University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Computer Science	Spring semester, 2013.... Open / Restricted access
Writer: Bikash Agrawal	…… (Writer's signature)
Faculty supervisor: Tomasz Wiktor Wlodarczyk Chunming Rong	
Thesis title: **Analysis of large time-series data in OpenTSDB**	
Credits (ECTS): 30	
Key words: Hadoop, MapReduce, HBase, OpenTSDB, R, RHIPE, R2Time, Time-series	Pages: ...67… + enclosure: program on CD Stavanger, …1/07/2013.. Date/year

Front page for master thesis
Faculty of Science and Technology
Decision made by the Dean October 30th 2009

Analysis of large time-series data in OpenTSDB

Bikash Agrawal

Faculty of Science and Technology
University of Stavanger

July 2013

Abstract

In recent years, the quantity of time series data generated in a wide variety of domains have grown consistently. Analyzing time-series datasets at a massive scale is one of the biggest challenges that data scientists are facing.

This thesis focuses on implementation of a tool for analyzing large time-series data. It describes a way to analyze the data stored by OpenTSDB. OpenTSDB is an open source distributed and scalable time series database. It has become a challenge for statisticians and data scientists to analyze such massive data sets with the same level of comprehensive details as is possible for smaller analyses.

Currently tools available for time-series analysis are time and memory consuming. Moreover, no single tool exists that specializes on providing an efficient implementations of analyzing time-series data through MapReduce programming model at massive scale. For these reason, we have designed an efficient and distributed computing framework - R2Time. R2Time integrates R open source project for statistical computing and visualization with the OpenTSDB [1] and RHIPE [2] based on the MapReduce framework for the distributed processing of large data sets across a cluster. It creates the programming environment by integrating R and HBase for the data scientists.

This thesis describes the architecture of R2Time framework. The usefulness of this framework is verified by the performance analysis based on carefully choosen types of statistical analysis for time-series data. With the increase in the time-series data size and complexity of statistical functions, we have noticed supralinear nature in the performance of R2Time framework. The performance of this framework is verified by the performance analysis based on different configurations setting. Configuration settings as scan cache and batch size plays vital role with the performances of time-series data.

Acknowledgements

I would like to thank Prof. Chunming Rong and Dr. Tomasz Wiktor Wlodarczyk, my supervisors for their valuable advises and contributions. My deepest gratitude goes to Dr. Tomasz Wiktor Wlodarczyk, for his relentless support and insightful comments. He was always available whenever, I needed help.

I would like to extend my sincere thanks to Antorweep Chakravorty, PhD candidate. for his support. The thesis would have never been possible without his help. I would like to thanks Dr. Dehua Chen for providing useful feedback.

I would like to thank my friends Santosh Maharjan, Stephen Michael Jothen and Prakash Thapa for their inspirations and help. I would also like to thanks Dr. Saptarishi Guha (Author of RHIPE) for providing answers to my questions.

Last but not the least, I would like to thank my family and friends in Norway and Nepal for making my project successful.

Bikash Agrawal
University of Stavanger

Preface

This thesis is submitted in partial fulfillment of the requirements to complete the Master of Science (M.Sc.) degree at the Department of Electrical and Computer Engineering at the University of Stavanger (UiS), Stavanger Norway.

The work has been done at the department of Computer Science under the supervision of Dr. Tomasz Wiktor Wlodarczyk and Prof. Chunming Rong. It contains work done from February to June 2013.

The paper of this thesis is submitted in the Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2013) .

The study is on statistical analysis for large time-series dataset in OpenTSDB. An attempt is made to implement a framework to analysis large time-series dataset. There exists many time-series analysis tools. None of them seems to work in an efficient and scalable way. We proposed a solution with MapReduce to analyze time-series data with an existing statistical tool like R.

This thesis might be helpful for those who think of analyzing large time-series data. It might pave a solid ground for the students, data scientists and statisticians who think of working with data analysis and visualization. For rest who are interested in big data, might find it interesting to know the implementation of HBase with R via MapReduce.

July 1, 2013
Bikash Agrawal

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Time series occur extensively in real world application domain, as diverse as epidemiology, biostatistics, engineering, and computer science. A time series is a sequence of data points measured at regular time intervals (e.g. every 1 seconds) [3]. Time series may be defined as a high-dimensional data instance where dimensionality equals to the number of observations. Huge time series databases are widely available and as computing becomes faster and more reliable, analysis tools are receiving increased attention. The high dimensionality time-series data poses many challenges for effective analysis and visualization. In order to increase measurement accuracy the sampling period of time-series plays an important role.

The data coming from sensors, power grid, stock exchange, social networking or cloud monitoring (logging) system is stored and processed in the cloud environment. The sequential, monotonously increasing nature of time series data has became a big challenge for data analytics to analyze. There should be an efficient way to process a tremendous amount of complex information in a short time. There is a need for a solution that is able to analyze data at different level of details while simultaneously collecting thousand metrics from various physical or logical sensors. It has become a challenge to data scientists to analyze such massive data sets with the same level of comprehensive details as is possible for smaller analyses.

It is time and memory consuming to deal directly with high-dimensional time-series data. Many techniques have been proposed for representing time series with reduced dimensionality: Discrete Fourier Transformation (DFT) [4], Singular Value Decomposition (SVD) [4], Discrete Wavelet Transf.(DWT) [5], Piecewise Aggregate Approximation (PAA) [6], Symbolic Aggregate approX.(SAX) [7] etc. There is a lack of tools that can analyze time-series data in an efficient and scalable way. With the development of MapReduce [8](and related techniques), it became easier to analyze large amounts of

data in a distributed environment.

R is a programming language and a software suite used for data analysis, statistical computing and data visualization that is used by companies like Google and Facebook. There are more than 4188 packages available for R, which makes R a powerful for statistical analysis. R and Hadoop can complement each other very well. They are considered to be natural match in big data analytics and visualization. There are frameworks like RHadoop and RHIPE(R and Hadoop Integrated Processing Environment) that integrate with R to analyze data within MapReduce workflows.

OpenTSDB is an open source distributed and scalable time series database used for storage and indexing of time-series metrics that works on top of HBase [1]. HBase is an open-source distributed database that runs on Hadoop [9]. OpenTSDB provides basic statistical functionalities like mean, sum, maximum and minimum. To perform advanced statistical analysis for high-dimensional time-series data, we need good statistical tool, which is R as suggested by most of statistician and mathematician. There is a lack of research about connecting R with OpenTSDB (time-series distributed database). This thesis focuses on the development of a framework (R2Time) that connect R with OpenTSDB(time-series distributed database) for statistical analysis.

There exist R connectors like rbase and rhbase which can connect with HBase. However, this connectors provide basic operations like getting and putting data in HBase. OpenTSDB or time-series database have special key format that is used to store data. R connectors, like rbase and rhbase cannot be used easily with time-series database because of the key format.

Being motivated by these observations we have designed a R2Time- framework for distributed time-series data analysis and visualization. R2Time provides R users to interact directly with time-series databases and to perform statistical analysis using MapReduce programming model.

R2Time framework supports efficient computation of typical statistical analysis for time-series data. It allows users to query large time-series data directly from HBase via MapReduce programming model. R2Time uses of distributed computing model for statistical analysis. We verify an efficiency of R2Time by applying statistical analysis of real-world datasets: trend of data, seasonal variation, systematic pattern and random noise, cyclical and irregular variation.

1.1 Related work

Because of importance and usefulness of time series data, there are large numbers of applications that deal with time series, based on different approaches. Most of these approaches are traditional approaches which require huge memory and are time consuming.

There are several tools specialized for analysis of time series. R is a statistics software that has extensive features for analyzing time series data. In the Open Source community, there are two popular tools: opentsdbr [10] and StatsD OpenTSDB publisher backend [11]. Both tools uses OpenTSDB HTTP/JSON API to query data from OpenTSDB. This API is only useful for small scale analysis due to its non distributed implementation that creates performance bottlenecks for real world applications. It requires huge memory to store time-series data at client side. Moreover, it is time consuming due to transfering of data through network interface. For visual analysis, both systems use third party packages in R for displaying high dimensional time series data.

Some of the most common time-series analysis tools are: GRETL (GNU Regression, Econometrics and Time-series Library) [12], TimeSearcher [13], Calendar-Based Visualisation [14] and Spiral [15] etc., but they are not specialised for real-world time-series analysis. These tools are not designed to work with distributed programming model. These tools works with single node, so the tasks are not distributed. If users want to do statistical analysis on massive amounts of data using these tools, it will take couple of days.

There are many packages in R [16] available for time-series analysis such as astsa, tsa3, tseries, tempr etc. [17] but none of them support distributed programming model to analyze data. These tools download data at client side and then perform statistical analysis which makes it time and memory consuming. There are a few R packages like rbase and rhbase available that work on HBase. R packages like rbase [18] and rhbase [19] provide basic connectivity to HBase. To run MapReduce on rbase data need to be read from HBase and stored in HDFS which is time and memory consuming.

RHIPE [2] provides possibility to analyze simple time-series data that is stored in HDFS. But the problem is it can not read data from HBase directly. RHIPE can read data from HBase with a help of third library rbase, developed by the same author (Dr. Saptarishi Guha). Still time-series data stored by OpenTSDB can not be read by rbase because of composite row key concept maintained by time-series data storage [20].

Our time series data analysis differs from related work in two main points. Firstly,

it allows users to query time-series data stored in HBase directly using composite key of OpenTSDB via MapReduce programming model. Secondly, it allows users to perform advanced statistical analysis such as trend of data, seasonal variation, systematic pattern and random noise, cyclical and irregular variation, using MapReduce programming model. This provides an efficient and scalable way to analyse high dimensional time-series data.

In this thesis, a thorough study has been done on performance analysis of time-series data at massive scale using R2Time. It presents the strengths and weaknesses of the approach. Different performance tests were performed. Therefore, we hope that the insights and experimental results presented in this thesis will be useful for the future development of time-series analysis at a massive scale. A paper on this thesis is submitted in the IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2013) under Big Data track.

1.2 Organization of the Thesis

The thesis is organized in the following way

Chapter 2 presents basic background theory needed to understand this thesis.

Chapter 3 has a detailed discussion on the approaches needed to build frameworks to analyze time-series data.

Chapter 4 describes the experiment setup for verification of the result.

Chapter 5 discusses about the results obtained from the R2Time framework. It also discusses about some performance analysis tests. Detail explanations on each result can be obtained in this chapter.

Chapter 6 concludes the thesis, also indicating some future enhancements.

CHAPTER 2

Background

2.1 Hadoop

Hadoop is an open source programming model that provides both distributed storage and computational capabilities across cluster of computers using simple programming model [21][22][8]. Being developed mainly by Yahoo, it is now a Apache project. It is mostly inspired by Google published paper [23][24] that describes it's novel distributed filesystem, the Google File System (GFS), and MapReduce. Hadoop is currently used by big companies like Yahoo, Facebook, Cloudera and Amazon.

Hadoop is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than relying on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer.

Hadoop platform generally consist of

- Hadoop common
- MapReduce and
- Hadoop Distributed File System (HDFS)

2.2 Hadoop Distributed File System(HDFS)

HDFS is a distributed, scalable, and portable file system written in Java for the Hadoop framework. It is a model based on Google File System(GFS) paper [23][24]. A distributed file system (DFS) is the storage of a computer in a cluster of one unified file

system. When file is stored on the DFS, it is partitioned into blocks and each block is replicated across the clusters. It provides a measure of fault-tolerance.

HDFS is designed to store big data in terabytes and stream them in an efficient way. It is inspired by the Google file system (GFS). Each node in a Hadoop has a single namenode and a cluster of datanodes to form the HDFS cluster. A Hadoop cluster will include master and multiple worker nodes. The master node consists of a JobTracker, TaskTracker, NameNode and DataNode. Clients use RPC to communicate each other. HDFS stores large files (ideally 64MB), across the clusters. It mainly separates file's metadata and application data. Metadata is stored in master(NameNode) and application data is store in workers(DataNode).

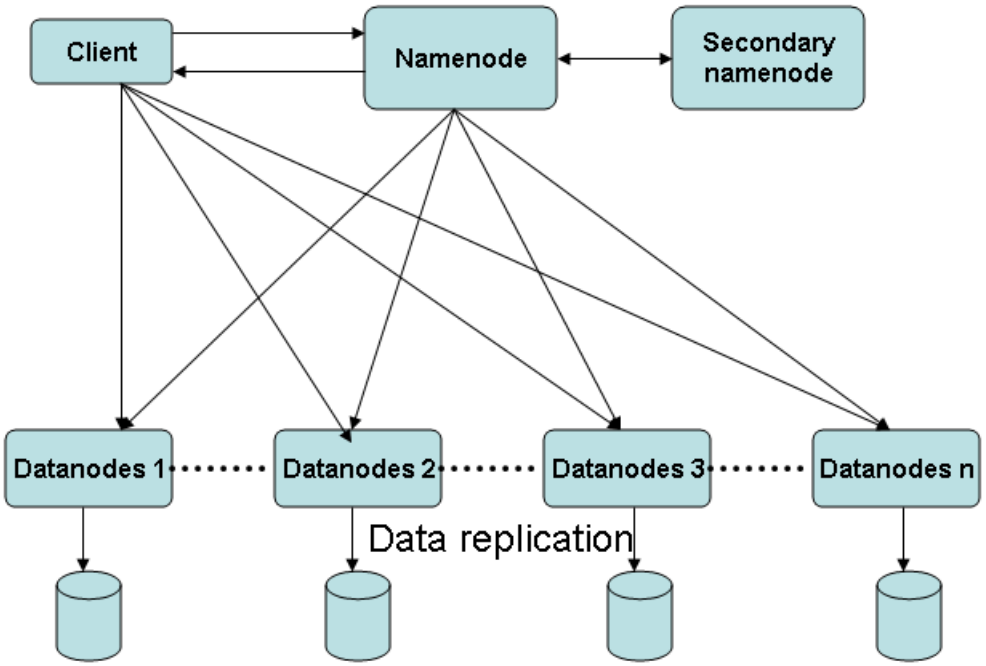

Figure 2.1: Overview of the HDFS Architecture [25].

