# Supporting a Social Media Observatory with Customizable Index Structures — Architecture and Performance

**Xiaoming Gao[1], Evan Roth[2], Karissa McKelvey[1], Clayton Davis[1], Andrew Younge[1], Emilio Ferrara[1], Filippo Menczer[1], Judy Qiu[1]**

[1]School of Informatics and Computing, Indiana University
[2]Department of Computer Science & Information Technology, University of the District of Columbia

**Abstract**. The intensive research activity in analysis of social media and micro-blogging data in recent years suggests the necessity and great potential of platforms that can efficiently store, query, analyze, and visualize social media data. To support these "social media observatories" effectively, a storage platform must satisfy special requirements for loading and storage of multi-terabyte datasets, as well as efficient evaluation of queries involving analysis of the text of millions of social updates. Traditional inverted indexing techniques do not meet such requirements. As a solution, we propose a general indexing framework, IndexedHBase, to build specially customized index structures for facilitating efficient queries on an HBase distributed data storage system. IndexedHBase is used to support a social media observatory that collects and analyzes data obtained through the Twitter streaming API. We develop a parallel query evaluation strategy that can explore the customized index structures efficiently, and test it on a set of typical social media data queries. 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 six times faster than Riak and is significantly more efficient in evaluating queries involving large result sets.

**Keywords:** Social Media Observatory, Distributed NoSQL Database, Customizable Index Structure, Parallel Query Evaluation, IndexedHBase

## 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, the usage of indices to facilitate query evaluation has been a well-researched topic in the area of databases [12], and inverted indices [23] are specially designed for full-text search. A basic idea is to first build index data structures through a full scan of data and documents and then facilitate fast access to the data via indices to achieve highly optimized search performance.

Beyond these system features, it is a challenge to enable real-time search and efficient analysis over a broader spectrum of social media data scenarios. For example, Derczynski et al. [10] discussed the temporal and spatial challenges in context-aware search and analysis on social media data. Padmanabhan et al. presented FluMapper [16], an interactive map-based interface for flu-risk analysis using near real-time processing of social updates collected from the Twitter streaming API [19]. As an additional scenario within this line of research, we utilize Truthy (http://truthy.indiana.edu) [14], a public social media observatory that analyzes and visualizes information diffusion on Twitter. Research performed on the data collected by this system covers a broad spectrum of social activities, including political polarization [6,xx], congressional elections [11,xx], protest events [7,8], and the spread of misinformation [xx,xx]. Truthy has also been instrumental in shedding light on communication dynamics such as user attention allocation [21] and social link creation [4]. This platform processes and analyzes some general entities and relationship, contained in its large-scale social dataset, such as *tweets*, *users*, *hashtags*, *retweets*, and *user-mentions* during specific time windows of social events. Truthy consumes a stream that includes a sample of public tweets. Currently, the total size of historical data collected continuously by the system since August 2010 is approximately 10 Terabytes (stored in compressed JSON format). At the time of this writing, the data rate of the Twitter streaming API is in the range of 45-50 million tweets per day, leading to a growth of approximately 20GB per day in the total data size.

This chapter describes our research towards building an efficient and scalable storage platform for this large set of social microblogging data collected by the Truthy system. Many existing NoSQL databases, such as Solandra (now known as DataStax) [9] and Riak [18], support distributed inverted indices [23] 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, and thus they are not suitable for handling social media data queries. For example, the issue of how to efficiently evaluate temporal queries involving text search on hundreds of millions of social updates remains a challenge. As a possible solution, we propose IndexedHBase, a general, customizable indexing framework. Current implementation is based on HBase [2] as the underlying storage platform. IndexedHBase provides users with the added flexibility to define the most suitable index structures to facilitate their queries. Using Hadoop MapReduce [1] we implement a parallel query evaluation strategy that can make the best use of the customized index structures to achieve efficient evaluation of social media data

queries typical for an application such as Truthy. We develop efficient data loading strategies that can accommodate fast loading of historical files as well as fast processing of streaming data from real-time tweets. We evaluate the performance of IndexedHBase on FutureGrid [20]. Our preliminary results show that, compared with Riak, IndexedHBase is significantly more efficient. It is six times faster for data loading, while requiring much less storage. Furthermore, it is clearly more efficient in evaluating queries derived from large result sets.

