SWE404/DMT413
BIG DATA ANALYTICS

Lecture 3: HBase

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

History

- How exactly an entity such as Google stores all those references and web pages for your use?
  - Petabytes of data, such as Maps, Finance, Earth, needs to be stored in Google's database.
- Google proposed its own data storage system called BigTable in 2004.
- HBase is an implementation of Google's BigTable distributed data storage system.
  - HBase joined the Apache Open Source community in early 2007.
  - By 2008 HBase had become a sub-project of Hadoop.
  - In 2010 HBase became an Apache top-level project.
  - Now, it is referred to as "the Hadoop database" on its Apache web page.

HBase

- HBase is an open source, multidimensional, distributed, scalable and **NoSQL (or non-relational) database** written in Java.
- HBase runs on top of HDFS and provides BigTable like capabilities to Hadoop.
  - Distributed data store.
  - Cluster of low-cost commodity servers.
- Allow more flexibility and adaptability as you design your application.
- Deal with massive amounts of unstructured data.
  - Most of big data applications have massive amount of unstructured data.
- Use key-value pairs to store data with *sparsity*.

RDBMS

- Represent and store structured data in tables with columns and rows.
- The tables may have dependencies on each other, or relationships.

Customer ID	*Last Name*	*First Name*	*Middle Name*	*E-mail Address*	*Street Address*
00001	Smith	John	Timothy	John.smith@xyz.com	1 Hadoop Lane, NY 11111
00002	Doe	Jane	NULL	NULL	7 HBase Ave, CA 22222

Column-Oriented Versus Row-Oriented Databases

- HBase is a column-oriented NoSQL database.
  - Column-oriented databases store table records in a sequence of columns.
  - The entries in a column are stored in contiguous locations on disks.
- Use the previous example
  - A row-oriented database stores the records as:

    00001 / Smith / John / Timothy / John.smith@xyz.com / 1 Hadoop Lane, NY 11111

    00002 / Doe / Jane / NULL / NULL / 7 HBase Ave, CA 22222
  - A column-oriented database stores the records as:

00001 / 00002	John / Jane	John.smith@xyz.com / NULL
Smith / Doe	Timothy / NULL	1 Hadoop Lane, NY 11111 / 7 HBase Ave, CA 22222

Column-Oriented Versus Row-Oriented Databases

In a column-oriented databases, all the column values are stored together.

- When the amount of data is very huge, column-oriented approach makes a single column stored together and be accessed faster.
  - Applications with large set of semi-structured or unstructured data like data mining, data warehousing, applications including analytics, etc.
- Row-oriented database stores structured data. It is efficient for less number of rows and columns.
  - Online transactional processing such as banking and finance domains.

Sparsity

- Traditional relational database stores all the information for each column.
  - NULL also takes disk storage.
- If we have millions of columns but only a few values of them are non-NULL, it will waste lots of spaces.

Customer ID	*Last Name*	*First Name*	*Middle Name*	*E-mail Address*	*Street Address*
00001	Smith	John	Timothy	John.smith@xyz.com	1 Hadoop Lane, NY 11111
00002	Doe	Jane	NULL	NULL	7 HBase Ave, CA 22222

Traditional customer contact information table

Sparsity

- To solve the problem, sparsity is designed for storing sparse data records at no cost.
  - We don't store NULL.
- A similar example is how some programming languages store sparse matrix.
  - Only a very small number of elements in a sparse matrix is non-zero.
  - It is wasteful if all the zeros are stored.
  - Instead, we only store the non-zero values and their indices
    - (0, 0): 1.0
    - (0, 2): 5.0
    - (1, 1): 3.0
    - …

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

a sparse matrix

HBase Data Type

- HBase data stores consist of one or more tables, which are indexed by *row keys*.
- Columns are grouped into *column families*, which must be defined up front during table creation.

Row Key	***Column Family: {Column Qualifier: Version: Value}***
00001	CustomerName: {'FN': 1383859182496: 'John', 'LN': 1383859182858: 'Smith', 'MN': 1383859183001: 'Timothy', 'MN': 1383859182915: 'T'} ContactInfo: {'EA': 1383859183030: 'John.Smith@xyz.com', 'SA': 1383859183073: '1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103: 'Jane', 'LN': 1383859183163: 'Doe', ContactInfo: {'SA': 1383859185577: '7 HBase Ave, CA 22222'}

Logical View of Customer Contact Information in HBase

Row Keys

- Row keys are implemented as *byte arrays*, and are sorted in byte-lexicographical order.
  - The row keys are sorted, byte by byte, from left to right.
    - If *row_key_1[0]<row_key_2[0]*, we don't need to compare *row_key_1[1]* and *row_key_2[2]*.
  - It's common to use printable (ASCII) characters rather than numeric values for row keys in HBase.
    - E.g. !<+<0<1<A<B<a<b.
- Sorted row keys can help you access your data faster.
  - One of the important features in HBase.

Column

Each column consists four elements:

- Column Family
- Column Qualifier
- Version
- Value

Row Key	*Column Family: {Column Qualifier: Version: Value}*
00001	CustomerName: {'FN': 1383859182496: 'John', 'LN': 1383859182858: 'Smith', 'MN': 1383859183001: 'Timothy', 'MN': 1383859182915: 'T'} ContactInfo: {'EA': 1383859183030: 'John.Smith@xyz.com', 'SA': 1383859183073: '1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103: 'Jane', 'LN': 1383859183163: 'Doe', ContactInfo: {'SA': 1383859185577: '7 HBase Ave, CA 22222'}

Column Families

- The administrator is required to define one or more column families when creating table.
  - Generally, column families remain fixed throughout the lifetime of an HBase table.
  - New column families can be added by using administrative commands.
- The official recommendation for the number of column families per table was three or less.
- Store data with similar access patterns in the same column family.
  - E. g. logically, a customer's middle name won't be stored in a separate column family from the first or last name.

Row Key	*Column Family: {Column Qualifier: Version: Value}*
00001	CustomerName: {'FN': 1383859182496: 'John', 'LN': 1383859182858: 'Smith', 'MN': 1383859183001: 'Timothy', 'MN': 1383859182915: 'T'} ContactInfo: {'EA': 1383859183030: 'John.Smith@xyz.com', 'SA': 1383859183073: '1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103: 'Jane', 'LN': 1383859183163: 'Doe', ContactInfo: {'SA': 1383859185577: '7 HBase Ave, CA 22222'}

Column Qualifiers

- Column qualifiers are specific field names assigned to data values for identification.
  - Unlimited in content, length and number.
  - The system will assign one if it is omitted.
- Column qualifiers can be different for each row.
  - This is how sparsity is implemented.
  - E.g. the second row doesn't have column qualifiers 'MN' in CustomerName and 'EA' in ContactInfo.

Row Key	*Column Family: {Column Qualifier: Version: Value}*
00001	CustomerName: {'FN': 1383859182496: 'John', 'LN': 1383859182858: 'Smith', 'MN': 1383859183001: 'Timothy', 'MN': 1383859182915: 'T'} ContactInfo: {'EA': 1383859183030: 'John.Smith@xyz.com', 'SA': 1383859183073: '1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103: 'Jane', 'LN': 1383859183163: 'Doe', ContactInfo: {'SA': 1383859185577: '7 HBase Ave, CA 22222'}

Versions

- In the table, the version is a number is associated with each value.
- The version is a timestamp by default with the Unix time.
  - It's also possible to create a custom versioning scheme.
  - Unix time: the number of milliseconds since midnight January 1, 1970 UTC.

