iNeuron

NoSQL Databases

Basic introduction

Challenges with traditional RDBMS

A relational database management system is software that stores, manages, queries, and retrieves data from a relational database (RDBMS). The RDBMS offers a user and application interface to the database, as well as administrative operations for data storage, access, and performance management. Though there are some challenges with RDBMS, let us understand those challenges

- Join - The data in a relational database is assumed to be stored in a tabular fashion, and duplication is eliminated by joining the data together. While this method works well for small datasets, as the data becomes large and dispersed, combining data from many tables becomes a difficult task.
- Transaction support - When we start distributing data, the distributed data store's consistency becomes a problem.
- Cost of RDBMS - RDBMS can fix some of the issues, but they are expensive.
- New applications, such as web and mobile applications, require high availability and low latency.
- It structures differently for XML and JSON files.

iNeuron

What is NoSQL database, why NoSQL database?

Non-tabular databases, such as NoSQL databases, store data in a different way than relational tables. NoSQL databases are classified according to their data model. Document, key-value, wide-column, and graph are the most common types. They have adaptable schemas and can handle big amounts of data and high user loads with ease. When individuals say "NoSQL database," they're usually referring to any database that isn't relational. Some people claim "NoSQL" means "not only SQL," while others say it means "non SQL." In any case, most people agree that NoSQL databases are databases that don't use relational tables to store data.

History behind the NoSQL database?

As the cost of storage fell substantially in the late 2000s, NoSQL databases emerged. Gone were the days when avoiding data duplication necessitated creating a sophisticated, difficult-to-manage data model. Because developers (rather than storage) were becoming the primary cost of software development , NoSQL databases were designed to maximize developer productivity.

Features of NoSQL database?

Each NoSQL database has its own set of features. Many NoSQL databases contain the following features at a high level -

- Flexible schemas - Unlike SQL databases, where you must first identify and declare a table's schema before inserting data, in NoSQL databases you can just insert data. The fields in a single collection do not have to be the same, and the data type for a field might vary between documents inside a collection.

- Support for multiple data models - While relational databases require data to be organised into tables and columns before being accessed and analysed, NoSQL databases are incredibly flexible when it comes to data management. They can easily consume structured, semi-structured, and unstructured data, whereas relational databases are quite rigid and only handle structured data. Specific application requirements are handled by different data models. To more easily handle diverse agile application development requirements, developers and architects pick a NoSQL database.

- Easily scalable via Peer-to-Peer architecture - It's not that relational databases can't scale; it's that they can't scale EASILY or CHEAPLY. This is because they're created with a classic master-slave architecture, which implies scaling UP via larger and larger hardware servers rather than scaling OUT or worse by sharding. Sharding is the process of splitting a database into smaller chunks and distributing them across numerous hardware servers rather than a single huge server, which causes operational administration issues. Instead, search for a NoSQL database with a peer-to-peer architecture with all nodes being the same.

- Distribution capabilities - Look for a NoSQL database that is built to distribute data at a worldwide scale, which means it can write and read data from many locations involving multiple data centres and/or cloud regions. In contrast, relational databases rely on a centralised application that is location-dependent (e.g., a single location), particularly for write operations. Because data is spread with numerous copies where it needs to be, adopting a distributed database with a masterless design allows you to maintain continuous availability.

Different types of NoSQL databases?

There are 4 types of NoSQL databases -

- Document databases - Data is stored in JSON, BSON, or XML documents in a document database (not Word documents or Google docs, of course). Documents in a document database can be nested. For speedier querying, certain elements can be indexed. Documents can be stored and retrieved in a format that is much more similar to the data objects used in applications, requiring less translation when using the data in applications. When moving SQL data between applications and storage, it is frequently assembled and dismantled.
  use cases - Trading platforms, E-commerce platforms etc.

- Key-value stores - A key-value store is the most basic sort of NoSQL database. Every database data element is kept as a key value pair, which consists of an attribute name (or "key") and a value. A key-value store is similar to a relational database in that it only has two columns: the name of the key or attribute (such as fruit) and the value (such as Mango)
  use cases - User profiles, Shopping cart etc

- Column-oriented databases - A column store is structured as a group of columns, whereas a relational database stores data in rows and reads data row by row. This means that if you just need to analyze a few columns, you can read those columns directly without wasting RAM on irrelevant data. Because columns are frequently of the same kind, they benefit from more efficient compression, which speeds up reads. The value of a column in a columnar database can be easily aggregated (for example, adding up the total number of students in a class). Analytics is an example of a use case.

- Graph databases - A graph database focuses on the relationship between data elements. Every element is represented by a node (such as a person in a social media graph). The term "links" or "relationships" refers to the connections that exist between items. Connections are first-class database elements in a graph database, and they are stored directly. Links are inferred in relational databases, which use data to express relationships. use cases - Knowledge graphs, fraud detection.

When NoSQL databases are used?

The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. Big data and real-time web apps both use NoSQL. Every day, firms like Twitter, Facebook, and Google, for example, collect gigabytes of user data.

Advantages of NoSQL databases

The limitations of traditional relational database technology prompted the development of NoSQL databases. NoSQL databases are frequently more scalable and give better performance than relational databases. Following are the advantages of NoSQL databases -

- Handle large columns of data at high speed with scale-out architecture - The most common way to construct SQL databases is to use a scale-up architecture, which is based on leveraging larger machines with more CPUs and memory to boost speed. In the Internet and cloud computing eras, NoSQL databases were developed, making it easier to construct a scale-out architecture. Scalability is achieved in a scale-out architecture by distributing data storage and processing operations over a large cluster of

computers. More computers are added to the cluster to boost capacity.