The rest of this chapter is organized as follows. Section 2 analyzes the characteristics of data and queries. Section 3 describes the architecture of IndexedHBase and explains the design and implementation of its data loading, indexing, and query evaluation strategies. Section 4 evaluates the performance of IndexedHBase and compares it with Riak. Section 5 discusses related work. Section 6 concludes and describes our future work.

## 2. Data and Query Patterns

The entire dataset consists of two parts: historical data in .json.gz files, and real-time data collected from the Twitter streaming API. 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 original tweet content 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 detailed information, such as the ID of the mentioned user and the expanded URLs.

In social network analysis, the concept of "*meme*" is often used to represent a set of related posts corresponding to a specific discussion topic, communication channel, or information source shared by users on platforms such as Twitter. Memes can be identified through elements contained in the text of tweets, such as *keywords*, *hashtags* (e.g., #euro2012), *user-mentions* (e.g., @youtube), and *URLs*. Our social media observatory, 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 ID or content of tweets created during a given time window from their text or user information, including:

***get-tweets-with-meme*** (*memes, time_window*)
***get-tweets-with-text*** (*keywords, 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]. The *memes* parameter is given as a list of hashtags, user-

mentions, or URLs; *memes* and *keywords* may contain wildcards, e.g., "#occupy*" will match all tweets containing hashtags starting with "#occupy."


```
{
    "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**

The second subset of queries extract needed information from the tweets returned by queries in the first subset. These include ***timestamp-count***, ***user-post-count***, ***meme-post-count***, ***meme-cooccurrence-count***, ***get-retweet-edges***, and ***get-mention-edges***. Here for example, *user-post-count* returns the number of posts about a given meme by each user. Each "edge" has three 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. 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, in the time window [2012-06-01, 2012-06-20] there are over 600 million tweets, while the number of tweets containing the most popular meme "@youtube" is less than two million, which is smaller by more than two orders of magnitude.

Traditional distributed inverted indices [23], supported by many existing distributed NoSQL database systems such as Solandra (DataStax) [9] and Riak [18], do not provide the most efficient solution to locate relevant tweets by their

text content. One reason is that 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 comprising 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, social media data queries 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 regarding frequency and position are extra overhead for the storage of inverted indices, and relevance scoring is unnecessary in the query evaluation process. The query evaluation performance can be further improved by removing these items from traditional inverted indices.

Secondly, social media queries do not favor query execution plans using traditional inverted indices. Fig. 2 illustrates a typical query execution plan for *get-tweets-with-meme*, using two separate indices on memes and tweet creation time. This plan uses the meme index to find the IDs of all tweets containing the given memes and utilizes the time index to find the set of tweet IDs within the given time window, finally computing 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 large social media and microblogging datasets, 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 [23] may be utilized to quickly return the top K most relevant results without evaluating all the related documents. However, such optimizations are not applicable to our social media observatory. Furthermore, in case of a 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 content of relevant tweets. However, valuable time is still wasted in checking irrelevant tweets falling out of the given time window. The query evaluation performance can be further improved if the unnecessary scanning cost can be avoided.

