Task1

Explain the below concepts with an example in brief.

●Nosql Databases

NoSQL Database is used to refer a non-SQL or non relational database.

It provides a mechanism for storage and retrieval of data other than tabular relations model used in relational databases. NoSQL database doesn't use tables for storing data. It is generally used to store big data and real-time web applications.

NoSQL refers to all databases and data stores that are not based on the Relational Database Management Systems or RDBMS principles. It relates to large data sets accessed and manipulated on a Web scale. NoSQL does not represent single product or technology. It represents a group of products and a various related data concepts for storage and management.

Database Features of NoSQL

NoSQL Databases can have a common set of features such as:

- Non-relational data model.
- Runs well on clusters.
- Mostly open-source.
- Built for the new generation Web applications.
- Is schema-less.

● Types of Nosql Databases

There are four basic types of NoSQL databases.

Key-Value database

Key-Value database has a big hash table of keys and values. Riak (Pronounce as REE-awk), Tokyo Cabinet, Redis server, Memcached ((Pronounce as mem-cached), and Scalaris are examples of a key-value store.

The key-value part refers to the fact that the database stores data as a collection of key/value pairs. This is a simple method of storing data, and it is known to scale well.

The key-value pair is a well established concept in many programming languages. Programming languages typically refer to a key-value as an associative array or data structure. A key-value is also commonly referred to as a dictionary or hash.

Examples of Key-Value Stores
Below are examples of key-value stores.
These are simple examples, but the aim is to provide an idea of the how a key-value database works.
Phone Directory

Key	Value
Bob	(123) 456-7890
Jane	(234) 567-8901
Tara	(345) 678-9012
Tiara	(456) 789-0123

Document-based database

A **document-oriented database** is a designed for storing, retrieving, and managing document-oriented, or semi structured data. Document-oriented databases are one of the main categories of**NoSQL databases**. The central concept of a document-oriented database is the notion of aDocument. While each document-oriented database implementation differs on the details of this definition, in general, they all assume documents encapsulate and encode data (or information) in some standard format(s) (or encoding(s)). Encodings in use include XML, YAML, JSON and BSON, as well as binary forms like PDF and Microsoft Office documents (MS Word, Excel, and so on).

MongoDB: MongoDB is a collection-oriented, schema-free document database. Data is grouped into sets that are called 'collections'. Each collection has a unique name in the database, and can contain an unlimited number of documents. Collections are analogous to tables in a RDBMS, except that they don't have any defined schema.

- **CouchDB**: CouchDB is a document database server, accessible via a RESTful JSON API. It is Ad-hoc and schema-free with a flat address space. Its Query-able and index-able, featuring a table oriented reporting engine that uses JavaScript as a query language. A CouchDB document is an object that consists of named fields. Field values may be strings, numbers, dates, or even ordered lists and associative maps.

- **Terrastore**: Terrastore is a modern document store which provides advanced scalability and elasticity features without sacrificing consistency. It is based on Terracotta, so it relies on an industry-proven, fast clustering technology.
- **RavenDB**: Raven is a .NET Linq enabled Document Database, focused on providing high performance, schema-less, flexible and scalable NoSQL data store for the .NET and Windows platforms.
  Raven store any JSON document inside the database. It is schema-less database where you can define indexes using C#'s Linq syntax.
- **OrientDB**: OrientDB is an open source NoSQL database management system written in Java. Even if it is a document-based database, the relationships are managed as in graph databases with direct connections between records. It supports schema-less, schema-full and schema-mixed modes. It has a strong security profiling system based on users and roles and supports SQL as a query languages.
- **ThruDB:** Thrudb is a set of simple services built on top of the Apache Thrift framework that provides indexing and document storage services for building and scaling websites. Its purpose is to offer web developers flexible, fast and easy-to-use services that can enhance or replace traditional data storage and access layers.
  It supports multiple storage backends such as BerkeleyDB, Disk, MySQL and also having Memcache and Spread integration.

Column-based database

A columnar database is a database management system (DBMS) that stores data in columns instead of rows.
The goal of a columnar database is to efficiently write and read data to and from hard disk storage in order to speed up the time it takes to return a query.

Each storage block contains data from only one column, Examples: BigTable, Cassandra, Hbase, and Hypertable.

Graph-based database

A graph-based database is a network database that uses nodes to represent and store data. Examples are Neo4J, InfoGrid, Infinite Graph, and FlockDB.

● CAP Theorem

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

- Availability - will a request made to the data store always eventually complete, no matter what (non-total) pattern of failures have occurred?
- Consistency - will all executions of reads and writes seen by all nodes be sequentially consistent? Roughly, this means that the results of 'earlier' writes are seen by 'later' reads, but the formal definition is a little more subtle.
- Partition tolerance - the network can suffer arbitrary failure patterns. This can be modelled as the refusal of the network to deliver any subset of the messages sent between nodes. Note that the failure of a single node can count as a 'partition'. See my blog post at [1] for more on this detail.

The CAP theorem categories systems into three categories:

- CP (Consistent and Partition Tolerant) - At first glance, the CP category is confusing, i.e., a system that is consistent and partition tolerant but never available. CP is referring to a category of systems where availability is sacrificed only in the case of a network partition.
- CA (Consistent and Available) - CA systems are consistent and available systems in the absence of any network partition. Often a single node's DB servers are categorized as CA systems. Single node DB servers do not need to deal with partition tolerance and are thus considered CA systems. The only hole in this theory is that single node DB systems are not a network of shared data systems and thus do not fall under the preview of CAP. [^11]
- AP (Available and Partition Tolerant) - These are systems that are available and partition tolerant but cannot guarantee consistency.

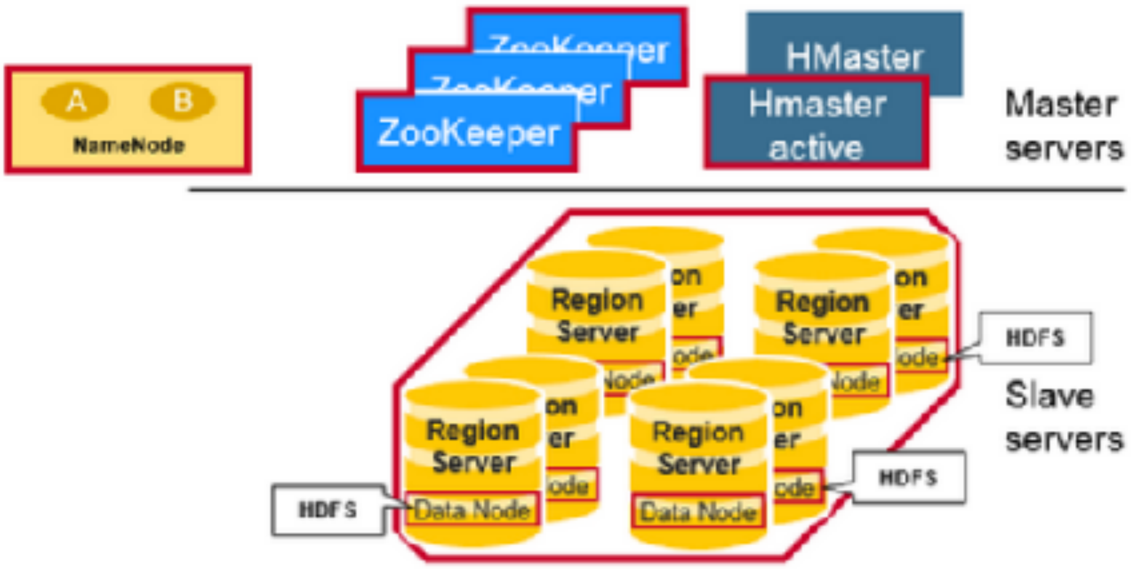
● HBase Architecture

HBase Architectural Components

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.

