Supporting Social Data Observatory with Customizable Index Structures on HBase - Architecture and Performance

Xiaoming Gao[1], Judy Qiu[1], Evan Roth[1], Karissa McKelvey[1], Clayton Davis[1], Andrew Younge[1], Emilio Ferrara[1], Fil Menczer[1]

[1] School of Informatics and Computing, Indiana University

Abstract. The intensive research activities in social data analysis in recent years suggest the necessity and great potential of a public social data observatory. To effectively support a social data observatory, the storage platform must satisfy its special requirements for loading and storage of Terabyte-level datasets, as well as efficient evaluation of queries involving analysis of the texts of millions of social updates. Traditional inverted indexing techniques do not meet such requirements due to their targeted use cases in text retrieval scenarios. . To address these problems, we propose a general indexing framework, IndexedHBase, to build specially customized index structures for facilitating efficient queries, and employ the HBase system for distributed data storage. IndexedHBase is used to support the Truthy system that collects and analyzes data obtained through the Twitter streaming API. To handle the special queries in Truthy, we develop a parallel query evaluation strategy that can explore the customized index structures efficiently. We evaluate the performance of IndexedHBase on FutureGrid, and compare it with Riak, a widely adopted commercial NoSQL database system. The results show that IndexedHBase provides a data loading speed that is 6 times faster than Riak, and is significantly more efficient in evaluating queries involving large result sets.

1. Introduction

Data intensive computing brings challenges in both large-scale batch analysis and real-time streaming data processing. To meet these challenges, improvements to various levels of cloud storage systems are necessary. Specifically, regarding the problem of search in Big Data, using indices to facilitate query evaluation has been a well-researched area in the field of database [1], and inverted indices [3] are specially designed for full-text search.

Google's Dremel [7] achieves efficient evaluation of aggregation queries on large-scale nested datasets by using distributed columnar storage and multi-level

serving trees. Moreover, Power Drill [12] explores special caching and data skipping mechanisms to provide even faster interactive query performance for certain selected datasets. Discretized Streams [13] proposes a fault-tolerant distributed processing model for streaming data by breaking continuous data streams into small batches and then applying existing fault-tolerance mechanisms used in batch processing frameworks.

Beyond these various data models and system features, it is still challenging to enable real-time search and efficient analysis over a broader spectrum of social data scenarios. For example, [15] discusses the temporal and spatial challenges in context-aware search and analysis on social media data. FluMapper [16] investigates an interactive map-based interface for flu-risk analysis based on near real-time processing of social updates collected from the Twitter streaming API [22]. Truthy [4] builds a public social data observatory that analyzes and visualizes information diffusion on Twitter, covering a broad spectrum of social activities, including presidential elections [8], protest events [14], and meme competition [10]. This process involves analysis of some general entities and relationships contained in its Terabyte-level large-scale social dataset, such as tweets, users, hashtags, retweets, and user-mentions during specific time windows of the social events.

This chapter describes our research on building an efficient and scalable storage platform for Truthy. Due to the sheer size of its structured social dataset, we consider distributed NoSQL database systems as good options for the storage backend. However, how to efficiently evaluate its temporal queries involving text search on hundreds of millions of social updates remains a challenge. Many existing NoSQL databases, such as Solandra (now known as DataStax) [20] and Riak [21], support distributed inverted indices [3] to facilitate searching text data. However, traditional distributed inverted indices are designed for text retrieval applications; they may incur unnecessary storage and computation overhead during indexing and query evaluation time, and thus are not suitable for handling Truthy queries. To solve these problems, specially customized index structures and query evaluation strategies are necessary. Therefore, we propose a general customizable indexing framework on HBase [18] as the basis of the storage platform, which allows users to flexibly define the most suitable index structures to facilitate their queries. Furthermore, based on Hadoop MapReduce [17], we implement a parallel query evaluation strategy that can make the best use of the customized index structures to achieve efficient evaluation of Truthy queries. We name our solution "IndexedHBase", and evaluate its performance on FutureGrid [6].

Currently the total size of historical data on Truthy is about 10 Terabytes. At the time of this writing, the data rate of the Twitter streaming API is about 45 million per day, leading to a growth of ~20GB in the total data size. We develop efficient data loading strategies on that can accommodate fast loading of the historical files as well as the growing speed of streaming data. Our preliminary results show that compared with Riak, a widely adopted commercial NoSQL database system, IndexedHBase provides significantly (6 times) faster data loading speed while re-

quiring much less storage size, and is more efficient (by multiple times) in evaluating queries derived from large result sets.

2. Data and Query Patterns

The entire dataset of Truthy consists of two parts: historical data in .json.gz files, and real-time data coming from the Twitter stream. Fig. 1 illustrates a sample data item, which is a structured JSON string containing information about a tweet and the user who posted it. Furthermore, if the tweet is a retweet, the content the original tweet is also included in a "retweeted_status" field. For hashtags, user-mentions, and URLs contained in the text of the tweet, an "entities" field is included to give more detailed information, such as the ID of the mentioned user, and the expanded URLs.


```
{
    "text":"RT @sengineland: My Single Best... ",
    "created_at":"Fri Apr 15 23:37:26 +0000 2011",
    "retweet_count":0,
    "id_str":"59037647649259521",
    "entities":{
        "user_mentions":[{
                "screen_name":"sengineland",
                "id_str":"1059801",
                "name":"Search Engine Land",
            }],
        "hashtags":[],
        "urls":[{
                "url":"http:\/\/selnd.com\/e2QPS1",
                "expanded_url":null
            }]
    },
    "user":{
        "created_at":"Sat Jan 22 18:39:46 +0000 2011",
        "friends_count":63,
        "id_str":"241622902",
        ...
    },
    "retweeted_status":{
        "text":"My Single Best... ",
        "created_at":"Fri Apr 15 21:40:10 +0000 2011",
        "id_str":"59008136320786432",
                    ...
    },
    ...
}
```


Fig. 1. An example tweet in JSON format

Truthy uses the concept of "memes" to represent sets of related tweets corresponding to specific discussion topics, communication channels or information sources shared among Twitter users. Memes can be identified through elements contained in the texts of tweets, such as keywords, hashtags (e.g., #euro2012), user-mentions (e.g., @youtube), and URLs. Truthy supports a set of temporal queries for extracting and generating various information about tweets, users, and memes. These queries can be categorized into two subsets. The first contains basic queries for getting the IDs or contents of tweets created during a given time window based on their text or user information, including:

get-tweets-with-meme (memes, time_window)
get-tweets-with-text (keyword, time_window)
get-tweets-with-user (user_id, time_window)
get-retweets (tweet_id, time_window)

For the parameters, “time_window” is given in the form of a pair of strings marking the start and end points of a time window, e.g., [2012-06-08T00:00:00, 2012-06-23T23:59:59]. “memes” is given as a list of hashtags, user-mentions, and URLs. “memes” and “keyword” may contain wildcards to specify prefix queries, e.g., “#occupy*”, which will match all tweets containing hashtags starting with “#occupy”..

