol. Data will be transferred from one node to the next. To put it another way, we duplicate the data.
Query Language	The HBase query language is a custom-based language that must be learned.	Cassandra has its own CQL which is in line with SQL language.

Documentation	It is not as easy to learn as Cassandra.	Easy to learn because of better documentation than HBase.
Setup Cluster	HBase Cluster setup is not easy.	Cluster setup of Cassandra is easier than HBase.
Rebalancing of Clusters	HBase supports automatic rebalancing within clusters.	Cassandra also supports the feature of rebalancing but not of the entire cluster.
Transactions	HBase provides two methods for handling the transactions-	Cassandra provides two methods for handling the transactions-

Partition Key

Apache Cassandra is an open-source, distributed NoSQL database designed for linear scalability and high availability without performance that's typically higher distributed SQL options. It uses the Cassandra query language (CQL) to communicate. The system includes tables made up of Cassandra partition keys, composite keys, and clustering keys.
Each table demands a unique primary key. In Cassandra, a primary key consists of one or more partition keys and may include clustering key components. The Apache Cassandra partition key always precedes the clustering key since its hashed value determines which node will store the data.

Clustering columns are any fields listed after the partition key. Clustering columns store data in descending or ascending order within the partition so similar values can be retrieved quickly. Together, all these fields comprise the primary key.



CQLSH

Cassandra Query Language (CQL) is the query language used in Neo4J.
The Cassandra Query Language (CQL) is the primary language for communicating with the Apache Cassandra™ database. The most basic way to interact with Apache Cassandra is using the CQL shell, cqlsh. Using cqlsh, you can create keyspaces and tables, insert and query tables, plus much more.
Example:
Creating a single node: create(n) return n
CREATE KEYSPACE − Creates a KeySpace in Cassandra.
INSERT − Adds columns for a row in a table.

Module 4

Simple functions of Key-Value Stores:

1. Put:
   - put(key , value)
2. Get
   - get(key)
3. Delete
   - delete(key)

Features of REDIS:

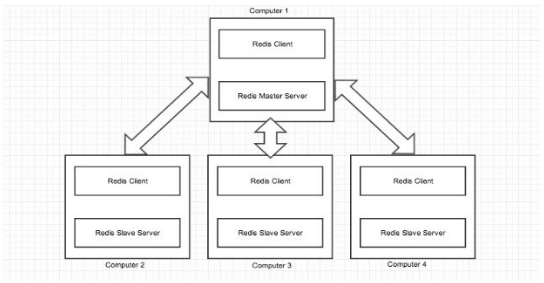
Speed:
Redis stores the whole dataset in primary memory that's why it is extremely fast
Persistence:
While all the data lives in memory, changes are asynchronously saved on disk using flexible policies based on elapsed time and/or number of updates since last save.
Data Structures:
Redis supports various types of data structures such as strings, hashes, sets, lists, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.
Atomic Operations:
Redis operations working on the different Data Types are atomic, so it is safe to set or increase a key, add and remove elements from a set, increase a counter etc.
Supported Languages:
Redis supports a lot of languages such as ActionScript, C, C++, C# etc.
Master/Slave Replication:
Redis follows a very simple and fast Master/Slave replication. It takes only one line in the configuration file to set it up
Sharding:
Redis supports sharding. It is very easy to distribute the dataset across multiple Redis instances, like another key-value store.
Portable:
Redis is written in ANSI C and works in most POSIX systems like Linux, BSD, Mac OS X, Solaris, and so on. Redis is reported to compile and work under WIN32 if compiled with Cygwin, but there is no official support for Windows currently.

REDIS

Redis is an open source, BSD licensed, advanced key value store. In-memory key value store with persistence (Database management systems which stores database on computers main memory rather than a hard disk).
It stands for REmote Dictionary Server. It is referred to as data structure server since the keys can contain strings, hashes, lists, sets and sorted sets.