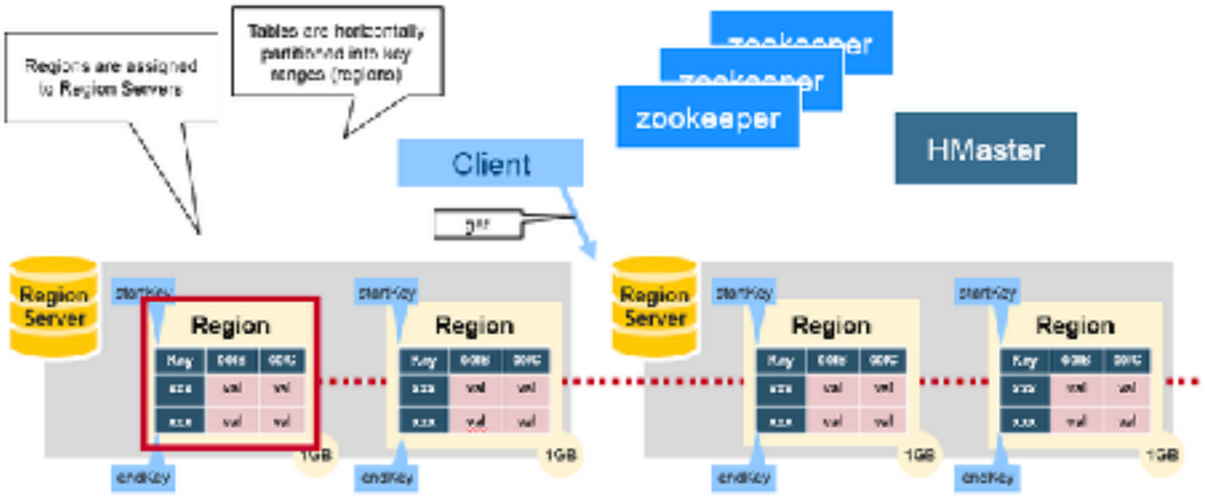
The Hadoop DataNode stores the data that the Region Server is managing. All HBase data is stored in HDFS files. Region Servers are collocated with the HDFS DataNodes, which enable data locality (putting the data close to where it is needed) for the data served by the RegionServers. HBase data is local 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.



HBase HMaster

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

A master is responsible for:

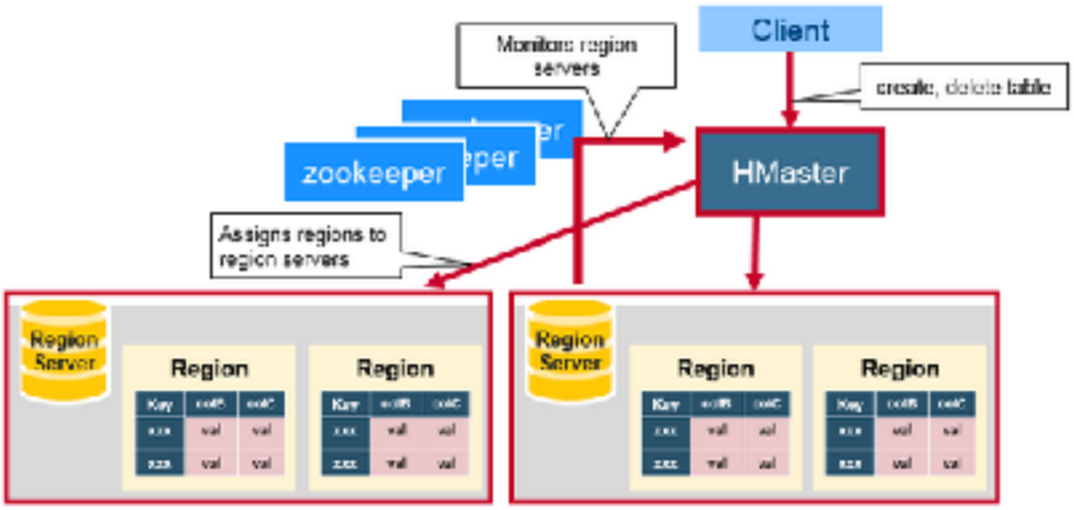
- Coordinating the region servers

  - Assigning regions on startup , re-assigning regions for recovery or load balancing