- **NameNode**:
  The NameNode is the master node. The NameNode is responsible for managing the filesystem namespace tree and map the location of all blocks to the DataNodes. When a client wants to read a file located in the system, it first contacts NameNode for the location of data block comprising the file and then read block contents from the DataNode closer to the client. Similarly, when the client wants to write data, it first queries NameNode to nominate a suite of three DataNodes to host block replicas. The client then writes data to the DataNodes in a pipeline fashion. The replication number three is by default. It can be changed to any numbers. For HBase NameNode is also know as Region Server.

- **DataNode**:
  DataNodes are the workers node. They store real application data. They have to periodically send report the NameNode about the information on which blocks they are storing the data. So, that NameNode can update meta data.
- **Secondary NameNode**:
  The Secondary NameNode is an assistant to NameNode for monitoring the state of the cluster's file system. Its main task is to take snapshots of the HDFS meta-data from the NameNode memory structures. By doing this, it helps in preventing file system corruption and reducing loss of data, and thus overcomes failures.
- **Job Tracker:**
  Job tracker accepts jobs from client and submit the jobs in clusters. It also helps to pipleline and distribute job across the cluster.
- **TaskTracker:**
  It maintains map and reduces task in DataNode.

2.3 Map and Reduce

MapReduce is parallel programming model introduced by Google in 2004 that processes very large data on clusters of computers. It has proven to be very attractive for parallel processing of arbitrary data [26]. It consists of two user-defined functions, Map and Reduce. The Map function is responsible for processing sub-data sets and produces intermediate results, and the Reduce function is responsible for reduction of the intermediate results and generates the final results of processing. MapReduce manages scheduling of the task across clusters, fault-tolerance, splitting the input data and managing communication between nodes.

```
1 Map Function:    map(k1,v1) => list(k2,v2)
2 Reduce Function: reduce (k2,list(v2))  =>  list (k3,v3)
3 Map function get input key/value pairs and output is new key/value (k2/v2) pairs. This new key/
      value pairs is input to reduce function, reduce function produce new key/value pair as ouput.
```

Example 2.1: Pseudocode for MapReduce.

MapReduce is similar to SQL statement "Select * from table group by id", where select * from table is map function and group by id can be similar to reduce function. A MapReduce program usually takes a set of key/value pairs as input and produce a set of key/value pairs as output. It splits input file into independent fixed-sized chunks

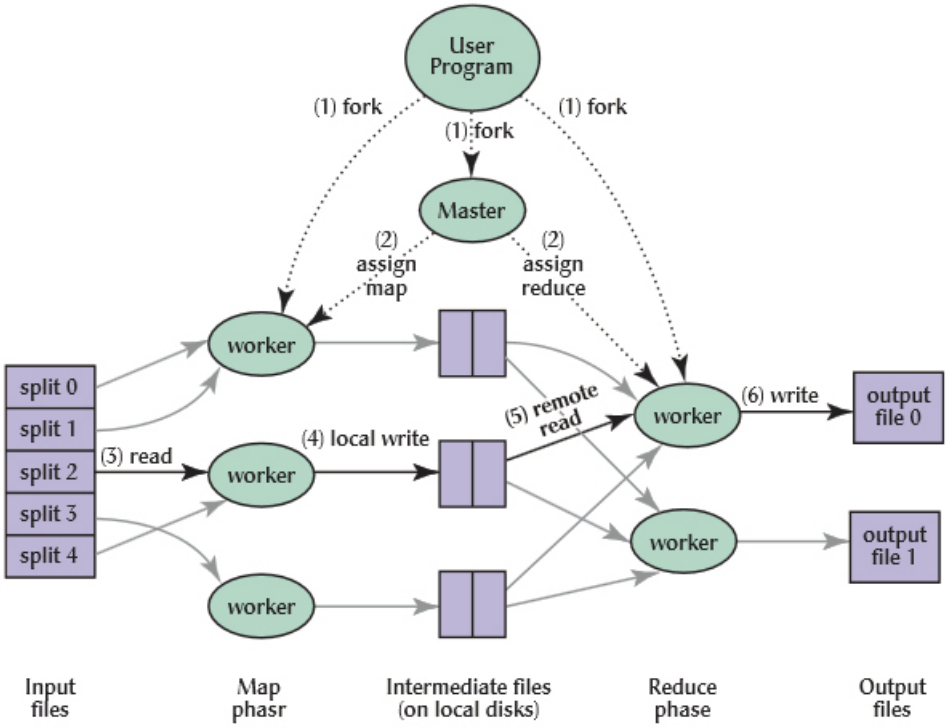

Figure 2.2: MapReduce execution overview [27].

called input splits. A set of key/value pairs is read from each chunks of data. Each chunks of data are processed by the map task in parallel manner. A map function is user defined function, which reads key/value pairs from input file. A map function generates another intermediate key/value pairs. This intermediate key/value pairs (output of map function) is written to a local disk. The values obtained are associated with the intermediate key and are grouped together by MapReduce and are fed into the reduce function. The reduce function then merges the values and finally outputs a set of key/value pairs as shown in figure 2.2. The sample code snapshot for MapReduce is explained in example 2.1.

```
1 function map(int key, float values[]):
2     key: filename
3     values: list of values in file
4   for each word w in document:
5     emit (1, mean(values))
6
7 function reduce(int key, float values[]):
8   emit (1, mean(values))
```

Example 2.2: Pseudocode for MapReduce arithmetic mean calculation.

For example, consider calculation of mean value from two file using MapReduce program. File1 contains list of values as "10,20,30,20,20,20", similarly file2 contains list

of values as "10,20,30,20,20". Mean is calculated for these values using MapReduce? A Map function will output <1,mean(values)>, where 1 is key and values is list of value in map function. Considering two file as two chunk of data, 2 Map task is executed. On each map task, user calculates mean and store in local disk. Output of maptask is:
<1,mean(10,20,30,20,20,20)> =><1,20>
<1,mean(10,20,30,20,20)> =><1,50>

All these ones/values pairs are then an input to the reducer function. Reducer function is responsible to calculate final mean of all input values. So final output is as:
<1,mean(20,50)> =><1,35>
Thus, mean calculated using MapReduce is 35. MapReduce user defined function is shown in example 2.2.

2.4 HBase

HBase is a distributed column-oriented and NoSQL database built on top of HDFS. HBase is the Hadoop application to use when we requires real-time read/write random-access to very large datasets[9]. It is modelled after Google's BigTable [28]. In HBase, a table is physically divided into many regions, which are in turn served by different RegionServers. One of the biggest utility is the one that is able to combine real-time HBase queries with batch MapReduce Hadoop jobs, using HDFS as a shared storage platform. HBase is extensively used by big companies like Facebook [29], FireFox and others.

HBase can be efficient to use if we have millions or billions of rows. All rows in HBase are always sorted lexicographically by their row key. In lexicographical sorting, each key is compared on a binary level, byte by byte, from left to right. HBase provides java API for client interaction. Using java API, user can create HTable instance. Creating HBase table instance is time consuming. Because each instance of HTable involves scanning the .META. table to check if the table actually exists and is enabled. So, it is always better to reuse the HTable instance and close the HTable instance after completion of the task. The .META. and -ROOT- tables are internal system tables. The -ROOT- table keeps list of all regions in the .META. table whereas the .META. table keeps list of all regions in the system.

Schema	Row	Family:Column	Version
2D	metricID - timestamp	varying properties	current timestamp
3D	metricID	varying properties	timestamps

Table 2.1: Different type of schema to store time-series data in HBase [20].

2.4.1 Schema in HBase

Data stored in HBase is grouped into tables. Conceptually, a table is a collection of rows and columns. Each row in HBase data has a unique row key and multiple column keys. The values are associated with column keys. Client can create an arbitrary number of columns using new column qualifier on the fly. Columns in HBase is the combination of the column family name and the column qualifier (column key), separated by colon: *column family:column qualifier*

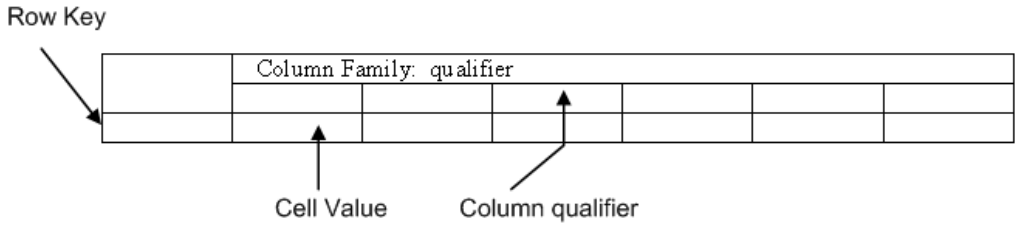

Figure 2.3: HBase table schema.

Column family name must be printable characters. The column qualifier can be composed of arbitrary binary characters. The reason why column family name must be character is: it is used as directory name by the lower-level storage layer. User can also have empty column qualifier. That is why it is called a *column-family-oriented store.*

Data can be organized in different ways in HBase. HBase is more like a multi-dimensional sorted map:*(Row Key, Family:Column, Timestamp)* value as shown in figure 2.4. It can be organized as 3D format *(Row, family:column, version)* which can be denoted by (X,Y,Z) of the cube in figure 2.5. It can be designed as 2D format *(Row, family:column)* which can be represented by (X,Y) axis of rectangle as show in figure 2.5 and table 2.1 [20]. With 3D model for same column qualifier, it can store multiple values. This values is identified uniquely by timestamp. For one row and column, there are multiple timestamp.

```
Row Key: Key1 {
        Column Family T {
                Column X:
                        Timestamp1 Valuel
                            Timestamp2 Value2
                            Timestamp3 Value3
                Column Y:
                        Timestamp4 Value4
                }
    Column Family P {
                Column Z:
                        Timestamp5 Value5
                }
        }
```

Figure 2.4: HBase data schema

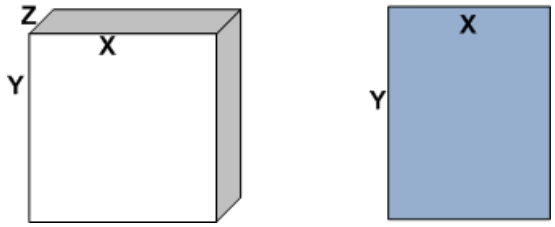

Figure 2.5: Data organization in HBase.

2.4.2 Version

Versions in HBase specify a number of versions for each value, a user can keep. By default maximum version value is 3. User can change this number. HBase also provides java API to change this value. HBase actually stores key-values written into columns. User can specify many versions of value to be stored. Version is referred to number of times value can be stored for particular row and column. This row and columns is distinguish by timestamp.

2.4.3 HBase Region Splitting

HBase achieves load balancing by evenly splitting across the number of regions. Tables are splitted into chunks of data rows called "regions". A region is a continuous range within the row key. All rows in the table are sorted between the region's start key and end key. Regions are unique and non-overlapping. That means particular row key

belongs to one region. Together with -ROOT-, .META. regions, and a table's region provide 3 levels of B-Tree for the purpose of searching row within a table as shown in figure 3.2. A table contains many regions which are hosted in many region servers. Using HBase API, user can create split of huge data, but the splitting of data depends on the number of region servers. For example, if we have 1000 record and 10 region servers, number of records on each region server is 100.

When a table is first created, HBase allocates only one region for the table. This causes all the request to go to a single region server, regardless of the number of region servers. As the data size goes on increasing, splitting is done by default. That is why it is not good to use HBase with small data. It cannot utilize the whole capacity of the cluster.

There are 3 type of splitting done:

- Pre-splitting
- Auto splitting
- Forced Splits

2.4.4 Pre-splitting

By default, HBase creates only one region for the table, it is possible that HBase may not know how to create the split points within the row key space. HBase provides client API/tools to manage splitting. The process of supplying splits point at the time of table creation to ensure initial load is evenly distributed throughout the cluster, is defined as *pre-splitting*. Pre-splitting is better if we know key distribution before hand. Pre-splitting also has a risk of creating regions that do not truly distribute the load evenly because of data skew or because of very large rows. Sometimes, choosing of splitting point is poor, which can end up with heterogeneous load distribution and poor cluster performance.

Major issue with pre-splitting is: choosing the split points for the table. There is a class call RegionSplitter that creates the split points, by using a SplitAlgorithm. For optimal load distribution, user should think about data model and key distribution. User can start loading data into the table using pre-splitting (defining lower multiple of the number of region servers as number of splits) and later on automate splitting to handle the rest.

2.4.5 Auto splitting

It is generally recommended to have auto splitting on. Auto splitting depends on configuration of HBase. We can configure HBase for making decisions of splitting a region.

The default split policy for HBase 0.94 is IncreasingToUpperBoundRegionSplitPolicy. Splitting is done based on the number of regions hosted in the same region server. "This split policy uses the maximum store file size based on
Min $(R^2 * "hbase.hregion.memstore.flush.size", "hbase.hregion.max.filesize")$,
where R is the number of regions of the same table hosted on the same regionserver." [30] For example, `hbase.hregion.memstore.flush.size` is 128 MB and the `hbase.hregion.max.filesize` is 10GB. The first region in the region server will be splitted just after the first flush at 128 MB. As number of regions in the region server increases, it will use the increasing split sizes: 512MB, 1152MB, 2GB, 3.2GB, 4.6GB, 6.2GB and so on until the split size reaches almost 10 GB. No further increase on split size takes place.

We can configure split policy in HBASE by configuring `hbase.regionserver.region.split.policy`. There are many splitting policies available in HBASE. They are : ConstantSizeRegionSplitPolicy, IncreasingToUpperBoundRegionSplitPolicy, and KeyPrefixRegionSplitPolicy. We can also implement our own custom split policy too.

