::Big Data::

Day 1

MySQL crashes after 2TB

latency - time to take to get the data and process the data

scalability- increasing ram or disk space

How to store Big Data ?

How to process Big Data ?

RDBMS - inadequate !

what to do ?

- MPP - Massively parallel processing systems - expensive

- BSP - Bulk Synchronous parallel system

Google - 1998 - Google search - Core of Google Search - GFS(Google file system), MapReduce, BigTable

Google published white-papers on GFS(Google file system), MapReduce, BigTable - 2004

Yahoo decided to bring code (GFS + MapReduce), hired Doug Cutting (Architect of Hadoop) - Java

HDFS + MapReduce = Hadoop

Hadoop project was handed over to Apache Software Foundation in 2006

distributed computing - Hadoop(Open Source) makes it easy - credit to Google

Hive was started by FB and was open source in 2008 - SQL engine for Hadoop

With Hadoop, using HDFS - store 10x more at the same cost

With Hadoop, using MapReduce - process 10x faster than RDBMS

Hadoop :: HDFS helps in storing ; MapReduce helps in processing

Hadoop Distributed File System (HDFS™): A distributed file system that provides high-throughput access to application data.

Hadoop YARN: A framework for job scheduling and cluster resource management.

Hadoop MapReduce: A YARN-based system for parallel processing of large data sets.

Hive™: A data warehouse infrastructure that provides data summarization and ad hoc querying.

Pig™: A high-level data-flow language and execution framework for parallel computation.

Spark™: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.

each computer is a node and together they form a cluster and work as one entity

Distributed computing :

- you have to manage resources at very node

- coordinate tasks

- if one node goes, the process should not get affected

GFS - Storage

MapReduce - processing data

Bigtable - database management

All these architecture abstract programmers from the complexity of distributed computing

In Hadoop :

GFS -HDFS - a file system to manage the storage of data

MapReduce - Hadoop MapReduce - parallel processing

Bigtable - Hbase

::Hadoop ecosystem::

- We can use Hadoop and related projects to achieve analytics on Big Data

- Hadoop systems are capable to query petabytes of scale of data

- A single Hadoop cluster can host 1000's of nodes

Hadoop = HDFS(storage) + MapReduce(processing)

What is YARN (yet another resource negotiator)?

MapReduce version 2

Hadoop 1.x - Old

Hadoop 2.x - Present (learning)

Hadoop 3.x - New

::HDFS ::

- abbreviated as Hadoop Distributed File System

- Manages the disks attached to the nodes within the cluster

- HDFS will internally split the data into small chunks (called as blocks), these blocks are going to be distributed and stored on different nodes within the cluster (DataNode)

- HDFS will only manage the distribution (distribution is automated and abstracted for the user) - (NameNode)

- HDFS uses replication to ensure data reliability and fault tolerance. Replication is also managed by HDFS

::YARN ::

- Abbreviated as Yet Another Resource Negotiator

- MapReduce was revamped into MapReduce V2 (called as YARN)

- YARN forms the layer for data processing (distributed by HDFS)

- YARN will allocate compute resources for data processing

- We look at minimized data movement, YARN makes an effort to put the computation on the node where the data resides (Data Locality)

- Because data is distributed on machines with compute power, computation can be sent directly to the machines storing the data

To choose the right hardware for a Hadoop cluster - since each node in a Hadoop cluster as well as processes data, those nodes need to be configured to satisfy both data storage and processing requirements

program under execution is a process

set of process is service

program is running in the background is called Daemon

to check if Hadoop is running, open terminal and run

- sudo jps

On the VM, we have

- Linux file system (local file system)

- Hadoop file system (HDFS)

Things that can be done in HDFS in Hadoop :

Create a file

Delete a file

Append a file

Create a directory

Update a file - No updates allowed in HDFS

HDFS datasets are immutable (Read only)

UI / CLI (command line interface)

::Commands on Terminal::

- pwd (present working directory)

- ls (what are the files available in home directory)

- cd (change directory)

ctrl + L = clears the screen

hadoop fs = list of function in hadoop

/ indicates root

- put used to put the data into Hadoop

- cp to copy file

- rm is used to remove

-D to override default configuration

block size is a maximum of 128MB

Any file that exceed 128MB becomes Blocks (file will get split)

Namenode will allocate each block of a file into different node

datanode are the location where the blocks are physically stored

Namenode decides which block goes to which node

################################################################################################################################################################

Day 2

All fs shell commands will be executed on the clients node

Three V's of Big Data (characteristics)

- Volume

- Variety

- Velocity

- Veracity

- Volatility

::HDFS Architecture ::

-> The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware.

-> Hardware failure is the norm rather than the exception.

-> Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

-> Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size

-> A file once created, written, and closed need not be changed (Immutable Datasets)

-> HDFS has a master/slave architecture.