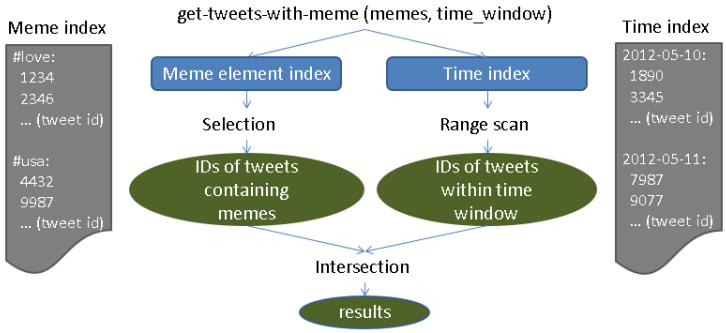


**Fig. 2. A typical query execution plan using indices on meme and creation time**

We propose using a customized index structure in IndexedHBase, as illustrated in Fig. 3. It 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*

HBase is used to host the entire dataset and related indices with two sets of tables: data tables for the original data, and index tables containing customized index structures for query evaluation (see Fig. 4). The customized indexing framework supports two mechanisms for building index tables: online indexing that indexes data upon upload to the tables, and batch indexing for building new index structures from existing data tables. Two data loading strategies are implemented

for historical and streaming data. The parallel query evaluation strategy provides efficient evaluation mechanisms for all queries, and is used by upper-level applications, such as Truthy, to generate various statistics and visualizations.

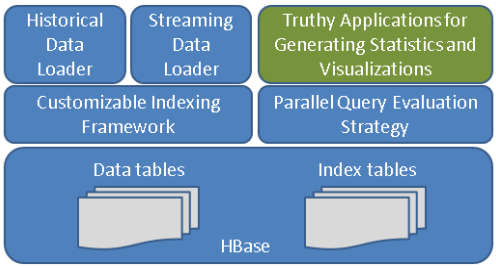


**Fig. 4. System Architecture of IndexedHBase**

## *3.2 Customizable Indexing Framework*

### Table Schemas on HBase

Working off the extendible "BigTable" data model [5], we design the table schemas in Fig. 5. Tables are managed in units of months. This has two benefits. First, the loading of streaming data only changes the tables relative to the current month. Secondly, during query evaluations, the amount of index data and original data scanned is limited by the time window parameter.

Some details need to be clarified before proceeding further. Each table contains only one column family, e.g. "details" or "tweets". The user table uses a concatenation of user ID and tweet ID as the row key, because analysis benefits from tracking changes in a tweet's user metadata. For example, a user can change profile information, which can give insights into her behavior. Another meme index table is created for the included *hashtags*, *user-mentions*, and *URLs*. This is because some special cases, such as expandable URLs, cannot be handled properly by the text index. The memes are used as row keys, each followed by a different number of columns, named after the IDs of tweets containing the corresponding meme. The timestamp of the cell value marks the tweet creation time (Fig. 5).

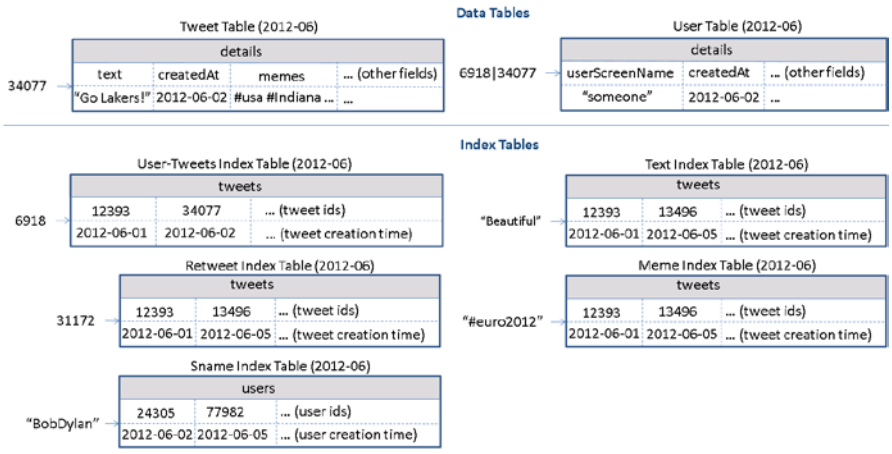


**Fig. 5. Table schemas used in IndexedHBase for Twitter data**

Using HBase tables for customized index has several advantages. The data model of HBase can scale out horizontally for distributed index structure and embed additional information within the columns. Since the data access pattern in social media analysis is "write-once-read-many", IndexedHBase builds a separate table for each index structure for easy update and access. Rows in the tables are sorted by row keys, facilitating prefix queries through range scans over index tables. Using Hadoop MapReduce, the framework can generate efficient parallel analysis on the index data, such as meme popularity distribution [21].