2.4.6 Forced splits

HBase also enables clients to force split a table from the client side. Users needs to supply a split point to do forced split, similar as pre-splitting. But difference between pre-splitting and forced splitting is: force split is done only after table creation, but pre-splitting is done when we are creating a table. Forced splitting is done, if the user found that HBase load distribution is uneven and some regions are getting uneven loads. User can use force splitting to balance and improve throughput. Sometimes, automated split is suboptiomal and under such condition, user can disable automated splits and can use manual split.

2.5 OpenTSDB

OpenTSDB is a distributed and scalable Time Series Database (TSDB), written on top of HBase. OpenTSDB was written to store, index, and serve metrics, collected from computer systems at a large scale. OpenTSDB also provides users to easily access massive data and provide visualization via graphicable representation.[1]. OpenTSDB is

developed by StumbleUpon. OpenTSDB allows us to collect thousands and thousands of metrics from thousands of hosts and applications, at a high rate (every 1 seconds). All the data is stored in HBase, and the simplified web user interface (WUI), enables users to query various metrics in a real time. OpenTSDB generally create two special tables in HBase: tsdb and tsdb-uid.

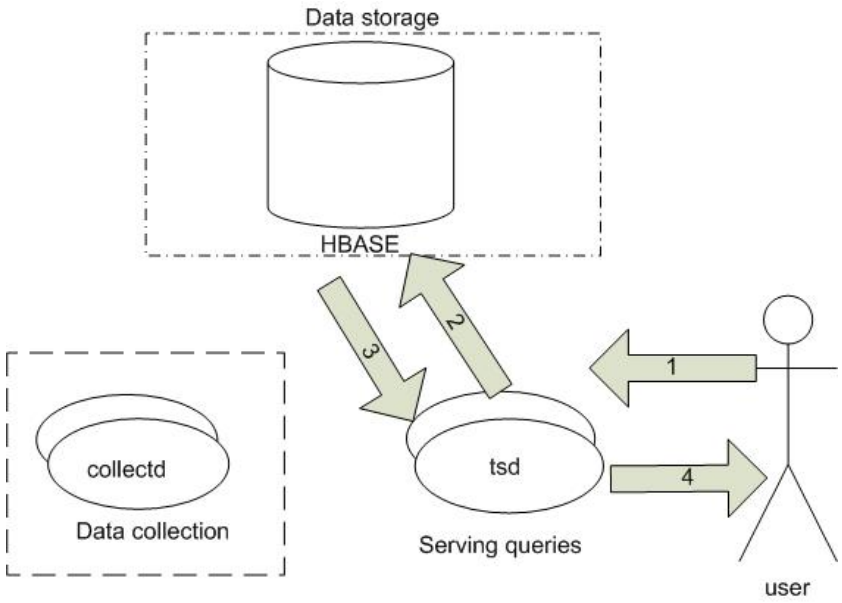

Figure 2.6: OpenTSDB architecture: The three areas of concern are data collection, data storage, and serving queries.User specify query parameters. TSD construct filter and reuest range scan, HBase scan key range and filter the tags and return result. tsd visualized data point [31].

OpenTSDB schema promotes the metric ID into the row key, forming the following structure:
<metric-id><base-timestamp>...

Since, a production system will have a considerable number of metrics and their IDs will be spread across a range, and all updates occurring across them, we end up with an access pattern akin to the salted prefix: the reads and writes are spread across the metric IDs. This approach is ideal for a system that queries primarily by the leading field of the composite key [31].

OpenTSDB goes further by "bucketing" values into rows using a base timestamp in the row key and offset timestamps in the column qualifiers, for more efficiency to form composite row key, so that data can be uniquely stored. OpenTSDB follows 2D time-series data storage schema as shown in figure 2.5. The tsdb-uid table is kind of meta-data. In tsdb table is a place where the actual time-series data is stored. When

trying to import huge amount of data in to HBase, it is better to have HBase table is pre-split.

OpenTSDB does not hit the disk for every query, it has its own query cache called Varnish [32]. OpenTSDB also uses HBase caching mechanism (called the Block Cache) which provides quick access to fetch data point. OpenTSDB provide Java API and HTTP API (JSON). OpenTSDB is built using Java library and Asynchronous HBase (an alternative Java library to interact HBase in applications that require a fully asynchronous, non-blocking, thread-safe, high-performance HBase API). OpenTSDB is better option to be used for time-series as mention on [33] [34].

2.5.1 OpenTSDB Schema

The row key is composite and combination of: the metric ID, a timestamp, the tags. Each tag is combination of tag key ID and tag value ID.
<metric-id><base-timestamp><tag-key-id><tag-value-id>.....

Row key is combination of metric-id is 3 byte, base-timestamp is 4 byte, tag key is 3 byte and tag value is 3 byte. The column qualifier is of the value 2 bytes. The first 12 bits are used to store an integer which is a delta in seconds from the timestamp in the row key, and the remaining 4 bits are flags. In 4 bit flag, the first bit indicates, the value is an integer value or a floating point value, the remaining 3 bits aren't really used at this time at version 1.0 but they are to be used for variable-length encoding in the future. The value in the cell is 8 bytes.

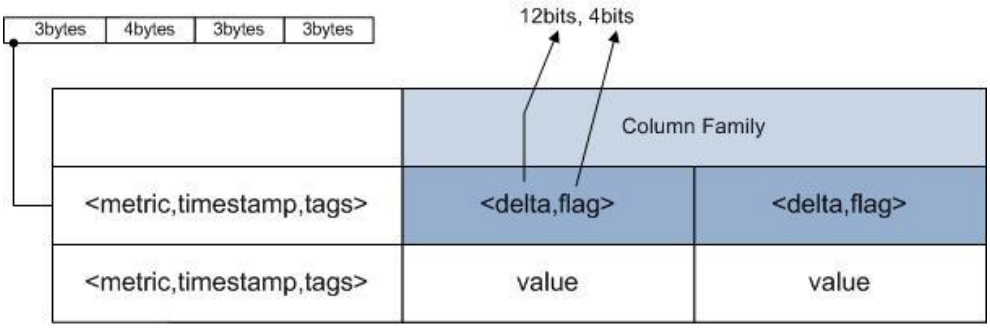

Figure 2.7: OpenTSDB Schema.

The problem with OpenTSDB is: it provides basic statistical functionality like average, maximum..etc. Advanced statistical analysis to reduce dimensionality time-series data is also needed.

2.6 Time Series

A time series is a collection of observations of well-defined data points obtained through repeated measurements over time. For example, collecting data from smart meter in every 15 minutes. Data are collected at regular intervals. [3]

Time series analysis is a method for analyzing time series data in order to extract meaningful statistics characteristics of the data.

2.7 R Statistical Computing

R is a language and environment for statistical computing and graphics [16]. It is a open source project which is similar to the S language and environment which was developed by John Chambers at Bell Laboratories. The current version of R is 3.0.1 and runs on several operating systems. R provides a wide range of statistical analysis (linear and nonlinear modelling, classical statistical tests, time-series analysis, clustering, etc) and graphical techniques.

For extra performance and interfacing to libraries in other languages, R provides an API for interacting with the R language from C. There is a package called rJava to interact R with Java. There are more than 4188 packages available like linear and nonlinear modelling, classical statistical tests, time-series analysis, clustering, etc. Nowadays, R is used by companies like Google and Facebook for data analysis, statistical computing and data visualization. R consists of several packages for parallel computing and time-series data analysis.

2.8 RHIPE

RHIPE, R and Hadoop Integrated Processing Environment is developed Dr. Saptarshi Guha. Rhipe development history begins from year 2009 and still is actively maintained by the author. The R and Hadoop Integrated Programming Environment is R package to compute massive data sets using the HDFS and MapReduce framework. This is accomplished from within the R environment, using standard R programming idioms.[35]

Using RHIPE, we just need little knowledge of hadoop map reduce. On R environment, users can do any type of statistical analysis. RHIPE consists of several functions

to interact with the HDFS, for e.g. save data sets, read data by MapReduce, delete files. RHIPE composes and launches MapReduce jobs from R using the command rhwatch. It monitors the status using rhstatus which returns an R object. RHIPE has three components: the R interface, the Java to R bridge and the engine written in C. The R interface has R functions that interact with HDFS.

RHIPE communicates with Hadoop via Java API. The serialization format used by RHIPE (converting R objects to binary data) uses Googles Protocol Buffers which is very fast and creates compact representations for R objects. The output of a RHIPE job is stored in HDFS.

2.8.1 rbase

This R package provides basic connectivity to HBase, using RHIPE. To run MapReduce on rbase, data need to be read from hbase and store in HDFS. But Rhipe 0.73 provides feature to send custom input format for MapReduce task.

This project provides the way to interact R with HBase which is helpful for development of R2Time framework. To store output of rhwatch()(MapReduce Task), we using rbase to store in HBase. In future, R2Time will have it's own method for storage too.

2.8.2 RhipeHbaseMozilla

This is an experimental project for Mozilla's socorro crash reporting done by Dr. Saptarshi Guha(author of RHIPE), specially designed for telemetry and crash−reports HBase table. This project is specially designed for mozilla crash reporting, not for time-series data. It fetches data from HBase and run MapReduce via RHIPE. The problem with this is, time-series have special composite row key combination. We need to design algorithm that can generate composite row key. Another problem with this is we need to specify column family to get data from HBase. In case of OpenTSDB column qualifier is unknown, it is generated automatically using bit operation.

This project provides the way to interact RHIPE with custom input format, which is useful for the development of R2Time.

CHAPTER 3

Design and Methodology

To interpret, analyze and visualize large time-series data, first it is necessary to understand how data is stored in HBase. Time-series data can be stored in different ways. HBase provides multi-dimensional storage as shown in the figure 2.5. Time-series data is stored with unique row key. Unique row key can be achieved by designing composite row key. Composite row key is similar to RDMS primary key. Composite row key consists of timestamp to uniquely distinguish each row. In this thesis, we have developed a framework, that helps to analyze time-series data. There already exists an open source framework RHIPE, which helps to do statistical analysis in R. But RHIPE works with HDFS, that is why it is needed to develop R2Time. R2Time has been developed, which acts as a bridge between RHIPE and OpenTSDB as shown in figure 3.1.

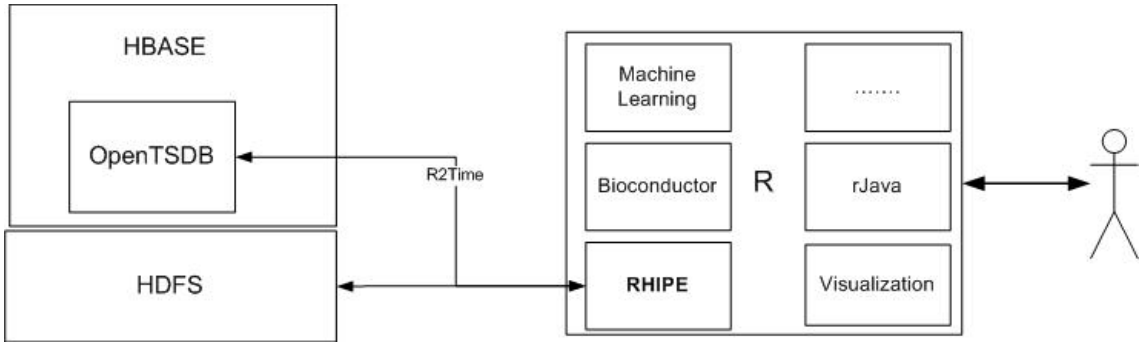

Figure 3.1: R2Time acts as a bridge between OpenTSDB and RHIPE.

Before, we discuss the design of OpenTSDB-RHIPE bridge, we have outlined OpenTSDB and RHIPE task.

3.1 Data structure in OpenTSDB

OpenTSDB is a distributed, scalable Time Series Database (TSDB) which is written on top of HBase [1]. OpenTSDB maintains two special tables in HBase, that is tsdb and tsdb-uid. OpenTSDB facilitates to collect many thousand and thousand of metrics from thousands of hosts and applications, at a high rate (every 1 seconds).

The TSDB table is the heart of the time-series database, that stores measurement of time series points. Tsdb-uid is a look-up table for metrics and tags. Tags mean anything we can use further for identify a measurement recorded.

3.1.1 Row key design in OpenTSDB

The sequentially, monotonically increasing nature of time-series data causes data to be written in same region. This causes *hotspotting* [9]. To avoid this issue, it is better to design composite row key as a row key that represents the event time.

OpenTSDB promotes the metric ID into the row key as shown in the figure below. <metric-id><base-timestamp><tags>....

Metric UID	base-timestamp	Tag Key	Tag Value	....
3 Bytes	4 Bytes	3 Bytes	3 Bytes	....

Table 3.1: OpenTSDB row key format [1].

