Session 11:

ADVANCE HBASE

Assignment 1

Task1

Explain the below concepts with an example in brief.

● Nosql Databases

● Types of Nosql Databases

● CAP Theorem

● HBase Architecture

● HBase vs RDBMS

Ans: NoSQL is an approach to database design that can accomodate a wide variety of data models, including key-value, document, columnar and graph formats. NoSQL, which stand for "not only SQL," is an alternative to traditional relational databases in which data is placed in tables and data schema is carefully designed before the database is built. NoSQL databases are especially useful for working with large sets of distributed data.

NoSQL is an approach to databases that represents a shift away from traditional relational database management systems (RDBMS).

NoSQL is particularly useful for storing unstructured data, which is growing far more rapidly than structured data and does not fit the relational schemas of RDBMS. Common types of unstructured data include: user and session data; chat, messaging, and log data; time series data such as IoT and device data; and large objects such as video and images.

Features of NoSQL Databases:

1. Multi-Model
Where relational databases require data to be put into tables and columns to be accessed and analyzed, the various data model capabilities of NoSQL databases make them extremely flexible when it comes to handling data. They can ingest structured, semi-structured, and unstructured data with equal ease, whereas relational databases are extremely rigid, handling primarily structured data.

Different data models handle specific application requirements. Developers and architects choose a NoSQL database to more easily handle different agile application development requirements. Popular data models include graph, document, wide-column, and key-value.

The ideal is to support multiple data models, which allows to use the same data in different data model types without having to manage a completely different database.

2. Easily Scalable
It's not that relational databases can't scale, it's that they can't scale EASILY or CHEAPLY, and that's because they're built with a traditional master-slave architecture, which means scaling UP via bigger and bigger hardware servers as opposed to OUT or worse via sharding. Sharding means dividing a database into smaller chunks across multiple hardware servers instead of a single large server, and this leads to operational administration headaches.

Instead, look for a NoSQL database with a masterless, peer-to-peer architecture with all nodes being the same. This allows easy scaling to adapt to the data volume and complexity of cloud applications. This scalabilty also improves performance, allowing for continuous availability and very high read/write speeds.

3. Flexible
Where relational databases require data to be put into tables and columns to be accesses and analyzed, the multi-model capabilities of NoSQL databases make them extremely flexible when it comes to handling data. They can easily process structured, semi-structured, and unstructured data, while relational databases, as stated previously, are designed to handle primarily structured data.

4. Distributed
Look for a NoSQL database that is designed to distribute data at global scale, meaning it can use multiple locations involving multiple data centers and/or cloud regions for write and read operations. Relational databases, in contrast, use a centralized application that is location-dependent (e.g. single location), especially for write operations. A key advantage of using a distributed database with a masterless architecture is that you can maintain continuous availability because data is distributed with multiple copies where it needs to be.

5. Zero Downtime
The final but certainly no less important key feature to seek in a NoSQL database is zero downtime. This is made possible by a masterless architecture, which allows for multiple copies of data to be maintained across different nodes. If a node goes down, no problem: another node has a copy of the data for easy, fast access. When one considers the cost of downtime, this is a big deal.

Examples of NoSQL Databases: Cassandra, MongoDb, HBase,Redis etc.

Types of NoSQL Databases:

There are 4 basic types of NoSQL databases:

1. **Key-Value Store** – It has a Big Hash Table of keys & values {Example- Riak, Amazon S3 (Dynamo)}

2. **Document-based Store- It** stores documents made up of tagged elements. {Example- CouchDB}
3. **Column-based Store-** Each storage block contains data from only one column, {Example- HBase, Cassandra}
4. **Graph-based**-A network database that uses edges and nodes to represent and store data. {Example- Neo4J}

1. Key Value Store NoSQL Database

The schema-less format of a key value database like Riak is just about what you need for your storage needs. The key can be synthetic or auto-generated while the value can be String, JSON, BLOB (basic large object) etc.

The key value type basically, uses a hash table in which there exists a unique key and a pointer to a particular item of data. A bucket is a logical group of keys – but they don't physically group the data. There can be identical keys in different buckets.

Performance is enhanced to a great degree because of the cache mechanisms that accompany the mappings. To read a value you need to know both the key and the bucket because the real key is a hash (Bucket+ Key).

2. Document Store NoSQL Database

The data which is a collection of key value pairs is compressed as a document store quite similar to a key-value store, but the only difference is that the values stored (referred to as "documents") provide some structure and encoding of the managed data. XML, JSON (Java Script Object Notation), BSON (which is a binary encoding of JSON objects) are some common standard encodings.

One key difference between a key-value store and a document store is that the latter embeds attribute metadata associated with stored content, which essentially provides a way to query the data based on the contents. For example, in the above example, one could search for all documents in which "City" is "Noida" that would deliver a result set containing all documents associated with any "3Pillar Office" that is in that particular city.

Apache CouchDB is an example of a document store. CouchDB uses JSON to store data, JavaScript as its query language using MapReduce and HTTP for an API. Data and relationships are not stored in tables as is a norm with conventional relational databases but in fact are a collection of independent documents.

The fact that document style databases are schema-less makes adding fields to JSON documents a simple task without having to define changes first.

- Couchbase and MongoDB are the most popular document based databases.

3. **Column Store NoSQL Database**–

In column-oriented NoSQL database, data is stored in cells grouped in columns of data rather than as rows of data. Columns are logically grouped into column families. Column families can contain a virtually unlimited number of columns that can be created at runtime or the definition of the schema. Read and write is done using columns rather than rows.

In comparison, most relational DBMS store data in rows, the benefit of storing data in columns, is fast search/ access and data aggregation. Relational databases store a single row as a continuous disk entry. Different rows are stored in different places on disk while Columnar databases store all the cells corresponding to a column as a continuous disk entry thus makes the search/access faster.

For example: To query the titles from a bunch of a million articles will be a painstaking task while using relational databases as it will go over each location to get item titles. On the other hand, with just one disk access, title of all the items can be obtained.

Data Model

- **ColumnFamily**: ColumnFamily is a single structure that can group Columns and SuperColumns with ease.
- **Key**: the permanent name of the record. Keys have different numbers of columns, so the database can scale in an irregular way.
- **Keyspace**: This defines the outermost level of an organization, typically the name of the application. For example, '3PillarDataBase' (database name).
- **Column**: It has an ordered list of elements aka tuple with a name and a value defined.

The best known examples are Google's BigTable and HBase & Cassandra that were inspired from BigTable.

BigTable, for instance is a high performance, compressed and proprietary data storage system owned by Google. It has the following attributes:

- **Sparse** – some cells can be empty
- **Distributed** – data is partitioned across many hosts
- **Persistent** – stored to disk
- **Multidimensional** – more than 1 dimension
- **Map** – key and value
- **Sorted** – maps are generally not sorted but this one is

A 2-dimensional table comprising of rows and columns is part of the relational database system.

City	Pincode	Strength	Project
Noida	201301	250	20
Cluj	400606	200	15
Timisoara	300011	150	10
Fairfax	VA 22033	100	5

For above RDBMS table a BigTable map can be visualized as shown below.

{

```
3PillarNoida: {
city: Noida
pincode: 201301
},
details: {
strength: 250
projects: 20
}
}
{
3PillarCluj: {
address: {
city: Cluj
pincode: 400606
},
details: {
strength: 200
projects: 15
}
},
{
3PillarTimisoara: {
address: {
city: Timisoara
pincode: 300011
},
details: {
strength: 150
projects: 10
}
}
{
3PillarFairfax : {
address: {
city: Fairfax
pincode: VA 22033
},
details: {
strength: 100
projects: 5
}
}
```

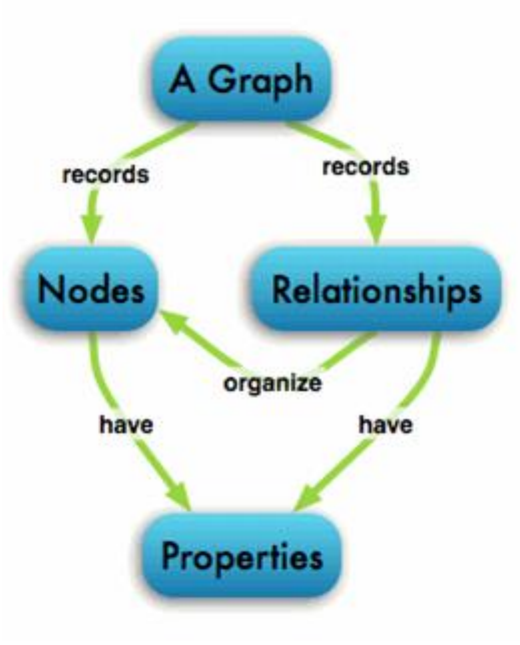
- The outermost keys 3PillarNoida, 3PillarCluj, 3PillarTimisoara and 3PillarFairfax are analogues to rows.
- ‘address’ and ‘details’ are called **column families**.
- The column-family ‘address’ has **columns** ‘city’ and ‘pincode’.
- The column-family details’ has **columns** ‘strength’ and ‘projects’.