This system works using three main mechanisms:

1. When a master and replica instances are well-connected, the master keeps the replica updated by sending a stream of commands to the replica to replicate the effects on the dataset happening in the master side due to client writes, keys expired or evicted, any other action changing the master dataset.
2. When the link between the master and the replica breaks, for network issues or because a timeout is sensed in the master or the replica, the replica reconnects and attempts to proceed with a partial resynchronization: it means that it will try to just obtain the part of the stream of commands it missed during the disconnection.
3. When a partial resynchronization is not possible, the replica will ask for a full resynchronization. This will involve a more complex process in which the master needs to create a snapshot of all its data, send it to the replica, and then continue sending the stream of commands as the dataset changes.



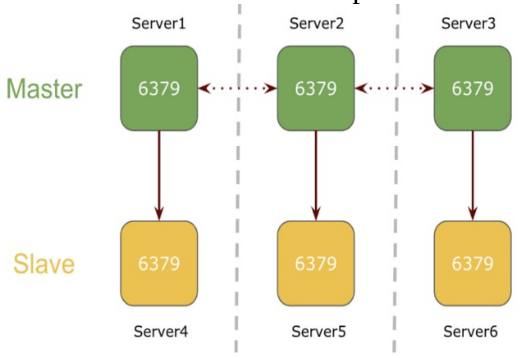
At the base of Redis replication (excluding the high availability features provided as an additional layer by Redis Cluster or Redis Sentinel) there is a leader follower (master-replica) replication that is simple to use and configure.
It allows replica Redis instances to be exact copies of master instances.
The replica will automatically reconnect to the master every time the link breaks and will attempt to be an exact copy of it regardless of what happens to the master.
Redis uses by default asynchronous replication, which being low latency and high performance, is the natural replication mode for the vast majority of Redis use cases.
However, Redis replicas asynchronously acknowledge the amount of data they received periodically with the master.
So, the master does not wait every time for a command to be processed by the replicas, however it knows, if needed, what replica already processed what command. This allows having optional synchronous replication.
Synchronous replication of certain data can be requested by the clients using the WAIT command.
However, WAIT is only able to ensure there are the specified number of acknowledged copies in the other Redis instances, it does not turn a set of Redis instances into a CP system with strong consistency: acknowledged writes can still be lost during a failover, depending on the exact configuration of the Redis persistence.
However, with WAIT the probability of losing a write after a failure event is greatly reduced to certain hard to trigger failure modes.

Concept of Redis Cluster

- Redis Cluster is an active-passive cluster implementation that consists of master and slave nodes.
- The cluster uses hash partitioning to split the keyspace into 16,384 key slots, with each master responsible for a subset of those slots.
- Each slave replicates a specific master and can be reassigned to replicate another master or be elected to a master node as needed.

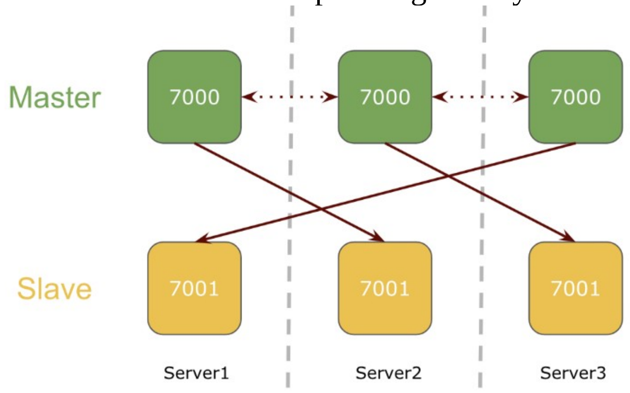
6 Node M/S Cluster

- In a 6 node cluster mode, 3 nodes will be serving as a master and the 3 nodes will be their respective slave.
- Here, Redis service will be running on port 6379 on all servers in the cluster. Each master server is replicating the keys to its respective Redis slave node assigned during the cluster creation process.

