Real Time Micro-Blog Summarization based on Hadoop/HBase

Sanghoon Lee and Sunny Shakya

Abstract –

Twitter is an immensely popular micro-blogging site where people post short messages of 140 characters called tweets. It has over 100 million active users who post about 200 million tweets every day. Twitter, the most popular micro-blogging service by the end of 2009, has gained much interest as a new social medium where tweets can be delivered in real-time, and via multiple access modes including the Web, SMS, and mobile device applications. Twitter has become very popular medium in order to share information among users. Many studies have shown that different agencies are primarily using Twitter to disperse information, particularly links to news articles, about themselves, and to report on their activities. In fact, it has been even a popular means of providing breaking news than many of the top newspapers. Some users put the #symbol in their tweets called a hashtag, in order to mark keywords or to specify a tweet topic. This can be used to categorize tweets. In order for users to retrieve a list of recent posts with the particular topic, Twitter provides a list of popular topics. However, the users have to read manually through the posts for understanding a specific topic because the posts they want to find are sorted by time, not relevancy. Hence, in this project, we intend to build an application which can make a summary relevant to the topic the users want to find based on the hashtag. Since, the data for the application which are the actual tweets can be of considerable amount, we will implement Hadoop/HBase as the application needs to be scalable and also, fault-tolerant.

Introduction –

Relational database systems have been hugely successful in storing and processing structured data over the years. However, relational database systems mostly deal with structured data. But with the advent of applications such as Google, Facebook, Twitter, a huge amount of data is generated which is not structured. All these and other companies also wanted to work with different kinds of data, and it was often unstructured or semi-structured. Hence, there was a need for systems that could work with different kinds of data formats and sources without any data constraint and also could scale greatly. This whole new requirement gives rise to Big Data systems and NoSQL. The publications made by Google on Google File System, MapReduce and Bigtable provided insight into a whole new database systems which can adhere to any types of data and generally don't hold ACID guarantees. These technologies as a collection have come to be known as NoSQL systems. A NoSQL database provides a mechanism for storage and retrieval of data that use looser consistency models than traditional relational databases in order to achieve horizontal scaling and higher availability. Hbase is one of the NoSQL systems which has gain widespread popularity in the Industry and is used in companies to store huge amount of unstructured data such as in Facebook to store facebook messages and in Twitter to store all the tweets. It is developed as part of Apache Software Foundation's Apache Hadoop project and runs on top of HDFS (Hadoop Distributed Filesystem) and Apache ZooKeeper

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Today, HBase depends on Hadoop for two separate concerns. First, Hadoop MapReduce provides HBase a distributed computation framework for high throughput data access and second, the Hadoop Distributed File System (HDFS) gives HBase a storage layer providing availability and reliability. Similary, Hbase depends on Apache Zookeeper to manage its cluster state. In addition, Hadoop and its components are shown in Fig. 1.

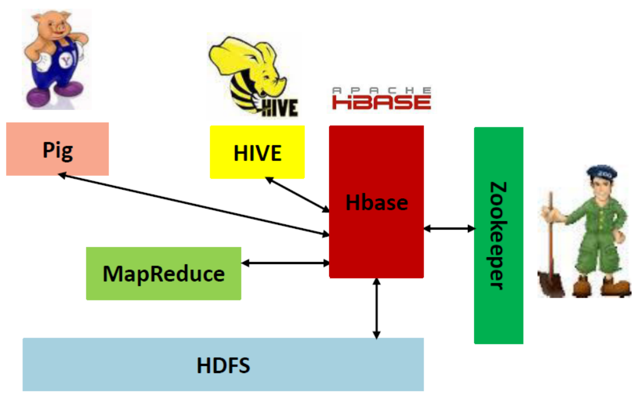


Figure 1. Hadoop and Its components

Hadoop MapReduce -

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.

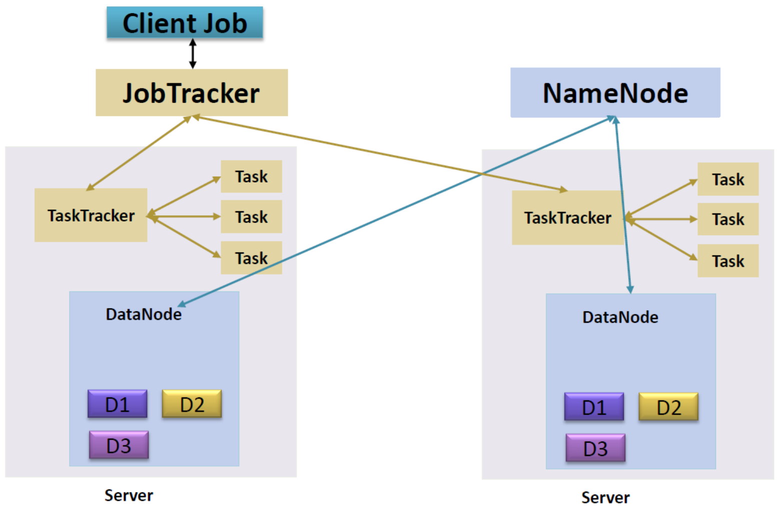


Figure 2. MapReduce

Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to recreate the map output. Reduce task then aggregates the final output for the client job.

Hadoop File System (HDFS)

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters on commodity hardware. Streaming data access HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. It's designed to run on clusters of commodity hardware which are commonly available hardware available from multiple vendors. HDFS supports a traditional hierarchical file organization.

The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file which is stored by the NameNode. HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode and once the NameNode detects that a block in the DataNode has failed, it will create another replica of the block from other replicas.

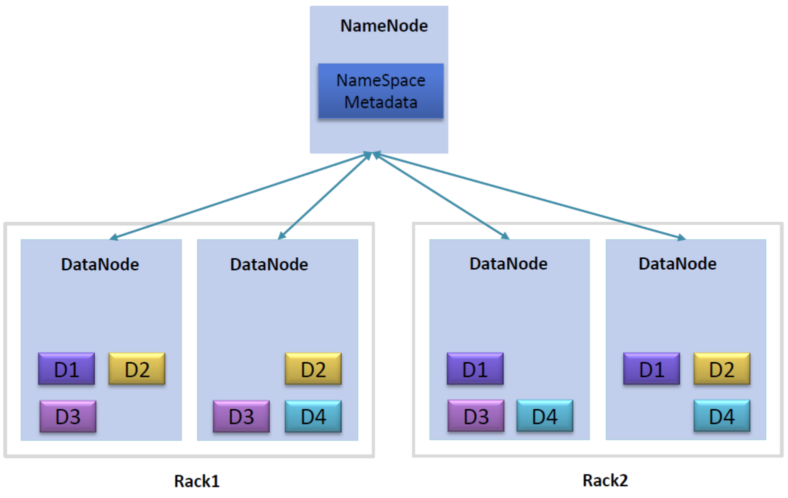


Figure 3. HDFS and data replication among DataNode

Zookeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed. ZooKeeper aims at distilling the essence of these different services into a very simple interface to a centralized coordination service. The service itself is distributed and highly reliable. Consensus, group management, and presence protocols will be implemented by the service so that the applications do not need to implement them on their own. Application specific uses of these will consist of a mixture of specific components of Zoo Keeper and application specific conventions. ZooKeeper Recipes shows how this simple service can be used to build much more powerful abstractions.

Hbase

HBase is a database: the Hadoop database. HBase is a distributed column-oriented database built on top of HDFS and often described as a sparse, distributed, persistent, multidimensional sorted map, which is indexed by rowkey, column key, and timestamp. HBase is designed to run on a cluster of computers instead of a single computer. The cluster can be built using commodity hardware; HBase scales horizontally as more machines are added to the cluster. Each node in the cluster provides a bit of storage, a bit of cache, and a bit of computation as well. This makes HBase incredibly flexible and forgiving. No node is unique, so if one of those machines breaks down, one can simply be replaced it with another. This adds up to a powerful, scalable approach to data storage. HBase can run in three different modes: *standalone, pseudo-distributed, and full-distributed*. In standalone mode, all of HBase runs in just one Java process whereas in pseudo-distributed mode, a single machine runs in many Java processes. In full-distributed mode, HBase is fully distributed across a cluster of machines. The other modes required dependency packages to be installed and HBase to be configured properly.