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

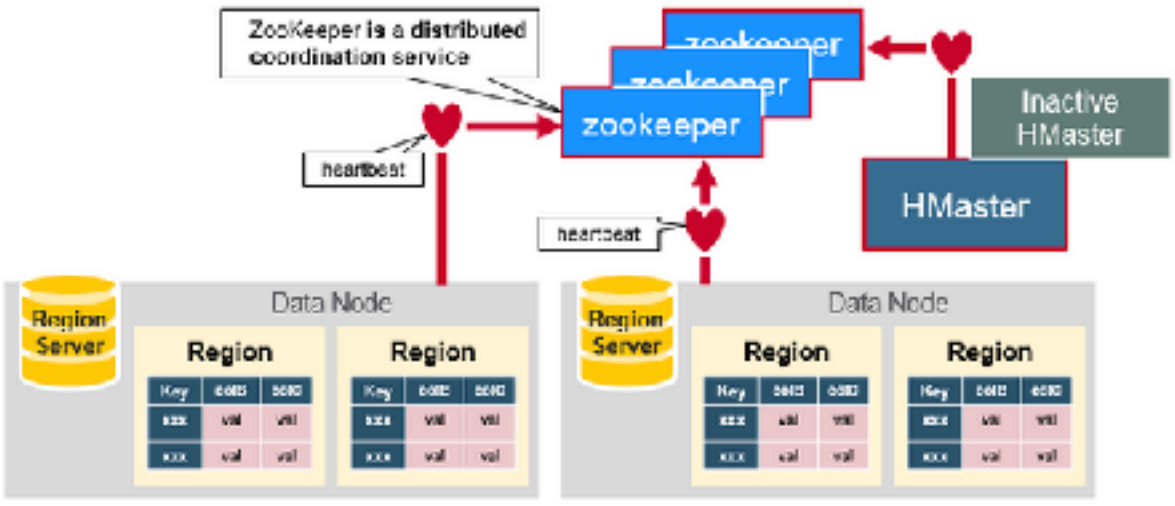
- Admin functions

  - Interface for creating, deleting, updating tables



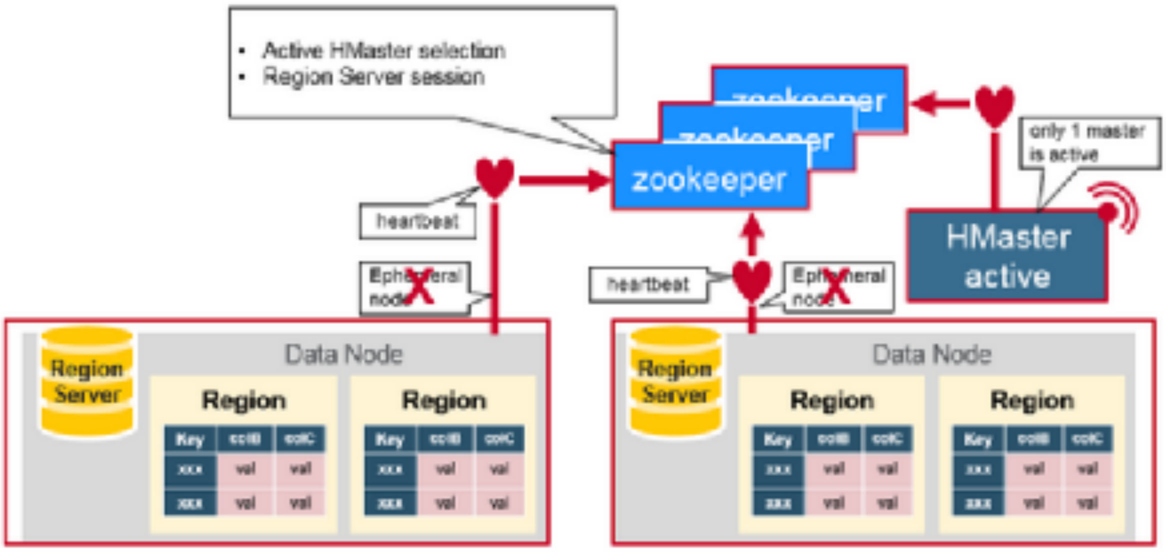
ZooKeeper: The Coordinator

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.

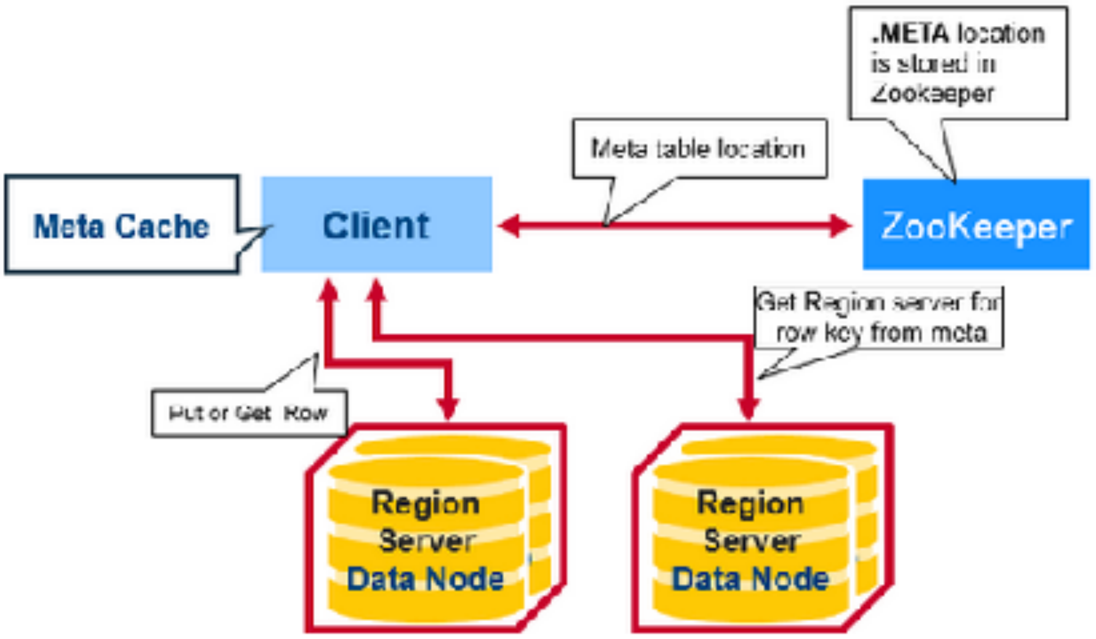
HBase First Read or Write

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.

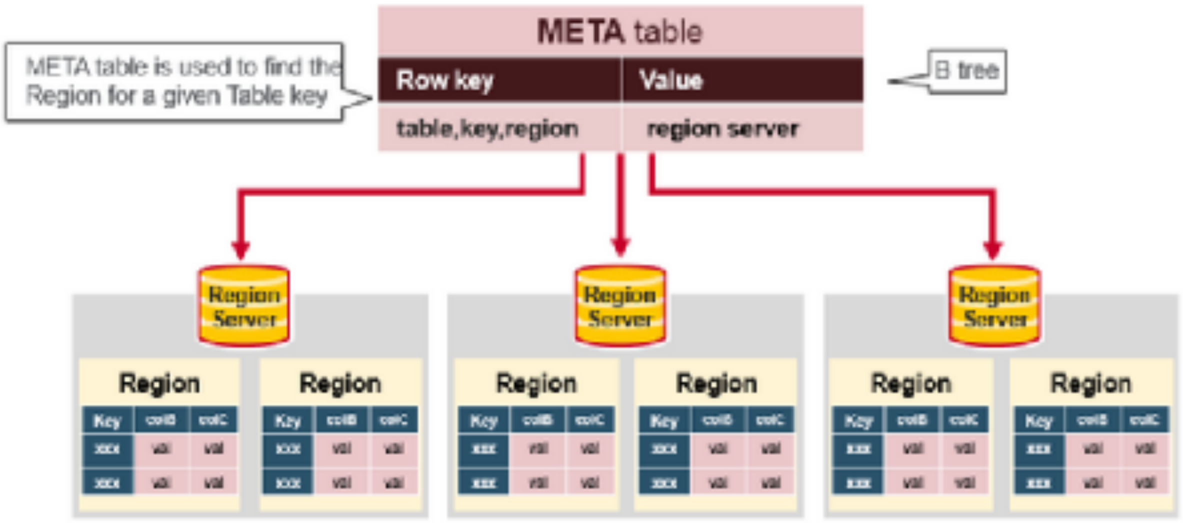


HBase Meta Table

- This META table is an HBase table that keeps a list of all regions in the system.
- The .META. table is like a b tree.
- The .META. table structure is as follows:

  - Key: region start key,region id

  - Values: RegionServer

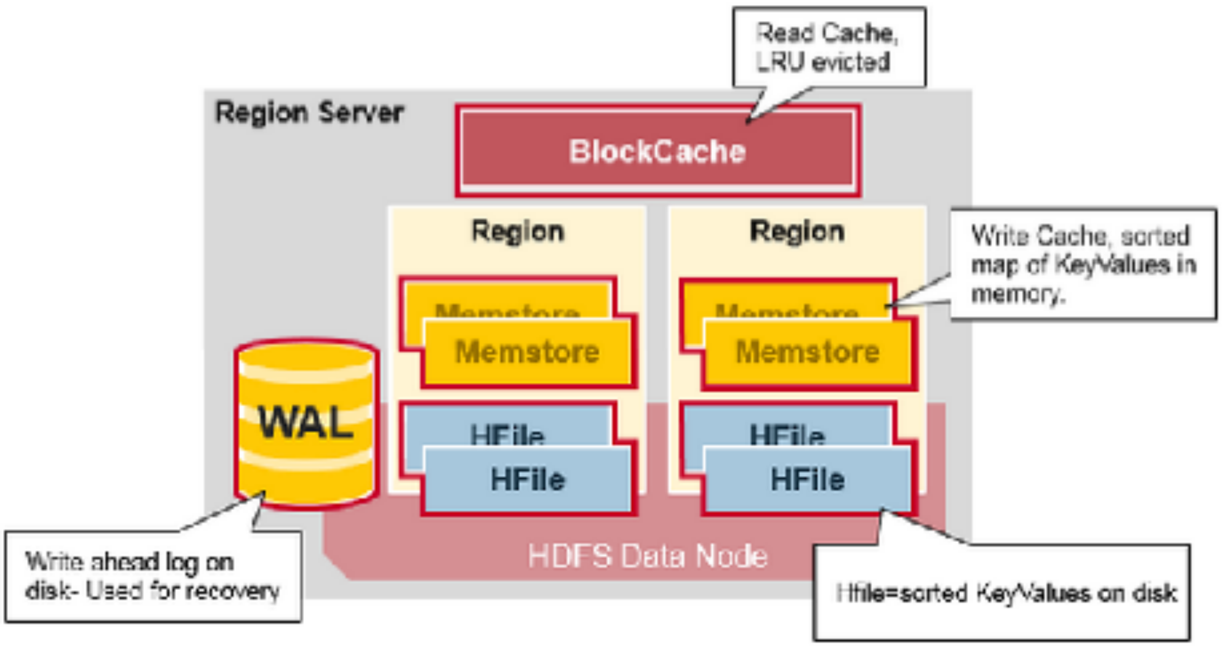


Region Server Components

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.

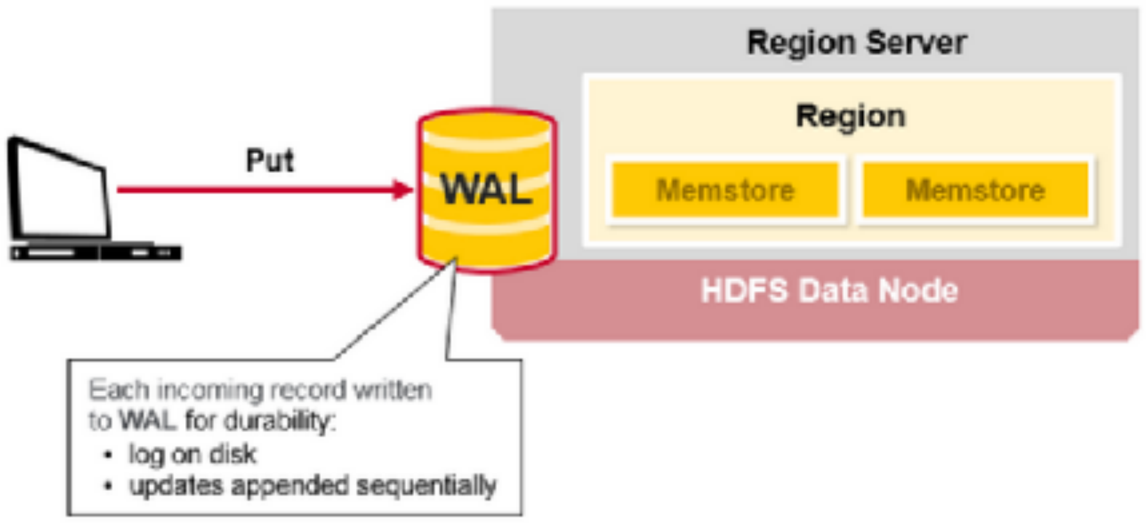
- MemStore: is the write cache. It stores new data which has not yet been written to disk. It is sorted before writing to disk. There is one MemStore per column family per region.
- Hfiles store the rows as sorted KeyValues on disk.



HBase Write Steps (1)

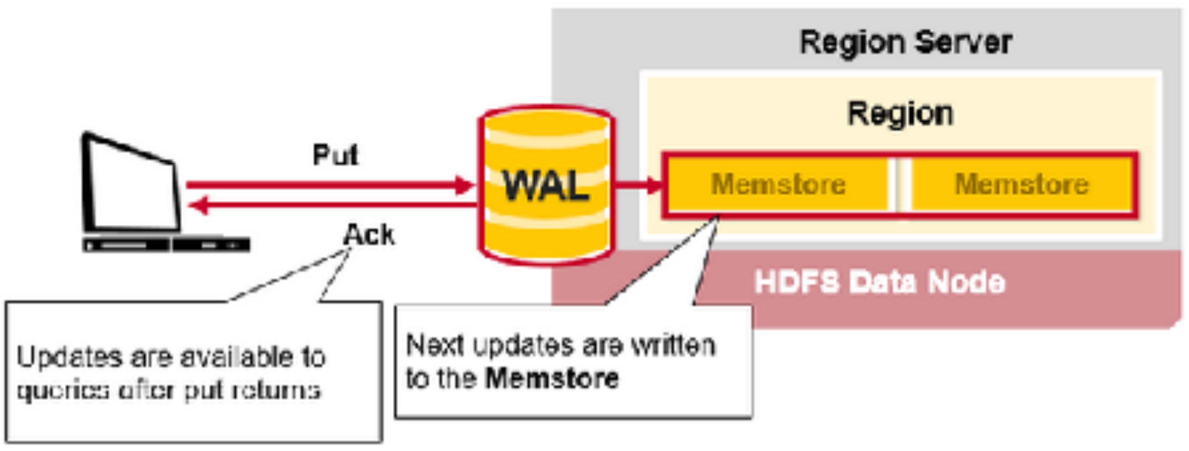
When the client issues a Put request, the first step is to write the data to the write-ahead log, the WAL:

- Edits are appended to the end of the WAL file that is stored on disk.
- The WAL is used to recover not-yet-persisted data in case a server crashes.