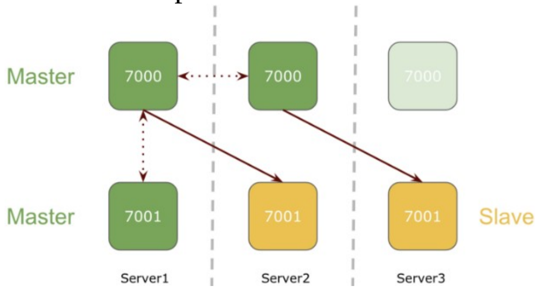


3 Node M/S Cluster

- In a 3 node cluster mode, there will be 2 redis services running on each server on different ports.
- All 3 nodes will be serving as a master with redis slave on cross nodes.
- Here, two redis services will be running on each server on two different ports and each master is replicating the keys to its respective redis slave running on other nodes.

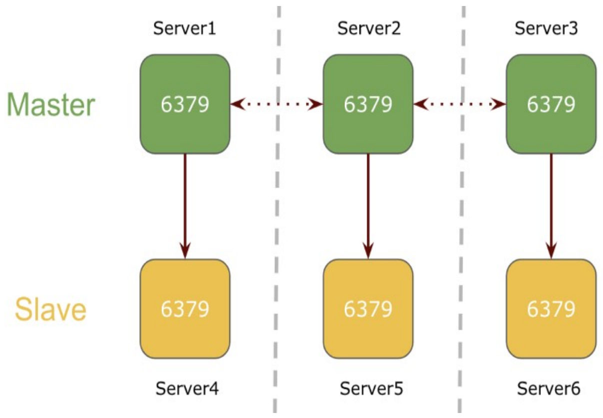


Redis service goes down:
If Redis service goes down on one of the nodes in Redis 3-node cluster setup, its respective slave will be promoted as the master.



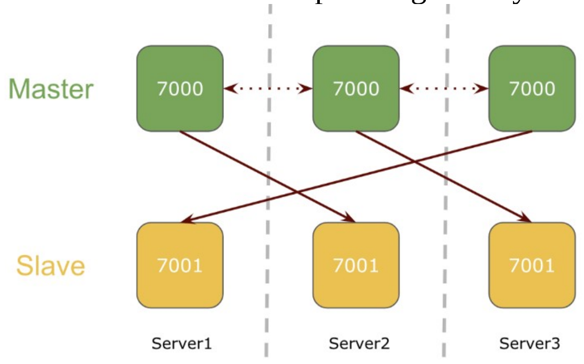
6 Node M/S Cluster

- In a 6 node cluster mode, 3 nodes will be serving as a master and the 3 nodes will be their respective slave.
- Here, Redis service will be running on port 6379 on all servers in the cluster. Each master server is replicating the keys to its respective Redis slave node assigned during the cluster creation process.



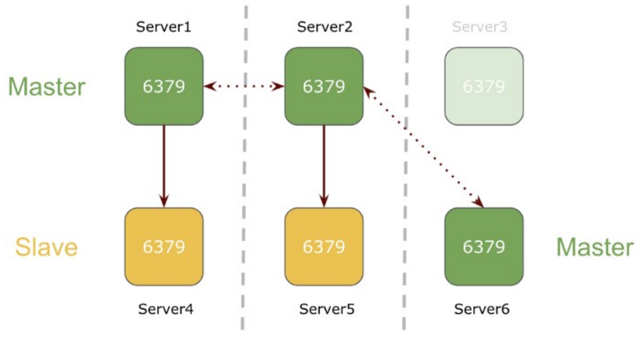
3 Node M/S Cluster

- In a 3 node cluster mode, there will be 2 redis services running on each server on different ports.
- All 3 nodes will be serving as a master with redis slave on cross nodes.
- Here, two redis services will be running on each server on two different ports and each master is replicating the keys to its respective redis slave running on other nodes.