The second subset contains queries for generating required information based on analysis of the tweets returned from the first subset of queries, including timestamp-count, user-post-count, meme-post-count, meme-cooccurrence-count, get-retweet-edges, get-mention-edges (meme(s), time_window). Here for example, “timestamp-count” returns the number of tweets concerning the given memes posted on each day within the time window. Each “edge” has 3 components: a “from” user ID, a “to” user ID, and a weight indicating how many times the “from” user has retweeted the tweets from the “to” user, or mentioned the “to” user in his/her tweets.

The most significant characteristic of these queries is that they all take a time window as a parameter. This originates from the temporal nature of social activities. In order to evaluate these queries, an obvious brute-force solution is to scan the whole dataset, try to match the content and creation time of each tweet with the query parameters, and generate the results using information contained in the matched tweets. However, due to the drastic difference between the size of the entire dataset and the size of the query result, this strategy is prohibitively expensive. For example, the total number of tweets for 06/01/2012 – 06/20/2012 is 626,958,383, while the number of tweets containing the most popular meme “@youtube” is only 1,906,108, which is smaller by more than two orders of magnitude. In order to efficiently locate the related tweets by their text content, a natural strategy is to utilize inverted indices [3], which are supported by many existing distributed NoSQL database systems, such as Solandra [20] and Riak [21]. However, traditional distributed inverted indices do not provide the best solution for Truthy queries for the following reasons:

First, traditional inverted indices are mainly designed for text retrieval applications, where the main goal is to efficiently find the top K (with a typical value of 20 or 50 for K) most relevant text documents regarding a query composed of a set of keywords. To achieve this goal, information such as frequency and position of keywords in the documents is stored and used for computing relevance scores between documents and keywords during query evaluation. In contrast, queries in Truthy are designed for analysis purposes, meaning that they have to process all the related tweets, instead of the top K most relevant ones, to generate the results. Therefore, data about frequency and position are pure overhead for the storage of inverted indices, the same for relevance scoring in the query evaluation process.

The query evaluation performance can be further improved by cutting these overheads from traditional inverted indices.

Second, query execution plans using traditional inverted indices are not efficient enough for handling Truthy queries. Fig. 2 illustrates a typical query execution plan for "get-tweets-with-meme" using two separate indices on memes and creation time of tweets. This plan uses the meme index to find the IDs of all tweets containing the given memes, and then utilizes the time index to find the set of tweet IDs within the given time window, and finally computes the intersection of these two sets to get the results. Assuming the size of the posting lists for the given memes to be m, and the number of tweet IDs coming from the time index to be n, the complexity of the whole query evaluation process will be $O(m + n) = O(\max(m, n))$, using a merge-based or hashing-based algorithm for the intersection operation. However, due to the characteristics of the dataset in Truthy, there is normally an orders-of-magnitude difference between m and n, as discussed above. As a result, although the size of the query result is bounded by $\min(m, n)$, a major part of query evaluation time is actually spent on scanning and checking irrelevant entries of the time index. In classic text search engines, techniques such as skipping or frequency-ordered inverted lists [3] may be utilized to quickly return the top K most relevant results without evaluating all the related tweets. However, such optimizations are not applicable in Truthy due to the analysis purpose of the queries. Furthermore, in case of high cost estimation for accessing the time index, the search engine may choose to only use the meme index, and generate the results by checking the contents of related tweets. However, a big part of time is still wasted in checking irrelevant tweets falling out of the given time window. The query evaluation performance can obviously be further improved if these unnecessary scanning cost can be avoided.

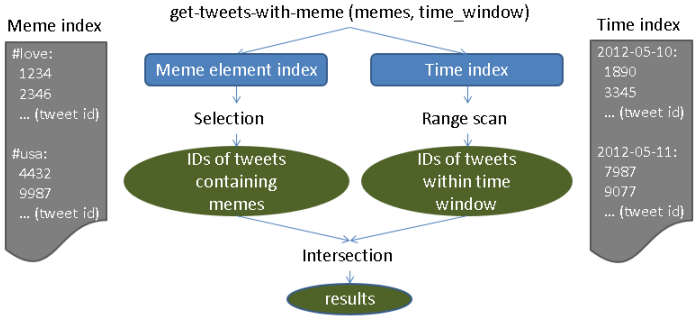

Fig. 2. A typical query execution plan using indices on meme and creation time

In order to solve these issues, we suggest using a customized index structure in IndexedHBase, as illustrated in Fig. 3. It basically merges the meme index and time index, and replaces the frequency and position information in the posting lists of the meme index with creation time of corresponding tweets. Facilitated by this customized index structure, the query evaluation process for "get-tweets-with-meme" can be easily implemented by going through the index entries related to

the given memes, and selecting the tweet IDs associated with a creation time within the given time window. The complexity of the new query evaluation process is O(*m*), which is significantly lower than O(max(*m*, *n*)). To support such index structures, IndexedHBase provides a general customizable indexing framework, which will be explained in Section 3.

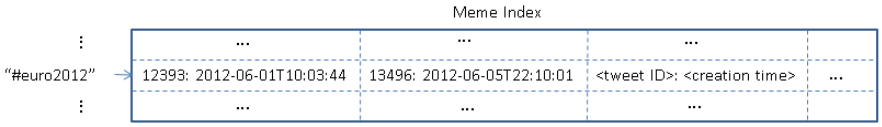

Fig. 3. A customized meme element index structure

3. Design and Implementation of IndexedHBase

3.1 System Architecture

Fig. 4 illustrates the system architecture of IndexedHBase. As the basis of the architecture, HBase is used to host the entire Truthy dataset and related indices with two sets of tables: data tables containing original data, and index tables containing customized index structures for query evaluation. The customized indexing framework supports two mechanisms for building index tables: online indexing that indexes data on the fly when they are loaded into data tables, and batch indexing that is used for building new index structures based on existing data tables. Based on the customizable indexing framework, two data loading strategies are supported to respectively load historical data and streaming data. The parallel query evaluation strategy provides efficient evaluation mechanisms for all the Truthy queries, and is used by upper level Truthy applications to generate various statistics and visualizations.

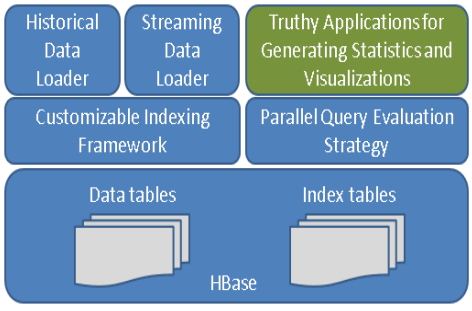

Fig. 4. System Architecture of IndexedHBase

3.2 Customizable Indexing Framework

Table Schemas on HBase

Based on the extendible "BigTable" data model [5] supported by HBase, we design the table schemas in Fig. 5 for Truthy. Tables are managed in units of months – one set of tables are created for each month's data. This management has two benefits. First, the loading of streaming data only changes the tables relative to the current month and does not impact tables for previous months. Secondly, during query evaluations, the amount of index data and original data that needs to be scanned is limited by the months covered under the time window parameter.

