Chapter 3: HDFS, Hive and HiveQL, HBase (part 3)

By Ms. Tina D’abreo

Content

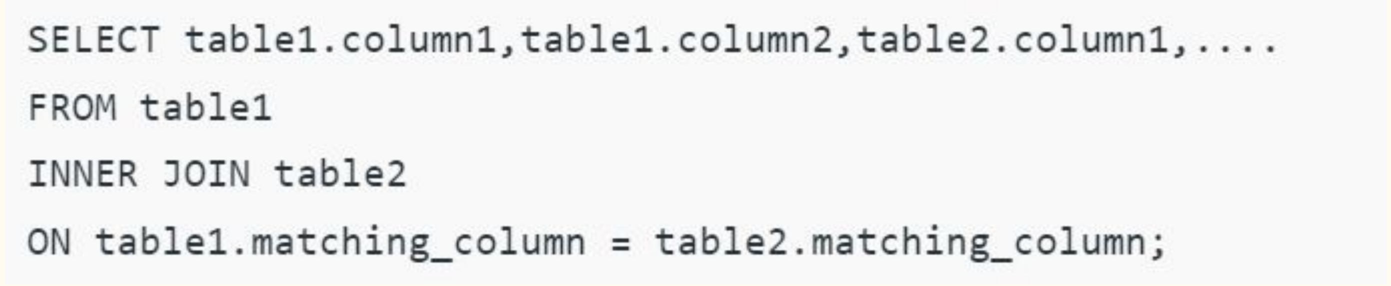
- HDFS-Overview, Installation and Shell, Java API
- Hive Architecture and Installation, Comparison with Traditional Database,
- HiveQL Querying Data, Sorting And Aggregating,
- **Map Reduce Scripts, Joins & Sub queries**
- **HBase concepts, Advanced Usage, Schema Design**, **Advance Indexing**, PIGGrunt – pig data model – Pig Latin – developing and testing Pig Latin scripts
- Zookeeper , how it helps in monitoring a cluster
- Build Applications with Zookeeper and HBase

Map Reduce Scripts in Hive

- Similar to any other scripting language, Hive scripts are used to execute a set of Hive commands collectively.
- Hive scripting helps us to reduce the time and effort invested in writing and executing the individual commands manually.
- Hive scripting is supported in Hive 0.10.0 or higher versions of Hive.

HiveQL- Joins

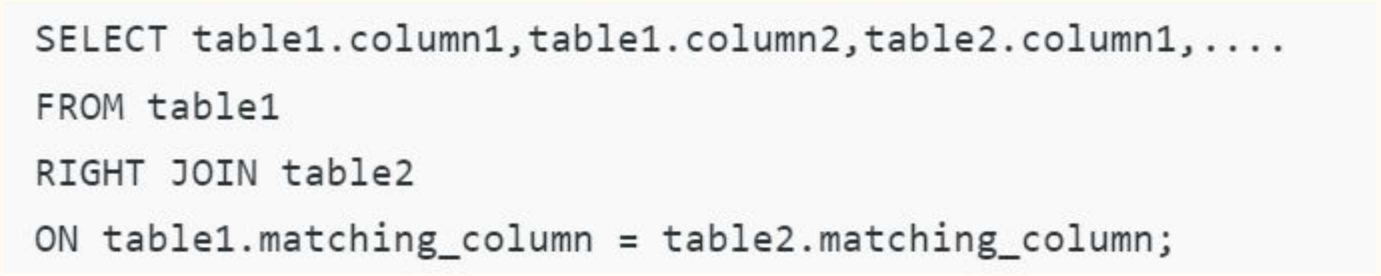
- HiveQL Join clause is used to combine the data of two or more tables based on a related column between them
  - ***Inner Join***
    - The Records common to both tables will be retrieved by this Inner Join.
    - Syntax:



```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

HiveQL- Joins

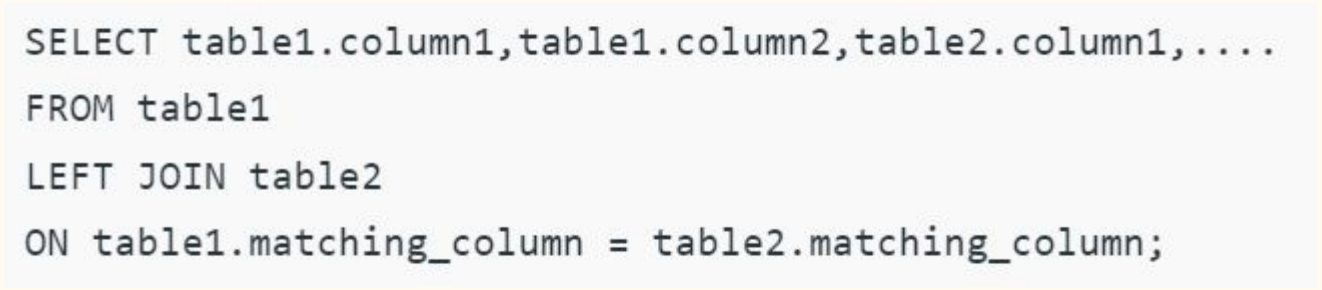
- HiveQL Join clause is used to combine the data of two or more tables based on a related column between them
  - ***Right Outer Join***
    - Returns all the rows from the Right table even though there are no matches in the left table.
    - Syntax:



```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```

HiveQL- Joins

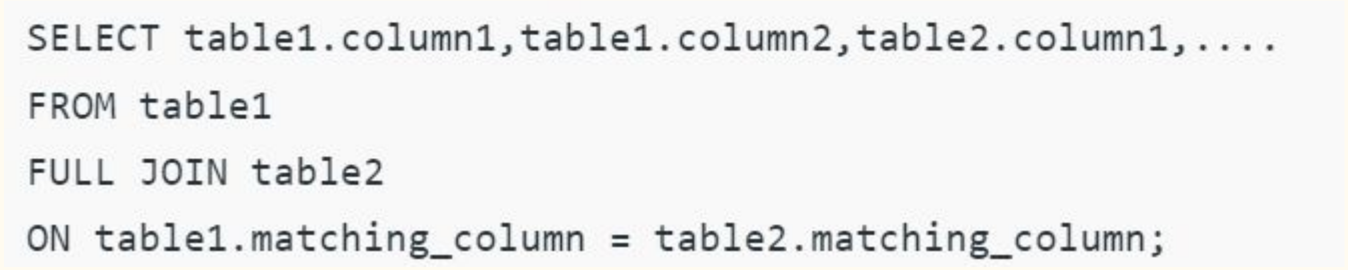
- HiveQL Join clause is used to combine the data of two or more tables based on a related column between them
  - ***Left Outer Join***
    - Returns all the rows from the left table even though there are no matches in the right table.