## Customizable Indexer Implementation

IndexedHBase implements a customizable indexer library, shown in Fig. 6, to generate index table records automatically according to the configuration file and insert them upon the client application's request.

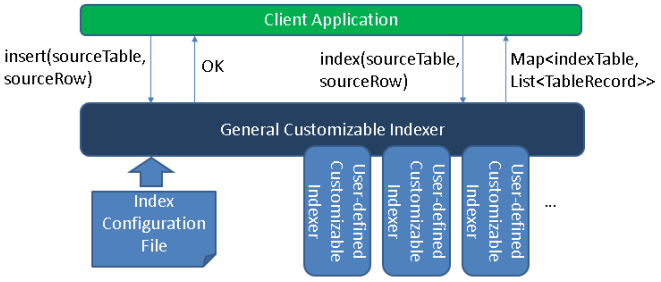


**Fig. 6. Components of customizable indexer**

Fig. 7 gives an example of the index configuration file in XML format containing multiple “index-config” elements that hold the mapping information between one source table and one index table. This element can flexibly define how to generate records for the index table off a given row from the source table. For more complicated index structures, users can implement a customizable indexer and use it 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**

Both general and user-defined indexers must 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, and the value is a list of that table’s records.

Upon initialization, the general customizable indexer reads the index configuration file from the user. If a user-defined indexer class is specified, a corresponding indexer instance will be created. When *index()* is invoked during runtime, all related “index-config” elements are used to generate records for each 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.

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

**Fig. 8. Pseudocode for the “CustomizableIndexer” interface**

### Online Indexing Mechanism and Batch Indexing Mechanism

IndexedHBase provides two means of indexing data: online and batch. The online 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 into a source table. The indexer will first insert the given row into the source table and then generate index table records for this row by invoking *index()* and insert them into 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 and 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. Fig. 9 shows the architecture of the streaming data loading strategy, where one or more distributed loaders are running concurrently and are connected to the same stream using the Twitter streaming API. Each loader is assigned a unique ID and works as a client application to the general customizable indexer. Upon receiving a tweet JSON string, 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 the relevant index tables.

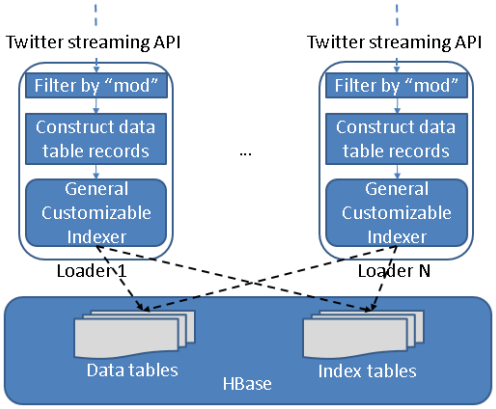


**Fig. 9. Streaming data loading strategy**

The historical data loading strategy is implemented as a MapReduce program. One separate job is launched to load the historical files for each month, and multiple jobs can be running simultaneously. Each job starts multiple mappers in parallel, each responsible for loading one file. At running time, each line in the .json.gz file is given to the mapper as one input, which contains the string of one tweet. The mapper first creates records for the tweet table and user table 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.

## *3.4 Parallel Query Evaluation Strategy*

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, and saves them in a series of files containing a fixed number (e.g., 30,000) 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 a particular 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 to finally generate the total tweet count of each user ID.

Two aspects of the query evaluation strategy deserve further discussion. First, as described in Section 2, prefix queries can be constructed by using parameters such as "#occupy*". IndexedHBase provides two options for getting the related

tweet IDs in the first phase. One is simply to complete a sequential range scan of rows in the corresponding index tables. The other is to use a MapReduce program to complete parallel scans over the range of rows. The latter option is only faster for parameters covering a large range spanning multiple regions of the index table.