Region Server
Region
Memstore
Memstore
Put
WAL
HDFS Data Node
Each incoming record written to WAL for durability:
• log on disk
• updates appended sequentially

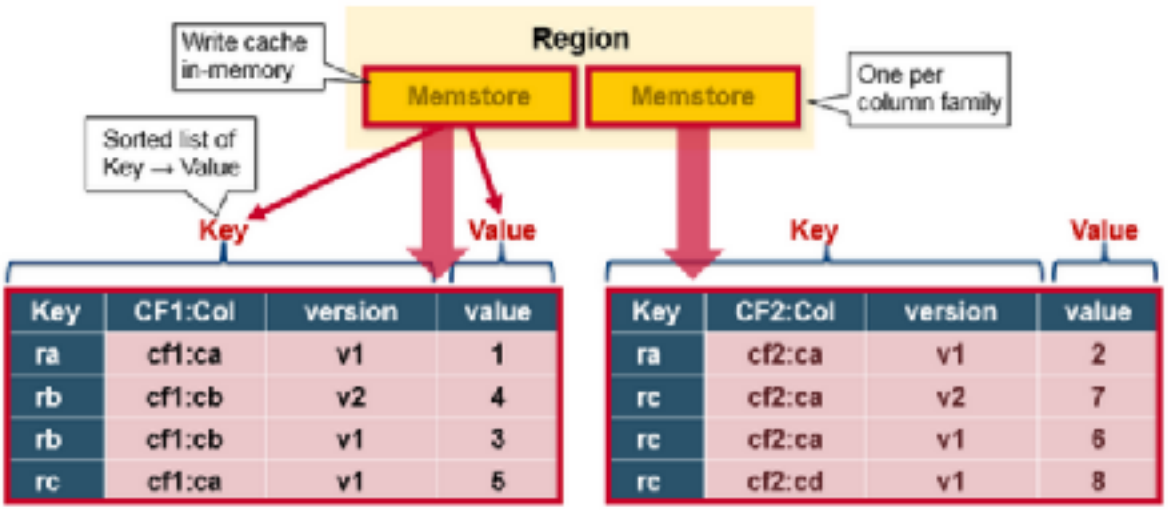
HBase Write Steps (2)

Once the data is written to the WAL, it is placed in the MemStore. Then, the put request acknowledgement returns to the client.


Region Server
Region
Put
WAL
Memstore
Memstore
Ack
HDFS Data Node
Updates are available to queries after put returns
Next updates are written to the Memstore

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

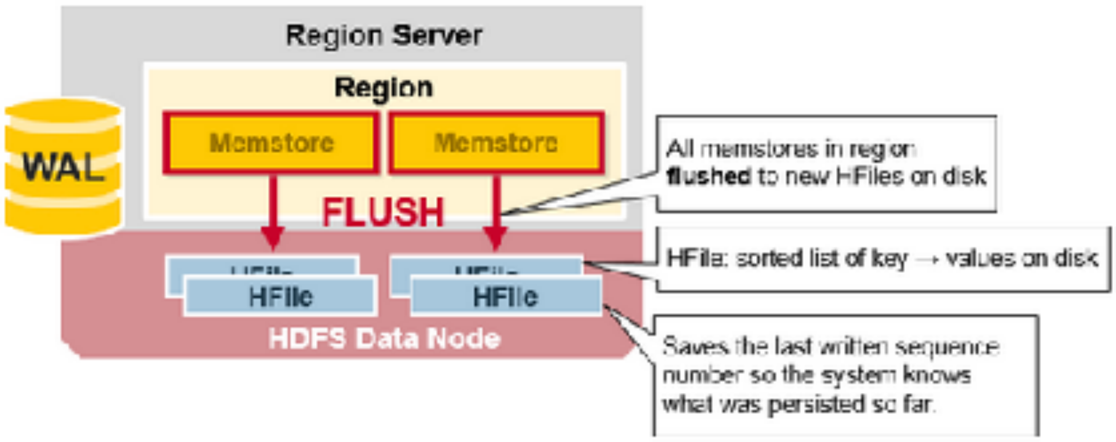
Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

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.

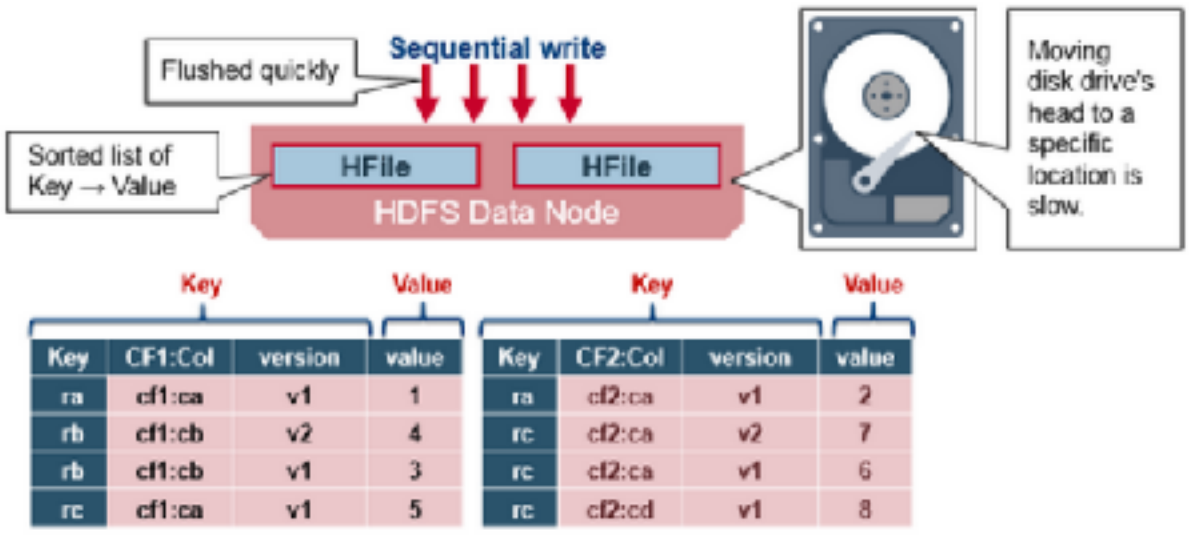
Note that this is one reason why there is a limit to the number of column families in HBase. There is one MemStore per CF; when one is full, they all flush. It also saves the last written sequence number so the system knows what was persisted so far.