HBase Table Structure

Like relational database, applications store data into labeled tables in HBase as well. Like relational database, tables are made of rows and columns. However, in HBase, table cells which are the intersection of row and column coordinates are versioned. By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion. A cell's content is an uninterpreted array of bytes. Table row keys are also byte arrays. Table rows are sorted by row key, the table's primary key. By default, the sort is byte-ordered. All table accesses are via the table primary key. Row columns are grouped into column families. All column family members have a common prefix, so, for example, the columns name:firstname and name:lastname are both members of the name column family. A table's column families must be specified up front as part of the table schema definition, but new column family members can be added on demand. For example, a new column name:middlename can be offered by a client as part of an update, and its value persisted, as long as the column family name is already in existence on the targeted table. Physically, all column family members are stored together on the filesystem. HBase tables are like those in an RDBMS, only cells are versioned, rows are sorted, and columns can be added as required by the client as long as the column family where they belong already preexists. Regions tables are automatically partitioned horizontally by HBase into regions. Each region comprises a subset of a table's rows. A region is defined by its first row, inclusive, and last row, exclusive, plus a randomly generated region identifier. Initially a table comprises a single region but as the size of the region grows, after it crosses a configurable size threshold, it splits at a row boundary into two new regions of approximately equal size. Until this first split happens, all loading will be against the single server hosting the original region. As the table grows, the number of its regions grows. Regions are the units that get distributed over an HBase cluster. In this way, a table that is too big for any one server can be carried by a cluster of servers with each node hosting a subset of the table's total regions. This is also the means by which the loading on a table gets distributed. At any one time, the online set of sorted regions comprises the table's total content.

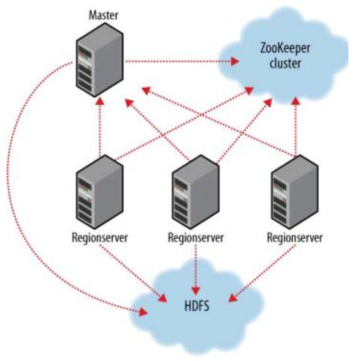


Figure 4. HBase Implementation

HBase Implementation

Hadoop ecosystems are built on a master and slave architecture where there are a cluster of slaves and a coordinating master—NameNode and DataNodes in HDFS and JobTracker and TaskTrackers in MapReduce. Similarly, HBase is characterized with an HBase master node managing a cluster of one or more region server slaves as shown in Figure 5. The HBase master is responsible for assigning regions to registered region servers, and for recovering region server failures. The master node is lightly loaded. The region servers carry zero or more regions and field client read/write requests. They also manage region splits informing the HBase master about the new daughter regions for it to manage the offlining of parent region and assignment of the replacement daughters. HBase depends on ZooKeeper and by default it manages a ZooKeeper instance as the authority on cluster state. Region server slave nodes are listed in the HBase conf/regionservers file as DataNodes and TaskTrackers are listed in the Hadoop conf/slaves file. Cluster site-specific configuration is made in the HBase *conf/hbase-site.xml* and *conf/hbase-env.sh* files.

HBase in operation

HBase, internally, keeps special catalog tables named -ROOT- and .META within which it maintains the current list, state, recent history, and location of all regions afloat on the cluster. The -ROOT- table holds the list of .META table regions. The .META table holds the list of all user-space regions. Entries in these tables are keyed using the region's start row. Row keys, as noted previously, are sorted so finding the region that hosts a particular row is a matter of a lookup to find the first entry whose key is greater than or equal to that of the requested row key. As regions transition—are split, disabled/enabled, deleted, redeployed by the region load balancer, or redeployed due to a region server crash—the catalog tables are updated so the state of all regions on the cluster is kept current. Fresh clients connect to the ZooKeeper cluster first to learn the location of -ROOT-. Clients consult -ROOT- to know the location of the .META. region whose scope covers that of the requested row. The client then does a lookup against the found .META. region to figure the hosting user-space region and its location. Thereafter the client interacts directly with the hosting region server. To save on having to make three round-trips per row operation, clients cache all they learn traversing -ROOT and .META. caching locations as well as user-space region start and stop rows so they can figure hosting regions themselves without having to go back to the .META. table. Clients continue to use the cached entry as they work until there is a fault. When this happens—the region has moved—the client consults the .META. again to learn the new location. If, in turn, the consulted .META. region has moved, then -ROOT- is reconsulted. Writes arriving at a region server are first appended to a commit log and then are added to an in-memory cache. When this cache fills, its content is flushed to the filesystem. The commit log is hosted on HDFS, so it remains available through a region server crash.

When the master notices that a region server is no longer reachable, it splits the dead region server's commit log by region. On reassignment, regions that were on the dead region server, before they open for business, pick up their just-split file of not yet persisted edits and replay them to bring themselves up-to-date with the state they had just before the failure. Reading, the region's memcache is consulted first. If sufficient versions are found to satisfy the query, we return. Otherwise, flush files are consulted in order, from newest to oldest until sufficient versions are found or until we run out of flush files to consult. A background process compacts flush files once their number has broached a threshold, rewriting many files as one, because the fewer files a read consults, the more performant it will be. On compaction, versions beyond the configured maximum, deletes and expired cells are cleaned out. A separate process running in the region server monitors flush file sizes splitting the region when they grow in excess of the configured maximum.

HBase Operations vs SQL Operations

Just like in SQL where there are queries to create table, add/edit/delete data in the table, in Hbase, there are commands that can be used in Hbase shell. HBase shell can be used to interact with HBase from the command line. The shell opens a connection to HBase. With the shell prompt, a table named 'users' can be created by command

```
create 'users', 'info'
```

In HBase, the table creation didn't involve any columns or types. Other than the column family name, HBase doesn't require anything about the data ahead of time. That's why HBase is often described as a *schema-less* database. Unlike in a relational database, tables in HBase are organized into rows and columns. But in Hbase, columns are organized into groups called column families and 'info' is a column family in the users table. A table in HBase must have at least one column family. After the table creation, there is a similar command like in SQL to know the properties of a table

```
describe 'users'
```

The shell describes your table as a map with two properties: the table name and a list of column families. To add data to the HBase table, there is a command put. The following put command puts the bytes '*sshakya1*' to a cell in 'users' table in the 'first' row at the 'info' column while the following get command can be used to get all the cells in the first row.

```
put 'users', 'first', 'info', 'sshakya1'
get 'users', 'first'
```

Every row in an HBase table has a unique identifier called its rowkey. Other coordinates are used to locate a piece of data in an HBase table, but the rowkey is primary. Just like a primary key in a table in a relational database, rowkey values are distinct across all rows in an HBase table. Every interaction with data in a table begins with the rowkey. The HBase API is broken into operations called commands. There are five primitive commands for interacting with HBase: *Get, Put, Delete, Scan,* and *Increment*. Unlike in SQL, where there is an insert into command to insert row into the table, in HBase, the command used to store data is *Put*. Creating a Put instance from a rowkey looks like this:

```
Put p = new Put(Bytes.toBytes("sshakya1"));
p.add(Bytes.toBytes("info"),
Bytes.toBytes("name"),
Bytes.toBytes("Sunny"));
p.add(Bytes.toBytes("info"),
Bytes.toBytes("email"),
        Bytes.toBytes("ss1@gmail.com"));
p.add(Bytes.toBytes("info"),
Bytes.toBytes("password"),
        Bytes.toBytes("ss123"));
```

