M.Tech Dissertation Report

Big Data and Streaming Data Analysis

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Sushmita Bhattacharya
Roll No: 133050022

under the guidance of

Prof. Nandlal L. Sarda

Figure 1: IITB Logo

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

July 2, 2015

Dissertation Approval

This dissertation entitled **"Big Data and Streaming Analysis"**, submitted by **Sushmita Bhattacharya (Roll No: 133050022)** is approved for the degree of **Master of Technology** in **Computer Science and Engineering** from **Indian Institute of Technology Bombay**.

Prof. N.L. Sarda
Dept. of CSE, IIT Bombay
Supervisor

Prof. S.Sudarshan
Dept. of CSE, IIT Bombay
Internal Examiner

Mr Manoj Kunkalienkar
Mumbai
External Examiner

Chairperson

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: ____________ ________________________

Place: IIT Bombay, Mumbai Sushmita Bhattacharya

Roll no: 133050022

Acknowledgements

I would like to thank my guide,Prof. N. L. Sarda for the consistent directions he has fed into my work. Also course Seminar [Spring 2014] offered by my guide helped me a lot to know more about basic concepts, problems and solutions to those problems.

Abstract

The recent bloom of internet and internet of things has created the huge amount of data. Companies, organizations are dealing with huge amount of data everyday. Interesting patterns and anomaly can be detected from the data. Unfortunately the traditional systems are incompetent for analysing such data because of the amount and complexity. Hadoop like platform is very popular for batch processing on big data.

Stream processing and analysis is the next generation big-data analytic. Once Hadoop like system performs processing/analysis on data, experts can manually identify the exact pattern on the data. This results in huge success on the organizations' planning and marketing strategies. But data processing/analysis requirements are changing day by day. Data coming from various streams are so important and vulnerable, that they need immediate analysis and action based on them in real time. The right solution to the problem can save many organizations from loosing millions of dollars business. There are several algorithms and open source technologies available for solving this problem. This is a good time for using these algorithms with the available tools and techniques to propose a real-time, fault tolerant, distributed solution.

Contents

List of Figures

Chapter 1

Introduction

Data is new world's newest resource. It is estimated that everyday we create 2.5 million of data from variety of sources. Starting from financial sector from social network post, data is everywhere. Big data is symbolized using 4 V's e.g, Volume, Velocity, Variety, Versatility. Expert prediction says that we would be generating 40 Zettabytes of data by 2020. Big data analytic s includes statistical analysis and forecasting, causal analysis, predictive analysis, text mining on structured or unstructured data. These kind of problem can be solved by Hadoop, Mapreduce, hugely parallel database, sometimes in-memory database, distributed file system, cloud etc.

If such kind of problem addressed using centralized approach, very costly servers would be required. But Hadoop in big data world simplifies the need, by using cluster of commodity machines with multi-core processor, lower storage cost, high speed LAN. The technology says that the data does not come close to computation but computation comes near data.

Such solution in conjunction with NoSql database the aspect of Velocity and Variety can be addressed. These database also provides reliable solution and scalable performance at low cost.

Data stream can come from various sources at different rate. To answer queries on the data stream, some in-memory or fast data store can be used. It is not feasible to answer queries on the data stream from an archival data store which can be used to store information about the stream for future off-line analysis, but not for real time analysis.

There are two types of queries on streaming data.

1. **Standing queries** are stored in a stream processor. The standing queries are applied to the data as soon as new data element of stream arrives. The queries can be answered with a window of the past data or a summary of past data.

2. **Ad-hoc queries** can be answered using a summary over the past streaming data if the types of ad-hoc queries are known. Otherwise such kind of queries cannot be answered using all the past data (as they are not stored). To facilitate unknown ad-hoc queries a window of latest n tuples or all tuples arrived within t time can be stored (based on design). These set of tuples can be treated as a relation and any kind of ad-hoc queries can be answered approximately. The stream processing engine should take care of deleting old tuples as soon as new ones arrive and window gets filled up. For storing these amount of data some working storage is required, but its size is much less than the storage which is required to store all the data in the stream.

If there are enough main memory storage many problems regarding the stream processing

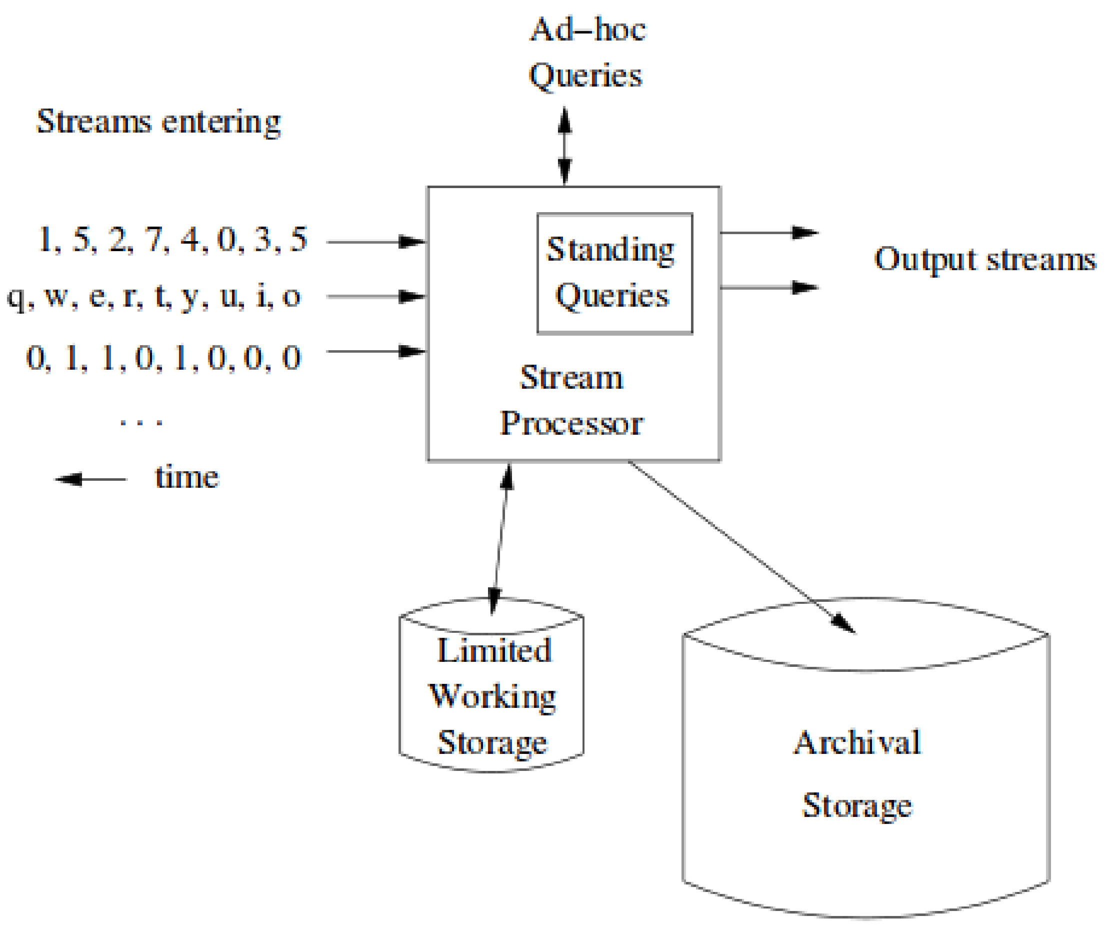


Figure 1.1: A data stream management system[23]

becomes easy. But in realistic scenario, this is not always feasible. So we need some approaches which generates approximate answers by making use of small available memory.

1.1 Problem Statement

1.1.1 Credit card fraud detection

There are applications where the system should react to an event that needs an immediate response, such as credit card fraud detection, a service outage or a change in a patient's medical condition. For example, in credit card fraud detection, the bank needs to produce real time response to the transaction (allow or block) as soon as the transaction is requested. In this application, either the card is physically stolen and customer is unaware of it, or the fraudster knows the sufficient details of the card, so that the sanity checking by the bank (before the transaction happens) cannot detect fraud, whether the transaction request came from a genuine customer or not. The only way to detect the transaction is genuine or not is to identify the normal spending behaviour of the customer and find out whether there is some deviation from this normal behaviour. Humanbeing has a tendency to follow a spending pattern and fraudsters will always want to make as much money as possible in small time. So deviation from customers' normal spending behaviour learnt from the transaction data can lead to potential fraud. The transaction cannot be delayed much as the genuine customers would not want to wait for a longer time for their transaction to be successfully completed. There can be millions of such transactions happening. So the detection mechanism needs to be fast to handle huge load in the system.

For Model Building

1. **Input:** Sequence of card transactions.
2. **Output:** Model parameters to Hbase table.

For Prediction

1. **Input:** Stream of ongoing transactions.
2. **Output:** Two streams classifying transactions into fraud and non-fraud.

1.1.2 ATM cash demand forecasting

One other application of real time stream analysis is the ATM cash out modelling. Neural network, time series based approach, regression method or other machine learning algorithms can be applied on the data to find pattern. The demand prediction can be used for estimating optimum amount of cash that should be filled or necessary actions to be taken if ATM cash out situation occurs frequently in order to meet the customer demand at minimum banker cost. In this example the real time response may not be necessary. But the huge data coming from various ATM should to analysed using algorithms in a distributed system for speed up.

Cash demand needs forecasting accurately before the actual demand occurs. The data can show seasonality or other effects (including uncertain holidays or socio-cultural/commercial event etc). The model also needs to re-evaluated omitting obsolete past data.

For Model Building

1. **Input:** Sequence of demand values.
2. **Output:** Model parameters to Hbase table.

For Prediction

1. **Input:** Stream of queries to find next demand
2. **Output:** Predicted demand.

The report is organized in the following manner. Chapter 2 talks about some related work. Chapter 3 demonstrates the Architecture. Chapter 4 and 5 outlines the Implementation details. Chapter 6 briefs about the experiments performed and analysis done on them. Chapter 7 describes the future work on the project. Chapter 8 concludes the report.

Chapter 2

Related Work

1. **Analytics on big fast data using real time stream data processing architecture** [21]:

   There are four aspects of big data analytics namely volume, velocity, and variety. Hadoop handles volume and variety part of it. On the other hand Stream processing needs to handle the velocity part of it. A system is described in the paper which makes use of apache Storm and Kafka. Storm by nature guarantees at-least once data processing, fault tolerance, scalability. Model is built off-line using HMM on a Hadoop cluster. This model is used to detect anomaly.

   A messaging systems is required to generate stream. But Traditional messaging systems do not scale very well.

   (a) Message can be deleted from queue before consumed. They are not reliable.
   (b) Message delivered to the stream processing system may not be acknowledged. So there can be a chance of consuming a single message twice.
   (c) Each disk can perform only one seek at a time - which limits parallelism.
   (d) Parallelism can be achieved by using B-Trees with a very sophisticated page or row-locking implementation to avoid locking the entire tree on each operation. But this solution is expensive.

   Kafka is used as messaging system which is integrated with the Storm spout. Kafka is a scalable, distributed, fault-tolerant messing system. Storm-bolt used Esper for event batching. It can combine event into a batch of say 1 minute. For instance, events can be batched for the previous 1 minute and a fault can be found within this batch. The probability of the sequence in 1 minute batch can be calculated using the model that is built earlier.

   If the probability is below some predefined threshold then the sequence is anomalous.

   Multiple models can be built for multiple predictions. Each prediction can be associated with a given Kafka topic. Different Kafka spout can fetch data for each prediction. Esper-bolts can fetch the model for that topic from the corresponding caching layer.

2. **Scalable distributed event detection for twitter** [25]:

   This paper talked about a distributed, real-time, event detection system. An on-line clustering algorithm is implemented up using a Storm topology on a cluster of machines.

   To get the clusters of documents the incoming document is matched with the similar documents and assigned to a suitable cluster id. Locality sensitive hashing technique is used for finding similar documents. In locality sensitive hashing, nearest documents have same key after hashing. This method scales linearly.

   The locality sensitive hashing technique is used to reduce the number of clusters to be created on the last bolt (k-means bolt). K-means clustering bolt maintains clusters of documents based upon the closest document found. A threshold is used to determine whether each incoming document should be added to an existing cluster or should form a new cluster

   The performance is measured in terms of effectiveness, efficiency and scalability. The effectiveness is determined by false negatives. The efficiency is measured by average and maximum throughput of the end-to-end system and each of the sub-phases of the system. The scalability is measured by checking whether increasing the processing cores increases the throughput. It increased linearly.

   Different bolts used for different sub-phases can have different replication factor. The replication factor of the bolt is determined by utilization of bolt for a particular sub-phase. Slower bolts are given higher replication factor. Increasing parallelism of bolts is a good idea if it increases the throughput (efficiency).

3. **Big data analytics on high velocity streams: a case study** [20]:

   This paper talks about real time analysis of twitter data and Bitly data. The goal of the paper is to process and combine two different data flows in real-time.

   The solution for this problem is solved using the distributed, fault-tolerant, real-time stream processing framework called storm (which also scales out very well to support parallel processing on a cluster of commodity machines). The main idea is to track the frequency of terms of stream over a moving window.