    - Syntax:



```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```

HiveQL- Joins

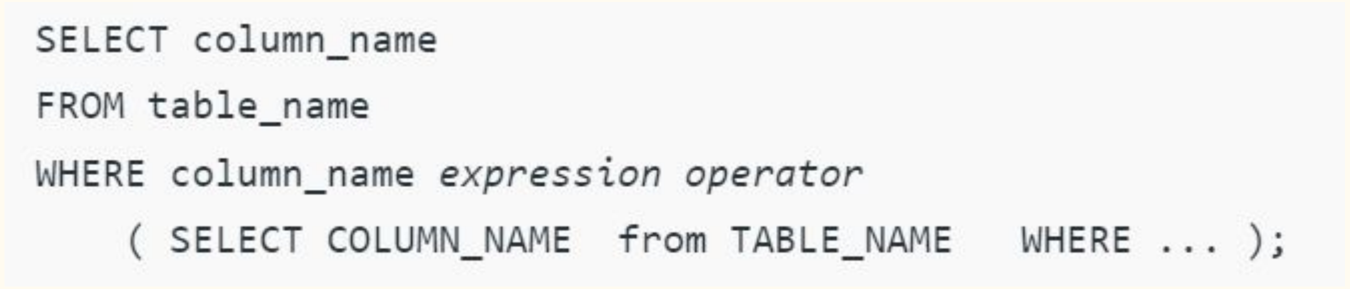
- HiveQL Join clause is used to combine the data of two or more tables based on a related column between them
  - ***Full Outer Join***
    - It combines records of both the tables based on the JOIN Condition given in the query. It returns all the records from both tables and fills in NULL Values for the columns missing values matched on either side.



```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
```

Hive QL - Subqueries

- A Query present within a Query is known as a subquery. The main query will depend on the values returned by the subqueries.
- Subqueries can be classified into two types:
    - Subqueries in FROM clause
    - Subqueries in WHERE clause
- When to use:
    - To get a particular value combined from two column values from different tables
    - Dependency of one table values on other tables
    - Comparative checking of one column values from other tables



```
SELECT column_name
FROM table_name
WHERE column_name expression operator
    ( SELECT COLUMN_NAME  from TABLE_NAME   WHERE ... );
```

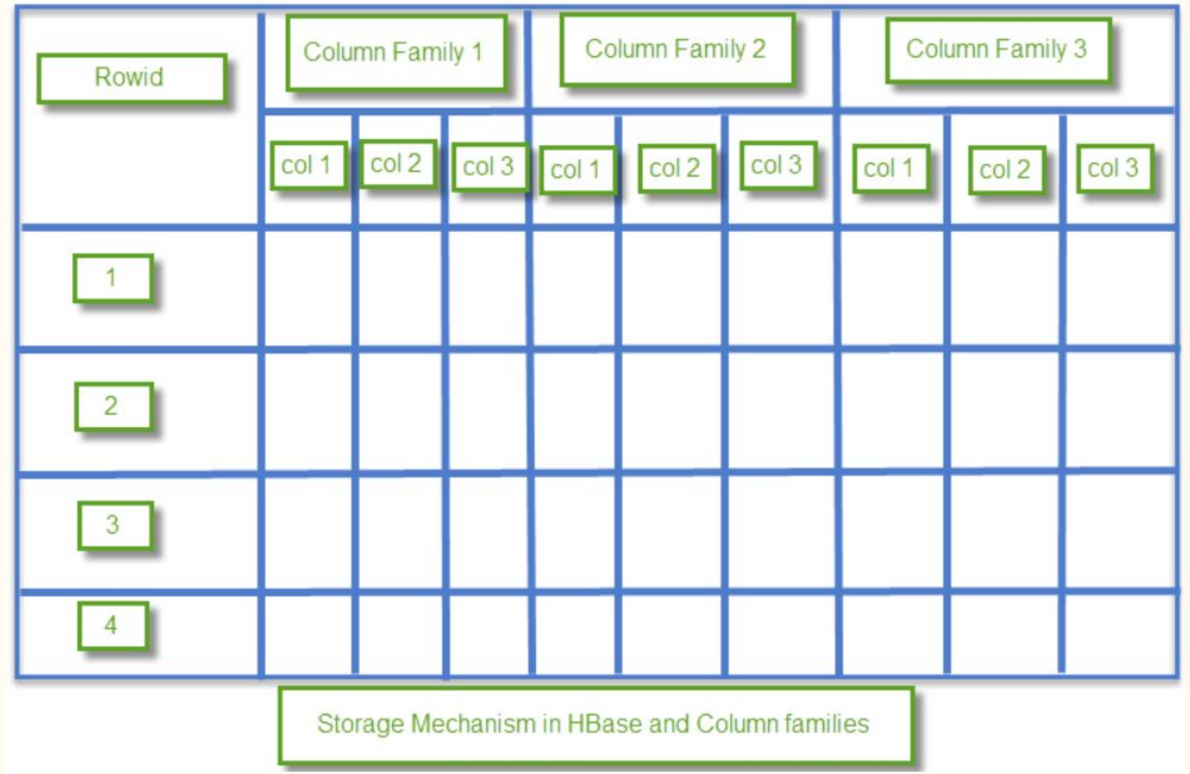
HBase

- It is an open source, distributed database developed by Apache software foundation written in Java. HBase runs on top of HDFS (Hadoop Distributed File System).
- It can store massive amounts of data from terabytes to petabytes.
- HBase is a distributed column-oriented database built on top of the Hadoop file system.
- It is an open-source project and is horizontally scalable.
- HBase is a data model that is similar to Google’s big table designed to provide quick random access to huge amounts of structured data.
- It leverages the fault tolerance provided by the Hadoop File System (HDFS).
- It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.
- One can store the data in HDFS either directly or through HBase.
- Data consumer reads/accesses the data in HDFS randomly using HBase.
- It is column oriented and horizontally scalable.

HBase Data Model

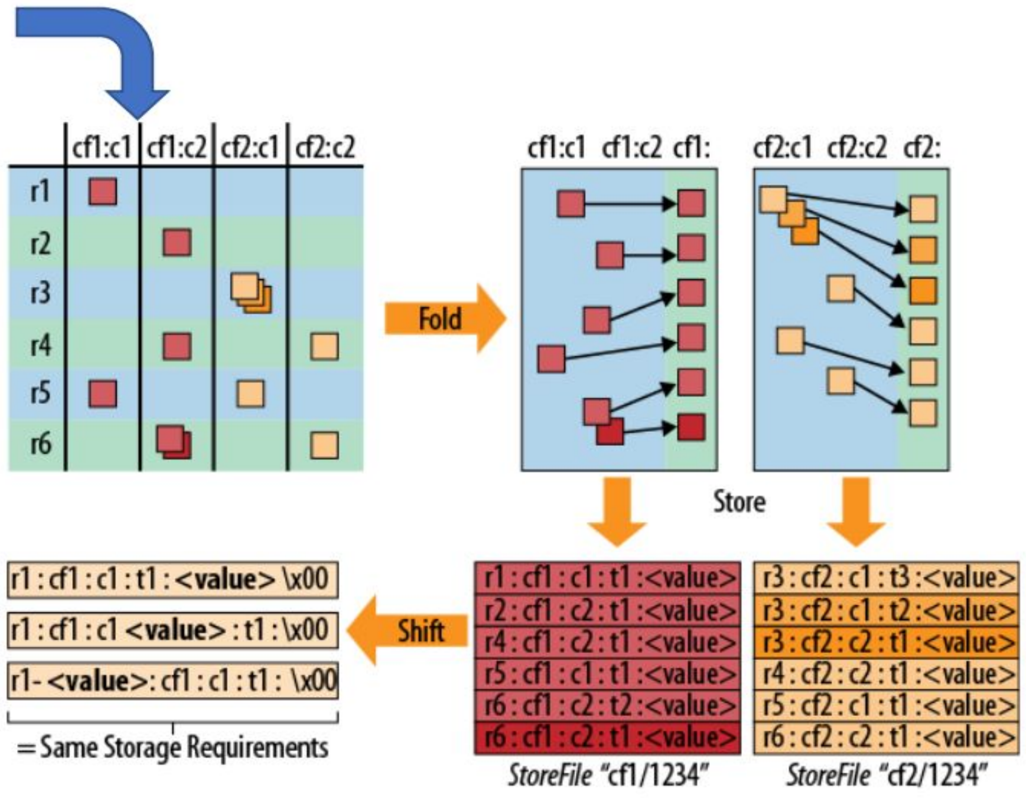
HBase Data Model consists of following elements,

- Set of tables
- Each table with column families and rows
- Each table must have an element defined as Primary Key.