HBase uses coordinates to locate a piece of data within a table. The rowkey is the first coordinate, followed by the column family. When used as a data coordinate, the column family serves to group columns. The next coordinate is the column qualifier. The column qualifiers in this example are name, email, and password. Because HBase is schema-less, there is no need to predefine the column qualifiers or assign them types. These three coordinates define the location of a cell. The cell is where HBase stores data as a value. A cell is identified by its [rowkey, column family, column qualifier] coordinate within a table. The previous code stores three values in three cells within a single row. The cell storing sshakya1's name has the coordinates [sshakya1, info, name]. All data in HBase is stored as raw data in the form of a byte array, and that includes the rowkeys. The Java client library provides a utility class, Bytes, for converting various Java data types to and from byte[]. Note that this Put instance has not been inserted into the table yet. The last step in writing data to HBase is sending the command to the table using the following command.

```
HTableInterface usersTable = pool.getTable("users");
Put p = new Put(Bytes.toBytes("sshakya1"));
p.add(...);
usersTable.put(p);
usersTable.close();
```

Reading data back out of HBase can be performed using the following Get command.

```
Get g = new Get(Bytes.toBytes("sshakya1"));
Result r = usersTable.get(g);
```

The table gives you back a Result instance containing your data. This instance contains all the columns from all the column families that exist for the row. That's potentially far more data than you need. Hence, the amount of data returned can be limited by placing restrictions on the Get instance.

```
Get g = new Get(Bytes.toBytes("sshakya1"));
g.addColumn(
Bytes.toBytes("info"),
Bytes.toBytes("password"));
Result r = usersTable.get(g);
```

Deleting data from HBase works just like storing it. You make an instance of the Delete command, constructed with a rowkey:

```
Delete d = new Delete(Bytes.toBytes("sshakya1"));
usersTable.delete(d);
```

Part of a row can be deleted by specifying additional coordinates as follows:

```
Delete d = new Delete(Bytes.toBytes("sshakya1"));
d.deleteColumns(
Bytes.toBytes("info"),
Bytes.toBytes("email"));
usersTable.delete(d);
```

Changing data in HBase is done the same way you store new data: create a Put object and give it some data at the the appropriate coordinates, and send it to the table.

```
Put p = new Put(Bytes.toBytes("sshakya1"));
p.add(Bytes.toBytes("info"),
Bytes.toBytes("password"),
Bytes.toBytes("abc123"));
usersTable.put(p);
```

Every time an operation is performed on a cell, HBase implicitly stores a new version. Creating, modifying, and deleting a cell are all treated identically; they're all new versions. Get requests reconcile which version to return based on provided parameters. The version is used as the final coordinate when accessing a specific cell value. HBase uses the current time in milliseconds when a version isn't specified, so the version number is represented as a long. By default, HBase stores only the last three versions; this is configurable per column family. When a cell exceeds the maximum number of versions, the extra records are dropped during the next major compaction.

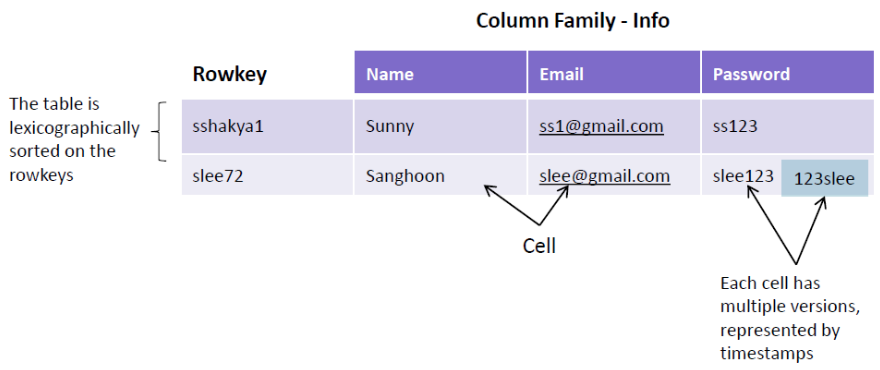


Figure 5. Hbase Table Structure

Querying Hbase

The only way to access records containing a specific value in HBase is by using the Scan command to read across some portion of the table, applying a filter to retrieve only the relevant records. The records returned while scanning are presented in sorted order. HBase is designed to support this kind of behavior so it's fast. To scan the entire contents of a table, use the bare Scan constructor:

```
Scan s = new Scan();
```

To retrieve users with IDs starting with the letter T, Provide the Scan constructor with start and end rows:

```
Scan s = new Scan(
Bytes.toBytes("T"),
Bytes.toBytes("U"));
```

Application: Real Time Micro-Blog Summarization based on Hadoop/HBase

For the implementation of Hadoop/HBase, we designed and implemented an application which provides a summary of the tweets that are posted on Twitter. Twitter is an immensely popular micro-blogging site where people post short messages of 140 characters called tweets. It has over 100 million active users who post about 200 million tweets every day. Twitter, the most popular micro-blogging service by the end of 2009, has gained much interest as a new social medium where tweets can be delivered in real-time, and via multiple access modes including the Web, SMS, and mobile device applications. Twitter has become very popular medium in order to share information among users. The analysis shows that agencies are primarily using Twitter to disperse information, particularly links to news articles about themselves, and to report on their activities. In fact, it has been even a popular means of providing breaking news than the most of the top newspapers. In order for users to retrieve a list of recent posts with the topic phrase, Twitter provides a list of popular topics. Some trends have pound # sign named hashtag before the words or phrase. The hashtag is included particularly in Tweets to explain its relevance to a topic. However, the users have to read manually through the posts for understanding a specific topic because the posts they want to find are sorted by time, not relevancy. For this reason, we tried to make a summary relevant to the topic the users want to find based on the hashtag. The data that is needed for our application are tweets. 1% of the tweets are available for free by Twitter whereas 10% of the tweets can be purchased from twitter. Twitter provides an API in order to get tweets. Since, the size of tweets can be of considerable amount, we are using HBase to store the tweets.

Twitter APIs have two different APIs: *REST APIs* and *Streaming APIs.* Figure 6 shows the *REST APIs*' data flow and *Streaming APIs*' data flow. In the *REST API*, when a web application accepts users' request, the application makes requests to Twitter's API and receives the result. Then, the application sends the result to the user as a response to the user's request. On the other hand, Streaming APIs' connection is executed in a process separate from the HTTP requests. In the streaming process, the server gets the data from Twitter APIs and performs any filtering before sending the data to a data store.

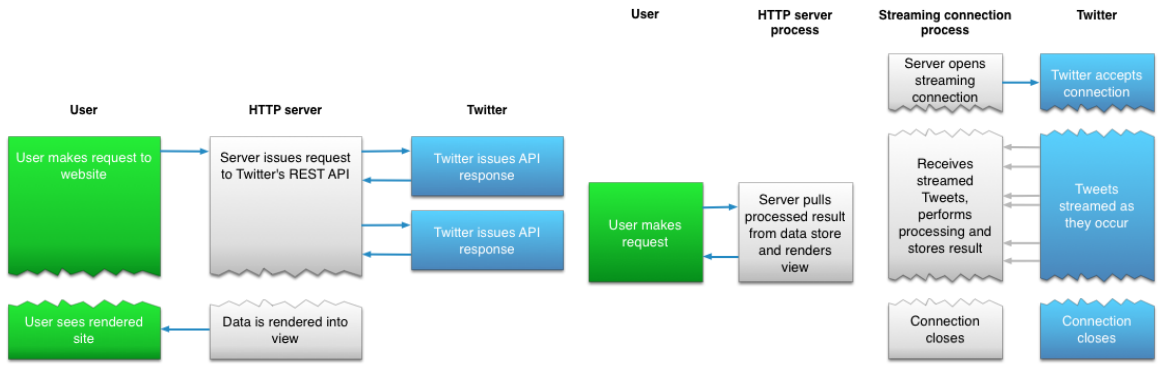

User
HTTP server
Twitter
User makes request to website
Server issues request to Twitter's REST API
Twitter issues API response
Twitter issues API response
User sees rendered site
Data is rendered into view
User
HTTP server process
Streaming connection process
Twitter
Server opens streaming connection
Twitter accepts connection
Receives streamed Tweets, performs processing and stores result
Tweets streamed as they occur
User makes request
Server pulls processed result from data store and renders view
Connection closes
Connection closes

Figure 6. Twitter API

For implementing real time summarization of tweets, we used the Twitter's *Streaming API* as an initial source. Figure 7 shows different types of streams that are available for streaming in Twitter. Twitter offers three kinds of streams for the *Streaming API – Public Streams, User Streams and Site Streams.* Public Streams are the public data flowing and suitable for following users and can be used for data mining. As a single-user streams, User Streams contain all of data corresponding with a single user's view of Twitter. For the multi-version of user streams, Twitter provides Site Streams for many users.