- Store unstructured, semi-structured or structured data - NoSQL databases have become popular because they allow data to be stored in more understandable or similar ways to how it is used by applications. When data is saved or retrieved for use, fewer changes are necessary. Many various types of data can be saved and accessed more simply, whether structured, unstructured, or semi-structured. Furthermore, many NoSQL databases' schemas are flexible and under the developers' control, making it easier to adapt the database to new types of data. This eliminates inefficiencies in the development process caused by requesting a SQL database be redesigned by a database administrator.
- Enable easy updates to schema and fields - NoSQL databases have become popular because they store data in simple straightforward forms that can be easier to understand than the type of data models used in SQL databases. Furthermore, NoSQL databases frequently allow developers to update the data structure directly. Document databases don’t have a set data structure to start with, so a new document type can be stored just as easily as what is currently being stored. New values and columns can be added to key-value and column-oriented stores without affecting the current structure. Developers of graph databases update nodes with new characteristics and arcs with new meanings in response to new types of data.
- Developer-friendly - Developers have been the primary drivers of NoSQL database adoption, as they find it easier to design many types of applications than with relational databases. JSON is used by document databases like MongoDB to transform data into something that resembles code. This gives the developer complete control over the data's structure. Furthermore, NoSQL databases store data in forms that are similar to the types of data objects used in applications, requiring fewer transformations when moving data in and out.

Disadvantages of NoSQL databases

Following are the disadvantages of NoSQL databases -

- Not all NoSQL databases take atomicity and data integrity into consideration. They can resist what is referred to as "ultimate consistency."
- SQL instructions have compatibility concerns. The query language of new databases has its own peculiarities, and it is not yet 100 percent compatible with SQL used in relational databases. Support for work query issues in a NoSQL database are more complicated.
- There is a lack of standardization. There are various NoSQL databases, but none of them follow the same standards as relational databases. These databases are expected to have an unclear future.
- Support for multiple platforms, some systems still require significant changes in order to work on non-Linux operating systems.
- Usability is a problem. They frequently have access to consoles or management tools that aren't really useful.

iNeuron

HBase

iNeuron

What is HBase?

On top of the Hadoop file system, HBase is a distributed column-oriented database. It is a horizontally scalable open-source project. HBase is a data format similar to Google's Big Table that allows users to access large volumes of structured data at random. It takes advantage of the Hadoop File System's fault tolerance. It's a component of the Hadoop ecosystem that allows users to read and write data in the Hadoop File System in real-time. The data can be stored in HDFS directly or through HBase. Using HBase, the consumer reads/accesses the data in HDFS at random. HBase is a read-write database that sits on top of the Hadoop File System.

iNeuron

HDFS vs HBase

HDFS	HBase
HDFS is a distributed file system designed for big file storage.	HBase is a database that runs on top of the HDFS file system.
Individual record lookups are not supported by HDFS.	For larger tables, HBase allows for fast lookups.
There is no concept of batch processing; it provides high latency batch processing.	It allows users to access single rows from billions of records with minimal latency.
It only allows for sequential data access.	Internally, HBase employs Hash tables to give random access to data, which is stored in indexed HDFS files for speedier lookups.

iNeuron

HBase vs RDBMS

HBase	RDBMS
HBase is schema-less, meaning it doesn't have a set column structure. Instead, it defines column families.	The schema of an RDBMS governs its operation, as it describes the entire table structure.
It's designed to fit large tables. HBase can be scaled horizontally.	It is thin and designed for compact tables. It's difficult to scale.
In HBase, there are no transactions.	RDBMS is transactional.
It has denormalized data.	It will have normalized data.
It works effectively with both semi-structured and structured data.	It works effectively with structured data.

iNeuron

HBase vs Hive

Hive	HBase
Hive is a query engine.	Data storage is especially important for unstructured data.
Batch processing is the most common use.	For transactional processing, it's widely used.
This is not a real-time processing system.	This is a real-time processing system.
Only used for analytical queries.	Used for real-time querying

iNeuron

HBase storage mechanism

The tables in HBase are sorted by row and it is a column-oriented database. Only column families, or key value pairs, are defined in the table schema. A table can contain numerous column families, each of which can contain any number of columns. The values of subsequent columns are saved on the disc in a logical order. A timestamp is included in each table cell value. In HBase -

- A table is a collection of rows.
- A row is a collection of column families.
- A collection of columns is referred to as a column family.
- A column is a collection of key-value pairs.

Features of HBase database

- Consistency - Because it provides consistent reads and writes, we can use this HBase capability for high-speed applications.
- Atomic read and write - During one read or write process, all other processes are prevented from performing any read or write operations this is what we call Atomic read and write. On a row level, HBase provides atomic read and write.
- Sharding - As soon as a region reaches a threshold size, HBase supports automatic and manual splitting of regions into smaller subregions to save I/O latency and overhead.

- High availability - It also includes LAN and WAN failover and recovery capabilities. At the heart of the system is a master server, which is responsible for monitoring the region servers as well as the cluster's metadata.
- Scalability - HBase allows scaling in both linear and modular forms. We can also remark that it is linearly scalable.
- Distribution storage - HBase's distributed storage functionality, such as HDFS, is enabled through this feature.

Applications of HBase

- Medical - HBase is used in the medical field for storing genome sequences and executing MapReduce on them, as well as keeping people's or an area's sickness history, among other things.
- Sports - In the sports world, HBase is utilised to store match history for better analytics and prediction.
- Web - For improved customer targeting, HBase is utilised to maintain user history and preferences.
- Oil and petroleum - In the oil and petroleum business, HBase is used to store exploration data for analysis and to anticipate where oil can be found.
- E-commerce - HBase is used to keep track of and store logs regarding consumer search histories, as well as to perform analytics and target advertisements for better business.