   Moving window size and emit frequency of the tweets can be tuned to improve memory requirement per bolt and delay without loss of information.

   The system is tested for scalability by checking the processing rate per second with varying number of supervisor nodes. The system throughput varied proportionally with the number of supervisor nodes (increasing parallelism)

4. **Cloud::Streams** [9]:

   Cloud::Streams is a linearly scalable, fault-tolerant distributed routing framework for data integration, collection, and streaming data processing and real-time analytics system. It can connect to various data sources and produce output to Hadoop, NoSql Database, Dashboards etc. Users can write application logic and it will be performed in real-time with Cloud::Streams. It can process millions of records per second. It can support Continuous queries with streams.

   Internally they couple Hadoop with Storm and Kafka for real-time stream processing - plus a variety of NoSQL and SQL databases for interactive querying.

5. **Microsoft Streams** [12]: It proposes novel architectures, processing techniques, models, and applications to real-time processing of data stream. The researchers are focused on factors like scale-out, high-availability, query optimization, to develop new platforms for stream processing.

6. **Forecast combinations of computational intelligence and linear models for the NN5 time series forecasting competition** [16]: In this paper authors have proposed mixing of different modelling algorithms. The prediction of the time series considered the seasonality of the data. Although various non-linear approach gives good results, multiple regression model was considered as a descent choice. The best choice of the algorithm combination gave a range of SMAPE between 19% - 21%.

Chapter 3

Architecture

3.1 Hadoop for batch processing

The Apache Hadoop [1] software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single server to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

Hadoop follows the master/ slave architecture. One node serves as Namenode (master) that manages the file system namespace and regulates access to files by clients. The other node serve as Datanode which contains data. The MapReduce framework consists of a single master ResourceManager and one slave NodeManager per cluster-node and one application manager per application. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. So it needs the reading and write back data loads. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Minimally, applications specify the input/output locations and supply map and reduce functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the job configuration. The Hadoop job client then submits the job (jar/executable etc.) and configuration to the ResouceManager and it in-turn chooses a node to execute ApplicationManager which then assumes the responsibility of distributing the software/configuration to the other slaves(including itself if more than one slot per node is there for running YarnChild), scheduling tasks and monitoring them, providing status and diagnostic information to the master.

3.2 Storm for real time stream processing

Apache Storm is an open source, distributed, real-time, fault tolerant, and highly scalable platform for processing streaming data.[10] It supports range of applications which includes real-time analytics, on-line machine learning, continuous computation. It is the real-time equivalent to Hadoop in Big data world. In contrast to Hadoops batch processing, it is extremely fast and can process over a million records per second per node on a modest sized cluster. It is a high throughput, low latency data processing engine. It also provides tight integration with many of the Hadoop tools and other queuing, database systems etc. Storm processes tuples on-the-fly, as streams cannot be stored in a persistent store and then retrieved for processing.

One of the basic ideas of Storm is its **fault-tolerance**. It is achieved in the following way. When workers die, Storm will automatically restart them. If a node dies, the worker will be restarted on another node. The Storm daemons, Nimbus and the Supervisors, are designed to be stateless and fail-fast. So if they die, they will restart like nothing happened. For the details about workers, nimbus and supervisors etc see the subsection 3.2.1. Storm also guarantees that all the tuples are processed.

Another idea of Storm is its **scalability** as topologies can run parallel across cluster of machines. Storm command line client provides rebalance command which can adjust the parallelism of running topologies on-the-fly. Storm partitions data and use available cores for data processing. Number of bolts in various nodes and number of threads in each node can be specified to exploit parallelism.

3.2.1 Storm Internals

1. **Topology** Storm and Hadoop clusters are very similar. Hadoop runs MapReduce Jobs and Storm runs Topologies. MapReduce Jobs eventually finishes but Topologies run forever unless manually killed. Topology is a graph for computation logic. Nodes in the graph indicate available nodes for processing and edges indicate how the data will flow between the nodes.

2. **Nimbus** There are two types of nodes in a Storm cluster namely **master node** and **worker nodes**. Master node runs nimbus daemon which is similar to Hadoop's Resource-Manager. It distributes code in the cluster, assigns tasks to machines and monitors workers for failure. If some worker fails it reassigns the task to some other worker.

3. **Supervisor** Each of the worker node runs supervisor daemon. Supervisor listens to the nimbus for work assignment and start or stop worker process on the worker node. Topology specifies a subset of task to each worker.

4. **Zookeeper** cluster is used for co-ordination between nimbus and supervisors. Nimbus and supervisor daemons are fast-fail and stateless, they can be recovered from failure and start normal operation. All states are stored in ZooKeeper cluster.

See Figure 3.1 interaction between the components.

3.2.2 Storm primitives

Core abstraction of Storm is stream. Stream is an unbounded sequence of tuples. A tuple is a named list of values, and a field in a tuple can be an object of any type. Storm transforms

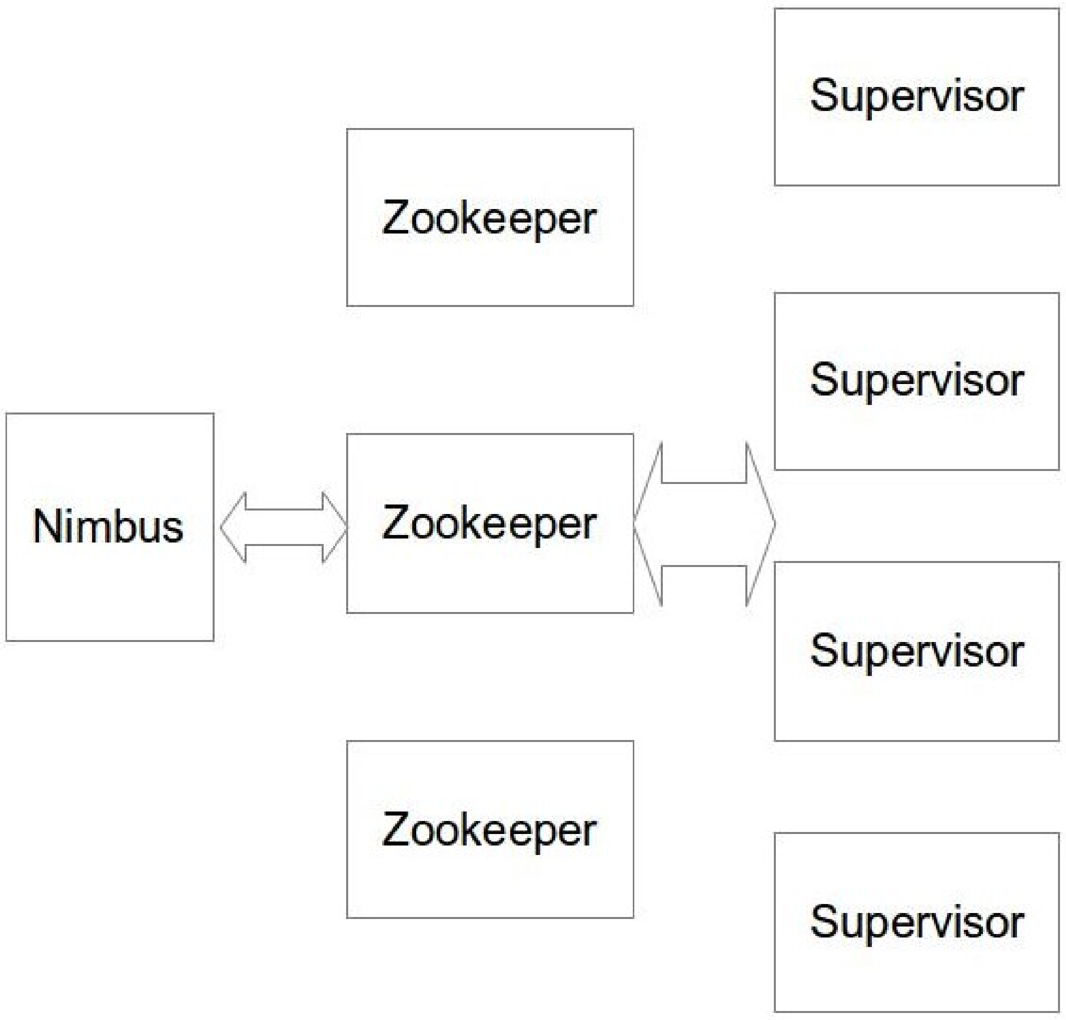


Figure 3.1: Storm nimbus, zookeeper and supervisor

one or more such streams to another in a distributed way. Two basic primitives provided by storm are Spouts and Bolts - constitute a topology. These two are the interfaces for application developer to implement the programming logic.

1. **Spout** acts as source of streams. It connects to the source of streams(like Redis queue, Kafka messaging system, tweeter API etc) to get tuples. There can be multiple spouts gathering streams from various systems in an application.

2. **Bolt** consumes tuples generated from spout(s). Bolts does some processing on the streams and generates another stream as output. It can filter tuples on some condition or can talk to database or can do stream aggregation, join etc. There can be multiple bolts for implementing a complex application logic. An edge from spout to bolt indicates how the tuples flow from spout to the bolt for processing. An edge between two bolts indicates how the tuples can be processed step by step by various bolts.

Storm topologies are extremely parallel in nature. One can specify the amount of parallelism per node in the cluster and storm will spawn threads for achieving the required parallelism.

3.2.3 Example of an Topology

This section explains a sample topology called ExclamationTopology from storm-starter project.

There are one spout and two bolts. The spout emits random strings from a collection of strings. Each bolts appends "!!!" to its input and emits the modified sting. First spout emits strings one by one at an interval of 100 milliseconds. Then first bolt consumes the strings emitted by the spout and emits the modified string to the second bolt. Second bolt consumes the output of the first bolt and emits the modified string stream as final output. Refer Figure 3.2 for the topology graph.

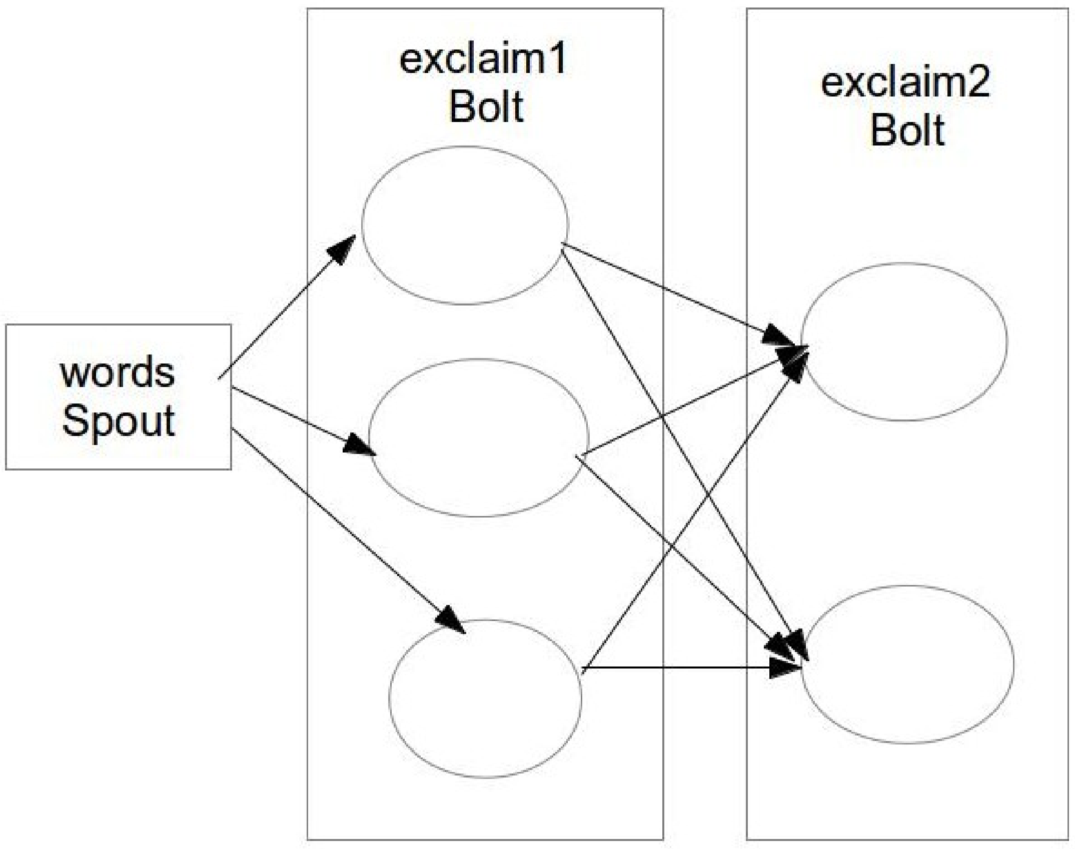


Figure 3.2: Exclaim Topology

The code for the topology is given below.

```
class TestWordSpout extends BaseRichSpout {
  private SpoutOutputCollector collector;
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)
      {
    this.collector = collector;
  }

  public void nextTuple()
  {
    Utils.sleep(100);
    final String[] words = new String[] {"nathan", "mike", "jackson", "golda",
        "bertels"};
    final Random rand = new Random();
    final String word = words[rand.nextInt(words.length)];
    collector.emit(new Values(word));
  }
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word"));
  }
}
```

