

PROJECT DOCUMENTATION

Name: Dinesh.R
Student id: S180030100342
Batch: PDTBD

Index

Sl.no	Content	Pg.no
1	Introduction to Big Data	3
2	Hadoop	3
3	HDFS	4
4	Yarn	8
5	Map Reduce	10
6	Hive	12
7	Pig	13
8	Sqoop	14
9	Project: H1B-Case study	15

1) Introduction to Big Data:

Big data can be defined as “extremely large data sets that may be analysed computationally to reveal patterns, trends, and associations, especially relating to human behaviour and interactions.”

Big data is data sets that are so voluminous and complex that traditional data processing application software are inadequate to deal with them. Big data challenges include capturing data, data storage, data analysis, search, sharing, transfer, visualization, querying, updating and information privacy. There are five V's associated with Big data which form the factors of Big data and they are as follows.

Volume

The quantity of generated and stored data. The size of the data determines the value and potential insight- and whether it can actually be considered big data or not.

Variety

The type and nature of the data. This helps people who analyse it to effectively use the resulting insight.

Velocity

In this context, the speed at which the data is generated and processed to meet the demands and challenges that lie in the path of growth and development.

Variability

Inconsistency of the data set can hamper processes to handle and manage it.

Veracity

The data quality of captured data can vary greatly, affecting the accurate analysis.

2) Hadoop:

Apache Hadoop is an open-source software framework used for distributed storage and processing of dataset of big data using the MapReduce programming model. It consists of computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common occurrences and should be automatically handled by the framework.

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part which is a MapReduce programming model. Hadoop splits files into large blocks and distributes them across nodes in a cluster. It then transfers packaged code into nodes to process the data in parallel. This approach takes advantage of data locality, where nodes manipulate the data they have access to. This allows the dataset to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are distributed via high-speed networking.

The base Apache Hadoop framework is composed of the following modules:

- Hadoop Common – contains libraries and utilities needed by other Hadoop modules;

- Hadoop Distributed File System (HDFS) – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- Hadoop YARN – a platform responsible for managing computing resources in clusters and using them for scheduling users' applications; and
- Hadoop MapReduce – an implementation of the MapReduce programming model for large-scale data processing.

The term Hadoop has come to refer not just to the aforementioned base modules and sub-modules, but also to the ecosystem, or collection of additional software packages that can be installed on top of or alongside Hadoop, such as Apache Pig, Apache Hive, Apache HBase, Apache Phoenix, Apache Spark, Apache ZooKeeper, Cloudera Impala, Apache Flume, Apache Sqoop, Apache Oozie, and Apache Storm.

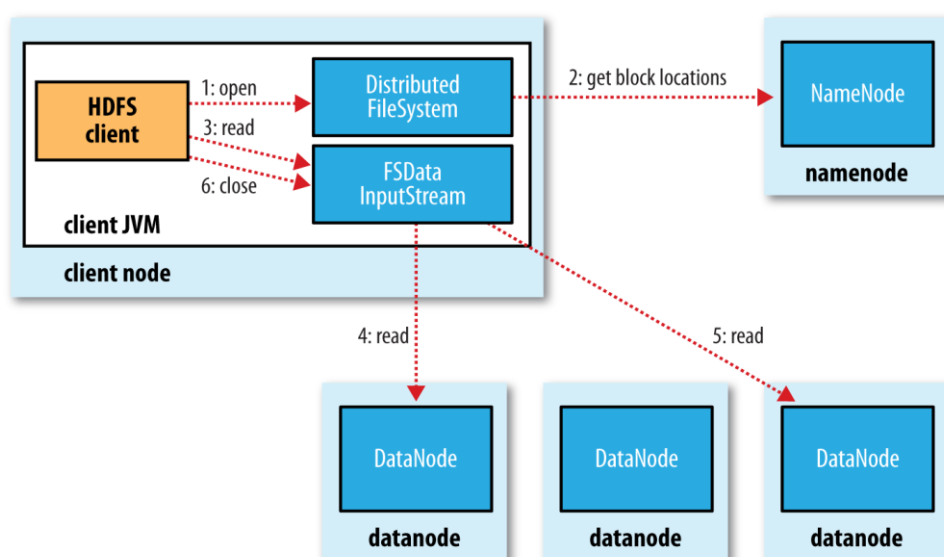
Apache Hadoop's MapReduce and HDFS components were inspired by Google papers on their MapReduce and Google File System.

The Hadoop framework itself is mostly written in the Java programming language, with some native code in C and command line utilities written as shell scripts. Though MapReduce Java code is common, any programming language can be used with "Hadoop Streaming" to implement the "map" and "reduce" parts of the user's program. Other projects in the Hadoop ecosystem expose richer user interfaces.

3) HDFS:

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called distributed filesystems. Since they are network based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss.

Hadoop comes with a distributed filesystem called HDFS, which stands for Hadoop Distributed Filesystem. HDFS is Hadoop's flagship filesystem and is the focus of this chapter, but Hadoop actually has a general purpose filesystem abstraction, so we'll see along the way how Hadoop integrates with other storage systems (such as the local filesystem and Amazon S3).

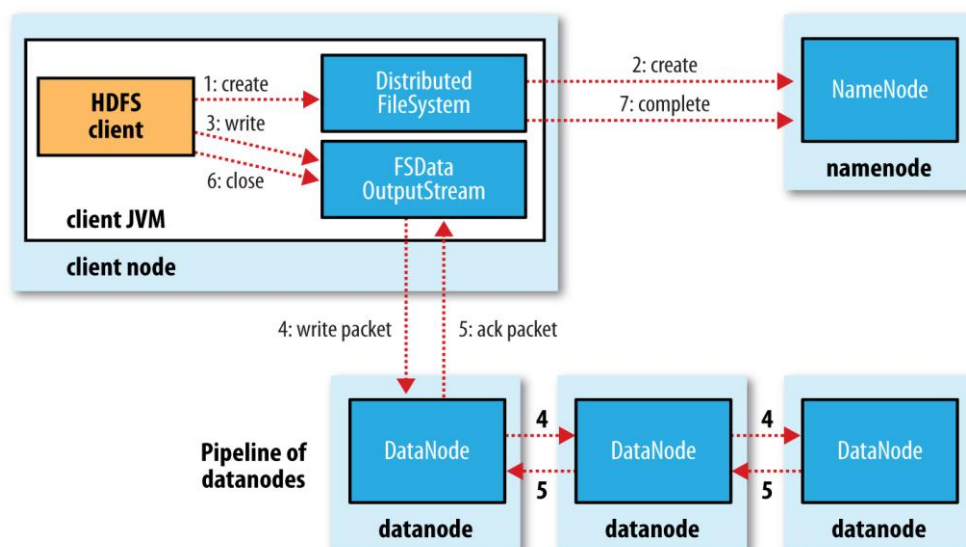


HDFS File Read operation

The client opens the file it wishes to read by calling `open()` on the `FileSystem` object, which for HDFS is an instance of `DistributedFileSystem`.

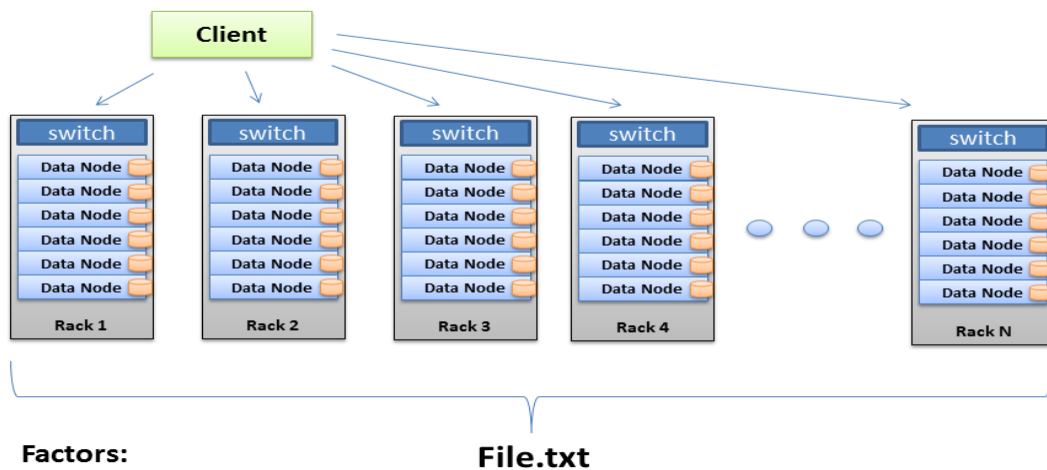
`DistributedFileSystem` calls the namenode, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client. If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the. The `DistributedFileSystem` returns an `FSDDataInputStream` (an input stream that supports file seeks) to the client for it to read data from. `FSDDataInputStream` in turn wraps a `DFSInputStream`, which manages the datanode and namenode I/O. The client then calls `read()` on the stream (step 3). `DFSInputStream`, which has stored




the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls `read()` repeatedly on the stream (step 4). When the end of the block is reached, `DFSInputStream` will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream. Blocks are read in order, with the `DFSInputStream` opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls `close()` on the `FSDDataInputStream` (step 6). During reading, if the `DFSInputStream` encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The `DFSInputStream` also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, the `DFSInputStream` attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode. One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