-> An HDFS cluster consists of a single NameNode, a master node. The Namenode manages the file system and regulates access to files by clients. (clients are going to write files into the cluster)

-> In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on

-> HDFS allows user data to be stored in files

-> Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes

-> The NameNode makes all decisions regarding replication of blocks and allocation of nodes

-> The NameNode executes file operations like opening, closing, and renaming files and directories

-> The NameNode stores the file system MetaData (file Vs Block Mapping, Block Vs Node Mapping)

-> The DataNode are the actual storage points

-> The DataNode executes the NameNode's instructions

-> The Data blocks are replicated for fault tolerance, and this replication is at a block level

-> As Hadoop users, we deal with files (to access, also to process)

-> The existence of a single NameNode in a cluster greatly simplifies the architecture of the system

-> The system is designed in such a way that user data never flows through the NameNode

-> HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. It provides a CommandLine interface called FsShell that lets a user interact with the data in HDFS

-> HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks

-> HDFS is meant for sequential access to data

-> NameNode periodically receives a Heartbeat from each of the DataNodes in the cluster (default interval = 3 sec). Receipt of a Heartbeat implies that the DataNode is functioning properly

How to write a file into the cluster ?

::Anatomy of a file write ::

- Client connects to the NameNode, to create a new file in the HDFS, with no blocks associated with it initially

- The NameNode makes a record of the new file, and returns an instruction for the client to start writing data

- As the client writes the data, DFSOutputStream splits it into blocks, which is written to an internal queue initially (called as data queue)

- The data queue is consumed by the DataStreamer, which is responsible for asking the NameNode to allocate new blocks by picking a list of suitable DataNodes to store the replicas

- The list of DataNodes allocated, forms a "pipeline of DataNodes". For the standard replication factor of 3, there will be 3 DataNodes in the pipeline

- The DataStreamer streams the blocks to the first DataNode in the pipeline, and the first DataNode stores it on its disk, and forwards it to the second DataNode in the pipeline. Similarly, the second DataNode stores the block and forwards it to the third (and last) DataNode in the pipeline

- DFSOutputStream also maintains an interval queue of packets that are waiting to be acknowledged by DataNodes, called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the DataNodes on the pipeline

- When the client has finished writing data, it calls close() on the DFSOutputStream. The MetaData is committed and the file write is complete

How to read a file which is on the cluster ?

::Anatomy of a file read::

- The client opens the file it wishes to read by calling open() on the FileSystem object (is an instance of DistributedFileSystem)

- DistributedFileSystem calls the NameNode, to determine the locations of the blocks for the file

- For each block, the NameNode returns the addresses of the DataNodes that have a copy of that block

- The DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from

- The client then calls read() on the DFSInputStream

- DFSInputStream,then connects to the first DataNode for the first block in the file

- Blocks are read in sequence

- When the client has finished reading, it calls close() on the FSDataInputStream

::NameNode And SecondaryNameNode::

A closer look at the MetaData (residing on NN)

How is the File System MetaData stored ?

- The HDFS MetaData is stored by the NameNode

- The NameNode uses a transaction log called the EditLog to persistently record every change that occurs to file system MetaData (EditLog is a file on the OS file system of the machine running NN)

- The entire file system MetaData, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage(File System Image, also resides on the OS file system of the machine running NN)

- NN has 2 files -> EditLog, FsImage

- Looking from backup of the MetaData angle, we need to have the above two files backed up

Checkpointing Procedure :

- All about merging Editing to FsImage

- Who does it ?

- NN can do it (only if we restart NN)

- SNN does Checkpointing

- Regular Checkpoint interval - 1 hour by default

- Or 1 million transactions on the EditLog

SNN is not a hot backup for the NN

SNN is a single point of failure (risk)

- to minimize the risk

- Run NN on hardware which does not fail

- Reduce the checkpoint interval

What Is MapReduce ?

- A programming model for distributed datasets

- From a developer's angle, involves 2 phases

- Map Phase

- Reduce Phase

- The MapReduce framework abstract the complexity of IO from the developer

- Hadoop provides API's, we extend the classes from API in our code ad write MapReduce programs

- MapReduce programs are mostly written on Java

- MapReduce programs are submitted to the cluster for execution

- A MapReduce program under execution, is called a JOB

- It is the YARN's responsibility to handle the job execution

- YARN does so by allocating adequate compute resources for the job to complete execution

- YARN makes an effort to put the program on date (hence achieving data locality, minimized data movement and Fast processing). It is easy to move program to data, rather than moving data to where the program is being executed

- MapReduce works on (Key,Value) pairs. Ex. Welcome, 1 where Welcome - Key and 1 is the value

- A MapReduce program will have 3 classes defined

- Mapper class

- Reducer class

- main()

How does MapReduce work ?

- Map part of the node runs where the data resides(block)

- Map and Reduce methods under execution are the actual units of execution within the program

- We need compute resources for execution of these tasks

- The challenge is to acquire a compute resource on the same node where the data block resides in order to run a map task. YARN addresses this!