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
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

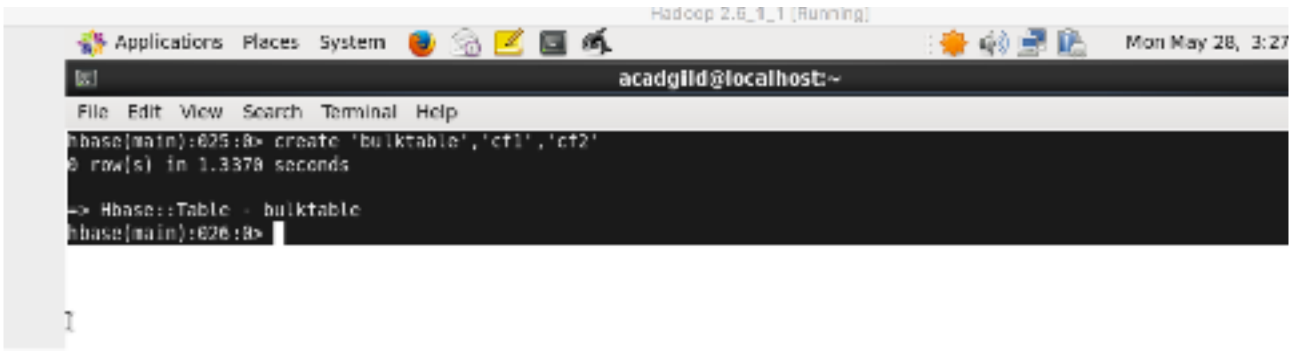
- HBase vs RDBMS

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.

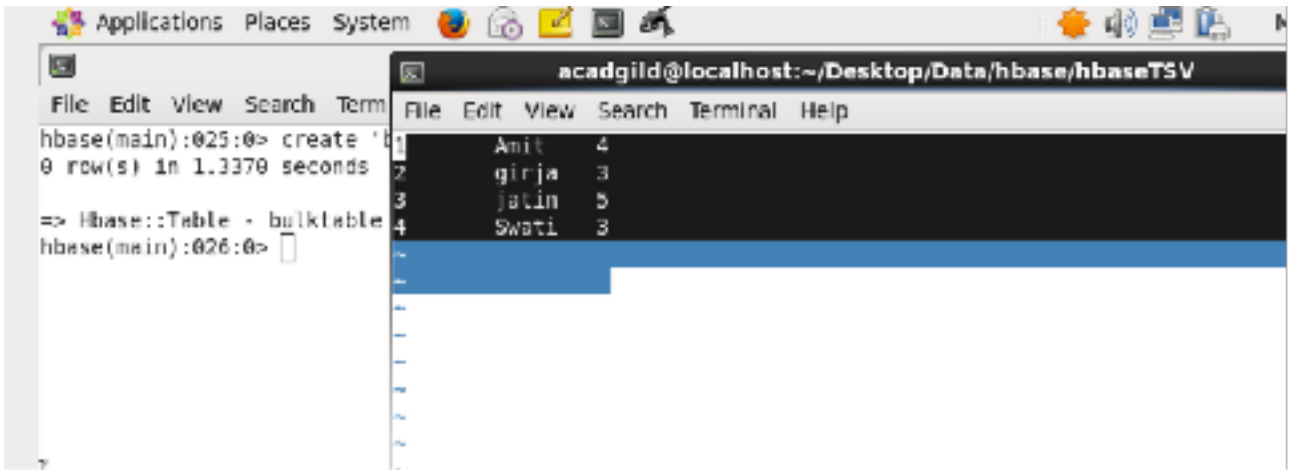
Task2

Execute blog present in below link

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

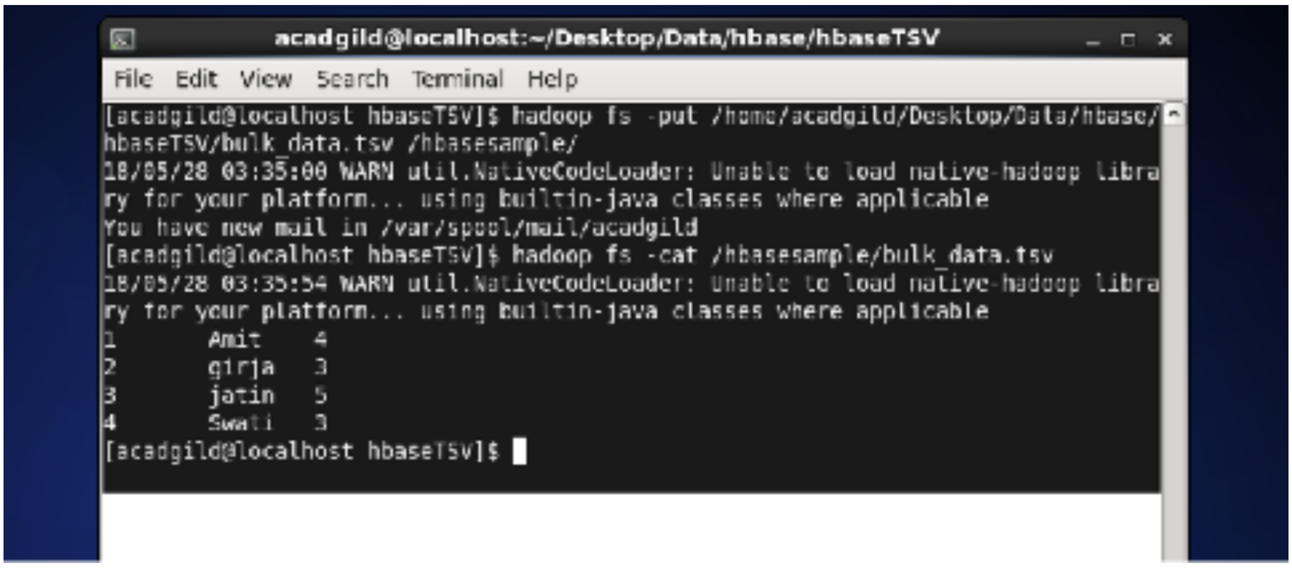

Hadoop 2.6_1_1 [Running]
Applications Places System
Mon May 28, 3:27
acadgild@localhost:~
File Edit View Search Terminal Help

```
hbase(main):025:0> create 'bulktable','cf1','cf2'
0 row(s) in 1.3370 seconds

=> Hbase::Table - bulktable
hbase(main):026:0> 
```

