Problem Statement:

Problem 1: **NoSQL Databases**

Solution:

A NoSQL database environment is, simply put, a non-relational and largely distributed database system that enables rapid, ad-hoc organization and analysis of extremely high-volume, disparate data types. NoSQL databases are sometimes referred to as cloud databases, non-relational databases, Big Data databases and a myriad of other terms and were developed in response to the sheer volume of data being generated, stored and analyzed by modern users (user-generated data) and their applications (machine-generated data).

In general, NoSQL databases have become the first alternative to relational databases, with scalability, availability, and fault tolerance being key deciding factors. They go well beyond the more widely understood legacy, relational databases (such as Oracle, SQL Server and DB2 databases) in satisfying the needs of today's modern business applications. A very flexible and schema-less data model, horizontal scalability, distributed architectures, and the use of languages and interfaces that are "not only" SQL typically characterize this technology.

From a business standpoint, considering a NoSQL or 'Big Data' environment has been shown to provide a clear competitive advantage in numerous industries. In the 'age of data', this is compelling information as a great saying about the importance of data is summed up with the following "if your data isn't growing then neither is your business".

NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications:

- Developers are working with applications that create massive volumes of new, rapidly changing data types — structured, semi-structured, unstructured and polymorphic data.
- Long gone is the twelve-to-eighteen month waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.
- Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users.
- Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.

Relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available today.

NoSQL does not have a prescriptive definition but we can make a set of common observations, such as:

- Not using the relational model
- Running well on clusters
- Mostly open-source
- Built for the 21st century web estates
- Schema-less

The Benefits of NoSQL

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address:

- Large volumes of rapidly changing structured, semi-structured, and unstructured data
- Agile sprints, quick schema iteration, and frequent code pushes
- Object-oriented programming that is easy to use and flexible
- Geographically distributed scale-out architecture instead of expensive, monolithic architecture

Problem 2: Types of NoSQL Databases

Solution:

There are four general types of NoSQL databases, each with their own specific attributes:

- **Graph database:**
  Based on graph theory, these databases are designed for data whose relations are well represented as a graph and has elements which are interconnected, with an undetermined number of relations between them. Examples include: Neo4j and Titan.

- **Key-Value store:**
  We start with this type of database because these are some of the least complex NoSQL options. These databases are designed for storing data in a schema-less way. In a key-value store, all of the data within consists of an indexed key and a value, hence the name. Examples of this type of database include: Cassandra, DyanmoDB, Azure Table Storage (ATS), Riak, BerkeleyDB.

- **Column store:**
  Also known as wide-column stores, This store instead of storing data in rows, these databases are designed for storing data tables as sections of columns of data, rather than as rows of data. While this simple description sounds like the inverse of a standard database, wide-column stores offer very high performance and a highly scalable architecture. Examples include: HBase, BigTable and HyperTable.

- **Document database:**

Expands on the basic idea of key-value stores where "documents" contain more complex in that they contain data and each document is assigned a unique key, which is used to retrieve the document. These are designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Examples include: MongoDB and CouchDB.

The following table lays out some of the key attributes that should be considered when evaluating NoSQL databases.

DATAMODEL	**PERFORMANCE**	**SCALABILITY**	**FLEXIBILITY**	**COMPLEXITY**	**FUNCTIONALITY**
Key-value store	High	High	High	None	Variable (None)
Column Store	High	High	Moderate	Low	Minimal
Document Store	High	Variable (High)	High	Low	Variable (Low)
Graph Database	Variable	Variable	High	High	Graph Theory

Problem 3: CAP Theorem:

Solution:

In a distributed system, managing consistency (C), availability (A) and partition toleration (P) is important, Eric Brewer put forth the CAP theorem which states that in any distributed system we can choose only two of consistency, availability or partition tolerance.

It states that fundamentally, there is a tension in asynchronous networks (those whose nodes do not have access to a shared clock) between three desirable properties of data store services distributed across more than one node:

- Availability - will a request made to the data store always eventually complete, no matter what (non-total) pattern of failures have occurred?
- Consistency - will all executions of reads and writes seen by all nodes be *sequentially consistent?* Roughly, this means that the results of 'earlier' writes are seen by 'later' reads, but the formal definition is a little more subtle.
- Partition tolerance - the network can suffer arbitrary failure patterns. This can be modelled as the refusal of the network to deliver any subset of the messages sent between nodes. Note that the failure of a single node can count as a 'partition'.