OpenTSDB row key design consists of minimum 13 bytes with one tag. If there are many tags, it goes on appending at the end of the row key. Table tsdb-uid is look-up table for metric id, tag key and tag value. To design row key, it is needed to use tsdb-uid to get value for metrics and tags. Timestamps in the row key are rounded down to a 60 minute boundary, that means, for every 60 minutes, it creates new row key. The column qualifier is of 2 bytes. The first 12 bits are used to store an integer value which is delta (difference in time interval) in seconds from the timestamp in the rowkey, and the remaining 4 bits are flags. For example, the data point is at time 3456123232 but the timestamp in the row key is 3456123000, so the delta is 232. And out of the remaining 4 bits of flag, the first bit indicates whether the value is an integer or a floating point number, the remaining 3 bits are not used for version 1.0 of OpenTSDB, but they are used on the version 2.0 for variable-length encoding. Both an integer and floating point value can be encoded in variable-length. The value in the cell is the value of data point, which is of 8 bytes [1].

```
function constructRowKey(start_data, end_date, metric) {
Const.metricsize = 3 bytes
/* it is configurable, there exist configuration file in R2Time. */
Const.timestamp  = 4 bytes
/* it is configurable, there exist configuration file in R2Time. */
byte[] startRow  = new byte[Const.metricsize + Const.timestamp]
byte[] endRow    = new byte[Const.metricsize + Const.timestamp]
byte[] metricID = tsdb.metrics.getId(metric)
/*get metric value from tsdb-uid HBase table, using OpenTSDB java API. */
long starttimestamp = start_data.getTime() / 1000;
long endtimestamp = end_date.getTime() / 1000;
/* calculate timestamp from given date, return timestamp is in millisecond.*/
setInt(startRow, getScanStartTime(starttimestamp), metricsize)
setInt(endRow, getScanEndTime(endtimestamp), metricsize)
/* copy starttimestam/endtimestamp to 7 byte array, starting from 3 byte */
System.arraycopy(metricID, 0, startRow, 0, Const.metricsize);
/* copy  metric ID byte to  startRow byte array from index 0 to 3. */
System.arraycopy(metricID, 0, endRow, 0, Const.metricsize);
/* copy  metric ID byte to  endRow byte array from index 0 to 3. */
return startRow and endRow
}
```

Listing 1: Construct start rowkey and end rowkey from user input.

```
/*OpenTSDB maintains 60 minutes boundary. After 60 minutes it will generate
new row. MAX_TIMESPAN is 3600 seconds. Because it is the boundary limit
of the row. If the starttime=12:41:00. If we initialize the scanner to
look only 10 minutes before, we will start scanning at time = 12:31,
which will give us row starts at 12:40. But we need to start scanning
at least 1 row before, so we actually we are looking back by twice
 MAX_TIMESPAN.  */
function getScanStartTime(starttimestamp){
           long ts = starttimestamp - Const.MAX_TIMESPAN * 2;
}
/* Suppose endtime =12:40:00, we will stop scanning when we get to 12:50,
but one again we need to look ahead one more row, so to avoid this problem
we add 1 second extra to endtime */
function getScanEndTime(endtimestamp) {
                if(endtimestamp==null)
                        setEndTime(System.currentTimeMillis() / 1000);
                        return endtimestamp + Const.MAX_TIMESPAN + 1;
        }
```

Listing 2: Generate start and end timestamp [1].

The above pseudocode listing 1 is used to create 7 byte (<metricID+timestamp>)

start and end row key. It takes input from the user: start-date, end-date and metric. OpenTSDB have fixed format of composite row key as show in figure 3.1. R2Time provides configuration file, where all these constants are defined.

```
/*To find the rows with the relevant tags, we use a server-side filter
that matches a regular expression on the row key.
This function create 6 byte tag filter.  */
/* Generate a regexp for tags. Say we have 2 tags: { 0 0 1 0 0 2 }
   and { 002 004 }, the regexp will be:
   "^.{7}(?:.{6})*\\Q\000\000\001\000\000\002\\E(?:.{6})*
   \\Q\000\000\002\000\000\004\\E(?:.{6})*" */

function createfilter(){
Initialized tagsize =  Const.tagvalue_size+ Const.tagkey_size
/* tag value size + tag key size i.e 3+3 for OpenTSDB */
StringBuilder filter = new StringBuilder(15+((13 + tagsize)
                                * (tags_width + (tags.size() * 3))));
/* allocate number of bit to create filter */

filter.append("(?s)" + "^.{").append(7).append("}");
/* start by skping metric ID and timestamp i.e 7 bytes */

/* build regular expression for each tags, bytes operation to
get bytes string for tags. */
if (isTagNext(tag)) {
        for ( byte by : tag) {
                        filter.append((char) (by & 0xFF));
                        if (by == 'E' && backslash) { .
                                filter.append("\\\\E\\Q");
                        } else {
                                backslash = by == '\\';
                        }
                }
                filter.append("\\E");
        if(taghasnext)?tags = tags.next(): null;
}
/* Skip any number of tags before the end. */
filter.append("(?:.{").append(tagsize).append("})$*$");
return filter;
}
```

Listing 3: Creating Tags filter.

MetricUID can be found by looking at HBase tsdb-uid table, which creates first 3 byte of row key. The next 4 bytes represents base timestamp, as mentioned by

OpenTSDB format in the figure 3.1. From users input start-date, timestamp of that date is calculated as shown in listing 1. OpenTSDB maintains 60 minutes boundary. After 60 minutes, new row are generated. For creating start row key from start timestamp, we need to scan one step previous record. For example, if start-date is 12:40:00 and if we initialize the scanner to look only 10 minutes before, it will start scanning from 12:31. This will give row-start at 12:40. To initialize scanner, we need at least 1 row before, that is why we are looking back by twice MAXTIMESTAMP. MAXTIMESTAMP in OpenTSDB is 3600 seconds. Similarly for end row key, scanner is needed to set ahead one more row as shown in the listing 2. If user end date is 12:20:00, we need to stop when the scanner reaches 12:30:00. So, the scanner needs to set ahead of one row [36].

As mentioned earlier, we have already created 7 bytes start row key and end row key. From the OpenTSDB composite row key format shown in figure 3.1, still row key consist of tags. If there are multiple tags present, row key keeps on increasing by 6 bytes for each tags as shown below:
<tag-key1-UID><tag-value1-UID><tag-key2-UID><tag-value2-UID>.....
HBase provide regular expression filter API for row key as shown in listing 3. At first we need to skip first 7 bytes of row key, because first 7 bytes consist of metric and timestamp. We need to create 6 bytes of regular expression for each tag. All the characters between the \Q and the \E are interpreted as literal characters. Example \Q*\bikas+*\E matches the literal text *\bikas+*. The \E may be omitted at the end of the regex, so \Q*\bikas+* is the same as \Q*\bikas+*\E [37]. For example we have 2 tags :0 0 1 0 0 2 and 0 0 2 0 0 4, the regexp will be as \Q\000\000\001\000\ 000\002\E and \Q\000\000\002\000 \000\004\E [36].

3.2 Finding Row in HBase

HBase maintains two special tables, -ROOT- and .META., which help to find the regions where various tables are hosted. -ROOT- does not split into more than one regions, but .META. behaves like all other tables, that can be split into many regions as required, to avoid hot-spotting.

When client wants to access a particular row, it goes to the -ROOT- table and asks for the region that is responsible for that particular row. -ROOT- points it to the region of the .META. table. The .META. table keeps a list of all regions in the system. The .META. table holds keys and values. Key consists of name of table, region start key and region id. Values consist of HRegionInfo, server:port of the RegionServer containing

this region and start-time of the RegionServer process containing this region.The empty key is used to denote table-start and table-end [9].

```
/* set scan object to get required block by data*/
ArrayList<Scan> scans = new ArrayList<Scan>();
Scan scan = new Scan();
scan.setCaching(caching);
scan.setCacheBlocks(cacheBlocks);
scan.setBatch(batch);
if(rhipe.hbase.start.key)
         scan.setStopRow(decodeBase64(rhipe.hbase.rowlim.start));
/* set start rowkey to scan object */
if(rhipe.hbase.end.key)
        scan.setStopRow(decodeBase64(rhipe.hbase.rowlim.end));
/* set end rowkey to scan object */
RowFilter rowFilterRegex = new RowFilter(
        CompareFilter.CompareOp.EQUAL,new RegexStringComparator(
        Bytes.toString(decodeBase64(filter))));
scan.setFilter(rowFilterRegex);
/* using regular expression filter to filter tags */
scans.add(scan);
```

Listing 4: Filter required data block in HBase by help of scan object.

A region with an empty start key is the first region. If region has both an empty start and an empty end key, it is only the region. Basically .META. contains start and end rowkey, which helps to find whether a particular row lies between the range or not. If it lies within the range, then we are at right RegionServer. If particular row does not lie on the range, then it looks at another RegionServer. RegionServer gives table name and we can get particular row, check figure 3.2.

From figure 3.2, we need to find start rowkey '00005' and end rowkey '00012' from tsdb HBase table. Zookeeper knows the location of the RegionServer holding -ROOT- table. It is a RegionServer RS4. which .META. region can tell us about row '00005' and row '00012'. Now RegionServer RS1 looks into -ROOT- table and finds out .META. table M1 at RegionServer RS2 and .META. table M3 at RegionServer RS3 is pointing to the particular rows. Now client interacts with RegionServer RS2 and RegionServer RS3 and looks into M1 and M3 table. RegionServer RS2 contains two regions, M1 table gives information which table contains row '00005'. Similarly, row '00012' can be found.

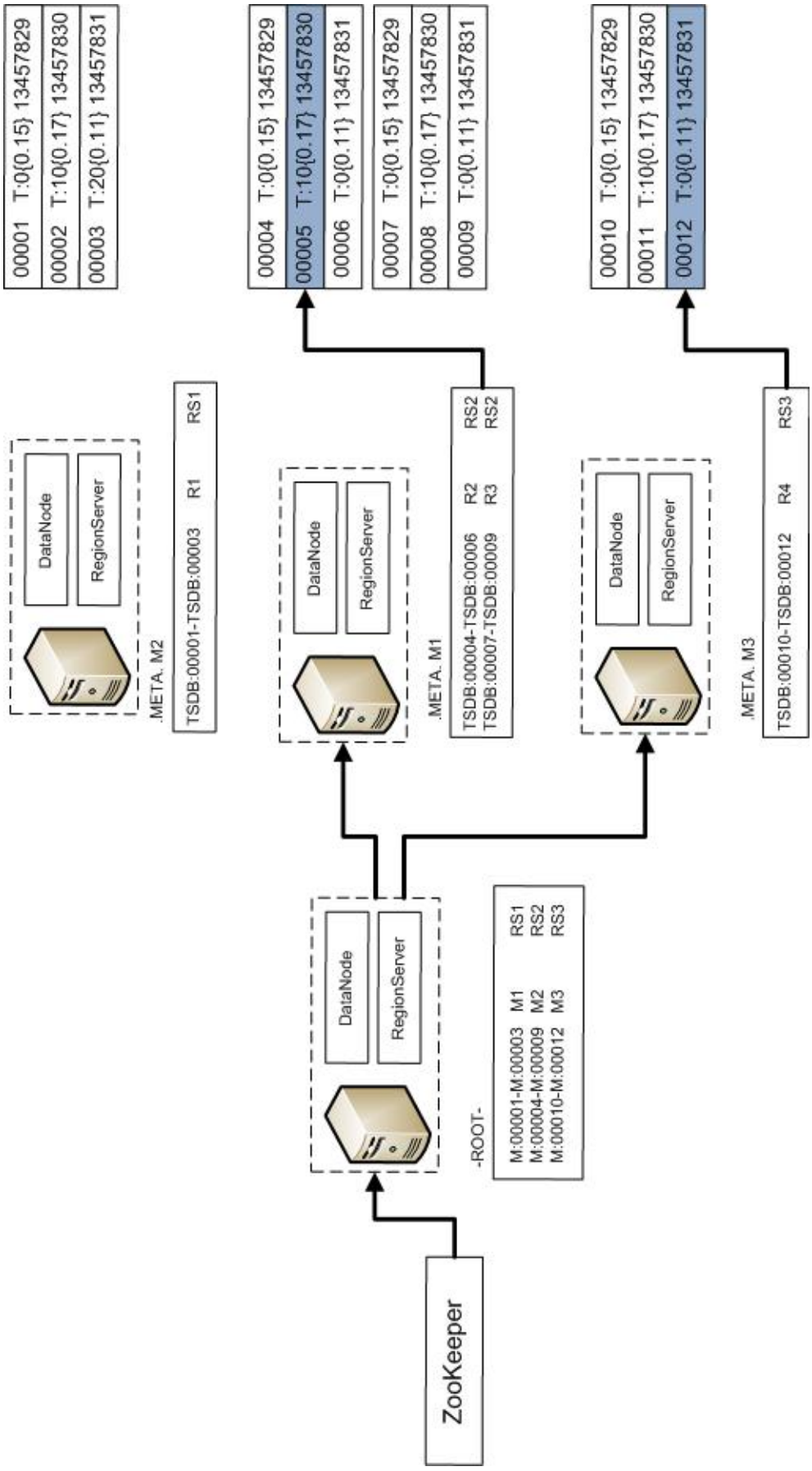

Figure 3.2: Steps that take place when client interact to find start rowkey and end rowkey in HBase system. Zookeeper holds the location of the RegionServer which has the -ROOT-.-ROOT- contains information about all the .META. and RegionServer holding .META. table. -ROOT- gives information which .META. table and RegionServer contain particular row.

Pseudocode listing 4 sets start and end pointers on the scanner object, that is passed through R2Time and RHIPE. Listing 4 shows how to use regular expression on the row key to filter tags.

3.3 Create a custom input format for RHIPE MapReduce Task

Input format in the MapReduce is responsible for two things. First, the actual splitting of the input data and returning a RecordReader instance that defines the classes of the Keys and the Value objects and also provides a next() method that is used to iterate over each input records.

According to Dean and Ghemawat (2008), a MapReduce [27] is a program that takes a set of input key-value pairs and returns a set of output key-value pairs. Key and value can play different roles depending upon the problem. A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks, in a completely parallel manner and returns a pairs <key, set<values>>.

RHIPE provides API to define Java class for input format. OpenTSDB stores data in HBase. We need a tool that can read data from HBase and send to RHIPE MapReduce task.

RHHBaseReader is a java class that reads data from HBase, define its own RecordReader. Similar work has been done for mozilla crash reporting [38], which is still under investigtion. HBase provides serveral inputformat API, which is useful to design own input format as mention in [22][9]. Just reading data from HBase will not solve the task. Formatting the data in Key and Value pairs is also needed.

MetricUID<1371121000><001><001>
MetricUID<1371121000><001><002>
MetricUID<1371121000><001><003>
.....
MetricUID<1371792000><001><001>
MetricUID<1371792000><001><002>
MetricUID<1371792000><001><003>

Table 3.2: Data store in HBase is in sequential order.