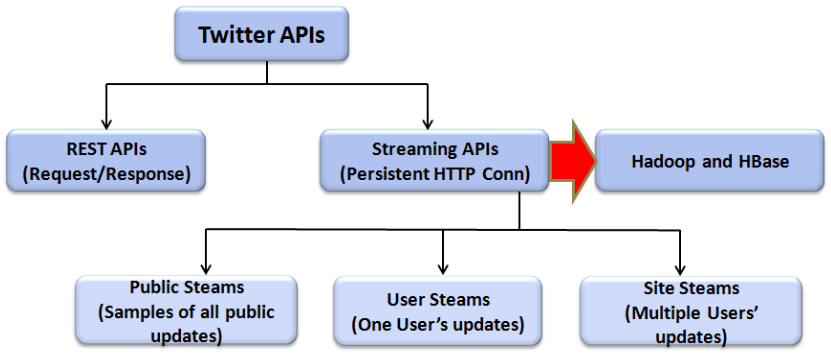

Twitter APIs
REST APIs (Request/Response)
Streaming APIs (Persistent HTTP Conn)
Hadoop and HBase
Public Steams (Samples of all public updates)
User Steams (One User's updates)
Site Steams (Multiple Users' updates)

Figure 7. API used in the application

Figure 7 shows the application data flow. First of all, the application starts with receiving the data from Twitter *Streaming API*. The data contain the username, id, location, hashtags, and posts. In every 1 minute, the application calculates the number of hashtags used in the tweets and produces the top 10 hashtags. These hashtags are used to select the posts or tweets that contain each hashtag. Then, the posts are summarized and the summaries are updated to HBase. Finally, a user can see the summarized posts in the web page. For the summarization, every post containing top 10 hashtags is preprocessed first. The preprocessing step includes tokenizing, stopword removing and stemming. After preprocessing, the posts are vectorized and computed by TF-IDF (Term frequency-Inverses document frequency). At this time, only first two posts which are top ranked by computing the TF-IDF are considered as a summary because we did not complete the experiment yet for the question how many sentences are good for the users. When the process for the summaries is complete, the summaries are updated to HBase in every one minute.

Summarization is the creation of a concise document that represents a meaningful content of a given set of documents. As the tremendous increase in information on the internet is overwhelming manual processing, it is necessary to employ computers to automate the process of summarizing documents. Summarization produces condensed information that maintains document coherence and avoids redundancy. Generally, two approaches in the automatic text summarization have been studied: extractive summary and abstractive summary. The first one is a summary which generates a summary taken exactly as they appear in the documents whereas the abstractive summary generates paraphrased sentences in the input documents. In this project, our approach is relevant to the extractive summary. When the streaming data from Twitter are updated to HBase, we determine the ranking of the hashtags according to the number of hashtags. The hashtags are used to search the raw posts in HBase. Then, the raw posts stored in HBase are extracted by the top 10 ranked hashtags among them. Finally, the selected raw posts are used as the input data for summarization. In general, the input data are preprocessed for automatic text summarization. The preprocessing steps include tokenizing, stopword removing, stemming. Tokenization is the process of replacing a stream of text into words. For this, we firstly tokenized all selected raw posts. In order to compare the weights between words, we need to remove the stop words which are generally filtered out prior to, or after, processing natural language data. We used the most common stop words list that includes words such as *a* and *the*. The stop words are removed from the source documents to increase search performance. Stemming is a process for reducing inflected words to their stem, base or root form. We used Porter Stemmer [1], a very widely used algorithm which became the standard algorithm used for English word stemming. After preprocessing, the raw posts are computed by TF-IPF (Term frequency-Inverses post frequency). TF is the frequency of term with a post and the IPF is a well-known measure of a word's importance. We defined the TF-IPF score as below:

$$TF \cdot IPF = TF(t,p) \times IPF(t)$$

$$IPF = 1 + log \frac{totalPost}{numPost + 1}$$

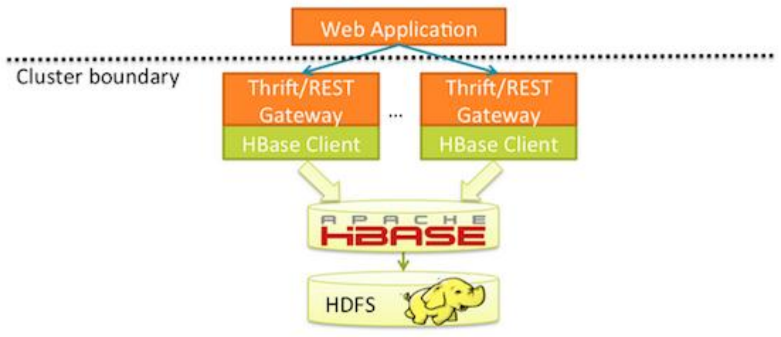
$$Score = \frac{\sum_{i=1}^{k} TF_i \times IPF_i}{Max(\sum_{i=1}^{k} TF_i^n \times IPF_i^n)}$$

Where

- *TF(t, p)* is the number of term t in the post
- *IPF (t)* is the inverse post frequency of the term t.
- *totalPost* is the total number of posts.
- *numPost* is the number of posts that the term t occurs.

When the TF-IPF is computed by the equation above, the top two posts are considered as a summary. When the process for the summaries is finished, the summaries are updated to HBase.

There are various ways to access and interact with Apache HBase. The Java API provides the most functionality, but many people want to use HBase without Java. There are two main approaches for doing that: One is the Thrift interface, which is the faster and more lightweight of the two options. The other way to access HBase is using the REST interface, which uses HTTP verbs to perform an action, giving developers a wide choice of languages and programs to use. For the real time application, we used the HBase *REST API*. For *REST API* to work, another HBase daemon needs to be running to handle these requests. These daemons can be installed in the HBase-rest packages. The diagram below illustrates where REST API is placed in the cluster.



Representational State Transfer (REST) is a kind of software architecture for distributed systems. The REST has emerged as a predominant web API. HBase provides *REST API*. HBase REST is a REST gateway to HBase. REST supports XML or JSON serialization and retrieves a list of all the tables in HBase and returns XML entity body that contains a list of the tables. A user cannot see the summaries on the web through the HBase REST gateway directly. So, we made a servlet to connect the user with the REST gateway. Finally, the user can see the summary. For more details, we will describe whole procedures in the next section.