WHAT IF Redis Goes Down

- If one of the nodes goes down in the Redis 6-node cluster setup, its respective slave will be promoted as the master.
- Master Server3 goes down and its slave Server6 is promoted as the master

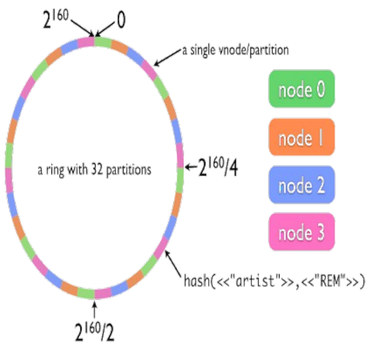


RIAK

Riak's default mode of operation is to work as a cluster consisting of multiple nodes, i.e. multiple well-connected data hosts.
Each host in the cluster runs a single instance of Riak, referred to as a Riak node. Each Riak node manages a set of virtual nodes, or vnodes, that are responsible for storing a separate portion of the keys stored in the cluster.
In contrast to some high-availability systems, Riak nodes are not clones of one another, and they do not all participate in fulfilling every request. Instead, you can configure, at runtime or at request time, the number of nodes on which data is to be replicated, as well as when replication occurs, and which merge strategy and failure model are to be followed.
Ring:
Though much of this section is discussed in our annotated discussion of the Amazon Dynamo paper, it nonetheless provides a summary of how Riak implements the distribution of data throughout a cluster.

Any client interface to Riak interacts with objects in terms of the bucket and key in which a value is stored, as well as the bucket type that is used to set the bucket's properties.

Internally, Riak computes a 160-bit binary hash of each bucket/key pair and maps this value to a position on an ordered ring of all such values. This ring is divided into partitions, with each Riak vnode responsible for one of these partitions (we say that each vnode claims that partition).

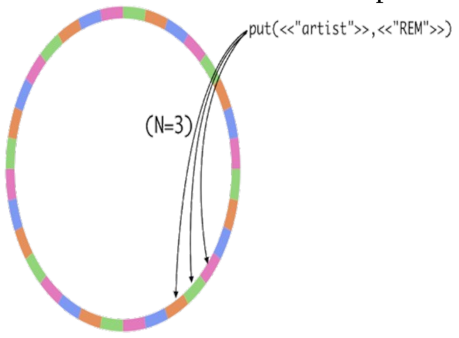


The nodes of a Riak cluster each attempt to run a roughly equal number of vnodes at any given time. In the general case, this means that each node in the cluster is responsible for 1/(number of nodes) of the ring, or (number of partitions)/(number of nodes) vnodes.

If two nodes define a 16-partition cluster, for example, then each node will run 8 vnodes. Nodes attempt to claim their partitions at intervals around the ring such that there is an even distribution amongst the member nodes and that no node is responsible for more than one replica of a key**.**
Intelligent Replication:
When an object is being stored in the cluster, any node may participate as the coordinating node for the request. The coordinating node consults the ring state to determine which vnode owns the partition in which the value's key belongs, then sends the write request to that vnode as well as to the vnodes responsible for the next N-1 partitions in the ring (where N is a configurable parameter that describes how many copies of the value to store). The write request may also specify that at least W (=< N) of those vnodes reply with success, and that DW (=< W) reply with success only after durably storing the value.

A read, or GET, request operates similarly, sending requests to the vnode that “claims” the partition in which the key resides, as well as to the next N-1 partitions. The request also specifies R (=< N), the number of vnodes that must reply before a response is returned.

Here is an illustration of this process:



Gossiping:
The ring state is shared around the cluster by means of a “gossip protocol.” Whenever a node changes its claim on the ring, it announces, i.e. “gossips,” this change to other nodes so that the other nodes can respond appropriately. Nodes also periodically re-announce what they know about ring in case any nodes happened to miss previous updates.