- Row key acts as a Primary key in HBase.
- Any access to HBase tables uses this Primary Key
- Each column present in HBase denotes attribute corresponding to object

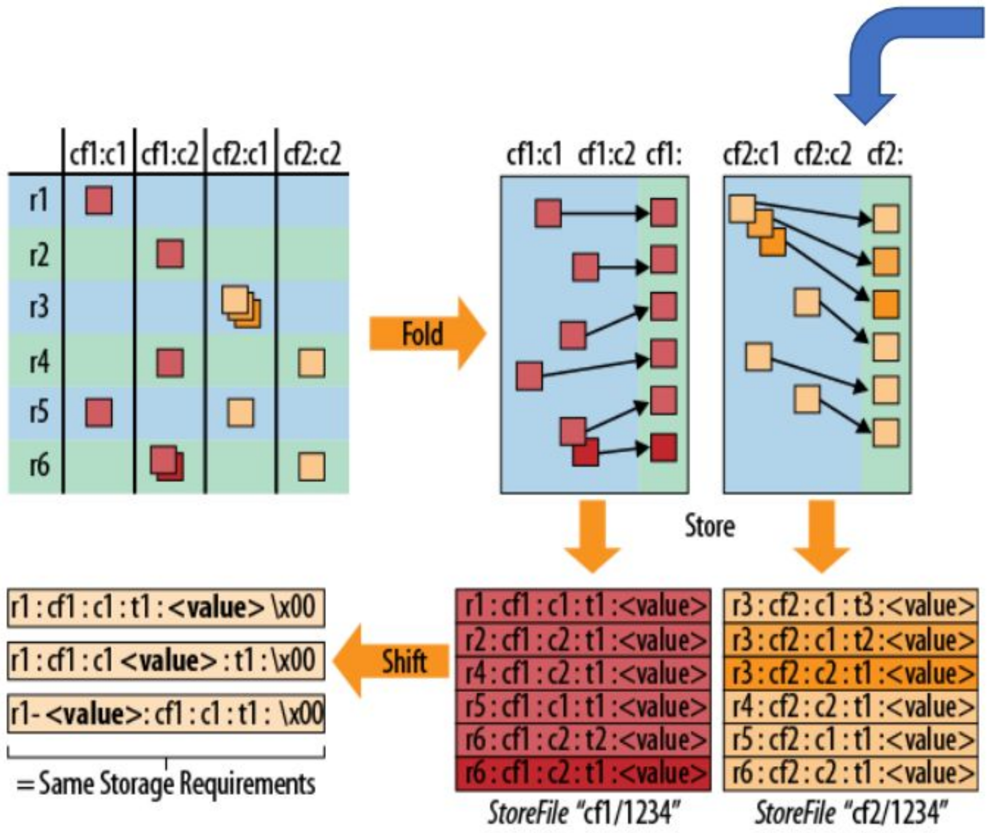


HBase Data Model - storage

The top-left part of the figure shows the logical layout of your data: you have rows and columns. The columns are the typical HBase combination of a column family name and a column qualifier, forming the column key. The rows also have a row key so that you can address all columns in one logical row.

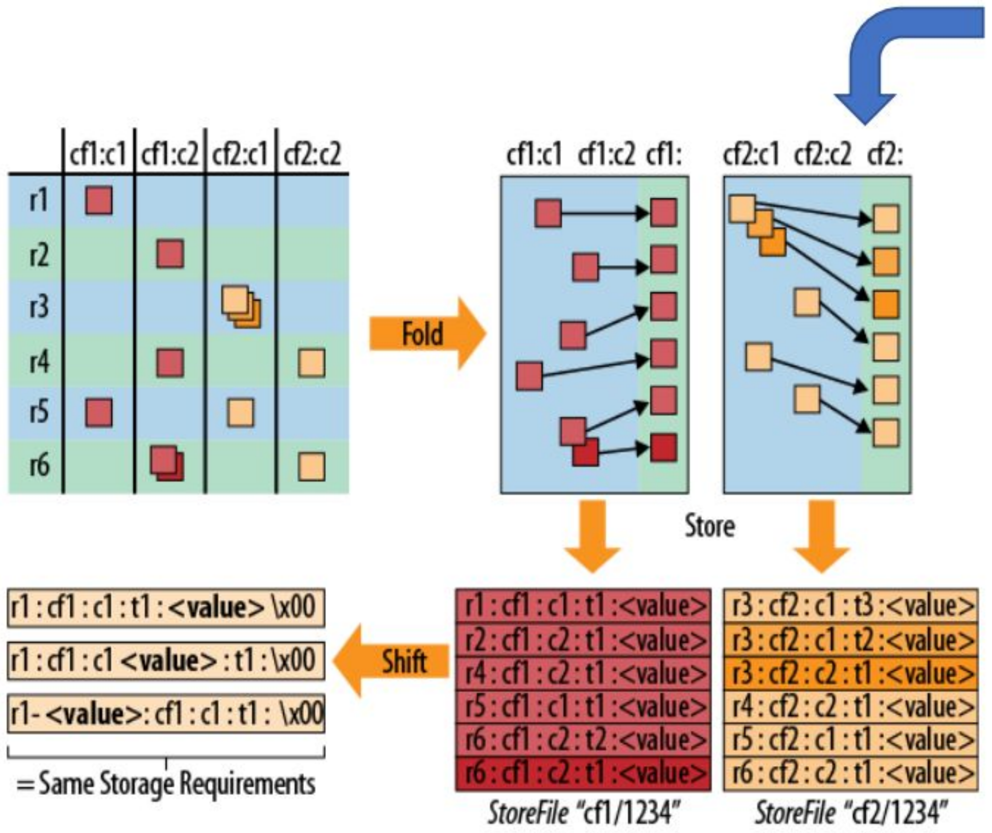

cf1:c1
cf1:c2
cf2:c1
cf2:c2
r1
r2
r3
r4
r5
r6
Fold
cf1:c1 cf1:c2 cf1:
cf2:c1 cf2:c2 cf2:
Store
r1 : cf1 : c1 : t1 :<value>
r2 : cf1 : c2 : t1 :<value>
r4 : cf1 : c2 : t1 :<value>
r5 : cf1 : c1 : t1 :<value>
r6 : cf1 : c2 : t2 :<value>
r6 : cf1 : c2 : t1 :<value>
StoreFile "cf1/1234"
r3 : cf2 : c1 : t3 :<value>
r3 : cf2 : c1 : t2 :<value>
r3 : cf2 : c2 : t1 :<value>
r4 : cf2 : c2 : t1 :<value>
r5 : cf2 : c1 : t1 :<value>
r6 : cf2 : c2 : t1 :<value>
StoreFile "cf2/1234"
Shift
r1 : cf1 : c1 : t1 : <value> \x00
r1 : cf1 : c1 <value> : t1 : \x00
r1- <value>: cf1 : c1 : t1 : \x00
= Same Storage Requirements

HBase Data Model - storage


cf1:c1
cf1:c2
cf2:c1
cf2:c2
r1
r2
r3
r4
r5
r6
Fold
cf1:c1 cf1:c2 cf1:
cf2:c1 cf2:c2 cf2:
Store
r1 : cf1 : c1 : t1 :<value>
r2 : cf1 : c2 : t1 :<value>
r4 : cf1 : c2 : t1 :<value>
r5 : cf1 : c1 : t1 :<value>
r6 : cf1 : c2 : t2 :<value>
r6 : cf1 : c2 : t1 :<value>
StoreFile "cf1/1234"
r3 : cf2 : c1 : t3 :<value>
r3 : cf2 : c1 : t2 :<value>
r3 : cf2 : c2 : t1 :<value>
r4 : cf2 : c2 : t1 :<value>
r5 : cf2 : c1 : t1 :<value>
r6 : cf2 : c2 : t1 :<value>
StoreFile "cf2/1234"
Shift
r1 : cf1 : c1 : t1 : <value> \x00
r1 : cf1 : c1 <value> : t1 : \x00
r1-<value>: cf1 : c1 : t1 : \x00
= Same Storage Requirements

The top-right hand side shows how the logical layout is folded into the actual physical storage layout. The cells of each row are stored one after the other, in separate storage files per column family. In other words, on disk you will have all cells of one family in (one or more) StoreFiles, and all cells of another in a different file, located in a different directory.