In RHHBaseReader, R2Time send start row key, end row key, filter for tags and cache information from R environment. At first, we need to create a scan[ ] object which returns whole data block of tsdb table. On that scan[ ] object, we can set start

row key and end row key to get range of required data block. Data stored in HBase is in sequential order. For example, if there are multiple tags, say 001, 002, 003 data store in HBase is shown in table 3.2. When we set pointer by the help of start and end row key, we get block of required data within that range. If we want to fetch data of all the tags, then we require block of data. But if we want to fetch data of tag value = 001, then we need to create tag filter. HBase provides several API for filtering of data. HBase filters have a powerful feature that can greatly enhance effectiveness in working with data stored in tables. All the filters are actually applied at the server side, also called **predicate pushdown** [9]. Some filter class provided by HBase are as follows:

- RowFilter
- FamilyFilter
- QualifierFilter
- ValueFilter
- DependentColumnFilter
- SingleColumnValueFilter
- SingleColumnValueExcludeFilter
- PrefixFilter
- PageFilter
- KeyOnlyFilter
- FirstKeyOnlyFilter
- FuzzyRowFilter
- InclusiveStopFilter
- TimestampsFilter
- ColumnCountGetFilter
- ColumnPaginationFilter
- ColumnPrefixFilter
- CompareFilter
- RandomRowFilter
- WhileMatchFilter ...

For our implementation, we have chosen combination of RowFilter and CompareFilter along with RegexStringComparator for CompareFilter. CompareFilter is a class based on FilterBase class, and add one more method named compare() operation. Each comparator has a constructor that takes the comparison value. Some of these constructors take a byte[], a byte array, to do the binary comparison. RowFilter is a comparison filter that comes bundled and allows us to filter data based on row keys[9]. We need to filter on row keys, that is why we need to use RowFilter. RegexStringComparator

```
RegexStringComparator RegEx = new RegexStringComparator
(tags filter);
/* Create Regular expression filter */
RowFilter rowFilter = new RowFilter(CompareOp.EQUAL, RegEx);
/* compare if regular expression filter value is equal with
row filter value */
scan.setFilter(rowFilter)
```

Listing 5: RowFilter with CompareFilter using Regular Expression to filter required tags.

compares the value with regular expression provided by the comparator. Regular expression for tags filter is explained above in the listing 3. Here is an example 5 of how to we can use RowFilter:

```
/* Fuzzy filter is als know as fast forwarding filter.
It can skip lot of data, kind of fast algorithm */
FuzzyRowFilter rowFilter = new FuzzyRowFilter(
Arrays.asList(new Pair<byte[], byte[]>(
 Bytes.toBytesBinary("\\x00\\x00\\x00\\x00\\x00\\
 x00\\x00<tagkey><tagvalue>"),
 new byte[] {1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0})));
scan.setFilter(rowFilter)
```

Listing 6: FuzzyRowFilter to filter required tags.

Filtering API is a powerful tool for finding exact data and allows us to optimize disk seeks. FuzzyRowFilter is another powerful filter that can be used with our solution to filter tags from row keys. It performs fast-forwaring based on the fuzzy row key mask provided by users. It is available for HBase version above 0.94. FuzzyRowFilter takes as parameters row key and a mask info. In an example above, in case we want to filter tags from row key, the fuzzy row key we are looking for is "???????<tagkey><tagvalue>". From listing 6 , first 7 bytes are variable (i.e we don't care about it) and last 6 bytes are fixed. We need to supply <tagkey> and <tagvalue> in our filter. The efficiency of using FuzzyRowFilter is determined by how many records filter can actually skip and how many jumps it has to do to skip them. If we need to do full table scan, then this filter is not suitable, however, regular expression filter is the same. Detail about FuzzyRowFilter can be found on [39]

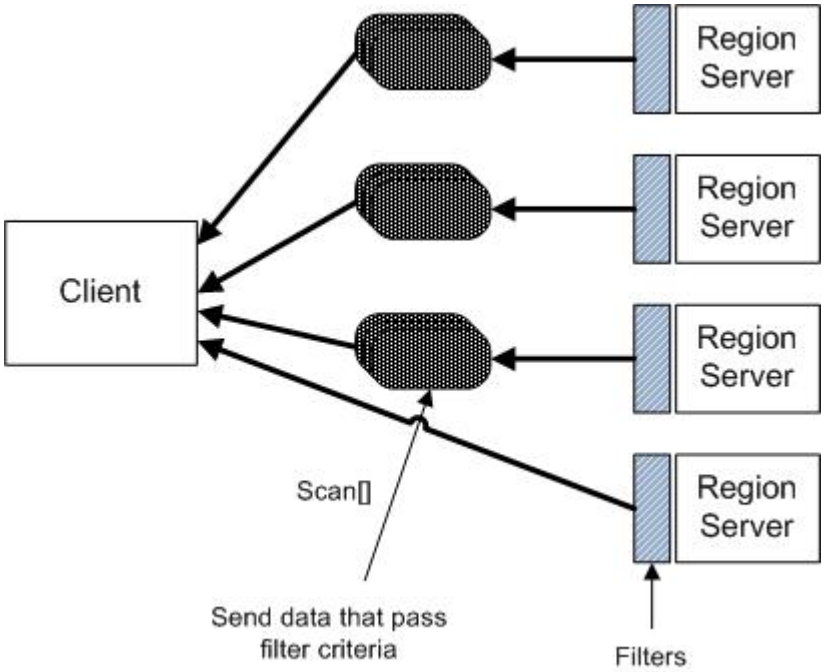

Figure 3.3: RowFilter example, which is done at server side. The main benefit of server side filtering reduce effort to loading unnecessary data, which increase performance to fetch data [9].

3.3.1 Scanner Caching

The HBase API Scan class takes a setting called `setCaching(int caching)` which tells the scanner about how many rows to fetch at a time from the server. The default is 1, which is optimal for single gets, but is decidedly suboptimal for big scans like working with large dataset. There is a property called `hbase.client.scanner.caching` in the `hbase-site.xml` file that forms the configuration for the MapReduce job. User can define caching size through R2Time. Default caching size is 1000 in R2Time framework. We have done some performance testing on section 5 with different cache size and their differences are visible in the results.

The HBase API Scan class takes another setting called `setCacheBlocks( boolean block)` which allows to load the cache with a HBase block. By default setCacheBlock is set true in the hbase-site.xml. User can define `setCacheBlocks` true/false through R2Time framework. If the scan is using the block cache true then it loads the cache with a block, reads the block, and then loads the next one; ignoring the first one for the rest of the scan [40] [9].

3.3.2 Batching

The HBase API Scan class takes a setting called `setBatch(int batch)` which tells the scanner maximum number of values to return for each call to `next()`. Batch works with the column cell of HBase.

For example, if we have 23 columns and we set batch as 5, we will get 5 results instances with 5,5,5,5,3 [41]

$$\boxed{RPCs = \frac{(Row * ColsperRow)}{Min(ColsperRow, BatchSize)} * \left(\frac{1}{ScanCache}\right)}$$

For example, Consider a case HBase table with 15 rows and 25 columns per row. Consider batch size is 25 and scanner caching is 5.
From the above equation,
RPCs = (15*25)/Min(25,25)/5
= 375/25/5
= 3
From this example, we can concluded that it is requires 3 or 4 request for open/close scanner.

3.3.3 InputSplit

An input-split is a chunk of the input data that is processed by a single map. Each map processes a single split. Each split is divided into records, and each map processes each record (key-value pairs) in turn. An input-split does not contain input data. It is just a reference to the data. The storage locations are used by the MapReduce job to place map tasks as close to the split's data as possible, and the size is used to order the splits so that the largest get processed first, in an attempt to minimize the job runtime (Greedy approach) [22]. An InputFormat is responsible for creating the input splits and dividing them into records. Generally, an InputFormat consists of two parts: RecordReader and getSplits.

JobClient calls getSplits() method to fetch splits to run map tasks. Having number of splits, client sends to jobtracker information about which schedule map tasks to process splits on the tasktrackers. On a tasktracker, map task passes the split to the RecordReader to get the <key-value> pairs of that split. The map task uses getRecordReader() method to generate record key-value pairs, which passes to map function.

```
 public class RHHBaseReader  extends InputFormat<RHRaw, RHResult>
{
        public RecordReader<RHRaw, RHResult>
            createRecordReader(InputSplit split, TaskAttemptContext context)
 /* Create RecordReader is to read key/value pairs from HBase table */
            ..............
        public List<InputSplit> getSplits(JobContext context)
/* Get the number of split to run map task, split is located in RegionServer*/
}
```

Listing 7: InputFormat with two method RecordReader and getSplits [22].

```
/* value sent to MapReduce task. value send is combination of
column qualifier and cell value. Each row in HBase contain many cell.*/
for(Map.Entry<byte[] , NavigableMap<byte[],byte[]> > entry: map.entrySet()){
String family = new String(entry.getKey());
for(Map.Entry<byte[], byte[]> columns : entry.getValue().entrySet()){
                        String column = new String(columns.getKey());
                        names.add( family +":"+column);
/* add column qualifier to R object */
                        REXP.Builder thevals = REXP.newBuilder();
/* Create R object */
                        thevals.setRawValue(columns.getValue());
/* add cell value to R object */
                }
        }
/*Final output is send as an array. */
```

Listing 8: Combination of column qualifier and value.

By default, in OpenTSDB, auto splitting is on to avoid *hot-spotting* [9]. Auto splitting is highly recommended for better performance in distributed system. Table are splitted around different region-servers as shown in figure 3.2. Method `getSplits()` checks whether the calculated start row key and end row key lie in which RegionServer and how many splits are needed for jobtracker to schedule task. If required, it performs some logical splitting.

RecordReader's nextKeyValue() method is called repeatedly to populate the key and value objects for the map task. When the reader reaches the end of the stream, the nextKeyValue() method returns false, and the map task completes. The nextKeyValue() method is boolean method which returns true/false. We have custom created RecordReader in R2Time, which fetches key as row key and value combination of column qualifier and value. In R2Time framework, value is of type RHResult, which is combination of column qualifier and value as mentioned in listing 8. Column family

is converted into string format and each value of cell is stored in byte format, which is sent to R environment via REXP Class [16]. REXP is R class that encapsulates and caches R objects as returned from Java. REXP class [16] is added to RHIPE library file.

3.4 Sending HBase data to RHIPE MapReduce task

RHIPE version 0.73 provides an API to read custom input format. In an older version of RHIPE, we need to specify input file which is store in HDFS. In RHIPE, we can define custom input format along with key and value as shown in the listing 9. As mentioned earlier RHHBaseReader is a Java class of R2Time for an input format, which set start/end row key and filter to filter tags. RHBytesWritable is a Java class of RHIPE package, which defines raw format. Key format is row key which is the combination of metric ID, base timestamp and tags that is stored in raw byte format. Raw byte format can be defined by RHBytesWritable Java class in the RHIPE package. Input format for value class is RHResult. As mention in listing 8, value is a combination of column qualifier and the data point which is store in the raw byte format. We need to define zookeeper quorum as mentioned in section 3.1, HBase -ROOT- information is maintain by zookeeper.

```
1  /* HBase has two extra table .META. and -ROOT-, -ROOT- location is
2  know to zookeeper, and then goes to .META. and final to actual
3  data. That's why we need to set zookeeper quorum in RHIPE */
4        mapred$zookeeper.znode.parent <- zooinfo$"zookeeper.znode.parent"
5        mapred$hbase.zookeeper.quorum <- zooinfo$"hbase.zookeeper.quorum"
6        mapred$rhipe_inputformat_class <- "RHHBaseReader"
7  /* Java class to define custom <key,value> pairs from HBase*/
8        mapred$rhipe_inputformat_keyclass <- "org.godhuli.rhipe.RHBytesWritable"
9  /* Java class to define key format, which is raw byte */
10       mapred$rhipe_inputformat_valueclass <- "$RHResult"
11 /* Java class to define value format. This is bit tricky
12 value in HBase is send as <list(column qualifier,values)> */
13       return mapred
```

Listing 9: Defining inputformat in RHIPE.

At R environment any type of statistical calculation like mean, standard deviation, regression etc can be perform.

3.5 Storing output

The final stage is the OutputFormat class and its job is to store the output data in various locations. There are specific implementations that sends output to HDFS or to HBase tables in case of the TableOutputFormat class. Writing to HBase table from MapReduce as a data sink is similar as reading data from a HBase table in terms of implementation. It is the final class that handles key/value pairs and writes them to a HBase table or a file.

```
1  mapred$hbase.mapred.outputtable <- table
2  /* table name of output to be stored */
3  mapred$rhipe_outputformat_class <- "RHTableOutputFormat"
4  /* HBase table format java class, user can define custom format*/
5  mapred$rhipe_outputformat_keyclass <- "org.godhuli.rhipe.RHBytesWritable"
6  /* key is raw byte, which can be define by java class */
7  mapred$rhipe_outputformat_valueclass <- "org.godhuli.rhipe.RHBytesWritable"
8  /* value to be store in HBase table is also raw byte */
```

Listing 10: Storing output in HBase in RHIPE.

In RHIPE, by default, we can store output in a file. To store output into a HBase table, we need to define output format, output format key and value class as mentioned in the listing 10. We need to define table name, where data are required to be stored. RHTableOutputFormat is a Java class available in RHIPE package that converts Map/Reduce output and writes it to an HBase table. RHTableOutputFormat uses TableOutputFormat HBase API. RHTableOutputFormat class has TableRecordWriter which writes key/value pairs as raw byte defined by RHBytesWritable [42].

3.6 Calculating data size in HBase

HBase stores the data in KeyValue format. From the figure 3.4, shows the details including the data types and the size required by each field:

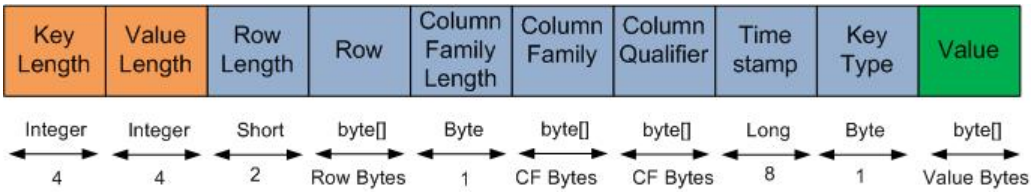

Figure 3.4: Key Value Format of HBase [43]

The difficult part is to calculate size of variable part. OpenTSDB has variable size

row key. Column Family, Column Qualifier and Value are almost known in advance. Value is of 8 bytes, Column Qualifier is of 2 bytes and Column Family is of 2 bytes. Row key size is 13 byte if it has just one tag.