You cannot build a general data store that is continually available, sequentially consistent and tolerant to any partition pattern. You can build one (trivially) that has any two of these three properties.

Many NoSQL databases try to provide options where the developer has choices where they can tune the database as per their needs.

For example if you consider Riak a distributed key-value database.

There are essentially three variables r, w, n where

- r=number of nodes that should respond to a read request before it's considered successful.
- w=number of nodes that should respond to a write request before it's considered successful.
- n=number of nodes where the data is replicated aka replication factor.

In a Riak cluster with 5 nodes, we can tweak the r,w,n values to make the system very consistent by setting r=5 and w=5 but now we have made the cluster susceptible to network partitions since any write will not be considered successful when any node is not responding. We can make the same cluster highly available for writes or reads by setting r=1 and w=1 but now consistency can be compromised since some nodes may not have the latest copy of the data.

The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency of data. Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.

NoSQL databases provide developers lot of options to choose from and fine tune the system to their specific requirements. Understanding the requirements of how the data is going to be consumed by the system, questions such as is it read heavy vs write heavy, is there a need to query data with random query parameters, will the system be able handle inconsistent data.

Understanding these requirements becomes much more important, for long we have been used to the default of RDBMS which comes with a standard set of features no matter which product is chosen and there is no possibility of choosing some features over other. The availability of choice in NoSQL databases, is both good and bad at the same time. Good because now we have choice to design the system according to the requirements. Bad because now you have a choice and we have to make a good choice based on requirements and there is a chance where the same database product may be used properly or not used properly.

An example of feature provided by default in RDBMS is transactions, our development methods are so used to this feature that we have stopped thinking about what would happen when the database does not provide transactions. Most NoSQL databases do not provide transaction support by default, which means the developers have to think how to implement transactions, does every write have to have the safety of transactions or can the write be segregated into “critical that they succeed” and “it's okay if I lose this write” categories. Sometimes deploying external transaction managers like ZooKeeper can also be a possibility.

Problem 4: HBase Architecture

Solution:

Components of HBase Architecture

HBase has three major components i.e., **HMaster Server**, **HBase Region Server, Regions** and **Zookeeper**.

The below figure explains the hierarchy of the HBase Architecture.

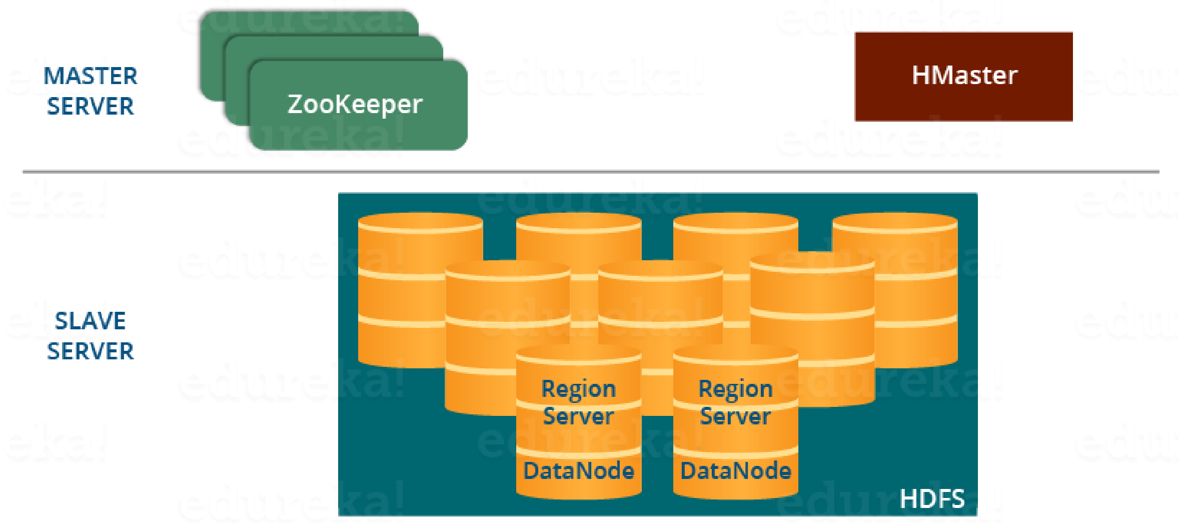

Figure: HMaster & Region Servers

Before going to the HMaster, let's understand Regions as all these Servers (HMaster, Region Server, and Zookeeper) are placed to coordinate and manage Regions and perform various operations inside the Regions.