Installation of HBase on Cloudera

We will see how to install HBase on windows with the following steps

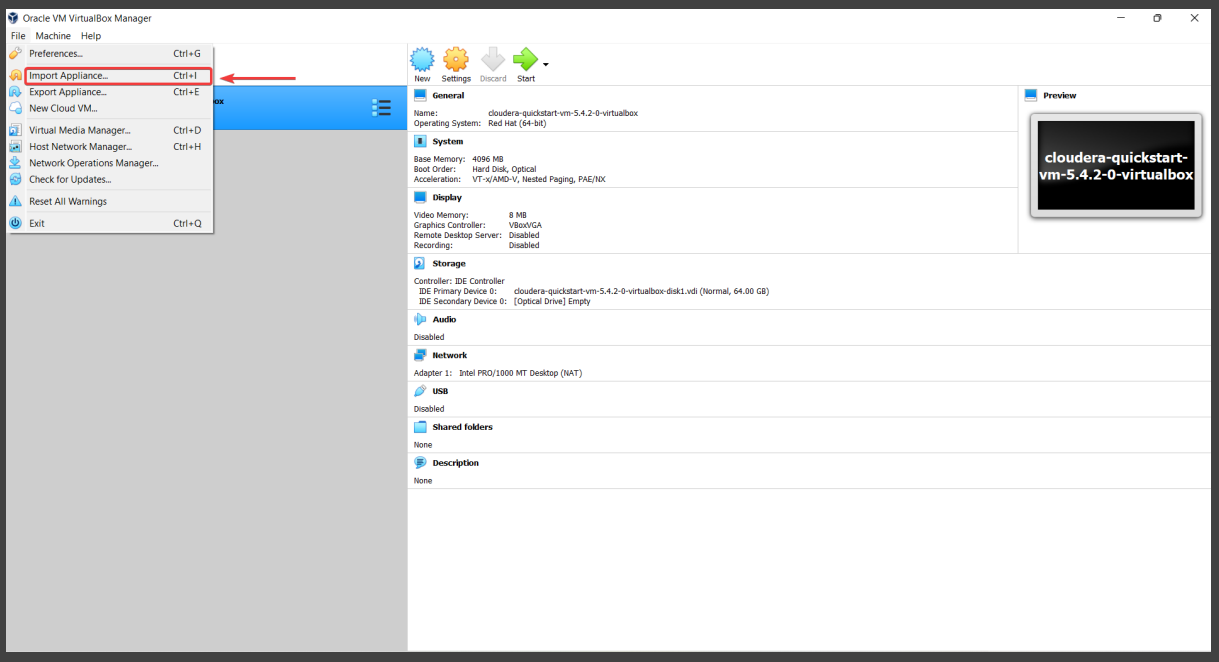
1. First we need to download the Virtual box.

```
Download link - https://www.virtualbox.org/wiki/Downloads
```

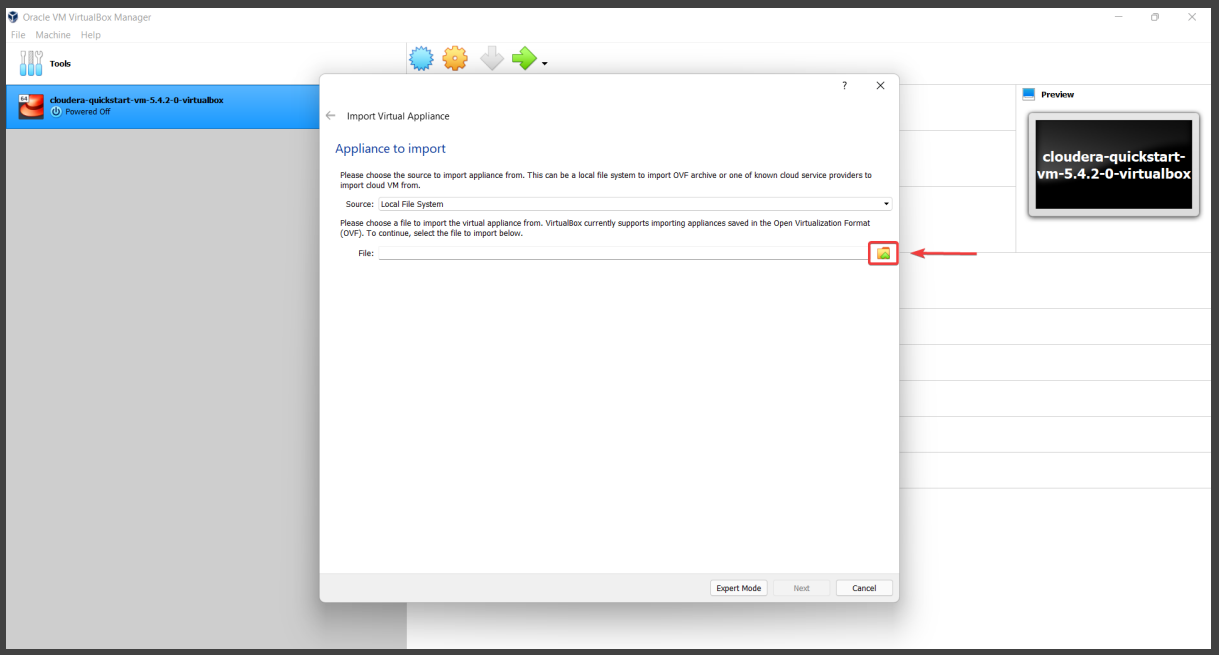
2. Now we will download cloudera quickstart image for Virtual box. And unzip the file using 7 zip.

```
Download link -
https://downloads.cloudera.com/demo_vm/virtualbox/cloudera-quickstart-vm-5.4.2-0-virtualbox.zip
```

3. Install cloudera quickstart image with default settings by clicking next -next.
4. Now open Virtual box application, go to file option and select import Appliance. (refer below image)