So to calculate the record size:
Key value consist of two part fixed part and variable part.
Fixed part needed by KeyValue format
= Key Length + Value Length + Row Length + CF Length + Key Value + Timestamp
= (4 + 4 + 2 + 1 + 1 + 8) = 20 Bytes

Variable part needed by KeyValue format
= Row + Column Family + Column Qualifier + Value
= 13+2+2+8 = 25 Bytes
Total bytes required = Fixed part + Variable part

Using above expression, we can calculate the record size:
column 1 = 20 + 25 = 45 Bytes.

Open TSDB consist of many column, if tcollector send data in every 15 second then, we will have 240 columns in each row for 60 minute boundary as mention by OpenTSDB [1]. Each row size is calculated as:
Row 1 = 240 * 45 = 10800 Bytes

To store 1 million records, the space required is 10GB.
= 10800 * 1000
= 10 GB

HBase provide API for calculating keyvalue length. Listing 11 shows the uses of HBase API to calculate file size in HBase.

This method is not an efficient method. If record size is massive, it takes large amount of time to calculate. The better solution to calculate size is to read metadata from HFile [44] in HBase.

```
int kvlen = 0;
int size = 0;
for(Result r:ss){
    kvlen = 0;
    for(KeyValue kv : r.raw()){
      kvlen = kvlen+kv.getLength();
/* uses getLength API to calculate keyvalue length in HBase */
      }
        size = size + col;
/* keep on summing up the size for each row */
}
return size in byte
```

Listing 11: Caculation of file size in HBase.

3.7 R2Time

R2Time mainly consist of two components: the R interface, the Java to R bridge. The R interface consists of several functions to interact with the HBase. The Java program uses HBase API to fetch and put data in HBase. The R interface also has several functions to connect HBase and submit computations across the cluster and monitor the status of computations on the R enivornment.

R2Time communicates with HBase via Java code. HBase provides a lot of Java API, which makes easier to interact HBase with other applications. Using Java API, we can fetch data from hbase and launch jobs across the cluster. RHIPE provide users functionality to write their own custom map and reduce function in the R enivronment.

Steps involved in visualizing data across the cluster using R2Time framework is as shown in figure 3.5.

- Step 1: User sets range of date, metrics and tags to visualize data. By the help of R2Time `r2t.rhwatch()` function. Users can supply start date, end date, and the name of metric which is needed to evaluate. Users need to supply tag keys and tag values.
- Step 2: Using OpenTSDB API and HBase Java API, R2Time calculate start/end rowkey and filter to filter data, based on tag keys and tag values. R2Time receive input from step 1. Using function `createFilter()` inside R2Time, It create regular expression for tag filter. Similarly, using function `constructRowKey` R2Time calculate start/end row key. If users want to see intermediate value of these function, R2Time provide another function for R users to print start/end row key and

filters: i.e `r2t.getRowkeyFilter`.

- Step 3: Using Rhipe and R2Time, we can define HBase as a source for MapReduce tasks. getSplit() is an API, that will read splitted data across different region servers. Internally, R2Time set start and end pointer, and use tags filter to get required block of data. Time-series data stored by OpenTSDB is distributed across the cluster. Time-series data is splitted in small block of data and stored in evenly distributed manner.
- Step 4: RHHBaseReader class has a RecordReader method, that returns <key,value> pairs. R2Time sends RowKey as key, and combination of column qualifier and it's value as a value (<Rowkey,list(column quailifier, value>). R2Time also support another type of input format where key is row key and value is list of value (<rowkey,list(value)>). R2Time provides flexible option for R users to choose any of these two input format. Depending on the tasks of data analysis, users can user any input formats.
- Step 5: MapReduce task is run with a help of RHIPE package. rhwatch() is used to send job across the clusters. RHIPE provides API to invoke input format and output format. R2Time define RHHBaseReader Java class to RHIPE. When users submit task via R2Time, rhwatch() function of RHIPE is called. This function trigger Hadoop MapReduce job.
- Step 6: On the R enivornment, users can define its own custom map and reduce function. Users can perform any type of statistical analysis.

```
1  library(r2time)
2  r2t.init()
3  /**Load R2Time library and all jars file*/
4
5  r2t.job( start date, end date, metric,
6  tagKeys, tagValues,
7  table, caching, cacheBlocks,batch,
8  jars, zooinfo, output, jobname="MapReduce job")
9  /* Invoke other sub functions rhwatch for RHIPE and r2t.watch,
10 and function to read data from HBase via R2Time */
```

Listing 12: Pseudocode: Example of sending job to MapReduce via R2Time.

The final stage is the OutputFormat class and its job is to store the output data. There are specific implementations that sends output to HDFS or to HBase tables in R2Time we have defined the TableOutputFormat class. The output data can be stored in HDFS, in the similar way as RHIPE.

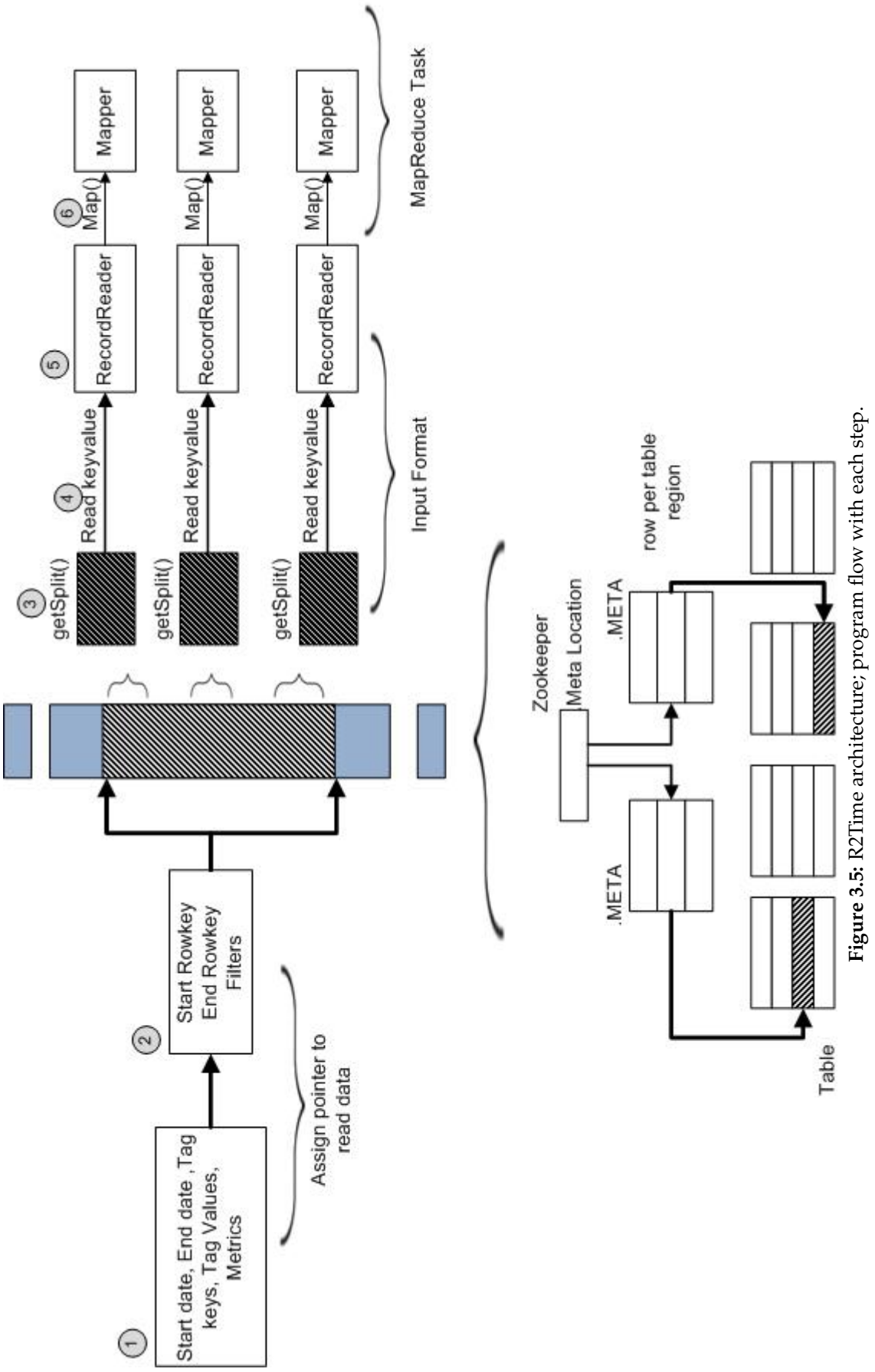

Figure 3.5: R2Time architecture; program flow with each step.

Listing 12 shows how tasks can be defined in R2Time. Users need to send start/end date, metric, tags, table, cache information, zookeeper information and output storage.

When r2t.job is called from R, it invoke RHIPE to read data from HBase via R2Time. Then it follow from step 1 to step 6 as mention earlier.

CHAPTER 4

Results and Analysis

Analysts, engineers, and scientists want to investigate how their information changes over time in order to uncovering interesting and unknown spatial and temporal relationships, to detect patterns of change, major data trends, and outliers (i. e., anomalies) in their data [45]. The goal of this thesis is to allows user to perform advance statistical analysis on the time-series distributed data. The goal has been achieved by building R2Time framework, which act as a bridge between OpenTSDB and RHIPE.

Various type of statistical analysis for time series distributed data can be performed by the help of R2Time and RHIPE. There are four main goals for time series analysis for performing those statistical analysis:

- Trend of data on particular time interval.
- Seasonal variation pattern in time.
- Systematic pattern and random noise.
- Cyclical and Irregular variation.

4.1 Experiment setup

All implementations and experiments were done on Hadoop cluster containing 15 nodes. Each node has a specification of AMD Opteron(tm) 4180 six-core 2.6GHz processor, 16 GB of ECC DDR-2 RAM, 3x3TB secondary hard-disk. Each machines have network card with specification of HP ProCurve 2650 at the network bandwidth of 100 BaseTx-FD. In addition to that, the cluster contains 3 racks including backup and UPS. The racks are connected through a 1Gbps Realtek Semiconductor Co., Ltd. RTL8111/8168B PCI Express Gigabit Ethernet link and a Gigabit Ethernet switch is used to interconnect

all the nodes. All nodes are loaded with Linux CentOS Operating system. All the experiments were conducted by using hadoop-0.20 release.

Out of these 15-nodes, one was configured to serve as NameNode, and the rest were used as DataNodes. The Namenode(Master) is responsible for coordinating the tasks and the datanodes are the ones that actually perform the work.

Hadoop Parameter	**Value**
Hadoop Version	0.20.2-cdh3u6
Replication	3
HDFS Block size	128 MB
io.file.buffer.size	128 MB
io.sort.factor	100
io.sort.mb	100
mapred.map.tasks	100
mapred.tasktracker.map.tasks.maximum	6
mapred.reduce.tasks	22
mapred.tasktracker.reduce.tasks.maximum	3
mapred.child.java.opts	Xmx1024m
mapred.child.ulimit	4194304 bytes
dfs.block.size	134217728
dfs.replicaton	3
HBase Version	0.90.6-cdh3u6
Load average	121.4
Zookeeper Quorum	haisen24.ux.uis.no
Zookeeper Port	2181

Table 4.1: Cloudera configuration.

Hadoop-0.20 Cloudera CH3 was installed in the cluster. Some configuration of hadoop were overridden. In Hadoop, in order to override parameters with new values, we must make changes in one of three files under the configuration folder. These files are core-site.xml, mapred-site.xml and hdfs-site.xml.

HBase-0.90 Apache was installed in the cluster. Haisen23.ux.uis.no is master and rest as datanode. Like Hadoop configuration, we make some changes with HBase configuration. HBase configuration can be overridden by making changes to files (hbase-site.xml) under configuration folder. Cloudera provide web interface to manage this configuration.

Zookeeper was installed in one of datanode (haisen24). When reading data from HBase, we need to pass zookeeper quorum via R2Time framework.

4.2 Data Format

For this experiment, OpenTSDB is an input data source for time-series data (see table 4.2). OpenTSDB used HBase for its data storage. OpenTSDB creates two table in HBase, tsdb and tsdb-uid table. Table tsdb-uid is lookup table for metrics and tags value. The tsdb table is the heart of the time-series database. This table stores time series of measurements and metadata.

.	T:0	T:10	T:20	T:30	..	timestamp
row1	14.44	15.44	17.24	34.34	...	1368187868217
row2	13.65	12.44	15.04	24.44	...	1368187868317
row3	12.22	14.44	15.40	14.34	...	1368187868417
...	...	...	....	...	...	...

Table 4.2: OpenTSDB: 'tsdb' table data format, Number of column T: depends upon the time mention on the tcollector.

4.3 Performance based on different statistical functions

Different statistical function and there performances were tested with R2Time framework. R2Time act as a bridge between RHIPE and OpenTSDB so, the performance analysis is similar as RHIPE.

The figure 4.1 shows the time taken by different statistical functions with respect to different file size using R2Time framework. The light green line in above figure 4.1 is a rowcounter, that counts number of rows present in tsdb HBase table. With simple statistical function row counter job took 88 second to complete on 20 GB of data and 55 second with 2 GB of data size. The graph for green line is almost steady. The red line in figure is used for the calculation of total number of data point present in tsdb table. This job took 310 second to complete on 20 GB data size. If we see in term of statistical analysis, both functions are using `count`, but still counting data point is taking almost 3 times that of row counter.