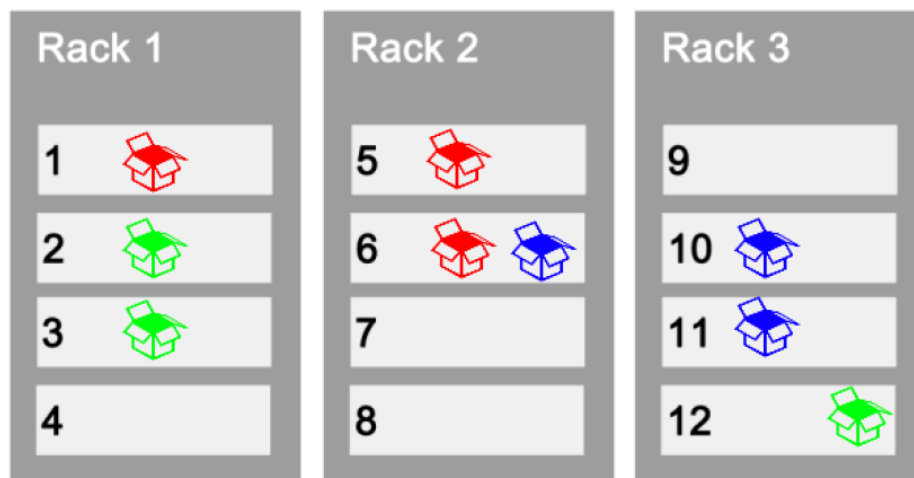


The client creates the file by calling `create()` on `DistributedFileSystem`. `DistributedFileSystem` makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an `IOException`. The `DistributedFileSystem` returns an `FSDDataOutputStream` for the client to start writing data to. Just as in the read case, `FSDDataOutputStream` wraps a `DFSOutputStream`, which handles communication with the datanodes and namenode. As the client writes data (step 3), the `DFSOutputStream` splits it into packets, which it writes to an internal queue called the 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 forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The `DataStreamer` streams the packets to the first datanode in the pipeline, which stores each packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4). The `DFSOutputStream` also maintains an internal 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 in the pipeline (step 5). If any datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode reappears later on. The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes. The remainder of the block's data is written to the good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal. It's possible, but unlikely, for multiple datanodes to fail while a block is being written. As long as `dfs.namenode.replication.min` replicas (which defaults to 1) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (`dfs.replication`, which defaults to 3). When the client has finished writing data, it calls `close()` on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of (because `Data Streamer` asks for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

Client writes Span the HDFS Cluster



Block A : 
Block B : 
Block C : 



Default Placement Algorithm

- Data Locality
- Network Traffic
- Redundancy

Rack Awareness in HDFS

In a large cluster of Hadoop, in order to improve the network traffic while reading/writing HDFS file, namenode chooses the datanode which is closer to the same rack or nearby rack to Read/Write request. Namenode achieves rack information by maintaining the rack id's of each datanode. This concept that chooses closer datanodes based on the rack information is called Rack Awareness in Hadoop.

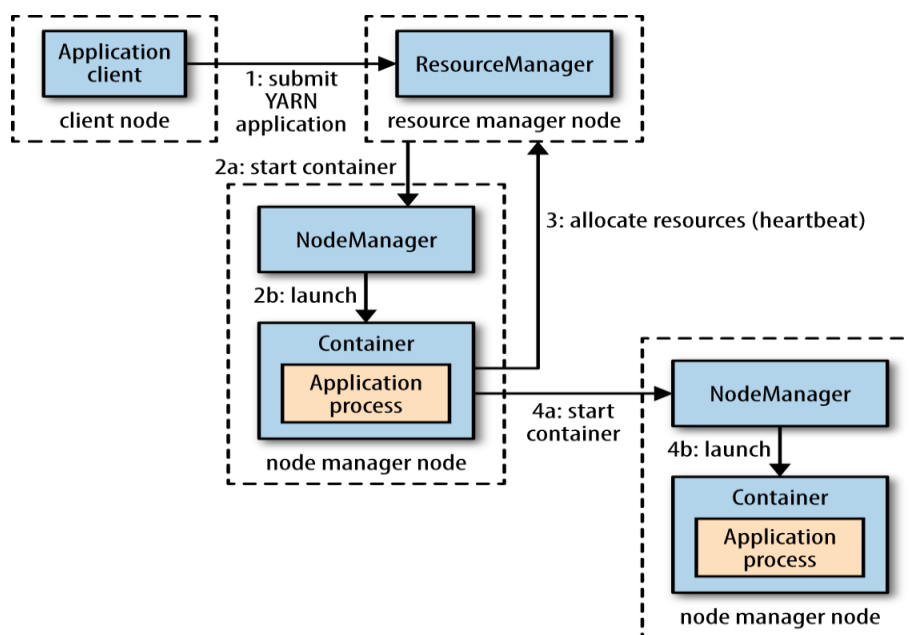
Rack awareness is having the knowledge of Cluster topology or more specifically how the different data nodes are distributed across the racks of a Hadoop cluster. Default Hadoop installation assumes that all data nodes belong to the same rack.

In Big data Hadoop, rack awareness is required for below reasons:

- To improve data high availability and reliability.

- Improve the performance of the cluster.
- To improve network bandwidth.
- Avoid losing data if entire rack fails though the chance of the rack failure is far less than that of node failure.
- To keep bulk data in the rack when possible.
- An assumption that in-rack id's higher bandwidth, lower latency.

4) Yarn:



Yarn Architecture

Apache YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system. YARN was introduced in Hadoop 2 to improve the MapReduce implementation, but it is general enough to support other distributed computing paradigms as well. YARN provides APIs for requesting and working with cluster resources, but these APIs are not typically used directly by user code. Instead, users write to higher-level APIs provided by distributed computing frameworks, which themselves are built on YARN and hide the resource management details from the user. The situation is illustrated in Figure, which shows some distributed computing frameworks (MapReduce, Spark, and so on) running as YARN applications on the cluster compute layer (YARN) and the cluster storage layer (HDFS and HBase).

There is also a layer of applications that build on the frameworks shown in Figure. Pig, Hive, and Crunch are all examples of processing frameworks that run on MapReduce, Spark, or Tez (or on all three), and don't interact with YARN directly.

Secondary namenode and Standby namenode:

Secondary NameNode in hadoop is a specially dedicated node in HDFS cluster whose main function is to take checkpoints of the file system metadata present on namenode. It is not a backup namenode. It just checkpoints namenode's file system namespace. The Secondary NameNode is a helper to the primary NameNode but not replace for primary namenode.

As the NameNode is the single point of failure in HDFS, if NameNode fails entire HDFS file system is lost. So in order to overcome this, Hadoop implemented Secondary NameNode whose main function is to store a copy of FsImage file and edits log file.

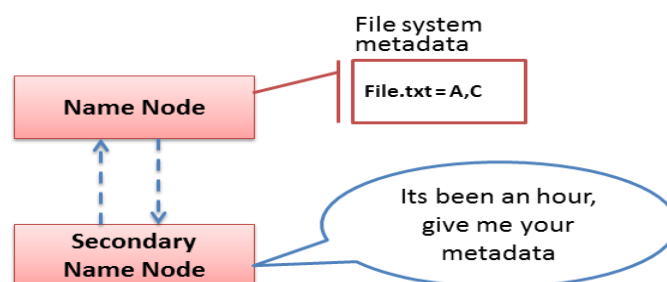
FsImage is a snapshot of the HDFS file system metadata at a certain point of time and EditLog is a transaction log which contains records for every change that occurs to file system metadata. So, at any point of time, applying edits log records to FsImage (recently saved copy) will give the current status of FsImage, i.e. file system metadata.

Whenever a NameNode is restarted, the latest status of FsImage is built by applying edits records on last saved copy of FsImage. Since, NameNode merges FsImage and EditLog files only at start up, the edits log file might get very large over time on a busy cluster. That means, if the EditLog is very large, NameNode restart process result in some considerable delay in the availability of file system. So, it is important keep the edits log as small as possible which is one of the main functions of Secondary NameNode.

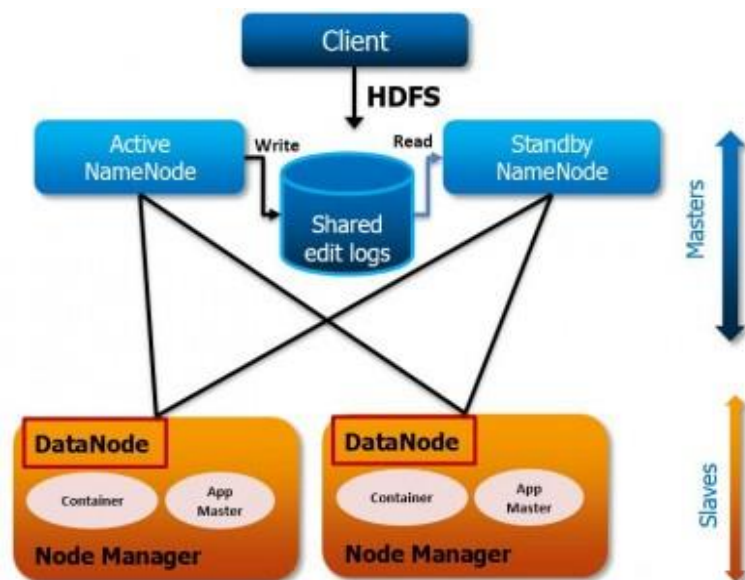
Secondary NameNode is not a true backup Namenode and cann't serve primary NameNode's operations.

It usually runs on a different machine than the primary NameNode since its memory requirements are same as the primary NameNode.

Secondary Name Node



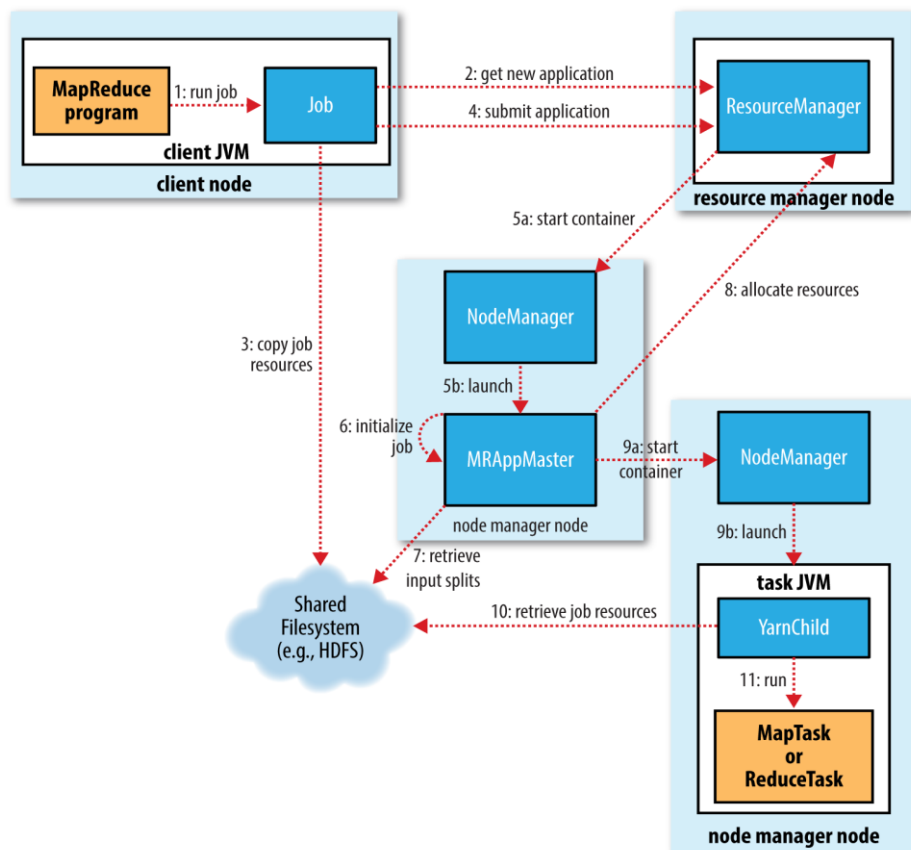
- Not a hot standby for the Name Node
- Connects to Name Node every hour*
- Housekeeping, backup of Name Node metadata
- Saved metadata can rebuild a failed Name Node