Understanding MapReduce with an example :

Example Application : The MapReduce WordCount program

WordCount is a simple application that counts the number of occurrences of each word in a given input set.

Input Set - /Sample/SampleFile.txt

Welcome to Hadoop

Learning Hadoop is fun

Hadoop Hadoop Hadoop is the buzz

Expected Output ?

\*Final Output

<Hadoop,5> <Learning,1> <Welcome,1> <buzz,1> <fun,1> <is,2> <the,1> <to,1>

\*Map Output

Hadoop - 1

to - 1

Welcome - 1

fun - 1

Hadoop - 1

is - 1

Learning - 1

Hadoop - 3

is - 1

the - 1

buzz - 1

The MapReduce program that we write to solve the WordCount problem would have a template as below :

Create a file called WorkCount.java, within the file

public class WordCount {

MyMapper extends Mapper {

Map() {

// Logic

}

}

MyMapper extends Reducer {

Reduce() {

// Logic

}

}

main() {

// Entry point for execution

// Configure job object

// Execute the program

}

}

################################################################################################################################################################

Day 3:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Assignment 1 :

What is

Big Data

Hadoop

OLTP

OLAP

RDBMS

HDFS

MapReduce

NameNode

DataNode

SecondaryNameNode

ResourceManager

NodeManager

Distributed File System

Parallel Processing

Commodity Hardware

JRE

JDK

a process ?

a daemon ?

a service ?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

YARN - \* cluster resource management which intents to run programs on data \*

MapReduce - an abstract complexity of distributed system

::MapReduce::

Steps involved in MapReduce :

- Input Split

- Map

- Shuffle & Sort

- Reduce

- Final Output

::Input Split::

- When HDFS distributed the data blocks, it did not respect the logical boundaries of the dataset (ex. A line might be split in-between and half the line might be on 1 node and another half on the other). While processing data, addressing this problem was challenging for any distributed computing framework.

- THe MapReduce framework understood this problem well and handled it with the "Input Split" functionality

- On each block of the dataset where the mapper would be running, the MapReduce framework creates an Input Split by respecting the logical boundaries of the dataset and then passing it to the Mapper

- We can be sure that our final output will not have any impact because of HDFS hard split

- Input Splits are logical (The data blocks are not physically changed after the execution completes)

- Input Splits are complex to be handled by the developer. Hence, the MapReduce framework handles it, and makes it easy for the developer to code the map logic with map() of the mapper class

\*\*Input Split is part of MapReduce

::Map::

- The MapReduce framework spawns a mapper for each input split(The number of map tasks = The number of Input Splits for a program)

- The Mapper class is inherited from the API and we overwrite the map() method (To apply logic on the dataset)

- Mostly we code the logic for transforming datasets in the map phase

- The map() method in the program will be the map task during execution

- The InputSplit reads one line at a time into a <Key,Value> pair and gives it to the map() = RecordReader

- map() will iterate for each line (record), and transform the dataset into <Key, value> pairs

- YARN will allocate a compute resource (CPU+RAM) for the execution of map

- Map transforms your data into a set of intermediate key, value pairs

- Input <K1,V1> --> Map --> List<K2,V2>

- The MapReduce framework operates exclusively on <Key,Value> pairs, that is, the framework views the input to the job as a set of <Key,Value> pairs and produces a set of <Key,Value> pairs as the output of the job

- The output of map is stored on the local file system of the node (where the map task was running)

For the WordCount example, the logic for map()

- Pick the value

- Tokenize (split by delimiter " " and convert into words)

- Mark every token (word) as the key, assign a value 1 (it occurred once)

<0, Welcome to Hadoop> --> Input (Map) --> <Welcome,1> <to,1>, <Hadoop,1>

<18, Learning Hadoop is fun> --> Input (Map) --> <Learning,1> <Hadoop,1> <is,1> <fun,1>

...

...

How to write the output of the map to the framework (Intermediate results)

--> context.write(k,v)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The Mapper code

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(Object key, Text value, Context context

) throws IOException, InterruptedException {

StringTokenizer itr = new StringTokenizer(value.toString());

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one);

}

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

::Shuffle & Sort::

- MapReduce makes the guarantee that the input to every reducer is sorted by key

- The process by which the system performs the sort, and transfers the map output to the reducers as inputs, is known as the shuffle

::Reduce::

- In this phase the Reduce method is called for each key

- The list of value pairs / key are in the grouped inputs

- The logic for aggregation (on the values) is written in the reduce() method

- The output of the reduce task is typically written to the FileSystem via context.write

- The output of the Reducer is not sorted

- For the WordCount example, the logic for reduce()

- Loop through the List (of values)

- Compute Sum (Add the values)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The Reducer Code