Some details about these tables need to be clarified before proceeding further. Each table contains only one column family, e.g., "details" or "tweets", on account of data in the columns being mostly accessed together. Also the user table uses a concatenation of user ID and tweet ID as the row key because Truthy requires keeping track of changes in user metadata associated with each tweet. Finally, besides the text index table, a separate meme index table is created to index the hashtags, user-mentions, and URLs contained in the tweet texts. This is because some special cases, such as expandable URLs and reused screen names in user-mentions, cannot be handled properly by the text index. The meme index table uses hashtags, user-mentions, and URLs as row keys, and each row contains a different number of columns. The name of each column is the ID of one tweet containing the corresponding meme, and the timestamp of the cell value marks the creation time of the tweet. The structures of other index tables can be similarly inferred from Fig. 5.

Using HBase tables to implement customized index structures has the following advantages:

(1) The flexible data model of HBase is suitable for storing the entries of our customized index structures. If needed, users can further extend the existing structures by embedding any additional information in the cell value of each column.
(2) Based on the distributed architecture of HBase, we can achieve high availability for index data and high performance for distributed index access.
(3) Since rows in HBase tables are sorted by row keys, prefix queries involving text keywords and meme elements can be easily completed through range scans over the corresponding index tables.
(4) Since each index structure is implemented as a separate table, it is easy to rebuild an index when its structure is modified, or to build a new customized index structure for handling new queries, without having to reload the dataset. Since the data access pattern in Truthy is mostly write-once-read-many, having multiple index tables will not incur any maintenance cost beyond the initial loading and indexing process.

(5) Based on the original support for Hadoop MapReduce on HBase, it is possible to complete efficient parallel analysis on the index data to generate useful measurements, such as meme popularity distribution used in [10].

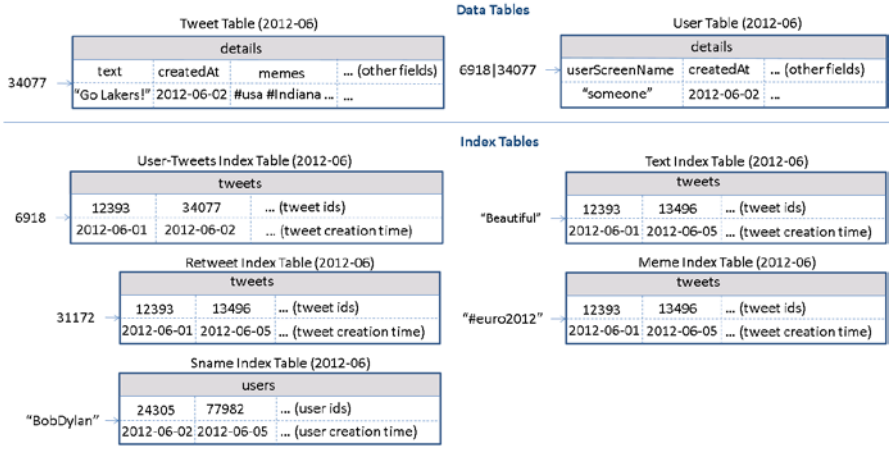

Fig. 5. Table schemas used in IndexedHBase for Truthy

Customizable Indexer Implementation

In order to generate records for the index tables, IndexedHBase implements a customizable indexer library, shown in Fig. 6. Users can customize their index tables by defining what to use as keys and what to use as entries in the index configuration file. The customizable indexer automatically generates index table records according to the configuration file, and inserts them into index tables upon the client application's request.

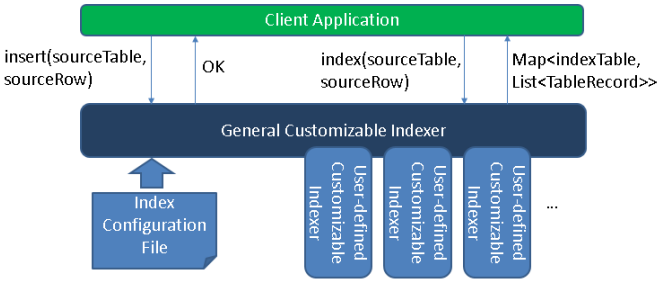

Fig. 6. Components of customizable indexer

Fig. 7 gives an example of the index configuration file in XML format. The whole file contains multiple "index-config" elements. Each element contains the mapping information between one source table and one index table. Users can employ this element to flexibly define how to generate records for the index table

based on a given row from the source table. To deal with even more complicated index structures, they can also implement a user defined customizable indexer on their own, and specify to use this indexer by setting the “indexer-class” element.

```
<?xml version="1.0" encoding="UTF-8"?>
<configurations>
    <index-config>
        <source-table>tweetTable-[month]</source-table>
        <source-column-family>details</source-column-family>
        <source-qualifier>text</source-qualifier>
        <source-timestamp></source-timestamp>
        <source-value-type>full-text</source-value-type>
        <index-table>textIndexTable-[month]</index-table>
        <index-column-family>tweets</index-column-family>
        <index-qualifier>{source}.{rowkey}</index-qualifier>
        <index-timestamp>{source}.details.createdAt</index-timestamp>
        <index-value>{null}</index-value>
    </index-config>
    <index-config>
        <source-table>userTable-[month]</source-table>
        <index-table>snameIndexTable-[month]</index-table>
        <indexer-class>iu.pti.hbaseapp.truthy.UserSnameIndexer</indexer-class>
    </index-config>
</configurations>
```

Fig. 7. An example customized index configuration file

The general customizable indexer and the user defined customizable indexer must both implement a common interface, which declares one “index()” method, as presented in Fig. 8. This method takes the name and row data of a source table as parameters, and returns a map as a result. The key of each map entry is the name of one index table name, and the value is a list of records for that index table.

```
public interface CustomizableIndexer {
    public Map<String, List<TableRecord>> index(sourceTableName, sourceTableRow);
}
```

Fig. 8. Pseudocodes for the “CustomizableIndexer” interface

Upon initialization, the general customizable indexer reads the index configuration file, and analyzes each “index-config” element. If a user defined indexer class is specified, a corresponding indexer instance will be created. When “index()” is invoked during runtime, the general customizable indexer will go through all the “index-config” elements related to the source table, and generate records for each related index table, either by following the rules defined in “index-config” or by invoking a user-defined indexer. Finally, all index table names and records are added to the result map and returned to the client application.

Online Indexing Mechanism and Batch Indexing Mechanism

IndexedHBase provides two means of indexing data in the tweet table and user table: online indexing and batch indexing. The online indexing mechanism is implemented through the “insert()” method of the general customizable indexer, displayed in Fig. 6. The client application invokes the “insert()” method of the

general customizable indexer to insert one row to a source table. The indexer will first insert the given row to the source table, and then generate index table records for this row by invoking "index()", and insert them to the corresponding index tables. Therefore, from the client application's perspective, data in the source table are indexed "online" when first inserted into the table.