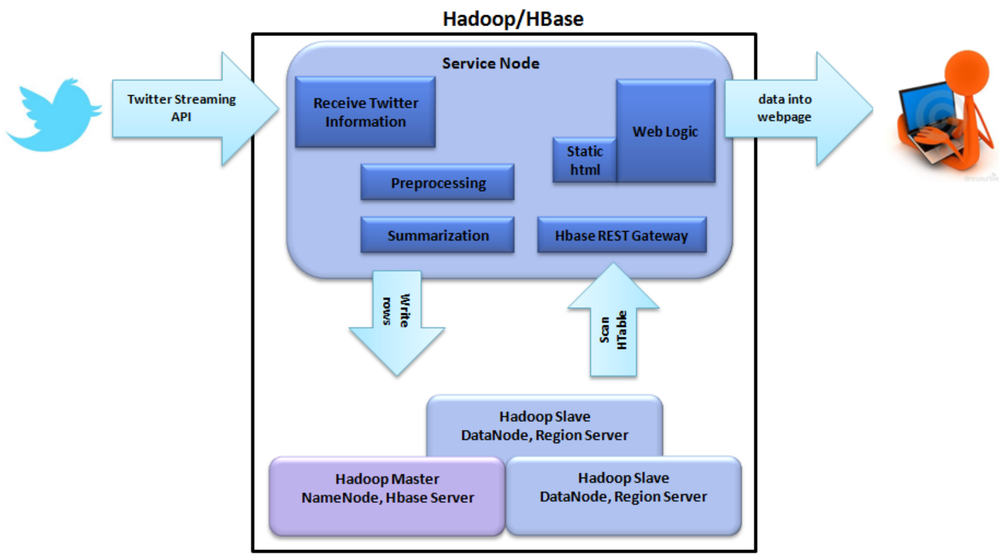
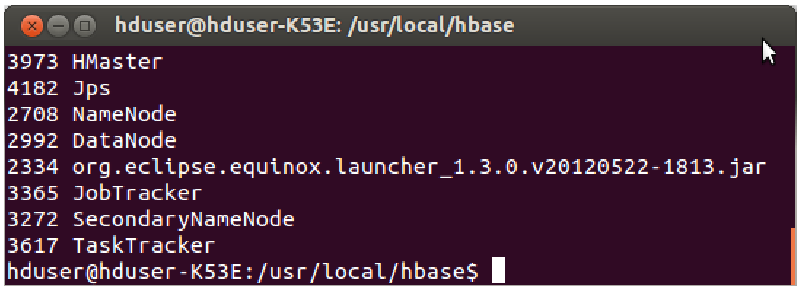


Figure 8. Application Data Flow

Procedure

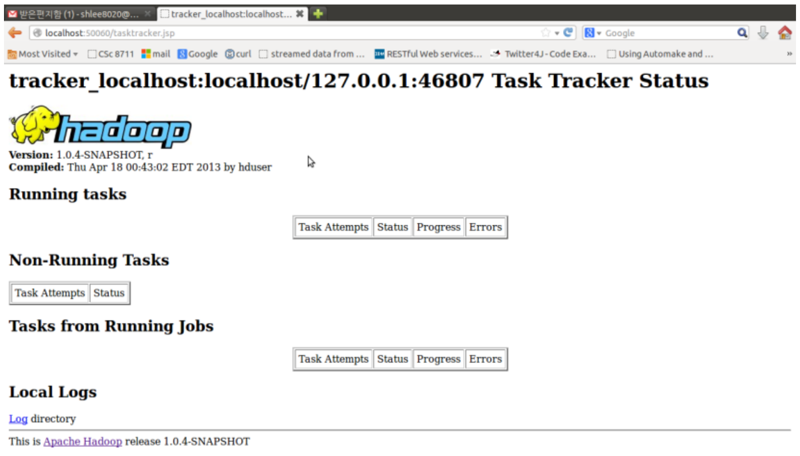
First of all, we need to run several daemons: Namenode, Secondarynamenode, DataNode, JobTracker, TaskTracker, and Hmaster. The following table shows the daemons name and their corresponding tasks.

Daemons	Tasks Performed
NameNode	Keeps the directory tree of all files in the file system. When clients want to locate a file, NameNode responds the requests returning DataNode
DataNode	Stores data in the file system.
SecondaryNameNode	Performs periodic checkpoints. This periodically downloads current NameNode image and edits log files. Even though NameNode can fail, we do not need to shutdown DataNode. Only Namenode needs to be restarted.
Jobtracker	This is a service within Hadoop. Clients submit jobs to JobTracker and the JobTrackers communicate with the NameNode to determine the location of the data.
Tasktracker	Accepts tasks and tracks the progress of the task.
HMaster	Master-server for HBase.



```
hduser@hduser-K53E: /usr/local/hbase
3973 HMaster
4182 Jps
2708 NameNode
2992 DataNode
2334 org.eclipse.equinox.launcher_1.3.0.v20120522-1813.jar
3365 JobTracker
3272 SecondaryNameNode
3617 TaskTracker
hduser@hduser-K53E:/usr/local/hbase$
```

Once all daemons run correctly, we can see web pages of specific ports. Figures below show each web page with ports.



tracker_localhost:localhost/127.0.0.1:46807 Task Tracker Status

Version: 1.0.4-SNAPSHOT, r
Compiled: Thu Apr 18 00:43:02 EDT 2013 by hduser

Running tasks

Task Attempts	Status	Progress	Errors

Non-Running Tasks

Task Attempts	Status

Tasks from Running Jobs

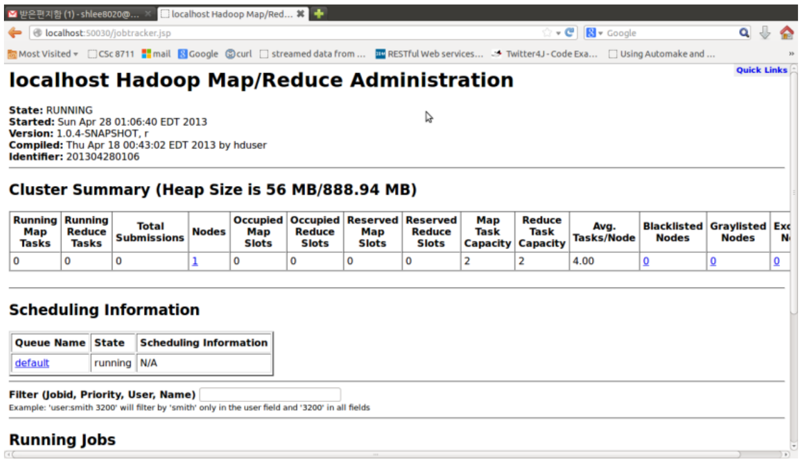
Task Attempts	Status	Progress	Errors

Local Logs

Log directory

This is Apache Hadoop release 1.0.4-SNAPSHOT

Figure 9. Localhost:50060 TaskTracker



localhost Hadoop Map/Reduce Administration

State: RUNNING
Started: Sun Apr 28 01:06:40 EDT 2013
Version: 1.0.4-SNAPSHOT, r
Compiled: Thu Apr 18 00:43:02 EDT 2013 by hduser
Identifier: 201304280106

Cluster Summary (Heap Size is 56 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Graylisted Nodes	Exc N
0	0	0	1	0	0	0	0	2	2	4.00	0	0	0

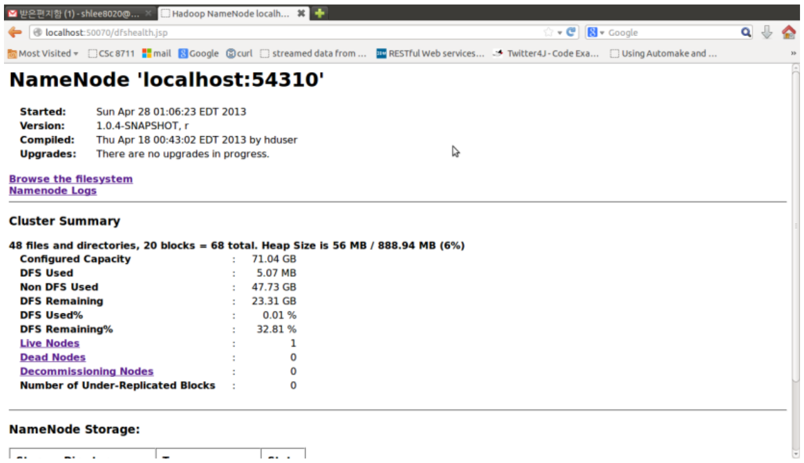
Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Figure 10. Localhost:50030 JobTracker



NameNode 'localhost:54310'

Started:	Sun Apr 28 01:06:23 EDT 2013
Version:	1.0.4-SNAPSHOT, r
Compiled:	Thu Apr 18 00:43:02 EDT 2013 by hduser
Upgrades:	There are no upgrades in progress.