Row Key	*Column Family: {Column Qualifier: Version: Value}*
00001	CustomerName: {'FN': 1383859182496: 'John', 'LN': 1383859182858: 'Smith', 'MN': 1383859183001: 'Timothy', 'MN': 1383859182915: 'T'} ContactInfo: {'EA': 1383859183030: 'John.Smith@xyz.com', 'SA': 1383859183073: '1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103: 'Jane', 'LN': 1383859183163: 'Doe', ContactInfo: {'SA': 1383859185577: '7 HBase Ave, CA 22222'}

Versions

- The versioned data is stored in decreasing order.
  - The most recent value is returned by default unless a query specifies a particular timestamp.
- For example
  - At first there is only an initial for John Smith's middle name "T".
  - But then later on they learned that the "T" stood for "Timothy" and store it.
  - The most recent value for the 'MN' column is stored first in the table.

Row Key	*Column Family: {Column Qualifier: Version: Value}*
00001	CustomerName: {'FN': 1383859182496: 'John', 'LN': 1383859182858: 'Smith', 'MN': 1383859183001: 'Timothy', 'MN': 1383859182915: 'T'} ContactInfo: {'EA': 1383859183030: 'John.Smith@xyz.com', 'SA': 1383859183073: '1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103: 'Jane', 'LN': 1383859183163: 'Doe', ContactInfo: {'SA': 1383859185577: '7 HBase Ave, CA 22222'}

Key Value Pairs

- Row key can be used as the primary key for data stored in HBase.
- However, query only with the row key can potentially return a ton of data.
  - An individual row can have millions of columns.
  - HBase can return every column qualifier, version, and value related to the row key.
- What if you want only a particular column or version of your data?
  - E.g. what happens if you want only the last name of a particular customer?
- The solution is to build a more complex key to specify exactly what you need. A key-value pair can look like this:

```
Row Key:(Column Family: Column Qualifier: Version) => Value
```

Key Value Pairs

- The more specific the query, the more granular the results.
  - Spend more time locating the exact value.
  - But less data is returned.
- For example
  - If you want the most recent middle name of the customer in row '00001' :

```
'00001:CustomerName:MN' => 'Timothy'
```

  - If you want the previously recorded middle name, specify the version:

```
'00001:CustomerName:MN:1383859182915' => 'T'
```

Row Key	*Column Family: {Column Qualifier: Version: Value}*
00001	CustomerName: {'FN': 1383859182496: 'John', 'LN': 1383859182858: 'Smith', 'MN': 1383859183001: 'Timothy', 'MN': 1383859182915: 'T'} ContactInfo: {'EA': 1383859183030: 'John.Smith@xyz.com', 'SA': 1383859183073: '1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103: 'Jane', 'LN': 1383859183163: 'Doe', ContactInfo: {'SA': 1383859185577: '7 HBase Ave, CA 22222'}

Summary of HBase Data Model

- Set of tables.
- Each table with column families and rows.
- Row key acts as a Primary key in HBase.
- Any access to HBase tables uses this Primary Key.
- Each column qualifier denotes the attribute of value.
- Version is the timestamp of the value.

Architecture

- We have learned how a HBase table looks like.
- Now, how these tables are stored in distributed manner?

Architecture

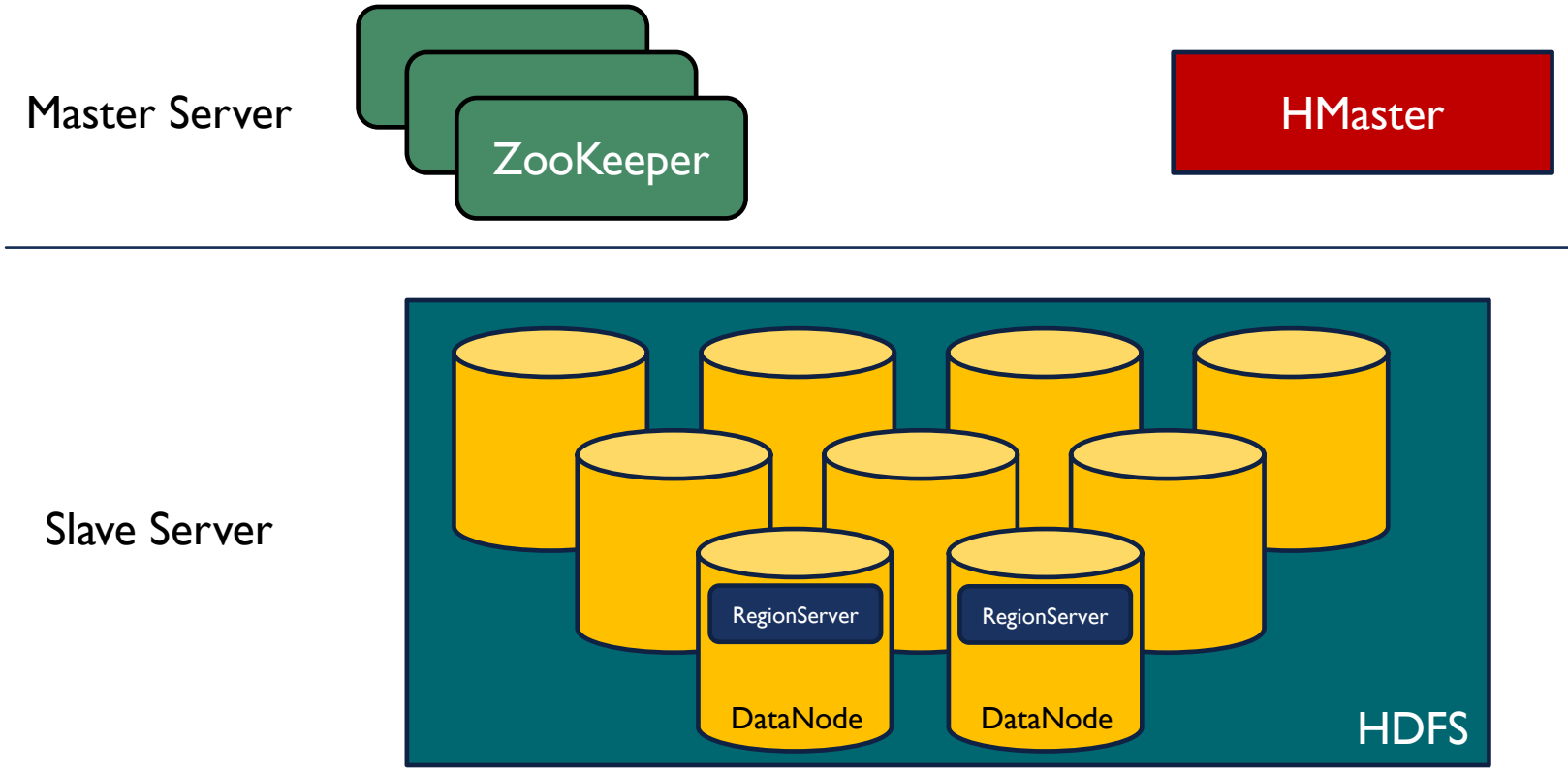


Image source: https://www.edureka.co/blog/hbase-architecture/

Key Concepts

From the top of the hierarchy:

- HMaster
- ZooKeeper
- RegionServer
- Regions

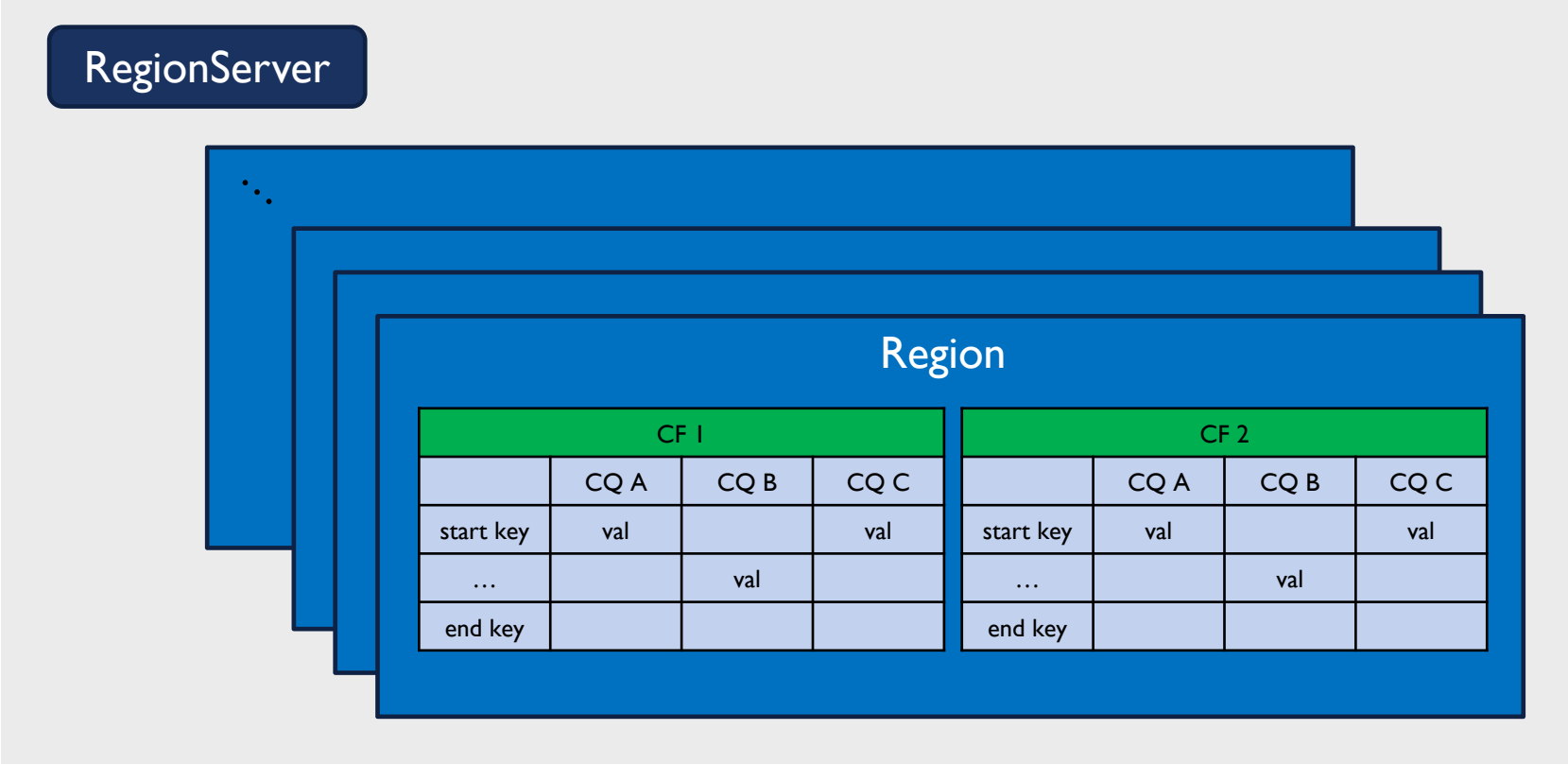
Regions

- HBase tables are *divided horizontally by row key* range into a number of regions.
  - In such a way, all the columns of a column family is stored in one region.
- A region contains all the rows between the start key and the end key assigned to that region in a sorted order.
- Many regions are assigned to a RegionServer, which is responsible for handling, managing, executing reads and writes operations on that set of regions.
- A Region has a default size of 1GB (configurable).
- A RegionServer can serve approximately 1000 regions to the client.

RegionServers

- RegionServers are daemons you activate to store and retrieve data in HBase.
- Each RegionServer is deployed on its own dedicated compute node (DataNode).
- When the table grows beyond a configurable limit, the RegionServer automatically splits the table and distributes the load to another RegionServer.
  - Often referred to as *auto-sharding*.
  - New RegionServer is allocated by HMaster.
  - HBase automatically scales as you add data to the system without manual intervention.

Regions


RegionServer
Region
CF 1
CQ A
CQ B
CQ C
start key
val
val
…
val
end key
CF 2
CQ A
CQ B
CQ C
start key
val
val
…
val
end key

HMaster

- Coordinate and manage the RegionServer.
  - Similar as NameNode manages DataNode in HDFS.
- Perform DDL operations (create and delete tables) and assign regions to the RegionServers.
- Monitor all the RegionServer's instances in the cluster (with the help of Zookeeper) and perform recovery activities whenever any RegionServer is crashed.
- Provide an interface for creating, deleting and updating tables.

ZooKeeper

- Zookeeper acts like a coordinator inside HBase distributed environment.
  - It helps in maintaining server state inside the cluster by communicating through sessions.
- Every RegionServer along with the HMaster sends continuous heartbeat at regular interval to Zookeeper and it checks which server is alive and available.