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public static class IntSumReducer

extends Reducer<Text,IntWritable,Text,IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,

Context context

) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The main() class - Entry point of the program (Execution begins from here)

- Login

Define MapReduce job

Set input and output locations

Set input and output formats (Optional)

Set Mapper and Reduce classes

Submit job

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The main code

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word count");

job.setJarByClass(WordCount.class);

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

################################################################################################################################################################

Day 4

::YARN Terminologies::

-> ResourceManager = 1 / cluster

-> NodeManager = 1 / DN

-> ApplicationsManager = One of the component of ResourceManager

-> Scheduler = One of the component of ResourceManager

-> ApplicationsMaster = 1 / job

-> Container = Compute resources (CPU+RAM)

-> YarnChiId = Container allocated for map task/ reduce task execution

job = Application (Both are same)

Within a single job !

# of map tasks = # of Input Splits (Most likely = no of blocks)

# f reduce tasks = 1

- The map and the reduce tasks are the actual units of execution - the compute resources are needed for these tasks

- In YARN terminologies, compute resource is termed as a "Container"

- Container = <CPU+RAM>

- For execution of map and reduce tasks, containers are needed

- The Master service of YARN (ResourceManager) is ultimate authority for allocation of the containers

sudo watch 'jps | grep -v jps | sort -k 2' - Used to check the nodes functions running in background

::YARN Architecture::

-> The ResourceManager and per-node slave, the NodeManager (NM), form the data-computation framework

-> The ResourceManager is the ultimate authority that allocates resources for all the Hadoop jobs

-> The ResourceManager has 2 components

- ApplcationsManager => The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container to start ApplicationsMaster

- Scheduler => The Scheduler is responsible for allocating resources to the various running jobs

-> The NodeManager is the agent of the ResourceManager, running on DN

-> The NodeManager is responsible for containers, monitoring their resource usage (CPU+RAM) and reporting the same to the ResourceManager (Scheduler)

-> The per-job ApplicationsMaster has the responsibility off negotiating appropriate resource containers from the scheduler, tracking their status and monitoring for progress

YARN Anatomy

-> The job is submitted on the client node. The submit() method on Job creates an internal JobSummitter

-> The new job ID is retrieved from the resource manager

-> The jars are created on the cluster with a higher replication

-> Finally, the job is submitted by calling submitApplication() on the resource manager

-> The applications manager component takes the request, and hands off the request to the scheduler (all this within resource manager)

-> The scheduler creates the first container for this job, called ApplicationMaster. The ApplicationMaster is monitored by the ApplicationsManager

-> The ApplicationsManager initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks

-> The ApplcationMaster retrieves the input split details from HDFS

-> It then creates a map task for each input split (The required containers are negotiated by ApplicationMaster against the ResourceManager). A container for reduce task is also created for each job

-> On job completion, the application master and the task container clean up their working state

-> Job information is archived by the job history server to enable later interrogation by users if desired

CREATE TABLE STOCKS(

EXCH STRING2(4),

SYMBOL STRING2(4),

YMD STRING2(4),

PRICE\_OPEN NUMBER(4),

PRICE\_HIGH NUMBER(4),

PRICE\_LOW NUMBER(4),

PRICE\_CLOSE NUMBER(4),

VOLUME NUMBER(4),

PRICE\_ADJ\_CLOSE NUMBER(4))

SELECT SYMBOL, MAX(PRICE\_CLOSE) FROM STOCKS GROUP\_BY(SYMBOL)

::Hive Architecture::

-> Data and schema are stored seperately, the data is validated against the schema when it is quired, a technique is called "Schema on Read" -> Flexibilty

-> Structure can be projected on to data already in storage (HDFS)

Htave an understanding of the following :

Inner join

Left outer join

Right outer join

Full outer join

################################################################################################################################################################

Day 5

The program is initially on the client node

RM takes the request - 2 componenets -

ApplicationsMaster takes it first and ask scheduler to give one container such that it creates a process for bookkeeping process

RM gives heartbeats to NN about the computation (CPU+RAM)

first container is called the Appmaster(Scheduler creates the container)

container allocate for map must be on the node where the data resides

the container is called as YARNchild

AM talks to NN where the locations of data reside

AM ask the scheduler for container for the ndoes that have the data

So container are created for the map process

AM with NN tracks the process of the containers that are used in the node where the data resides

kills the containers once over

asks container for reduce and ask NM to track the progress and kills the container once done

jar has to come from client node to some location in the cluster decided by NN

if jar is not in the same lication, YARNclid gets the jar file

YARN is about moving program to data

client node talk to NN to push the jar file somewere in the cluster

CREATE EXTERNAL TABLE employee(

name STRING,

salary STRING,

city STRING)

row format delimited

fields terminated by ','

location '/emp';

CREATE EXTERNAL TABLE mailid(

name STRING,

email STRING)

row format delimited

fields terminated by ','

location '/email';