Neo4J

Nodes: Node is a fundamental unit of a Graph. It contains properties with key-value pairs as shown in the following image.
Relationships: Relationships are another major building block of a Graph Database. It connects two nodes as depicted in the following figure.
Properties: Property is a key-value pair to describe Graph Nodes and Relationships.
Labels: Label associates a common name to a set of nodes or relationships. A node or relationship can contain one or more labels. We can create new labels to existing nodes or relationships. We can remove the existing labels from the existing nodes or relationships.

Features of Neo4J

SQL Like easy query language Neo4j CQL
•It follows Property Graph Data Model
•It supports Indexes by using Apache Lucence
•It supports UNIQUE constraints
•It contains a UI to execute CQL Commands: Neo4j Data Browser
•It supports full ACID (Atomicity, Consistency, Isolation and Durability) rules
•It uses Native graph storage with Native GPE (Graph Processing Engine)
•It supports exporting of query data to JSON and XLS format
•It provides REST API to be accessed by any Programming Language like Java, Spring, Scala etc.
•It provides Java Script to be accessed by any UI MVC Framework like Node JS.
•It supports two kinds of Java API: Cypher API and Native Java API to develop Java applications.

Use Case Neo4J databases.
Fraud Detection
While no fraud prevention measures are perfect, significant improvements occur when you look beyond individual data points to the connections that link them. Understanding the connections between data, and deriving meaning from these links, doesn’t necessarily mean gathering new data. You can draw significant insights from your existing data simply by reframing the problem in a new way: as a graph. Unlike most other ways of looking at data, graphs are designed to express relatedness. Graph databases uncover patterns that are difficult to detect using traditional representations such as tables. An increasing number of companies use graph databases to solve a variety of connected data problems
Real-Time Recommendation Engines
Whether your enterprise operates in the retail, social, services or media sectors, offering your users highly targeted, real-time recommendations is essential to maximizing customer value and staying competitive. Unlike other business data, recommendations must be inductive and contextual in order to be considered relevant by your end consumers. With a graph database, you’re able to capture a customer’s browsing behavior and demographics and combine those with their buying history to instantly analyze their current choices and then immediately provide relevant recommendations – all before a potential customer click to a competitor’s website.
Master Data Management
Master data is the lifeblood of your enterprise, including data such as:
• Users
• Customers

• Products
• Accounts
• Partners
• Sites
• Business units

Many business applications use master data and its often held in many different places, with lots of overlap and redundancy, in different formats, and with varying degrees of quality and means of access. Master data management (MDM) is the practice of identifying, cleaning, storing, and – most importantly – governing this data.

Network and IT Operations

By their nature, networks are graphs. Graph databases are, therefore, an excellent fit for modeling, storing and querying network and IT operational data no matter which side of the firewall your business is on – whether it's a communications network or a data center. Today, graph databases are being successfully employed in the areas of telecommunications, network management, impact analysis, cloud platform management and data center and IT asset management. In all these domains, graph databases store configuration information to alert operators in real time to potential shared failure modes in the infrastructure and to reduce problem analysis and resolution times from hours to seconds.

Module 5

Big Data
The definition of big data is data that contains greater variety, arriving in increasing volumes and with more velocity. This is also known as the three Vs. Put simply, big data is larger, more complex data sets, especially from new data sources.
BIG Data Use Cases

- Bulk image processing
- Public web page data
- Remote sensor data
- Event log data
- Mobile phone data
- Social media data
- Game data
- Open linked data

Types of Big Data
Structured data:
- Structured data is data that has been predefined and formatted to a set structure before being placed in data storage, which is often referred to as schema-on-write. The best example of structured data is the relational database: the data has been formatted into precisely defined fields, such as credit card numbers or address, to be easily queried with SQL.
- Examples of structured data:
- Structured data is an old, familiar friend. It's the basis for inventory control systems and ATMs. It can be human- or machine-generated.