HMaster and ZooKeeper

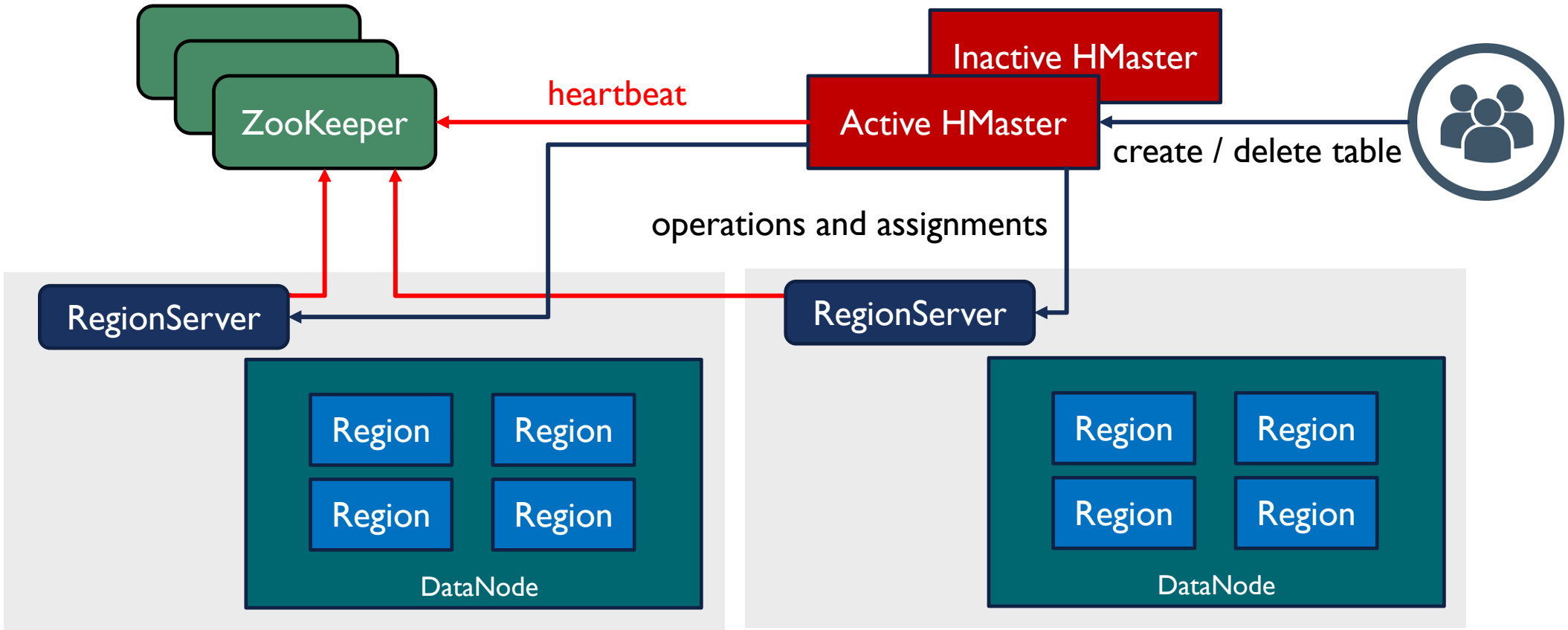


Image refer to: https://www.edureka.co/blog/hbase-architecture/

Meta Table

- The META table is a special HBase catalog table.
  - It holds the location of the regions in the cluster.
  - ZooKeeper stores the location of the META table.
- 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 RegionServer.

Row Key	*Value*
table 1, key 1, region 1	RegionServer 1
table 1, key 2, region 1	RegionServer 2
table 1, key 1, region 2	RegionServer 3
…	…

META table

HBase Reading and Writing

Four key components for HBase reading and writing:

- BlockCache
- MemStore
- Write Ahead Log (WAL)
- HFile

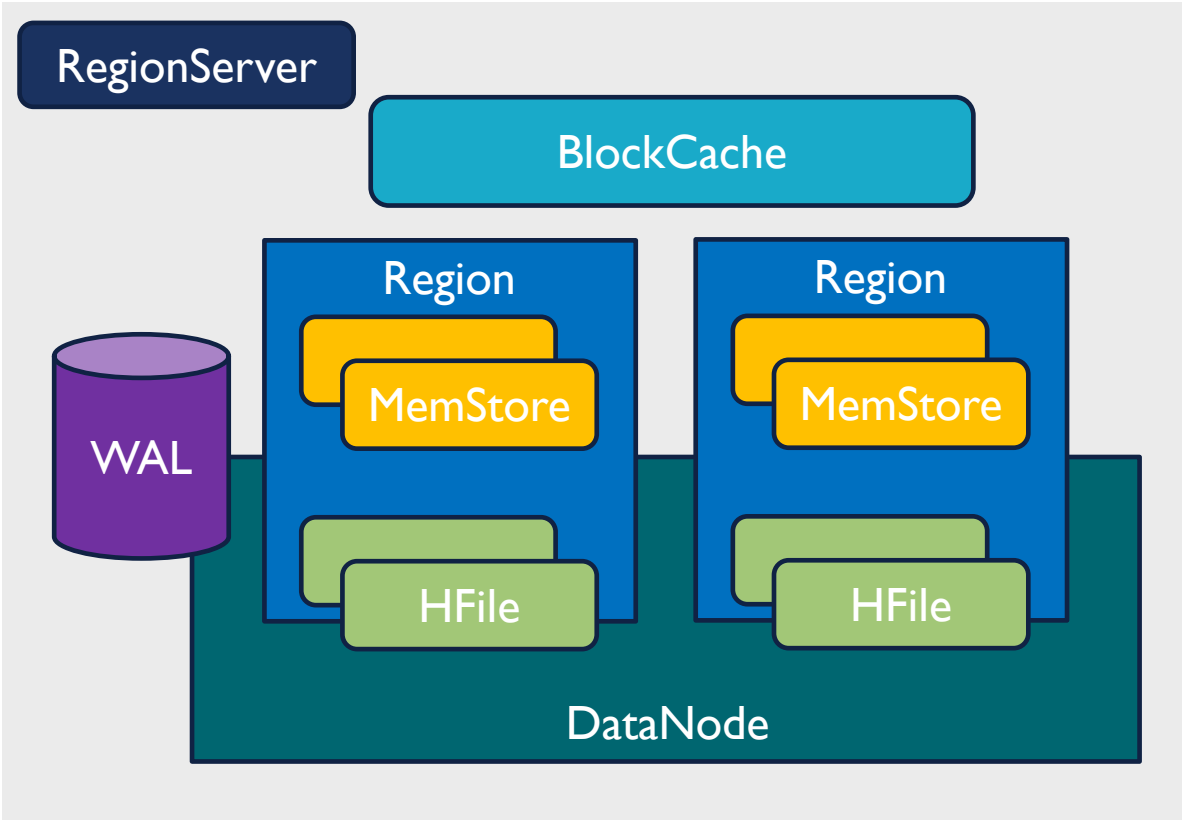


Image refer to: https://www.edureka.co/blog/hbase-architecture/

BlockCache

BlockCache resides on the top of RegionServer.

- Data is read in blocks from the HDFS and stored in the BlockCache.
- 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.