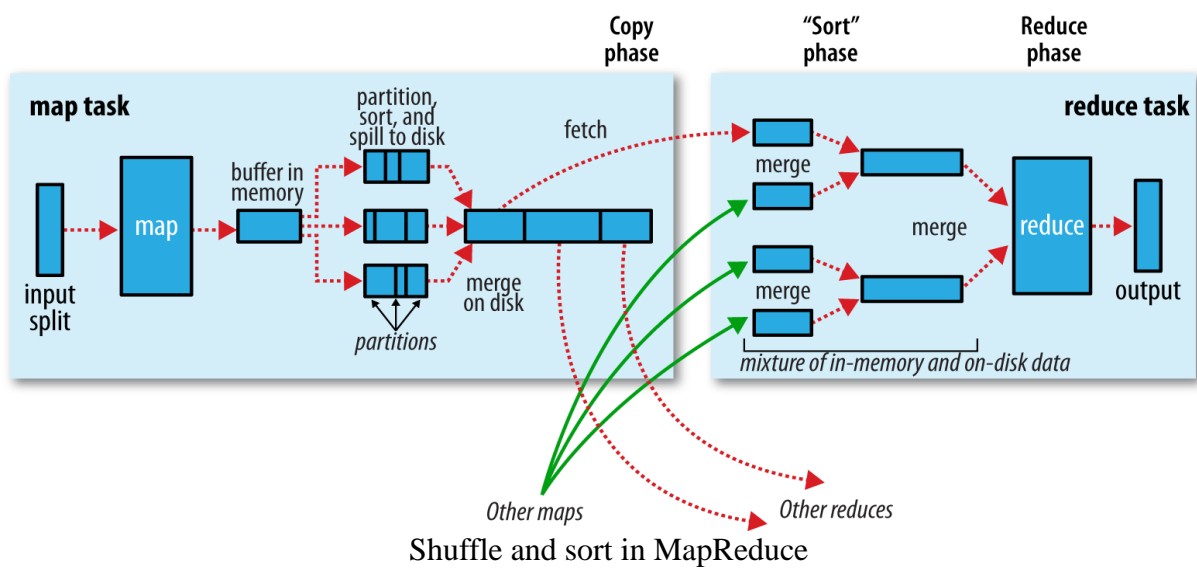
5) Map Reduce:

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages. Most importantly, MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal. MapReduce comes into its own for large datasets, so let's start by looking at one.

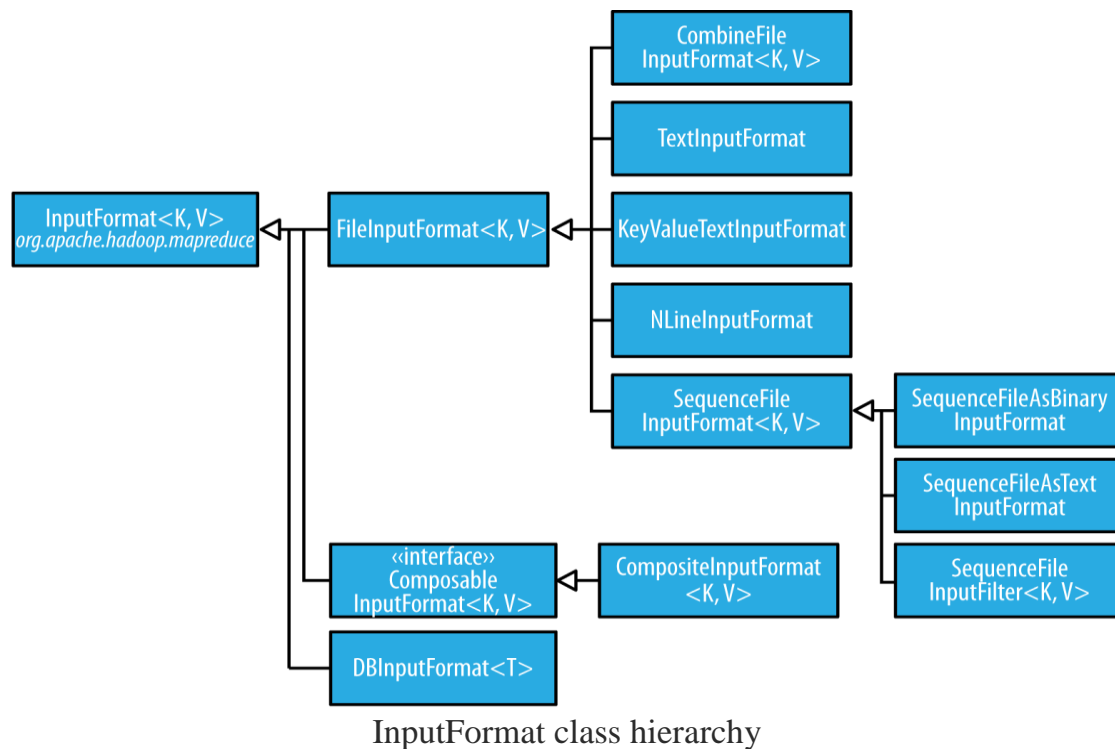
Writing a program in MapReduce follows a certain pattern. You start by writing your map and reduce functions, ideally with unit tests to make sure they do what you expect. Then you write a driver program to run a job, which can run from your IDE using a small subset of the data to check that it is working. If it fails, you can use your IDE's debugger to find the source of the problem. With this information, you can expand your unit tests to cover this case and improve your mapper or reducer as appropriate to handle such input correctly. When the program runs as expected against the small dataset, you are ready to unleash it on a cluster. Running against the full dataset is likely to expose some more issues, which you can fix as before, by expanding your tests and altering your mapper or reducer to handle the new cases. Debugging failing programs in the cluster is a challenge, so we'll look at some common techniques to make it easier. After the program is working, you may wish to do some tuning, first by running through some standard checks for making MapReduce programs faster and then by doing task profiling. Profiling distributed programs is not easy, but Hadoop has hooks to aid in the process. Before we start writing a MapReduce program, however, we need to set up and configure the development environment. And to do that, we need to learn a bit about how Hadoop does configuration.



Map Reduce operation

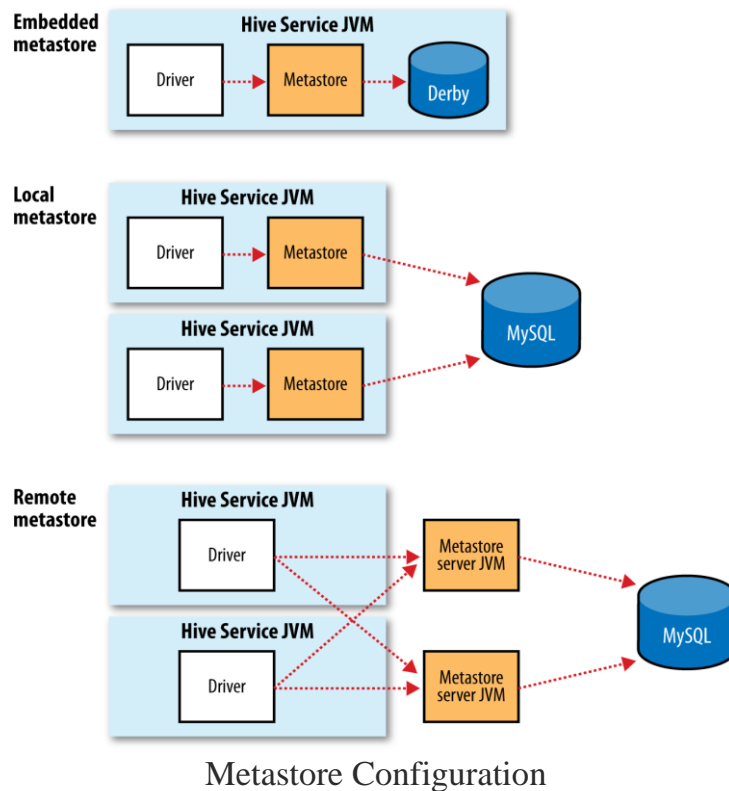
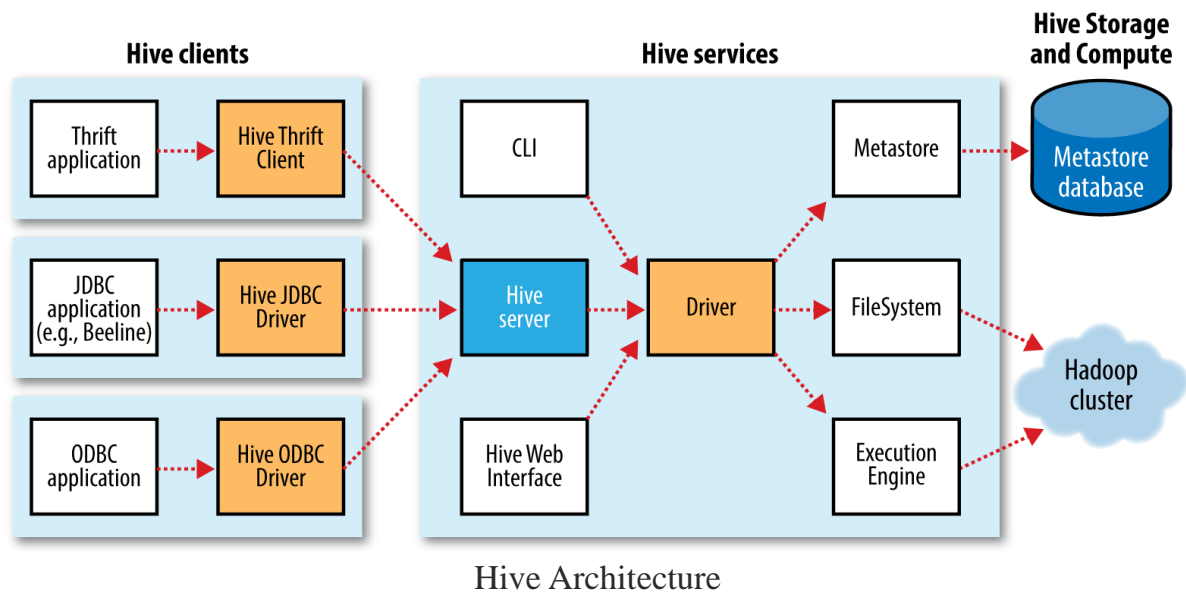