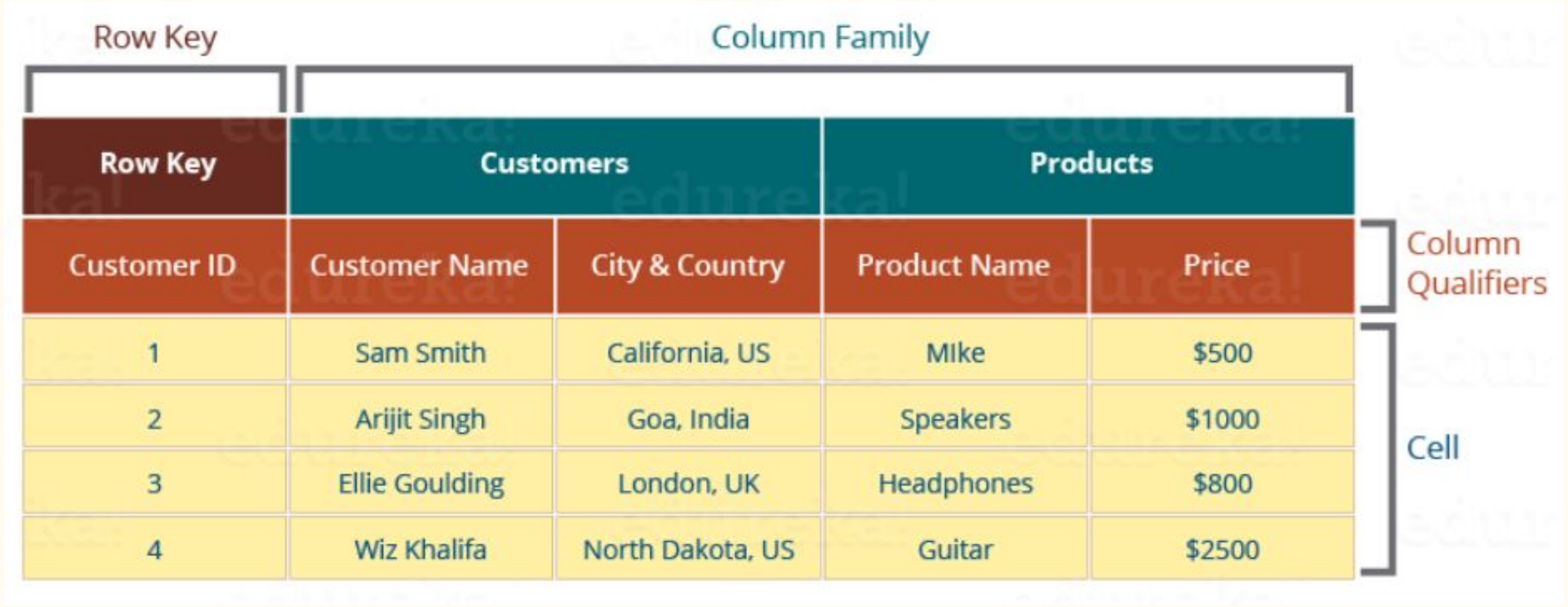
HBase Data Model - Storage



cf1:c1
cf1:c2
cf2:c1
cf2:c2
r1
r2
r3
r4
r5
r6
Fold
cf1:c1 cf1:c2 cf1:
cf2:c1 cf2:c2 cf2:
Store
r1 : cf1 : c1 : t1 :<value>
r2 : cf1 : c2 : t1 :<value>
r4 : cf1 : c2 : t1 :<value>
r5 : cf1 : c1 : t1 :<value>
r6 : cf1 : c2 : t2 :<value>
r6 : cf1 : c2 : t1 :<value>
StoreFile "cf1/1234"
r3 : cf2 : c1 : t3 :<value>
r3 : cf2 : c1 : t2 :<value>
r3 : cf2 : c2 : t1 :<value>
r4 : cf2 : c2 : t1 :<value>
r5 : cf2 : c1 : t1 :<value>
r6 : cf2 : c2 : t1 :<value>
StoreFile "cf2/1234"
Shift
r1 : cf1 : c1 : t1 : <value> \x00
r1 : cf1 : c1 <value> : t1 : \x00
r1- <value>: cf1 : c1 : t1 : \x00
= Same Storage Requirements

The top-right hand side shows how the logical layout is folded into the actual physical storage layout. The cells of each row are stored one after the other, in separate storage files per column family. In other words, on disk you will have all cells of one family in (one or more) StoreFiles, and all cells of another in a different file, located in a different directory.