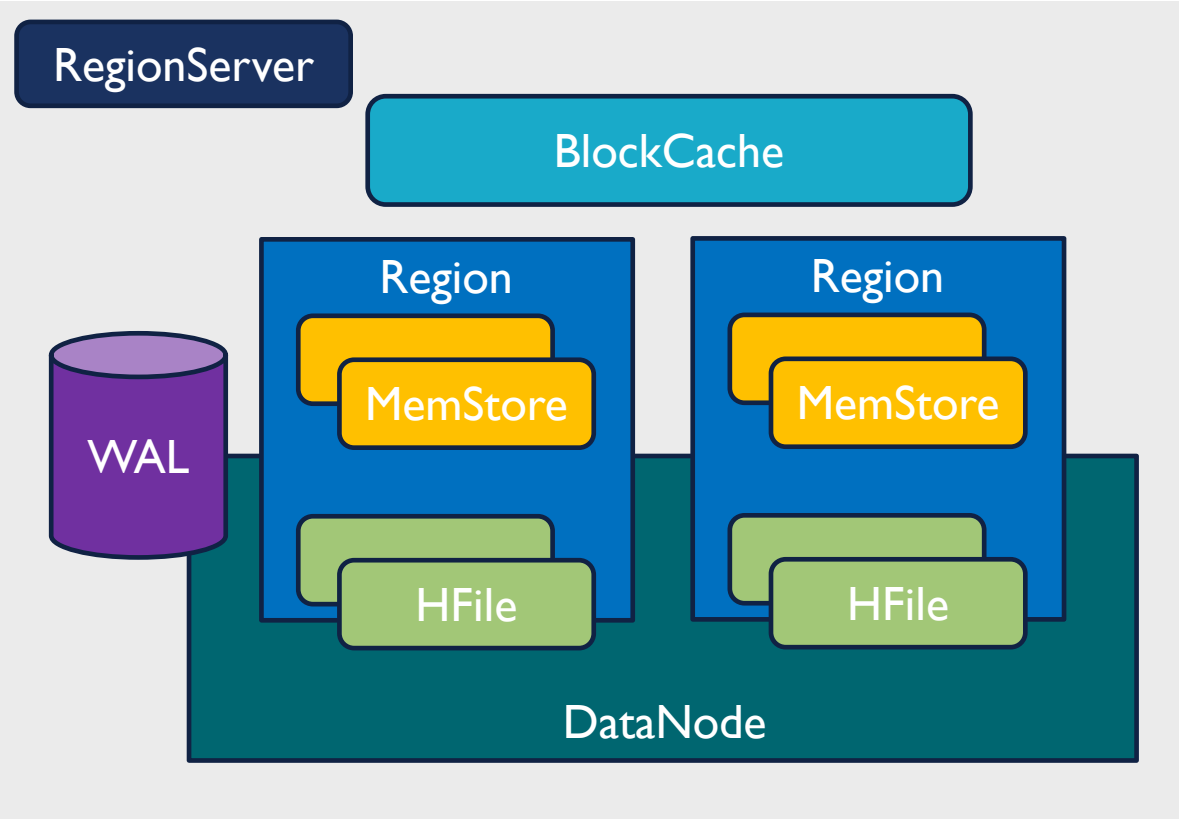


Image refer to: https://www.edureka.co/blog/hbase-architecture/

MemStore

MemStore is a write cache *for each column family*.

- It stores all the incoming data before committing it to the disk or permanent memory.
- The data is sorted in lexicographical order before committing it to the disk.

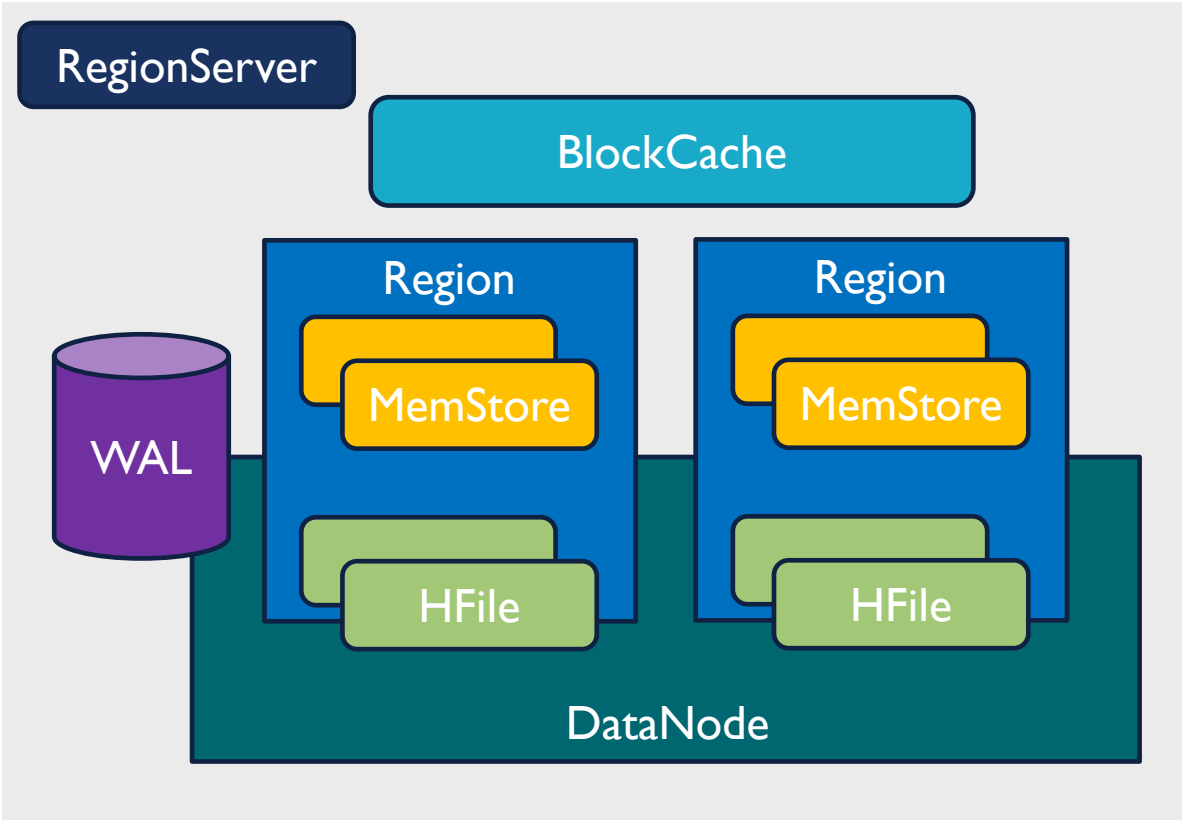


Image refer to: https://www.edureka.co/blog/hbase-architecture/

Write Ahead Log (WAL)

Write Ahead Log (WAL) is a file *on the disk* attached to every RegionServer.

- It 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.

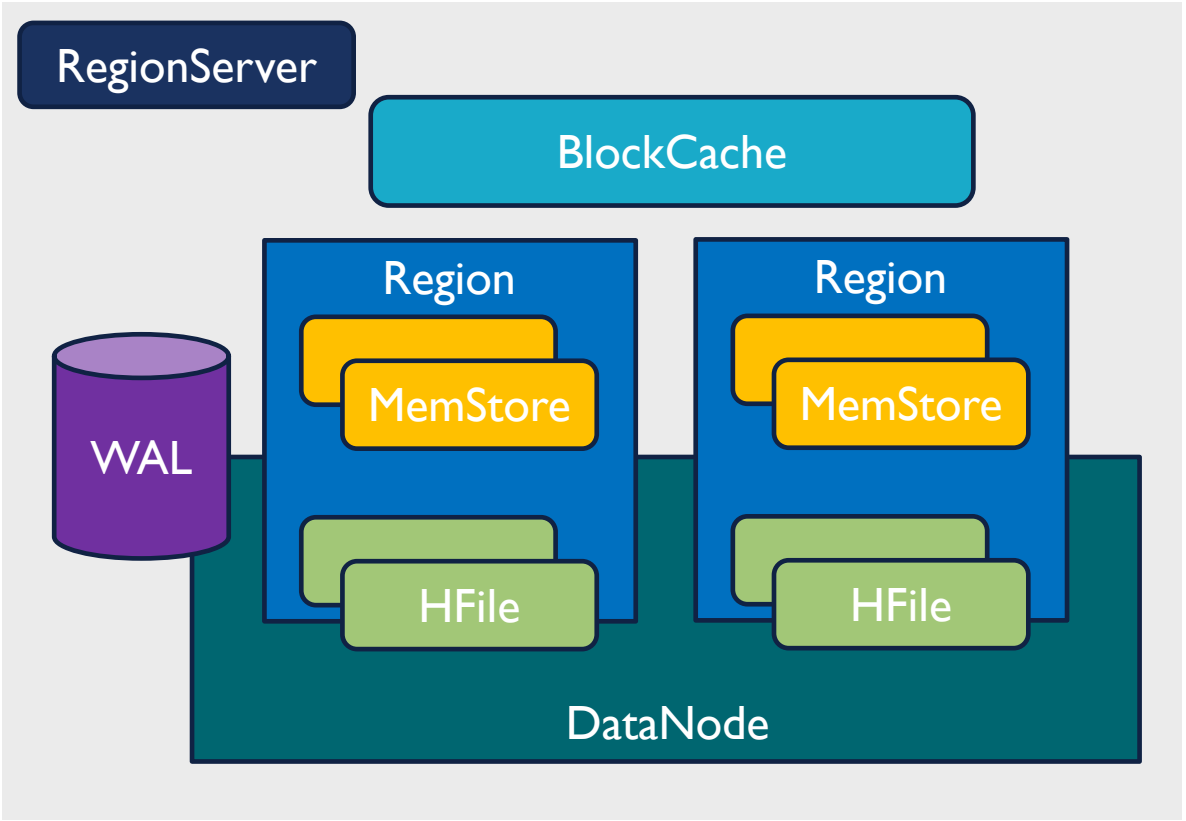


Image refer to: https://www.edureka.co/blog/hbase-architecture/

HFile

HFile is stored on HDFS.