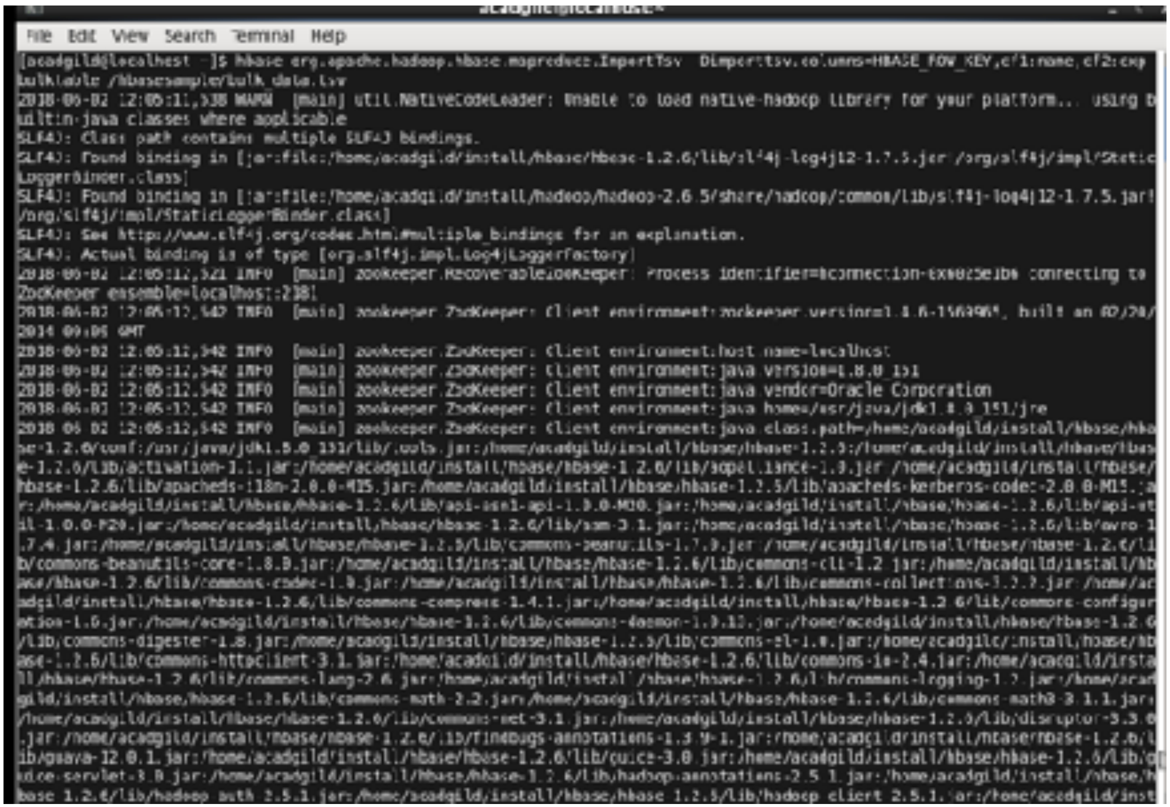

Applications Places System
File Edit View Search Term
acadgild@localhost:~/Desktop/Data/hbase/hbaseTSV
File Edit View Search Terminal Help

```
hbase(main):025:0> create 'b
0 row(s) in 1.3370 seconds

=> Hbase::Table - bulktable
hbase(main):026:0>
```

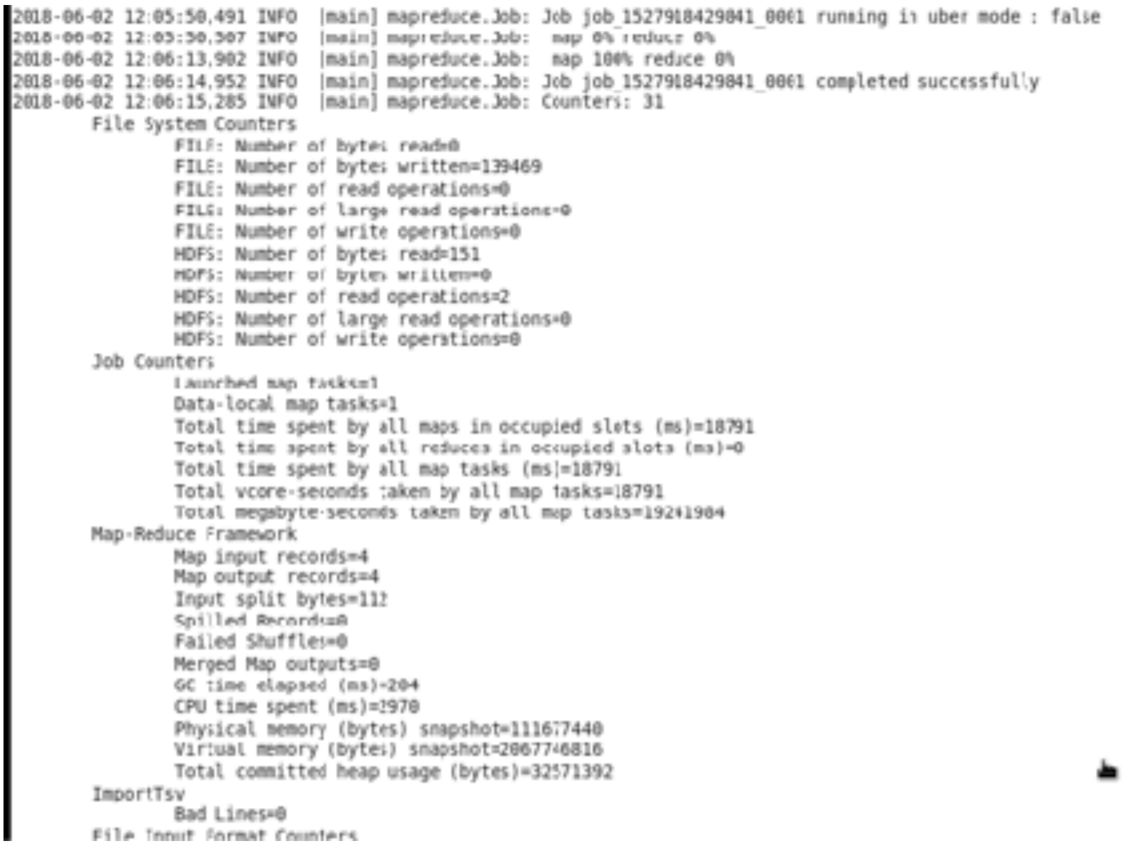
```
1       Amit    4
2       girja   3
3       jatin   5
4       Swati   3
~
~
~
~
~
~
~
~
```


acadgild@localhost:~/Desktop/Data/hbase/hbaseTSV
File Edit View Search Terminal Help

