Lecture Notes

NoSQL Databases and Apache HBase

In the IT industry, there is no one-size-fits-all solution to problems. You went through a few use cases in which the relational data model failed. In this module, you learnt about NoSQL databases. Not only SQL (NoSQL) is a non-relational database that can efficiently handle big data in certain cases. As part of this module, you were introduced to Apache HBase, which is a NoSQL database modelled after Google's Bigtable. You also learnt about the features of HBase, its data model, and the programming concepts that are related to it. Further, you gained an understanding of the HBase architecture, read/write operations and some industry use cases of using HBase.

RDBMS and its Drawbacks

Relational database management system (RDBMS) is a solution to all storage needs. It supports Structured Query Language (SQL) to make changes to the database. The data in an RDBMS is stored in the form of tables with rows and columns. With tremendous growth in data today, there is a need for a good database to store and process all this data, and traditional databases fail to meet the optimal storage needs.
The drawbacks of using an RDBMS are as follows:

- RDBMSs cannot accommodate huge volumes of data. To handle large volumes of data, they need to add more storage or more processing units to scale vertically. Horizontal scalability involves adding new servers and spreading the load across them. It is difficult to run read/write operations in parallel in RDBMSs, and so, they are scaled vertically.
- Most of the data generated today is in a semi-structured or an unstructured format. Relational databases cannot store unstructured data because RDBMS is schema-oriented and can store only structured data in the form of tables.
- Big data is generated at a high velocity. Relational databases cannot provide high velocity, i.e., they cannot deliver the optimal speed at which big data is generated and needs to be processed.

NoSQL Databases and its Features

A NoSQL database, where NoSQL stands for 'Not only SQL', is a distributed database. In a NoSQL database, unstructured data is stored across multiple servers (a cluster of machines).

NoSQL databases have the following features:

- NoSQL datastores can store and handle huge volumes of data.
- They provide horizontal scalability, i.e., additional storage can be created easily by adding new nodes to a cluster, without taking the cluster offline.
- They do not follow a strict schema. Hence, they offer a flexible schema, which can be changed dynamically.

- They generally use commodity machines for the servers. This lowers the processing and storage costs per gigabyte compared with that in SQL databases.

The CAP Theorem

The CAP theorem is a tool that is used by system designers to make trade-offs while designing networked shared-data systems.
The three basic characteristics of a distributed database are as follows:

- **Consistency:** This guarantees that all the nodes of the system will return the most recent write to the users.
- **Availability:** This guarantees that every request receives a response with the most recent successful write.
- **Partition tolerance:** A **p**artition-tolerant system continues to work despite network partition.

The CAP theorem states that a distributed database can fulfil at most two of the three guarantees: consistency, availability and partition tolerance.

Since it is not possible to fulfil all three requirements, a combination of two must be chosen for deciding the technology to be used. Based on such combinations, the RDBMS and NoSQL databases can be compared as follows:

- RDBMSs provide availability and consistency. A traditional RDBMS is a single-node system; hence, there is no case of network partitioning.
- NoSQL databases store data in a distributed manner across a cluster of interconnected machines and provide network partitioning tolerance. So, in the case of a network partitioning, one has to make a trade-off between consistency and availability. The two types of NoSQL databases that fulfil two different sets of guarantees are as follows:
    - Consistency and partition tolerance (CP)
    - Availability and partition tolerance (AP)

Depending on the type of application, while designing a networked shared-data system, system designers should make a trade-off between consistency and availability.

Types of NoSQL Databases

The four basic types of NoSQL databases are as follows:

- **Key-value stores:** Data is stored as a key along with its value. A pointer and a unique identifier are associated with every data element. Arbitrary strings are used as keys, and the value could be a document or an image. Key-value datastores consist of large hash tables, which contain keys and values. Some of the popular key-value NoSQL datastores include Cassandra and Redis.
- **Document-based stores:** Each record and all the associated data are stored within a document. The documents stored consist of tagged elements. Some examples of document-based datastores are CouchDB and MongoDB.

- **Column-based stores:** Data is stored by column, not by row. Each storage block contains data from only one column. HBase and Hypertable are popular examples of column-based datastores.
- **Graph-based stores:** A network database that uses edges and nodes to represent and store data. A popular example of this is Neo4J.

Introduction to HBase

Apache HBase is a non-relational distributed data store that is built on the HDFS and is also part of the Hadoop ecosystem. Thus, HBase can leverage all the features that are offered by the HDFS and are available in the Hadoop ecosystem. HBase was released as an open-source implementation of Google's Bigtable.
Google's Bigtable is a high-performance data storage system that is built on the Google File System. It is a distributed storage system that is used to manage data and is designed to scale to a very large size.
HBase has the following features:

- **Distributed storage:** Apache HBase is a distributed, column-oriented database that is built on the HDFS. It allows data to be stored and processed in a distributed manner.
- **Flexible schema:** HBase does not follow any strict schema, i.e., you can add any number of columns dynamically to an HBase table. HBase columns do not have any specific data type, and all the data in HBase is stored in the form of bytes.
- **Sorted:** HBase records are sorted by RowKey. An HBase RowKey must be unique individually, and no two rows can have the same RowKey.
- **Data replication:** It supports the replication of data across a cluster.
- **Faster lookups:** HBase stores data in indexed HDFS files and uses HashMap internally. It also allows random access to the data, which enables a faster lookup.
- **Horizontal scalability:** HBase is horizontally scalable; this means that if the clusters require more resources, then HBase can scale up according to the need; it can horizontally scale up to thousands of commodity servers.