Columns can be referenced using CloumnFamily.

- Google’s BigTable, HBase and Cassandra are the most popular column store based databases.

4. **Graph Base NoSQL Database**

In a Graph Base NoSQL Database, you will not find the rigid format of SQL or the tables and columns representation, a flexible graphical representation is instead used which is perfect to address scalability concerns. Graph structures are used with edges, nodes and properties which provides index-free adjacency. Data can be easily transformed from one model to the other using a Graph Base NoSQL database.

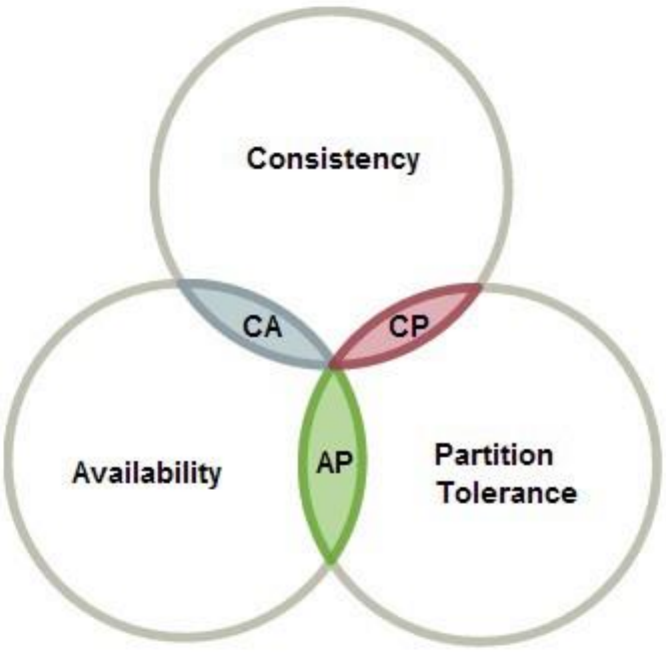


- These databases that uses edges and nodes to represent and store data.
- These nodes are organised by some relationships with one another, which is represented by edges between the nodes.
- Both the nodes and the relationships have some defined properties.

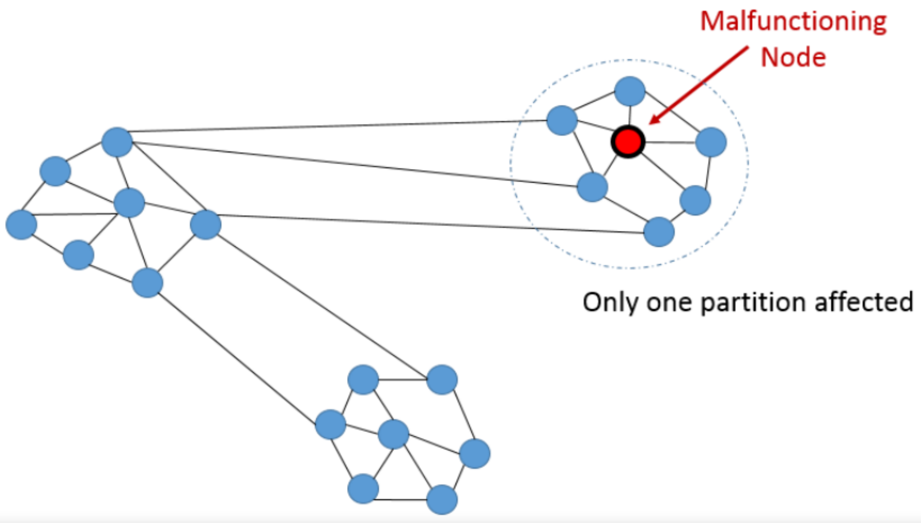
Examples: AllegroGraph, Neo4j, IBM Graph

- ➔ CAP Theorem:

  CAP Theorem is a concept that a distributed database system can only have 2 of the 3: Consistency, Availability and Partition Tolerance.

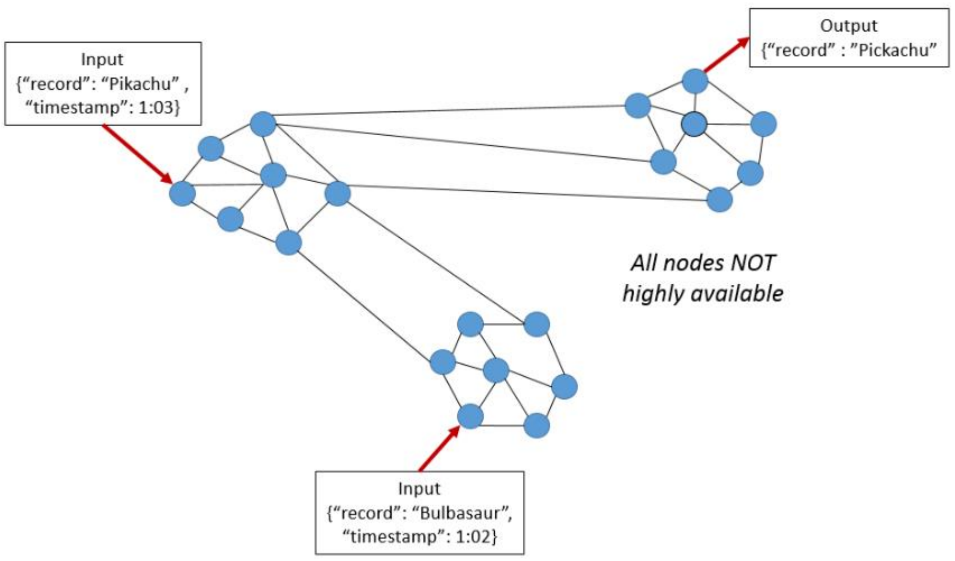


Partition Tolerance



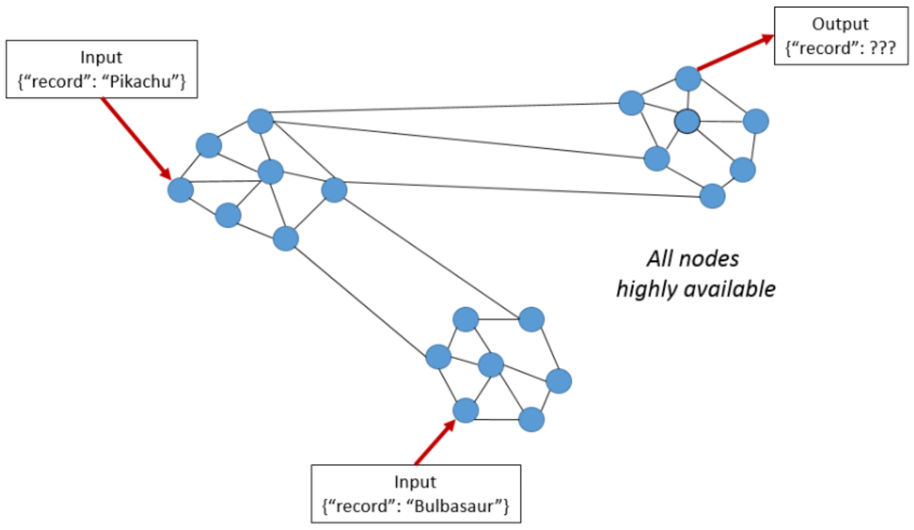
This condition states that the system continues to run, despite the number of messages being delayed by the network between nodes. A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data records are sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages. When dealing with modern distributed systems, Partition Tolerance is not an option. It's a necessity. Hence, we have to trade between Consistency and Availability.

High Consistency


Input
{“record”: “Pikachu” , “timestamp”: 1:03}
Output
{“record” : ”Pickachu”
All nodes NOT highly available
Input
{“record”: “Bulbasaur”, “timestamp”: 1:02}

This condition states that all data see the same data at the same time. Simply put, performing a *read* operation will return the value of the most recent *write*operation causing all nodes to return the same data. A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state. In this model, a system can (and does) shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process. In the image, we have 2 different records (“Bulbasaur” and “Pikachu”) at different timestamps. The output on the third partition is “Pikachu”, the latest input. However, the nodes will need time to update and will not be Available on the network as often.