The batch indexing mechanism is designed for generating new customized index tables after all the data have been loaded into the source table. This mechanism is implemented as a "map-only" MapReduce job using the source table as input. The job accepts a source table name and an index table name as parameters, and starts multiple mappers to index data in the source table in parallel, each processing one region of the table. Each mapper works as a client application to the general customizable indexer, and creates one indexer instance at its initialization time. The indexer is initialized using the given index table name so that when "index()" is invoked, it will only generate index records for that single table. The "map()" function takes a <key, value> pair as input, where "key" is a row key in the source table and "value" is the corresponding row data. For each row of the source table, the mapper uses the general customizable indexer to generate index table records and write these records as output. All output records are handled by the table output format, which will automatically insert them into the index table.

3.3 Data Loading Strategies

IndexedHBase supports distributed loading strategies for both streaming data and historical data in Truthy. Fig. 9 shows the architecture of the streaming data loading strategy. In this strategy, one or multiple distributed loaders are running concurrently. All loaders are connected to the same stream using the Twitter streaming API, and each is responsible for loading a portion of the data. Each loaders is assigned a unique loader ID, and works as a client application to the general customizable indexer. Upon receiving a tweet JSON string from the stream, the loader will first take the tweet ID and do a modulus operation over the total number of loaders in the system. If the result equals its loader ID, it will load the tweet to IndexedHBase. Otherwise the tweet is skipped. To load a tweet, the loader first generates records for the tweet table and user table based on the JSON string, then loads them into the tables by invoking the "insert()" method of the general customizable indexer, which will complete online indexing and update all the data tables as well as relevant index tables.

The historical data loading strategy is implemented as a MapReduce program. Since tables are managed in the unit of months, one separate MapReduce job is launched to load the historical .json.gz files for each month, and multiple jobs can be running simultaneously in the system. Each one will start multiple mappers in parallel, and every mapper is responsible for loading data from one file. At running time, each line in the .json.gz file is given to the mapper as one input, which

contains the JSON string of one tweet. The mapper first creates records for the tweet table and user table based on the JSON string and then invokes the general customizable indexer to get all the related index table records. All table records are handled by the multi-table output format, which automatically inserts them into the related tables. Finally, if the JSON string contains a "retweeted_status", the corresponding substring will be extracted and processed in the same way.

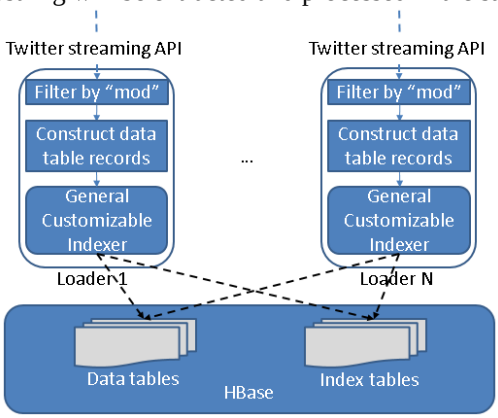

Fig. 9. Streaming data loading strategy

3.4 Parallel Query Evaluation Strategy

Based on the customized index tables generated by the data loading and indexing process, we develop a two-phase parallel query evaluation strategy viewable in Fig. 10. For any given query, the first phase uses multiple threads to find the IDs of all related tweets from the index tables in relevant months, and saves them in a series of files containing a fixed number (e.g., 30000) of tweet IDs. The second phase launches a MapReduce job to process the tweets in parallel and extract the necessary information to complete the query. For example, to evaluate "user-post-count", each mapper in the job will access the tweet table to figure out the user ID corresponding to each tweet ID, count the number of tweets by each user, and output all counts when it finishes. The output of all the mappers will be processed by multiple reducers in parallel to finally generate the total tweet count of each user ID. Implementation of the other queries can be similarly inferred.

Beyond the basic description above, two special aspects of the query evaluation strategy are worth discussing.

First, as described in Section 2, prefix queries can be constructed by using parameters such as "#occupy*". For this type of queries, IndexedHBase provides two options for getting the related tweet IDs in the first phase. One option is to simply complete a sequential range scan of rows in the corresponding index tables and get all the qualified tweet IDs. The other option is to use a MapReduce pro-

gram to complete parallel scanning over the range of rows. This option is only faster for parameters covering a large range spanning multiple regions of the index table. When using prefix queries, users are allowed to specify which option to use based on their estimation of the covered range size.

Next, the number of tweet IDs in each tweet ID file actually implies a tradeoff between parallelism and scheduling overhead. When this number is set lower, more mappers will be launched in the parallel evaluation phase, which means the amount of work done by each mapper decreases while the total task scheduling overhead increases. The optimal number to use actually depends on the total number of related tweets and the amount of resources available in the infrastructure. Therefore, we set the default value of this number to 30,000 and leave it configurable by the user when they run specific queries.

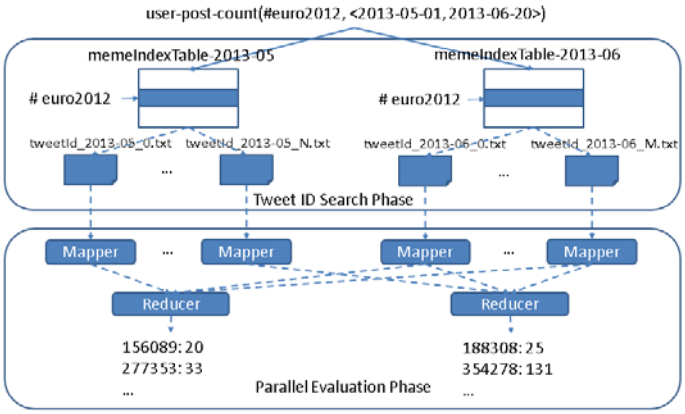

Fig. 10. Two-phase parallel evaluation process for an example "user-post-count" query

4. Performance Evaluation Results and Comparison with Riak

4.1 Testing Environment Configuration

We use 8 nodes on the Bravo cluster of FutureGrid to complete tests for both IndexedHBase and Riak. The hardware configuration for all eight nodes is listed in Table 1. Each node runs CentOS 6.4 and Java 1.7.0_21. For IndexedHBase, Hadoop 1.0.4 and HBase 0.94.2 are used. One node is used to host the HDFS headnode, Hadoop jobtracker, Zookeeper, and HBase master; the other 7 are used to host HDFS datanodes, Hadoop tasktrackers, and HBase region servers. Data replication level is set to 2 on HDFS. The configuration details of Riak will be given in Section 4.2.

Table 1. Per-node configuration on Bravo

CPU	RAM	Hard Disk	Network
8 * 2.40GHz (Intel Xeon E5620)	192GB	2TB	40Gb InfiniBand

Besides Bravo, we also use the Alamo HPC cluster of FutureGrid to test the scalability of the historical data loading strategy of IndexedHBase, since Alamo can provide a larger number of nodes through dynamic HPC jobs. The hardware configuration of each node on Alamo is listed in Table 2. Software configuration is mostly the same as Bravo.