Data Model of HBase

The HBase data model is constituted of various logical components that are as follows:

- **Tables:** HBase tables are collections of rows and columns. Basic CRUD operations – create, read, update and delete – can be performed on tables using HBase shell commands or the application program interface (API).
- **Rows:** Rows are collections of column families and are sorted based on the RowKey of a table. A RowKey is used to uniquely identify a record. Its design optimises the scan, as related rows or rows that are read together are stored together.
- **Column families:** These are collections of columns. Column quantifiers integrated with a column family name are used to identify a single column. Each row in an HBase table can have multiple column families, and one or more columns can be associated with each column family.

- **Version:** The data in an HBase table is stored in a cell. A cell is a combination of a Rowkey, a column family and a column qualifier. It contains a value and a timestamp. The data stored in a cell can have multiple versions. Each version is identified using its own timestamp.

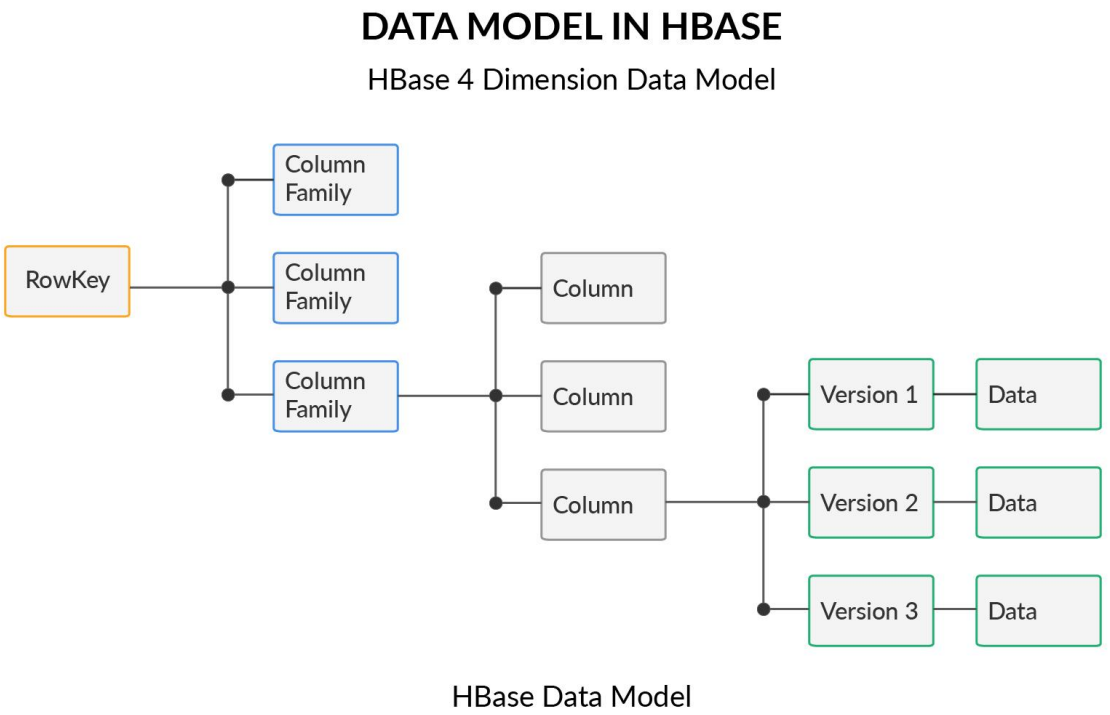

Figure 01: HBase Data Model

HBase Shell Commands

The three basic types of shell commands are as follows:

- **General commands**
  - **Status:** This command shows the status of the cluster. It shows the number of servers present in the cluster, the active server count and the average load value. It can be 'simple', 'summary' or 'detailed'. The default status is 'summary'.

    Syntax: `status`
  - **Version:** This command displays the currently used version of HBase.

    Syntax: `version`
  - **Table Help:** This command provides a guide on how to use table-referenced commands. It also provides the usage and syntax of various HBase shell commands.

    Syntax: `table_help`

- **Table management commands**
  - **Create:** This command allows you to create a new table in HBase with a specified name.

    Syntax: `create '<table_name>', '<column_family_name>'`
  - **List:** This command displays all the tables that are present in HBase. It is used to check whether your table is created or not.

    Syntax: `list`
  - **Describe:** This command provides information about the column families in the mentioned table.

    Syntax: `describe '<table name>'`
  - **Disable:** This command disables the mentioned table. Tables need to be disabled before they are deleted or dropped.

    Syntax: `disable '<tablename>'`
  - **Enable:** This command enables the mentioned table. This command is used to retrieve the disabled table to its previous state.

    Syntax: `enable '<tablename>'`
  - **Is Enabled:** This command verifies whether the mentioned table is enabled or not.

    Syntax: `is_enabled '<table_name>'`
  - **Is Disabled:** This command verifies whether the mentioned table is disabled or not.

    Syntax: `is_disabled '<table_name>'`
  - **Exists:** This command verifies whether a given table is present in the HBase storage or not.

    Syntax: `exists '<table_name>'`
  - **Alter:** This command is used to alter the column family schema. It can be used for operations such as adding column families and updating the version number of column families. It can also be used to delete a column family by applying the delete method to it.

    Syntax:

    ```
    alter '<tablename>', NAME=>'<column familyname>', VERSIONS=><Number>
    ```

    ```
    alter '<tablename>', 'delete'=> '<column familyname>'
    ```

- **Drop:** This command drops the mentioned table. Before dropping, the table should be disabled first.

  Syntax: `drop '<table name>'`

- **Data manipulation commands**
  - **Put:** This command is used to add a cell value in the mentioned table. A single put command can add a single cell value to the table.