HBase Architecture: Region

A region contains all the rows between the start key and the end key assigned to that region. HBase tables can be divided into a number of regions in such a way that all the columns of a column family is stored in one region. Each region contains the rows in a sorted order.

Many regions are assigned to a **Region Server**, which is responsible for handling, managing, executing reads and writes operations on that set of regions.

So, concluding in a simpler way:

- A table can be divided into a number of regions. A Region is a sorted range of rows storing data between a start key and an end key.
- A Region has a default size of 256MB which can be configured according to the need.
- A Group of regions is served to the clients by a Region Server.
- A Region Server can serve approximately 1000 regions to the client.

Now starting from the top of the hierarchy, let’s look at the HMaster Server which acts similarly as a NameNode in **HDFS**. Then, moving down in the hierarchy, I will take you through ZooKeeper and Region Server.

HBase Architecture: HMaster

As in the below image, you can see the HMaster handles a collection of Region Server which resides on DataNode. Let us understand how HMaster does that.

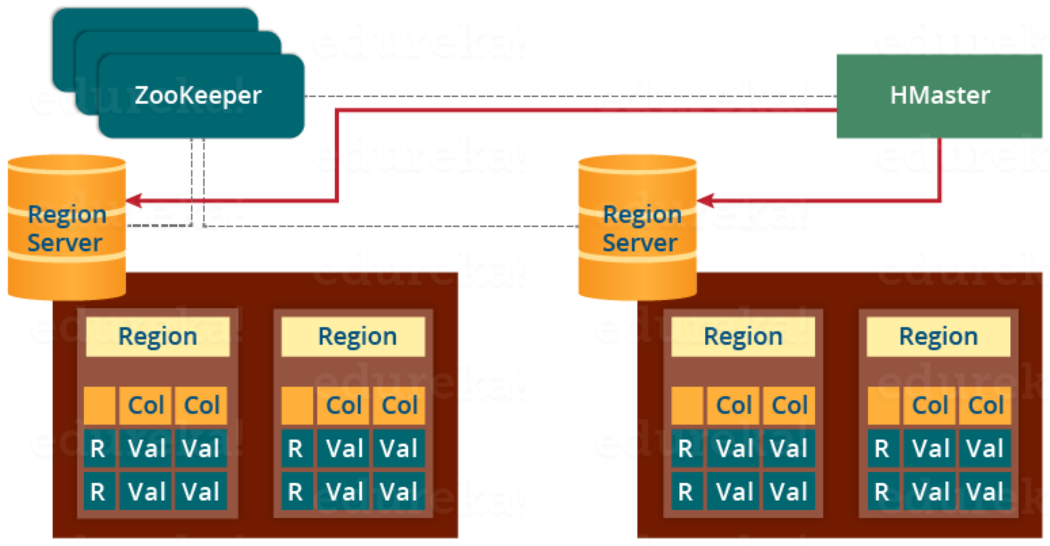

Figure: Components of HBase

- HBase HMaster performs DDL operations (create and delete tables) and assigns regions to the Region servers as you can see in the above image.
- It coordinates and manages the Region Server (similar as NameNode manages DataNode in HDFS).
- It assigns regions to the Region Servers on startup and re-assigns regions to Region Servers during recovery and load balancing.
- It monitors all the Region Server’s instances in the cluster (with the help of Zookeeper) and performs recovery activities whenever any Region Server is down.

- It provides an interface for creating, deleting and updating tables.

HBase has a distributed and huge environment where HMaster alone is not sufficient to manage everything. So, you would be wondering what helps HMaster to manage this huge environment? That's where ZooKeeper comes into the picture. After we understood how HMaster manages HBase environment, we will understand how Zookeeper helps HMaster in managing the environment.

HBase Architecture: ZooKeeper – The Coordinator

This below image explains the ZooKeeper's coordination mechanism.