Table 2. Per-node configuration on Alamo

CPU	RAM	Hard Disk	Network
8 * 2.66GHz (Intel Xeon X5550)	12GB	500GB	40Gb InfiniBand

4.2 Configuration and Implementation on Riak

Riak is a distributed NoSQL database for storing data in the form of <key, value> objects. It organizes distributed nodes based on a P2P architecture with no central servers, and distributes data objects among different nodes using consistent hashing over the keys. Data are replicated to achieve high availability, and failures are handled through a "hinted handoff" mechanism among neighboring nodes.

Riak supports various mime types for the value of data objects, including JSON, plain text, Erlang binaries, etc. It provides a "Riak Search" module that can build distributed inverted indices on data objects for full-text search purposes. Users can use buckets to organize their data objects, and configure indexed fields on the bucket level. Besides basic inverted indexing functionality, Riak supports a special feature called "inline fields." If a field is specified as an "inline" field, its value will be attached to the document IDs in the related posting lists, as illustrated in Fig. 11.

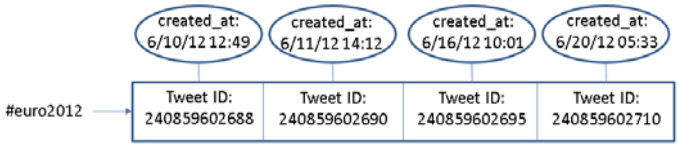

Fig. 11. An example of inline field (created_at) in Riak

Similar to our customized index tables in IndexedHBase, inline fields can be used to carry out an extra filtering operation to speed up queries involving multiple fields. However, they are different in two basic aspects:

For starters, support for inline fields is still an extension to traditional inverted indices, which means overhead such as frequency information and document scoring is still inevitable in Riak Search.

Secondly, customizable index structures are totally flexible in the sense that the structure of each index can be independently defined to contain any subset of fields from the original data. In contrast, if one field is defined as an inline field in Riak Search, its value will be attached to the posting lists of the indices of all the other indexed fields, regardless of whether it is useful. As a demonstration of this problem, the "sname index table" in Fig. 5 uses the creation time of user accounts as timestamps, while the "meme index table" uses creation time of tweets. Such flexibility is not achievable on Riak – users can attach similar information to the indices by specifying the creation time of user accounts and tweets as two separate inline fields, but that will obviously result in further unnecessary storage overhead.

In our tests, all 8 nodes of Bravo are used to construct a Riak ring. Each node runs Riak 1.2.1, using LevelDB as the storage backend. We create two different buckets to index data with different search schemas. Data replication level is set to 2 on both buckets. Within each bucket, <key, value> pairs are employed to directly store the tweet ID and JSON string of each tweet. The original JSON string is extended with an extra "memes" field, which contains all the hashtags, user-mentions, and URLs in the tweet, separated by a '\t' character. Riak search is enabled on both buckets to facilitate query evaluation, and the "user_id", "memes", "text", "retweeted_status_id", "user_screen_name", and "created_at" fields are indexed. Specifically, "created_at" is defined as a separate indexed field on one bucket, and as an "inline only" field on the other bucket, meaning that it does not have a separate index but is stored together with the entries of other indices to enable inline filtering for queries on the other fields.

Riak also provides a lightweight MapReduce framework for users to query the data by defining MapReduce functions in JavaScript. Furthermore, Riak supports MapReduce over the results of Riak Search. We use this feature to implement Truthy Queries, and Fig. 12 shows an example query implementation. When this query is submitted, Riak will first use the index on "memes" to find related tweet objects (as specified in the "input" field), then apply the map and reduce functions to these tweets (as defined in the "query" field) to get the final result.

```
{"inputs":{
            "bucket":"truthyTest201206",
            "query":"memes:'#euro2012'",
            "filter":"created_at:['2012-06-08' TO '2012-06-20']"
          },
 "query":[
            {"map":{ /* JavaScript function */ }
            },
            {"reduce":{ /* JavaScript function */ }
            }
         ]
}
```

Fig. 12. An example Truthy query implementation on Riak

4.3 Data Loading Performance

Historical Data Loading Performance

We use all the .josn.gz files of June 2012 to test the historical data loading performance of IndexedHBase and Riak. The total data size is 352GB. On IndexedHBase, a MapReduce job is launched for historical data loading, with each mapper processing one file. On Riak, all 30 files are distributed among 8 nodes of the cluster, so each node ends up with 3 or 4 files. Then an equal number of threads per node were created to load all the files concurrently to the bucket where "created_at" is configured as an inline field. Threads continue reading the next tweet, apply preprocessing with the "created_at" field and "memes" field, and then send the tweet as an object of mime type "JSON" to the Riak server, which will automatically index all the fields as defined in the search schema.

Table 3 summarizes the data loading time and loaded data size on both platforms. We can see that IndexedHBase is over 6 times faster than Riak in loading historical data, and uses significantly less disk space for storing the data. Considering the original file size of 352GB and a replication level of 2, the storage space overhead for index data on IndexedHBase is moderate.

Table 3. Historical data loading performance comparison

	Loading time (hours)	Loaded total data size (GB)	Loaded original data size (GB)	Loaded index data size (GB)
Riak	294.11	3258	2591	667
IndexedHBase	45.47	1167	955	212
Comparative ratio of Riak / IndexeHBase	6.47	2.79	2.71	3.15

We analyze these performance measurements below. By storing data with tables, IndexedHBase applies a certain degree of data model normalization, and thus avoids storing some redundant data. For example, many tweets in the original .json.gz files contain retweeted status, and many of them are retweeted multiple times. On IndexedHBase, even if a tweet is retweeted repeatedly, only one record is kept for it in the tweet table. On Riak, such a "popular" tweet will be stored within the JSON string of every corresponding retweet. The difference in loaded index data size clearly demonstrates the advantage of having a fully customizable indexing framework. By avoiding frequency and position information and only incorporating useful fields in the customized index tables, IndexedHBase saves 455GB of disk space in storing index data, which is more than 1/3 the total loaded data size of 1167GB. Also note that IndexedHBase compresses table data using Gzip, which generally provides a better compression ratio than Snappy used in Riak.

The difference in loaded data size explains only a part of the gap in total loading time. Two other major reasons are:

(1) On IndexedHBase, the loaders are responsible for generating both data tables and index tables. Therefore, the JSON string of each tweet is parsed only once when it is read from the .json.gz files and converted to table records. On Riak, since indexing is done by Riak servers instead of the loaders, the JSON string of each tweet is actually parsed twice – first by the loaders for preprocessing, and again by the server for extracting indexed fields.

(2) When building inverted indices, Riak not only uses more space to store the frequency and position information, but also spends more time collecting such information. Therefore, the customized index structures on IndexedHBase not only reduce disk storage requirement, but also lead to a faster loading speed.