    Syntax: `put '<table_name>', '<row_key>', '<column_value>', '<value>'`

  - **Get:** This command is used to fetch data from the HBase table.

    Syntax: `get '<table_name>', '<row_key>', {'<Additional Parameters>'}`

  - **Count:** This command will return the number of rows present in the table.

    Syntax: `count '<table_name>'`

  - **Delete:** This command is used to delete the cell value in the mentioned table in the specified row or column.

    Syntax:

    `delete '<table_name>', '<row_key>', '<column_value>', <timestamp_value>`

  - **Scan:** This command is used to view all the contents of the table created.

    Syntax: `scan '<table_name>' {Optional parameters}` The optional parameters in the syntax include TIMERANGE, FILTER, TIMESTAMP, LIMIT, MAXLENGTH, COLUMNS, CACHE, STARTROW and STOPROW.

  - **Get data based on filters:** In HBase, fetching data based on a filtering condition is achieved using filters. In HBase, filters are like Java methods that take two input parameters: a logical operator and a comparator. The logical operator specifies the type of the test, such as 'equal to' and 'less than'. The comparator is the number/value against which you want to compare your record. Some commonly used filter functions are as follows:
    - **ValueFilter:** The ValueFilter takes a comparison operator and a comparator as parameters. It compares each value with the comparator using the comparison operator. If the check is true, then the result is displayed on the console.

      Syntax: `"ValueFilter(<compareOp>, '<value_comparator>')"`

    - **QualifierFilter:** The QualifierFilter also takes two parameters: comparison operator and comparator. Each qualifier name is compared with the comparator using the compare operator, and if the comparison is true, then it returns the key values in that column.

      Syntax: `"QualifierFilter(<compareOp>,'<qualifier_comparator>')"`

- **FamilyFilter:** The FamilyFilter is used to fetch key values for a specified column family.

  Syntax: `"FamilyFilter(<compareOp>, '<family_comparator>')"`

- **Delete all:** This command deletes all the cells in the mentioned row.

  Syntax: `deleteall '<tablename>', '<rowkey>'`

- **Truncate:** This command deletes all the data from the table. This command performs three tasks: disabling the table, dropping it and then recreating it.

  Syntax: `truncate '<table_name>'`

HBase Architecture

The HBase architecture works on the concept of master-slave architecture. It consists of three main servers: HMaster, Region Server and ZooKeeper. Each server manages different components of the HBase architecture.

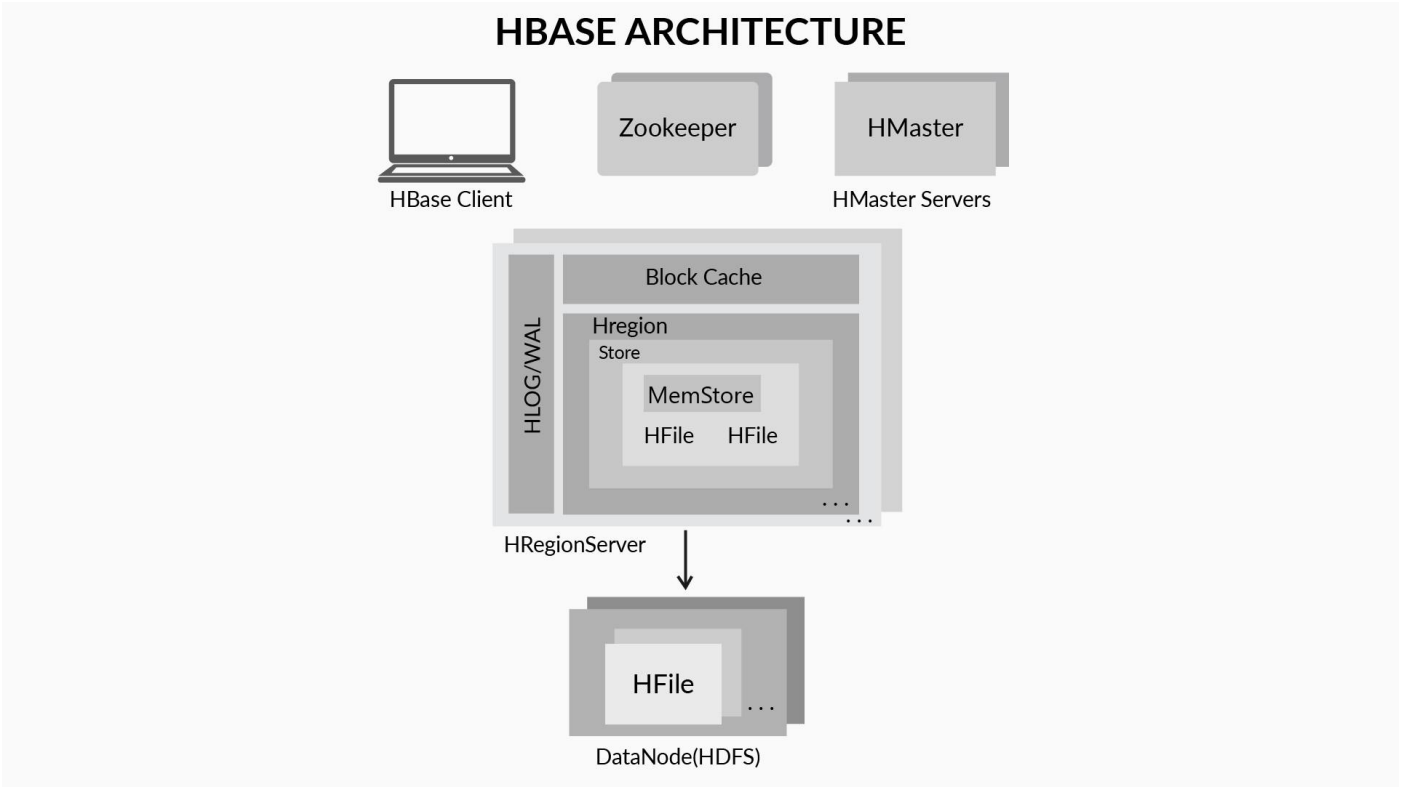

Figure 02: HBase Architecture

Region Servers

- The region server manages regions. Multiple region servers are present in a cluster.
- Each region server contains the following:

- A Write Ahead Log (WAL): It is also known as the HLog and stores new or updated data that has not been written to permanent storage (similar to HFiles in the HDFS). It can be used for recovery in the case of a region server failure. A region server stores the WAL file in the HDFS.
- A block cache: It is a read cache, which frequently stores the read data from the HDFS.
- A region: It is assigned a region of the table. Each region contains one store per column family. The store consists of a MemStore, and therefore, each column family has one MemStore.

- A client interacts directly with the region server to perform read/write operations. This region server assigns the request to the specific region where the data resides.

Regions

An HBase table is a collection of rows and columns. When these tables are distributed horizontally with a fixed size, each portion is called a region.

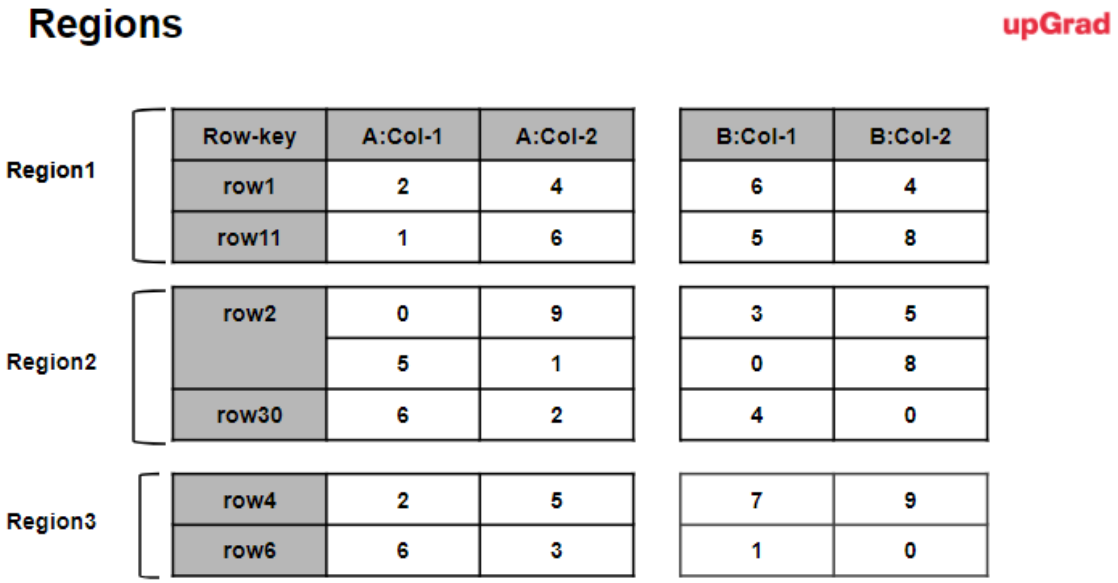

Figure 03: Regions

A region server stores multiple regions. Every region is further allocated with one store per column family. A store in a region server contains a MemStore and multiple HFiles. A MemStore is an in-memory write buffer, which stores new or recently updated data that has not yet been written to the HDFS. The data stored in the MemStore gets flushed to the HDFS in a new HFile at regular intervals or based on the MemStore size. The actual data resides in the HFiles in the form of sorted key-value pairs.

DataNode (HDFS)

HBase stores the following two types of files in the DataNode of HDFS:

1. Write Ahead Log (WAL)
   - There is one WAL file per region server that stores the MemStore data.
   - The WAL is maintained by region servers and is stored in the HDFS.
   - Every edit is appended in the WAL file.
   - The WAL is replayed by region servers in the case of a region server crash.
   - The log file limits the write throughput of the region server because an HDFS pipeline write is involved in appending the data to the WAL file with every write request.

- When the WAL file grows to approximately 95% of the HDFS block size, a new WAL file is created, and all the additional changes are now appended to this new WAL file. This process is referred to as the Rolling of WAL files. Please note that once the rolling has accrued, no further changes are made to the old WAL files.
- The WAL file block size can be configured using the parameter 'hbase.regionserver.hlog.blocksize'. The WAL file is also rolled periodically based on the configured interval "hbase.regionserver.logroll.period", with the default time being one hour. (hbase-site.xml defines these properties.)

2. HFile
    - HFiles store the HBase table's data as sorted key-value pairs.
    - They are immutable, which means that once they are written, they cannot be modified.
    - They are large in size depending on the MemStore flush size (before compaction).
    - HFiles further store the data as a set of blocks, which helps in reading only the block that contains the data of interest and not the complete HFile.
    - The HFile BLOCKSIZE is configured from the column family descriptor (alter command).
    - The default block size in an HFile is 64 KB.
    - The data block index in an HFile is used to locate the data block of interest. It contains the key range stored in each data block.