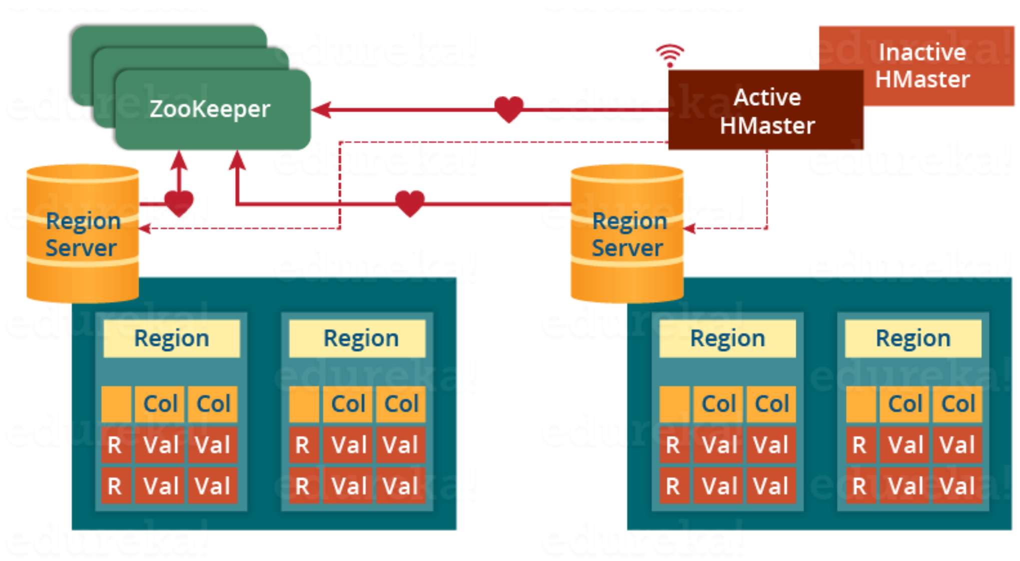

Figure: ZooKeeper as Coordination Service

- Zookeeper acts like a coordinator inside HBase distributed environment. It helps in maintaining server state inside the cluster by communicating through sessions.
- Every Region Server along with HMaster Server sends continuous heartbeat at regular interval to Zookeeper and it checks which server is alive and available as mentioned in above image. It also provides server failure notifications so that, recovery measures can be executed.
- Referring from the above image you can see, there is an inactive server, which acts as a backup for active server. If the active server fails, it comes for the rescue.
- The active HMaster sends heartbeats to the Zookeeper while the inactive HMaster listens for the notification send by active HMaster. If the active HMaster fails to send a heartbeat the session is deleted and the inactive HMaster becomes active.
- While if a Region Server fails to send a heartbeat, the session is expired and all listeners are notified about it. Then HMaster performs suitable recovery actions which we will discuss later in this blog.

- Zookeeper also maintains the .META Server's path, which helps any client in searching for any region. The Client first has to check with .META Server in which Region Server a region belongs, and it gets the path of that Region Server.

HBase Architecture: Meta Table

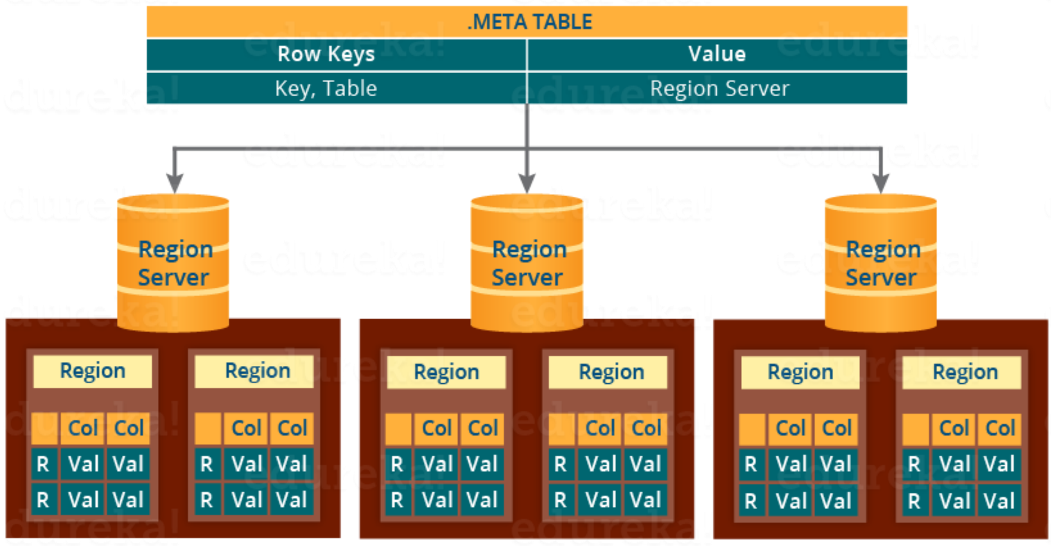

Figure: META Table

- The META table is a special HBase catalog table. It maintains a list of all the Regions Servers in the HBase storage system, as you can see in the above image.
- Looking at the figure you can see, **.META** file maintains the table in form of keys and values. Key represents the start key of the region and its id whereas the value contains the path of the Region Server.