- It stores the actual cells on the disk.
- MemStore commits the data to HFile when the size of MemStore exceeds.

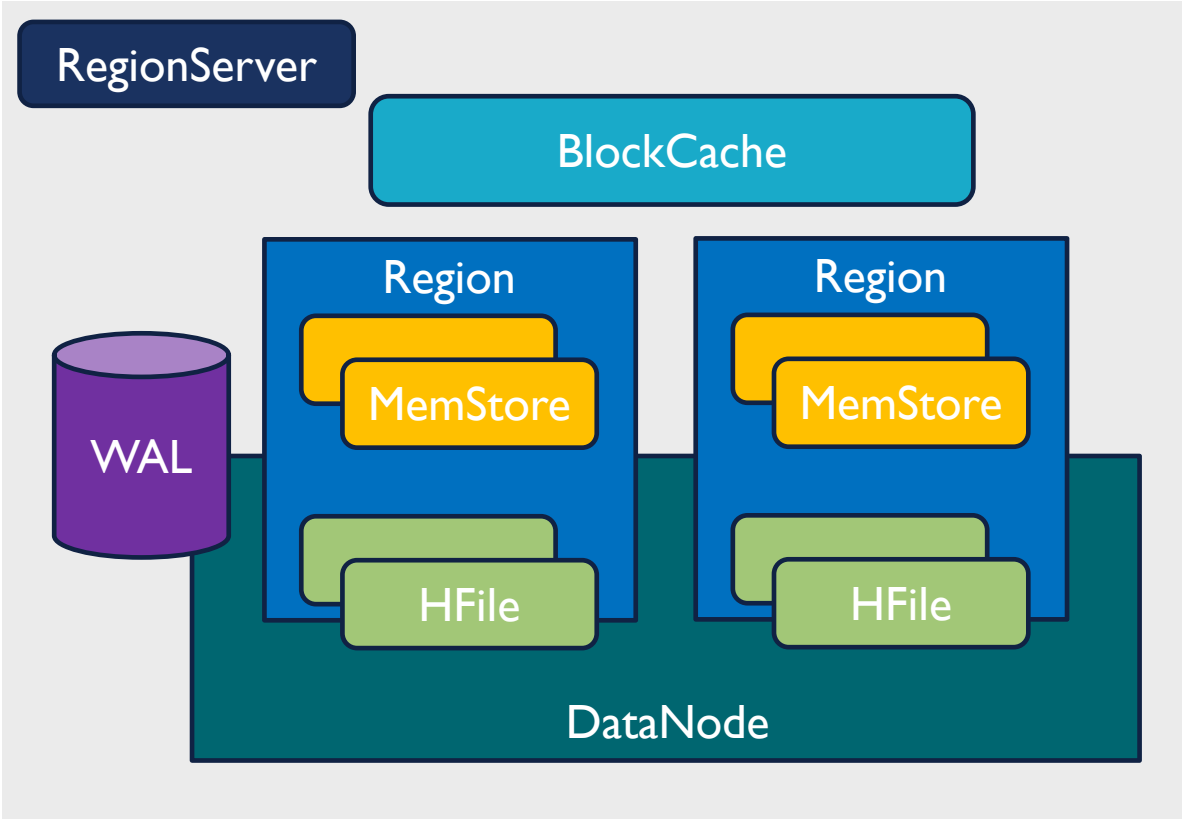
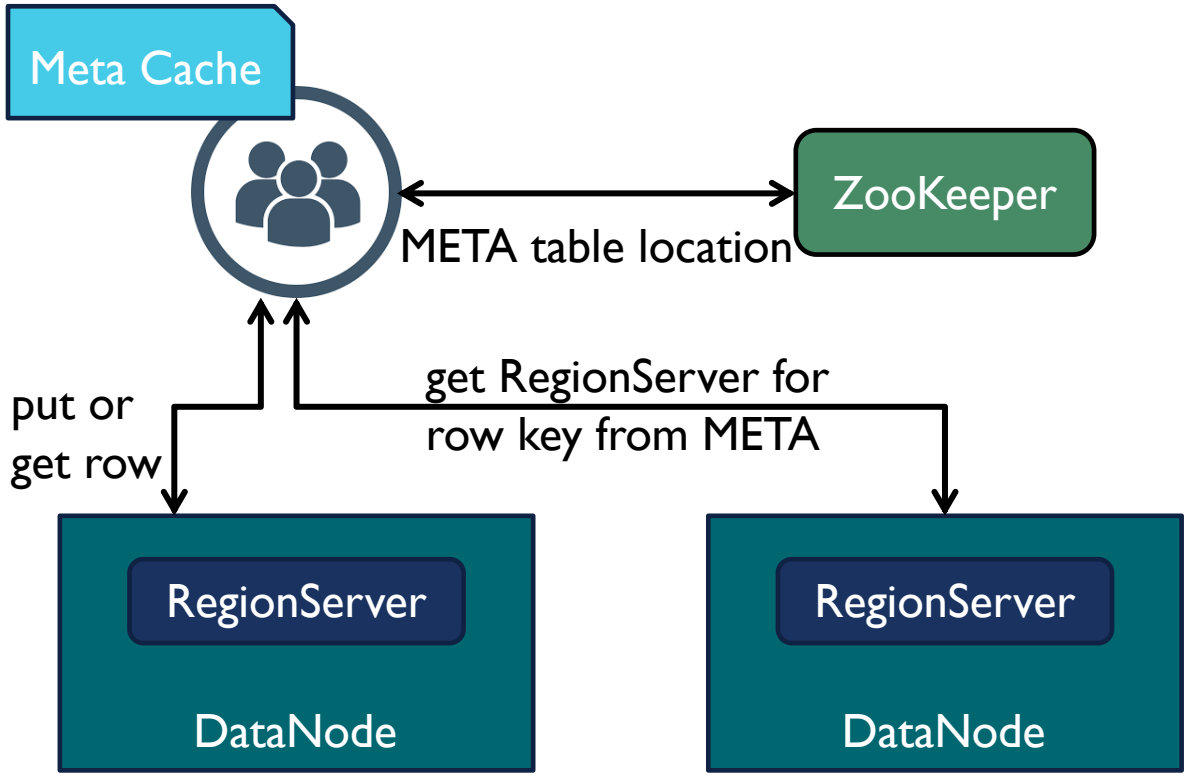