High Availability



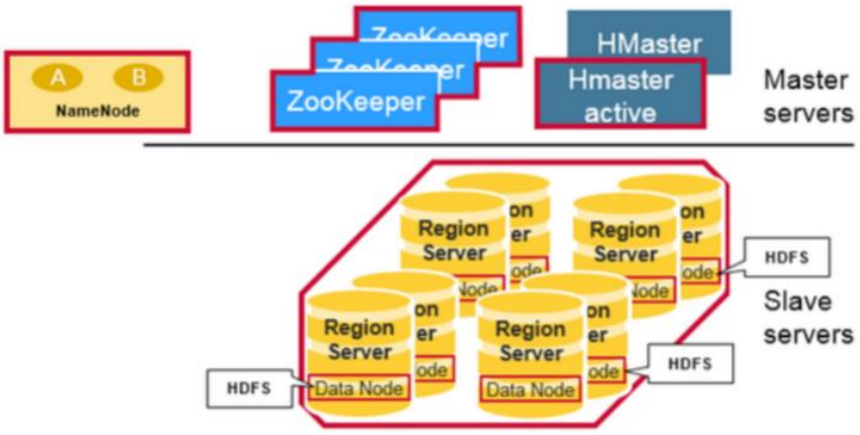
This condition states that every request gets a response on success/failure. Achieving availability in a distributed system requires that the system remains operational 100% of the time. Every client gets a response, regardless of the state of any individual node in the system. This metric is trivial to measure: either you can submit read/write commands, or you cannot. Hence, the databases are time independent as the nodes need to be available online at all times. This means that, unlike the previous example, we do not know if "Pikachu" or "Bulbasaur" was added first. The output could be either one. Hence why, high availability isn't feasible when analyzing streaming data at high frequency.

HBase Architecture:

Physically, HBase is composed of three types of servers in a master slave type of architecture. Region servers serve data for reads and writes. When accessing data, clients communicate with HBase RegionServers directly. Region assignment, DDL (create, delete tables) operations are handled by the HBase Master process. Zookeeper, which is part of HDFS, maintains a live cluster state.

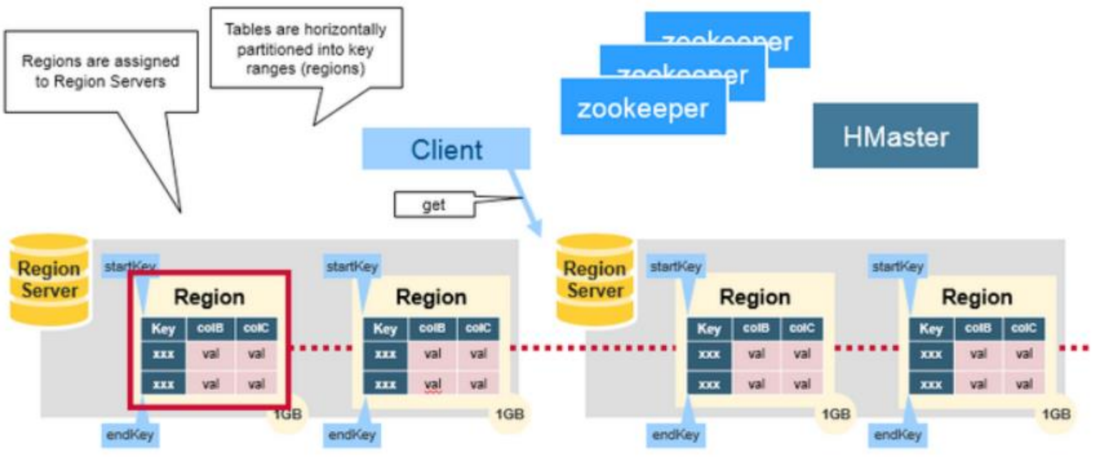
The Hadoop DataNode stores the data that the Region Server is managing. All HBase data is stored in HDFS files. Region Servers are collocated with the HDFS DataNodes, which enable data locality (putting the data close to where it is needed) for the data served by the RegionServers. HBase data is local when it is written, but when a region is moved, it is not local until compaction.

The NameNode maintains metadata information for all the physical data blocks that comprise the files.



Regions

HBase Tables are divided horizontally by row key range into “Regions.” A region contains all rows in the table between the region’s start key and end key. Regions are assigned to the nodes in the cluster, called “Region Servers,” and these serve data for reads and writes. A region server can serve about 1,000 regions.


Regions are assigned to Region Servers
Tables are horizontally partitioned into key ranges (regions)
Client
get
zookeeper
zookeeper
zookeeper
HMaster
Region Server
startKey
Region
Key
colB
colC
xxx
val
1GB
endKey

HBase HMaster

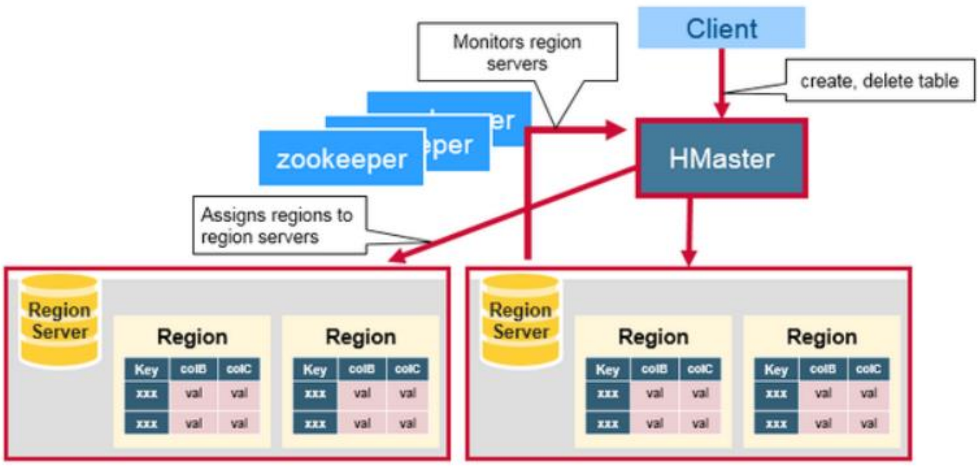
Region assignment, DDL (create, delete tables) operations are handled by the HBase Master.

A master is responsible for:

- Coordinating the region servers
  - Assigning regions on startup , re-assigning regions for recovery or load balancing

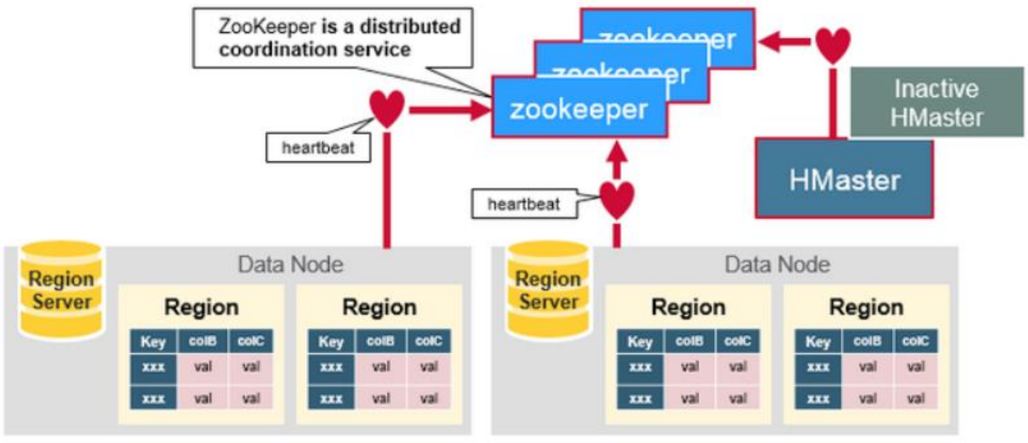
- Monitoring all RegionServer instances in the cluster (listens for notifications from zookeeper)

- Admin functions
  - Interface for creating, deleting, updating tables



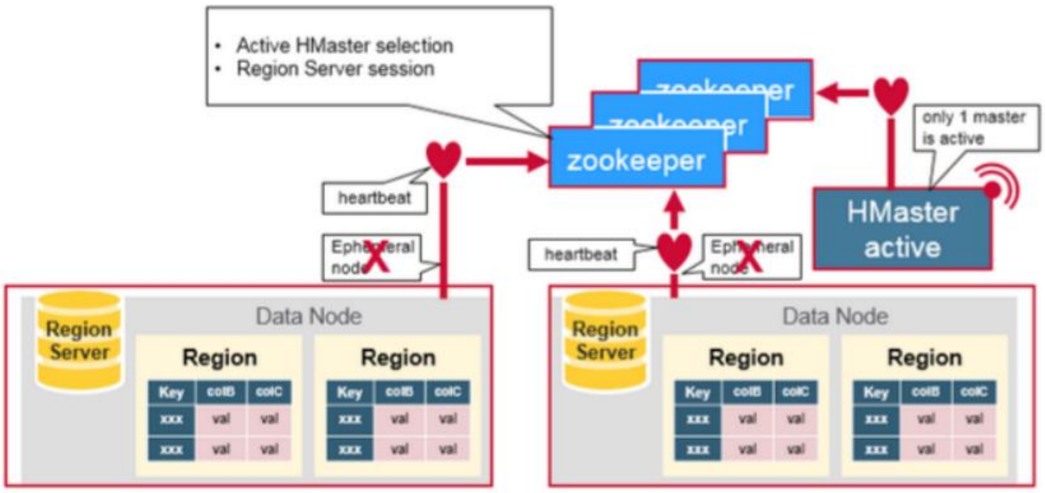
ZooKeeper: The Coordinator

HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available, and provides server failure notification. Zookeeper uses consensus to guarantee common shared state. Note that there should be three or five machines for consensus.