HBase Schema Design

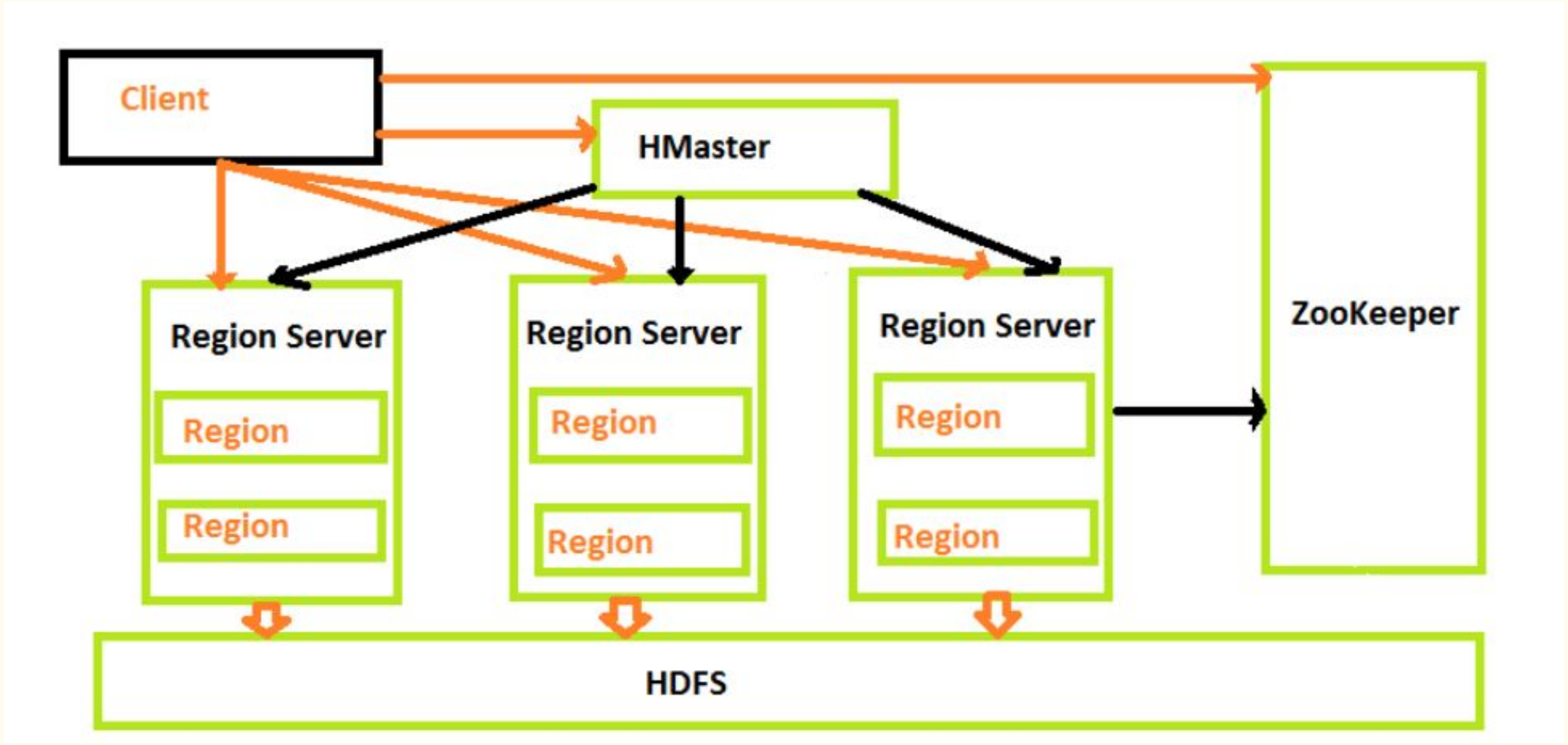


Row Key	Customers		Products	
Customer ID	Customer Name	City & Country	Product Name	Price
1	Sam Smith	California, US	MIke	$500
2	Arijit Singh	Goa, India	Speakers	$1000
3	Ellie Goulding	London, UK	Headphones	$800
4	Wiz Khalifa	North Dakota, US	Guitar	$2500

HBase Schema Design

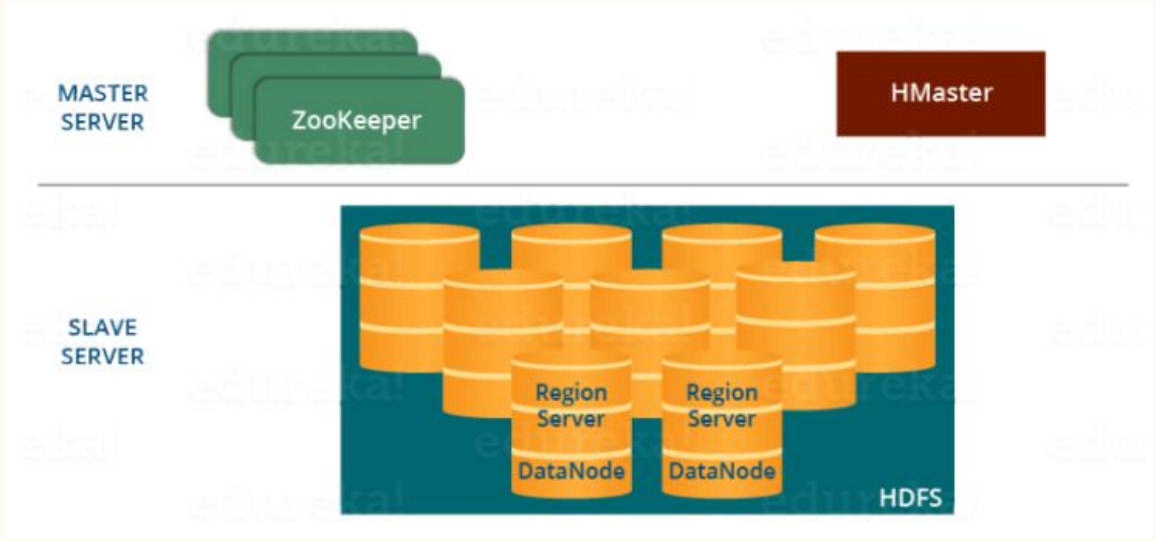
- Some of the general concepts that should be followed while designing schema in HBase:
    - ***Row key:*** Each table in the HBase is indexed on the row key. There are no secondary indices available on the HBase table.
    - ***Atomicity****:* Avoid designing a table that requires atomicity across all rows. All operations on HBase rows are atomic at row level.
    - ***Even distribution:*** Read and write should be uniformly distributed across all nodes available in the cluster. Design row key in such a way that, related entities should be stored in adjacent rows to increase read efficacy.