This is the Spout which extends BaseRichSpout and defines nextTuple() and declareOutputFields() methods. The **nextTuple** method specifies the logic for next tuple to be emitted from the spout (Source of stream) as an object of SpoutOutputCollector class. The **declareOutputFields** defines the field name for the emitted tuple.

```
class ExclamationBolt extends BaseRichBolt {
    OutputCollector _collector;

    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
        _collector.ack(tuple);
    }

    public void cleanup() {
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    public Map getComponentConfiguration() {
        return null;
    }
}
```

The **prepare** method simply saves the OutputCollector as an instance variable to be used later on in the execute method. The bolt will emit output tuple as an object of OutputCollector class. The processing logic for the bolt is written inside **execute** method which emits tuples by calling *emit* method on the object of OutputCollector class. *emit* method inside execute method takes incoming tuple as input and processed output as the second argument. The tuple should be acked which indicates successful completion of data processing. **declareOutputFields** declares the fields for tuples emitted by the bolt (here it is tuple with single field called *word*).

```
public class ExclamationTopology {
    public static void main(String args[])
    {
        int spoutThreads = 10;
        int exclaim1BoltThreads = 3;
        int exclaim2BoltThreads = 2;
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("words", new TestWordSpout(), spoutThreads);
        builder.setBolt("exclaim1", new ExclamationBolt(),
            exclaim1BoltThreads).shuffleGrouping("words");
        builder.setBolt("exclaim2", new ExclamationBolt(),
            exclaim2BoltThreads).shuffleGrouping("exclaim1");
    }
}
```

This part of the code specifies the topology. The topology contains one **spout** called *words* which executes the logic specified by an object of *TestWordSpout* class (logic for generating new tuple) and the number of *threads* for the spout. There are two *Bolts exclaim1 and exclaim2. setBolt* method sets the bolt name, the processing logic for the particular bolt (here it is implemented using an object of ExclamationBolt class). Two bolts get input from spout *word* and *exclaim1* respectively using shuffleGrouping. **"shuffleGrouping"** specifies that the tuples would be randomly distributed from source to bolts. There can be other types of grouping of incoming tuples for a bolt -

1. **Fields grouping** If tuples are partitioned with some field, tuples with same field value with go to the same bolt for processing.

2. **All grouping** All the tuples will go to all the available bolts.

and many more. For details refer to [10].

```
        Config conf = new Config();
        conf.setDebug(true);
        conf.setNumWorkers(2);
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("test", conf, builder.createTopology());
        Utils.sleep(10000);
        cluster.killTopology("test");
        cluster.shutdown();
```

For running topology in local mode an object of LocalCluster is created and the topology is submitted to the virtual cluster using the *submitTopology* method. The topology is run for 10000 milliseconds and then it is killed (unless it would run forever).

For running topology in distributed mode use the following method.

```
        StormSubmitter.submitTopology("topologyName", conf, builder.createTopology());
```

setNumWorkers method specifies how many worker processes are to be there per node in a storm cluster. Each node of a storm cluster may run one or more topologies. Each worker process can contain executers which are threads for some components (bolt/spout) of a particular topology. It runs subset of all tasks for the topology. Each executer executes tasks for the component. Refer Figure 3.3.

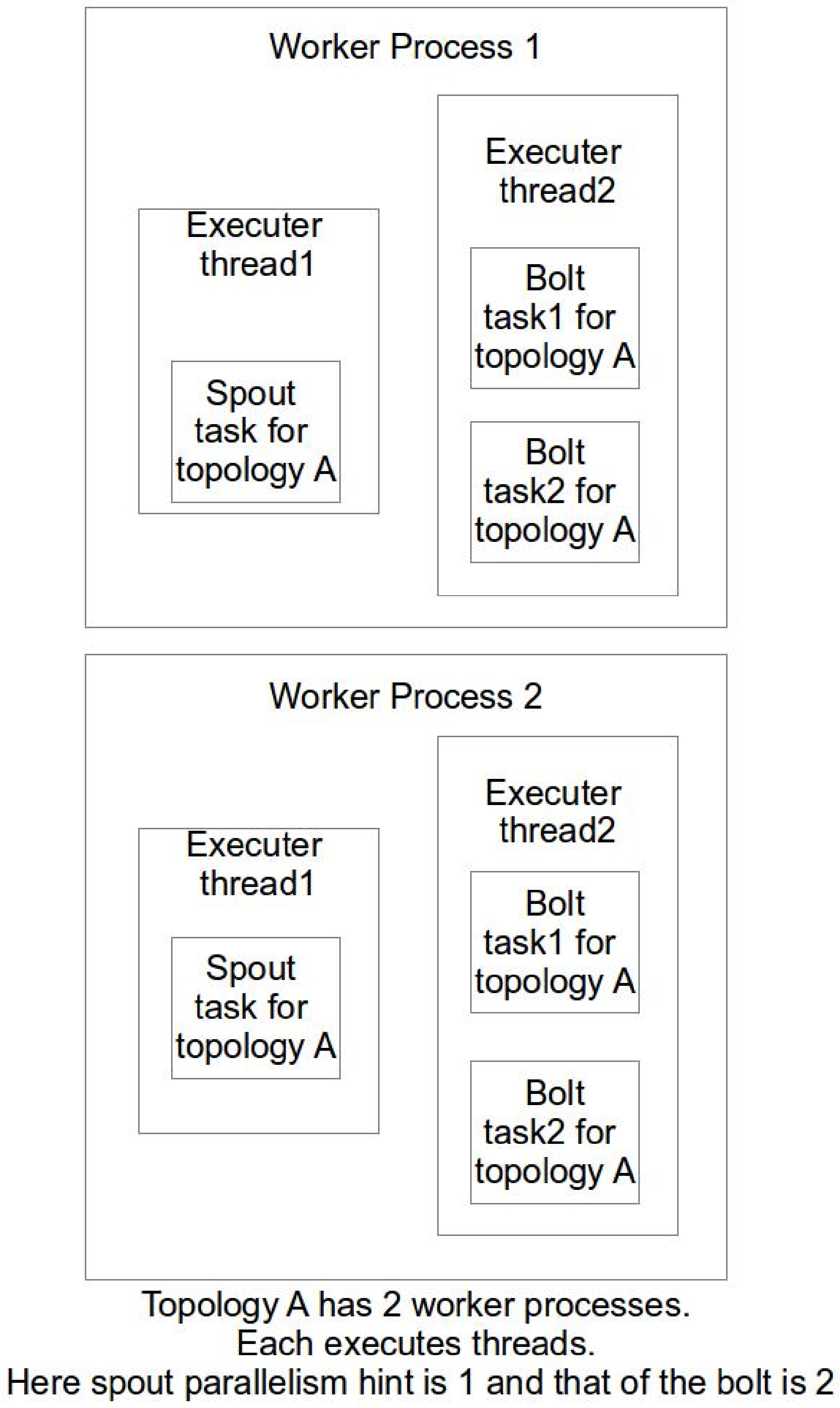

Worker Process 1
Executer thread1
Spout task for topology A
Executer thread2
Bolt task1 for topology A
Bolt task2 for topology A
Worker Process 2
Executer thread1
Spout task for topology A
Executer thread2
Bolt task1 for topology A
Bolt task2 for topology A
Topology A has 2 worker processes.
Each executes threads.
Here spout parallelism hint is 1 and that of the bolt is 2

Figure 3.3: Workers, executers, threads for storm topology

setDebug method with argument true denotes that debugging messages will be shown for each tuple. It is useful for local cluster mode.

3.2.4 Running Storm topology

To run a simple topology all the jars of the application should be packaged into a single jar and jar is run using the following command.

storm jar all-my-code.jar backtype.storm.MyTopology arg1 arg2

The main class here is in the backtype.storm.MyTopology class. *storm jar* connects to nimbus and uploads the jar. Storm nimbus submits the topology to the cluster.

3.2.5 Killing Storm Topology

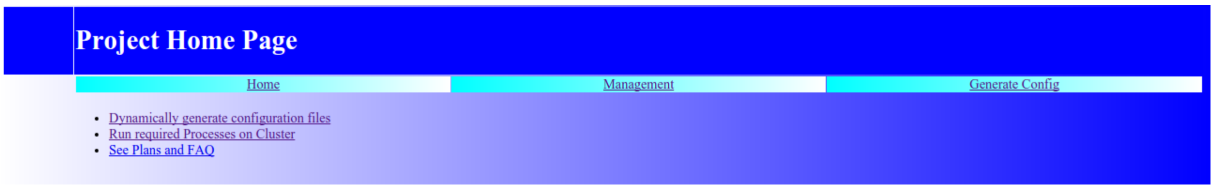
A running topology can be killed using the following command

storm kill {*topology_name*}

3.3 Automation and Web content

The process of making the cluster up and running (the required java processes) has been automated using a web page. See Figure 3.4

1. **Home page**



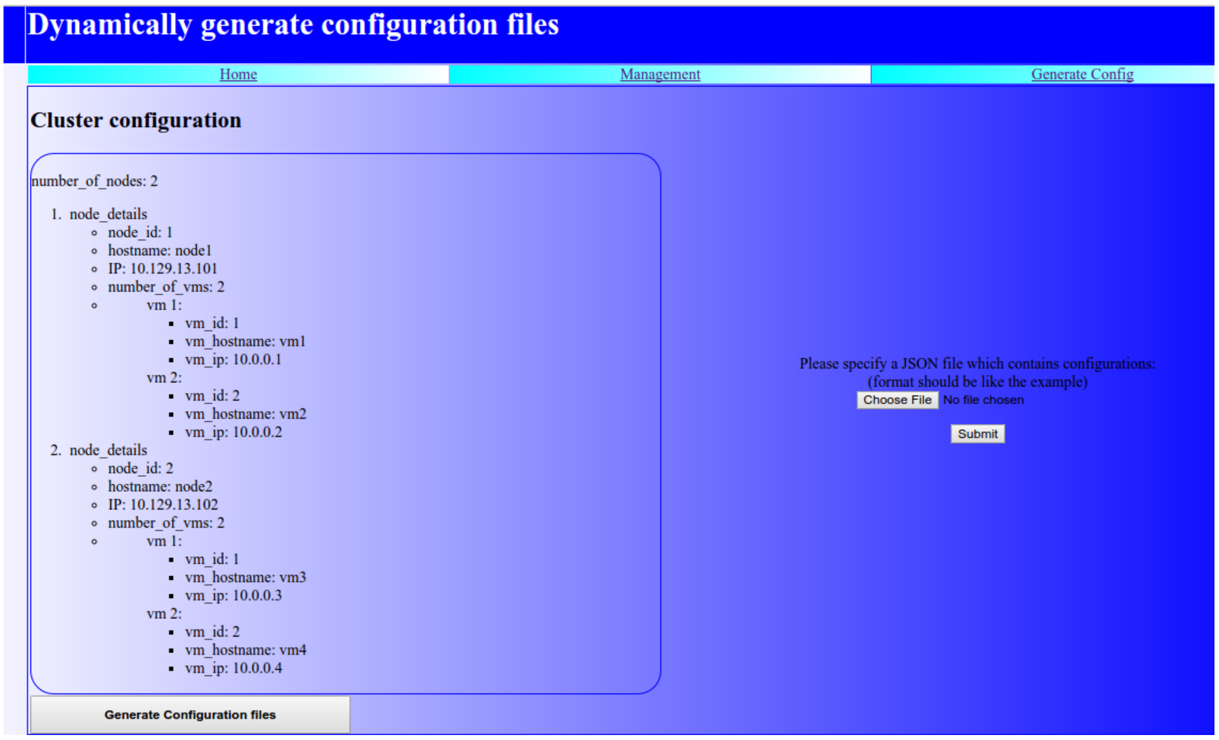
Project Home Page

Home | Management | Generate Config

- Dynamically generate configuration files
- Run required Processes on Cluster
- See Plans and FAQ

Figure 3.4: The home page

2. **Generate configuration file page** The configuration files of the open source software should be changed according to the cluster setup each time. This is automated using the webpage. The only work needs to be done is to create and start nodes in the cluster; Define the cluster structure in a json file. It will generate the script from the page and the generated shell scripts can be copied and pasted and run on individual machine . The cluster should be up and running. See Figure 3.5



Dynamically generate configuration files

Home | Management | Generate Config

Cluster configuration

number_of_nodes: 2

1. node_details
   - node_id: 1
   - hostname: node1
   - IP: 10.129.13.101
   - number_of_vms: 2
   - vm 1:
     - vm_id: 1
     - vm_hostname: vm1
     - vm_ip: 10.0.0.1

     vm 2:
     - vm_id: 2
     - vm_hostname: vm2
     - vm_ip: 10.0.0.2
2. node_details
   - node_id: 2
   - hostname: node2
   - IP: 10.129.13.102
   - number_of_vms: 2
   - vm 1:
     - vm_id: 1
     - vm_hostname: vm3
     - vm_ip: 10.0.0.3