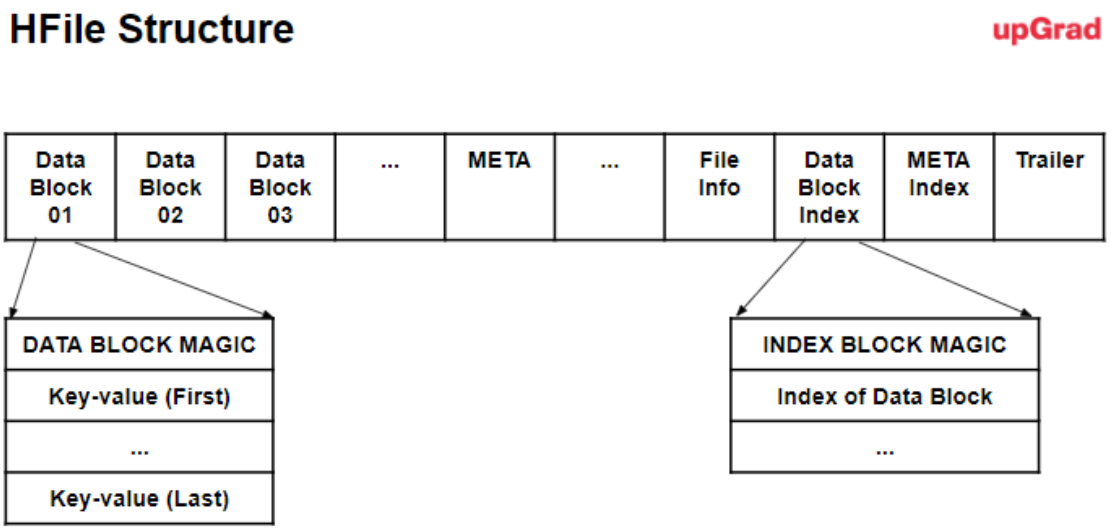

Figure 04: HFile Structure

HMaster

- It acts as the Master server and manages multiple region servers.
- An HBase cluster may have one or more master nodes, but only one HMaster is active at a given time. This active node is responsible for the following:
    - Performing admin functionalities: The HMaster performs DDL operations such as creating and deleting tables.
    - Coordinating with the region server: The HMaster assigns regions to the region servers. In the case of load balancing or recovery, it reassigns regions to region servers in the following ways:
        - Load balancing: It occurs when an HMaster assigns the load from a server with a high load to a less-occupied region server.

- Recovery: The HMaster also handles region server failures by reassigning the load of the failed region servers to non-failing servers.

ZooKeeper

- ZooKeeper is a distributed, open source coordinating service for distributed applications.
- It is a cluster of nodes and maintains configuration information, naming, provides distributed synchronisation and group services, etc.
- It maintains the live server state in the cluster by receiving heartbeat messages from all the HMasters and region servers.
- It also provides server failure notifications so that recovery measures can be taken.
- It also stores the location of the metatable.

Read Operation in HBase

Metatable

A metatable stores the locations of the regions and region servers. It is used to identify the regions in which the range of a key value pair is stored. The location of the metatable is stored in the ZooKeeper and is itself stored in one of the region servers. The metatable has two components, a key and a value. The key is the unique identifier in the metatable and is comprised of the table name, the start row key and the region ID.

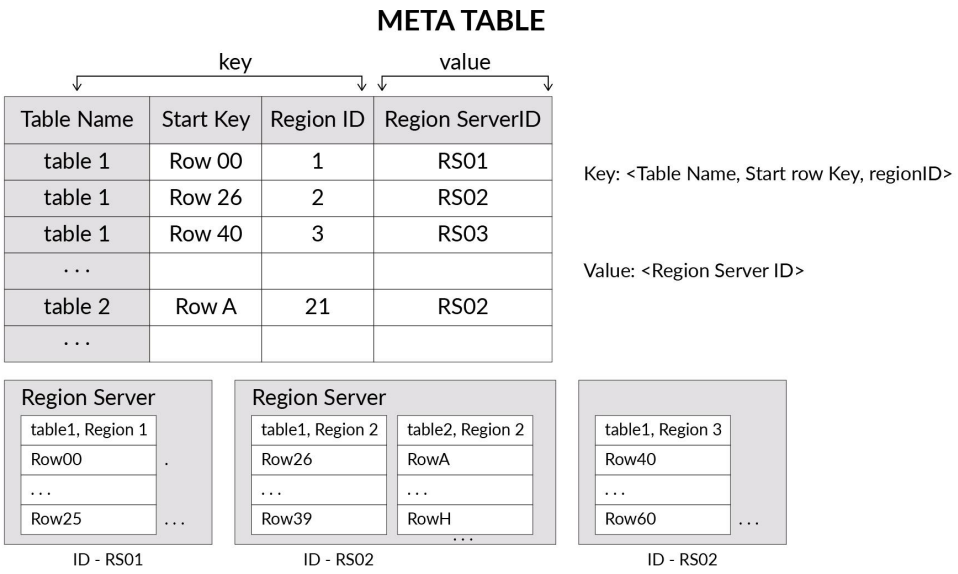

Figure 05: Metatable Structure

Common Steps

Whether it is a read operation or a write operation, the client should know where the region is present in the region server in which the key value pair is stored.

Fetching the region server location

To find the region, clients should follow the basic initial steps given below before performing a read/write operation:

- Step 1: The client contacts the ZooKeeper to know the location of the metatable.
- Step 2: The client queries the metatable to know the location of the region server that contains the region along with the key-value pair that the client is looking for.
- Note: In case there is any change in the locations of the region servers in the metatable owing to load balancing or any other factor, the cache of the client needs to be updated.
- Step 3: Now, the client saves the information of the region server and the location of the metatable for further interaction, if any.
- Step 4: The client can now communicate with the region server that was identified earlier. The region server that is identified will further assign the read/write request to the region for the operation to be executed.