Image refer to: https://www.edureka.co/blog/hbase-architecture/

First Time Reading & Writing

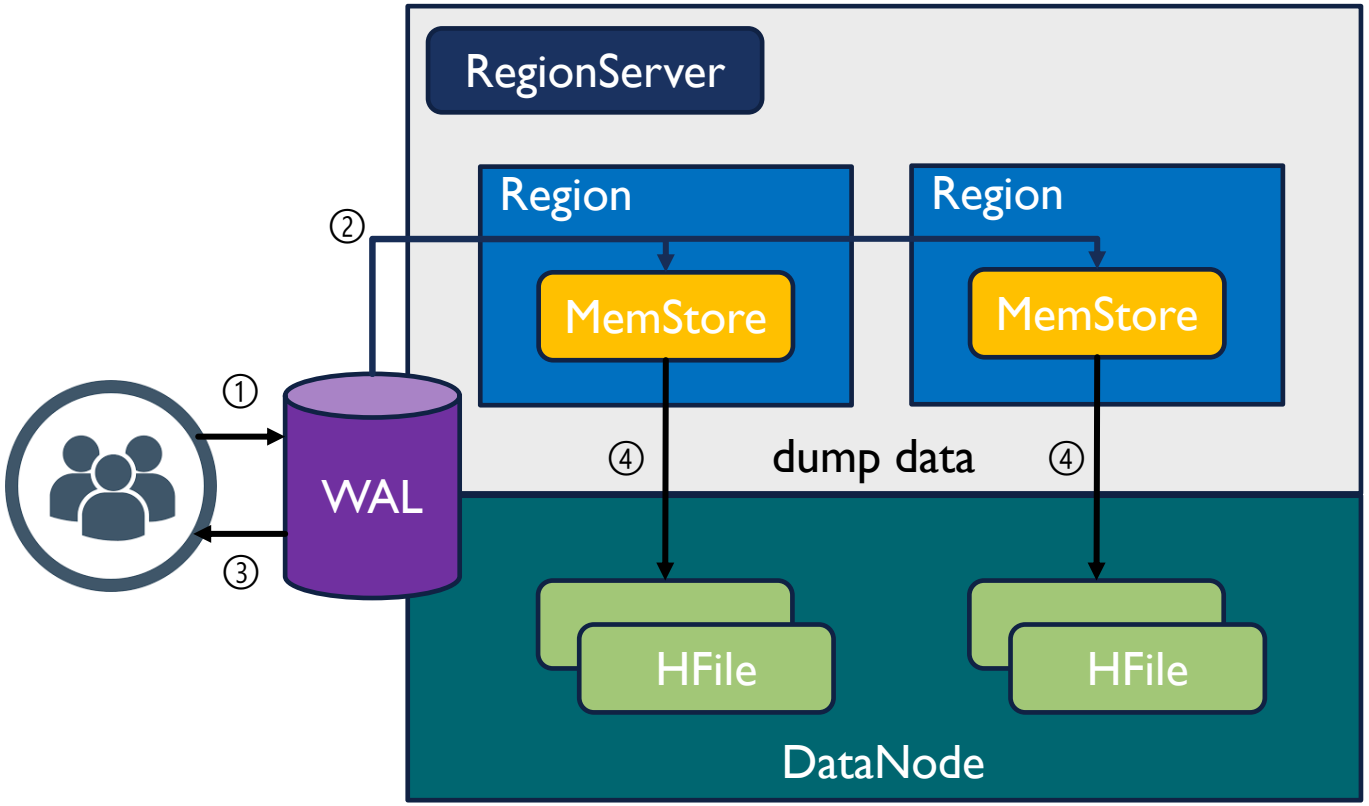
The first time a client reads or writes to HBase:

- The client gets the RegionServer that hosts the META table from ZooKeeper.
- The client will query the server to get the RegionServer corresponding to the row key it wants to access.
  - The client caches this information along with the META table location.
- It will get the row from the corresponding RegionServer.
- For future reads, the client uses the cache to retrieve the META location and previously read row keys.



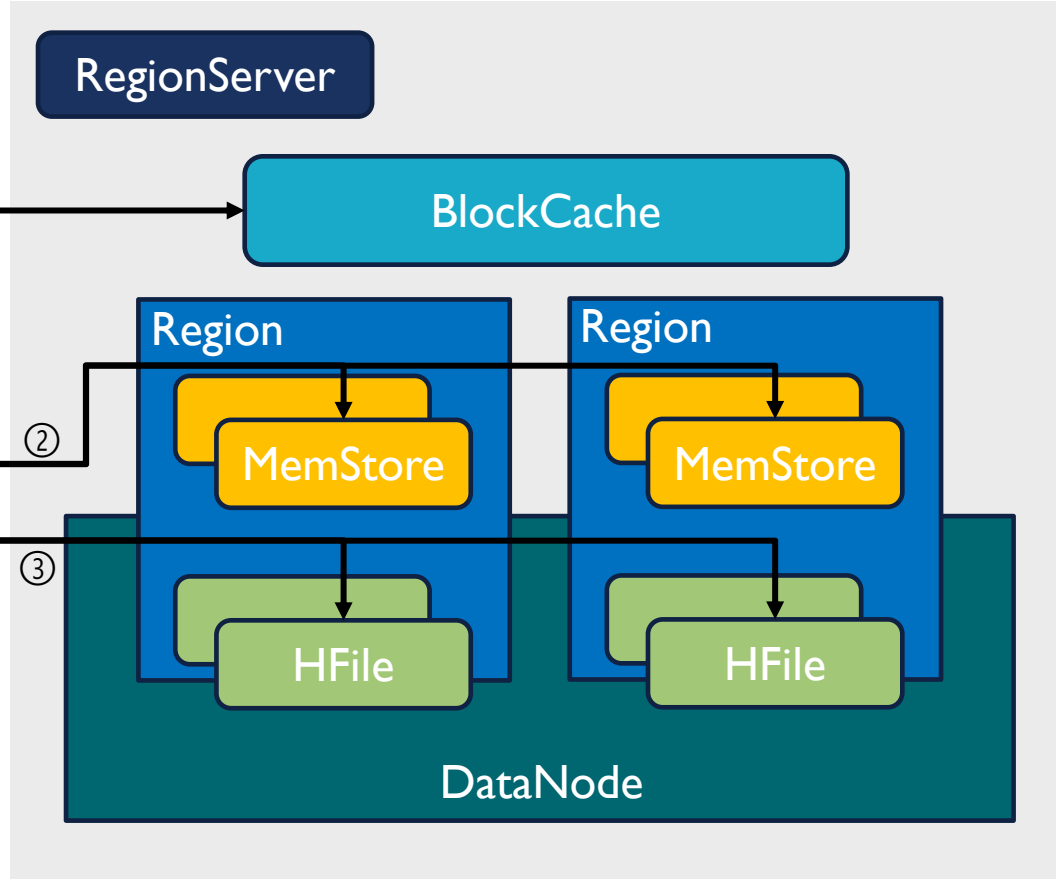
Write Mechanism

- **Step 1**: The client writes the data to WAL on every RegionServer first.
  - The edits are then appended at the end of the WAL file.
  - RegionServer uses WAL 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 the data into a HFile.



Read Mechanism

- **Step 1**: The client scanner first looks for the row cell in BlockCache.
  - Here all the recently read key value pairs are stored.
- **Step 2**: If scanner fails to find the required result, it moves to the MemStore.
  - Search for the most recently written files, which has not been dumped yet in HFile.
- **Step 3**: If the scanner does not find all of the row cells in the MemStore and BlockCache, then HBase will use the BlockCache indices and bloom filters to load target row cells from HFiles into memory.