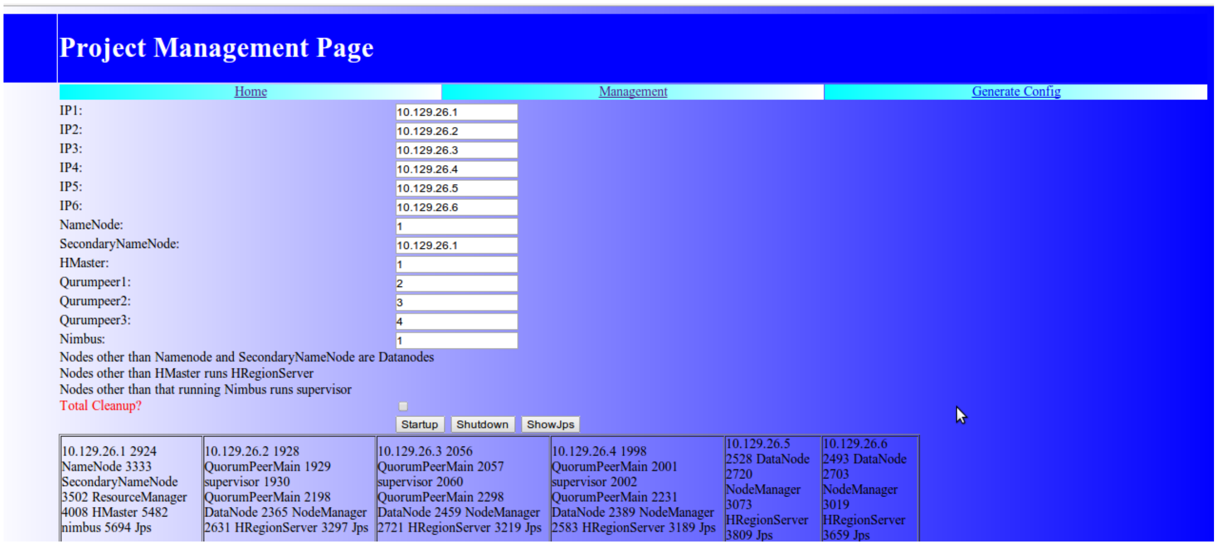
     vm 2:
     - vm_id: 2
     - vm_hostname: vm4
     - vm_ip: 10.0.0.4

Please specify a JSON file which contains configurations:
(format should be like the example)

Figure 3.5: Generating configuration file page

3. **Start/Stop page** This reduces the effort of start various processes in a cluster and shut down the same. The failure of different processes are handled and taken care of at the back-end. In the back end Java Servlet connects to the nodes by ssh and executes the commands. It also checks if some process is not running, it again re-runs the process. It

saves huge time for making the cluster up and running and shut down when necessary. See Figure 3.6



IP1:	10.129.26.1
IP2:	10.129.26.2
IP3:	10.129.26.3
IP4:	10.129.26.4
IP5:	10.129.26.5
IP6:	10.129.26.6
NameNode:	1
SecondaryNameNode:	10.129.26.1
HMaster:	1
Qurumpeer1:	2
Qurumpeer2:	3
Qurumpeer3:	4
Nimbus:	1

Nodes other than Namenode and SecondaryNameNode are Datanodes
Nodes other than HMaster runs HRegionServer
Nodes other than that running Nimbus runs supervisor
Total Cleanup? ☐

10.129.26.1 2924 NameNode 3333 SecondaryNameNode 3502 ResourceManager 4008 HMaster 5482 nimbus 5694 Jps	10.129.26.2 1928 QuorumPeerMain 1929 supervisor 1930 QuorumPeerMain 2198 DataNode 2365 NodeManager 2631 HRegionServer 3297 Jps	10.129.26.3 2056 QuorumPeerMain 2057 supervisor 2060 QuorumPeerMain 2298 DataNode 2459 NodeManager 2721 HRegionServer 3219 Jps	10.129.26.4 1998 QuorumPeerMain 2001 supervisor 2002 QuorumPeerMain 2231 DataNode 2389 NodeManager 2583 HRegionServer 3189 Jps	10.129.26.5 2528 DataNode 2720 NodeManager 3073 HRegionServer 3809 Jps	10.129.26.6 2493 DataNode 2703 NodeManager 3019 HRegionServer 3659 Jps

Figure 3.6: Start/Stop page

Chapter 4

Implementation part 1: System description for Fraud Detection

There are two main modules - Training module and Testing module. Each input transaction record is represented using three fields. Customer id, transition id, and a state of transaction represented by a pair of letters. The first letter(L / M / H) indicates *amount spent* (Low / Medium / High respectively), and the second letter (L / M / H) represents *time elapsed since the last transaction* (Large / Medium / High respectively). So there are 9 such states.

A script is used to generate transaction record data in files for both the modules (*training_data, testing_data* respectively). For each customer, the training module trains the system by calculating the probabilities for each state transition and storing them in a HBase table. To simulate streams, transaction data from the *testing_data* file is dumped into a Redis queue and read by testing module one by one. The testing module is used to determine the fraud transactions.

In order to make the system flexible, all the configurations used in the project is read from configuration files (*training.conf* for training module and *testing.conf* for testing module).

Six virtual machines are created in a machine for cluster setup. Each virtual machine has 16 GB of hard disk space and 1024MB RAM. One machine runs Namenode, HMaster, Storm Nimbus daemons. Among the other nodes, three nodes form ZooKeeper quorum. There are two Zookeeper quorum. One ZooKeeper quorum is running for HBase (on port 2181) and other is for Storm (on port 2182). These three nodes also run HRegionServer, Datanode, Storm Supervisor daemons. The remaining two nodes run HRegionServer, Datanode, Storm Supervisor daemons. The Redis server is running on the first node on port 6379.

4.1 Training Module

Training module consists of a preprocessing step followed by a chain of map-reduce jobs.

4.1.1 Various Hadoop Jobs

1. **Data generation** One script (*generate_transactions.py*) is used to generate training data using the command

   python generate_transactions.py arg1

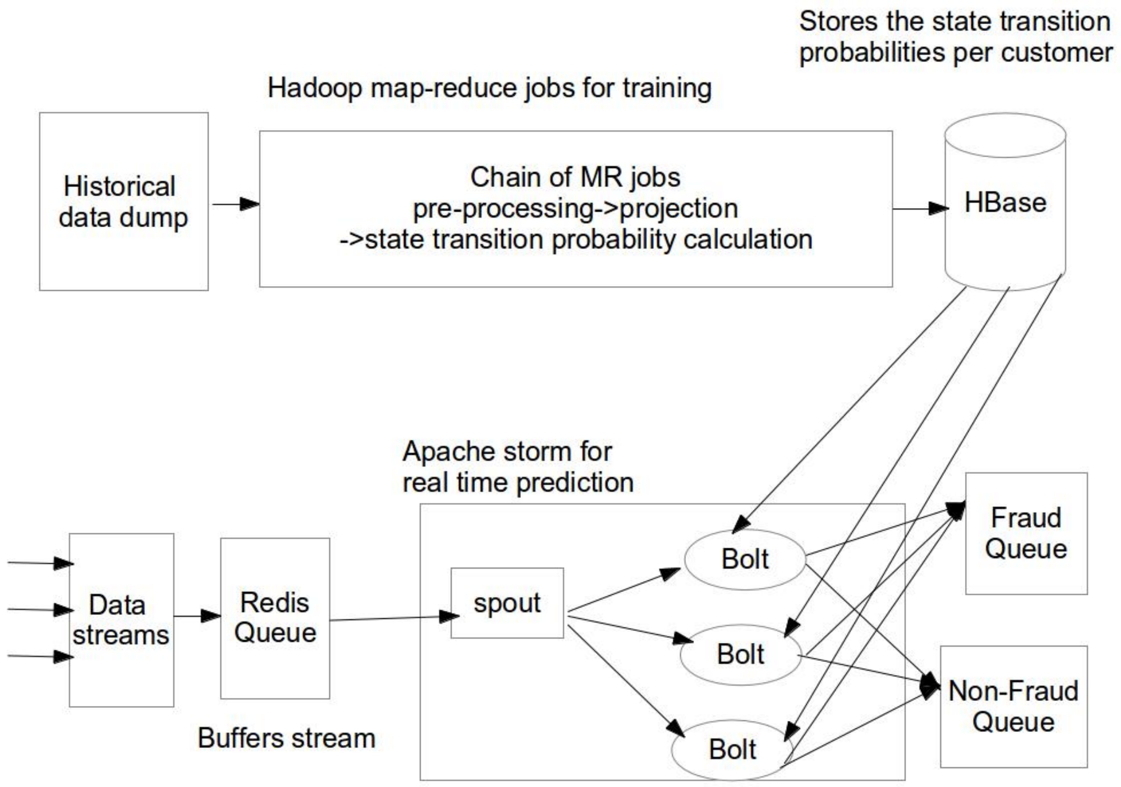

Stores the state transition probabilities per customer
Hadoop map-reduce jobs for training
Historical data dump
Chain of MR jobs pre-processing->projection ->state transition probability calculation
HBase
Apache storm for real time prediction
Data streams
Redis Queue
spout
Bolt
Bolt
Bolt
Fraud Queue
Non-Fraud Queue
Buffers stream

Figure 4.1: System Architecture

where arg1 represents the number of customers for which the model needs to be generated. *Customers id*, *transaction id* and *time since last transaction* are generated randomly. *Transaction amount* is generated as follows. Customers have three spending groups e.g high, medium and low . Customer from high spending group have 90% of their transactions of high amount and 5% of their transactions of medium and low amount each. Similarly customer from medium spending group have 90% of their transactions of medium amount and 5% of the other two amounts. And so on and so forth for the customers of low spending group. Each of the customer has 100 transaction records to train the system. In the processing steps, output of any step is the input for the next state.

2. **Preprocessing** In this step the dump file is read from the file system and each transaction is augmented with an unique id. The id is simply the line number in which the transaction record occurs in the dump file. This id helped the projection step to identify the order in the sequence of customers' transactions. The modified file is then stored to HDFS (*sqdata*) for further processing.

3. **Projection** Project is the first MapReduce job. Input read by map is the HDFS file generated from the previous phase. In the map phase the customer's id is output key and transaction details is the output value (transaction id is omitted from the data - hence the name projection). So data of a customers always go to same reducer. In reduce phase, a customer's data collected from the map phase is chained together to make a sequence

for the particular customer. After successful completion of the reduce phase the output is stored in a HDFS file (*projdata*) The input data file for Projection job (*sqdata*) is then deleted from HDFS.

4. **Impressionability** StateTransitionProbability is another MapReduce job. Map phase reads input from HDFS file (*projdata*) generated by the previous phase. In this job, one is added to the state transition table if a previous state to next state transition is present in the training data set, per customer. For normalization each state transition count is initialized to 1. Let us say for a previous state i and next state j and the number of transitions present in the training set is $count1[i][j]$. Then for each step i, it is calculated that how many transitions are possible to any state.

   $count2[i] = \sum_{j=1}^{j=9} count1[i][j], \forall i \in [1, 9]$

   (i can take values from 1 to 9, because there are 9 states). State transition probability from previous state i to next state j is calculated like this

   $transition\_probability[i][j] = \frac{count[i][j]}{count2[i]}$

   These state transition probabilities per customer are stored in a HDFS file (*mmdata*) and the old file (*projdata*) is deleted from HDFS

5. **ImportHBase** In this phase output file (*mmdata*) from the previous phase is imported to a HBase table *transition_table*. The row key for the table is the customer id and columns (under a single column family prob) are prob# and count#. Where # denotes states involved in a transition which is evaluated as $9 * i + j$ (where i is the previous state and j is the next state). File (*mmdata*) generated from the previous stage is deleted from HDFS after successful completion of table import phase.

4.1.2 Run code

Maven is used for dependency resolution. The application is compiled into a single jar. The jar is named Training-1.0-SNAPSHOT.jar. The application is run using the following command

```
HADOOP_CLASSPATH=‘${HBASE_HOME}/bin/hbase classpath‘
$HADOOP_HOME/bin/hadoop jar target/Training-1.0-SNAPSHOT.jar
```

4.1.3 Tools used

The technology stack used in the training module are described below.

1. **Hadoop**[1] There is a need for horizontally scalable, fault tolerant system for the training phase. The jobs can be batched together for achieving high throughput. Hadoop's MapReduce paradigm is best suited for this application.

2. **HBase**[2] There is a need for a database which is batch writable from Hadoop. The database should have high random read/write performance and hence support for real time application.

   The Database for the application should support both batch layer and speed layer. In batch layer, it should act like a high throughput, horizontal scalable, fault tolerant database system. In Speed layer it should act like hight performance random read database. Hbase is a good fit for both the requirements.

4.2 Testing Module

Testing module checks whether the incoming transaction is fraudulent or not in real time. Testing module is implemented using a Storm topology. The topology had two components, a Redis Spout and a predictor Bolt. The various parts of the implementation are outlined below:

1. **Data generation and stream simulation** A script *generate_transactions.py* is used to generate testing data. Testing dump file is created using the command (it also creates the data for training).

   python generate transactions.py arg1

   where arg1 is the number of customer. One script file is used to generate data for both the phases. It is convenient because the testing data is generated for the same customers for whom models are built. The script generated testing data for the same customers with different transaction ids and random transaction details (transaction amount and time since last transaction occurred).