SET hive.auto.convert.join=false;

#Inner join

SELECT \* FROM employee emp, mailid em WHERE emp.name = em.name;

#Left inner join

SELECT \* from employee emp LEFT JOIN mailid em ON emp.name = em.name;

#Right inner join

SELECT \* from employee emp RIGHT JOIN mailid em ON emp.name = em.name;

#Full outer join

SELECT \* from employee emp FULL OUTER JOIN mailid em ON emp.name = em.name;

::Apache Sqoop::

Apache Sqoop(TM) is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.

sqoop is a mapreduce engine

#Job 1

sqoop import \

--connect jdbc:mysql://localhost:3306/retail\_db \

--username retail\_dba -P \

--table categories \

--target-dir /retail\_dataset/categories \

-m 1

#Job 2

sqoop import \

--connect jdbc:mysql://localhost:3306/retail\_db \

--username retail\_dba -P \

--table departments \

--target-dir /retail\_dataset/departments \

-m 1

#Job 3

sqoop import \

--connect jdbc:mysql://localhost:3306/retail\_db \

--username retail\_dba -P \

--table products \

--target-dir /retail\_dataset/products \

-m 1

#Job 4

sqoop import \

--connect jdbc:mysql://localhost:3306/retail\_db \

--username retail\_dba -P \

--table orders \

--target-dir /retail\_dataset/orders \

-m 1

#Job 5

sqoop import \

--connect jdbc:mysql://localhost:3306/retail\_db \

--username retail\_dba -P \

--table order\_items \

--target-dir /retail\_dataset/order\_items \

-m 1

#Job 6

sqoop import \

--connect jdbc:mysql://localhost:3306/retail\_db \

--username retail\_dba -P \

--table customers \

--target-dir /retail\_dataset/customers \

-m 1

create database retail;

use retail;

#Create table 1

CREATE EXTERNAL TABLE departments(

department\_id INT,

department\_name STRING)

row format delimited

fields terminated by ','

location '/retail\_dataset/departments';

#Create table 2

CREATE EXTERNAL TABLE categories(

category\_id INT,

department\_id INT,

category\_name STRING)

row format delimited

fields terminated by ','

location '/retail\_dataset/categories';

#Create table 3

CREATE EXTERNAL TABLE products(

product\_id INT,

product\_category\_id INT,

product\_name STRING,

product\_description STRING,

product\_price FLOAT,

product\_image STRING)

row format delimited

fields terminated by ','

location '/retail\_dataset/products';

#Create table 4

CREATE EXTERNAL TABLE order\_items(

order\_item\_id INT,

order\_item\_order\_id INT,

order\_item\_product\_id INT,

order\_item\_quantity INT,

order\_item\_subtotal FLOAT,

order\_item\_product\_price FLOAT)

row format delimited

fields terminated by ','

location '/retail\_dataset/order\_items';

#Create table 5

CREATE EXTERNAL TABLE orders(

order\_id INT,

order\_date STRING,

order\_customer\_id INT,

order\_status STRING)

row format delimited

fields terminated by ','

location '/retail\_dataset/orders';

#Create table 6

CREATE EXTERNAL TABLE customers(

customer\_id INT,

customer\_fname STRING,

customer\_lname STRING,

customer\_email STRING,

customer\_password STRING,

customer\_street STRING,

customer\_city STRING,

customer\_state STRING,

customer\_zipcode STRING)

row format delimited

fields terminated by ','

location '/retail\_dataset/customers';

case 1 :

Return the count of COMPLETED and CLOSED orders

> select count(order\_id) from orders where order\_status='COMPLETE' OR order\_status='CLOSED';

Case 2 :

Return the orders with status SUSPECTED\_FRAUD

> select order\_id from orders where order\_status='SUSPECTED\_FRAUD';

Case 3 :

Sort products by product\_price descending

> select product\_name,product\_price from products order by product\_price desc;

Case 4 :

Sort products data by product category and then product price descending

> select product\_name, product\_category\_id, product\_price from products order by product\_category\_id asc, product\_price desc;

Case 5 :

For closed and complete orders with order revenue over $1000, get the order\_id, date, status and order revenue

SELECT o.order\_id, o.order\_date, o.order\_status, sum(oi.order\_item\_subtotal) FROM orders o, order\_items oi WHERE o.order\_id = oi.order\_item\_id AND

o.order\_status='COMPLETE' OR o.order\_status='CLOSED' group by o.order\_id, o.order\_date, o.order\_status having sum(oi.order\_item\_subtotal) > 1000;

SELECT o.order\_id, o.order\_date, o.order\_status, sum(oi.order\_item\_subtotal) FROM orders o JOIN order\_items oi ON order\_id = order\_item\_id WHERE o.order\_status='COMPLETE' OR o.order\_status='CLOSED' GROUP BY o.order\_id, o.order\_date, o.order\_status HAVING sum(oi.order\_item\_subtotal) > 1000;

::Impala::

Faster version to run SQL quires

impala-shell = to enter impala

invalidate metadata; (to get all the data from hive metastore)

sqoop

hive

impala

################################################################################################################################################################

Day 6

Use Hbase when you need operational capabilities on the existing decision support system

Hbase is a NoSQL

NoSQL -> Non Relational Databases

Most NoSQL databases are distributed

Ability to tackle Big Data

How many NoSQL system ?--> 100+

Ex: MongoDB, Cassandra, Couch DB, Dynamo, Hbase....