How the Components Work Together

Zookeeper is used to coordinate shared state information for members of distributed systems. Region servers and the active HMaster connect with a session to ZooKeeper. The ZooKeeper maintains ephemeral nodes for active sessions via heartbeats.



Each Region Server creates an ephemeral node. The HMaster monitors these nodes to discover available region servers, and it also monitors these nodes for server failures. HMasters vie to create an ephemeral node. Zookeeper determines the first one and uses it to make sure that only one master is active. The active HMaster sends heartbeats to Zookeeper, and the inactive HMaster listens for notifications of the active HMaster failure.

If a region server or the active HMaster fails to send a heartbeat, the session is expired and the corresponding ephemeral node is deleted. Listeners for updates will be notified of the deleted nodes. The active HMaster listens for region servers, and will recover region servers on failure. The Inactive HMaster listens for active HMaster failure, and if an active HMaster fails, the inactive HMaster becomes active.

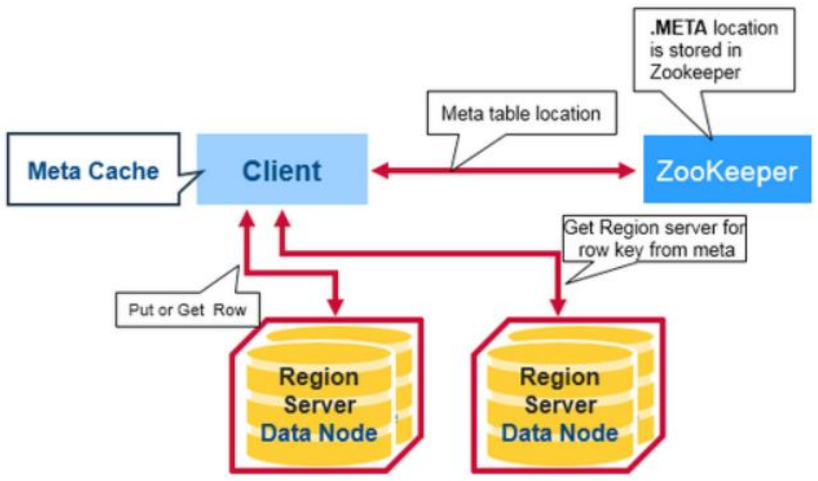
HBase First Read or Write

There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster. ZooKeeper stores the location of the META table.

This is what happens the first time a client reads or writes to HBase:

1. The client gets the Region server that hosts the META table from ZooKeeper.
2. The client will query the .META. server to get the region server corresponding to the row key it wants to access. The client caches this information along with the META table location.
3. It will get the Row from the corresponding Region Server.

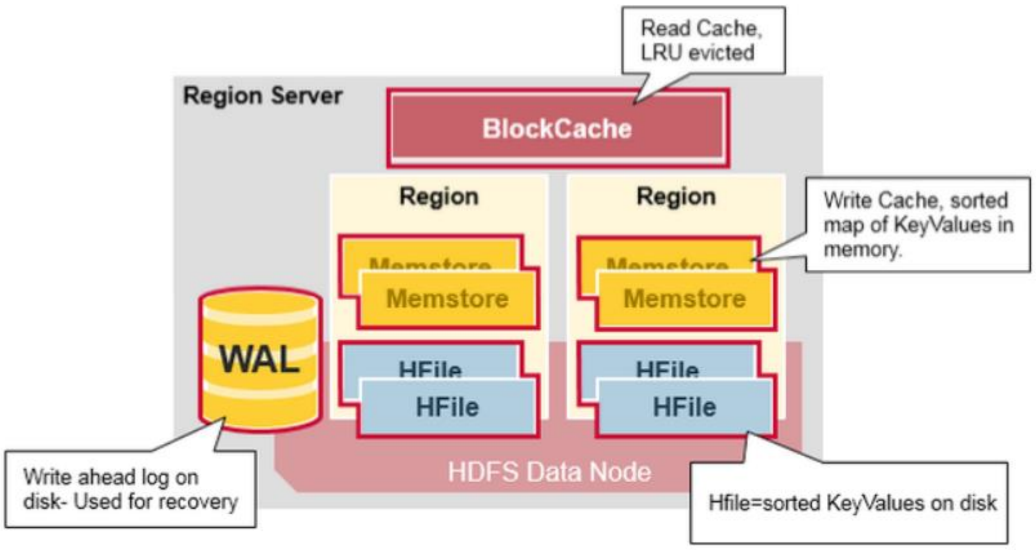
For future reads, the client uses the cache to retrieve the META location and previously read row keys. Over time, it does not need to query the META table, unless there is a miss because a region has moved; then it will re-query and update the cache.



Region Server Components

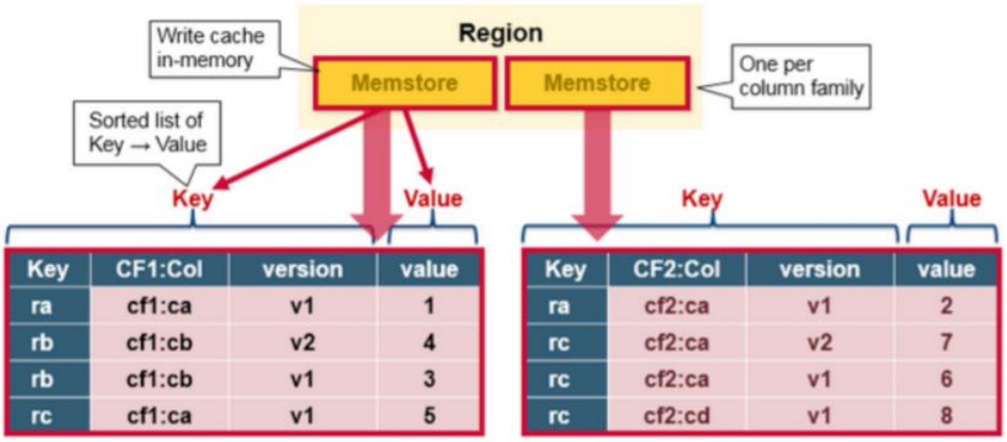
A Region Server runs on an HDFS data node and has the following components:

- WAL: Write Ahead Log is a file on the distributed file system. The WAL is used to store new data that hasn't yet been persisted to permanent storage; it is used for recovery in the case of failure.
- BlockCache: is the read cache. It stores frequently read data in memory. Least Recently Used data is evicted when full.
- MemStore: is the write cache. It stores new data which has not yet been written to disk. It is sorted before writing to disk. There is one MemStore per column family per region.
- Hfiles store the rows as sorted KeyValues on disk.



HBase MemStore

The MemStore stores updates in memory as sorted KeyValues, the same as it would be stored in an HFile. There is one MemStore per column family. The updates are sorted per column family.



Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

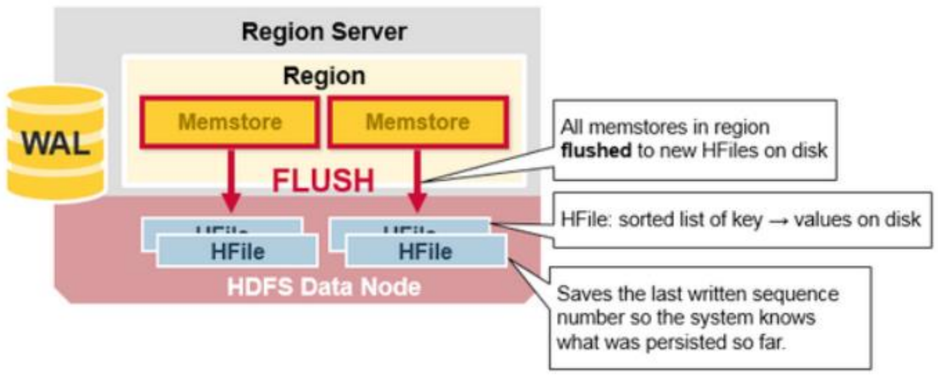
Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

HBase Region Flush

When the MemStore accumulates enough data, the entire sorted set is written to a new HFile in HDFS. HBase uses multiple HFiles per column family, which contain the actual cells, or KeyValue instances. These files are created over time as KeyValue edits sorted in the MemStores are flushed as files to disk.