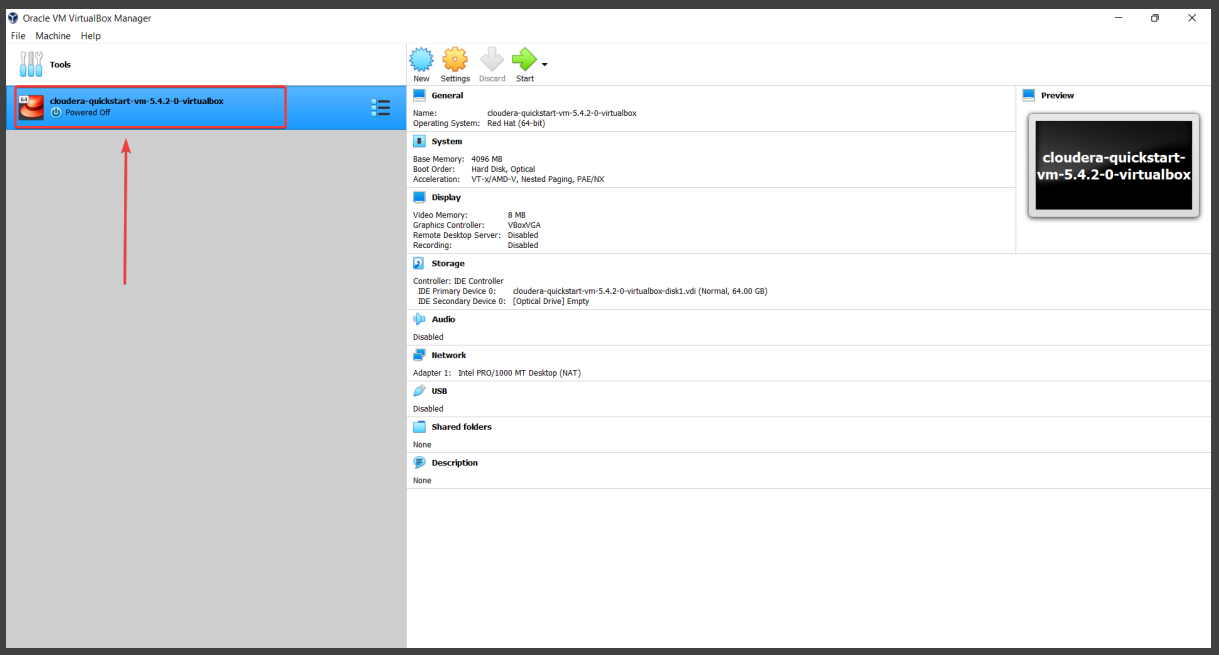


5. After clicking there navigate to the cloudera quickstart image and open it. (refer below image)



6. After navigating to the cloudera image click on the import button.

7. Double click on the ‘cloudera-quickstart-vm-5.4.2-0-virtualbox’ on the top left side (as shown in the figure below) to start the cloudera vm.



8. After double clicking on the vm the below screen will appear -

9. For confirmation we will run a basic command

```
Command - hdfs dfs -ls /
```

After running the above command it will show some hdfs files.

10. To log in from the vm's browser to cloudera manager, we have to write the following command -

```
sudo /home/cloudera/cloudera-manager --express --force
```

11. After successful execution of the above command we will get one link with user id and password. Copy that link and paste it in vm's browser and provide the user id and password.

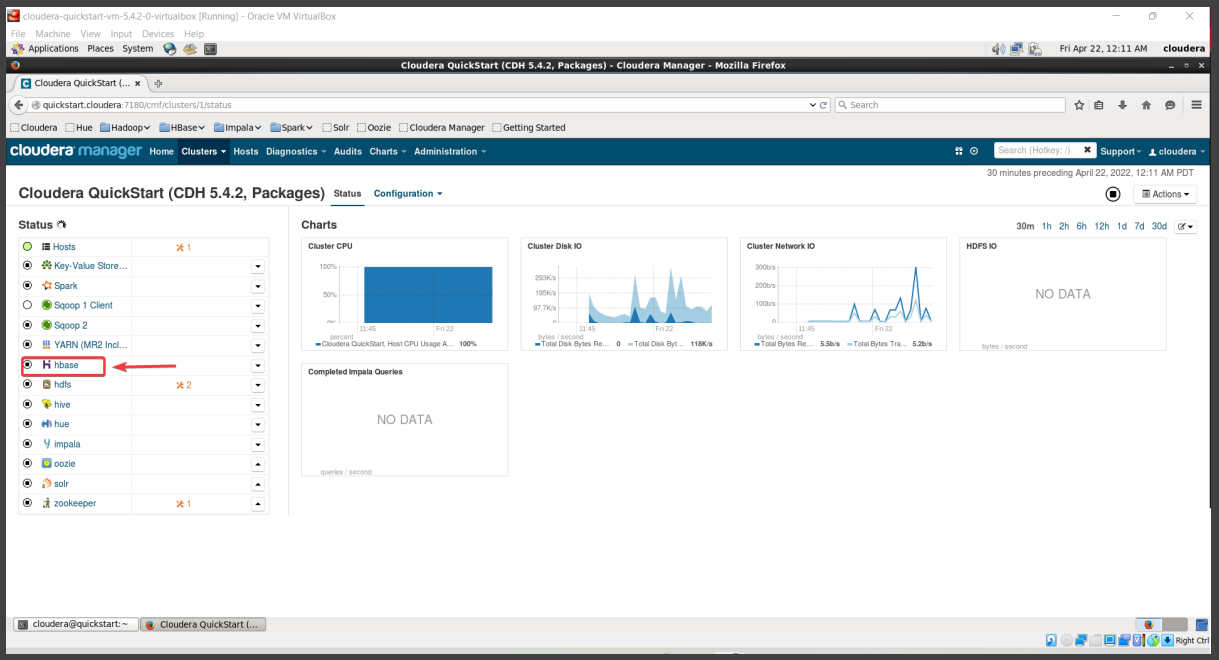
Basic static configuration

Configurations are settings, environments and machine roles for nodes and clusters. We will learn about how we can set those settings manually. Each service has its own configuration files. E.g.