Why are NoSQL systems Non Relational ?

-> Not ACID compliant (Atomicity, Consistency, Isolation, Durability)

Hadoop Limitations :

-> Unstructured data

- Hadoop stores data in HDFS

- The data in HDFS is Unstructured

- Unlike databases, HDFS data doesn't have any schema

- Its basically in the form of files. Ex: Text file, Log file, Video/Audio File.

- There's no concept of rows/columns. There are no tables.

- But unlike DB's, Hadoop will not enforce the schema or any constraints on these rows/tables

-> No random access

- Applications that use DB require random access i.e the ability to create,access and modify individual rows of a table. This is not possible with Hadoop

- HDFS is optimal for storing large files. MapReduce is optimal for processing these files as a whole

- If an HDFS file consists of many rows in a table, There is no provision to access or modify a specific row without processing the entire file

-> High latency

- Application require low latency, any operations like inserting, updating or deleting data should occur as fast as possible

- All processing in Hadoop occurs via MapReduce tasks on complete files. Even on large clusters, these tasks might take minutes or hours at times

-> Not ACID compliant

- DB's are the source of truth for the data that they store

- DB's guarantee ACID properties to maintain the integrity of their data

- ACID -> Atomicity = Operations (aka transactions) must be all-or-nothing. Ex. Cash withdrawal from an ATM

Consistency = Any changes to the DB must not violate any specified DB constraints

Isolation = If multiple/concurrent operations occur, the result is as if these operations are applied in sequence

Durability = Once a transaction is executed, the changes are permanent

ACID guarantee require that the DB management system is aware of the structure and contents of the data. HDFS being just a file storage system has no such awareness

All these limitations makes Hadoop unsuited for transaction processing

Hadoop cannot run on OLTP but it is an OLAP (used only for analyzing)

Only ACID compliant systems can be used for OLTP

::HBase::

BigTable (data management) = HBase

- It is a distributed DB management system thats part of the Hadoop ecosystem

- HBase uses HDFS to store its underlying data

- HBase has the architecture benefits of HDFS

- Distributed storage

- Fault tolerance

HBase Vs RDBMS

- In traditional RDBMS, all operations like creating, inserting, updating rows are done using SQL (HBase does not support SQL)

- Only CRUD operations, HBase only supports a basic set of operations(Create-Read-Update-Delete). All these operations have to be applied at a row level.

- HBase does not support any operations across rows or across tables. This means that you cannot perform operations like joins, group by etc

- De-normalized

- HBase tables are not designed using a relational data model

- All the data pertaining to an entity is stored in 1 row(i.e tables are De-normalized)

- Column oriented storage

- HBase has a special kind of data model

- ACID at a row level

- HBase is ACID compliant for limited kinds of transactions

::Column oriented storage::

- Each row represent a data point (row oriented) It has a fixed schema

- Each cell represent a data point (column oriented)

- Data is stored in a map (key value pairs)

key = <Row id, Col id>

value = <data>

- An HBase table is in fact a sorted map

- In RDBMS table, each attributes becomes a new column which results in table that are very sparse. In RDBMS, sparse tables utilizes disk space even for these empty cells

- In a column-oriented store, these cells can be skipped completely

Column oriented storage has some powerful advantages :

- You can store really sparse table very efficiently

- You can accommodate dynamically changing attributes

- The schema for a row id is not fixed, you can keep changing it i.e Add or Remove new col id's

- You can accommodate dynamically changing attributes

::De-normalized::

In HBase data is stored in De-Normalized manner. Instead of making separate tables (in RDBMS), we can keep all the data in one table(using structure or array)

::Only CRUD operations::

- HBase architecture is designed that you can get random read-write access to a specific row

- Unlike traditional RDBMS, HBase does not support SQL but NoSQL

- HBase only supports a limited set of operations

CREATE - add a new value to the table

READ - read the value for a specific row id and col id

UPDATE - update the value for a specific row id and col id

DELETE - delete the value for a specific row id and col id

- All HBase operations deal with a specific row

- HBase does not support any operations across tables. No joins, no foreign key or constraints

- HBase does not support any operations across row id's. No grouping/aggregation

- This is another reason why De-normalization is important in HBase

- All the data needed to describe an entity should be self-contained with its row id

::ACID at a row level::