   Each customer has a sequence of 45 transactions. Each of the 45 transactions are to be validated against the model created using the training data. Each customer's transaction data is prepended by the last 14 transactions from their training data set.

   The testing data file now dumped into a Redis queue for simulating the streaming behaviour. The idea behind the streaming data simulation is, the data streams can be dumped into the Queue as and when they are generated. The stream data engine can fetch data for processing from the queue.

2. **Outlier detection topology** There can be one or more topologies run in parallel for performing the task. The topology(s) is(are) named as outlierPredictor#, where # denotes the number of topology that can be run on cluster in parallel. There are 3 workers processes and 3 threads for each of the two components (Spout and Bolt). These numbers can be configured from the configuration file *testing.conf.*

3. **Redis Spout** Redis spout is used to read data from Redis queue one by one. The tuples are fed to the bolts for processing. As explained in the section 4, in this application, it is necessary to send same customer's data to a particular bolt for processing. So the tuples are shuffled using **fieldsGrouping** on the customer id. By doing this, same customer records will always go to same bolt.

4. **Predictor bolt** Predictor bolt contains logic for fraud detection. Fraud detection for a particular customer's transaction should be carried out in a single bolt. Because the sequence in which the transactions came into the system needs to be maintained for applying the algorithm.

   When the first tuple for a customer comes into the bolt. It reads next 14 transactions into main memory and stores them in a HashMap (as the first 14 transactions in the chain is obtained from the training data and hence they reflect the behaviour of the system and the 15th transaction is the testing transaction data). The HashMap contains customer id as key and sequence seen till now as the value. In a chain of 15 transactions, like $O_1, O_2, \ldots, O_{15}$ each of the transaction is checked with its previous (except the first one) transaction and cumulative miss-probability is calculated from previous state to next state.

$missProbability[prevState][nextState] = stateTransitionProb[previousState][j]$,
$\forall j \in [1,9], j \neq nextState$

The cumulative miss-probability of the sequence is then calculated using
$metric = \frac{\sum_{previousState=1}^{previousState=9} missProbability[previousState][nextStateInTheSequence]}{lengthOfChain}$

If the metric value is grater than threshold (specified in the configuration file) then the transaction is said to be fraud and inserted into the fraud queue and the transaction is deleted from the chain. Otherwise the transaction is non-fraud and inserted into nonFraud queue and the transaction is kept in the chain for future transaction fraud prediction. If the transaction is not fraud then the 1st transaction in the chain is deleted for making room for the new transaction to come into the chain.

The state transition probabilities are read from the HBase table called *transition_table.*

5. **Output** There are two queues of transactions depending on the outcome of the prediction, e.g *fraudQueue* and *nonFraudQueue.* The transactions on the nonFraudQueue are accepted by the system. The transactions on the fraudQueue are cancelled and some more actions can be taken by the bank authority.

Chapter 5

Implementation part 2: System description for Cash Demand Forecasting

The task is to come up with a predictive model for time series data using multiple linear regression.

The training set is a time ordered observations $y_1, y_2, .., y_t$ and the model should predict the next unknowns $y_{t+1}, y_{t+2}, .., y_{t+h}$ i.e h next observations from the past seen values.

A sample data for an ATM is shown in 5.1

There are two main modules - Training module and Testing module. Each input transaction is denoted as triplet of 3 fields (ATM id, date, amount) Hadoop application is written for modelling each of the time series (for each ATM transactional data) In the model training part one multi-linear regression is fitted for the incoming transactions per ATM. The coefficient of the linear regression are stored in the Hbase table for prediction. The test data can only contain the ATM id for which the next days demand is predicted (other parameters of the test data can be ignored e.g date field) Apache Storm is used for this.

In order to make the system flexible, all the configurations used in the project is read from configuration files (prof.conf for training module and testing.conf for testing module).

All the services of the Hadoop, Storm nimbus, Storm supervisor, Zookeeper instances are run.

5.1 Training module

5.1.1 Various Hadoop Jobs

(a) **LinearRegression_ATM job:** In the map phase the transactions are sorted according to the ATM ids. In the reduce phase multiple linear regression model is fitted for each of the ATM. In multiple linear regression, the fitted curve is not assumed to be a straight line. It is only assumed that the relationship between the dependant and

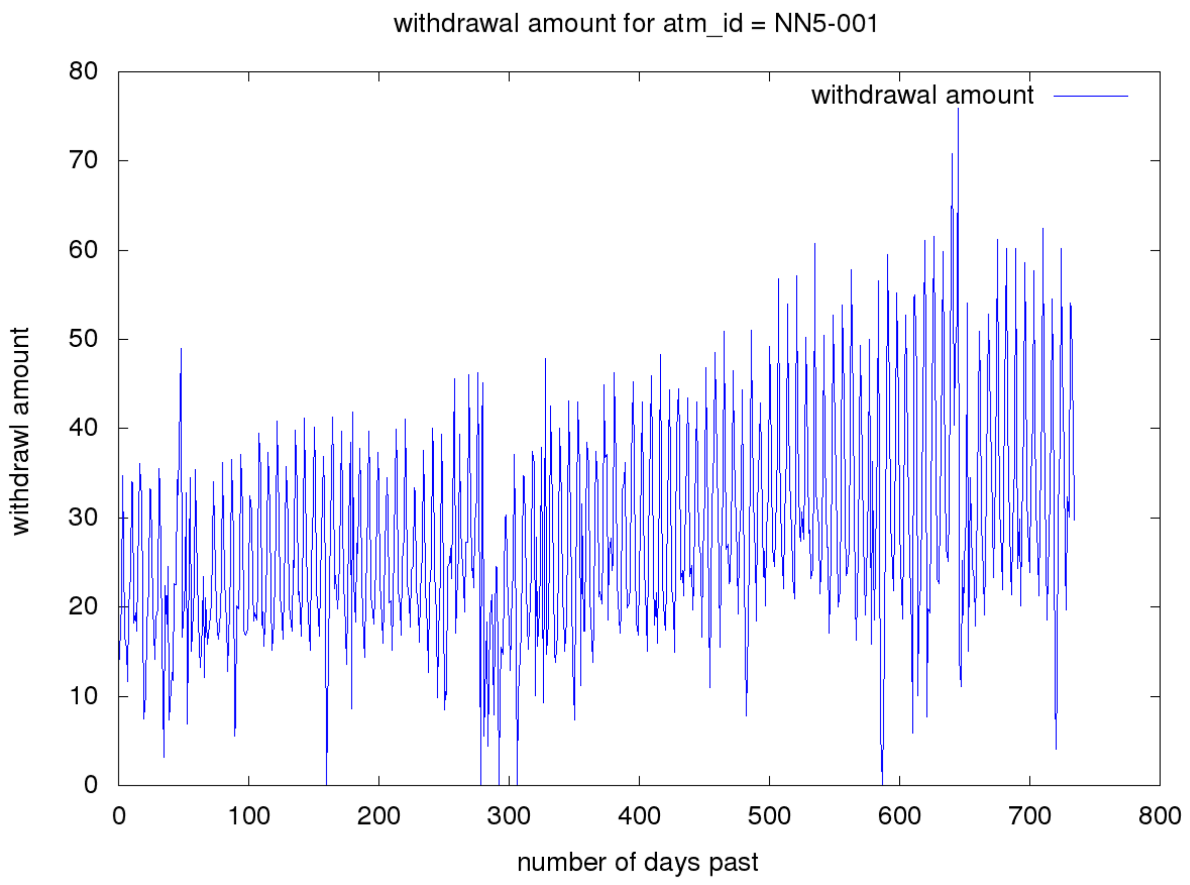


Figure 5.1: Sample training data

each of the independent variables is linear. The considered model is

$$f(d) = \sum_i \alpha_i h_i(d_i l_i)$$

where α_i regression co-efficient , and h_i are information about cash demand. The cash prediction is done for a day d. l_i are used to find h_i. [27]

The various independent variables are:

i. The average recent demand history (h_1) - the average of l_1 days cash demand before the given day d.

ii. The average weekly history (h_2) - the average of l_2 day of week's (same as the day of week of d) cash demand before the given day d. If d is a Tuesday, h_2 may be an average of cash demand on l_i previous Tuesdays.

iii. The average monthly history (h_3) - the average of l_3 day of month's (same as the day of month of d) cash demand before the given day d. For instance, if d is 1st of the d 's month, then h_3 may be an average of cash demand on 1st of previous three months. For 31st and leap years the previous day is calculated using last available month date.

The least square method was used to fit the multiple linear regression curve and it was implemented using **Normal Equation** method

$$\theta = (X^T X)^{-1} X^T \vec{y}$$

Where X is the feature vector of dimension $m * n$, where there are m number of training examples and n features ($n = 3$ here), y is the demand vector ($m * 1$) For the time being I have used the value of $l_1 = 20, l_2 = 20, l_3 = 12$

The accuracy was measured using two parameters R2 and SMAPE. Accuracy figures are explained and plotted in Section 6.1 .

Although The current modelling gave promising accuracy, there are several rooms for improvements

i. More features can be added. Experts can review the model.

ii. Some event other than the seasonality of the data may be useful for modelling the behaviour. e.g natural calamity, huge commercial or cultural event happening nearby may boost up/down the demand. These needs extra data as part of training.

iii. The demand pattern is time varying. So after some time there would be a considerable amount of error if the model is considered stationary. The model should be re-evaluated after certain time or after error goes beyond threshold. Research needs to be done for finding the optimum amount of threshold to minimize both tolerance and the cost of computing.

iv. Non-linear relationship between the dependant and independent variable can be modelled.

v. Different time in a year can have different models.

(b) **HBaseTableImporter Job:** This Hadoop job connects to HBASE client and uploads the linear regression coefficients in the HBASE table named 'theta_per_atm'. The key of the table is the 'ATM id' and the column family contains the theta and pastData column family. Theta column family contains columns for corresponding coefficient and pastData contains the daywise demands of past data (for future prediction).

5.1.2 Run Application

Maven is used for dependency resolution. The application is compiled into a single jar. The jar is named Atm_ex-1.0-SNAPSHOT.jar . The application is run using the following command

HADOOP_CLASSPATH=‘$HBASE_HOME/bin/hbase classpath ‘$HADOOP_HOME/bin/hadoop jar target/Atm_ex-1.0-SNAPSHOT.jar

The other details about running the application are consolidated in the README file under the base directory of the project.

The technology stack used in the training module are described below.

(a) **Hadoop** There is a need for horizontally scalable, fault tolerant system for the training phase. The jobs can be batched together for achieving high throughput. Hadoops MapReduce paradigm is best suited for this application.

(b) **Hbase** There is a need for a database which is batch writeable from Hadoop. The database should have high random read/write performance and hence support for real time application. The Database for the application should support both batch layer and speed layer. In batch layer, it should act like a high throughput, horizontal scalable, fault tolerant database system. In Speed layer it should act like high performance random read database. Hbase is a good fit for both the requirements.

5.2 Testing Module

Testing module predicts the next transaction of the incoming ATM id. Testing module is implemented using a Storm topology. The topology had two components, a Redis Spout and a predictor Bolt.

The testing data file now dumped into a Redis queue for simulating the streaming behaviour. The idea behind the streaming data simulation is, the data streams can be dumped into the Queue as and when they are generated. The stream data engine can fetch data for processing from the queue.

5.2.1 Various Storm Components

(a) **Topology predict_cash:** There can be one or more topologies run in parallel for performing the task. The topology is named as predict_cash. There are 3 workers processes and 3 threads for each of the two components (Spout and Bolt). These numbers can be configured from the configuration file testing.conf.

(b) **Redis Spout:** is used to read data from Redis queue one by one. The tuples are fed to the bolts for processing. The tuples are shuffled using fieldsGrouping on the ATM id. By doing this, same ATM records will always go to same bolt and make use of the memory and cache of the node which improves the performance.

(c) **Predictor bolt:** This contains the logic for prediction of next days cash of the incoming transaction. It is a good idea if a particular ATM's data is processed in a particular node. When a ATM id is entered into the bolt it consults the 'theta_per_atm' table which contains the the next day for which the demand is to predict and it also contains the latest predicted/stored values of the statistics. Each bolt predicts the next day cash demand and and stores the predicted demand in the 'theta_per_atm' table along with the next day for which cash demand will be predicted by the bolt (particular ATM).

The previous days demand and regression parameters are read from the HBASE table ('theta_per_atm')

5.2.2 Running Application and Output

(a) **Run Application:** The application is run using the following command

```
$STORM_HOME/bin/storm jar target/Atm_ex_test-1.0-SNAPSHOT.jar com.atm_ex_test.atm_ex_tes
```

Other information for running the application and associated processes are all listed in the README file under the project base folder.

(b) **Output:** The ouput of the bolts after processing is stored in the ouput queue of the redis queue which can be used to read and analyse later.