Shuffle and sort in MapReduce



6) Hive:

One of the biggest ingredients in the Information Platform built by Jeff's team at Facebook was Apache Hive, a framework for data warehousing on top of Hadoop. Hive grew from a need to manage and learn from the huge volumes of data that Facebook was producing every day from its burgeoning social network. After trying a few different systems, the team chose Hadoop for storage and processing, since it was cost effective and met the scalability requirements. Hive was created to make it possible for analysts with strong SQL skills (but meager Java programming skills) to run queries on the huge volumes of data that Facebook stored in HDFS. Today, Hive is a successful Apache project used by many organizations as a general-purpose, scalable data processing platform. Of course, SQL isn't ideal for every big data problem—it's not a good fit for building complex machine-learning algorithms, for example—but it's great for many analyses, and it has the huge advantage of being very well known in the industry. What's more, SQL is the lingua franca in business intelligence tools (ODBC is a common bridge, for example), so Hive is well placed to integrate with these products.



7) Pig:

Apache Pig raises the level of abstraction for processing large datasets. MapReduce allows you, as the programmer, to specify a map function followed by a reduce function, but working out how to fit your data processing into this pattern, which often requires multiple MapReduce stages, can be a challenge. With Pig, the data structures are much richer, typically being multi-valued and nested, and the transformations you can apply to the data are much more powerful. They include joins, for example, which are not for the faint of heart in MapReduce. Pig is made up of two pieces:

- The language used to express data flows, called Pig Latin.
- The execution environment to run Pig Latin programs. There are currently two environments: local execution in a single JVM and distributed execution on a Hadoop cluster.

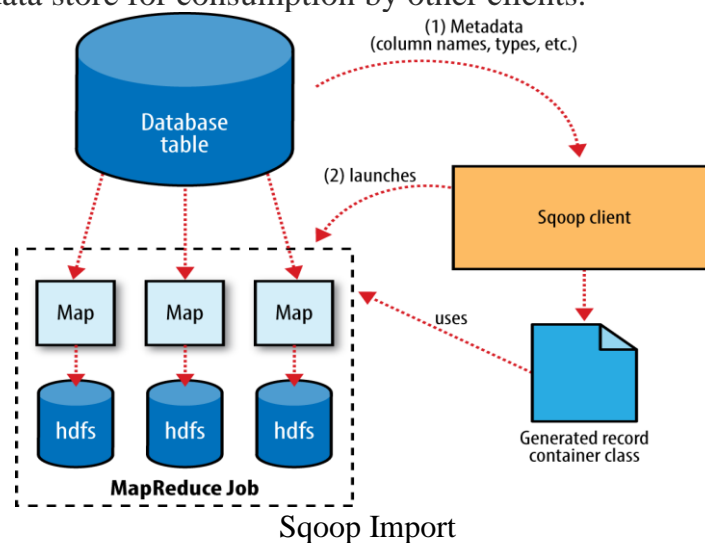
A Pig Latin program is made up of a series of operations, or transformations, that are applied to the input data to produce output. Taken as a whole, the operations describe a data flow, which the Pig execution environment translates into an executable representation and then runs. Under the covers, Pig turns the transformations into a series of MapReduce jobs, but as a programmer you are mostly unaware of this, which allows you to focus on the data rather than the nature of the execution.

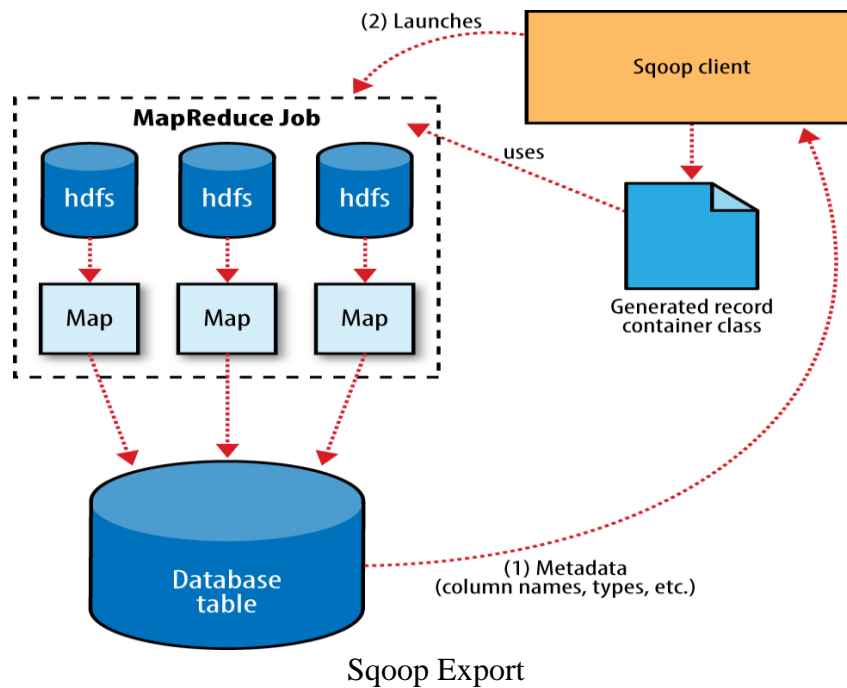
Pig is a scripting language for exploring large datasets. One criticism of MapReduce is that the development cycle is very long. Writing the mappers and reducers, compiling and packaging the code, submitting the job(s), and retrieving the results is a time consuming business, and even with Streaming, which removes the compile and package step, the experience is still involved. Pig's sweet spot is its ability to process terabytes of data in response to a half-dozen lines of Pig Latin issued from the console.

8) Sqoop:

A great strength of the Hadoop platform is its ability to work with data in several different forms. HDFS can reliably store logs and other data from a plethora of sources, and MapReduce programs can parse diverse ad hoc data formats, extracting relevant information and combining multiple datasets into powerful results.

But to interact with data in storage repositories outside of HDFS, MapReduce programs need to use external APIs. Often, valuable data in an organization is stored in structured data stores such as relational database management systems (RDBMSs). Apache Sqoop is an open source tool that allows users to extract data from a structured data store into Hadoop for further processing. This processing can be done with MapReduce programs or other higher-level tools such as Hive. (It's even possible to use Sqoop to move data from a database into HBase.) When the final results of an analytic pipeline are available, Sqoop can export these results back to the data store for consumption by other clients.





9) PROJECT : H1-B Case Study

Project overview: The H1B is an employment-based, non-immigrant visa category for temporary foreign workers in the United States. For a foreign national to apply for H1B visa, an US employer must offer a job and petition for H1B visa with the US immigration department. This is the most common visa status applied for and held by international students once they complete college/ higher education (Masters, Ph.D.) and work in a full-time position.

We will be performing analysis on the H1B visa applicants between the years 2011-2016. After analysing the data, we can derive the following facts.

Tasks:

- 1 a) Is the number of petitions with Data Engineer job title increasing over time?
b) Find top 5 job titles who are having highest average growth in applications. [ALL]
- 2 a) Which part of the US has the most Data Engineer jobs for each year?
b) find top 5 locations in the US who have certified visa for each year. [certified]
- 3) Which industry(SOC_NAME) has the most number of Data Scientist positions? [certified]
- 4) Which top 5 employers file the most petitions each year? - Case Status - ALL

- 5) Find the most popular top 10 job positions for H1B visa applications for each year?
 - a) for all the applications
 - b) for only certified applications.
- 6) Find the percentage and the count of each case status on total applications for each year. Create a line graph depicting the pattern of All the cases over the period.
- 7) Create a bar graph to depict the number of applications for each year [All]
- 8) Find the average Prevailing Wage for each Job for each Year (take part time and full time separate). Arrange the output in descending order - [Certified and Certified Withdrawn.]
- 9) Which are the employers along with the number of petitions who have the success rate more than 70% in petitions. (total petitions filed 1000 OR more than 1000) ?
- 10) Which are the job positions along with the number of petitions which have the success rate more than 70% in petitions (total petitions filed 1000 OR more than 1000)?
- 11) Export result for question no 10 to MySql database.

Dataset Sample and Explanation:

1st Column: Sl.no

Data Eg: 1

Description: The serial number column offers unique numbers to each records.

2ndColumn: Case Status

Data Eg: CERTIFIED, CERTIFIED-WITHDRAWN, DENIED, WITHDRAWN.

Description: Status associated with the last significant event or decision. Valid values include “Certified,” “Certified-Withdrawn,” Denied,” and “Withdrawn”.

Certified: Employer filed the LCA, which was approved by DOL

Certified Withdrawn: LCA was approved but later withdrawn by employer

Withdrawn: LCA was withdrawn by employer before approval

Denied: LCA was denied by DOL

3rd Column: Employer Name

Data Eg: University of Michigan

Description: Name of employer submitting labour condition application.

4thColumn: Soc_name

Data Eg: BIOCHEMISTS AND BIOPHYSICISTS

Description: the Occupational name associated with the SOC_CODE. SOC_CODE is the occupational code associated with the job being requested for temporary labour condition, as classified by the Standard Occupational Classification (SOC) System.