Browse the filesystem
Namenode Logs

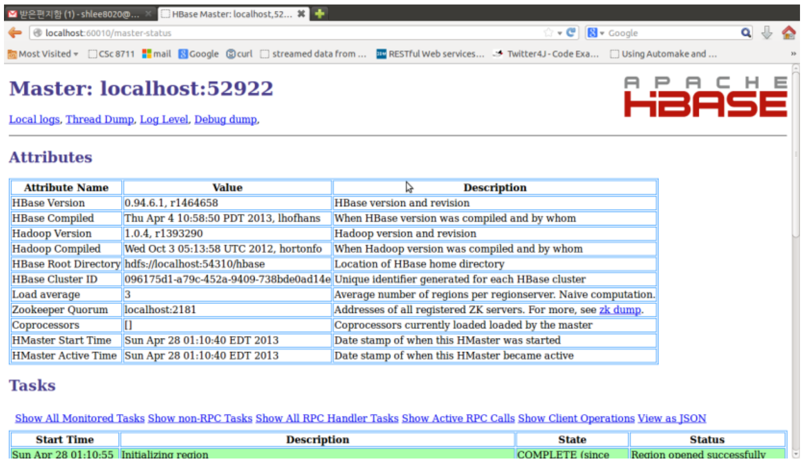
Cluster Summary

48 files and directories, 20 blocks = 68 total. Heap Size is 56 MB / 888.94 MB (6%)

Configured Capacity	:	71.04 GB
DFS Used	:	5.07 MB
Non DFS Used	:	47.73 GB
DFS Remaining	:	23.31 GB
DFS Used%	:	0.01 %
DFS Remaining%	:	32.81 %
Live Nodes	:	1
Dead Nodes	:	0
Decommissioning Nodes	:	0
Number of Under-Replicated Blocks	:	0

NameNode Storage:

Figure 11. Localhost:50070 Namenode



Master: localhost:52922

Local logs, Thread Dump, Log Level, Debug dump,

Attributes

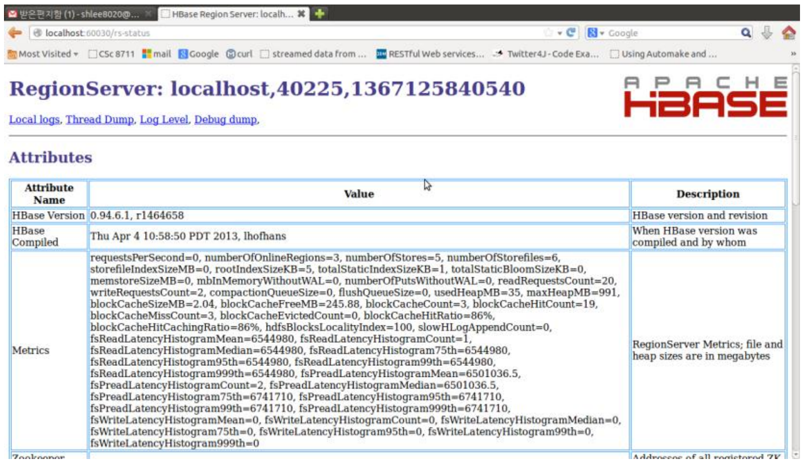
Attribute Name	Value	Description
HBase Version	0.94.6.1, r1464658	HBase version and revision
HBase Compiled	Thu Apr 4 10:58:50 PDT 2013, lhofhans	When HBase version was compiled and by whom
Hadoop Version	1.0.4, r1393290	Hadoop version and revision
Hadoop Compiled	Wed Oct 3 05:13:58 UTC 2012, hortonfo	When Hadoop version was compiled and by whom
HBase Root Directory	hdfs://localhost:54310/hbase	Location of HBase home directory
HBase Cluster ID	096175d1-a79c-452a-9409-738bde0ad14e	Unique identifier generated for each HBase cluster
Load average	3	Average number of regions per regionserver. Naive computation.
Zookeeper Quorum	localhost:2181	Addresses of all registered ZK servers. For more, see zk dump.
Coprocessors	[]	Coprocessors currently loaded loaded by the master
HMaster Start Time	Sun Apr 28 01:10:40 EDT 2013	Date stamp of when this HMaster was started
HMaster Active Time	Sun Apr 28 01:10:40 EDT 2013	Date stamp of when this HMaster became active

Tasks

Show All Monitored Tasks Show non-RPC Tasks Show All RPC Handler Tasks Show Active RPC Calls Show Client Operations View as JSON

Start Time	Description	State	Status
Sun Apr 28 01:10:55	Initializing region	COMPLETE (since	Region opened successfully

Figure 12. Localhost:60010 Master



RegionServer: localhost,40225,1367125840540

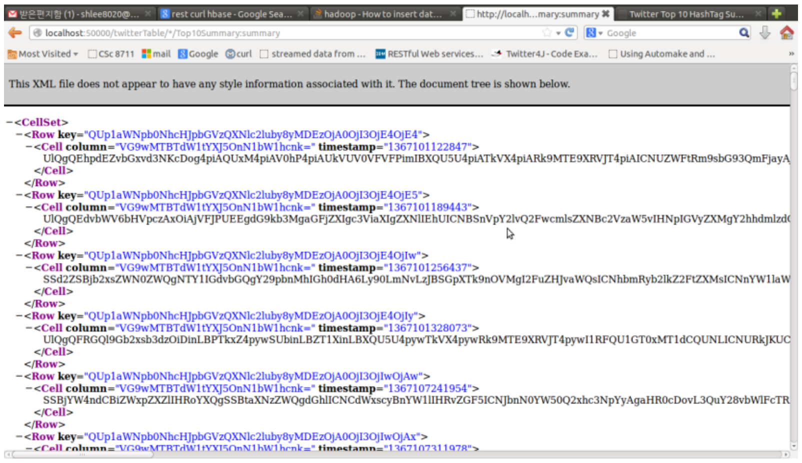
Local logs, Thread Dump, Log Level, Debug dump,

Attributes

Attribute Name	Value	Description
HBase Version	0.94.6.1, r1464658	HBase version and revision
HBase Compiled	Thu Apr 4 10:58:50 PDT 2013, lhofhans	When HBase version was compiled and by whom
Metrics	requestsPerSecond=0, numberOfOnlineRegions=3, numberOfStores=5, numberOfStorefiles=6, storefileIndexSizeMB=0, rootIndexSizeKB=5, totalStaticIndexSizeKB=1, totalStaticBloomSizeKB=0, memstoreSizeMB=0, mbInMemoryWithoutWAL=0, numberOfPutsWithoutWAL=0, readRequestsCount=20, writeRequestsCount=2, compactionQueueSize=0, flushQueueSize=0, usedHeapMB=35, maxHeapMB=991, blockCacheSizeMB=2.04, blockCacheFreeMB=245.88, blockCacheCount=3, blockCacheHitCount=19, blockCacheMissCount=3, blockCacheEvictedCount=0, blockCacheHitRatio=86%, blockCacheHitCachingRatio=86%, hdfsBlocksLocalityIndex=100, slowHLogAppendCount=0, fsReadLatencyHistogramMean=6544980, fsReadLatencyHistogramCount=1, fsReadLatencyHistogramMedian=6544980, fsReadLatencyHistogram75th=6544980, fsReadLatencyHistogram95th=6544980, fsReadLatencyHistogram99th=6544980, fsReadLatencyHistogram999th=6544980, fsPreadLatencyHistogramMean=6501036.5, fsPreadLatencyHistogramCount=2, fsPreadLatencyHistogramMedian=6501036.5, fsPreadLatencyHistogram75th=6741710, fsPreadLatencyHistogram95th=6741710, fsPreadLatencyHistogram99th=6741710, fsPreadLatencyHistogram999th=6741710, fsWriteLatencyHistogramMean=0, fsWriteLatencyHistogramCount=0, fsWriteLatencyHistogramMedian=0, fsWriteLatencyHistogram75th=0, fsWriteLatencyHistogram95th=0, fsWriteLatencyHistogram99th=0, fsWriteLatencyHistogram999th=0	RegionServer Metrics; file and heap sizes are in megabytes