Scalable Historical Data Loading on IndexedHBase

We test the scalability of historical data loading on IndexedHBase with the Alamo cluster of FutureGrid, which allows us to use a larger number of nodes through dynamic HPC jobs. In this test, we fix the dataset to files for two months, May 2012 and June 2012, and measure the total loading time at different cluster sizes with 16, 24, and 32 data nodes. The results are illustrated in Fig. 13. As shown here, when the cluster size is doubled from 16 to 32 data nodes, the total loading time drops from 142.72 hours to 93.22 hours, which implies a sub-linear scalability. Due to concurrent access from the mappers of the historical data loading jobs to HBase region servers, it is almost impossible to get an ideal linear scalability. Nonetheless, our results here clearly demonstrate that we can get more system throughput and faster data loading speed by adding more nodes to the cluster.

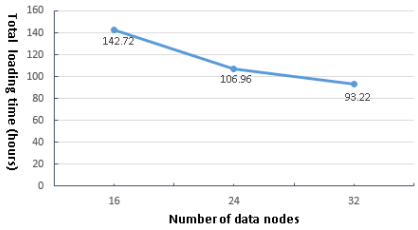

Fig. 13. Historical data loading scalability to cluster size

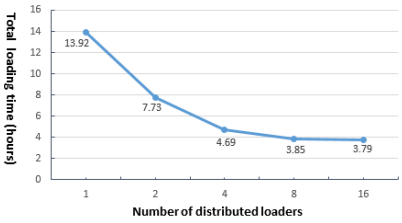

Fig. 14. Results for streaming data load-ing test

Streaming Data Loading Performance on IndexedHBase

The purpose of streaming data loading tests on IndexedHBase is to verify that it can provide enough data throughput to accommodate the growing data speed of

the Twitter streaming API. To test the performance of IndexedHBase for handling potential data rates even faster than the current streams, we design a simulation test using a recent .json.gz file for July 03, 2013. In this test, we vary the number of distributed streaming loaders and test the system data loading speed against different number of loaders. For each case, the whole 2013-07-03.json.gz file is split into the same number of fragments with equal size, which are then distributed evenly across all the nodes. One loader is started to process each fragment on the same node. The loader reads data from the stream of the local file fragment rather than Twitter streaming API. So this test measures how the system performs when each loader gets an extremely high incoming data rate that is equal to local disk I/O speed.

Fig. 14 shows the total loading time when the number of distributed loaders increases by powers of 2 from 1 to 16. Once again, concurrent access to the fixed number of HBase region servers results in a decrease in speed-up as the number of loaders is doubled each time. Specifically, the system throughput is almost saturated when we have 8 distributed loaders. For the case of 8 loaders, it takes 3.85 hours to load all 45,753,194 tweets for July 3, 2013, indicating the number of tweets that can be processed per day on 8 nodes is about 6 times the current daily data rate. Therefore, IndexedHBase can easily handle the streaming data load in Truthy. In the case of vastly accelerated data rates, we can always increase the system throughput by adding more nodes.

4.4 Query Evaluation Performance

Separate Index Structures vs. Customized Index Structures

As analyzed in Section 2, one major purpose of using customized index structures is to achieve lower query evaluation complexity than building traditional inverted indices on separate data fields. To verify this, we use a simple "get-tweets-with-meme" query to compare the performance of IndexedHBase and a solution using separate indices on the fields of memes and tweet creation time, which is implemented through the Riak bucket where "created_at" is defined as a separately indexed field instead of an inline field.

In this test, we load the first 4 days' data of June 2012 to both IndexedHBase and the Riak bucket and measure the query evaluation time with different memes and time windows. For memes, we choose "#usa", "#ff", and "@youtube", each contained in a different subset of tweets. "#ff" is a popular hashtag on Twitter, meaning "follow Friday". For each meme, we use 3 different time windows with a varied length of 1 to 3 hours. Queries in this test only return tweet IDs – they don't launch an extra MapReduce phase to get the tweets' content. Fig. 15 and 16 present the query evaluation time for each indexing strategy. As shown in the results,

using the customized meme index table, IndexedHBase not only achieves a query evaluation speed that is tens to hundreds of times faster, but also demonstrates a different pattern in query evaluation time. When separate meme index and creation time index are used, the query evaluation time mainly depends on the length of time window; the meme parameter has little impact. In contrast, when customized meme index is used, the query evaluation time mainly depends on the meme parameter. For the same meme, the evaluation time only increases marginally as the time window gets longer. These observations verify our theoretical analysis in Section 2.

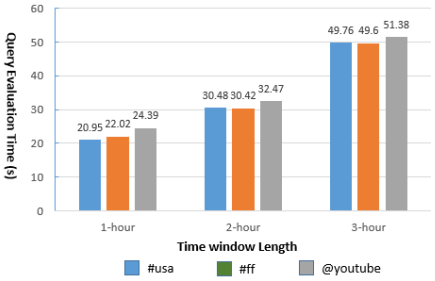

Fig. 15. Query evaluation time with separate meme and time indices (Riak)

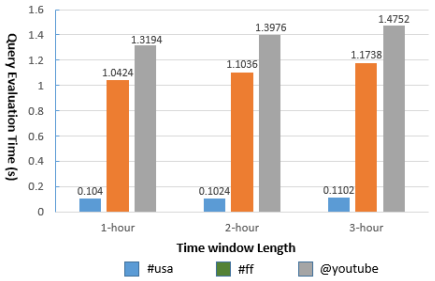

Fig. 16. Query evaluation time with customized meme index (IndexedHBase)

Query Evaluation Performance Comparison

This set of tests is designed to compare the performance of Riak and IndexedHBase for evaluating queries involving different number of tweets and different result sizes. Since using separate indices has proven inefficient on Riak, we choose to test the query implementation using "created_at" as an inline field. Queries are executed on both platforms against the data loaded in the historical data loading tests. For query parameters, we choose one popular meme "#euro2012", along with a time window with a varied length of 3 hours to 16 days. The start point of the time window is fixed at 2012-06-08T00:00:00, and the end point is correspondingly varied exponentially from 2012-06-08T02:59:59 to 2012-06-23T23:59:59. This time period covers a major part of the 2012 UEFA European Football Championship.

The queries can be grouped into 3 categories based on the manner in which they are evaluated on Riak and IndexedHBase:

(1) No MapReduce on either Riak or IndexedHBase

The "meme-post-count" query falls into this category. On IndexedHBase, query evaluation is done by simply going through the rows in meme index tables for each meme in the query and counting the number of qualified tweet IDs. In case of Riak, since there is no way to directly access the index data, this is accomplished

by issuing an HTTP query for each meme to fetch the "id" field of matched tweets.

Fig. 17 shows the query evaluation time on Riak and IndexedHBase. As the time window gets longer, the query evaluation time increases for both. However, the absolute evaluation time is much shorter for IndexedHBase, because Riak has to spend extra time to retrieve the "id" field.

(2) No MapReduce on IndexedHBase; MapReduce on Riak

"timestamp-count" falls under this category. Inferring from the schema of the meme index table, this query can also be evaluated by only accessing the index data on IndexedHBase. On Riak, it is implemented with MapReduce over Riak search results, where the MapReduce phase completes the timestamp counting based on the content of the related tweets. Fig. 18 shows the query evaluation time on both platforms. Since IndexedHBase does not need to analyze the content of the tweets at all, its query evaluation speed is orders of magnitude faster than Riak.