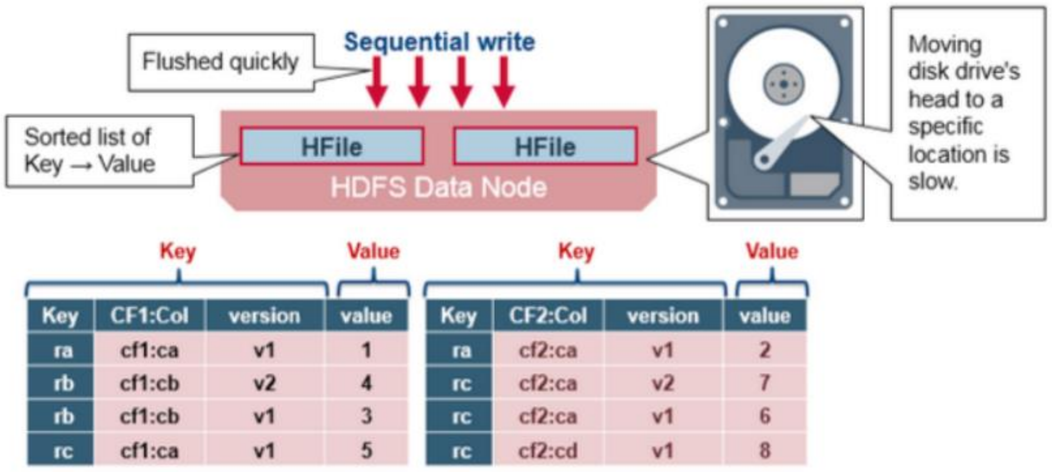
Note that this is one reason why there is a limit to the number of column families in HBase. There is one MemStore per CF; when one is full, they all flush. It also saves the last written sequence number so the system knows what was persisted so far.

The highest sequence number is stored as a meta field in each HFile, to reflect where persisting has ended and where to continue. On region startup, the sequence number is read, and the highest is used as the sequence number for new edits.


Region Server
Region
WAL
Memstore
Memstore
FLUSH
HFile
HFile
HDFS Data Node
All memstores in region **flushed** to new HFiles on disk
HFile: sorted list of key → values on disk
Saves the last written sequence number so the system knows what was persisted so far.

HBase HFile

Data is stored in an HFile which contains sorted key/values. When the MemStore accumulates enough data, the entire sorted KeyValue set is written to a new HFile in HDFS. This is a sequential write. It is very fast, as it avoids moving the disk drive head.


Flushed quickly
Sequential write
Sorted list of Key → Value
HFile
HFile
HDFS Data Node
Moving disk drive's head to a specific location is slow.
Key
Value
Key
Value

Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

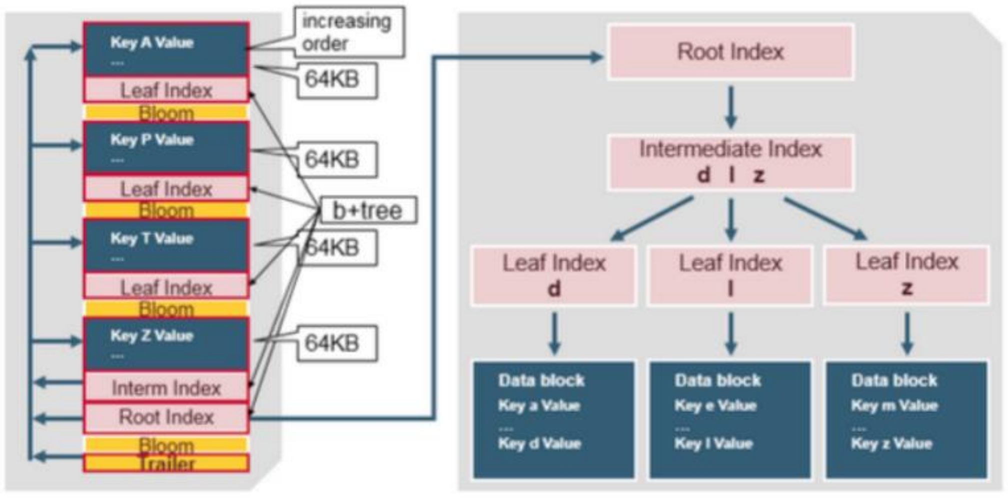
Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

HBase HFile Structure

An HFile contains a multi-layered index which allows HBase to seek to the data without having to read the whole file. The multi-level index is like a b+tree:

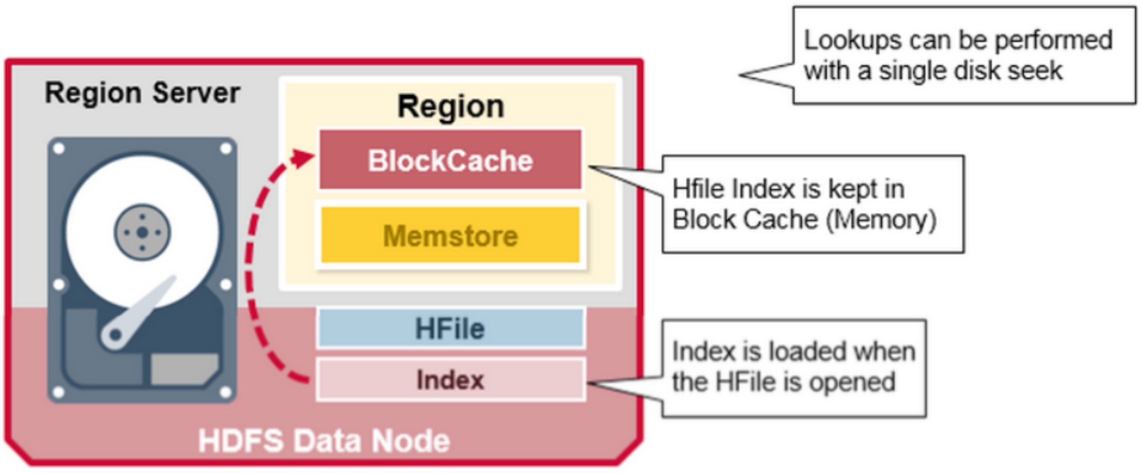
- Key value pairs are stored in increasing order
- Indexes point by row key to the key value data in 64KB “blocks”
- Each block has its own leaf-index
- The last key of each block is put in the intermediate index
- The root index points to the intermediate index

The trailer points to the meta blocks, and is written at the end of persisting the data to the file. The trailer also has information like bloom filters and time range info. Bloom filters help to skip files that do not contain a certain row key. The time range info is useful for skipping the file if it is not in the time range the read is looking for.



HFile Index

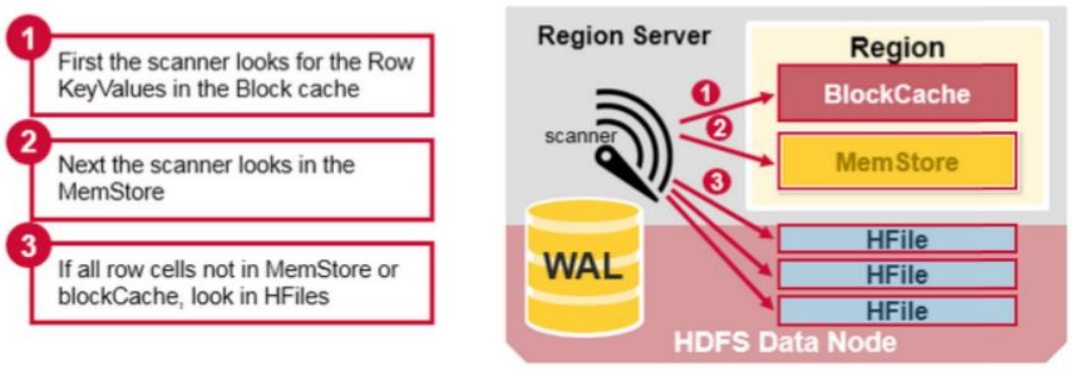
The index, which we just discussed, is loaded when the HFile is opened and kept in memory. This allows lookups to be performed with a single disk seek.