Figure 13. Localhost:60030 RegionServer

When all ports for Hadoop and HBase run properly, we can start REST and Oracle Web Logic Server. Now, we build the java scourcs to recevie the streamming data from Twitter. The streamming data are analyzed in order to put it in HBase. In this step, we count the number of hashtags and extract top 10 hashtags from HBase. We searched all posts having the hashtags and computed TF-IDF for extracting summaries. The summareis and hashtags are stored in HBase. Meanwhile, we can use a web page to extract summaries with top 10 hashtags. Every minute we can extract the summaries. In order for client to get the data from Hbase. we need use the Servlet which returns REST URI. Figure 14 shows RESTful web page which represents the summary results.

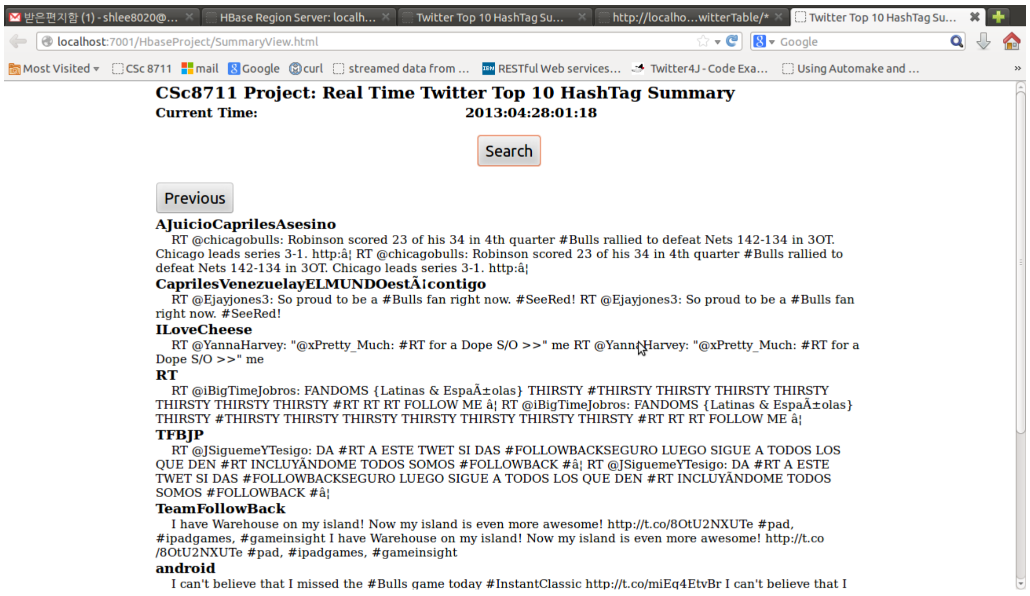

localhost:50000/twitterTable/*/Top10Summary:summary

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
-<CellSet>
 -<Row key="QUp1aWNpb0NhcHJpbGVzQXNlc2luby8yMDEzOjA0OjI3OjE4OjE4">
  -<Cell column="VG9wMTBTdW1tYXJ5OnN1bW1hcnk=" timestamp="1367101122847">
    UlQgQEhpdEZvbGxvd3NKcDog4piAQUxM4piAV0hP4piAUkVUV0VFVFPimIBXQU5U4piATkVX4piARk9MTE9XRVJT4piAICNUZWFtRm9sbG93QmFjayA
   </Cell>
  </Row>
 -<Row key="QUp1aWNpb0NhcHJpbGVzQXNlc2luby8yMDEzOjA0OjI3OjE4OjE5">
  -<Cell column="VG9wMTBTdW1tYXJ5OnN1bW1hcnk=" timestamp="1367101189443">
    UlQgQEdvbWV6bHVpczAxOiAjVFJPUEEgdG9kb3MgaGFjZXIgc3ViaXIgZXNlIEhUICNBSnVpY2lvQ2FwcmlsZXNBc2VzaW5vIHNpIGVyZXMgY2hhdmlzd(
   </Cell>
  </Row>
 -<Row key="QUp1aWNpb0NhcHJpbGVzQXNlc2luby8yMDEzOjA0OjI3OjE4OjIw">
  -<Cell column="VG9wMTBTdW1tYXJ5OnN1bW1hcnk=" timestamp="1367101256437">
    SSd2ZSBjb2xsZWN0ZWQgNTY1IGdvbGQgY29pbnMhIGh0dHA6Ly90LmNvLzJBSGpXTk9nOVMgI2FuZHJvaWQsICNhbmRyb2lkZ2FtZXMsICNnYW1laW
   </Cell>
  </Row>
 -<Row key="QUp1aWNpb0NhcHJpbGVzQXNlc2luby8yMDEzOjA0OjI3OjE4OjIy">
  -<Cell column="VG9wMTBTdW1tYXJ5OnN1bW1hcnk=" timestamp="1367101328073">
    UlQgQFRGQl9Gb2xsb3dzOiDinLBPTkxZ4pywSUbinLBZT1XinLBXQU5U4pywTkVX4pywRk9MTE9XRVJT4pywI1RFQU1GT0xMT1dCQUNLICNURkJKUC
   </Cell>
  </Row>
 -<Row key="QUp1aWNpb0NhcHJpbGVzQXNlc2luby8yMDEzOjA0OjI3OjIwOjAw">
  -<Cell column="VG9wMTBTdW1tYXJ5OnN1bW1hcnk=" timestamp="1367107241954">
    SSBjYW4ndCBiZWxpZXZlIHRoYXQgSSBtaXNzZWQgdGhlICNCdWxscyBnYW1lIHRvZGF5ICNJbnN0YW50Q2xhc3NpYyAgaHR0cDovL3QuY28vbWlFcTR
   </Cell>
  </Row>
 -<Row key="QUp1aWNpb0NhcHJpbGVzQXNlc2luby8yMDEzOjA0OjI3OjIwOjAx">
  -<Cell column="VG9wMTBTdW1tYXJ5OnN1bW1hcnk=" timestamp="1367107311978">
```

Figure 14. Summary Page

Finally, we can use the web page for displaying the extracted summaries with top 10 hashtags as shown in Figure 15. Once we click the Search button, the summaries are displayed in the same page. In every minute, we can see the results on the page.


localhost:7001/HbaseProject/SummaryView.html

CSc8711 Project: Real Time Twitter Top 10 HashTag Summary

Current Time: **2013:04:28:01:18**

AJuicioCaprilesAsesino
RT @chicagobulls: Robinson scored 23 of his 34 in 4th quarter #Bulls rallied to defeat Nets 142-134 in 3OT. Chicago leads series 3-1. http:â¦ RT @chicagobulls: Robinson scored 23 of his 34 in 4th quarter #Bulls rallied to defeat Nets 142-134 in 3OT. Chicago leads series 3-1. http:â¦
CaprilesVenezuelayELMUNDOestÃ¡contigo
RT @Ejayjones3: So proud to be a #Bulls fan right now. #SeeRed! RT @Ejayjones3: So proud to be a #Bulls fan right now. #SeeRed!
ILoveCheese
RT @YannaHarvey: "@xPretty_Much: #RT for a Dope S/O >>" me RT @YannaHarvey: "@xPretty_Much: #RT for a Dope S/O >>" me
RT
RT @iBigTimeJobros: FANDOMS {Latinas & EspaÃ±olas} THIRSTY #THIRSTY THIRSTY THIRSTY THIRSTY THIRSTY THIRSTY THIRSTY #RT RT RT FOLLOW ME â¦ RT @iBigTimeJobros: FANDOMS {Latinas & EspaÃ±olas} THIRSTY #THIRSTY THIRSTY THIRSTY THIRSTY THIRSTY THIRSTY THIRSTY #RT RT RT FOLLOW ME â¦
TFBJP
RT @JSiguemeYTesigo: DA #RT A ESTE TWET SI DAS #FOLLOWBACKSEGURO LUEGO SIGUE A TODOS LOS QUE DEN #RT INCLUYÃNDOME TODOS SOMOS #FOLLOWBACK #â¦ RT @JSiguemeYTesigo: DA #RT A ESTE TWET SI DAS #FOLLOWBACKSEGURO LUEGO SIGUE A TODOS LOS QUE DEN #RT INCLUYÃNDOME TODOS SOMOS #FOLLOWBACK #â¦
TeamFollowBack
I have Warehouse on my island! Now my island is even more awesome! http://t.co/8OtU2NXUTe #pad, #ipadgames, #gameinsight I have Warehouse on my island! Now my island is even more awesome! http://t.co/8OtU2NXUTe #pad, #ipadgames, #gameinsight
android
I can't believe that I missed the #Bulls game today #InstantClassic http://t.co/miEq4EtvBr I can't believe that I

Figure 15. The result page

Conclusion and Future Work

In this project, we studied about HBase/Hadoop and implemented a real-time summarization application that can summarized posts or tweets in Twitter on the basis of hashtags. Since, the data source for this application is real-time posts on Twitter and the amount of such data could be very enormous, we have implemented Hadoop/HBase as the data store since HBase is very suitable for storing enormous amount of data. In addition, using Hadoop/HBase also makes this application very scalable and fault tolerant. In fact, twitter also use Hadoop/HBase to store the tweets/posts. In this application, summaries are generated. However, Summaries need to be evaluated but, the experiments for evaluating the summaries are not performed yet. So, in the future, we will evaluate the summaries comparing with other types of summaries, such as human summary. Also, the implementation of this project is based on pseudo-distributed mode where each Hadoop daemon runs on a single host. In the future, we have a plan to run Hadoop/Hbase on a fully-distributed operation mode where the daemons are spread across all nodes in the cluster. The work for running on a fully-distributed mode would be better than one single node.

Appendix

Environment

We used lots of libraries for the CSc8711 project. In this part, we will explain our system environment for the application.

Hardware:

OS: UBUNTU 12.04
OS type: 64-bit
Memory: 3.6GB
Processor: Intel core i3-2310M CPU@2.10GHz x4

Software:

Eclipse-Java EE-JUNO-SR2
Oracle WebLogic Server 12c (12.1.1) : A commercial JAVA EE application server that implements the whole Java EE specification.

OpenSSH_5.9

SSH should be installed and SSHD should be running to use the Hadoop scripts managing remote Hadoop daemons.

Libraries:

We used following apache commons libraries which enable our Java application to read configuration data from various sources.