The figure 4.1 shows gradual increment in the computation time with the increment in file size. R2Time is working with MapReduce model, that is why task are distributed

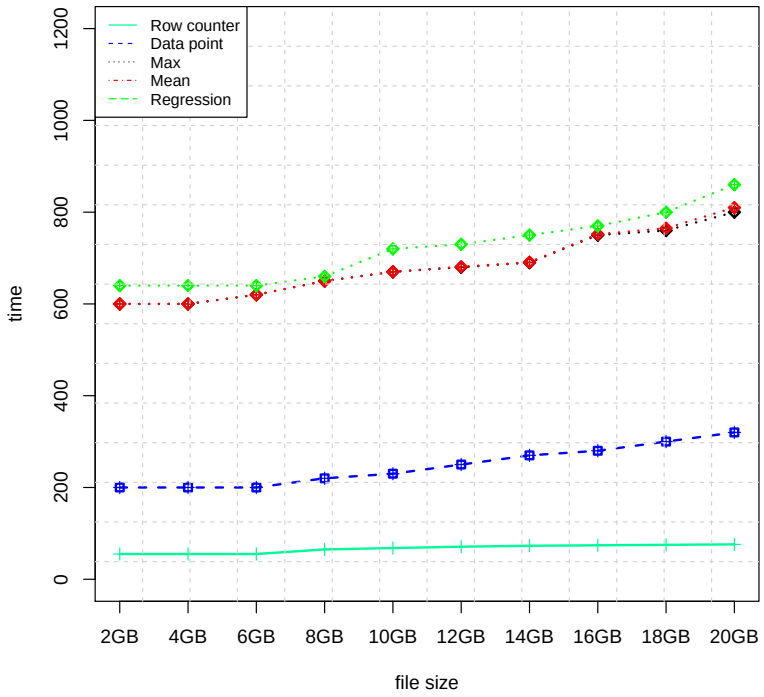

Figure 4.1: Performance based on different statistical functions

between different nodes. Even with the increment in the file size, there was slight change in performance because the task were equally distributed between different nodes. Number of map and reduce was increased with different file size.

As mentioned in the listing 13, row counter is calculated directly by calculating length of map key because number of row is equal to the number of input record inside map function. Similarly listing 14, can explain why calculating total number of data point is taking 3 times more than that of row counter. In this example we are using an extra loop because at first we need to calculate data point present in each row, and then sum up to get total number of data point in tsdb table. This extra loop `lapply` is more time consuming compared to listing 13.

```
1 ### Map function to calculate number of Rows.
2 map <- expression({
3         rowcounter <- length(map.keys)
4 ### count number of rows.
5         rhcollect(1, rowcounter)
6 })
```

Listing 13: R psuedocode for row counter.

From the figure 4.1, it can be seen that other statistical functions like mean, max and min took almost 810 seconds for 20 GB data size.

```
1  ### Map function to calculate total number of data point.
2  map1 <- expression({
3          data <- lapply(seq_along(map.keys),function(i){
4                  v <- length(unlist(map.values[[i]]))
5  ### count number of data point in each row.
6                })
7          datapoint <- sum(as.numeric(data))
8  ### add all the data point present in each row.
9      rhcollect(1, datapoint)
10 })
```

Listing 14: R psuedocode for counting data point.

As shown in listing 15, it is using two loops (`lapply`), which makes it more expensive, compared to the above two functions (count). Line no. 8 is the most expensive operation in this example which took 810 seconds. Function `r2t.toFloat` is used to convert all serialized raw bytes to Float/Int numbers. There are billion of rows, and each row contains many data point, approximately 240 or more. Function `r2t.toFloat` is called billions of time. In terms of statistical analysis, functions like mean, max and min need just one operation. From the 4.1, it can be seen that these statistical functions took almost same time. The red and black line (under red line) had overlapped each other.

Similarly, other statistical functions like regression and correlation, took a bit more time compared to mean operation because regression and correlation need to do some matrix multiplication and matrix transpose. We can use R2Time to do any type of statistical operations.

Performance of R2Time framework was analyzed with different file size. The performance was same as that of RHIPE. Only the difference is RHIPE work with HDFS, while R2Time work with HBase. Figure 4.1, shows time taken to run Hadoop MapReduce task with different file size on different type of statistical functions. As the size of file increases, the graph gets almost steady. Time-series data is stored in column-oriented form, we need to use loop to performance any analysis on this data. With increase in size of data, there is also small increase in time for computation because it needs to compute each column. From the figure 4.1, it was clear that with increases in data size, the graph have supralinear performance.

```
### Map function to calculate maximum data point.
map1 <- expression({
        library(r2time)
        r2t.init()
        val <- lapply(seq_along(map.keys),function(i){
                        v <-  map.values[[i]]
                m.1 <- lapply(seq_along(v), function(i) {
                                v <- as.numeric(r2t.toFloat(unlist(v[[i]])))
### Convert raw bytes to float number
                            } )
                m.4 <- max(unlist(m.1))
### get max value
    })
        max <- max(unlist(val))
        rhcollect(1, max)
})
```

Listing 15: R psuedocode for maximum data point.

4.4 Performance based on InputFormat(Key, Value)

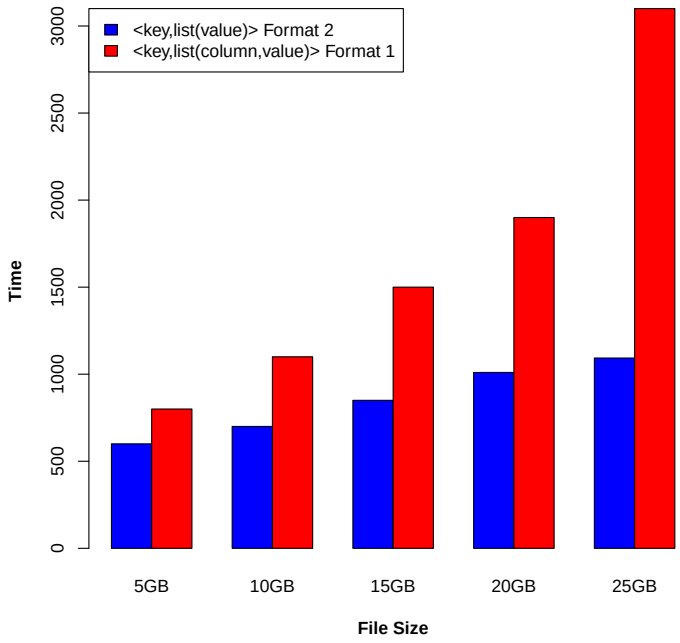

Figure 4.2: Performance based on input formats.

```
#In this map function we are using three nested loop
# which is very expensive.
map1 <- expression({
# for each row key, there is list of column and cell value
        val <- lapply(seq_along(map.keys),function(i){
                v <-  map.values[[i]]
# Each value consist of combination of column qualifer and value
         m.1 <- lapply(seq_along(v), function(j) {
         v.1<-v[[j]]
                 m.1 <- lapply(seq_along(v.1), function(i) {
                        v <- as.numeric(r2t.toFloat(v.1[[i]]))
                        }
                   } )
                m.4 <- max(unlist(m.1))
    })
        max <- max(unlist(val))
        rhcollect(1, max)
})
```

Listing 16: R psuedocode for input format <rowkey,list(column qualifier, value)> .

```
# In this map function we can see two nested loop
map1 <- expression({
        val <- lapply(seq_along(map.keys),function(i){
                v <-  map.values[[i]]
        m.1 <- lapply(seq_along(v), function(i) {
                  v <- as.numeric(r2t.toFloat(unlist(v[[i]])))
                   } )
                m.4 <- max(unlist(m.1))
    })
        max <- max(unlist(val))
        rhcollect(1, max)
})
```

Listing 17: R psuedocode for input format <rowkey,list(value)> .

RHHBaseReader is a Java class that reads data from HBase, defines its own RecordReader. HBase provides several input format API, which is useful to design own input format as mentioned in [22][9]. To perform MapReduce task, R2Time has de-

fined its own custom input formats with row key as a key and value as a combination of column qualifier and cell value (<rowkey,list(column qualifier, value)>. There is another version of input format available in R2Time. In this input format we send row key as key and value as list of cell value (<rowkey,list(value)>).

Input Format 1: <rowkey,list(column qualifier, value)>
Input Format 2: <rowkey,list(value)>

In the input format 1, it requires to use an extra loop to calculate data point for particular column qualifier. But input format 2, it does not need to use this loop because we don't care about column qualifier. We send <key,list(value)> pairs, for MapReduce task as we do for normal MapReduce operation. Only the difference is that it requires multiple values for one key. From the figure 4.2, it is quite clear that <rowkey,list(value)> format is much more faster than that of <rowkey,list(column qualifier, value)>. As the size increases, the performance of input format 1 degrades. Listing 16 shows that input format 1 uses three nested loops which is very expensive. It is expensive because for example, if it needs to compute billions of data point then these nested loops goes billion times.

R2Time provides flexible option for users to select any of the input formats. Depending on the type tasks, or data analysis, users can select required input format. The performance of input format 1 was degraded exponentially with the increases in file size. Because it required extra overload for the framework to read and process extra 2 bytes of data for every billions of data point.

4.5 Performance based on scan cache

We have been attempting to improve performance of HBase map-reduce using various configuration setting for scan cache and batch. HBase table 'tsdb' is used, which has 41 million rows. Each row of tsdb table have hundreds of columns. It has 25 GB of data on HDFS, distributed on 15 different nodes/RegionServers.

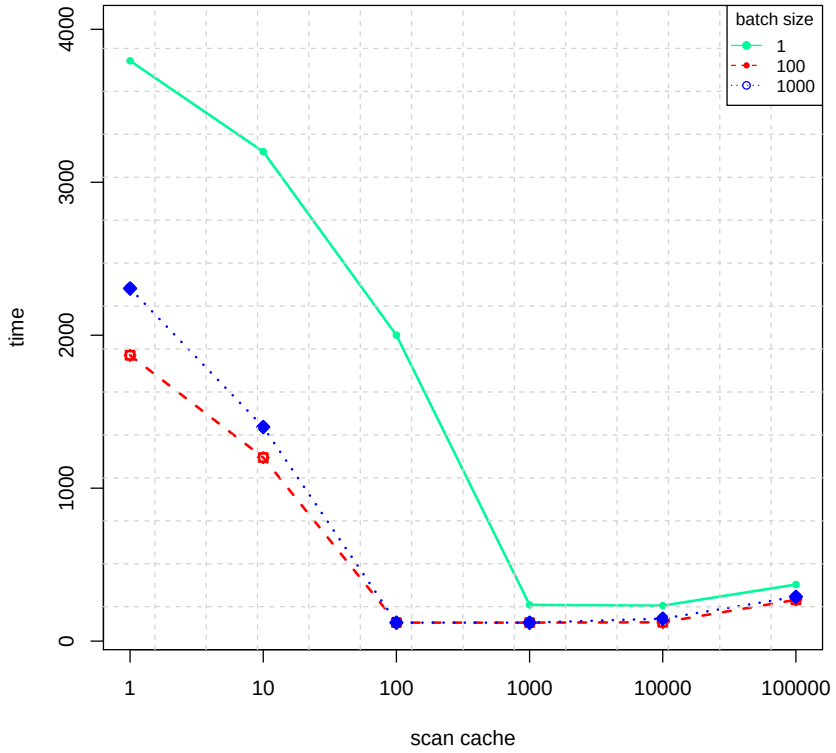

Figure 4.3: Performance based on scan cache

Scan cache determines how many rows were sent from a region server to a client at the same time. The `setBatch` parameter will give us better control over the network bandwidth. By tuning these settings, we have managed to go from processing 7,000 to 25,000 records per second. Scan cache control transfer of rows over the network at once, and keep in memory for fast retrieval on the RegionServer [9]. Just increasing scan cache to maximum will not solve the problem, because sometimes parallel scanning gives us almost no improvement over serial scanning. If we are maxing out network, parallel scanning will make worse. From the figure 4.3, three different curves denote batch size used in this performance analysis, green line is for batch size 1, red line is for batch size 100 and blue line is for batch size 1000. This graph is a plot between scan cache vs time, x-axis denote different size of scan cache used during this experiment and y-axis denote the time taken to complete the task.

Green curve in the figure 4.3 denotes batch size 1 when scan cache size is 1, the time taken to complete task is very high i.e 3700 second. Slowly increasing the size of scan cache make the curve tend to be a straight line in certain range. Further, on increasing scan cache size, increases the time taken to complete the task because sometime parallel

scanning gives us almost no improvement over serial scanning. Similar behaviour can be noticed with other two curve (red and blue). As increasing the size of scan cache, there is a drastic change in performance. The scan caching avoids the rpc calls for each invocation from the map task to the region-server. From the figure 4.3, when scan cache size is increased from 1000 to 10000 and batch size is from 100 to 1000, a straight line was observed. In this data range, the time taken to compute MapReduce task was less compared to other data range. Approximately column present in the 'tsdb' table is 240, so, further increase of batch size from 100 to 1000 had definetly result in more efficient performance.

If we further increase scan cache size from 10000 to 100000, the performance goes on degraded because it had reached to the maximum client memory to hold rows in buffer. To reduce number of parallel scans we need to make each scanner (object of scan) read more rows by increasing size of scan cache. The perfect combination of scan cache and batch size will result in better performance for reading data in HBase. R2Time allows user to have custom setting for scan cache and batch size. The users can place these parameter of their own will. It is necessary for the users to carefully select these parameter, otherwise it may result in worst scenario.

Mathematically, RPC calls need for innovation from the map task to the region-server is given by:

$$\boxed{RPCs = \frac{(Row * ColsperRow)}{Min(ColsperRow, BatchSize)} * \left(\frac{1}{ScanCache}\right)}$$

Figure 4.4: Mathematical equation for calculating RPCs call.

From the above equation, users can calculate RPC call needed from given data. For example, consider a case where total number of rows is 40000, each row have approximately 240 column. So, total data point is 9.6 million.

$$RPCs = \frac{(40000*240)}{Min(240,100)} * \left(\frac{1}{10000}\right)$$

RPCs = 10

So, 10 RPC call is needed. Similarly, if we use batch size = 240, then we can achieve best result:

$$RPCs = \frac{(40000*240)}{Min(240,240)} * \left(\frac{1}{10000}\right)$$

RPCs = 4

Using batch size = 240 and scan cache size = 10000 give us the best result.