HBase Architecture: Components of Region Server

This below image shows the components of a Region Server. Now, I will discuss them separately.

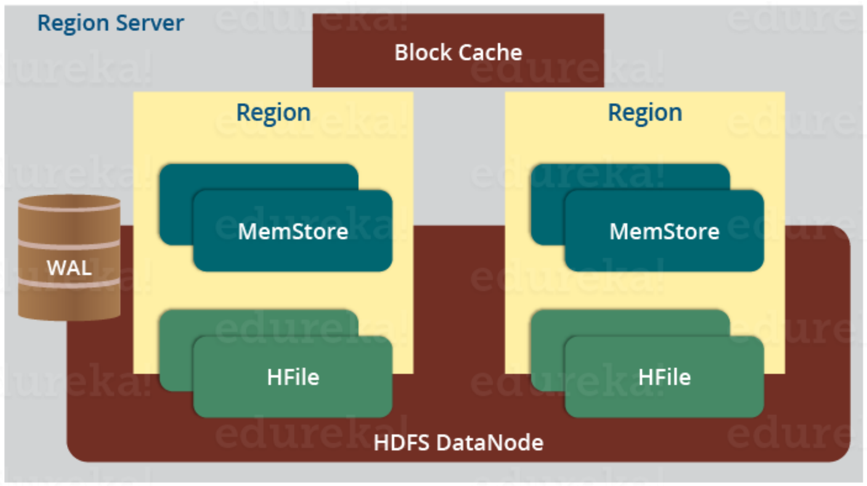

Figure: Region Server Components

A Region Server maintains various regions running on the top of **HDFS**. Components of a Region Server are:

- **WAL:** As you can conclude from the above image, Write Ahead Log (WAL) is a file attached to every Region Server inside the distributed environment. The WAL stores the new data that hasn't been persisted or committed to the permanent storage. It is used in case of failure to recover the data sets.

- **Block Cache:** From the above image, it is clearly visible that Block Cache resides in the top of Region Server. It stores the frequently read data in the memory. If the data in BlockCache is least recently used, then that data is removed from BlockCache.

- **MemStore:** It is the write cache. It stores all the incoming data before committing it to the disk or permanent memory. There is one MemStore for each column family in a region. As you can see in the image, there are multiple MemStores for a region because each region contains multiple column families. The data is sorted in lexicographical order before committing it to the disk.

- **HFile:** From the above figure you can see HFile is stored on HDFS. Thus it stores the actual cells on the disk. MemStore commits the data to HFile when the size of MemStore exceeds.

HBase Architecture: How Search Initializes in HBase?

Zookeeper stores the META table location. Whenever a client approaches with a read or writes requests to HBase following operation occurs:

1. The client retrieves the location of the META table from the ZooKeeper.
2. The client then requests for the location of the Region Server of corresponding row key from the META table to access it. The client caches this information with the location of the META Table.
3. Then it will get the row location by requesting from the corresponding Region Server.

The client uses its cache to retrieve the location of META table and previously read row key's Region Server. Then the client will not refer to the META table, until and unless there is a miss because the region is shifted or moved. Then it will again request to the META server and update the cache.

As every time, clients does not waste time in retrieving the location of Region Server from META Server, thus, this saves time and makes the search process faster.

HBase Architecture: HBase Write Mechanism

This below image explains the write mechanism in HBase.