Second, the number of tweet IDs in each file 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 a mapper decreases while the total task scheduling overhead increases. The optimal number depends on the total number of related tweets and the amount of resources available in the infrastructure. We set the default value of this number to 30,000 and leave it configurable by the user. Future work will explore automatic optimization.

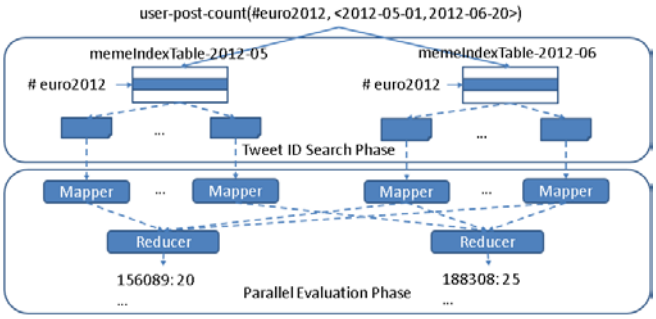


**Fig. 10. Two-phase parallel evaluation process for an example *user-post-count* query**

## 4. Performance Evaluation Results and Comparison with Riak

### *4.1 Testing Environment Configuration*

We use eight 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 seven nodes are used to host HDFS datanodes, Hadoop tasktrackers, and HBase region servers. The data replication level is set to two on HDFS. The configuration details of Riak will be given in Section 4.2. In addition to 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. Software configuration of Alamo is mostly the same as Bravo.

**Table 1. Per-node configuration on Bravo and Alamo Clusters**

| Cluster | CPU | RAM | Hard Disk | Network |
|---|---|---|---|---|
| Bravo | 8 * 2.40GHz (Intel Xeon E5620) | 192G | 2T | 40Gb InfiniBand |
| Alamo | 8 * 2.66GHz (Intel Xeon X5550) | 12G | 500G | 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 uses a P2P architecture to organize the distributed nodes and distributes data objects among them using consistent hashing. Data are replicated to achieve high availability, and failures are handled by a handoff mechanism among neighboring nodes. A "Riak Search" module 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. 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 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. First, inline fields are an extension of traditional inverted indices, which means overhead such as frequency information and document scoring still exist in Riak Search. Second, 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 on Riak, its value will be attached to the posting lists of the indices of all indexed fields, regardless of whether it is useful. As an example, 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.

In our tests, all eight 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. The data replication level is set to two on both buckets. The tweet ID and JSON string of each tweet

are directly stored into <key, value> pairs. The original JSON string is extended with an extra "memes" field, which contains all the hashtags, user-mentions, and URLs in the tweet, separated tab characters. Riak Search is enabled on both buckets, 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 indices of other fields.

Riak 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 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 query implementation on Riak**

## *4.3 Data Loading Performance*

### Historical Data Loading Performance

We use all the .json.gz files from June 2012 to test the historical data loading performance of IndexedHBase and Riak. The total data size is 352GB. With IndexedHBase, a MapReduce job is launched for historical data loading, with each mapper processing one file. With Riak, all 30 files are distributed among eight nodes of the cluster, so each node ends up with three or four 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" and "memes" field, and then send the tweet to the Riak server for indexing and insertion.

**Table 2. 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 |
| Riak / IndexedHBase | 6.47 | 2.79 | 2.71 | 3.15 |

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

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. With IndexedHBase, even if a tweet is retweeted repeatedly, only one record is kept for it in the tweet table. With 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 a fully customizable indexing framework. By avoiding frequency and position information and only incorporating useful fields in the 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 on Riak.

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

(1) The loaders of IndexedHBase 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 the other hand, Riak uses servers for its indexing and so each JSON string is actually parsed twice – first by the loaders for preprocessing, and again by the server for indexing;

(2) When building inverted indices, Riak not only uses more space to store the frequency and position information, but also spends more time collecting them.