5th Column: Job Title

Data Eg: Post Doctoral Research Fellow

Description: Title of the job

6th Column: Full_Time_Position

Data Eg: Y,N

Description: Y = Full Time Position; N = Part Time Position

7th Column: Prevailing Wage

Data Eg:36067

Description: Prevailing Wage for the job being requested for temporary labour condition. The wage is listed at annual scale in USD. The prevailing wage for a job position is defined as the average wage paid to similarly employed workers in the requested occupation in intended employment. The prevailing wage is based on the employer's minimum requirements for the position.

8th Column: Year

Data Eg:2011

Description: Year in which the H1B visa petition was filed

9th Column: Worksite

Data Eg: Ann Arbor, Michigan

Description: City and State information of the foreign worker's intended area of employment

10th Column: Longitude

Data Eg:-83.7430378

Description: longitude of the Worksite

11th Column: Latitude

Data Eg:42.2808256

Description: latitude of the Worksite

Cleansing the Dataset:

Introduction:

The raw dataset provided had values enclosed within “” and columns separated by “,” and also the records may contain invalid or erroneous records like NA. Hence it is necessary to clean the Dataset and make it processable before carrying out the analysis.

Step 1:

The records from the raw dataset are loaded into a table h1b where the SERDE functions are used to separate the values with “”. Then the data is loaded into the table from local file system.

```
CREATE TABLE h1b_applications(s_no int,case_status string,
employer_name string, soc_name string, job_title string,
full_time_position string,prevailing_wage bigint,year string, worksite
string, longitude double, latitude double )
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
"separatorChar" = ",",
"quoteChar" = "\""
) STORED AS TEXTFILE;
```

```
load data local inpath '/home/hduser/h1b.csv' overwrite into table
h1b_applications;
```

Step 2:

Another table h1b_app2 is created with field delimiter “\t” and data is loaded from the h1b applications table but the field separators “,” is replaced by “\t” while loading the data. Then a condition is specified to avoid records with NA in the case status.

```
CREATE TABLE h1b_app2(s_no int,case_status string, employer_name
string, soc_name string, job_title string, full_time_position
string,prevailing_wage bigint,year string, worksite string, longitude
double, latitude double )
row format delimited
fields terminated by '\t'
STORED AS TEXTFILE;
```

```
INSERT OVERWRITE TABLE h1b_app2 SELECT regexp_replace(s_no, "\t", ""),
regexp_replace(case_status, "\t", ""), regexp_replace(employer_name,
"\t", ""), regexp_replace(soc_name, "\t", ""),
regexp_replace(job_title, "\t", ""),
regexp_replace(full_time_position, "\t", ""), prevailing_wage,
regexp_replace(year, "\t", ""), regexp_replace(worksite, "\t", ""),
regexp_replace(longitude, "\t", ""), regexp_replace(latitude, "\t",
"") FROM h1b_applications where case_status != "NA";
```

Step 3:

Another table h1b_final is created and records from the previous table are loaded but the case status column is filtered only for CERTIFIED, CERTIFIED-WITHDRAWN, DENIED AND WITHDRAWN. Then this table offers the final cleansed dataset with which all the analysis is carried out.

```
CREATE TABLE h1b_final(s_no int, case_status string, employer_name
string, soc_name string, job_title string, full_time_position
string, prevailing_wage bigint, year string, worksite string, longitude
double, latitude double )
row format delimited
fields terminated by '\t'
STORED AS TEXTFILE;
```

```
INSERT OVERWRITE TABLE h1b_final SELECT s_no,
case when trim(case_status) = "PENDING QUALITY AND COMPLIANCE REVIEW -
UNASSIGNED" then "DENIED"
when trim(case_status) = "REJECTED" then "DENIED"
when trim(case_status) = "INVALIDATED" then "DENIED"
else case_status end,
employer_name, soc_name, job_title, full_time_position,
case when prevailing_wage is null then 100000
else prevailing_wage end,
year, worksite, longitude, latitude
FROM h1b_app2;
```

Solution and Explanation:

1 a) Is the number of petitions with Data Engineer job title increasing over time?

Solution: This problem has been solved using the Map Reduce method using Java programming.

Algorithm:

Step 1: The concept of Mapper and Reducer is used here. The data set has numerous columns but the columns which we need to solve the problem are Year and Sl.no. Hence the Year column is passed as the key and the Sl.no is passed as the count.

Step 2: Now before generating the mapper output the required records can be filtered with the condition that Job Title is equal to Data Engineer, this will help in reducing the number of records that go into the reducer.

Step 3: The mapper output is fed into the partitioner where the records are split based on their year values and sent into respective partitioners.

Step 4: Then in the reducer a simple counting operation is done and the results are displayed. And the results will be sorted based on the key which is the year and hence the year is in increasing order and so is the number of applicants.

Code:

```
import java.io.*;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Partitioner;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;
```

//Mapper code

```
public class bdprj1a {

    public static class MapClass extends Mapper<LongWritable,Text,Text,LongWritable>
    {
        public void map(LongWritable key, Text value, Context context)
        {
            try{
                String[] str = value.toString().split("\t");
                long slno = Long.parseLong(str[7]);
                if(str[4].equals("DATA ENGINEER"))
                {
                    context.write(new Text(str[4]),new LongWritable(slno));
                }
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

// Partitioner code

```
public static class CaderPartitioner extends Partitioner < Text, LongWritable >
{
    @Override
```

```

public int getPartition(Text key, LongWritable value, int numReduceTasks)
{
    String[] str6 = value.toString().split("\t");

    if(str6[0].contains("2011"))
    {
        return 0 % numReduceTasks;
    }
    else if(str6[0].contains("2012"))
    {
        return 1 % numReduceTasks ;
    }
    else if(str6[0].contains("2013"))
    {
        return 2 % numReduceTasks;
    }
    else if(str6[0].contains("2014"))
    {
        return 3 % numReduceTasks;
    }
    else if(str6[0].contains("2015"))
    {
        return 4 % numReduceTasks;
    }
    else
    {
        return 5 % numReduceTasks;
    }
}
}

```

//Reducer Code

```

public          static          class          ReduceClass          extends
Reducer<Text,LongWritable,Text,LongWritable>
{
    private LongWritable result = new LongWritable();

    public void reduce(Text key, Iterable<LongWritable> values,Context context)
throws IOException, InterruptedException {

        long count = 0;

        for ( LongWritable val : values)

```

```

        {
            count++;
        }

        result.set(count);

        context.write(key, result);

    }
}

```

//Driver Code

```

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Volume Count");
    job.setJarByClass(StockVolume.class);
    job.setMapperClass(MapClass.class);
    job.setPartitionerClass(CaderPartitioner.class);
    job.setReducerClass(ReduceClass.class);
    job.setNumReduceTasks(6);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(LongWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

Output:

```

2011  DATA ENGINEER 18
2012  DATA ENGINEER 32
2013  DATA ENGINEER 41
2014  DATA ENGINEER 89
2015  DATA ENGINEER 160
2016  DATA ENGINEER 251

```

1b) Find top 5 job titles who are having highest average growth in applications.[ALL]

Solution: This question has been solved using Pig Latin.

Algorithm:

Step 1: In pig the query is solved by proceeding with bags and loading and processing data in them. In the first step the cleansed data is loaded into a bag.

Step 2: Then the bag is filtered year wise and then stored together in another bag.

Step 3: Then the progressive growth formula is applied.

Step 4: The average progressive growth is found out and the result is ordered in descending order and printed.

Code:

--loading the cleansed data into bag

```
loaddata = LOAD '/user/hive/warehouse/bdprj.db/h1b_final' USING PigStorage('\t') AS
(slno:int,case_status:chararray
employer_name:chararray,soc_name:chararray,job_title:chararray,full_time_position:chararr
ay,prevailing_wage:int,year:chararray,worksite:chararray
,longitude:double,latitude:double);
```

--filtering by each year and storing it in respective bags

```
filtered= filter loaddata by $7 == '2011'; --filtering dataset by year
a= group filtered by $4;
bag11= foreach a generate group,COUNT($1);
```

```
filtered= filter loaddata by $7 == '2012'; --filtering dataset by year
a= group filtered by $4;
bag12= foreach a generate group,COUNT($1);
```

```
filtered= filter loaddata by $7 == '2013'; --filtering dataset by year
a= group filtered by $4;
bag13= foreach a generate group,COUNT($1);
```

```
filtered= filter loaddata by $7 == '2014'; --filtering dataset by year
a= group filtered by $4;
bag14= foreach a generate group,COUNT($1);
```

```
filtered= filter loaddata by $7 == '2015'; --filtering dataset by year
a= group filtered by $4;
bag15= foreach a generate group,COUNT($1);
```

```
filtered= filter loaddata by $7 == '2016'; --filtering dataset by year
a= group filtered by $4;
bag16= foreach a generate group,COUNT($1);
```

--joining all the filtered bags into one

```
joined= join bag11 by $0,bag12 by $0,bag13 by $0,bag14 by $0,bag15 by $0,bag16 by $0;
yearly= foreach joined generate $0,$1,$3,$5,$7,$9,$11;
```

--applying progressive growth formula to the joined bag

```
growth= foreach yearly generate $0,
(float)($2-$1)*100/$1,(float)($3-$2)*100/$2,
(float)($4-$3)*100/$3,(float)($5-$4)*100/$4,
(float)($6-$5)*100/$5;
```

--finding average progressive growth

```
avggrowth= foreach growth generate $0,($1+$2+$3+$4+$5)/5;
```

--ordering the results in descending order

```
orderedavggrowth= order avggrowth by $1 desc;
```

```
topavggrowth = limit orderedavggrowth 5;
```

--printing the results

```
dump topavggrowth;
```

Output:

```
(SENIOR SYSTEMS ANALYST JC60,4255.4644)
(SOFTWARE DEVELOPER 2,3480.5925)
(PROJECT MANAGER 3,3233.3335)
(SYSTEMS ANALYST JC65,2984.8809)
(MODULE LEAD,2917.112)
```

2) a) Which part of the US has the most Data Engineer jobs for each year?

Solution: This query is solved using hive with SQL programming.

Algorithm:

The cleaned dataset is grouped by year and worksite and then the where condition to find only the Data Engineer applicants and year is specified from 2011 to 2016 individually as a condition. The output is printed in descending order of the count total and the top value is printed.

Code:

```
select year,worksite,count(*)as total from h1b_final where job_title="DATA ENGINEER"
and year=$variable and case_status="CERTIFIED" GROUP BY year,worksite order by total
desc limit 1;
```

the year is received as an input from the user and the solution is derived for the respective year.