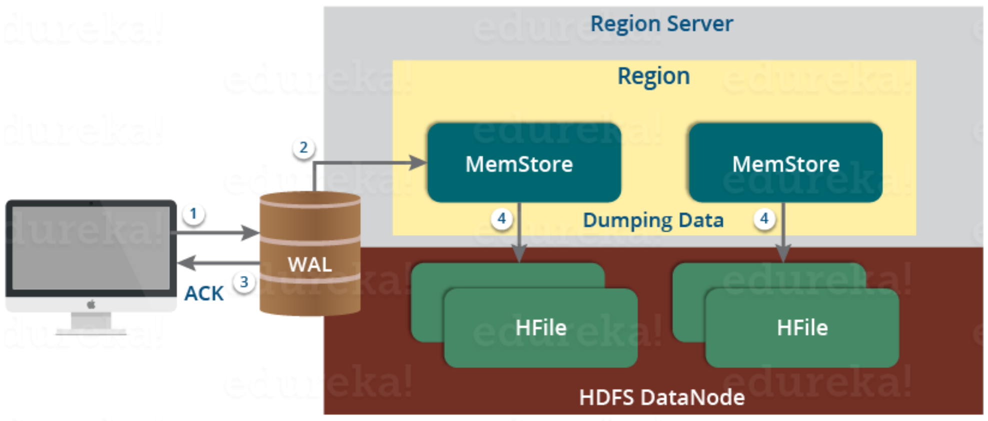

Figure: Write Mechanism in HBase

The write mechanism goes through the following process sequentially (refer to the above image):

Step 1: Whenever the client has a write request, the client writes the data to the WAL (Write Ahead Log).

- The edits are then appended at the end of the WAL file.
- This WAL file is maintained in every Region Server and Region Server uses it to recover data which is not committed to the disk.

Step 2: Once data is written to the WAL, then it is copied to the MemStore.

Step 3: Once the data is placed in MemStore, then the client receives the acknowledgment.

Step 4: When the MemStore reaches the threshold, it dumps or commits the data into a HFile.

HBase Write Mechanism- MemStore

- The MemStore always updates the data stored in it, in a lexicographical order (sequentially in a dictionary manner) as sorted KeyValues. There is one MemStore for each column family, and thus the updates are stored in a sorted manner for each column family.

- When the MemStore reaches the threshold, it dumps all the data into a new HFile in a sorted manner. This HFile is stored in HDFS. HBase contains multiple HFiles for each Column Family.

- Over time, the number of HFile grows as MemStore dumps the data.
- MemStore also saves the last written sequence number, so Master Server and MemStore both knows, that what is committed so far and where to start from. When region starts up, the last sequence number is read, and from that number, new edits start.

the HFile is the main persistent storage in an HBase architecture. At last, all the data is committed to HFile which is the permanent storage of HBase. Hence, let us look at the properties of HFile which makes it faster for search while reading and writing.

HBase Architecture: HBase Write Mechanism- HFile

- The writes are placed sequentially on the disk. Therefore, the movement of the disk's read-write head is very less. This makes write and search mechanism very fast.

- The HFile indexes are loaded in memory whenever an HFile is opened. This helps in finding a record in a single seek.

- The trailer is a pointer which points to the HFile's meta block. It is written at the end of the committed file. It contains information about timestamp and bloom filters.

- Bloom Filter helps in searching key value pairs, it skips the file which does not contain the required rowkey. Timestamp also helps in searching a version of the file, it helps in skipping the data.

HBase Architecture: Read Mechanism

first the client retrieves the location of the Region Server from .META Server if the client does not have it in its cache memory. Then it goes through the sequential steps as follows:

- For reading the data, the scanner first looks for the Row cell in Block cache. Here all the recently read key value pairs are stored.
- If Scanner fails to find the required result, it moves to the MemStore, as we know this is the write cache memory. There, it searches for the most recently written files, which has not been dumped yet in HFile.
- At last, it will use bloom filters and block cache to load the data from HFile.

HBase Architecture: Compaction

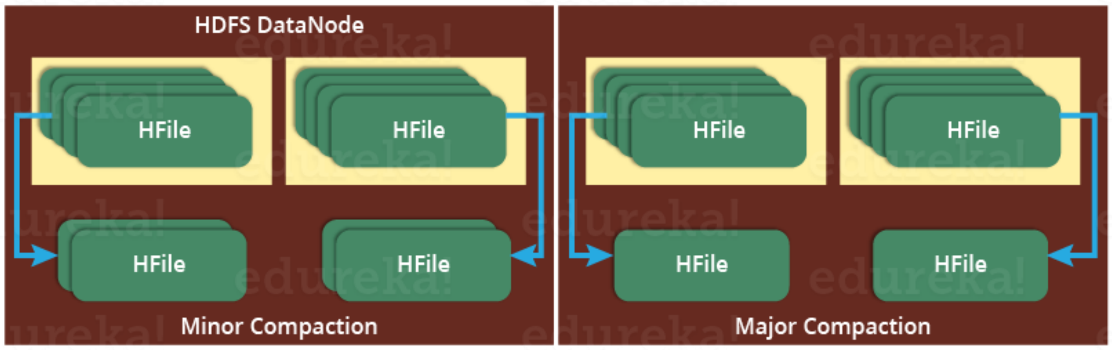

Figure: Compaction in HBase

HBase combines HFiles to reduce the storage and reduce the number of disk seeks needed for a read. This process is called **compaction**. Compaction chooses some HFiles from a region and combines them. There are two types of compaction as you can see in the above image.