```
commons-configuration-1.6.jar
commons-lang-2.5.jar
commons-logging-1.1.1jar
```

Since the task attempt contains the job-id in its name we can find out what logs where create by a specific job. For example, task logs are from Job-Tracker. We changed the version of slf4j.xx to slf4j-1.7.5.

```
log4j-1.2.16.jar
slf4j-api-1.7.5.jar
slf4j-log4j12-1.7.5.jar
```

Twitter4J is an unofficial Java library for the Twitter API so that we can integrate our Java application with the Twitter service.

```
twitter4j-async-3.0.3.jar
twitter4j-core-3.0.3.jar
twitter4j-stream-3.0.3.jar
```

Following is the Hadoop and HBase library that we used.

```
hadoop-core-1.0.4.jar: Hadoop library
hbase-0.94.6.1.jar: Hbase library
```

snappy-1.1.0

Snappy is a compression/decompression library and is widely used in everything from BigTable and MapReduce to internal RPC system. For the CSc8711 project, we add it to the native Hadoop library /usr/local/hadoop/lib/native/Linux-amd64-64

Hadoop Configuration

We run the Hadoop on a single-node in pseudo-distributed mode where each Hadoop daemon runs in a separate Java process.

Project Table

We designed an HBase table for the application. The information of the table below:

```
Hbase Table name: twitterTable

RowKey: username/time
ColumnFamily: UserInfo(column,timestamp, value)
Columns: name, profilelocation, tweetId, content, hashtag

RowKey: hashtag(column,timestamp, value)
ColumnFamily: HashTag
Columns: count

RowKey: hashtag/time (column,timestamp, value)
ColumnFamily: Top10Summary
Columns: summary
```

Sources

In this part, we will explain our source code. We built our application using several Java codes, Servlet, and Javascript.

```
src/servlet/RestfulService.java
```

We made a restful servlet java source. Hbase REST support XML or JSON serialization. For example, we can retrieve a list of tables in Hbase and create, update, and delete a table using the REST api. For the CSc8711 project, we added a httpGet() script to get a URI information on Server. Because a Client could not extract the information of the URI which includes the information of all tables in Hbase, we made a servlet to connect the Client to Server.

```
src/getSummary/getTwitterData.java
```

getTwitterData receives the streaming data from the Twitter and updates information to Hbase.
Because we don't have a permission to access the Twitter, we need an authorization for the data.

```
ConfigurationBuilder cb = new ConfigurationBuilder();
        cb.setDebugEnabled(true)
          .setOAuthConsumerKey("xxxxxxxxxxxxxxxxxxxx")
         .setOAuthConsumerSecret("xxxxxxxxxxxxxxxxxxxx")
          .setOAuthAccessToken("xxxxxxxxxxxxxxxxxxxx")
          .setOAuthAccessTokenSecret("xxxxxxxxxxxxxxxxxxxx");
```

The configuration above allows us to acces to the streaming data. A class "myStatusListener" provides the streaming data, such as user name, user id, user location, hashtags, and contents. The streaming data we collected is updated to Hbase.

When we update the streaming data to Hbase, top 10 hashtags are computed by counting the number of hashtags. Then, the hashtags are used to search posts which has the hashtags in Twitter. Finally, the posts are summarized new summary from Hbase.

```
src/getSummary/getOriginalPost.java
src/getSummary/SentenceFeatures.java
src/getSummary/Tokenization.java
```

In order to make a summary, we used all post having same hashtag performing a preprocessing step. The step is performed by tokenizing the post, filtering stopwords, and stemming the post. The stemmed post is computed by TF-IDF (Term frequency-Inverse document frequency) and the final score is calculated by normalizing each score.

```
WebContent
SummaryView.css
Cascading style sheets for html.

SummaryView.html
SummaryView retrieves the RowKey in Hbase and display current top 10 summaries.

Web-INF/lib/jersey-archive-1.17
Web-INF/web.xml Jersey REST Service
Jersey is an open source library for building RESTful web serivce. We added the Jersey
library to our directory WEB-INF/lib
Web-INF/weblogic.xml
We used WebLogic Server.
```

References

[1] Porter, Martin F., “ An Algorithm for Suffix Stripping,” Program, vol. 14, no. 3, pp. 130–137, 1980
[2] Nick Dimiduk and Amandeep Khurana, “HBase in Action” , MANNING, 2013.