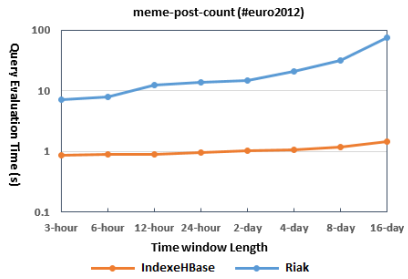

Fig. 17. Query evaluation time for "meme-post-count"

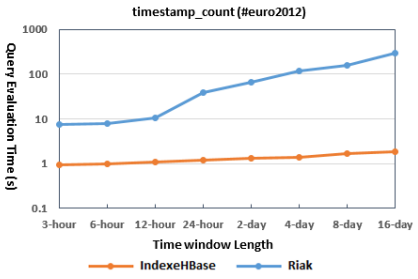

Fig. 18. Query evaluation time for "timestamp-count"

(3) MapReduce on both Riak and IndexedHBase

Most queries require a MapReduce phase on both Riak and IndexedHBase. Fig. 19 shows the query evaluation time for several of these. An obvious trend is that Riak is faster on queries involving a smaller number of related tweets and a small result set, but IndexedHBase is significantly faster on queries involving a larger number of related tweets and results. Table 4 lists the results sizes for "get-tweets-with-meme" (row 1) and "get-mention-edges" (row 2). The other queries have a similar pattern in result sizes.

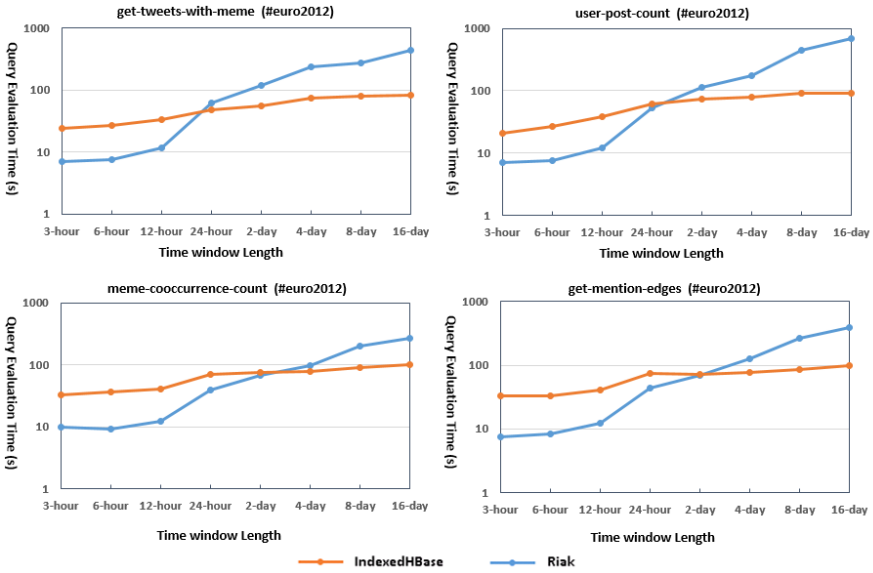

Fig. 19. Query evaluation time for queries requiring MapReduce on both platforms

Table 4. Result sizes for get-tweets-with-meme and get-mention-edges

3-hour	6-hour	12-hour	1-day	2-day	4-day	8-day	16-day
1287	2539	9342	87596	144575	234643	434043	606062
673	1367	4885	31330	49265	80547	145498	207783

The main reason for the performance difference observed is the different characteristics of the MapReduce framework on these two platforms. IndexedHBase relies on Hadoop MapReduce, which is designed for fault tolerant parallel processing of large batches of data. It implements the full semantics of the MapReduce computing model and applies a heavyweight initialization process for setting up the runtime environment on the worker nodes. Hadoop MapReduce uses local disks on worker nodes to save intermediate data and does grouping and sorting before passing them to reducers. A job can be configured to use zero or multiple reducers.

By comparison, the MapReduce framework on Riak is designed for lightweight use cases where users can write simple query logic with JavaScript and get them running on the data nodes quickly without a complicated initialization process. There is always only one reducer running for each MapReduce job. Intermediate data are transmitted directly from mappers to the reducer without being sorted or grouped. The reducer relies on its memory stack to store the whole list of intermediate data, and thus has the risk of crashing for large intermediate data sizes. Furthermore, the default timeout of the reducer is set to 5 seconds, and we actually

had to change this parameter in the source code and recompile Riak to get some of the above queries working.

Since most queries in Truthy use time windows at the level of weeks or months, IndexedHBase is more suitable for the queries above.

Improving Query Evaluation Performance with Modified Index Structures

One advantage of IndexedHBase is that it can accept dynamic changes to the index structures to achieve more efficient query evaluation. To verify this, we extend the meme index table to also include user IDs of tweets in the cell values, as illustrated in Fig. 20. Using this new index structure, IndexedHBase is able to evaluate the "user-post-count" query by only accessing index data.

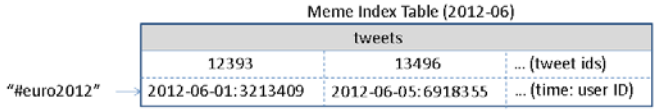

Fig. 20. Extended meme index table schema

We test this schema change on the tables for the 2012-06 dataset. We used the batch indexing mechanism of IndexedHBase to rebuild the meme index table, which took 3.89 hours. The table size increased from 14.23GB to 18.13GB, which is 27.4% larger. Fig. 21 illustrates the query evaluation time comparison. Obviously, query implementation using the new index structure is faster by more than an order of magnitude. In cases where "user-post-count" is frequently used, the query evaluation speed improvement is definitely worthy the storage overhead.

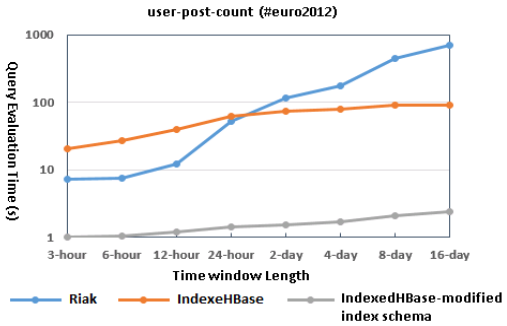

Fig. 21. Query evaluation time comparison with modified meme index table schema

5. Related Work

[15] discusses the temporal, spatial, and spatio-temporal challenges towards context-aware search and analysis on social media data. From this perspective, our current work on Truthy tries to address the temporal challenge in analysis scenarios, where the target is to apply analysis on all social updates that match a given query, instead of finding the most related ones. References in [15] provides a more complete list of related work about temporal and spatial analysis involving social data.

FluMapper [16] is a Cyber-GIS-enabled [2] environment designed to facilitate early detection and progression of influenza like illnesses (ILI). It provides an interactive map-based interface for flu-risk analysis based on filtering and near real-time processing of social updates collected from the Tiwtter streaming API. Compared with the more specialized application scenarios in FluMapper, Truthy collects a much larger dataset covering a broader spectrum of social activities, and provides a set of temporal queries that are generally applicable in many social data analysis scenarios.