Lookups can be performed with a single disk seek
Region Server
Region
BlockCache
Memstore
HFile
Index
HDFS Data Node
Hfile Index is kept in Block Cache (Memory)
Index is loaded when the HFile is opened

HBase Read Merge

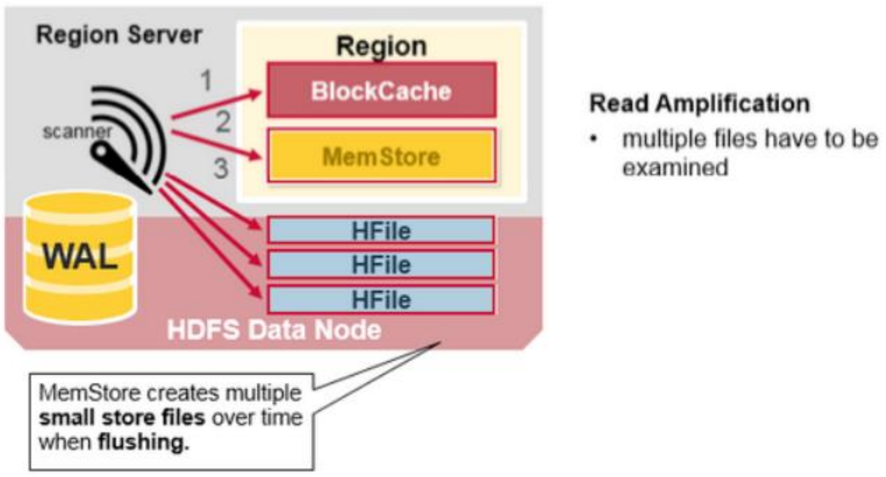
We have seen that the KeyValue cells corresponding to one row can be in multiple places, row cells already persisted are in Hfiles, recently updated cells are in the MemStore, and recently read cells are in the Block cache. So when you read a row, how does the system get the corresponding cells to return? A Read merges Key Values from the block cache, MemStore, and HFiles in the following steps:

1. First, the scanner looks for the Row cells in the Block cache - the read cache. Recently Read Key Values are cached here, and Least Recently Used are evicted when memory is needed.
2. Next, the scanner looks in the MemStore, the write cache in memory containing the most recent writes.
3. If the scanner does not find all of the row cells in the MemStore and Block Cache, then HBase will use the Block Cache indexes and bloom filters to load HFiles into memory, which may contain the target row cells.


1
First the scanner looks for the Row KeyValues in the Block cache
2
Next the scanner looks in the MemStore
3
If all row cells not in MemStore or blockCache, look in HFiles
Region Server
Region
BlockCache
MemStore
scanner
WAL
HFile
HFile
HFile
HDFS Data Node

HBase Read Merge

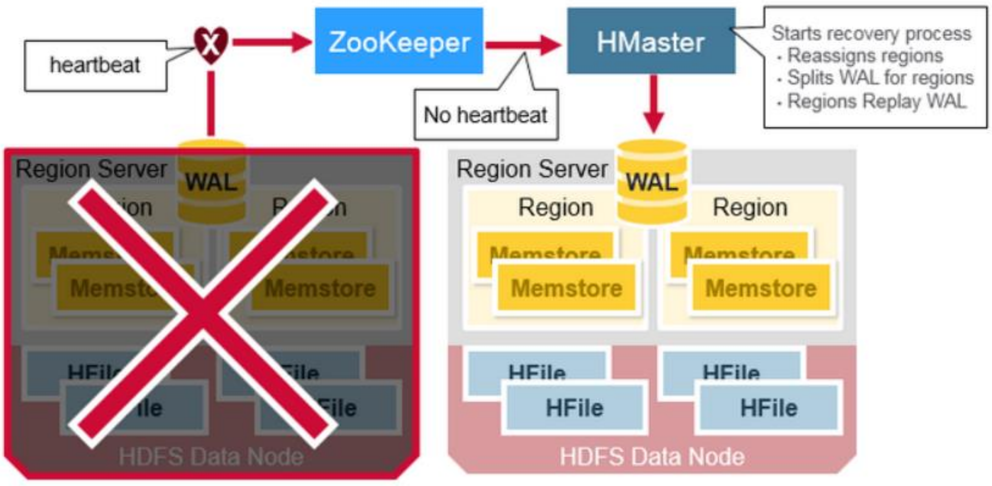
As discussed earlier, there may be many HFiles per MemStore, which means for a read, multiple files may have to be examined, which can affect the performance. This is called read amplification.


Region Server
Region
BlockCache
MemStore
1
2
3
scanner
WAL
HFile
HFile
HFile
HDFS Data Node
MemStore creates multiple small store files over time when flushing.
Read Amplification
multiple files have to be examined

HBase Crash Recovery

When a RegionServer fails, Crashed Regions are unavailable until detection and recovery steps have happened. Zookeeper will determine Node failure when it loses region server heart beats. The HMaster will then be notified that the Region Server has failed.

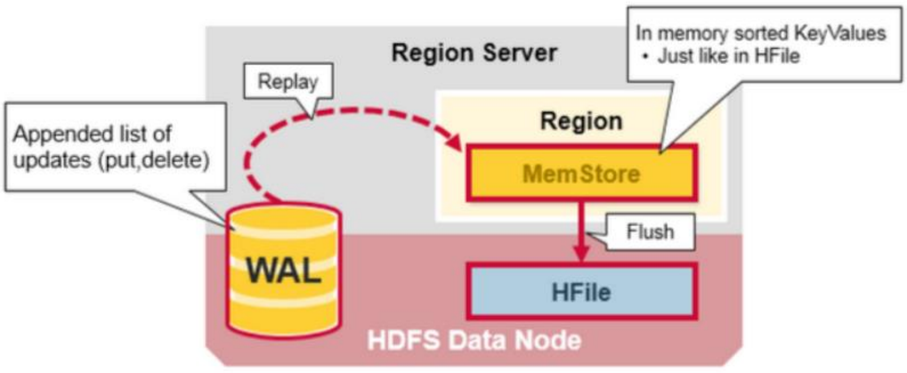
When the HMaster detects that a region server has crashed, the HMaster reassigns the regions from the crashed server to active Region servers. In order to recover the crashed region server's memstore edits that were not flushed to disk. The HMaster splits the WAL belonging to the crashed region server into separate files and stores these file in the new region servers' data nodes. Each Region Server then replays the WAL from the respective split WAL, to rebuild the memstore for that region.



Data Recovery

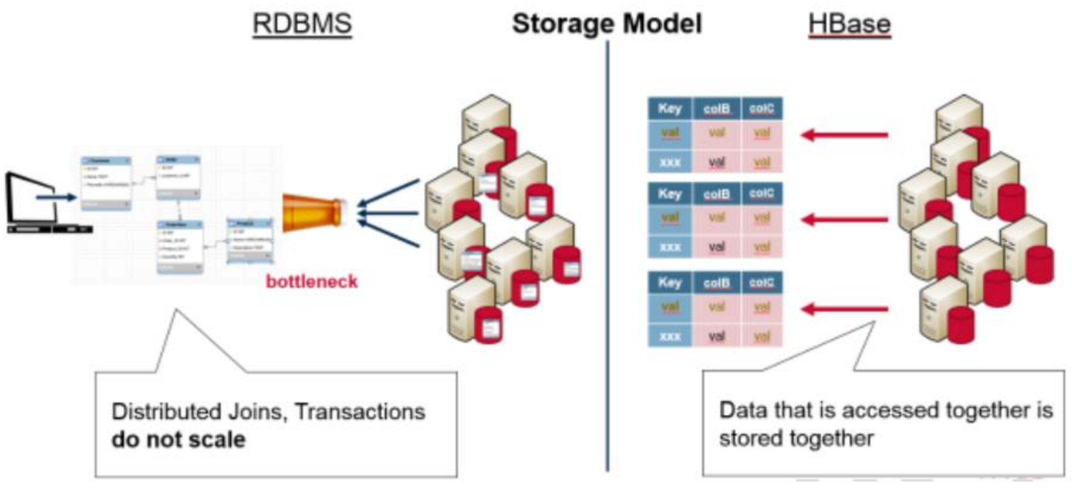
WAL files contain a list of edits, with one edit representing a single put or delete. Edits are written chronologically, so, for persistence, additions are appended to the end of the WAL file that is stored on disk.

What happens if there is a failure when the data is still in memory and not persisted to an HFile? The WAL is replayed. Replaying a WAL is done by reading the WAL, adding and sorting the contained edits to the current MemStore. At the end, the MemStore is flush to write changes to an HFile.



HBase vs RDBMS

H Base	RDBMS
1. Column-oriented	1. Row-oriented(mostly)
2. Flexible schema, add columns on the Fly	2. Fixed schema
3. Good with sparse tables.	3. Not optimized for sparse tables.
4. No query language	4. SQL
5. Wide tables	5. Narrow tables
6. Joins using MR – not optimized	6. optimized for Joins(small, fast ones)
7. Tight – Integration with MR	7. Not really
8. De-normalize your data.	8. Normalize as you can
9. Horizontal scalability-just add hard war.	9. Hard to share and scale.
10. Consistent	10. Consistent
11. No transactions.	11. transactional
12. Good for semi-structured data as well as structured data.	12. Good for structured data.


RDBMS
Storage Model
HBase
bottleneck
Key colB colC
val val val
xxx val val
Distributed Joins, Transactions do not scale
Data that is accessed together is stored together

Task2

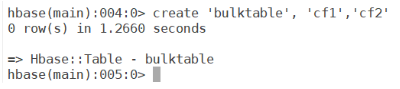
Execute blog present in below link

https://acadgild.com/blog/importtsv-data-from-hdfs-into-hbase/

Step1:

Inside Hbase shell give the following command to create table along with 2 column family.