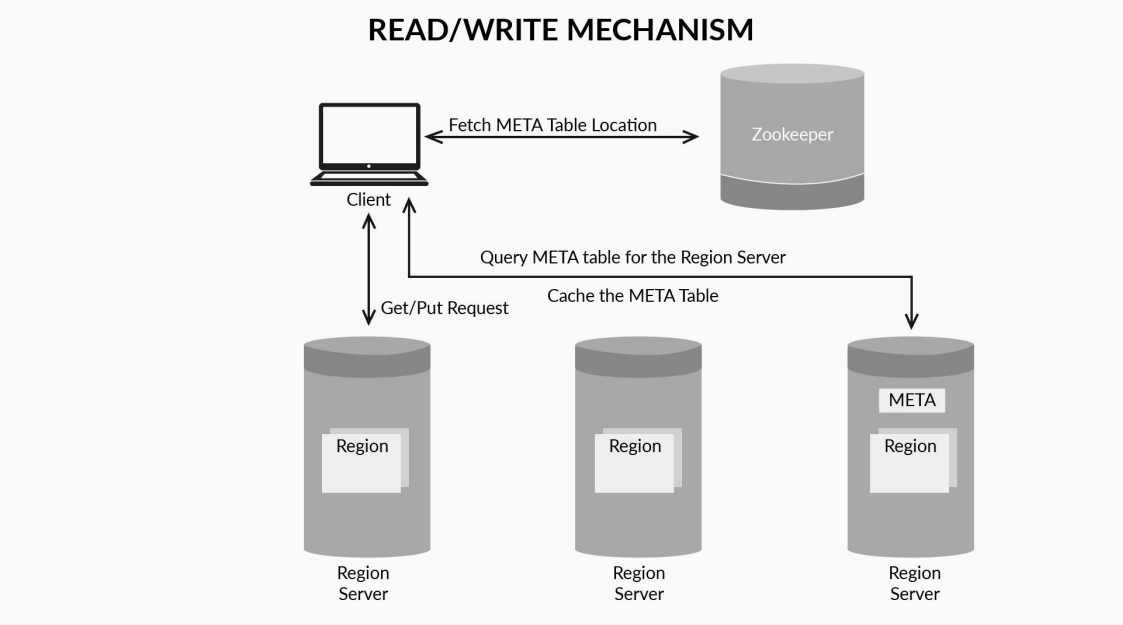

Figure 06: Fetch Server Location

HBase Read Operation

Now that the client knows the location of the region server where the matching region is present, it will send a read request to the region server.

The steps to perform a read operation in HBase are as follows:

- **Step 1:** The region server checks the block cache. The block cache of the region server contains frequently or recently accessed data.
- **Step 2:** The MemStore is checked if the data is not found in the block cache.
- **Step 3:** If the MemStore does not contain the required data, then the region servers use the Bloom filter to find the HFile that contains the required key-value pair. Each HFile has a Bloom filter that is used to check whether the key-value pair is present in the HFile or not. Once the HFile is identified,

the data block index is used to obtain the data block that contains the required key-value pair. As the data is stored in a sorted manner, a Binary search is used to retrieve the required value.

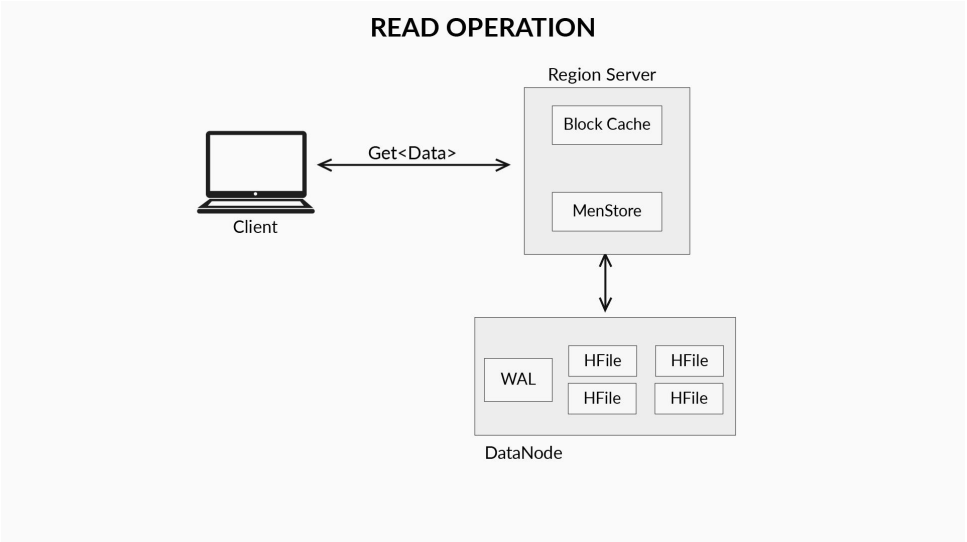

Figure 07: Read Operation in HBase

Write Operation in HBase

Whenever a client wants to perform a write operation, a put request is assigned to the region server, which further allocates the request to the corresponding region. This is achieved by the following steps:

- **Step 1:** The data is first written to the Write Ahead Log (WAL). The WAL stores the new or updated data, which is not yet written to the HDFS.
- **Step 2:** After the data is written to the WAL, it is further placed in the MemStore, which is an in-memory write buffer. When the MemStore has enough data, the data is flushed to the HDFS, which forms a new HFile.
- **Step 3:** Finally, an acknowledgement is sent to the client.