Unstructured data:
- Unstructured data is data stored in its native format and not processed until it is used, which is known as schema-on-read. It comes in a myriad of file formats, including email, social media posts, presentations, chats, IoT sensor data, and satellite imagery.
- Examples of unstructured data:
- It lends itself well to determining how effective a marketing campaign is, or to uncovering potential buying trends through social media and review websites. It can also be very useful to the enterprise by assisting with monitoring for policy compliance, as it can be used to detect patterns in chats or suspicious email trends.

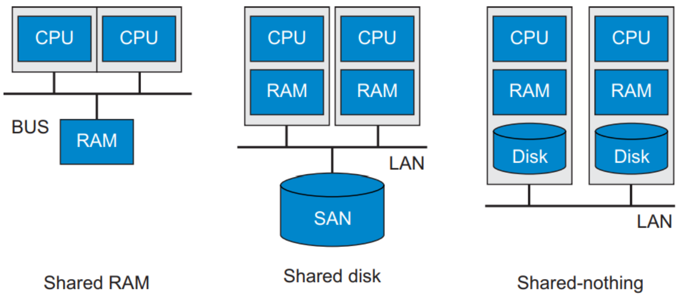
Semi-structured data:
- Semi-structured data is data that does not conform to a data model but has some structure. It lacks a fixed or rigid schema. It is the data that does not reside in a rational database but that have some organizational properties that make it easier to analyze. With some processes, we can store them in the relational database.
- Examples of semi-structured Data:
- E-mails
- XML and other markup languages
- Binary executables

- TCP/IP packets
- Zipped files
- Integration of data from different sources
- Web pages

Big data with a shared-nothing architecture, master-slave versus peer-to-peer models.

Shared-nothing architecture

- In a shared RAM architecture, many CPUs access a single shared RAM over a high-speed bus.
- In a shared disk system, processors have independent RAM but share disk using a storage area network (SAN).
- In a Shared nothing architecture, processors have independent RAM as well as disks and connected with high-speed Bus. It works effectively in a high volume and read-write environment.
- This System is used in big data solutions: cache-friendly, using low-cost commodity hardware

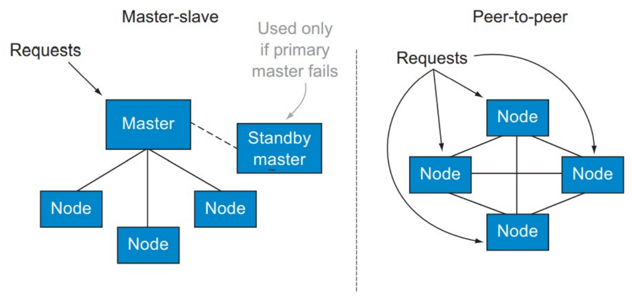


Distribution models

- From a distribution perspective, there are two main models:
- ***Master-Slave*** *and* ***Peer-to Peer.***
- Distribution models determine the responsibility for processing data when a request is made
- Peer-to-peer models may be more resilient to failure than master-slave models.
- Some master-slave distribution models have single points of failure that might impact your system availability, so you might need to take special care when configuring these systems

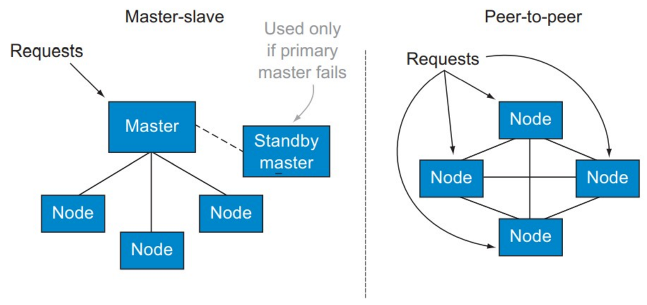
Master Slave

Master-slave replication is most helpful for scaling when you have a read-intensive dataset. You can scale horizontally to handle more read requests by adding more slave nodes. Should the master fail, the slaves can still handle read requests.
Master-slave replication helps with read scalability but doesn't help with scalability of writes. It provides resilience against failure of a slave, but not of a master.