A sample ATM's output(actual versus predicted) on the training and test data are shown in 5.2 and 5.3

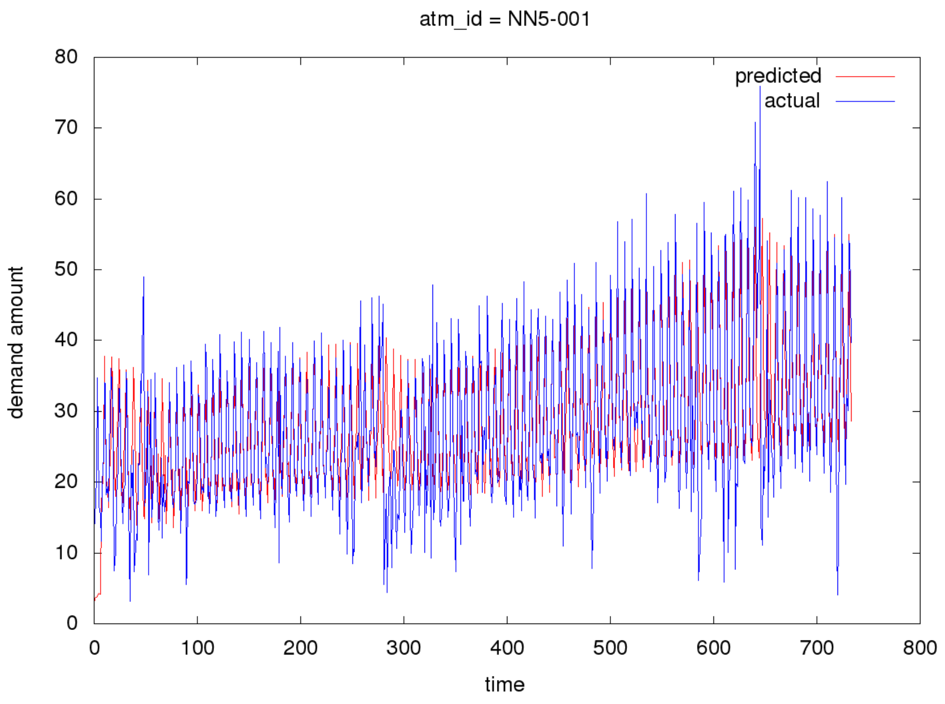


Figure 5.2: A sample output of the training data (actual versus output of regression fitted)

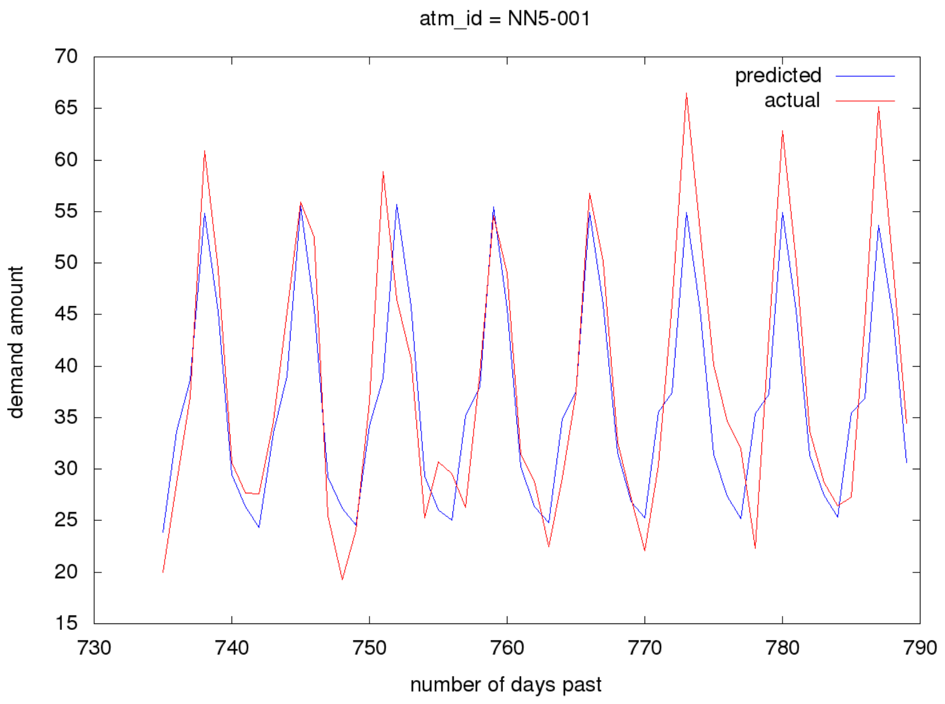


Figure 5.3: A sample output of the testing data

Chapter 6

Experimental Evaluation

6.1 Accuracy Measures

6.1.1 SMAPE

[8]: Symmetrical Mean Absolute Percentage Error is measured using the following equation

$$SMAPE_s = \frac{1}{n}\sum_{t=1}^{n}\frac{\mid X_t - F_t \mid}{\frac{X_t+F_t}{2}} * 100$$

The SMAPE calculates the absolute error in percent between the actual X and the forecast F across all observations t of set of size n for each time series s

The SMAPE measure distribution for the various ATM (training phase) is show in 6.2

The SMAPE measure distribution for the various ATM (testing phase) is show in 6.1

For training data

- Average SMAPE: 21%
- Median SMAPE: 20%

For test data

- Average SMAPE: 23%
- Median SMAPE 22%

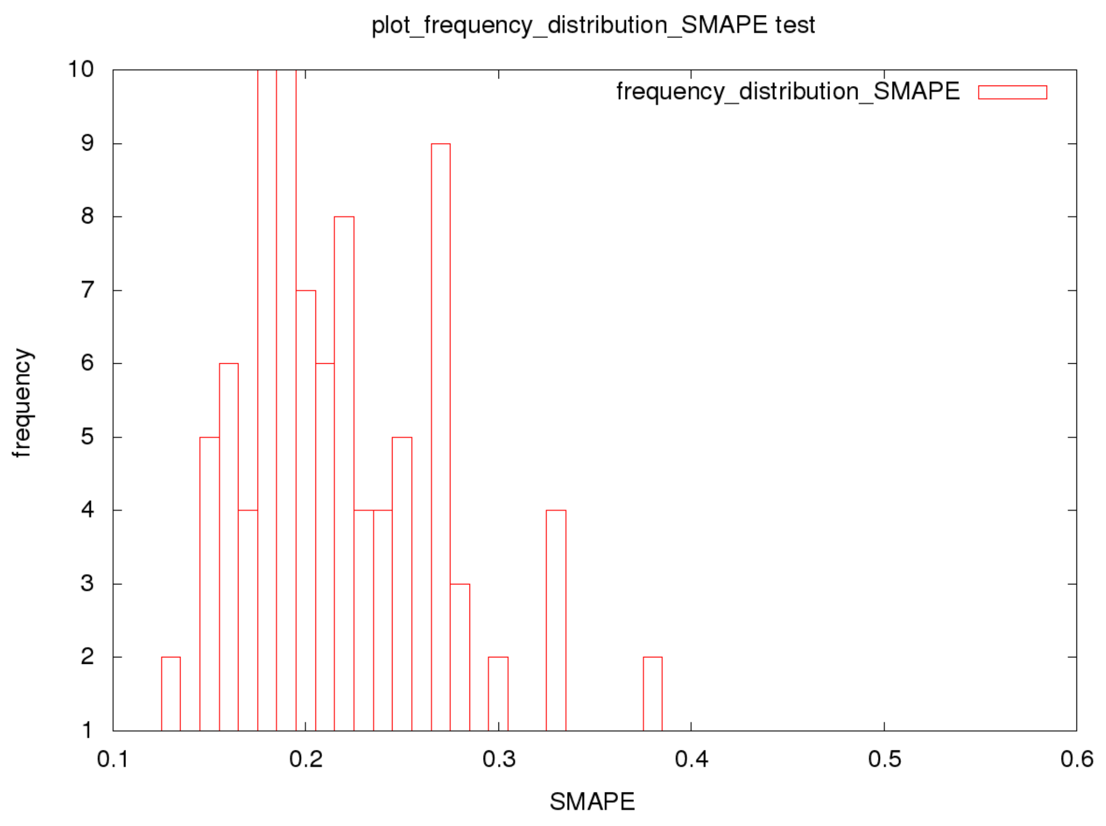


Figure 6.1: Distribution of SMAPE of test data

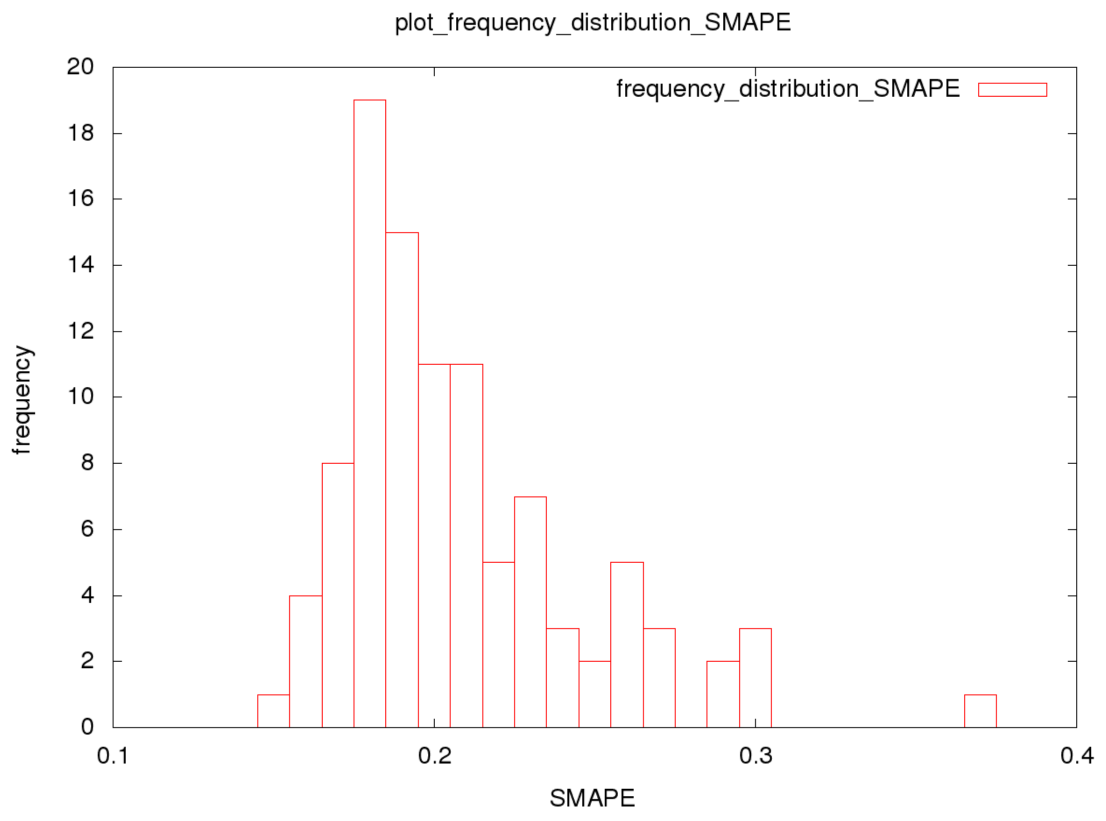


Figure 6.2: Distribution of SMAPE of training data

6.1.2 R^2 measure

[6]If data is a vector of n items $y_1, y_2, ..., y_n$. The corresponding predicted values are $f_1, f_2, ..., f_n$. The mean of all y's is

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

Then the variability of the data set can be measured using three sum of squares formulas: The total sum of squares of deviation of input data (proportional to the variance of the data):

$$SS_{tot} = \sum_i (y_i - \bar{y})^2$$

The sum of squares of residuals, also called the residual sum of squares (sum of squares of difference between actual and predicted)

$$SS_{res} = \sum_i (y_i - f_i)^2$$

The most general definition of the coefficient of determination is

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

For training

1. Average R^2 measure is 49%
2. Median R^2 measure is 50%

For testing

1. Average R^2 measure is 99%
2. Median R^2 measure is 99%

The R^2 measure distribution for the various ATM is show in 6.3 and 6.4 for training and testing respectively.