HBase Architecture



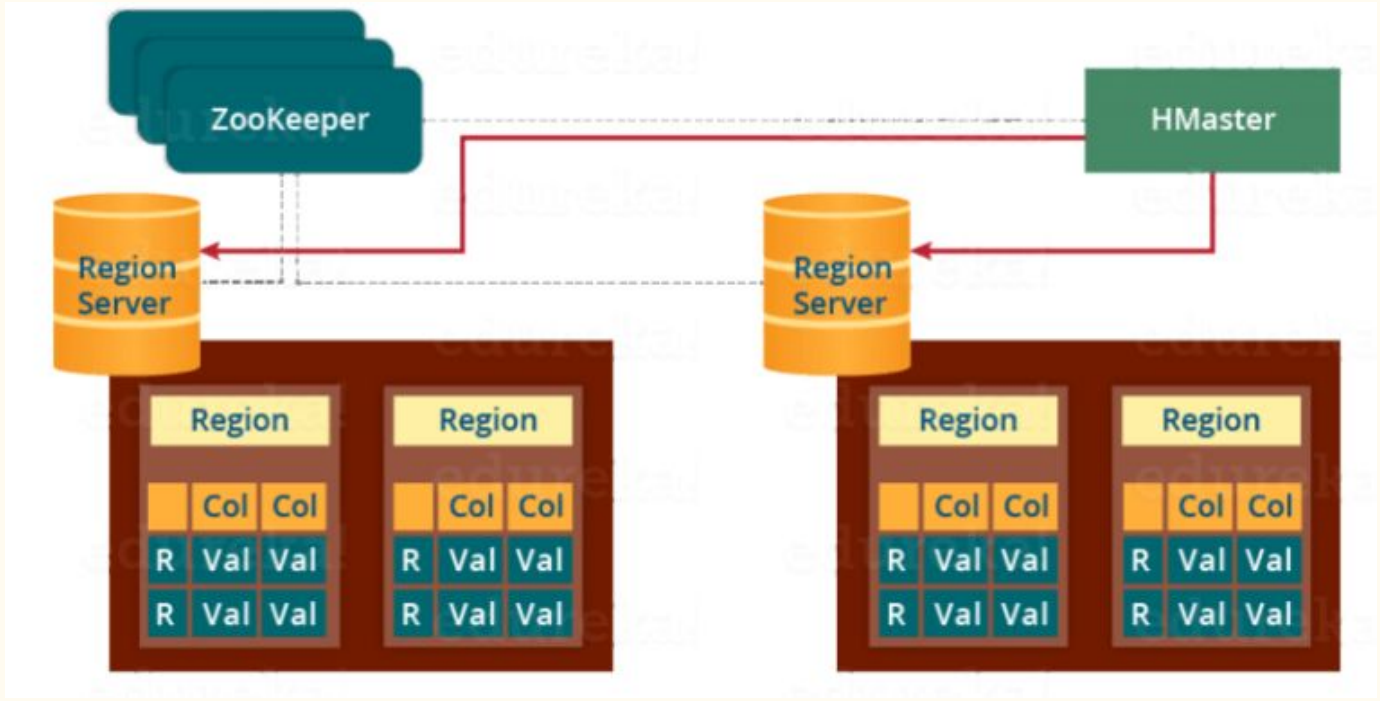
HBase Architecture

- HMaster
- Region Server
- Regions
- **Zookeeper**



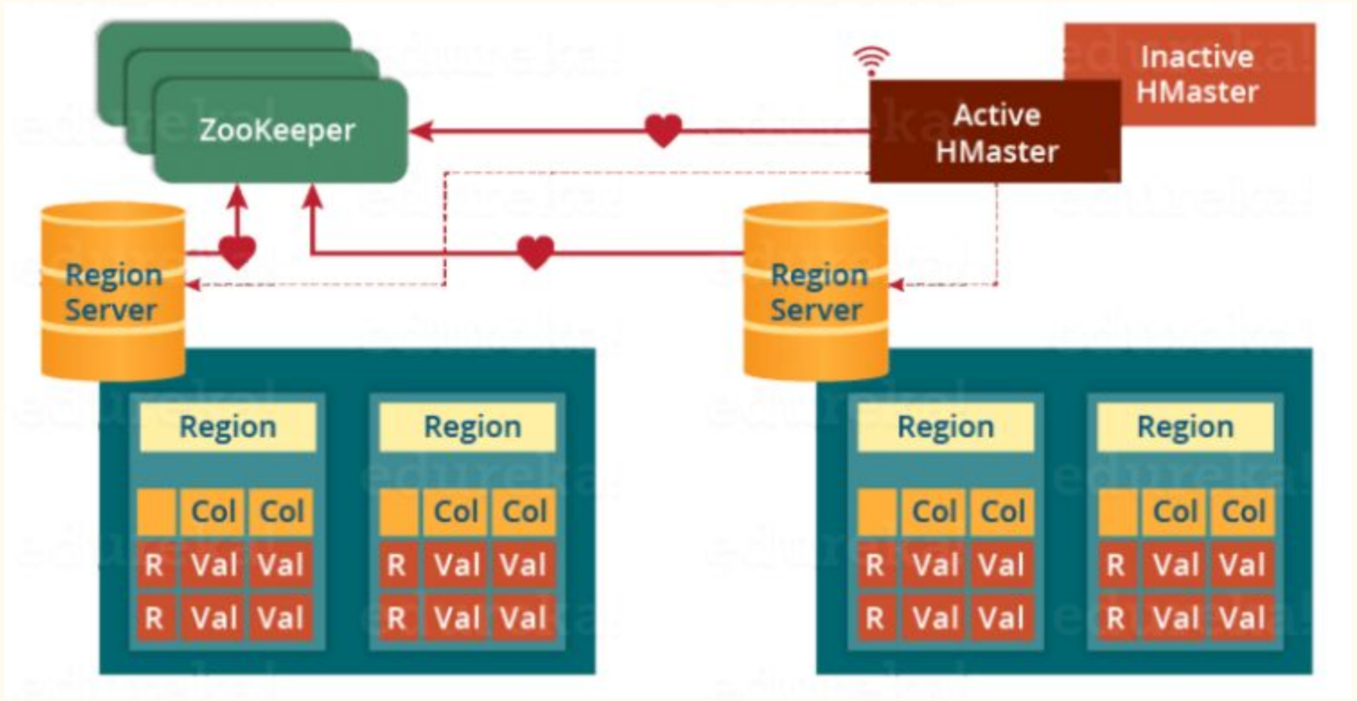
HBase Architecture

- **HMaster**
- Region Server
- **Regions**
- Zookeeper



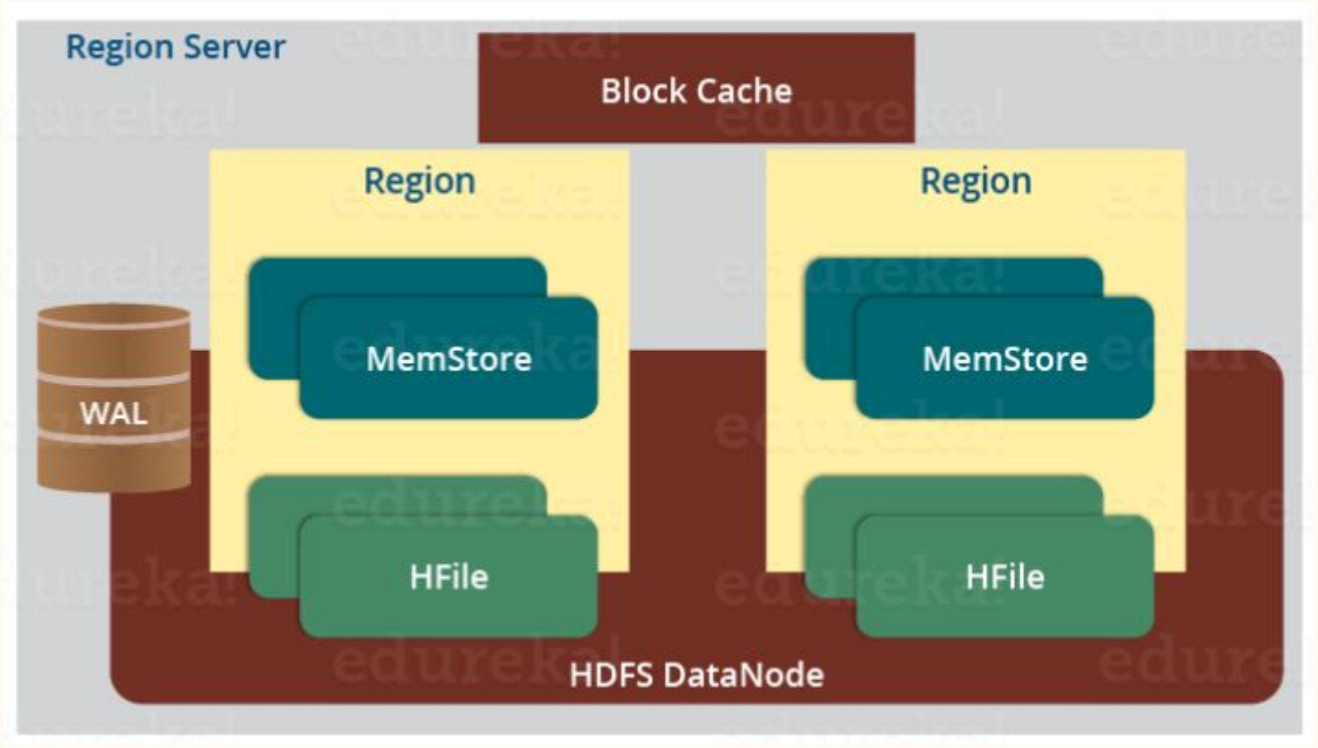
HBase Architecture

- HMaster
- Region Server
- Regions
- **Zookeeper**

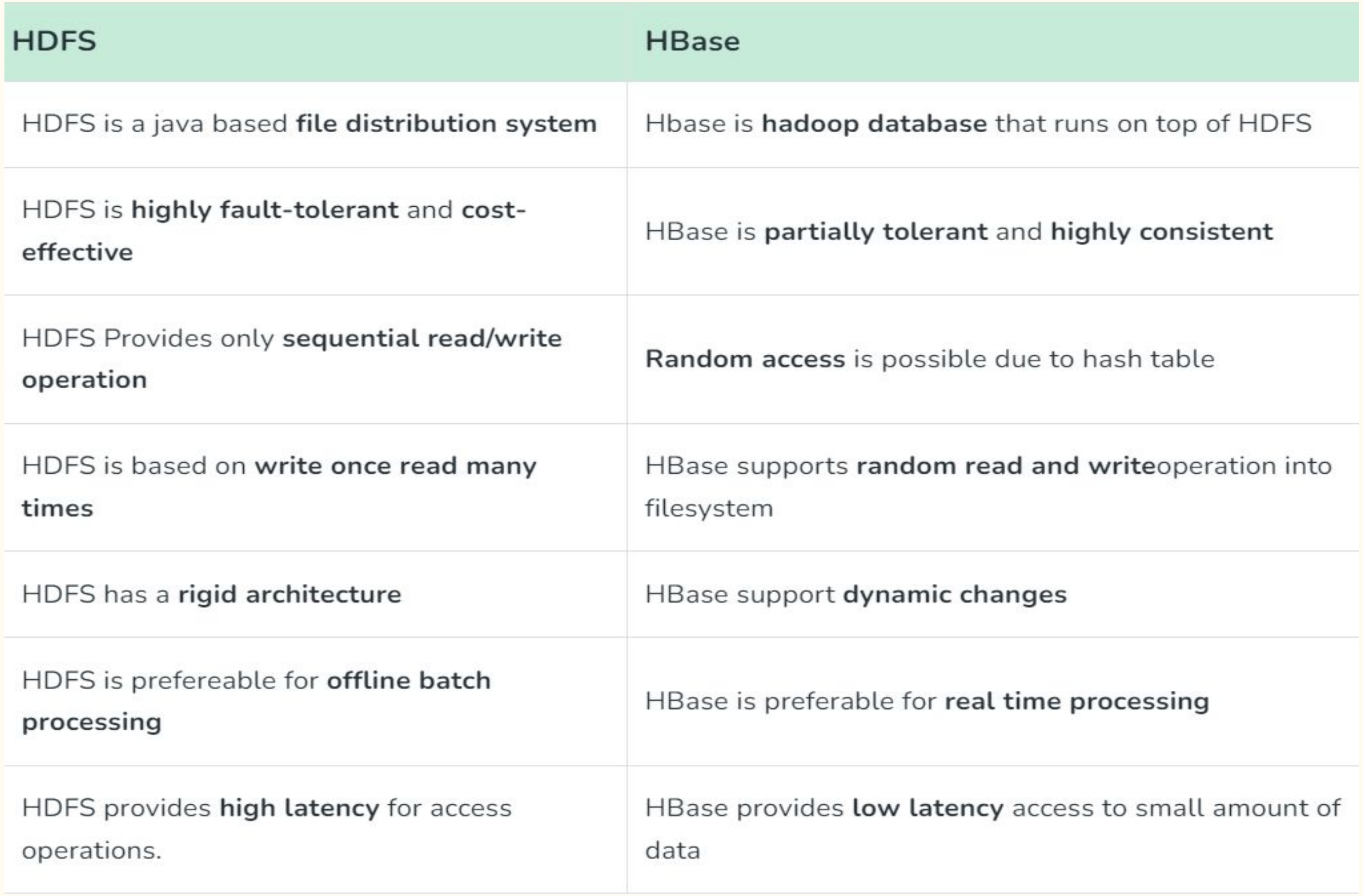


HBase Architecture

- ***Components of Region Server***

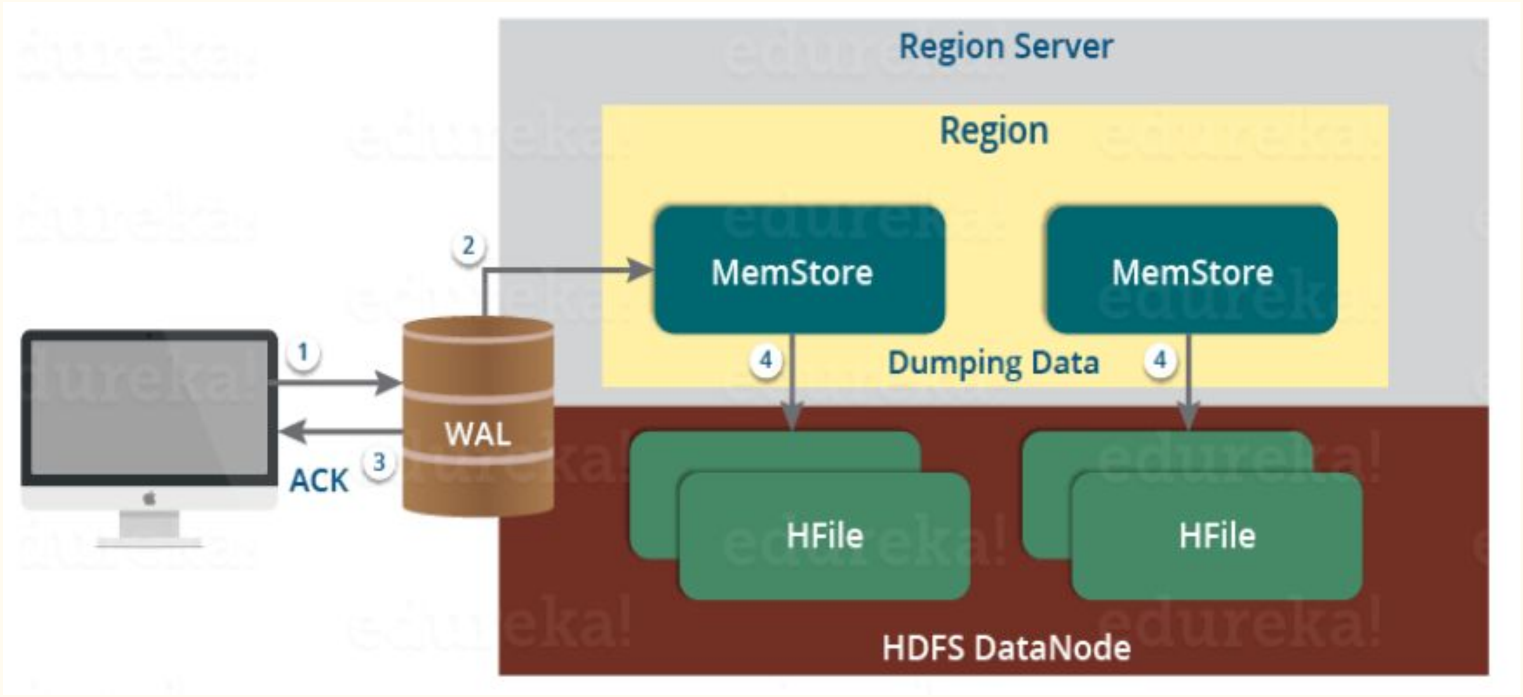


HBase vs Hadoop



HDFS	HBase
HDFS is a java based **file distribution system**	Hbase is **hadoop database** that runs on top of HDFS
HDFS is **highly fault-tolerant** and **cost-effective**	HBase is **partially tolerant** and **highly consistent**
HDFS Provides only **sequential read/write operation**	**Random access** is possible due to hash table
HDFS is based on **write once read many times**	HBase supports **random read and write**operation into filesystem
HDFS has a **rigid architecture**	HBase support **dynamic changes**
HDFS is prefereable for **offline batch processing**	HBase is preferable for **real time processing**
HDFS provides **high latency** for access operations.	HBase provides **low latency** access to small amount of data

HBase Architecture - HBase Write Mechanism



HBase Architecture - HBase Write Mechanism