### Scalable Historical Data Loading on IndexedHBase

We test the scalability of historical data loading on IndexedHBase with the Alamo cluster of FutureGrid. In this test we take a dataset for two months, May and June 2012, and measure the total loading time with different cluster sizes. The results

are illustrated in Fig. 13. 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 coming from the concurrent access from mappers of the loading jobs to HBase region servers. Nonetheless, these results clearly demonstrate that we get more system throughput and faster data loading speed by adding more nodes to the cluster.

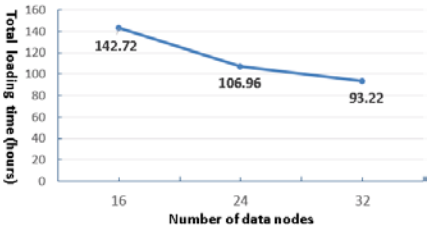


**Fig. 13. Historical data loading scalability to cluster size**

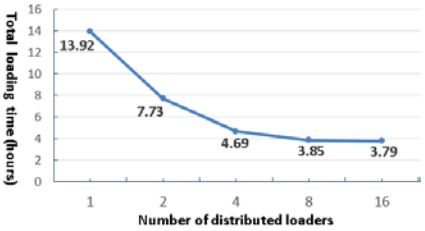


**Fig. 14. Results for streaming data loading test**

### Streaming Data Loading Performance on IndexedHBase

The purpose of streaming data loading tests is to verify that IndexedHBase can provide enough 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 from July 3, 2013. We vary the number of distributed streaming loaders and test the corresponding system data loading speed. For each case, the whole file is evenly split into the same number of fragments as the loaders and then distributed across all the nodes. One loader is started to process each fragment. The loader reads data from the stream of the local file fragment rather than from the Twitter streaming API. So this test measures how the system performs when each loader gets an extremely high 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 two from one to 16. Once again, concurrent access to HBase region servers results in a decrease in speed-up as the number of loaders is doubled each time. The system throughput is almost saturated when we have eight distributed loaders. For the case of eight loaders, it takes 3.85 hours to load all 45,753,194 tweets, indicating the number of tweets that can be processed per day on eight nodes is about six times the current daily data rate. Therefore, IndexedHBase can easily handle a high-volume stream of social media data. In the case of vastly accelerated data rates, as would be the case for the Twitter *firehose* (a stream of *all* public tweets), one could increase the system throughput by adding more nodes.

## *4.4 Query Evaluation Performance*

**Separate Index Structures vs. Customized Index Structures**

As discussed in Section 2, one major purpose of using customized index structures is to achieve lower query evaluation complexity compared to 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 with 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.

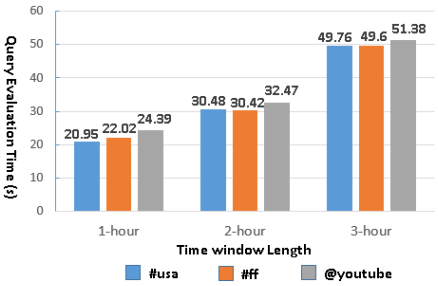


**Fig. 15. Query evaluation time with separate meme and time indices (Riak)**

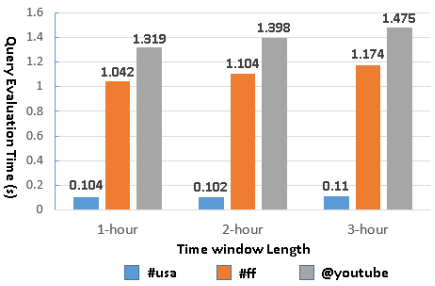


**Fig. 16. Query evaluation time with customized meme index (IndexedHBase)**

In this test we load four days' data 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. The "#ff" hashtag is a popular meme for "Follow Friday." For each meme, we use three different time windows with a length between one and three hours. Queries in this test only return tweet IDs – they don't launch an extra MapReduce phase to get the content. Figs. 15 and 16 present the query execution time for each indexing strategy. As shown in the plots, 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 time index are used, the query evaluation time mainly depends on the length of time window; the meme parameter has little impact. In contrast, using a customized meme index, 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 confirm our theoretical analysis in Section 2.