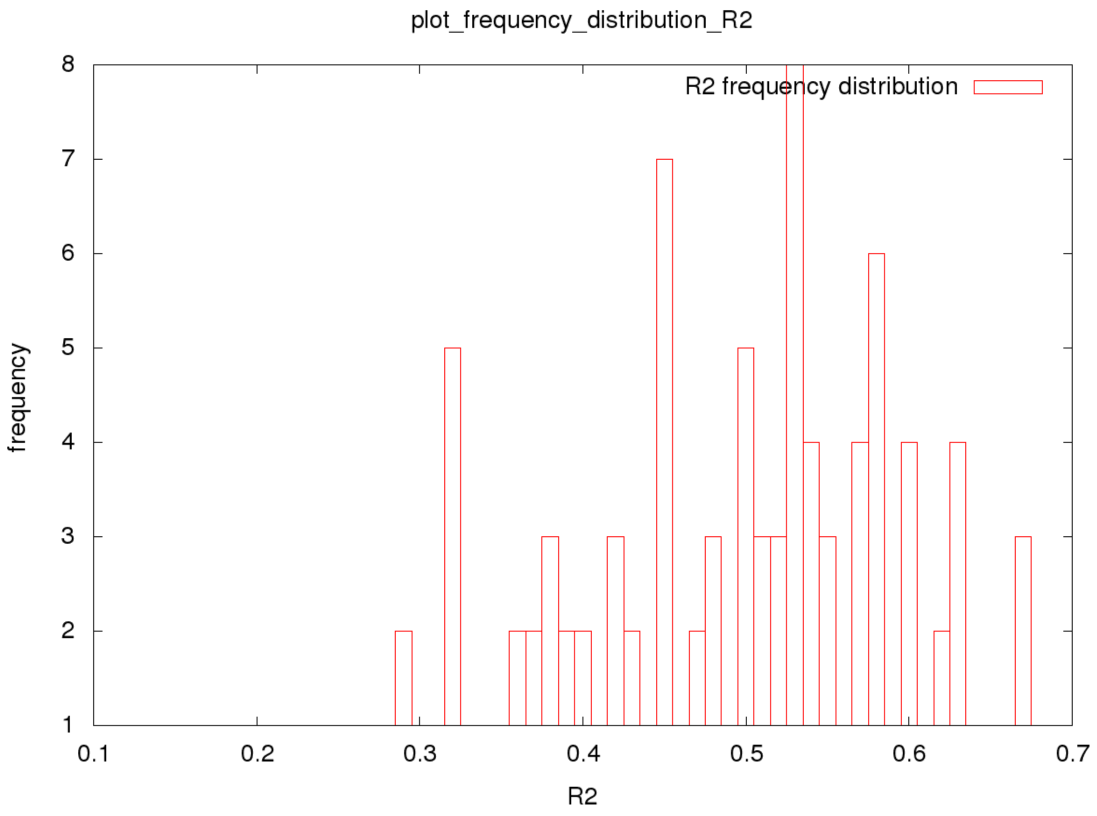


Figure 6.3: The R2 measure distribution for the various ATM (training phase)

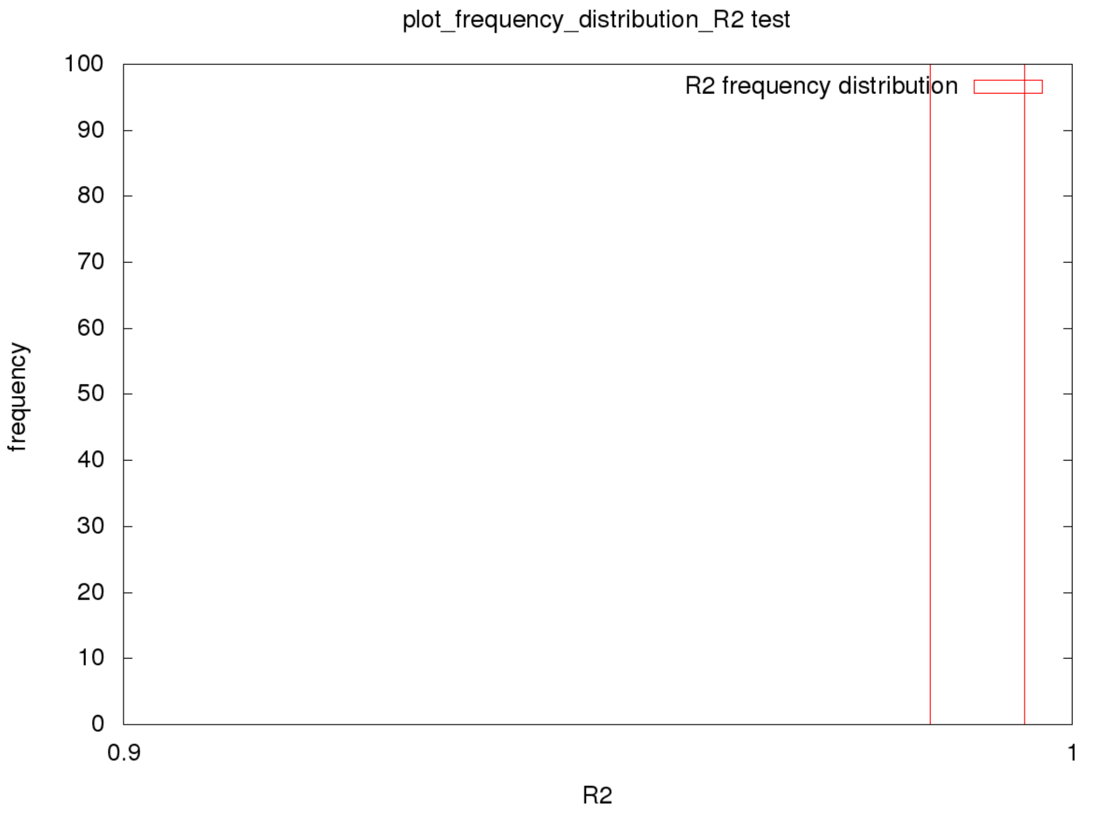


Figure 6.4: The R^2 measure distribution for the various ATM (testing phase)

A sample ATM's prediction deviation from actual value is shown 6.5:

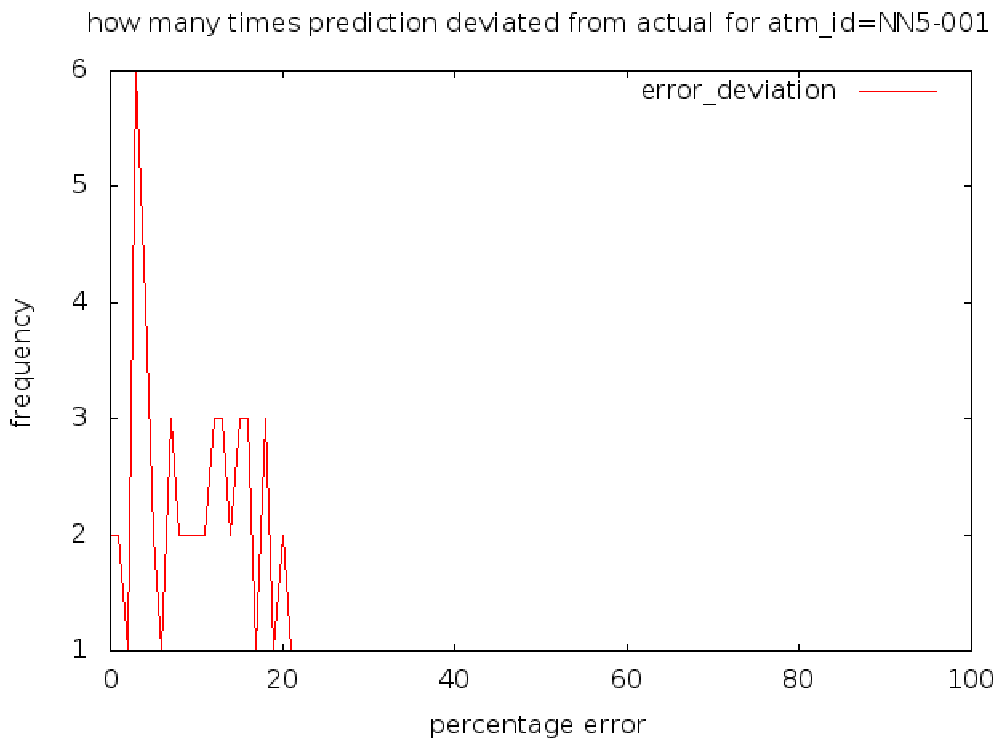
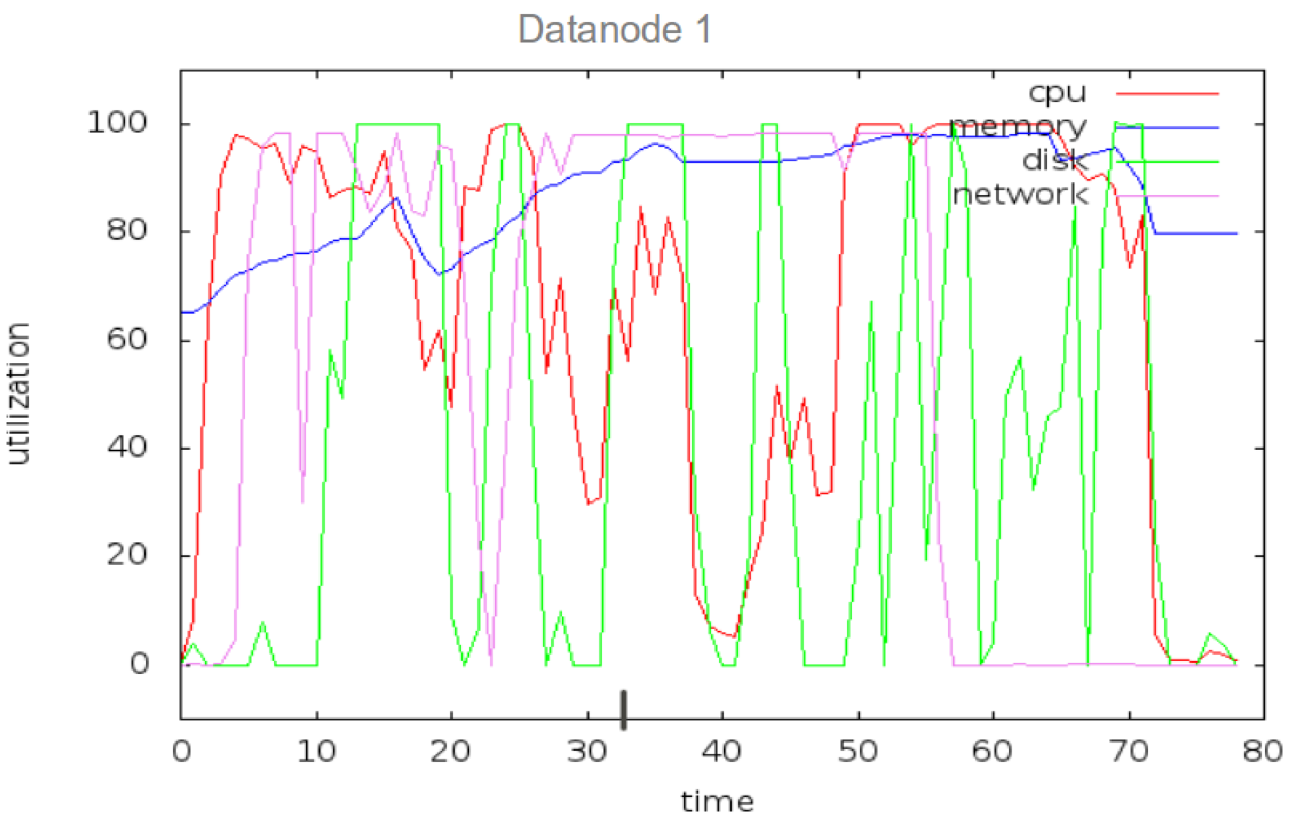


Figure 6.5: A sample ATM's prediction deviation from actual

6.2 Performance measures

6.2.1 Scale Test

1. **Goal:** To find out whether the application suits parallelism.
2. **Hypothesis:** The solution to the problem divides the input into independent sets and operations are done on those sets. So it should scale well in parallel architecture .
3. **Setup:** The training module was run on
   (a) A single machine
   (b) A cluster of 3 machines where 1 node was used as master node and other two were used as slaves
4. **Observations**:
   (a) **Time** For comparing, load was multiplied by 4 for cluster setup (load in terms of input size). Time taken for running the algorithm was less than twice in the clustered setup than that of single node setup.
   (b) **Resources** were distributed evenly among the nodes in the cluster. The resource utilization and time taken (in seconds) in are shown for the two datanodes (slave) in the cluster in Figure 6.6 .



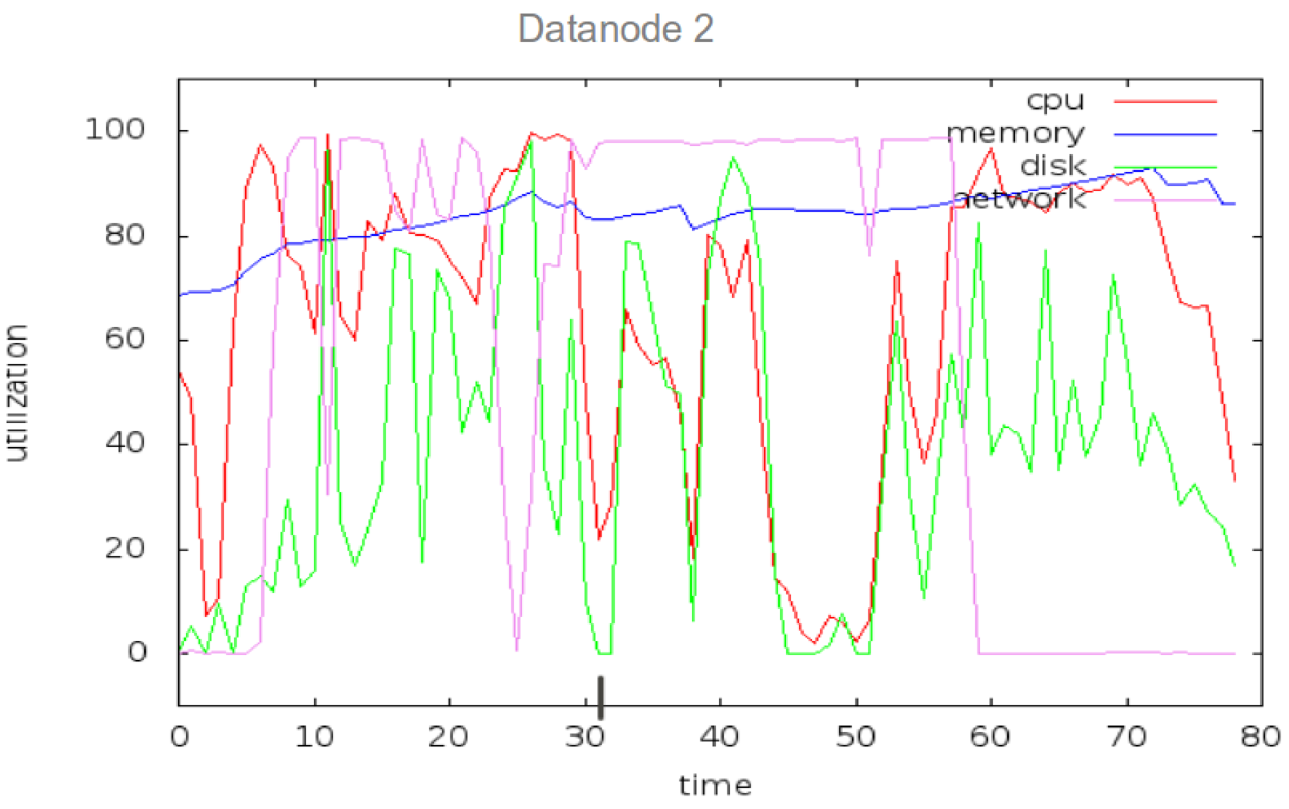


Figure 6.6: Resource utilization of the two datanodes in the cluster while running 4X load as compared to single machine

5. **Inference:** The Hypothesis was true. The application would scale for larger number of

nodes.