Output:

```
2011    PLANO, TEXAS    2
```

2) b) find top 5 locations in the US who have certified visa for each year. [certified]

Solution: This query is solved using hive with SQL programming.

Algorithm:

The cleansed data is grouped on the basis of locations and year and the conditions are specified to the required year and the case status to be CERTIFIED and the output is ordered by the total number of applicants in descending order and top five is printed.

Code:

```
select year,count(*)as total ,worksite from h1b_final where year=$variable and
case_status="CERTIFIED" group by year,worksite order by total desc limit 5;
```

the year is received as an input from the user and the solution is derived for the respective year.

Output:

```
2011    23172    NEW YORK, NEW YORK
2011    8184     HOUSTON, TEXAS
2011    5188     CHICAGO, ILLINOIS
2011    4713     SAN JOSE, CALIFORNIA
2011    4711     SAN FRANCISCO, CALIFORNIA
```

**3) Which industry(SOC_NAME) has the most number of Data Scientist positions?
[certified]**

Solution: This query is solved using hive with SQL programming.

Algorithm:

The cleansed data is grouped by soc_name and job title and conditions are applied to filter CERTIFIED case status and DATA SCIENTIST jobs.

Code: `select soc_name,job_title,count(*)as total from h1b_final where case_status="CERTIFIED" and job_title="DATA SCIENTIST" group by soc_name,job_title order by total desc limit 1;`

Output:

STATISTICIANS DATA SCIENTIST369

4) Which top 5 employers file the most petitions each year? - Case Status – ALL

Solution: This query is solved using hive with SQL programming.

Algorithm:

The cleansed data is grouped by year and employer name and then the required year is passed as the condition and the total applicants are counted and the top 5 are printed.

Code:

`select year,employer_name,count(*)as total from h1b_final where year=$variable group by year,employer_name order by total desc limit 5;`

the year is received as an input from the user and the solution is derived for the respective year.

Output:

2011	TATA CONSULTANCY SERVICES LIMITED	5416
2011	MICROSOFT CORPORATION	4253
2011	DELOITTE CONSULTING LLP	3621
2011	WIPRO LIMITED	3028
2011	COGNIZANT TECHNOLOGY SOLUTIONS U.S. CORPORATION	2721

**5) a) Find the most popular top 10 job positions for H1B visa applications for each year?
for all the applications**

Solution: This problem has been solved using the Map Reduce method using Java programming.

Algorithm:

Step 1: The concept of Mapper and Reducer is used here. The data set has numerous columns but the columns which we need to solve the problem are job positions and year. Hence the job position column is passed as the key and the year is passed as the count.

Step 2: The mapper output is fed into the partitioner where the records are split based on their year values and sent into respective partitioners.

Step 3: Then in the reducer a simple counting operation is done, and the results are sent to a Tree map which stores the top 10 values. And the results will be sorted based on the key which is the year and hence the year is in increasing order and so is the number of applicants for top job positions is displayed.

Code:

```
import java.io.IOException;
import java.util.TreeMap;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable.DecreasingComparator;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Partitioner;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

//Mapper code

```
public class bdprj5atree {
    public static class Top5Mapper extends
        Mapper<LongWritable, Text, Text, Text> {
        public void map(LongWritable key, Text value, Context context
        ) throws IOException, InterruptedException {
            try {
                String[] str = value.toString().split("\t");
                String job_pos= str[4];
                String str2 = str[7];
                context.write(new Text(job_pos),new Text(str2));
            }
        }
    }
}
```

```

    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
}

```

//Partitioner Code

```

public static class CaderPartitioner extends Partitioner < Text, Text >
{
    @Override
    public int getPartition(Text key, Text value, int numReduceTasks)
    {
        String[] str6 = value.toString().split(",");

        if(str6[0].contains("2011"))
        {
            return 0 % numReduceTasks;
        }
        else if(str6[0].contains("2012"))
        {
            return 1 % numReduceTasks ;
        }
        else if(str6[0].contains("2013"))
        {
            return 2 % numReduceTasks;
        }
        else if(str6[0].contains("2014"))
        {
            return 3 % numReduceTasks;
        }
        else if(str6[0].contains("2015"))
        {
            return 4 % numReduceTasks;
        }

        else
        {
            return 5 % numReduceTasks;
        }
    }
}

```

//Reducer Code

```

public static class Top5Reducer extends

```

```

Reducer<Text, Text, NullWritable, Text> {
    private TreeMap<Long, Text> repToRecordMap = new TreeMap<Long, Text>();

    public void reduce(Text key, Iterable<Text> values,
        Context context) throws IOException, InterruptedException {
        long sum=0;
        String totcnt= "";
        String year="";
        for (Text val : values) {

            String[] str3=val.toString().split(",");
            LongWritable value = new LongWritable(Integer.parseInt(str3[0]));
            year= value.toString();
            sum=sum+1;
        }
        totcnt= key.toString();
        totcnt= totcnt + ',' + sum+', '+year;
        repToRecordMap.put(new Long(sum), new Text(totcnt));
        if (repToRecordMap.size() > 10) {
            repToRecordMap.remove(repToRecordMap.firstKey());
        }
    }

    protected void cleanup(Context context) throws IOException,
        InterruptedException {
        for (Text t : repToRecordMap.descendingMap().values()) {
            context.write(NullWritable.get(), t);
        }
    }
}

```

//Driver Code

```

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

    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "5a");
    job.setJarByClass(bdprj5atree.class);
    job.setMapperClass(Top5Mapper.class);
    job.setPartitionerClass(CaderPartitioner.class);
    job.setReducerClass(Top5Reducer.class);
    job.setNumReduceTasks(6);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);
    job.setOutputKeyClass(NullWritable.class);
    job.setOutputValueClass(Text.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
}

```

```

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

```

Output:

```

2011  PROGRAMMER ANALYST  31799
2011  SOFTWARE ENGINEER  12763
2011  COMPUTER PROGRAMMER   8998
2011  SYSTEMS ANALYST 8644
2011  BUSINESS ANALYST   3891
2011  COMPUTER SYSTEMS ANALYST  3698
2011  ASSISTANT PROFESSOR   3467
2011  PHYSICAL THERAPIST  3377
2011  SENIOR SOFTWARE ENGINEER   2935
2011  SENIOR CONSULTANT  2798

```

```

2012  PROGRAMMER ANALYST  33066
2012  SOFTWARE ENGINEER  14437
2012  COMPUTER PROGRAMMER   9629
2012  SYSTEMS ANALYST 9296
2012  BUSINESS ANALYST   4752
2012  COMPUTER SYSTEMS ANALYST  4706
2012  SOFTWARE DEVELOPER   3895
2012  PHYSICAL THERAPIST  3871
2012  ASSISTANT PROFESSOR   3801
2012  SENIOR CONSULTANT  3737

```

```

2013  PROGRAMMER ANALYST  33880
2013  SOFTWARE ENGINEER  15680
2013  COMPUTER PROGRAMMER  11271
2013  SYSTEMS ANALYST 8714
2013  TECHNOLOGY LEAD - US   7853
2013  TECHNOLOGY ANALYST - US  7683
2013  BUSINESS ANALYST   5716
2013  COMPUTER SYSTEMS ANALYST  5043
2013  SOFTWARE DEVELOPER   5026
2013  SENIOR CONSULTANT  4326

```

```

2014  PROGRAMMER ANALYST  43114
2014  SOFTWARE ENGINEER  20500
2014  COMPUTER PROGRAMMER  14950
2014  SYSTEMS ANALYST 10194
2014  SOFTWARE DEVELOPER   7337
2014  BUSINESS ANALYST   7302
2014  COMPUTER SYSTEMS ANALYST  6821
2014  TECHNOLOGY LEAD - US   5057

```

2014	TECHNOLOGY ANALYST - US	4913
2014	SENIOR CONSULTANT	4898
2015	PROGRAMMER ANALYST	53436
2015	SOFTWARE ENGINEER	27259
2015	COMPUTER PROGRAMMER	14054
2015	SYSTEMS ANALYST	12803
2015	SOFTWARE DEVELOPER	10441
2015	BUSINESS ANALYST	8853
2015	TECHNOLOGY LEAD - US	8242
2015	COMPUTER SYSTEMS ANALYST	7918
2015	TECHNOLOGY ANALYST - US	7014
2015	SENIOR SOFTWARE ENGINEER	6013
2016	PROGRAMMER ANALYST	53743
2016	SOFTWARE ENGINEER	30668
2016	SOFTWARE DEVELOPER	14041
2016	SYSTEMS ANALYST	12314
2016	COMPUTER PROGRAMMER	11668
2016	BUSINESS ANALYST	9167
2016	COMPUTER SYSTEMS ANALYST	6900
2016	SENIOR SOFTWARE ENGINEER	6439
2016	DEVELOPER	6084
2016	TECHNOLOGY LEAD - US	5410

5) b) Find the most popular top 10 job positions for H1B visa applications for each year? For only certified applications.

Solution: This problem has been solved using the Map Reduce method using Java programming.

Algorithm:

Step 1: The concept of Mapper and Reducer is used here. The data set has numerous columns but the columns which we need to solve the problem are job positions and year. Hence the job position column is passed as the key and the year is passed as the count.

Step 2: The mapper output is fed into the partitioner by filtering it with a condition that case status must be certified and only those records are fed into the partitioner where the records are split based on their year values and sent into respective partitioners.

Step 3: Then in the reducer a simple counting operation is done, and the results are sent to a Tree map which stores the top 10 values. And the results will be sorted based on the key which is the year and hence the year is in increasing order and so is the number of applicants for top job positions is displayed.

Code:

```
import java.io.IOException;
import java.util.TreeMap;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable.DecreasingComparator;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Partitioner;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

//Mapper Code

```
public class bdprj5btree {
    public static class Top5Mapper extends
    Mapper<LongWritable, Text, Text, Text> {
        public void map(LongWritable key, Text value, Context context
        ) throws IOException, InterruptedException {
            try {
                String[] str = value.toString().split("\t");
                String job_pos= str[4];
                String str2 = str[7];
                if(str[1].equals("CERTIFIED"))
                {
                    context.write(new Text(job_pos),new Text(str2));
                }
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

// Partitioner Code

```
public static class CaderPartitioner extends Partitioner < Text, Text >
{
    @Override
```