**Query Evaluation Performance Comparison**

This set of tests is designed to compare the performance of Riak and IndexedHBase for evaluating queries involving different numbers 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 the popular meme "#euro2012," along with a time window with a length varied from three 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 three 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 given meme and counting the number of qualified tweet IDs. In the 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**

The *timestamp-count* query belongs to 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.

**(3) MapReduce on both Riak and IndexedHBase**

Most queries require a MapReduce phase on both Riak and IndexedHBase. Fig. 19 shows the evaluation time for several of them. An obvious trend is that Riak is faster on queries involving a smaller number of related tweets, but IndexedHBase is significantly faster on queries involving a larger number of related tweets and results. Fig. 20 lists the results sizes for two of the queries. The other queries have a similar pattern.

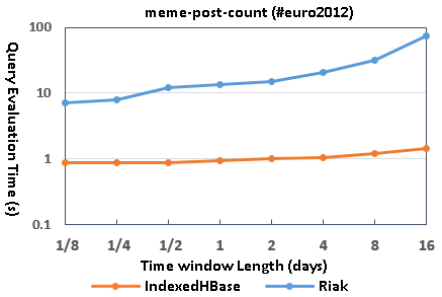


**Fig. 17. Query evaluation time for *meme-post-count***

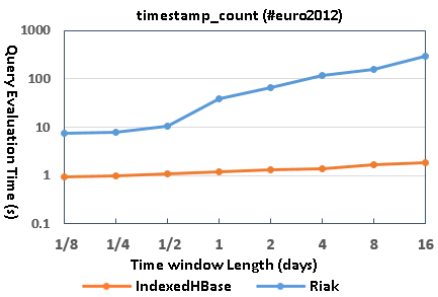


**Fig. 18. Query evaluation time for *timestamp-count***

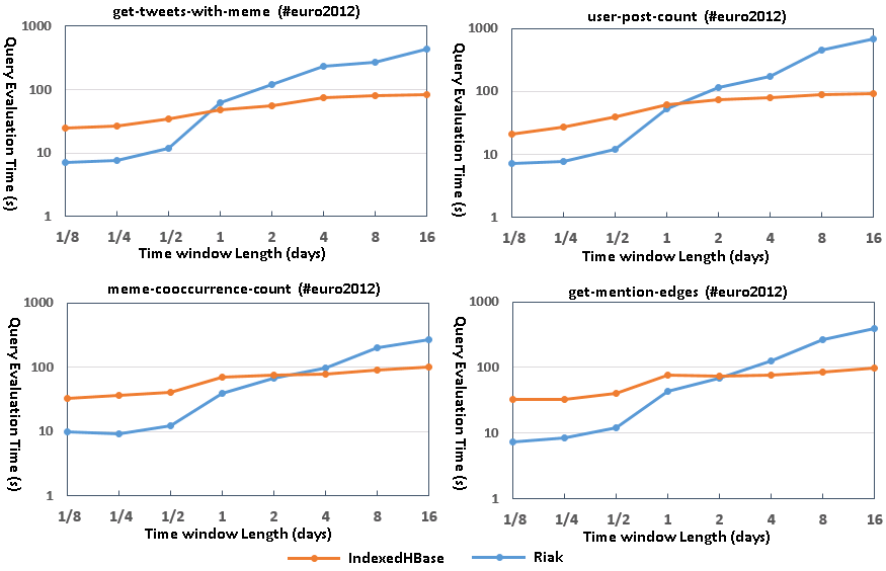


**Fig. 19. Query evaluation time for queries requiring MapReduce on both platforms**

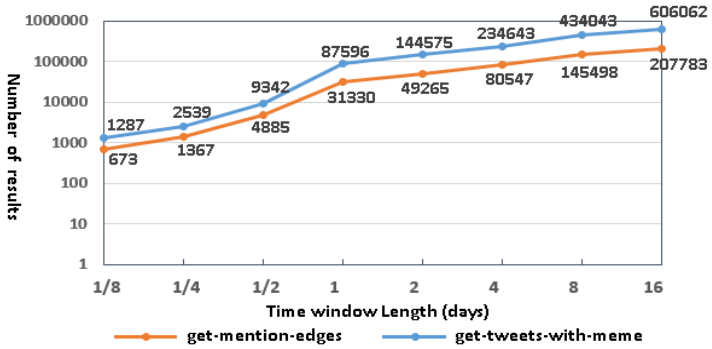


**Fig. 20. Result sizes for get-tweets-with-meme (top row) and get-mention-edges (bottom row) queries**

The main reason for the observed performance difference 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 comprehensive initialization process for setting up the runtime environment on the worker nodes. Hadoop MapReduce uses 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. Since most social media queries use time windows at the level of weeks or months, IndexedHBase can handle these long time period queries well.