Using indices to facilitate query evaluation has been a well-researched area in the field of database [1], and inverted indices [3] are specially designed for full-text search. Our customizable index structures share similar inspiration to multiple-column indices used in relational databases, but index a combination of full-text and primitive-type fields. Compared with traditional inverted indices, IndexedHBase provides more flexibility about what to use as keys and entries, so as to achieve more efficient query evaluation with less storage and computation overhead.

Google's Dremel [7] uses distributed columnar storage and multi-level serving trees to achieve efficient evaluation for aggregation queries on large datasets following a nested data model. Power Drill [12] provides even faster interactive query performance by exploring special caching and data skipping mechanisms on certain selected datasets in Google. Inspired by Dremel and Power Drill, we will consider splitting the tweet table into more column families for even better query evaluation performance. On the other hand, our customizable indexing strategies could also potentially help Dremel for handling aggregation queries with highly selective operations.

[13] proposes a distributed streaming data processing model that achieves fault-tolerant processing of streaming data by breaking continuous data streams into small batches and then processing them with existing fault-tolerant mechanisms used in batch processing frameworks such as MapReduce. Experience in [13] will be useful for our next step on developing a fault-tolerant streaming data processing framework for Truthy. However, since streaming data are mainly involved in the loading and indexing phase, simpler failure recovery mechanisms may be more suitable.

6. Conclusions and Future Work

This chapter describes our use case study about building an efficient and scalable storage platform, IndexedHBase, to support the Truthy social data observatory. As a result of our experimentation, we came to some interesting conclusions.

For starters, parallelization and indexing are key factors in addressing the challenges brought by the sheer data size and temporal queries of social data observatories. In particular, parallelization should be explored through every stage of data processing, including loading, indexing, and query evaluation.

Furthermore, index structures should be flexible and customizable, rather than static, to effectively take advantage of the special characteristics of the dataset and queries and achieve the best query evaluation performance at the cost of less storage and computation overhead. In order to achieve this, a general customizable indexing framework is necessary. To deal with the large size of intermediate data and results involved in the query evaluation process, complete and reliable parallel processing frameworks such as Hadoop MapReduce are needed. Lightweight frameworks like Riak MapReduce are not capable of handling queries involving analysis of large datasets.

To the best of our knowledge, IndexedHBase is a first in developing a totally customizable indexing framework on a distributed NoSQL database. Although our motivation originally came from social data observatories, the customizable indexing framework and two-phase query evaluation strategies are generally applicable in all kinds of applications. There are four major directions that we can work on in the future:

First, our current distributed streaming data loading strategy is simple and does not take failure recovery of data loaders into consideration. Building a fault tolerant streaming data loading mechanism with a more sophisticated data distribution framework will be a major part of our future work.

Secondly, we will try to further improve the efficiency of the parallel query evaluation strategy by taking data locality into consideration.

Thirdly, another major part of our future work is to add support for spatial queries by inferring and indexing spatial information contained in tweets. Thanks to the batch index building mechanism supported by IndexedHBase, adding spatial indices can be done efficient without completely reloading the original dataset.

Finally, we will try to integrate IndexedHBase with Hive [19] to provide a SQL-like data operation interface for Truthy users. How to make the customized index structures visible and useful to the query execution engine in Hive will be an interesting research issue to explore.

References

[1] G. Graefe (1993). *Query evaluation techniques for large databases.* ACM Computing Surveys (CSUR), 25(2): 73-169, 1993.

[2] S. Wang (2010). *A CyberGIS Framework for the Synthesis of Cyberinfrastructure, GIS, and Spatial Analysis.* Annals of the Association of American Geographers, 100(3): 535-557, 2010.

[3] J. Zobel, A. Moffat (2006). *Inverted files for text search engines.* ACM Computing Surveys, 38(2) - 6, 2006.

[4] K. McKelvey, F. Menczer (2013). *Design and Prototyping of a Social Media Observatory.* Proceedings of the 22nd international conference on World Wide Web companion, (WWW 2013).

[5] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber (2006). *Bigtable: A Distributed Storage System for Structured Data.* Proceedings of the 7th Symposium on Operating System Design and Implementation, (OSDI 2006).

[6] G. von Laszewski, G. Fox, F. Wang, A. Younge, A. Kulshrestha, G. Pike (2010). *Design of the FutureGrid Experiment Management Framework.* Proceedings of Gateway Computing Environments Workshop, (GCE 2010).

[7] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis (2010). *Dremel: Interactive Analysis of Web-Scale Datasets.* Proceedings of the 36th International Conference on Very Large Data Bases, (VLDB 2010).

[8] M. Conover, J. Ratkiewicz, M. Francisco, B. Goncalves, A. Flammini, F. Menczer (2011). *Political Polarization on Twitter.* Proceedings of the 5th International AAAI Conference on Weblogs and Social Media, (ICWSM 2011).

[9] E. Bakshy, J. Hofman, W. Mason, D. Watts (2011). *Everyone's an influencer: quantifying influence on Twitter.* Proceedings of the 4th ACM international conference on Web search and data mining, (WSDM 2011).

[10] L. Weng, A. Flammini, A. Vespignani, F. Menczer (2012). *Competition among memes in a world with limited attention.* Nature Sci. Rep., (2) 335, 2012.

[11] A. Choudhary, W. Hendrix, K. Lee, D. Palsetia, W. Liao (2012). *Social media evolution of the Egyptian revolution.* Communications of the ACM 55: 74–80, 2012.

[12] A. Hall , O. Bachmann , R. Büssow , S. Gănceanu , M. Nunkesser (2012). *Processing a Trillion Cells per Mouse Click.* Proceedings of the 38th International Conference on Very Large Data Bases, (VLDB 2012).

[13] M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica (2012). *Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters.* Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing, (HotCloud 2012).

[14] M. Conover, C. Davis, E. Ferrara, K. McKelvey, F. Menczer, A. Flammini (2013*). The Geospatial Characteristics of a Social Movement Communication Network.* PLoS ONE 8(3): e55957, 2013.

[15] L. Derczynski, B. Yang, C. Jensen (2013*). Towards Context-Aware Search and Analysis on Social Media Data.* Proceedings of the 16th International Conference on Extending Database Technology, (EDBT 2013).

[16] A. Padmanabhan, S. Wang, G. Cao, M. Hwang, Y. Zhao, Z. Zhang, Y. Gao (2013). *FluMapper: An Interactive CyberGIS Environment for Massive Location-based Social Media Data Analysis.* Proceedings of Extreme Science and Engineering Discovery Environment: Gateway to Discovery, (XSEDE 2013).

[17] Apache Hadoop. http://hadoop.apache.org/.

[18] Apache HBase. http://hbase.apache.org/.

[19] Apache Hive. http://hive.apache.org/.

[20] DataStax. http://www.datastax.com/.

[21] Riak. http://basho.com/riak/.

[22] Twitter Streaming API. https://dev.twitter.com/docs/streaming-apis.