- HBase is ACID compliant, but only at a row id level. Ex. Atomicity

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

HDFS:

To check the status of HDFS daemons

sudo service hadoop-hdfs-namenode status

sudo service hadoop-hdfs-datanode status

sudo service hadoop-hdfs-secondarynamenode status

Starting HDFS Daemons

sudo service hadoop-hdfs-namenode start

sudo service hadoop-hdfs-datanode start

sudo service hadoop-hdfs-secondarynamenode start

Stopping HDFS Daemons

sudo service hadoop-hdfs-namenode stop

sudo service hadoop-hdfs-datanode stop

sudo service hadoop-hdfs-secondarynamenode stop

YARN:

To check the status of YARN Daemons

sudo service hadoop-yarn-resourcemanager status

sudo service hadoop-hdfs-nodemanager status

Starting YARN Daemons

sudo service hadoop-yarn-resourcemanager start

sudo service hadoop-hdfs-nodemanager start

Stopping YARN Daemons

sudo service hadoop-yarn-resourcemanager stop

sudo service hadoop-hdfs-nodemanager stop

HBase:

To check the status of HBase Daemons

sudo service hbase-master status

sudo service hbase-regionserver status

Starting HBase Daemons

sudo service hbase-master start

sudo service hbase-regionserver start

Stopping HBase Daemons

sudo service hbase-master stop

sudo service hbase-regionserver stop

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

::Hbase Architecture::

A sorted nested map

<Row id,

columnFamily,

<Column,

<Timestamp,value>>>

- When you read data from HBase, it performs a lookup for the specific row id

- When you write data to HBase, it needs to insert the row id in the right place, so the rows are sorted

- HBase does this by using Region Servers

::Region Server::

- Row id's in a tale are divided into ranges called regions

- Each region is handled by a region server

- Regions server handles all read-write operations to Regions that are allotted to it

- Initially all writes are stored in memory

- whenever there is a new change, the data is updated in the Memstore and a change log is written to disk

- The WriteAheadLog is created for recovery in case the region server crashes

- Periodically the Memstore gets full, and the data in Memstore is flushed to disk

- The data for a row key is either in the Memstore or in a HFile

- HFile is stored in HDFS

- HDFS will break up the HFile into blocks and store it on different nodes

Region server has the following components :

WAL - write ahead log(change log) = required for fault tolerance

Memstore - In memory place holder for data (data)

HFile - resides on DataNodes (data)

When you try to read/insert data

- the region server containing the row key is identified

- the region server will lookup the Memstore or the HFile and do the needful

- Clients interact directly with a region server handling the relevant row keys

- They need t know which region server their row key is handled by

- Zoo-keeper(Master) helps clients lookup the relevant region server for a specific row id

- The Master assigns regions to region servers manages load balancing etc

In HBase

list - list of all tables

Create table using HBase Shell

create 'notifications','attributes','metrics'

name of table column families

Inserting values using HBase shell

put 'notifications',2,'attributes:for\_user','chaz'

row id value

column name is specified along with its column family

put operations is used for inserting and updates values

put 'notifications',2,'metrics:open','0'

################################################################################################################################################################

Day 7

HDFS - NN,SNN,DN

HBase - HM,RS,ZK

Data - table - Distributed - RS

On every RS, there are 2 paces where the data is stored

HFile (storage)

Memstore (In Memory)

WAL - Recovery - you should not be worried !

data at rest

data in motion

Hadoop MapReduce is capable for processing data at rest

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

::MapReduce Recap::

Welcome to Hadoop | |

learning Hadoop is fun | -> Input Split -> | > RecordReader <K,V>

Hadoop Hadoop Hadoop is the buzz | |

<0, Welcome to Hadoop> --> Input (Map) --> <Welcome,1> <to,1>, <Hadoop,1>

<18, Learning Hadoop is fun> --> Input (Map) --> <Learning,1> <Hadoop,1> <is,1> <fun,1>

> Shuffle & Sort

<Hadoop,{1,1,1,1,1}>

<Learning,{1}

.........

> Reduce()

Aggregation happens here from the above list by summing

<Hadoop,5>

<learning,1>

.........

> RecordWriter

> HDFS Output Directory

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Context - Handle to the environment. Context is responsible for taking map outputs and passing on to reduce, taking reduce outputs, passing on to RecordWriter

\*\*Write python program instead of creating jar file\*\*

RecordReader - Input to Map

context - takes output of map, gives input to reduce (after shuffle & sort)

RecordWriter - Output of reduce, gives it to HDFS

If we are using the MapReduce API (built in Java), we don't need to write code for above steps

MapReduce programs can be written in other languages too! (Ex. Python, C++, Ruby, Perl etc...)