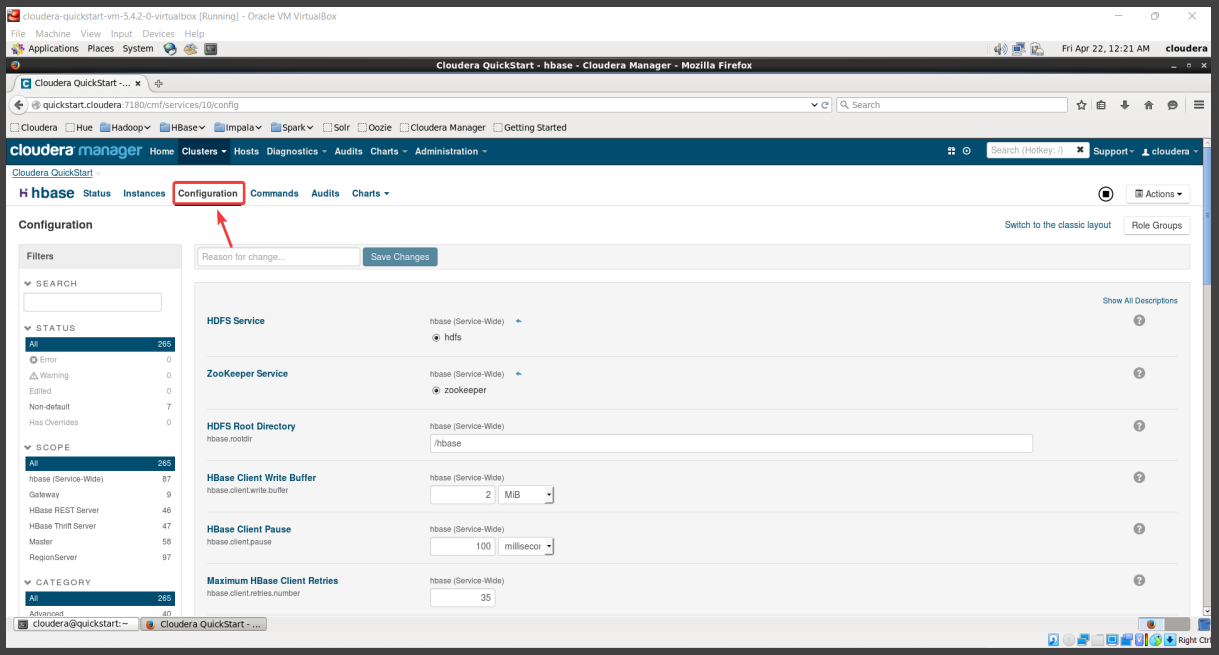
- HBase: hbase-site.xml / hbase-env.sh
- HDFS: hdfs-site.xml / hdfs-env.sh
- YARN: yarn-site.xml / yarn-env.sh
- Zookeeper: zoo.cfg

For Hbase, HDFS and YARN we have site.xml and env.sh configuration files for settings and to set up the environment of daemons. For Zookeeper we have a zoo.cfg file.

For that configuration please go to vm's browser and click on the hbase. (as shown in the below figure)



After clicking on the hbase button click on the Configuration button on the top menu bar and you can configure the settings of HBase -



Architecture of HBase

The 3 main components of HBase architectures are HMaster, Region Server and Zookeeper.

- HMaster - HMaster is the HBase implementation of Master Server. It is a process in which regions and DDL (create table, delete table, alter table) operations are assigned to a region server. It keeps track of all Region Server instances in the cluster. Master runs numerous background threads in a distributed system. HMaster includes a lot of functions, such as load balancing, failover, and so on.
- Region Server - HBase Tables are separated into Regions horizontally by row key range. Regions are the fundamental building blocks of an HBase cluster, consisting of a distribution of tables and Column families. The Region Server runs on an HDFS DataNode in the Hadoop cluster. Regions of Region Server are responsible for a variety of tasks, including handling, administering, and performing HBase operations on that set of regions. A region's default size is 256 MB.
- Zookeeper - In HBase, it works similarly to a coordinator. It offers features such as configuration information management, naming, distributed synchronization, and server failure notification etc. Clients use zookeeper to communicate with region servers.

HMaster server

Following are the roles and the responsibilities of HMaster server -

- Assigns regions to region servers with the help of Apache Zookeeper.
- Handles region load balancing across region servers. It shifts the regions to less occupied servers after unloading the busy servers.
- Maintains the state of the cluster by negotiating the load balancing.
- HMaster is responsible for schema modifications and other metadata actions like table and column formation.

HBase regions server

Regions are nothing but the tables that are split up and spread across the region servers. Following are the regions that region servers has -

- Handle data-related actions and communicate with the client.
- It handles read and write requests for all the regions which come under it.
- It decides the size of the region by following the region size thresholds.

Zookeeper

- Zookeeper is an open-source project that provides services such as configuration management, naming, and distributed synchronization.

- Zookeeper contains ephemeral nodes that represent various area servers. These nodes are used by master servers to find available servers.
- The nodes are used to track server failures and network partitions in addition to ensuring availability.
- Clients communicate with region servers via zookeeper.

HBase Commands

HBase commands are mainly distributed into 3 sections - General commands, Data definition commands and Data manipulation commands. In this section we will learn all these commands in detail.

What is HBase shell?

HBase provides a shell that you can use to interface with it. HBase stores its data in the Hadoop File System. There will be a master server as well as region servers. Data will be stored in the form of regions (tables). These areas will be divided and stored in separate region servers. These region servers are managed by the master server, and all of these functions are performed on HDFS.