Create 'bulktable', 'cf1', 'cf2'



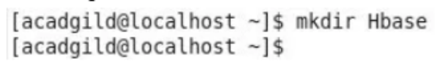
```
hbase(main):004:0> create 'bulktable', 'cf1','cf2'
0 row(s) in 1.2660 seconds

=> Hbase::Table - bulktable
hbase(main):005:0> 
```

Step2 :

Come out of HBase shell to the terminal and also make a directory for Hbase in the local drive; So,

since you have your own path you can use it.

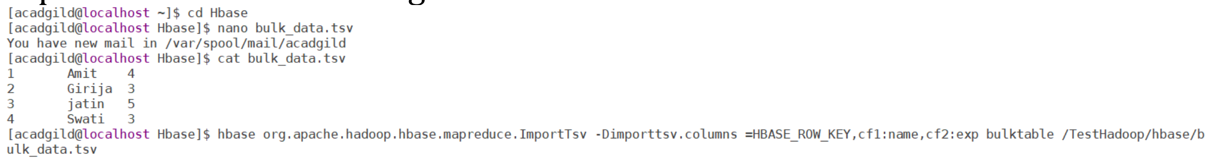
mkdir -p hbase



```
[acadgild@localhost ~]$ mkdir Hbase
[acadgild@localhost ~]$
```

Step3:

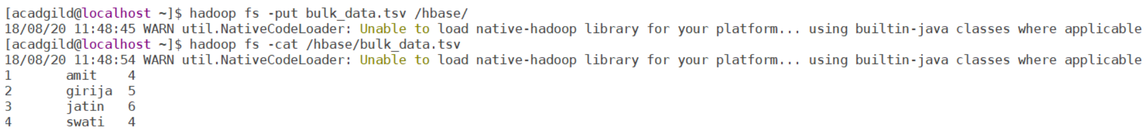
Create a file inside the HBase directory named bulk_data.tsv with tab separated data inside using below command in terminal.



```
[acadgild@localhost ~]$ cd Hbase
[acadgild@localhost Hbase]$ nano bulk_data.tsv
You have new mail in /var/spool/mail/acadgild
[acadgild@localhost Hbase]$ cat bulk_data.tsv
1       Amit    4
2       Girija  3
3       jatin   5
4       Swati   3
[acadgild@localhost Hbase]$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.columns =HBASE_ROW_KEY,cf1:name,cf2:exp bulktable /TestHadoop/hbase/b
ulk_data.tsv
```

Step4:

Our data should be present in HDFS while performing the import task to Hbase.
In real time projects, the data will already be present inside HDFS.
Here for our learning purpose, we copy the data inside HDFS using below commands in terminal.
Command: **hadoop fs -mkdir /hbase**



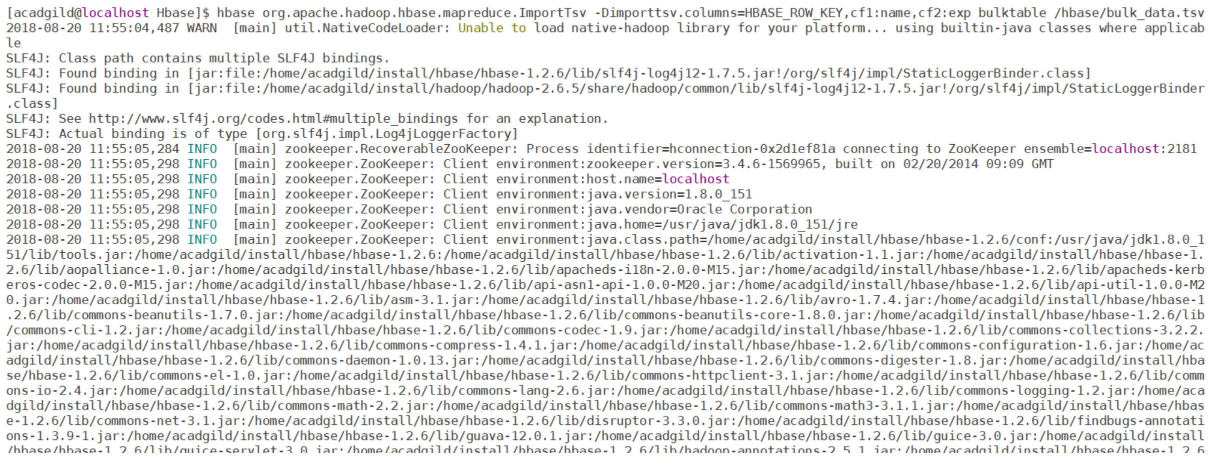
```
[acadgild@localhost ~]$ hadoop fs -put bulk_data.tsv /hbase/
18/08/20 11:48:45 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
[acadgild@localhost ~]$ hadoop fs -cat /hbase/bulk_data.tsv
18/08/20 11:48:54 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
1       amit    4
2       girija  5
3       jatin   6
4       swati   4
```

Step5:

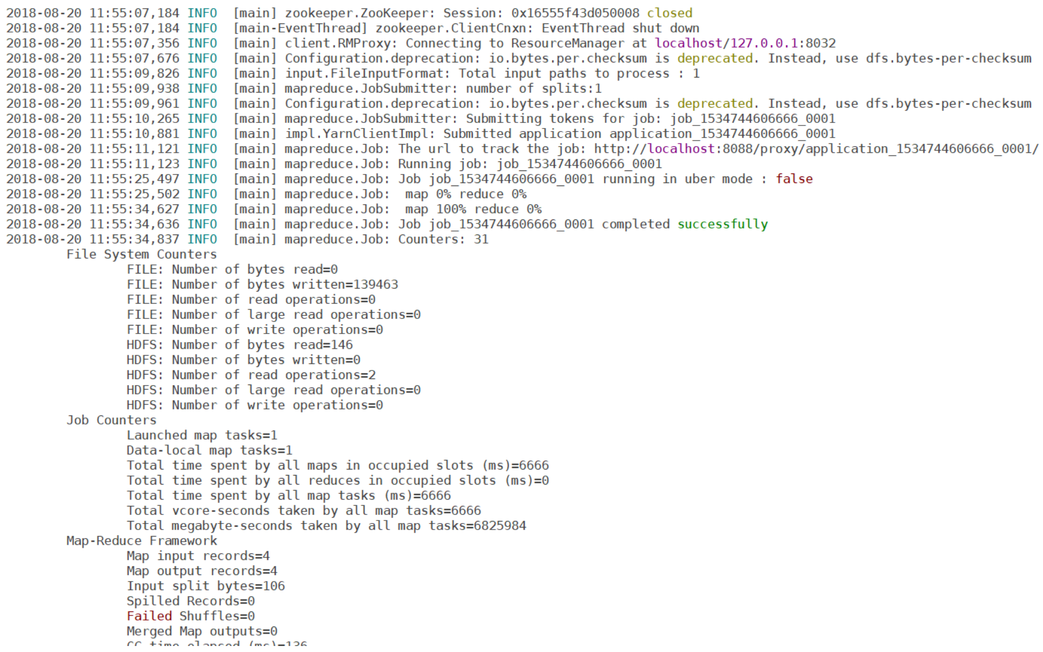
After the data is present now in HDFS.In terminal, we give the following command along with arguments<tablename> and <path of data in HDFS>

Command:

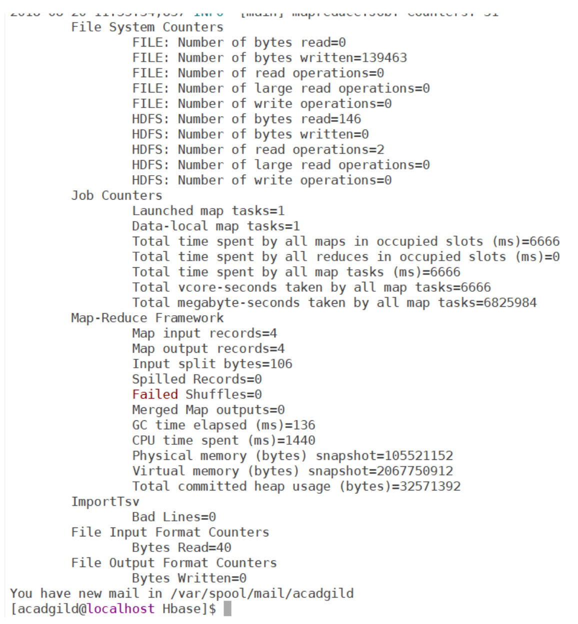
hbase org.apache.hadoop.hbase.mapreduce.ImportTsv –Dimporttsv.columns=HBASE_ROW_KEY,cf1:name,cf2:exp bulktable /hbase/bulk_data.tsv