```

public int getPartition(Text key, Text value, int numReduceTasks)
{
    String[] str6 = value.toString().split(",");

    if(str6[0].contains("2011"))
    {
        return 0 % numReduceTasks;
    }
    else if(str6[0].contains("2012"))
    {
        return 1 % numReduceTasks ;
    }
    else if(str6[0].contains("2013"))
    {
        return 2 % numReduceTasks;
    }
    else if(str6[0].contains("2014"))
    {
        return 3 % numReduceTasks;
    }
    else if(str6[0].contains("2015"))
    {
        return 4 % numReduceTasks;
    }

    else
    {
        return 5 % numReduceTasks;
    }
}
}

```

// Reducer Code

```

public static class Top5Reducer extends
Reducer<Text, Text, NullWritable, Text> {
    private TreeMap<Long, Text> repToRecordMap = new TreeMap<Long, Text>();

    public void reduce(Text key, Iterable<Text> values,
        Context context) throws IOException, InterruptedException {
        long sum=0;
        String totcnt= "";
        String myyear="";
        for (Text val : values) {

            String[] str3=val.toString().split(",");

```

```

        LongWritable value = new LongWritable(Integer.parseInt(str3[0]));
        myyear= value.toString();
        sum=sum+1;
    }
    totcnt= key.toString();
    totcnt= totcnt + ',' + sum+','+myyear;
    repToRecordMap.put(new Long(sum), new Text(totcnt));
    if (repToRecordMap.size() > 10) {
        repToRecordMap.remove(repToRecordMap.firstKey());
    }
}

protected void cleanup(Context context) throws IOException,
InterruptedException {
    for (Text t : repToRecordMap.descendingMap().values()) {
        context.write(NullWritable.get(), t);
    }
}
}

```

//Driver Code

```

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

    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "5b");
    job.setJarByClass(bdprj5atree.class);
    job.setMapperClass(Top5Mapper.class);
    job.setPartitionerClass(CaderPartitioner.class);
    job.setReducerClass(Top5Reducer.class);
    job.setNumReduceTasks(6);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);
    job.setOutputKeyClass(NullWritable.class);
    job.setOutputValueClass(Text.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

6) Find the percentage and the count of each case status on total applications for each year. Create a line graph depicting the pattern of All the cases over the period.

Solution: This question has been solved using Pig Latin.

Algorithm:

Step 1: In pig the query is solved by proceeding with bags and loading and processing data in them. In the first step the cleansed data is loaded into a bag.

Step 2: Then the bag is grouped year wise and then stored together in another bag. And count operation is performed in another.

Step 3: Then another bag is grouped year and case status wise and then stored together in another bag. And count operation is performed in another.

Step 4: Then the above two bags are joined, and the percentage formula is applied and the required results are generated.

Code:

--loading the cleansed data into a bag

```
loaddata = LOAD '/user/hive/warehouse/bdprj.db/h1b_final' USING PigStorage('\t') AS
(sln:int,case_status:chararray
employer_name:chararray,soc_name:chararray,job_title:chararray,full_time_position:chararray
,prevailing_wage:int
,year:chararray,
worksite:chararray
,longitude:double,latitude:double);
```

--grouping the loaded bag by year

```
group_by_year = group loaddata by $7;
```

--finding total number of applications for each year with the grouped by year bag

```
year_total = foreach group_by_year generate group, COUNT(loaddata.$1);
```

--grouping the loaded bag by both year and case status

```
group_by_year_status = group loaddata by ($7,$1);
```

--finding total number of applications for each case status for each year

```
year_status_total = foreach group_by_year_status generate group,$0,COUNT($1);
```

--Joining the two totals generated before into a single bag to perform the math operation

```
joined = join year_status_total by $1, year_total by $0;
```

--now we have both the totals required for the calculation and the percentage is calculated.

```
percentage = foreach joined generate FLATTEN($0),ROUND_TO((float)($2*100)/$4,2),$2;
```

--then the output is printed.
dump percentage;

Output:

2011,DENIED,8.12,29130
2011,CERTIFIED,85.83,307936
2011,WITHDRAWN,2.82,10105
2011,CERTIFIED-WITHDRAWN,3.23,11596
2012,DENIED,5.08,21096
2012,CERTIFIED,84.86,352668
2012,WITHDRAWN,2.58,10725
2012,CERTIFIED-WITHDRAWN,7.49,31118
2013,CERTIFIED-WITHDRAWN,8.01,35432
2013,WITHDRAWN,2.62,11590
2013,CERTIFIED,86.62,382951
2013,DENIED,2.75,12141
2014,CERTIFIED-WITHDRAWN,7.0,36350
2014,WITHDRAWN,3.09,16034
2014,CERTIFIED,87.62,455144
2014,DENIED,2.29,11899
2015,DENIED,1.77,10923
2015,CERTIFIED,88.45,547278
2015,WITHDRAWN,3.14,19455
2015,CERTIFIED-WITHDRAWN,6.64,41071
2016,CERTIFIED,87.94,569646
2016,WITHDRAWN,3.38,21890
2016,CERTIFIED-WITHDRAWN,7.27,47092
2016,DENIED,1.42,9175

7) Create a bar graph to depict the number of applications for each year [All]

Solution: This problem has been solved using the Map Reduce method using Java programming.

Algorithm:

Step 1: The concept of Mapper and Reducer is used here. The data set has numerous columns but the columns which we need to solve the problem are Year and Sl.no. Hence the Year column is passed as the key and the Sl.no is passed as the count.

Step 2: Then in the reducer a simple counting operation is done and the results are displayed. And the results will be sorted based on the key which is the year and hence the year is in increasing order and so is the number of applicants.

Code:

```
import java.io.*;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;
```

//Mapper Code

```
public class bdprj7 {

    public static class MapClass extends Mapper<LongWritable,Text,Text,LongWritable>
    {
        public void map(LongWritable key, Text value, Context context)
        {
            try{
                String[] str = value.toString().split("\t");
                long slno = Long.parseLong(str[0]);

                context.write(new Text(str[7]),new LongWritable(slno));
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

//Reducer Code

```
public          static          class          ReduceClass          extends
Reducer<Text,LongWritable,Text,LongWritable>
{
    private LongWritable result = new LongWritable();

    public void reduce(Text key, Iterable<LongWritable> values,Context context)
throws IOException, InterruptedException {
```

```

        long count = 0;

        for ( LongWritable val : values)

        {
            count++;
        }

        result.set(count);

        context.write(key, result);

    }
}

```

// Driver Code

```

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

    Job job = Job.getInstance(conf, "Application count");
    job.setJarByClass(StockVolume.class);
    job.setMapperClass(MapClass.class);
    job.setReducerClass(ReduceClass.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(LongWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

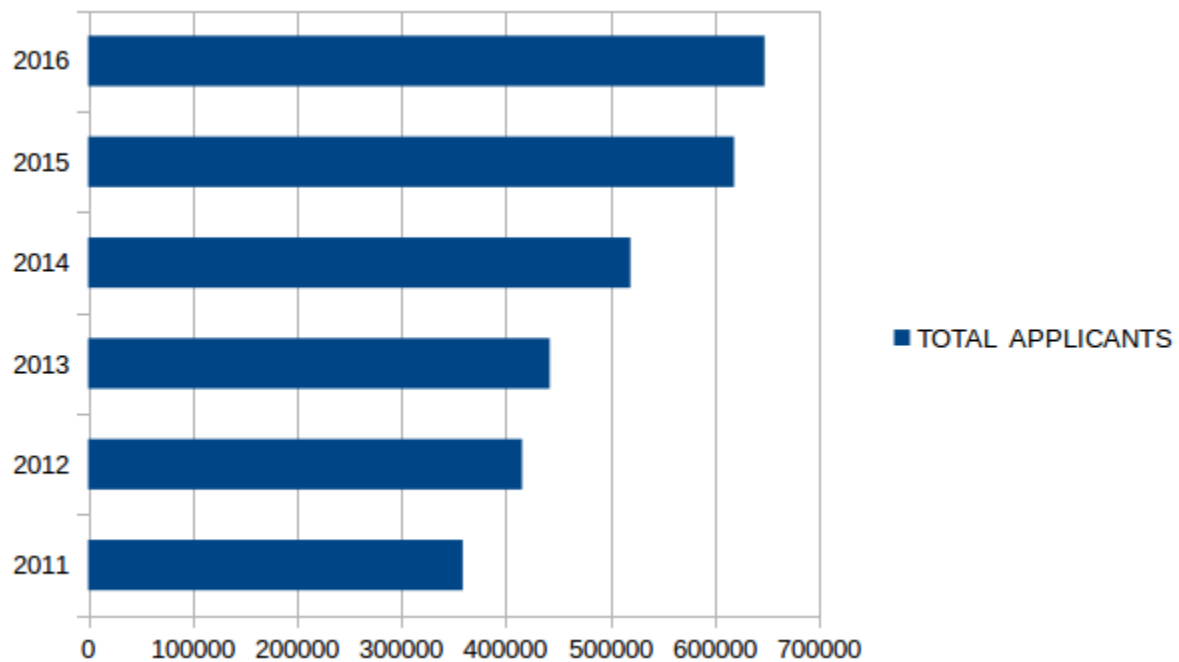
```

Output:

```

2011    358767
2012    415607
2013    442114
2014    519427
2015    618727
2016    647803

```



8) Find the average Prevailing Wage for each Job for each Year (take part time and full time separate). Arrange the output in descending order - [Certified and Certified Withdrawn.]

Solution: This query is solved using hive with SQL programming.

Algorithm:

The cleaned dataset is grouped by year, job title and full time position and then the where condition to find only the CERTIFIED AND CERTIFIED-WITHDRAWN applicants and year is specified from 2011 to 2016 individually as a condition and so is the full time position .The output is printed in descending order of the count total and the top value is printed.

Code:

```
select year,job_title,full_time_position,avg(prevailing_wage)as average from h1b_final
where year=$variable and full_time_position="$variable" and (case_status="CERTIFIED" or
case_status="CERTIFIED-WITHDRAWN") group by year,job_title,full_time_position order
by average desc;
```

Year and Full-time position are received from users in runtime.

Output:

```
2011  LEGISLATIVE SECRETARY Y  15110.0
2011  PURE TALENT TRAINING STYLIST  Y  15080.0
2011  LECTURER IN PHYSICS Y  15080.0
2011  SPECIAL CLASS ASSISTANT  Y  15070.0
```

Time taken: 7.662 seconds, Fetched: 59324 row(s)

9) Which are the employers along with the number of petitions who have the success rate more than 70% in petitions. (total petitions filed 1000 OR more than 1000)?

Solution: This question has been solved using Pig Latin.

Algorithm:

Step 1: In pig the query is solved by proceeding with bags and loading and processing data in them. In the first step the cleansed data is loaded into a bag.

Step 2: Then the bag is grouped based on employer name and then stored together in another bag. And count operation is performed in another.

Step 3: Then another bag is grouped based on employer name but with filtered data of certified and certified withdrawn case statuses and case status wise stored in another bag. And count operation is performed in another.

Step 4: Then the above two bags are joined, and the success rate formula is applied and the required results are generated.

Code:

--Loading the cleansed data into a bag

```
loaddata = LOAD '/home/hduser/Desktop/h1b' USING PigStorage('\t') AS
(slno:int,case_status:chararray
,
employer_name:chararray,soc_name:chararray,job_title:chararray,full_time_position:chararr
ay
,prevailing_wage:int
,year:chararray,
worksite:chararray
,longitute:double,latitude:double);
```

--grouping the data bag by the employer name

```
grouping_all = group loaddata by $2;
```

--counting the number of application for each employer

```
total_petitions = foreach grouping_all generate group, COUNT(loaddata.$1);
```

--filtering by the case status CERTIFIED and CERTIFIED-WITHDRAWN

```
filter_by_status = filter loaddata by $1 == 'CERTIFIED' OR $1 == 'CERTIFIED-
WITHDRAWN';
```

--grouping the filtered bag by employer name


```

grouping_all = group filter_by_status by employer_name;

--counting the number of applications for each employer

total_certified_petition = foreach grouping_all generate group,COUNT(filter_by_status);

--joining the totals from previous two bags

joined = join total_certified_petition by $0, total_petitions by $0;

joined = foreach joined generate $0,$1,$3;

--applying the formula for success rate

op1 = foreach joined generate $0,(float)$1*100/($2),$2;

--filtering with the required conditions

op2 = filter op1 by $1>70 and $2>1000;

op3 = order op2 by $1 Desc;

--printing the output

dump op3;

```

Output:

```

(HTC GLOBAL SERVICES INC.,100.0,2632)
(HTC GLOBAL SERVICES, INC.,100.0,1164)
(INFOSYS LIMITED,99.54055,130592)
(DIASPARK, INC.,99.5067,1419)
(ACCENTURE LLP,99.39307,33447)
(TECH MAHINDRA (AMERICAS),INC.,99.338425,10732)
(TATA CONSULTANCY SERVICES LIMITED,99.337204,64726)
(YASH TECHNOLOGIES, INC.,99.27733,2214)
(YASH & LUJAN CONSULTING, INC.,99.27113,1372)
(HCL AMERICA, INC.,99.26801,22678)
(RELIABLE SOFTWARE RESOURCES, INC.,99.14658,1992)
(NTT DATA, INC.,99.13251,4611)
(ERP ANALYSTS, INC.,99.10364,1785)
(PATNI AMERICAS INC.,99.07907,3149)
(MINDTREE LIMITED,99.06565,4067)
(KFORCE INC.,99.06015,1596)
(TECH MAHINDRA ( AMERICAS), INC,99.05983,1170)
(GRANDISON MANAGEMENT, INC.,98.9899,1386)

```

10) Which are the job positions along with the number of petitions which have the success rate more than 70% in petitions (total petitions filed 1000 OR more than 1000)?

Solution: This question has been solved using Pig Latin.

Algorithm:

Step 1: In pig the query is solved by proceeding with bags and loading and processing data in them. In the first step the cleansed data is loaded into a bag.

Step 2: Then the bag is grouped based on job positions and then stored together in another bag. And count operation is performed in another.

Step 3: Then another bag is grouped based on employer name but with filtered data of certified and certified withdrawn case statuses and case status wise stored in another bag. And count operation is performed in another.

Step 4: Then the above two bags are joined, and the success rate formula is applied and the required results are generated and stored in a file in local file system.

--Loading the cleansed data into a bag

```
loaddata = LOAD '/user/hive/warehouse/bdprj.db/h1b_final' USING PigStorage('\t') AS
(slno:int,case_status:chararray
,
employer_name:chararray,soc_name:chararray,job_title:chararray,full_time_position:chararray
,prevailing_wage:int
,year:chararray,
worksite:chararray
,longitute:double,latitute:double);
```

--grouping the data bag by the job positions

```
grouping_all = group loaddata by $4;
```

--counting the number of application for each job

```
total_petitions = foreach grouping_all generate group, COUNT(loaddata.$1);
```

--filtering by the case status CERTIFIED and CERTIFIED-WITHDRAWN

```
filter_by_status = filter loaddata by $1 == 'CERTIFIED' OR $1 == 'CERTIFIED-
WITHDRAWN';
```

--grouping the filtered bag by job positions

```
grouping_all = group filter_by_status by $4;
```

--counting the number of applications for each job

```
total_certified_petition = foreach grouping_all generate group,COUNT(filter_by_status);
```

```

--joining the totals from previous two bags

joined = join total_certified_petition by $0, total_petitions by $0;

joined = foreach joined generate $0,$1,$3;

--applying the formula for success rate

op1 = foreach joined generate $0,(float)$1*100/($2),$2;

--filtering with the required conditions

op2 = filter op1 by $1>70 and $2>1000;

op3 = order op2 by $1 Desc;

--printing the output

store op3 into '/home/hduser/Desktop/pig10solution1' using PigStorage('\t');

dump op3;

```

Output:

```

(PRODUCTION SUPPORT LEAD - US,100.0,1301)
(COMPUTER PROGRAMMER / CONFIGURER 2,100.0,1276)
(ASSOCIATE CONSULTANT - US,99.93171,4393)
(SYSTEMS ENGINEER - US,99.90036,10036)
(TEST ENGINEER - US,99.86351,2198)
(PRODUCTION SUPPORT ANALYST - US,99.86217,1451)
(TEST ANALYST - US,99.818474,4958)
(CONSULTANT - US,99.81147,7426)
(TECHNOLOGY LEAD - US,99.80247,28350)
(TECHNICAL TEST LEAD - US,99.79531,5374)
(SENIOR TECHNOLOGY ARCHITECT - US,99.788284,1417)
(TECHNOLOGY ARCHITECT - US,99.766304,4707)
(TECHNOLOGY ANALYST - US,99.76204,26055)
(SENIOR PROJECT MANAGER - US,99.74766,2774)
(DEVELOPER USER INTERFACE,99.71412,5247)
(COMPUTER SYSTEMS ANALYST 2,99.70231,4031)
(SYSTEMS ANALYST - II,99.70127,1339)
(PROJECT MANAGER - III,99.69715,1651)
(PROJECT MANAGER - US,99.68777,7046)
(PROGRAMMER ANALYST - II,99.66555,3588)
(LEAD CONSULTANT - US,99.64726,3402)
(COMPUTER SYSTEMS ANALYST 3,99.58525,2170)
(COMPUTER PROGRAMMER/CONFIGURER 2,99.56903,6729)

```

(PROGRAMMER ANALYST - I,99.51117,1432)
(SYSTEMS ANALYST - III,99.50298,1006)
(PRINCIPAL CONSULTANT - US,99.48225,1352)
(COMPUTER SPECIALIST/TESTING AND QUALITY ANALYST 2,99.42471,3998)
(COMPUTER PROGRAMMER/CONFIGURER 3,99.38865,1145)
(COMPUTER SPECIALIST/SYSTEM SUPPORT AND DEVELOPMENT,99.32786,1339)
(COMPUTER SPECIALIST/SYSTEM SUPPORT AND DEVELOPMENT ADMIN
2,99.26267,1085)
(DATA WAREHOUSE SPECIALIST,99.20294,1631)
(SPECIALIST MASTER,99.19571,1119)
(ASSURANCE STAFF,99.05741,2334)
(COMPUTER SYSTEMS ENGINEER/ARCHITECT,98.79052,2067)
(SOFTWARE QUALITY ASSURANCE ENGINEER AND TESTER,98.66071,1568)
(ADVISORY SENIOR,98.652145,5416)
(AUDIT SENIOR,98.59813,1070)
(TEST CONSULTANT,98.55571,1454)
(SOFTWARE ENGINEER AND TESTER,98.51974,1216)
(ARCHITECT LEVEL 2,98.51314,2892)
(PROGRAMMER/DEVELOPER,98.46154,1560)
(TEST ENGINEER LEVEL 2,98.44013,2372)
(MODULE LEAD,98.33782,2226)
(ADVISORY MANAGER,98.310295,3255)
(AUDIT ASSISTANT,98.25726,1205)
(LEAD ENGINEER,98.23429,11157)
(COMPUTER SPECIALIST,98.206894,2175)
(SPECIALIST SENIOR,98.20318,1447)
(CONSULTANT LEVEL 3,98.12126,1171)
(ADVISORY STAFF,98.01077,2413)

11) Export result for question no 10 to MySql database.

Solution: This query is solved using Sqoop Export command

Code:

```
hadoop fs -rm -r -f /bdprj
```

```
hadoop fs -mkdir -p /bdprj
```

```
hadoop fs -put /home/hduser/Desktop/pig10sol/ /bdprj/
```

```
mysql -u root -p'1' -e 'drop database bdprj;create database if not exists bdprj;use bdprj;create  
table q11(job_title varchar(100),success_rate float,total_petitions int);';
```

```
sqoop export --connect jdbc:mysql://localhost/bdprj --username root --password '1' --table  
10sol --update-mode allowinsert --export-dir /bdprj/p* --input-fields-terminated-by '\t' ;
```

```
mysql -u root -p'1' -e 'select * from bdprj.q11';
```

Observation and Conclusion:

The H1b dataset is used and many predictions and observations are made. The trend in the increase in number of applicants and the job positions which keep on increasing are some examples. These observations help in better understanding of the process and it also helps in future planning and precautionary measures to be taken.

The dataset can further be used along with machine learning algorithms to get exact prediction about future trends. The jobs, states and company's individual performances can be predicted, and we can expect the changes that had been predicted to happen in the future.