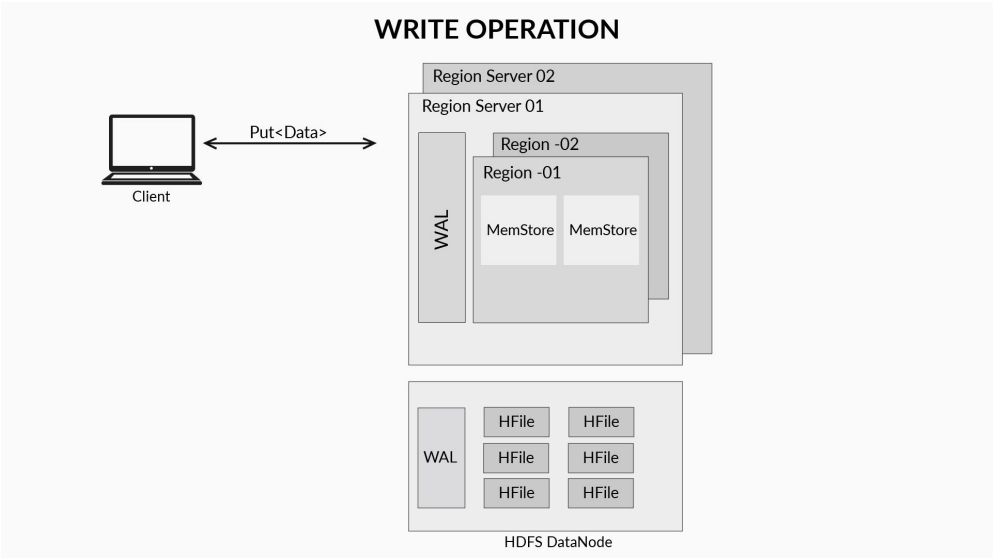

Figure 08: Write Operation in HBase

Compactions

Heavy incoming writes lead to the following two major problems:

- The read efficiency becomes low because heavy writes increase the number of disk seeks that are needed for the read operation.
- There is an increase in redundant and inconsistent data due to an increase in the number of HFiles.

The process of combining small HFiles to form large HFiles to reduce the number of disk seeks is known as compaction. The two types of compactions in HBase are as follows:

- **Minor compaction:** It is the process of combining smaller HFiles to form larger HFiles.
- **Major compaction:** It is the process of combining all the store files of a region to form a single store file. In case the size of the newer store file is greater than the required size, the region in which it is present splits into two separate regions. This splitting is known as auto-sharding. Major compaction removes all the deleted and expired data from HFiles.

The key difference between the two is that during major compaction, the values with tombstone markers (deleted data) and expired values (whose TTL is over) are removed from the HFiles.

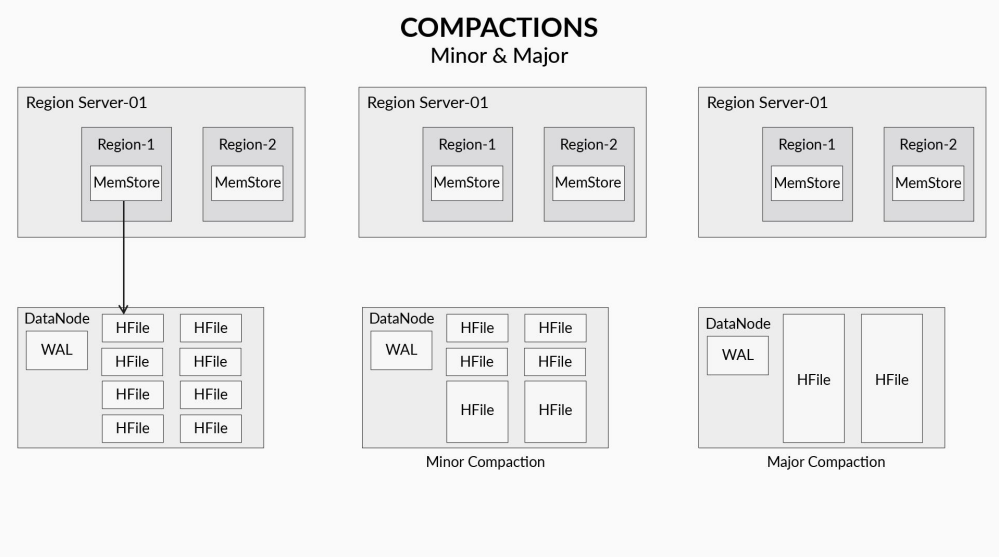

Figure 09: Compactions

Delete Operation in HBase

The delete operation in HBase is a special type of Update operation. The values for which a delete request is submitted are marked with a tombstone marker. In case a client tries to read a deleted value or a value with a tombstone marker, a NULL is returned, which tells the client that the value has been deleted.

The reason for this is that HFiles cannot be updated or modified, as we discussed earlier. All the values with a tombstone marker are deleted when a major compaction occurs, which takes place every 24 hours. The delete markers are of the following three types:

- Version delete marker: It is used to mark a specific version of a column.
- Column delete marker: It is used to mark all the versions of a column.

- Family delete marker: It marks all the versions of all the columns of a column family.

HBase Schema Design

Row-key design

For faster lookups, the rows of an HBase table are sorted lexicographically. This optimises the scan because related rows or rows that can be read together are stored together. As the rowkey design is responsible for better lookups, a poor design of rowkey can lead to hotspotting.

When a large amount of client traffic is directed towards a single node or only towards a few nodes of a cluster, it is known as hotspotting. The incoming traffic can be of any type: reads, writes and others. This huge amount of traffic on a single machine can lead to the degradation of the machine's performance. Hence, it is mandatory to design the schema in such a way that the whole cluster is evenly and properly utilised.