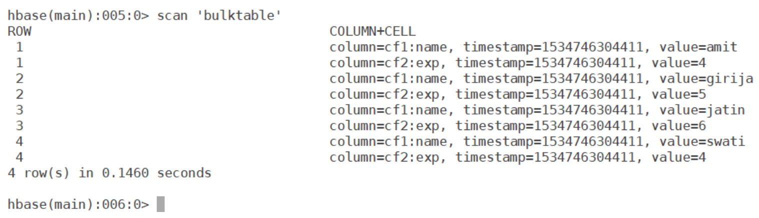
```
[acadgild@localhost Hbase]$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.columns=HBASE_ROW_KEY,cf1:name,cf2:exp bulktable /hbase/bulk_data.tsv
2018-08-20 11:55:04,487 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicab
le
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/acadgild/install/hbase/hbase-1.2.6/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/acadgild/install/hadoop/hadoop-2.6.5/share/hadoop/common/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder
.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2018-08-20 11:55:05,284 INFO  [main] zookeeper.RecoverableZooKeeper: Process identifier=hconnection-0x2d1ef81a connecting to ZooKeeper ensemble=localhost:2181
2018-08-20 11:55:05,298 INFO  [main] zookeeper.ZooKeeper: Client environment:zookeeper.version=3.4.6-1569965, built on 02/20/2014 09:09 GMT
2018-08-20 11:55:05,298 INFO  [main] zookeeper.ZooKeeper: Client environment:host.name=localhost
2018-08-20 11:55:05,298 INFO  [main] zookeeper.ZooKeeper: Client environment:java.version=1.8.0_151
2018-08-20 11:55:05,298 INFO  [main] zookeeper.ZooKeeper: Client environment:java.vendor=Oracle Corporation
2018-08-20 11:55:05,298 INFO  [main] zookeeper.ZooKeeper: Client environment:java.home=/usr/java/jdk1.8.0_151/jre
2018-08-20 11:55:05,298 INFO  [main] zookeeper.ZooKeeper: Client environment:java.class.path=/home/acadgild/install/hbase/hbase-1.2.6/conf:/usr/java/jdk1.8.0_1
51/lib/tools.jar:/home/acadgild/install/hbase/hbase-1.2.6:/home/acadgild/install/hbase/hbase-1.2.6/lib/activation-1.1.jar:/home/acadgild/install/hbase/hbase-1.
2.6/lib/aopalliance-1.0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/apacheds-i18n-2.0.0-M15.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/apacheds-kerb
eros-codec-2.0.0-M15.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/api-asn1-api-1.0.0-M20.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/api-util-1.0.0-M2
0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/asm-3.1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/avro-1.7.4.jar:/home/acadgild/install/hbase/hbase-1
.2.6/lib/commons-beanutils-1.7.0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-beanutils-core-1.8.0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib
/commons-cli-1.2.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-codec-1.9.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-collections-3.2.2.
jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-compress-1.4.1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-configuration-1.6.jar:/home/ac
adgild/install/hbase/hbase-1.2.6/lib/commons-daemon-1.0.13.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-digester-1.8.jar:/home/acadgild/install/hba
se/hbase-1.2.6/lib/commons-el-1.0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-httpclient-3.1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/comm
ons-io-2.4.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-lang-2.6.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-logging-1.2.jar:/home/aca
dgild/install/hbase/hbase-1.2.6/lib/commons-math-2.2.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-math3-3.1.1.jar:/home/acadgild/install/hbase/hbas
e-1.2.6/lib/commons-net-3.1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/disruptor-3.3.0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/findbugs-annotati
ons-1.3.9-1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/guava-12.0.1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/guice-3.0.jar:/home/acadgild/install
/hbase/hbase-1.2.6/lib/guice-servlet-3.0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/hadoop-annotations-2.5.1.jar:/home/acadgild/install/hbase/hbase-1.2.6
```



```
2018-08-20 11:55:07,184 INFO  [main] zookeeper.ZooKeeper: Session: 0x16555f43d050008 closed
2018-08-20 11:55:07,184 INFO  [main-EventThread] zookeeper.ClientCnxn: EventThread shut down
2018-08-20 11:55:07,356 INFO  [main] client.RMProxy: Connecting to ResourceManager at localhost/127.0.0.1:8032
2018-08-20 11:55:07,676 INFO  [main] Configuration.deprecation: io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
2018-08-20 11:55:09,826 INFO  [main] input.FileInputFormat: Total input paths to process : 1
2018-08-20 11:55:09,938 INFO  [main] mapreduce.JobSubmitter: number of splits:1
2018-08-20 11:55:09,961 INFO  [main] Configuration.deprecation: io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
2018-08-20 11:55:10,265 INFO  [main] mapreduce.JobSubmitter: Submitting tokens for job: job_1534744606666_0001
2018-08-20 11:55:10,881 INFO  [main] impl.YarnClientImpl: Submitted application application_1534744606666_0001
2018-08-20 11:55:11,121 INFO  [main] mapreduce.Job: The url to track the job: http://localhost:8088/proxy/application_1534744606666_0001/
2018-08-20 11:55:11,123 INFO  [main] mapreduce.Job: Running job: job_1534744606666_0001
2018-08-20 11:55:25,497 INFO  [main] mapreduce.Job: Job job_1534744606666_0001 running in uber mode : false
2018-08-20 11:55:25,502 INFO  [main] mapreduce.Job:  map 0% reduce 0%
2018-08-20 11:55:34,627 INFO  [main] mapreduce.Job:  map 100% reduce 0%
2018-08-20 11:55:34,636 INFO  [main] mapreduce.Job: Job job_1534744606666_0001 completed successfully
2018-08-20 11:55:34,837 INFO  [main] mapreduce.Job: Counters: 31
        File System Counters
                FILE: Number of bytes read=0
                FILE: Number of bytes written=139463
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=146
                HDFS: Number of bytes written=0
                HDFS: Number of read operations=2
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=0
        Job Counters
                Launched map tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=6666
                Total time spent by all reduces in occupied slots (ms)=0
                Total time spent by all map tasks (ms)=6666
                Total vcore-seconds taken by all map tasks=6666
                Total megabyte-seconds taken by all map tasks=6825984
        Map-Reduce Framework
                Map input records=4
                Map output records=4
                Input split bytes=106
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=136
```



```
2018-08-20 11:55:34,837 INFO  [main] mapreduce.Job: Counters: 31
        File System Counters
                FILE: Number of bytes read=0
                FILE: Number of bytes written=139463
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=146
                HDFS: Number of bytes written=0
                HDFS: Number of read operations=2
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=0
        Job Counters
                Launched map tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=6666
                Total time spent by all reduces in occupied slots (ms)=0
                Total time spent by all map tasks (ms)=6666
                Total vcore-seconds taken by all map tasks=6666
                Total megabyte-seconds taken by all map tasks=6825984
        Map-Reduce Framework
                Map input records=4
                Map output records=4
                Input split bytes=106
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=136
                CPU time spent (ms)=1440
                Physical memory (bytes) snapshot=105521152
                Virtual memory (bytes) snapshot=2067750912
                Total committed heap usage (bytes)=32571392
        ImportTsv
                Bad Lines=0
        File Input Format Counters
                Bytes Read=40
        File Output Format Counters
                Bytes Written=0
You have new mail in /var/spool/mail/acadgild
[acadgild@localhost Hbase]$
```



```
hbase(main):005:0> scan 'bulktable'
ROW                                      COLUMN+CELL
 1                                       column=cf1:name, timestamp=1534746304411, value=amit
 1                                       column=cf2:exp, timestamp=1534746304411, value=4
 2                                       column=cf1:name, timestamp=1534746304411, value=girija
 2                                       column=cf2:exp, timestamp=1534746304411, value=5
 3                                       column=cf1:name, timestamp=1534746304411, value=jatin
 3                                       column=cf2:exp, timestamp=1534746304411, value=6
 4                                       column=cf1:name, timestamp=1534746304411, value=swati
 4                                       column=cf2:exp, timestamp=1534746304411, value=4
4 row(s) in 0.1460 seconds

hbase(main):006:0>
```

We see all the data are present in the table, thus confirming our mapping successful for tab separated values.

Notes: Hfiles of data to prepare for a bulk data load , we pass the option:

-**Dimporttsv.bulk.output**=/path/for/output

The target table will be created with default column family descriptors if it does not already exist Otheroptions that may be specified with -D include:

-**Dimporttsv.skip.bad.lines**=false – fail if encountering an invalid line

'-**Dimporttsv.separator**=|' – eg separate on pipes instead of tabs

-**Dimporttsv.timestamp=**currentTimeAsLong –use the specified timestamp for

the import.

-**Dimporttsv.mapper.class**=my.Mapper – A user-defined Mapper to use

instead of org.apache.hadoop.hbase.mapreduce.TsvImporterMapper