From this analysis we have concluded that, it is always better to have same size of batch as number of column. When the batch size is not specified but scan cache is specified, the result of the call will contain complete rows, because each row will contain one result instance because by default batch size is assume to be 1. Even users can configure this setting through configuration file in HBase or for Cloudera users, they can change values through web interface.

CHAPTER 5

Conclusions and Discussion

In this thesis, an R connector (R2Time) between OpenTSDB and RHIPE was implemented and various statistical analyses were performed on time-series data.

The approach to implement a tool for the statistical analysis of time-series data was driven by practical reasons. It is unfeasible to analyse large time-series datasets. So, some sort of distributed computation model is required to reach a satisfiable performance.

OpenTSDB works with large time-series data, but it does not provide advanced statistical functionality to analyze these data. Data scientists and statisticians to analyze data extensively use R. RHIPE was already moving towards reading and analyzing larger sets of data in HDFS. RHIPE allows R users to analyze large sets of data in a similar manner as done by Java with Hadoop. HBase has the advantage of being closely tied to Hadoop (HDFS), which means that there were already tools available for writing HBase MapReduce jobs in Hadoop. There are number of experimental projects under going on to analyse time-series data. A number of different systems that fit the requirements were tested but none of them proved to be efficient. RhipeHBaseMozilla is an experimental project used for Mozilla crash reporting which provides the way to connect HBase to RHIPE. The approach for developing R2Time is quite simple. HBase provides a Java API. There are many different client frameworks like asynchbase written to fetch data from HBase. R2Time was developed with the help of Java (to invoke HBase API) and R.

In this thesis, we presented a framework R2Time that can read time-series data from HBase and perform MapReduce tasks via R. R2Time allows users to write user defined functions for map and reduce to perform any statistical analysis. The output of MapReduce can be stored in HDFS or HBase. The performance analysis with different sizes of

data read from HBase and performing some statistical analysis, gives us a clear view that the task is scalable.

We evaluated the performance of HBase MapReduce through R with representative combinations of a different statistical functionality. The results shows that with increases in the file size, the curve had supralinear nature as shown in figure 4.1. It also shows that, there is a slight increment in time with the complexity of statistical analysis. The results show that by carefully tuning each scan cache size and batch size, the performance of MapReduce task can approach that of parallel database systems for certain analytical tasks. R2Time provides two different type of input formats for processing MapReduce tasks. Depending upon the tasks needed to analyse time-series data, users can choose input format. With the input format 1 (<rowkey,list(column qualifier, value)>), the performance is degraded with the increase in file size. The time taken to complete MapReduce tasks increases exponentially with the increase in file size compared to input format 2 as shown in figure 4.2. R2Time framework enables us to reduce the processing time for one of the larger time-series datasets.

We hope that the insights and experimental results presented in this thesis would be useful for the future development of time-series analysis at a massive scale.

5.1 Future Work

We are using rbase to store output data from MapReduce to HBase table. R2Time framework functions can be elaborated to have this feature in the future. We need to define one new function, which will put data into the HBase table. This `Put` operation is similar to the HBase `Get` operation. The `Get` operation was already implemented in R2Time to read data from HBase.

The next important step for improvement of R2Time framework is to develop methods to read the file size of HBase table by the help of HFile [44] metadata. For the experimental purpose, we have used some temporary solution to know exact file size. With large data sets, this method is time consuming. This method calculates keyvalue length of each column and keeps on adding it, to get the overall file size. Looking at HBase architecture, calculating file size from HFile is the best optimal solution.

Few default statistical functions like max, count, min, mean, linear regression etc. were already implemented. In future, we can implement more complex statistical function by default, but still the user can do advanced statistical analysis with the help of R programming. For advanced statistical analysis, the users need to have some basic

concept of R.

R2Time sends value as a combination of column qualifier and value. In the map function, we need to use three loop: first loop for each row, second loop for column qualifier and third loop for value associated with that column qualifier. If we send the value instead of combination of column qualifier and value, the performance is much better. In the future, we can send column qualifier and value as a 2 dimensional array, instead of sending in a one dimensional array. Sometimes it is necessary to know the delta value to know timestamp of the data point. In the future, OpenTSDB is planning to use variable length encoding, whose information is present in one of a bit in column qualifier.

Appendix A

A.1 Installation Prerequisites

For installating R2Time, it require some prerequisites environment is needed:

- A working Hadoop cluster
- HBase installed in the cluster
- R installed as a shared library
- Google protocol buffers
- RHIPE
- OpenTSDB
- set environment variables, HADOOP-HOME and HBASE-HOME

A.1.1 Installing R2Time

This R CMD INSTALL will install r2time in the cluster.

```
R CMD INSTALL r2time1.0.tar.gz
```

A.1.2 Uninstalling R2Time

This R CMD REMOVE will remove r2time package.

```
R CMD REMOVE r2time
```

A.1.3 Functions in R2Time

Function in R2Time is related to HBase java api functions. The detail of each function is discuss below:

Function related to OpenTSDB

`r2t.init()`

This function initialize, all the jars file needed to run R2Time. Jar file like Hadoop core, HBase, OpenTSDB, RHIPE and R2Time. It basically add all this jar file to rJava classpath.

`r2t.getBaseTimestamp(rowkey)`

Get base timestamp from rowkey, which is 4 byte from the design in OpenTSDB.

`r2t.getRowkeyFilter(startdate,enddate,meticid,tagkeys,tagvalues)`

This function create composite row key, which matches with the design of OpenTSDB row key. This function return 7 byte start/end row key and regular expression for tag filter.

```
r2t.hbaseinput(table, colspec=NULL, rows = NULL,caching = 1000, batch = 100,
cacheBlocks=FALSE, jars="" ,zooinfo, filter = "", fulltable=0)
```

This function define all the java classes for RHIPE MapReduce task. It define R2Time input format as well output format. Input format is define by java class `RHHbaseReader.java` and Output format is define by RHIPE java class (RHTableOutputFormat). This function takes input parameter, table name, start/end row key, zookeeper information, tag filter, full table scan. This function is a connector function for RHIPE with R2Time.

```
r2t.rhwatch(table, rows=row,caching = 1400L, cacheBlocks = FALSE,
autoReduceDetect = FALSE , batch=100 jars="" ,zooinfo, filter = "", fulltable=0,
output="", jobname="MapReduce job", mapred="")
```

This function is derived from RHIPE `rhwatch` function. This function is responsible to submit job to MapReduce. This function read data from HBase via `r2t.hbaseinput()`

function and submit this input to RHIPE `rhwatch()`.

`r2t.getMetrics()`

This function help to get the name of all the metrics present in the 'tsdb' table.

`r2t.getTagKeys()`

This function help to get the name of all the tag keys present in the 'tsdb-uid' table.

`r2t.getTagValue(tagkey)`

This function help to get the name of tag value for particular tagkey present in the 'tsdb-uid' table.

`r2t.connect(table)`

Connect the HBase table.

`r2t.getTimeSereiesData(sdate,edate,tagk,tagv,metrics)` This function return all the data present in tsdb table.

Function related to Datatype conversion

`r2t.toFloat(raw Bytes)`

Convert raw bytes to floating-point number.

`r2t.toInt(raw Bytes)`

Convert raw bytes to integer number.

`r2t.intBittoFloat(integer)`

Convert integer to floating-point number.

References

[1] T.Suna. Opentsdb. URL http://opentsdb.net/.

[2] S. Guha. *Computing Environment for the Statistical Analysis of Large and Complex Data*. PhD thesis, Purdue University Department of Statistics, 2010.

[3] D.R. Brillinger. *Time Series: Data Analysis and Theory. Society for Industrial and Applied Mathematics*. SIAM: Society for Industrial and Applied Mathematics, 2001.

[4] Man-Soon Kim, Sang-Wook Kim, and Miyoung Shin. Optimization of subsequence matching under time warping in time-series databases. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 581–586, New York, NY, USA, 2005. ACM. ISBN 1-58113-964-0. doi: 10.1145/1066677.1066814. URL http://doi.acm.org/10.1145/1066677.1066814.

[5] . Efficient time series matching by wavelets. In *Proceedings of the 15th International Conference on Data Engineering*, ICDE '99, pages 126–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0071-4. URL http://dl.acm.org/citation.cfm?id=846218.847201.

[6] Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.*, 27(2):188–228, June 2002. ISSN 0362-5915. doi: 10.1145/568518.568520. URL http://doi.acm.org/10.1145/568518.568520.

[7] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing sax: a novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15(2):107–144, October 2007. ISSN 1384-5810. doi: 10.1007/s10618-007-0064-z. URL http://dx.doi.org/10.1007/s10618-007-0064-z.

[8] G. Thilina P. Srinath. *Hadoop MapReduce Cookbook*. PACKT Pubblishing, 2013.

[9] Lars George. *HBase: The Definitive Guide*. pub-ORA, pub-ORA:adr, 2011.

ISBN 1-4493-9610-0 (paper), 1-4493-1577-1 (e-book). URL http://proquest.safaribooksonline.com/9781449314682.

[10] David Holstius. opentsdbr. URL https://github.com/holstius/opentsdbr.

[11] Emmet Murphy. Statsd opentsdb publisher backend. URL https://github.com/emurphy/statsd-opentsdb-backend.

[12] Vladimir Kurbalija, Mil Radovanovi, Zoltan Geler, and Mirjana Ivanovi. A framework for time-series analysis. In *Proceedings of the 14th international conference on Artificial intelligence: methodology, systems, and applications*, AIMSA'10, pages 42–51, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15430-1, 978-3-642-15430-0. URL http://dl.acm.org/citation.cfm?id=1885962.1885968.

[13] Harry Hochheiser and Ben Shneiderman. Interactive exploration of time series data. In *Proceedings of the 4th International Conference on Discovery Science*, DS '01, pages 441–446, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42956-5. URL http://dl.acm.org/citation.cfm?id=647858.738708.

[14] Jarke J. Van Wijk and Edward R. Van Selow. Cluster and calendar based visualization of time series data. In *Proceedings of the 1999 IEEE Symposium on Information Visualization*, INFOVIS '99, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0431-0. URL http://dl.acm.org/citation.cfm?id=857189.857665.

[15] Marc Weber, Marc Alexa, and Wolfgang Müller. Visualizing time-series on spirals. In *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, INFOVIS '01, pages 7–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1342-5. URL http://dl.acm.org/citation.cfm?id=580582.857719.

[16] Robert Kabacoff. *R in action*. Manning ; [Pearson Education [distributor], 1 edition, August 2010. ISBN 1935182390. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1935182390.

[17] Jonathan D. Cryer and Kung-Sik Chan. *Time Series Analysis: With Applications in R (Springer Texts in Statistics)*. Springer, 2nd edition, June 2009. ISBN 0387759581. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0387759581.

[18] Dr.Saptarshi Guha. rbase, . URL https://github.com/saptarshiguha/rbase.

[19] Revolution Analytics. rhbase. URL https://github.com/RevolutionAnalytics/RHadoop/wiki/rhbase.

[20] Dan Han and Eleni Stroulia. A three-dimensional data model in hbase for large time-series dataset analysis. In *MESOCA*, pages 47–56, 2012. URL http://dx.doi.org/10.1109/MESOCA.2012.6392598.

[21] Alex Holmes. *Hadoop in Practice*. Manning, 2012.

[22] Tom White. *Hadoop: The Definitive Guide*. O'REILLY, 2012.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945450. URL http://doi.acm.org/10.1145/1165389.945450.

[25] Sundaramurthy. Sundaramurthy blog. URL http://sundar5.wordpress.com/2010/03/19/hadoop-basic/.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: PROCEEDINGS OF THE 6TH CONFERENCE ON SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*. USENIX Association, 2004.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267308.1267323.

[29] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY,

USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989438. URL http://doi.acm.org/10.1145/1989323.1989438.

[30] Horton Works Blog. Apache hbase region splitting and merging. URL http://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/.

[31] Nick Dimiduk and Amandeep Khurana. *HBase in action*. Manning, 2013. ISBN 1617290521. URL http://www.worldcat.org/isbn/1617290521.

[32] . Varnish cache, . URL https://www.varnish-cache.org/docs.

[33] Tomasz Wiktor Wlodarczyk. Overview of time series storage and processing in a cloud environment. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, CLOUDCOM '12, pages 625–628, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-4511-8. doi: 10.1109/CloudCom.2012.6427510. URL http://dx.doi.org/10.1109/CloudCom.2012.6427510.

[34] Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: a compressed database for time series. In *Proceedings of the 4th international conference on Traffic Monitoring and Analysis*, TMA'12, pages 143–156, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28533-2. doi: 10.1007/978-3-642-28534-9_16. URL http://dx.doi.org/10.1007/978-3-642-28534-9_16.

[35] Dr. Saptarshi Guha. Rhipe, . URL https://www.datadr.org/doc/introduction.html#r-and-hadoop-integrated-programming-environment.

[36] . Opentsdb source code, . URL https://github.com/OpenTSDB/opentsdb.

[37] Regular expression. URL http://www.regular-expressions.info/characters.html.

[38] Dr.Saptarshi Guha. Rhipehbasemozilla, . URL https://github.com/saptarshiguha/RhipeHbaseMozilla.

[39] Sematext blog. URL http://blog.sematext.com/?s=RowFilter.

[40] Gbif developer blog. URL http://gbif.blogspot.no/2012/02/performance-evaluation-of-hbase.html.

[41] Scott Miao. Hbase client api. URL http://www.slideshare.net/takeshi_miao/002-hbase-clientapi.

[42] Dr.Saptarshi Guha. Rhipe source code, . URL https://github.com/saptarshiguha/RHIPE.

[43] Prafull Kumar. Prafull's blog. URL http://prafull-blog.blogspot.no/2012/06/how-to-calculate-record-size-of-hbase.html.

[44] Cloudera. Cloudera blog. URL http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/.

[45] Johannes Kehrer. *Integrating Interactive Visual Analysis ofLargeTimeSeriesData into the SimVis System*. PhD thesis, Vienna University of Technology, 2007.