Compaction

- Compaction is the process that HBase combines HFiles to reduce the storage and reduce the number of disk seeks needed for a read.
- Compaction chooses some HFiles from a region and combines them. There are two types of compaction.
  - **Minor Compaction**: Pick smaller HFiles and recommits them to bigger HFiles. This helps in storage space optimization.
  - **Major Compaction**: Merge and recommit the smaller HFiles of a region to a new HFile, such that the same column families are placed together in the new HFile. 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.

Region Split

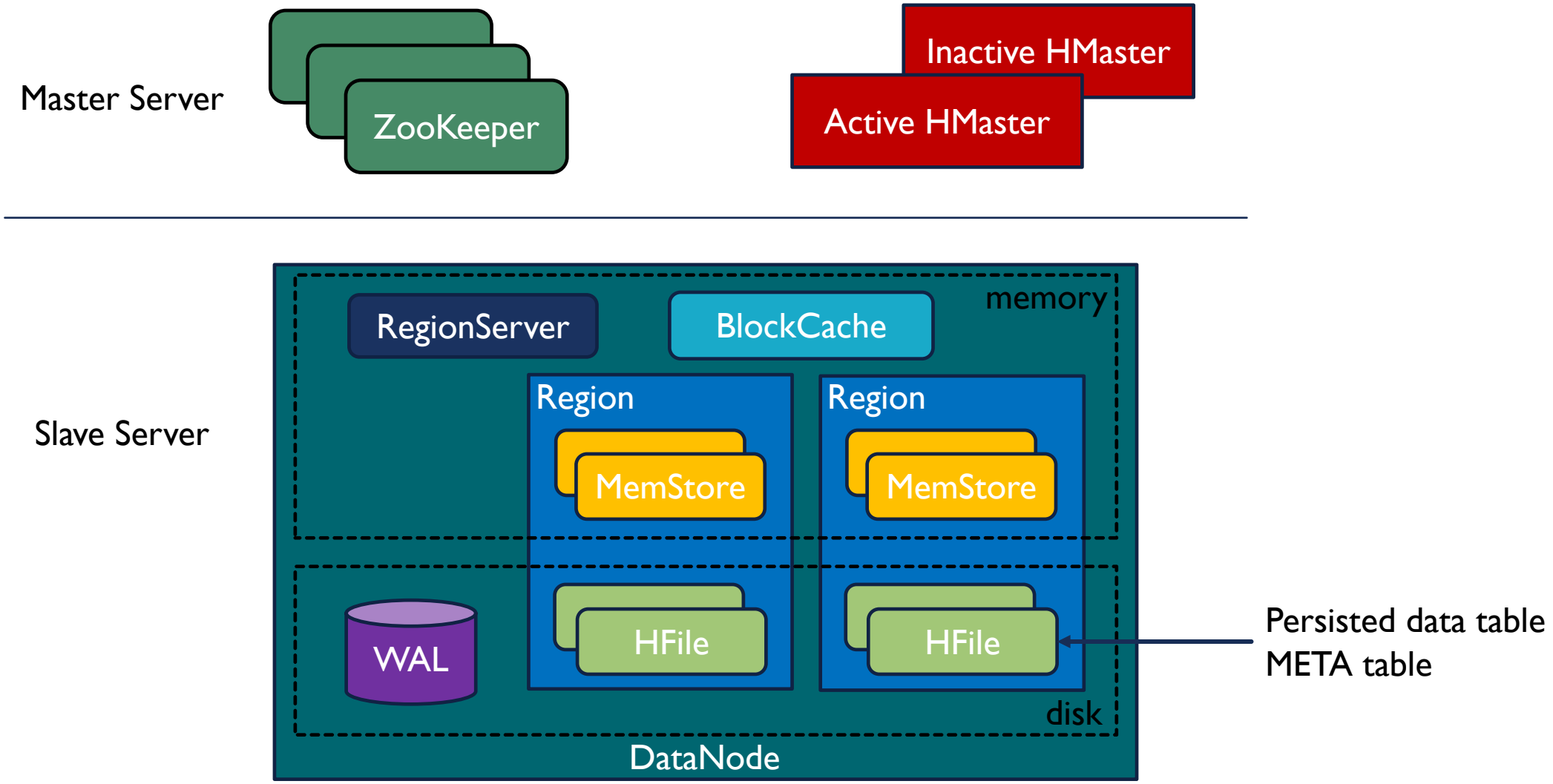
- Whenever a region becomes large, it is divided into two child regions.
- Each region represents exactly a half of the parent region.
- Then this split is reported to the HMaster.
- This is handled by the same RegionServer until the HMaster allocates them to a new RegionServer for load balancing.

HDFS Data Replication

- HBase relies on HDFS to provide the data safety as it stores its files.
- HDFS automatically replicates the WAL and HFile blocks.
- The WAL file and the HFiles are persisted on disk and replicated, so how does HBase recover the MemStore updates not persisted to HFiles?

Crash and Data Recovery

- Zookeeper will determine node failure when it loses RegionServer heart beats. The HMaster will then be notified that the RegionServer has failed.
- Then HMaster reassigns the regions from the crashed server to active RegionServers.
- To recover MemStore, the HMaster splits the WAL belonging to the crashed RegionServer into separate files and stores these file in the new RegionServers' data nodes.
  - WAL is stored on disk and replicated by HDFS.
  - Each RegionServer then replays the WAL from the respective split WAL, to rebuild the MemStore for that region.



Big picture of HBase architecture

Features of HBase

- **Atomic read and write:** During one read or write process, all other processes are prevented from performing any read or write operations.
- **Consistent reads and writes:** HBase provides consistent reads and writes due to above feature.
- **Linear and modular scalability:** It is linearly scalable across various nodes, as well as modularly scalable, as it is divided across various nodes.
- **Automatic and configurable sharding of tables:** HBase tables are distributed across clusters and these clusters are distributed across regions. These regions and clusters split, and are redistributed as the data grows.

Features of HBase

- **Easy to use Java API for client access:** It provides easy to use Java API for programmatic access.
- **BlockCache and Bloom Filters:** HBase supports a BlockCache and Bloom Filters for high volume query optimization .
- **Automatic failure support:** HBase with HDFS provides WAL across clusters which provides automatic failure support.
- **Sorted row keys:** As searching is done on range of rows, HBase stores row keys in order. Using these sorted row keys and timestamp, we can build an optimized request.

Where Should We Use HBase?

- We should use HBase where we have large data sets (millions or billions or rows and columns) and we require fast, random and real time, read and write access over the data.
- The data sets are distributed across various clusters and we need high scalability to handle data.
- The data is gathered from various data sources and it is either semi structured or unstructured data or a combination of all.
- You want to store column oriented data.
- You have lots of versions of the data sets and you need to store all of them.

Conclusion

After this lecture, you should know:

- What is non-relational database and column oriented database.
- How is data represented by HBase table.
- What are the regions.
- What are the specific jobs for HMaster, ZooKeeper and RegionServer.
- How HBase conduct reading and writing.
- When do you need HBase other than RDBMS.

Thank you!

Reference (recommend for further reading):

- **The official guide**: https://hbase.apache.org/book.html#_preface
- **The paper**: Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E., 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, *26*(2), pp.1-26.
- **The book**: Chapter 12, DeRoos, Dirk. *Hadoop for dummies*. John Wiley & Sons, 2014.
- **Edureka Blogs**: https://www.edureka.co/blog/hbase-tutorial
- **MAPR Blogs**: https://mapr.com/blog/in-depth-look-hbase-architecture/