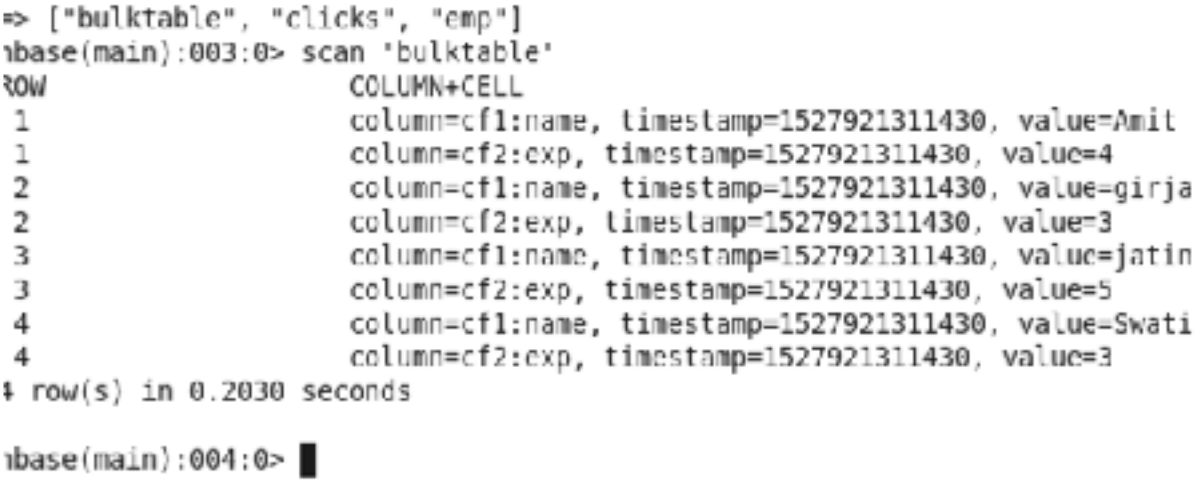
```
[acadgild@localhost hbaseTSV]$ hadoop fs -put /home/acadgild/Desktop/Data/hbase/
hbaseTSV/bulk_data.tsv /hbasesample/
18/05/28 03:35:00 WARN util.NativeCodeLoader: Unable to load native-hadoop libra
ry for your platform... using builtin-java classes where applicable
You have new mail in /var/spool/mail/acadgild
[acadgild@localhost hbaseTSV]$ hadoop fs -cat /hbasesample/bulk_data.tsv
18/05/28 03:35:54 WARN util.NativeCodeLoader: Unable to load native-hadoop libra
ry for your platform... using builtin-java classes where applicable
1       Amit    4
2       girja   3
3       jatin   5
4       Swati   3
[acadgild@localhost hbaseTSV]$
```



```
[acadgild@localhost ~]$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv  Dimporttsv.columns=HBASE_ROW_KEY,cf1:name,cf2:exp
bulktable /hbasesample/bulk_data.tsv
2018-06-02 12:05:11,538 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using b
uiltin-java classes where applicable
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/Static
LoggerBinder.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]
2018-06-02 12:05:12,521 INFO  [main] zookeeper.RecoverableZooKeeper: Process identifier=hconnection-0x802e1b6 connecting to
ZooKeeper ensemble=localhost:2181
2018-06-02 12:05:12,542 INFO  [main] zookeeper.ZooKeeper: Client environment:zookeeper.version=3.4.6-1569965, built on 02/20/
2014 09:09 GMT
2018-06-02 12:05:12,542 INFO  [main] zookeeper.ZooKeeper: Client environment:host.name=localhost
2018-06-02 12:05:12,542 INFO  [main] zookeeper.ZooKeeper: Client environment:java.version=1.8.0_151
2018-06-02 12:05:12,542 INFO  [main] zookeeper.ZooKeeper: Client environment:java.vendor=Oracle Corporation
2018-06-02 12:05:12,542 INFO  [main] zookeeper.ZooKeeper: Client environment:java.home=/usr/java/jdk1.8.0_151/jre
2018-06-02 12:05:12,542 INFO  [main] zookeeper.ZooKeeper: Client environment:java.class.path=/home/acadgild/install/hbase/hba
se-1.2.6/conf:/usr/java/jdk1.8.0_151/lib/tools.jar:/home/acadgild/install/hbase/hbase-1.2.6:/home/acadgild/install/hbase/hbas
e-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-kerberos-codec-2.0.0-M15.ja
r:/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-ut
il-1.0.0-M20.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/li
b/commons-beanutils-core-1.8.0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-cli-1.2.jar:/home/acadgild/install/hb
ase/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/ac
adgild/install/hbase/hbase-1.2.6/lib/commons-compress-1.4.1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-configur
ation-1.6.jar:/home/acadgild/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/hbase/hbase-1.2.6/lib/commons-el-1.0.jar:/home/acadgild/install/hbase/hb
ase-1.2.6/lib/commons-httpclient-3.1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/commons-io-2.4.jar:/home/acadgild/insta
ll/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/acad
gild/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/hbase-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-annotations-1.3.9-1.jar:/home/acadgild/install/hbase/hbase-1.2.6/l
ib/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/g
uice-servlet-3.0.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/hadoop-annotations-2.5.1.jar:/home/acadgild/install/hbase/h
base-1.2.6/lib/hadoop-auth-2.5.1.jar:/home/acadgild/install/hbase/hbase-1.2.6/lib/hadoop-client-2.5.1.jar:/home/acadgild/inst
```



```
2018-06-02 12:05:50,491 INFO  [main] mapreduce.Job: Job job_1527918429841_0001 running in uber mode : false
2018-06-02 12:05:50,507 INFO  [main] mapreduce.Job:  map 0% reduce 0%
2018-06-02 12:06:13,902 INFO  [main] mapreduce.Job:  map 100% reduce 0%
2018-06-02 12:06:14,952 INFO  [main] mapreduce.Job: Job job_1527918429841_0001 completed successfully
2018-06-02 12:06:15,285 INFO  [main] mapreduce.Job: Counters: 31
        File System Counters
                FILE: Number of bytes read=0
                FILE: Number of bytes written=139469
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=151
                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)=18791
                Total time spent by all reduces in occupied slots (ms)=0
                Total time spent by all map tasks (ms)=18791
                Total vcore-seconds taken by all map tasks=18791
                Total megabyte-seconds taken by all map tasks=19241984
        Map-Reduce Framework
                Map input records=4
                Map output records=4
                Input split bytes=112
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=204
                CPU time spent (ms)=2970
                Physical memory (bytes) snapshot=111677440
                Virtual memory (bytes) snapshot=2067746816
                Total committed heap usage (bytes)=32571392
        ImportTsv
                Bad Lines=0
        File Input Format Counters
```



```
=> ["bulktable", "clicks", "emp"]
hbase(main):003:0> scan 'bulktable'
ROW                        COLUMN+CELL
 1                         column=cf1:name, timestamp=1527921311430, value=Amit
 1                         column=cf2:exp, timestamp=1527921311430, value=4
 2                         column=cf1:name, timestamp=1527921311430, value=girja
 2                         column=cf2:exp, timestamp=1527921311430, value=3
 3                         column=cf1:name, timestamp=1527921311430, value=jatin
 3                         column=cf2:exp, timestamp=1527921311430, value=5
 4                         column=cf1:name, timestamp=1527921311430, value=Swati
 4                         column=cf2:exp, timestamp=1527921311430, value=3
4 row(s) in 0.2030 seconds

hbase(main):004:0> 
```