6.2.2 Load Test

1. **Goal:** To find whether the system scales well and is distributed design always suitable.
2. **Hypothesis:** Lightly loaded application should be run on single machine instead of a cluster.
3. **Setup:** The application is run
   (a) with a small load in a single machine and twice that load in a cluster of 3 machines(2 slave node).
   (b) with twice as that load in both setting.
4. **Results:** Time taken for running application for different number of nodes is shown in below table. The load is scaled according to size of cluster

	time for Single machine	time for Cluster with 2 datanodes
load X	43s	49s
Load 2X	85s	78s

   It was observed that for small load distributed approach takes more time. Where as for relatively higher load distributed approach takes less time.
5. **Inference:** The hypothesis was true. One machine (master) was not utilized by the Hadoop architecture. It is used for job submission, assigning task and monitoring the slaves. But it has very tiny load when the application is running. The application will definitely scale up for more and more loads with high load given to the system. Also for small amount of load there would be more distribution overhead than computation overhead.

6.2.3 Some Insights

1. It was also observed in a single node reduce phase starts much more faster, but once all map completes, the reduce takes time because all computations are done on a single machine. In cluster mode reduce phase starts later but completes faster as load is shared among the slaves. Figure 6.7 and 6.8 shows the map reduce progress for single and cluster mode respectively.

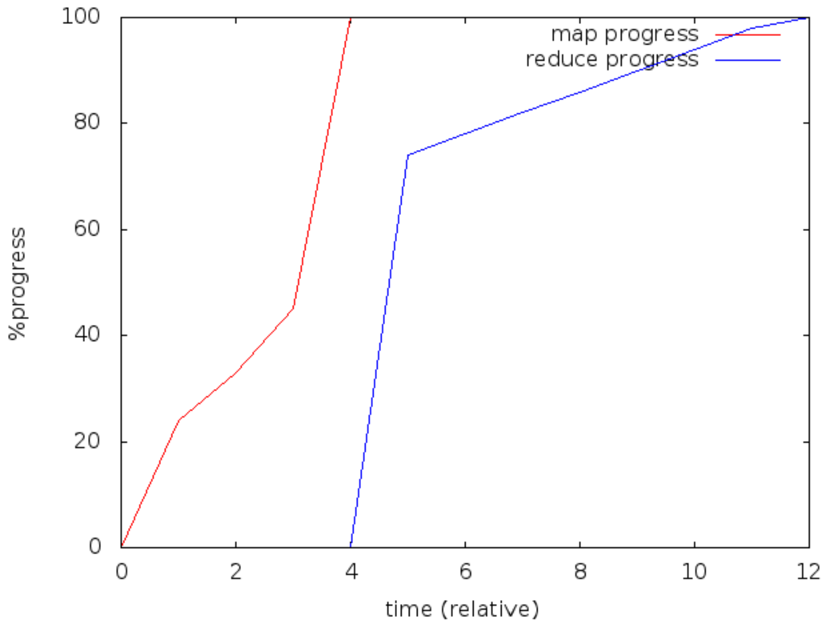


Figure 6.7: Map reduce progress for single machine

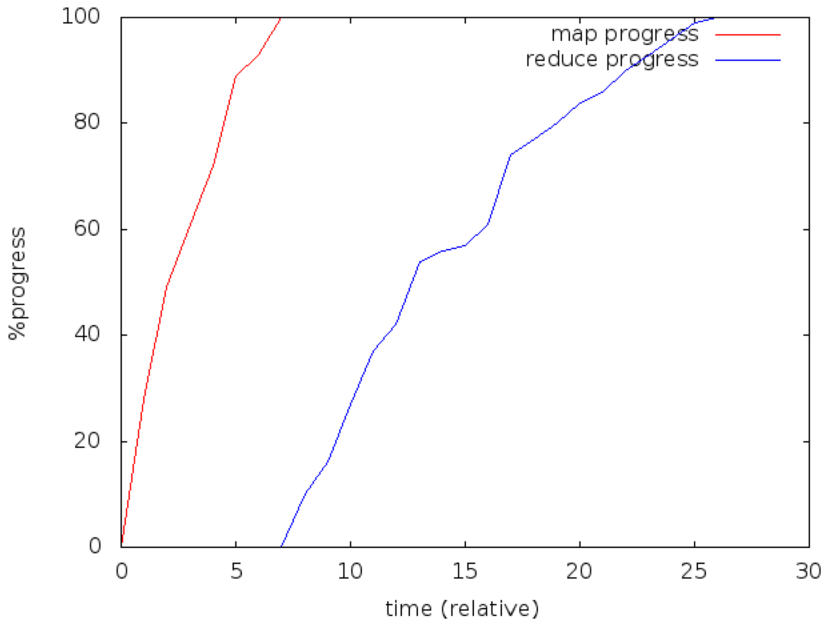


Figure 6.8: Map reduce progress for cluster of 3 machines

Chapter 7

Future Work

1. Wide range of machine learning algorithms can be applied for modelling and predicting the behaviour of the data. Particular application can be solved using a specific family of algorithms. The models can be built using well known machine learning packages (e.g Apache Mahout) available for using with big data platform. Some machine learning algorithms can be made parallel (e.g k-means) over a cluster where as some are inherently sequential in nature (e.g HMM). Studies/ experiments can be done to find out whether they can be made parallel or not. There would be some gain from parallelism as many computations are done on different processing units at a time. But at the same time there would be some amount of overhead (e.g initialization, data distribution, resource sharing and many more) as side effects of parallelism. Trade-offs between these two needs to be determined.

2. Other file systems can be used for the processing and experiments can be carried out to find the best suited setup. There are alternate solutions apart from Hadoop and Storm, like Map-Reduce on cloud operating system[24] instead Hadoop, Spark(Storm's replacement) and many more

3. Till now the implementation of model building is static. This can be made incremental by adding newer data if the error of prediction goes beyond certain threshold. The decision of the trade-off between tolerance and computation cost should be wisely taken.

Chapter 8

Conclusion

My project goal is to understand and propose some Big data analysis algorithm and streaming algorithms. This also includes a well mix of tools and techniques to build a scalable, real-time, distributed, fault tolerant system. The idea is not only to achieve the best solution to some specific application but also find out a modular and general purpose solution for the set of similar applications.

The implementation includes two distributed/real time solution to some interesting problems out there. The big data giant Hadoop and Stream processing system Storm was thoroughly explored and understood. The performance in terms of accuracy and scalability both investigated. It gave satisfactory results for the Cash demand forecasting.

The huge boost of available data is forcing the conventional enterprises to adapt more and more big data solution. So it is high time to perform experiment and infer various interesting facts about the same.

Bibliography

[1] *Apache hadoop*, http://hadoop.apache.org/.

[2] *Apache hbase*, http://hbase.apache.org/.

[3] *Apache kafka - a high-throughput distributed messaging system*, http://kafka.apache.org/.

[4] *Apache zookeeper*, http://zookeeper.apache.org/.

[5] *Beymani project*, https://github.com/pranab/beymani.

[6] *Coefficient of determination*, https://en.wikipedia.org/wiki/Coefficient_of_determination.

[7] *How to use data streaming for big data*, http://www.dummies.com/how-to/content/how-to-use-data-streaming-for-big-data.html.

[8] *Nn5 motivation*, http://www.neural-forecasting-competition.com/NN5/motivation.htm.

[9] *Real-time big data analytics - streaming data— cloud::streams — infochimps*, http://www.infochimps.com/infochimps-cloud/cloud-services/cloud-streams/.

[10] *Storm, distributed and fault-tolerant realtime computation*, https://storm.incubator.apache.org.

[11] *Storm spring*, https://github.com/granthenke/storm-spring.

[12] *Stream processing - microsoft research*, http://research.microsoft.com/en-us/projects/streams/.

[13] *Virtualized hadoop performance with vmware vsphere 5.1*, http://www.vmware.com/files/pdf/vmware-virtualizing-apache-hadoop.pdf.

[14] Shamik Sural Arun K. Majumdar Abhinav Srivastava, Amlan Kundu, *Credit card fraud detection using hidden markov model*, IEEE Transactions on Dependable and Secure Computing, Vol. 5,No. 1, January-March 2008, pp. 37–48.

[15] Rao B Aleskerov E, Freisleben B., *Cardwatch: a neural network based database mining system for credit card fraud detection*, IEEE Transactions on Dependable and Secure Computing, Vol. 5,No. 1, January-March 2008, pp. 37–48.

[16] El-Shishiny Hisham Andrawis Robert R., Atiya Amir F., *Forecast combinations of computational intelligence and linear models for the nn5 time series forecasting competition*, International Journal of Forecasting 27, July 2011, pp. 672,688.

[17] Shehzad H. Shaikh Akash B. Koli Ashphak P. Khan, Vinod S. Mahajan, *Credit card fraud detection system through observation probability using hidden markov model*, International Journal of Thesis Projects and Dissertations (IJTPD), Vol. 1, Issue 1, October-December 2013, pp. 1,10.

[18] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom, *Models and issues in data stream systems*, Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (New York, NY, USA), PODS '02, ACM, 2002, pp. 1–16.

[19] Y. Dora Cai, David Clutter, Greg Pape, Jiawei Han, Michael Welge, and Loretta Auvil, *Maids: Mining alarming incidents from data streams*, Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '04, ACM, 2004, pp. 919–920.

[20] Cudre-Mauroux P. Grund M. Perroud B. Chardonnens, T., *Big data analytics on high velocity streams: a case study*, Big Data, 2013 IEEE International Conference on (Silicon Valley, CA), IEEE, 6-9 Oct. 2013, pp. 784 – 787.

[21] Manidipa Mitra Dibyendu Bhattacharya, *Analytics on big fast data using a realtime stream data processing architecture*, EMC2.

[22] Javad Hosseinkhani Hossein Moradi Koupaie, Suhaimi Ibrahim, *Outlier detection in stream data by clustering method*, International Journal of Advanced Computer Science and Information Technology (IJACSIT), Vol. 2, No. 3, 2013, pp. 25–34.

[23] Jeff Ullman Jure Leskovec, Anand Rajaraman, *Mining of massive datasets.*

[24] Huan Liu and Dan Orban, *Cloud mapreduce: A mapreduce implementation on top of a cloud operating system*, Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Washington, DC, USA), CCGRID '11, IEEE Computer Society, 2011, pp. 464–474.

[25] Ounis I. Osborne M. Petrovic S. McCreadie R., Macdonald C., *Scalable distributed event detection for twitter*, Big Data, 2013 IEEE International Conference on (Silicon Valley, CA), IEEE, 6-9 Oct. 2013, pp. 543 – 549.

[26] Saša Petrović, Miles Osborne, and Victor Lavrenko, *Streaming first story detection with application to twitter*, Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Stroudsburg, PA, USA), HLT '10, Association for Computational Linguistics, 2010, pp. 181–189.

[27] Milena Kresoja Marina Mareta Hermann Mena Mladen Nikoli Tijana Stojanevi Piotr Broda, Tijana Levajkovi , *Optimization of atms filling-in with cash*, March 2014.

[28] O. S. Adewale J. O. Ayeni G. A. Aderounmu W. O. Ismaila S. O. Falaki, B. K. Alese, *Probabilistic credit card fraud detection system in online transactions*, International Journal of Software Engineering and Its Applications, Vol. 6, No. 4, October, 2012, pp. 69–78.

[29] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik, *The 8 requirements of real-time stream processing*, SIGMOD Rec. **34** (2005), no. 4, 42–47.