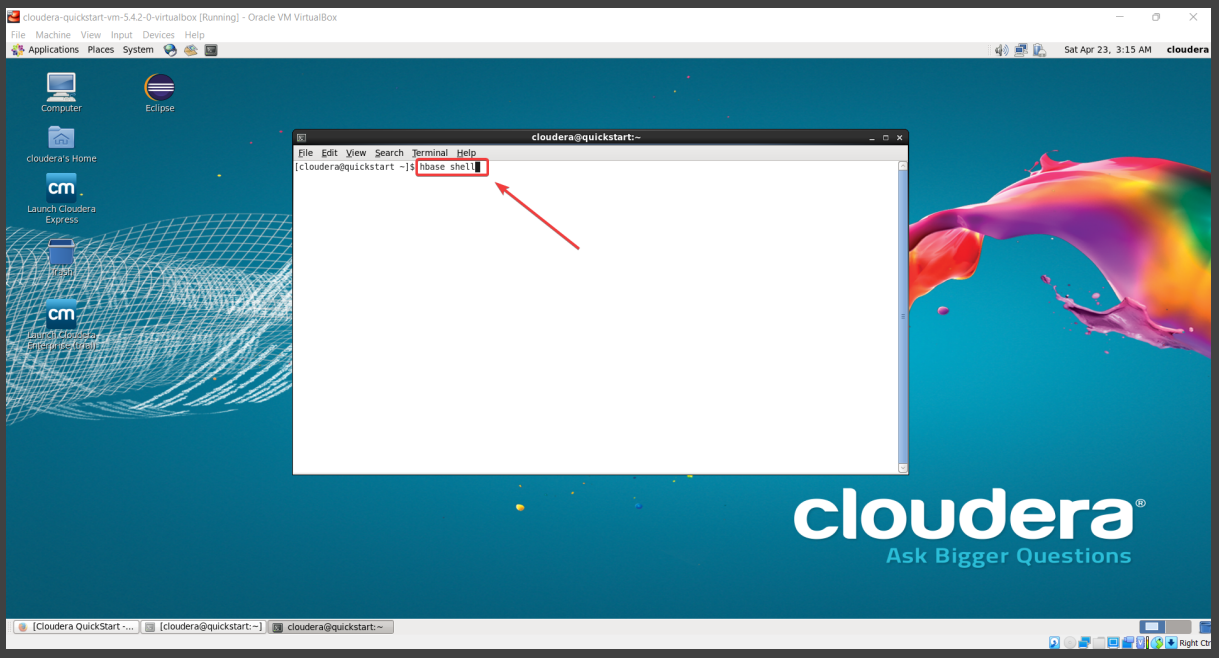
HBase shell usage

Generally to communicate with HBase, we use the HBase shell.

- All the names we write in HBase shell should be quoted. E.g. table and column names.
- If we want to create and alter a table, we use dictionaries of configuration. E.g. {‘key1’ => ‘value1’, ‘key2’ => ‘value2’}
- Generally all the keys are constant here like - NAME, VERSIONS etc.
- If we want to see a list of all constants in the environment, type ‘Object.constants’.

Starting HBase shell

You have to click on the command prompt as shown in the figure below and type `hbase shell` to start the shell.



iNeuron

General commands

1. status -
   It shows the status of a cluster.

```
Syntax - hbase(main):001:0> status
```

2. table_help -
   This command deals in Table reference commands such as scan, put, disable, drop etc.

```
Syntax - hbase(main):001:0> table_help
```

3. version -
   It displays the version of HBase.

```
Syntax - hbase(main):001:0> version
```

4. whoami -
   It displays the current user details of HBase

```
Syntax - hbase>whoami
```

Data Definition commands

Cell in HBase is a combination of the row, column family, and version contains a value and a timestamp, which represents the column family version. Following is the data definition commands

1. alter -
   We use the alter command to add/modify/delete column families, also to change the configuration of the table.
   a. Add/modify column family -
      In order to change and add the ‘family 1’ column family in table ‘table 1’ from the current value to keep a maximum of 3 cell VERSIONS.

Command -

```
hbase> alter 'table 1', NAME => 'family 1', VERSIONS => 3
```

Here VERSIONS=> 3 means when you call a particular record it will show you the last 3 records if you update the record.

b. Delete column family -

Use the following command to delete the 'family 1' column in table 'table 1'

Command -

```
hbase> alter 'table 1', NAME => 'family 1', METHOD=>
'delete'
```

c. Alter table properties -

We can change table-scope attributes such as MAX_FILESIZE, READONLY, MEMSTORE_FLUSHSIZE, DEFFERED_LOG_FLUSH etc. We have to put this at the end of the command.

Command -

```
hbase> alter 'table 1', MAX_FILESIZE => '126987455'
```

2. alter_async -

The only difference between alter and alter async is that alter async does not wait for all regions to receive the schema modifications before proceeding.

3. alter_status -

The command alter status returns the status of the alter command. It also shows the number of table regions that have been modified using the new schema.

Command -

```
hbase> alter_status 'table 1'
```

4. create -

We utilize it to create tables. A table name, a set of column family requirements (at least one), and, optionally, table configuration can also be included as parameters.

a. Create table -

Creating table with the namespace = ns1 and table name = 'table 1'

Command -

```
Hbase> create 'ns1:table 1', {NAME => 'family 1',
VERSIONS => 5}
```

b. Create table

Creating table with the namespace = default and

Command -

```
Hbase> create 'table 1' {NAME => 'family 1'}, {NAME =>
'family 2'}, {NAME => 'family 3'}
```

We can write this above command in short way also -

```
Hbase> create 'table 1' , 'family 1', 'family 2', 'family
3'
```

5. drop -

drop command will delete the table permanently.