Peer to Peer:

- With a peer-to-peer replication cluster, you can ride over node failures without losing access to data.
- Furthermore, you can easily add nodes to improve your performance.
- The biggest complication is, again, consistency.
- When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time—a write-write conflict. Inconsistencies on read lead to problems but at least they are relatively transient.



The four ways that NoSQL systems handle big data problems.

1 Moving queries to the data, not data to the queries

- Apart from large graph databases, most NoSQL systems use commodity processors that each hold a subset of the data on their local shared-nothing drives.
- When a client wants to send a general query to all nodes that hold data, it's more efficient to send the query to each node than it is to transfer large datasets to a central processor.
- This may seem obvious, but it's amazing how many traditional databases still can't distribute queries and aggregate query results.
- This simple rule helps you understand how NoSQL databases can have dramatic performance advantages over systems that weren't designed to distribute queries to the data nodes.
- Keeping all the data within each data node in the form of logical documents means that only the query itself and the result need to be moved over a network. This keeps your big data queries fast.

2 Using hash rings to evenly distribute data on a cluster

- One of the most challenging problems with distributed databases is figuring out a consistent way of assigning a document to a processing node.
- Using a hash ring technique to evenly distribute big data loads over many servers with a randomly generated 40-character key is a good way to evenly distribute a network load.
- Hash rings are common in big data solutions because they consistently determine how to assign a piece of data to a specific processor.
- Hash rings take the leading bits of a document's hash value and use this to determine which node the document should be assigned.
- This allows any node in a cluster to know what node the data lives on and how to adapt to new assignment methods as your data grows.
- Partitioning keys into ranges and assigning different key ranges to specific nodes is known as keyspace management.

3 Using replication to scale reads:

- Databases use replication to make backup copies of data in real time.
- Using replication allows you to horizontally scale read requests. Figure shows this structure.

4. Letting the database distribute queries evenly to data nodes:
In order to get high performance from queries that span multiple nodes, it's important to separate the concerns of query evaluation from query execution.

- Figure NoSQL systems move the query to a data node, but don't move data to a query node.
- In this example, all incoming queries arrive at query analyzer nodes.
- These nodes then forward the queries to each data node.
- If they have matches, the documents are returned to the query node.
- The query won't return until all data nodes (or a response from a replica) have responded to the original query request.
- If the data node is down, a query can be redirected to a replica of the data node.
- This approach is somewhat similar to the concept of federated search. Federated search takes a single query and distributes it to distinct servers and then combines the results together to give the user the impression they're searching a single system.

Linear scalability and expressivity

What's the relationship between scalability and your ability to perform complex queries on your data? As we mentioned earlier, linear scalability is the ability to get a consistent amount of performance improvement as you add additional processors to your cluster. Expressivity is the ability to perform fine-grained queries on individual elements of your dataset. Understanding how well each NoSQL technology performs in terms of scalability and expressivity is necessary when you're selecting a NoSQL solution. To select the right system, you'll need to identify the scalability and expressivity requirements of your system and then make sure the system that you select meets both criteria. Scalability and expressivity can be difficult to quantify, and vendor claims may not match actual performance for a particular business problem. If you're making critical business decisions, we recommend you create a pilot project and simulate an actual load using leased cloud systems.

- Let's look at two extreme cases:
    - a key-value store and
    - a document store.
- After reviewing the vendor brochures, you feel that both systems have similar linear scalability rules that meet your business growth needs.
- Which one is right for your project?
- The answer is how you want to retrieve the data from each of these systems.
- If you only need to store images and using a URL to specify an image is appropriate, then a key-value store is the right choice.
- If you need to be able to store items and query on a specific subset of items based on the items' properties, then a key-value store isn't a good match since the value portion of a key is opaque to queries.
- In contrast, a document store that can index dates, amounts, and item descriptions might be a better match.