1. **Minor Compaction**: HBase automatically picks smaller HFiles and recommits them to bigger HFiles as shown in the above image. This is called Minor Compaction. It performs merge sort for committing smaller HFiles to bigger HFiles. This helps in storage space optimization.

2. **Major Compaction:** As illustrated in the above image, in Major compaction, HBase merges and recommits the smaller HFiles of a region to a new HFile. In this process, the same column families are placed together in the new HFile. It drops deleted and expired cell in this process. It increases read performance.

But during this process, input-output disks and network traffic might get congested. This is known as **write amplification**. So, it is generally scheduled during low peak load timings.

Now another performance optimization process which will be discussed is *Region Split*. This is very important for load balancing.

HBase Architecture: Region Split

The below figure illustrates the Region Split mechanism.

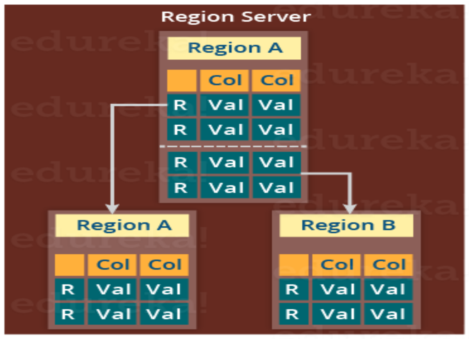

Figure: Region Split in HBase

Whenever a region becomes large, it is divided into two child regions, as shown in the above figure. Each region represents exactly a half of the parent region. Then this split is reported to the HMaster. This is handled by the same Region Server until the HMaster allocates them to a new Region Server for load balancing.

Moving down the line, last but the not least, I will explain you how does HBase recover data after a failure. As we know that **Failure Recovery** is a very important feature of HBase, thus lets us know how HBase recovers data after a failure.

HBase Architecture: 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.

Problem 5: HBase vs RDBMS

Solution:

The following table shows the functional differences between RDBMs and HBase:

When we need more **Online Transaction Processing** (**OLTP**) and the transaction type of processing, RDBMS is easy to go. When we have a huge amount of data (in terabytes and petabytes), we should look towards HBase, which is always better for aggregation on columns and faster processing.

RDBMS	HBase
This supports scale up. In other words, when more disk and memory processing power is needed, we need to upgrade it to a more powerful server.	This supports scale out. In other words, when more disk and memory processing power is needed, we need not upgrade the server. However, we need to add new servers to the cluster.
This uses SQL queries for reading records from tables.	This uses APIs and MapReduce for accessing data from HBase tables.

RDBMS	HBase
This is row oriented, that is, each row is a contiguous unit of page.	This is column oriented, that is, each column is a contiguous unit of page.
The amount of data depends on configuration of server.	The amount of data does not depend on the particular machine but the number of machines.
Its Schema is more restrictive.	Its schema is flexible and less restrictive.
This has ACID support.	There is no built-in support for HBase.
This is suited for structured data.	This is suited to both structured and nonstructural data.
Conventional relational database is mostly centralized.	This is always distributed.
This mostly guarantees transaction integrity.	There is no transaction guaranty in HBase.
This supports JOINs.	This does not support JOINs.
This supports referential integrity.	There is no in-built support for referential integrity.

Following is the difference between **Column-Oriented Data Store** (Ex: HBase) and **Row-Oriented Data Store** (Ex: RDBMS):

Row-oriented data stores	Column-oriented data stores
These are efficient for addition/modification of records	These are efficient for reading data
They read pages containing entire rows	They read only needed columns
These are best for OLTP	These are not so optimized for OLTP yet. Good for OLAP
This serializes all the values in a row together, then the value in the next row, and so on	This serializes all the value of columns together and so on
Row data are stored in contiguous pages in memory or on disk	Columns are stored in pages in memory or on disk