Command -

```
hbase> drop 'table 1'
```

6. drop_all -

Here we use regex functions to drop the tables.

Command -

```
hbase> drop_all 't.*'
```

7. enable -

To enable a currently disabled table we can use the enable command.

Command -

```
hbase> enable 'table 1'
```

8. enable_all -

To enable a currently disabled table by matching regex we use the enable_all command.

Command -

```
hbase> enable_all 't.*'
```

9. exists -

To check the existence of an HBase table we use the exists command.

Command -

```
hbase> exists 'table 1'
```

10. get_table -

"Get table" retrieves the specified table name and also returns it as a real object that the user can manage.

Command -

```
hbase(main):014:0> t1 = get_table 'blog'
```

11. is_disabled -

To get to know whether a table is disabled or not we use the is_disabled command.

Command -

```
hbase> is_disabled 'table 1'
```

12. is_enabled -

To get to know whether a table is enabled or not we use the is_enabled command.

Command -

```
hbase> is_enabled 'table 1'
```

13. show_filters -

This command shows all the filters in an HBase.

Command -

```
hbase> show_filters
```

Data Manipulation commands

1. append -

append command appends a cell value at specified table/row/column coordinates.

Command -

```
hbase> append 'table 1', 'row 1', 'column 1', 'value',
ATTRIBUTES=> {'mykey'=>'myvalue'}
```

2. count -

The count command counts the number of the rows in a table.

Command -

```
hbase> count 'table 1', CACHE => 500
```

Default cache size is 10 rows.

```
hbase> count 'table 1', CACHE => 500, INTERVAL => 1000
```

The interval in this case is 1000, thus it will print when the count process reaches 1000.

3. delete -
   This command deletes the cell value at specified table/row/column and optionally timestamp coordinates.
   Command -

```
hbase> delete 'table 1', row 1', 'column 1', timestamp 1
```

4. deleteall -
   deleteall command deletes all cells in a given row. Pass a table name and row.
   Command -

```
hbase> deleteall 'table 1', 'row 1'
```

5. get_counter -
   get_counter returns a counter cell value at specified table/row/columns coordinates.
   Command -

```
hbase> get_counter 'table 1', 'row 1', 'column 1'
```

6. put -
   This command puts a cell ‘value’ at a specified table/row/column.
   Command -

```
hbase> put 'table 1', 'row 1', 'column 1', 'value'
```

7. truncate -
   This command disables, drops and recreates the specified table. Make sure, the schema will be present but not the records, after truncate of an HBase table. This command performs 3 functions -
   a. Disables table if already present.
   b. Drops table if already present.
   c. Recreates the mentioned table.
   Command -

```
hbase> truncate 'table 1'
```

8. truncate_preserve -
   This will retain the previous region boundaries defined by the pre-split.

Other HBase commands

- Admin commands -
  There are some admin commands which you should know. Those commands include - assignbalance_switch, balancer, catalogjanitor_enabled, catalogjanitor_run, catalogjanitor_switch, close_region, compact, flush etc.
- Replication commands -
  Replication commands include - add_peer, disable_peer, enable_peer, list_peer, list_replicated_tables, remove_peer, set_peer_tableCFs etc.
- Snapshot Commands -
  Snapshot commands include - clone_snapshot, delete_snapshot, list_snapshot, list_snapshot, rename_snapshot, restore_snapshot etc.
- Visibility labels commands -
  Visibility labels commands include - add_lables, clear_auths etc.

- Security commands -
  Security commands include - grant, revoke, user_permission etc.

CRUD Operations

Following are the commands for CRUD operations.

1. Creating a table -
   With the help of 'create' command you can create a table. Here you have to mention table name and column family name.
   Command -

```
create '<table name>', '<column family name>'
```

   Example -
   Now we will create a table of employee info.

```
hbase(main):001:0> create 'employee', 'personal data',
'professional data'
```

   For verification we can use the 'list' command to verify that our table got created or not.
   command -

```
hbase(main):001:0> list
```

   It will show you all the tables which successfully got created.

2. Inserting row -
   With the help of the 'put' command you can insert the first row.
   Command -

```
put '<table name>', 'row1' '<col family:colname>',
'<value>'
```

   Let's insert the first row values into our employee table.

```
hbase(main):001:0> put 'employee', '1', 'personal
data:name', 'iNeuron'
```

```
hbase(main):001:0> put 'employee', ‘1’, 'personal
data:city', 'Banglore'
```

3. Updating a row -

   You can update the existing cell value using the ‘put’ command.

   Command -

```
put '<table name>', 'row1' '<col family:colname>',
'<value>'
```

   Let’s change the existing value of personal data.

```
hbase(main):001:0> put 'employee', '1', 'personal
data:city', 'Pune'
```

4. Retrieving a row -

   To read the data from a table in HBase we use the ‘get’ command.

   Command -

```
get '<table name>', 'row1'
```

   Let’s retrieve the value of name in personal data in row 1.

```
hbase(main):001:0> get 'employee', 'row 1', {COLUMN =>
'personal data:name'}
```

5. Retrieving a range of rows -

   To retrieve a specific range of rows we use the ‘scan’ command.

   Command -

```
scan '<table name>', {COLUMNS=>['column name:column
family'], LIMIT=>1, STARTROW=>"2"}
```

6. Deleting row -

   To delete a row in the table we use the ‘delete’ command.

   Command -

```
delete '<table name>', '<row1>', '<column name>', '<time stamp>'
```

   Let’s delete the last city update.

```
hbase(main):001:0> delete 'employee', 'row 1', 'personal data:city', 1523655956552
```

7. Deleting a table -

   To delete a table use the ‘drop’ command. But before dropping/deleting any table you have to ‘disable’ it.