To avoid hotspotting, some of the following techniques should be used:

- Salting: Adding a random prefix to the start of a rowkey is known as salting. This causes the rows to sort differently than they would have if you provided it with a normal sequence. This can be helpful if you have some specific rowkey patterns.
- Hashing: A one-way hash can be used rather than a random assignment. A given row will be salted with the same prefix. A predetermined hash will allow the client to reconstruct the whole rowkey and also use the get operation to retrieve the row normally.
- Reversing the rowkey: Another way of spreading the load across various clusters and preventing hotspotting is to simply reverse the fixed-length rowkey, i.e., the least significant digit becomes the first one. This randomises the rows across various region servers.

Minimise row and column sizes

In HBase, a cell value can be accessed using a unique key, i.e., <rowkey, column name, timestamp>. Now, if your row and column names are significantly larger than the size of a cell value, then the random access will occupy large chunks of RAM because of large cell value coordinates.

To avoid this scenario, the following steps should be taken:

- The names of column families should be kept as short as possible.
- The names of attributes should also be kept short.
- The length of the rowkey should be kept such that it is short and useful for accessing data easily.

Reverse Timestamps

One of the most common issues with database processing is to find the most recent version of a value. A reverse timestamp as part of the key can be used in this case.

The rowkey can be designed as [key][reverse_timestamp].

HBase Use Cases

Some of the industries in which HBase provides better solutions than any other database are as follows:

- Telecom: The telecom industry needs database solutions to store large amounts of call recordings and to provide real-time access to these call records, customer information and the billing costs related to them. Compared with any other database, HBase enables random and faster lookups in real-time big data.
- Banking: Massive amounts of data are generated in the banking industry, which require analytics solutions for detecting frauds of any type. Integrated with other components of the Hadoop ecosystem, HBase can provide analytics solutions and massive storage ability in this industry.
- Finance: In the finance industry, HBase is used to store financial data for trading and analysing risk factors.
- Healthcare: The healthcare industry faces a great need to store a large amount of patient data; for example, disease history and analysis of diseases, to find treatment solutions. In this case, HBase can be used to store such massive data for running analytics to find better solutions.
- Sports: HBase can be used to store match history, player information, and the viewing history and preferences of the viewers in the field of sports.
- E-commerce: HBase can be used to store customer search history and preferences, customer storing logs, customer information, merchant details, etc. It can also be used to analyse the customer base for advertisements to improve business.

Use Cases of HBase

Now that you are familiar with some industries in which HBase is used, let's take a look at the following use cases of HBase:

- Yahoo uses HBase for content personalisation and a web cache for searching.
- eBay uses HBase for improving its search engine.
- Pinterest uses HBase for relevant search.
- Bloomberg uses HBase for storing time series data.
- Facebook used HBase for their messaging application.

Advantages of HBase

The advantages of using HBase are as follows:

- As you already know, HBase is built on theHDFS, which is a distributed file system. This gives HBase the ability to store large amounts of data and perform analytics in a short period of time. HBase provides a cost-effective solution in case the data size is in the petabyte range, as it uses commodity hardware.
- HBase is schema-less, and therefore, HBase tables can be added, updated or deleted dynamically.
- HBase provides read/write consistency because it enables random lookups, unlike the HDFS.
- Data reading and processing take less time in HBase compared with that in traditional databases because it provides faster lookups.
- Features such as the Bloom filter (testing the presence of an element in a set of data) and the block cache (storing frequent and recent data), which are taken from the Google Bigtable, can be used for query optimisation.
- Newer nodes can be added in case the application grows in size.

Disadvantages of HBase

The disadvantages of using HBase are as follows:

- HBase is quite resource-intensive, as it enables random and faster lookups on the HDFS.
- It does not provide any type of built-in authentication; it allows read and write access to everyone on every table.
- HBase allows only one default row key for sorting, whereas RDBMSs provide multiple keys.
- It has a single point of failure, i.e., if the active HMaster fails, then it may take some time to have another HMaster in place. So, to have an always-available system, one should opt for Cassandra.
- HBase does not require any specific query language to access the data from the datastore. To achieve querying in HBase, it needs to be integrated with other technologies, such as Hive, which can lead to latency. To overcome this limitation, one can use Cassandra, as it has its own query language, which, somehow, resembles the traditional query language and can be used easily.
- The query model of HBase uses key-value pairs; it does not provide various filters, aggregate functions, comparison, etc. On the other hand, the expressive query language model of MongoDB provides powerful query operators, which can handle advanced analytics workloads.

Disclaimer: *All content and material on the upGrad website is copyrighted material, belonging to either upGrad or its bona fide contributors, and is purely for the dissemination of education. You are permitted to access, print, and download extracts from this site purely for your own education only and on the following basis:*

- *You can download this document from the website for self-use only.*
- *Any copies of this document, in part or full, saved to disc or to any other storage medium, may be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.*
- *Any further dissemination, distribution, reproduction, and copying of the content of the document herein, or the uploading thereof on other websites, or use of the content for any other commercial/unauthorized purposes in any way that could infringe the intellectual property rights of upGrad or its contributors is strictly prohibited.*
- *No graphics, images, or photographs from any accompanying text in this document will be used separately for unauthorized purposes.*
- *No material in this document will be modified, adapted, or altered in any way.*
- *No part of this document or upGrad content may be reproduced or stored on any other website or included in any public or private electronic retrieval system or service without upGrad's prior written permission.*
- *Any rights not expressly granted in these terms are reserved.*