- 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.
- Once data is written to the WAL, then it is copied to the MemStore.
- Once the data is placed in MemStore, then the client receives the acknowledgment.
- When the MemStore reaches the threshold, it dumps or commits the data into a HFile.

HBase Architecture - 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.

HBase Architecture - HBase Write Mechanism - Hfile

- 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.

HBase Architecture - HBase Read Mechanism

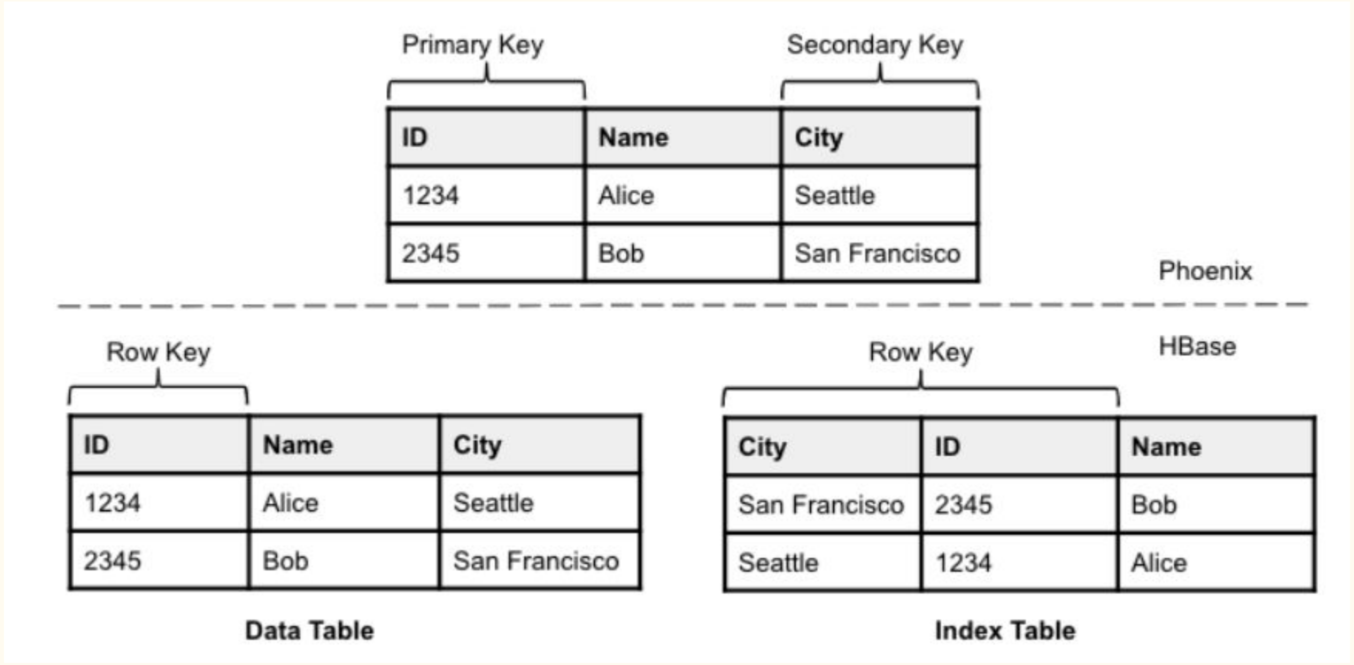
- First the client retrieves the location of the Region Server from .META Server if the client does not have it in its cache memory. Then it goes through the sequential steps as follows:
    - For reading the data, the scanner first looks for the Row cell in Block cache. Here all the recently read key 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 - 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.

Advance Indexing

- In HBase, the row key provides the same data retrieval benefits as a primary index. So, when you create a secondary index, use elements that are different from the row key.
-



ID	Name	City
1234	Alice	Seattle
2345	Bob	San Francisco

ID	Name	City
1234	Alice	Seattle
2345	Bob	San Francisco

City	ID	Name
San Francisco	2345	Bob
Seattle	1234	Alice

Advance Indexing

- Secondary indexes allow you to have a secondary way to read an HBase table.
- They provide a way to efficiently access records by means of some piece of information other than the primary key.
- Secondary indexes require additional cluster space and processing because the act of creating a secondary index requires both space and processing cycles to update.
- A method of index maintenance, called Diff-Index, can help IBM® Big SQL to create secondary indexes for HBase, maintain those indexes, and use indexes to speed up queries.

Importance of Indexing

- With secondary indexing, I can either find a single row to find all of the rows that contain an attribute with a specific value.
- Like everything in HBase, its stored in sorted order so it becomes rather trivial for the client to fetch several rows and join them in sort order, or to take the intersection if we are trying to find the records that meet a specific qualification. (e.g. find all of Bob's employees who live in Cleveland, OH. Or find the average cost of repairing a Volvo S80 that was involved in a front end collision....)
- As more people want to use HBase like a database and apply SQL, using secondary indexing makes filtering and doing data joins much more efficient. One just takes the intersection of the indexed qualifiers specified, and then apply the unindexed qualifiers as filters further reducing the result set

END (Part 3)