We are required to code the logic in our programs (No API)

Hadoop streaming (plug-in) - it is a python interface for MapReduce

Hadoop streaming is a utility which allows users to create and run MapReduce jobs with any executables (Ex. Python)

Hadoop streaming is a utility that comes with the Hadoop Distribution

Both the mapper and the reducer are executable that read the input from stdin (line by line) and emit the output to stdout

yarn jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar

yarn jar \

/usr/lib/hadoop-mapreduce/hadoop-streaming.jar \

-file /home/cloudera/Desktop/Labs/map.py -mapper map.py \

-file /home/cloudera/Desktop/Labs/reduce.py -reducer reduce.py \

-input /Sample/SampleFile.txt \

-output /SamplePy

Python code for stocks :

yarn jar \

/usr/lib/hadoop-mapreduce/hadoop-streaming.jar \

-file /home/cloudera/Desktop/Labs/Map.py -mapper Map.py \

-file /home/cloudera/Desktop/Labs/Reduce.py -reducer Reduce.py \

-input /Stocks/NYSE\_daily\_prices\_Q.csv \

-output /Sample2Py

::PIG::

- It is an open-source technology that offers a high-level mechanism for parallel programming of MapReduce jobs to be executed on Hadoop clusters

- It is a part of the Hadoop ecosystem, ideal for use with unstructured data where schema's are inconsistent or unknown

- PIG doesn't require data to be Loaded into tables first it can operate on data as soon as it is copied into HDFS

- PIG is great for getting data ready for data warehouse from poorly structured data sources such as web logs that makes it a great complement to hive

- PIG is a procedural data flow language and can be stores in one or more systems

- In PIG Latin, the user describes exactly how to process the data

[cloudera@quickstart Labs]$ pig -x local

[cloudera@quickstart Labs]$ pig -x mapreduce

grunt> fs -mkdir /test

grunt> fs -put /Stocks/\* /test/

put: `/Stocks/\*': No such file or directory

grunt> fs -cp /Stocks/\* /test/

grunt>

grunt>

A = LOAD '/test' using PigStorage(',');

B = FOREACH A GENERATE $1, $6;

STORE B INTO '/OP9';

fs -cat /OP9/part-m-00000

# PIG way for stock

A = LOAD '/test' using PigStorage(',');

B = Group A BY $1;

C = FOREACH B GENERATE group, MAX(A.$6);

STORE C INTO '/OP12';

fs -cat /OP12/part-m-00000

PIG Latin -> Logical Plan -> Physical Plan (blueprint for execution) -> MapReduce Job -> HDFS

PIG will start a job only when we enter a trigger (STORE,DUMP). Till then PIG will not execute

stock\_records = LOAD '/test' using PigStorage (',') AS (exchange:chararray,symbol:chararray,date:datetime,open:float,high:float,low:float,close:float,volume:int,adj\_close:float);

group\_by\_sym = Group stock\_records BY symbol;

max\_closing = FOREACH group\_by\_sym GENERATE group, MAX(stock\_records.close);

STORE max\_closing INTO '/OP12';

DUMP max\_closing;

fs -cat /OP12/part-m-00000

::Case sensitivity::

-The names (aliases) of relations (A,B,C) and fields (f1, f2, f3) are case sensitive

- The names of Pig Latin functions (PigStorage,COUNT) are case sensitive

- Keywords LOAD, USING, AS, GROUP, BY, FOREACH, GENERATE, and DUMP are case insensitive. They can also be written as load, using, group,by etc

- Fields are referred to by positional notation or by name (alias). Positional notation is generated by the system. Positional notation is indicated with the dollar sign ($) and begins with zero (0)

- Use Schema's to assign types to fields. If you don't assign types, fields default to type bytearray

PIG is OMNIVOROUS - it will consume anything - if a schema is given, PIG will make use of it - if a schema is not give, PIG will still process the data guessing along the way

pig /home/cloudera/Desktop/Labs/MaxClosePrice.pig

#From local File system:

stock\_records = LOAD '/home/cloudera/Desktop/Labs/NYSE\_daily\_prices\_Q.csv' using PigStorage (',') AS (exchange:chararray,symbol:chararray,date:datetime,open:float,high:float,low:float,close:float,volume:int,adj\_close:float);

group\_by\_sym = Group stock\_records BY symbol;

max\_closing = FOREACH group\_by\_sym GENERATE group, MAX(stock\_records.close);

STORE max\_closing INTO '/home/cloudera/Desktop/Labs/';

DUMP max\_closing;

import sys

res = {}

key = ""

val = ""

for line in sys.stdin:

line = line.strip()

key.val = line.split('\t')

val = float(val)

if key in res:

value = res{key}

if value > val:

res{key} = value

else

res{key} = val

p <- plot\_ly(data = df1,x = ~Date,type = "candlestick",open = ~AAPL.Open,close=~AAPL.Close,

high=~AAPL.High,low=~AAPL.Low) %>% layout(title="Basic Candlestick Chart")

################################################################################################################################################################