   Command -

```
disable '<table name>'
drop '<table name>'
```

   Let’s delete our employee table.

```
hbase(main):001:0> disable 'employee'
hbase(main):001:0> drop 'employee'
```

Filters in HBase

When reading data from HBase using Get or Scan operations, you can use custom filters to return a subset of results to the client. While this does not reduce server-side IO, it does decrease network bandwidth and the amount of data that the client must digest. Filters are often used with the Java API, but they can also be used with the HBase Shell for testing and debugging.

KeyOnlyFilter -

KeyOnlyFilter takes no arguments. It returns the key portion of each key-value pair.

Command -

```
KeyOnlyFilter()
```

PrefixFilter -

Accepts only one argument: a row key prefix. It only returns key-values in a row that begin with the provided row prefix.

Command -

```
PrefixFilter('<column_prefix>')
```

ColumnCountGetFilter -

It has one parameter, which is a limit. It yields the table's first limit number of columns.

Command -

```
ColumnCountFilter('<limit>')
```

PageFilter -

only accepts one argument: the page size. It returns the table's page size number of rows.

Command -

```
PageFilter('<page_size>')
```

RowFilter -

Takes a comparator and a compare operator. It uses the compare operator to compare each row key with the comparator, and if the comparison returns true, it returns all the key-values in that row.

Command -

```
RowFilter(<compareOperator>, '<row_comparator>')
```

FamilyFilter -
Takes a comparator and a compare operator. It uses the compare operator to compare each family name to the comparator, and if the comparison returns true, it returns all the key-values in that family.
Command -

```
FamilyFilter(<compareOperator>, '<family_comparator>')
```

SingleColumnValueFilter -
SingleColumnValueFilter needs a column family, a qualifier, a comparison operator, and a comparator are all required. If the given column cannot be discovered, the entire row's columns will be emitted. If the column is found and the comparator comparison returns true, all of the row's columns are emitted. The row will not be emitted if the condition fails.
Command -

```
SingleColumnValueFilter ('<family>', '<qualifier>',
<compareOperator>, '<comparator>')
```

Custom Filters -
Implementing the Filter class allows you to construct your own unique filter. All RegionServers must have the JAR (Java ARchive) installed.

Understanding the troubleshooting in HBase

While working with HBase, you may run into a variety of issues. We'll talk about a variety of issues and how to solve them.

Thrift server crashing

Due to a buffer overrun, a Thrift server may crash if it gets a large amount of incorrect data.

Cause -

Thrift server allocates RAM to examine the correctness of the data it receives. If there is a big amount of incorrect data, it may need to allocate more RAM than is available. It arises, however, due to a limitation in the Thrift library.

Solution -

Use the framed and compact transport protocols to avoid any crashes caused by buffer overruns. Because they may require changes to your client code, some protocols are disabled by default. hbase.regionserver.thrift.framed and hbase.regionserver.thrift.compact are two options to include in your hbase-site.xml. As shown in the XML below, set each of these to true. You may also use the hbase.regionserver.thrift.framed.max frame size in mb option to determine the maximum frame size.

```
<property>
  <name>hbase.regionserver.thrift.framed</name>
  <value>true</value>
</property>
<property>
<name>hbase.regionserver.thrift.framed.max_frame_size_in_
mb</name>
  <value>2</value>
</property>
<property>
  <name>hbase.regionserver.thrift.compact</name>
  <value>true</value>
</property>
```

Region server not initializing

Sometimes the master server initializes but region servers do not initialize.

Cause -

In dual communication between region servers and master servers, region servers continually tell Master servers of their IP addresses, which are 127.0.0.1

Solution -

Remove the master server name node from localhost, which is present in the hosts file.

Address not found issue

Sometimes zookeeper quorum throws an error that file address is not found.

Cause -

HBase attempts to establish a zookeeper server on some machines, but the machines are unable to identify the quorum configuration.

Solution -

We can replace a hostname with the hostname that is presented in the hostname

Files not found in root directory

When we create a root directory for hbase through hadoop DFS, we need to run the hbase migration script, the hbase migration script responds like no files in the root directory.

Cause -

It is possible that hbase's previous running instance has been initialized before only.

Solution -

We have to delete the hbase root directory. By Itself hbase creates and initializes the directory.

Zookeeper session expired

Cause -

HMaster or HRegion servers shutting down because of zookeeper expired

Solution -

We can increase the session timeout for zookeeper. By default the session timeout is 60 seconds so we can change the session timeout to 120 seconds.

```
<property>
   <name> zookeeper.session.timeout </name>
   <value>1200000</value>
</property>
<property>
   <name> hbase.zookeeper.property.tickTime </name>
   <value>6000</value>
</property>
```

HBase is using more disc space than expected

Cause -

HBase StoreFiles (also known as HFiles) are disk-based storage for HBase row data. Other data, such as write-ahead logs (WALs), snapshots, and data that would otherwise be erased but would be needed to restore from a stored snapshot, is stored on disc by HBase.

Solution -

- Snapshots - Contains one subdirectory per snapshot.
  Location - `/hbase/.snapshots`
  Troubleshooting - To remove a snapshot, use shell command and write delete_snapshot
- Logs - Contains hbase WAL files that are required to recover regions in the event of a RegionServer failure.
  Location - `/hbase/.logs`

Troubleshooting - When the contents of WALs have been validated to have been written to StoreFiles, they are removed. Do not manually remove them. Examine the HBase server logs to see why WALs are not being processed correctly if the size of any subfolder of `/hbase/.logs/` is rising.

- Old WALs - Contains hbase WAL files that have been written to disk.
  Location - `/hbase/logs/.oldWALs`
  Troubleshooting - To customize the length of time a WAL stays in the .oldWALs before it is removed, configure the `hbase.master.logCleaner.ttl` property, which defaults to 60000 milliseconds or 1 hour.