The MapReduce framework on Riak, on the other hand, 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 has a default timeout of only five seconds. Therefore, Riak MapReduce is not suitable for processing the large datasets produced by queries corresponding to long time periods.

### Improving Query Evaluation Performance with Modified Index Structures

IndexedHBase accepts dynamic changes to the index structures for 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. 21. Using this new index structure, IndexedHBase is able to evaluate the *user-post-count* query by only accessing index data.

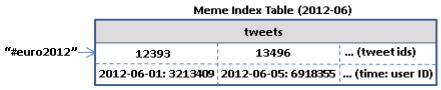


**Fig. 21. Extended meme index table schema**

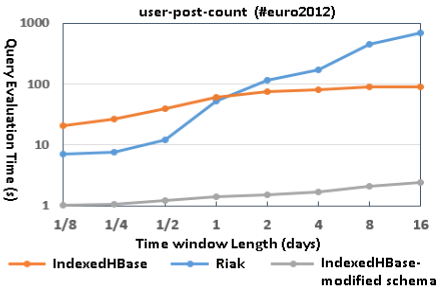


**Fig. 22. Query evaluation time modified meme index table schema**

We use the batch indexing mechanism of IndexedHBase to rebuild the meme index table, which takes 3.89 hours. The table size increases from 14.23GB to 18.13GB, which is 27.4% larger. Fig. 22 illustrates the query evaluation time comparison. The query with the new index structure is faster by more than an order of magnitude. In cases where *user-post-count* is frequently used, the query speed improvement is clearly worth the additional storage required.

## 5. Related Work

IndexedHBase aims to address the temporal challenge in social media analytics scenarios. Derczynski et al. [10] provide a more complete list of related work about temporal and spatial queries involving social data. 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 [23], IndexedHBase provides more flexibility about what fields to use as keys and entries, so as to achieve more efficient query evaluation with less storage and computation overhead.

Solandra (DataStax) [9] and Riak [18] are two typical NoSQL database systems that support distributed inverted indices for full-text search. Specifically, Solandra is an offshoot of Cassandra, which uses a similar data model to HBase. Comparable to Riak, Cassandra also employs P2P architecture to support scalable data storage and relies on data replication to achieve fault-tolerance. As discussed in Section 2, inverted indices on Solandra and Riak are designed for text retrieval applications, making them unsuitable for social media analytics.

Google's Dremel [15] achieves efficient evaluation of aggregation queries on large-scale nested datasets by using distributed columnar storage and multi-level serving trees. Power Drill [13] explores special caching and data skipping mechanisms to provide even faster interactive query performance for certain selected datasets. Percolator [17] replaces batch indexing system with incremental processing for Google search. 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.

Zaharia et al. [22] propose 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. This idea of discretized streams will be useful for our next step of developing a fault-tolerant streaming data processing framework. 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 studies an efficient and scalable storage platform supporting a large Twitter stream that powers the Truthy system [14], a public social media observatory. Our experimentation with IndexedHBase led to interesting conclusions of general significance. Parallelization and indexing are key factors in addressing the sheer data size and temporal queries of social data observatories. Parallelism in particular requires attention to every stage of data processing. Furthermore, a general customizable indexing framework is necessary. Index structures should be flexible, rather than static, to facilitate special characteristics of the dataset and queries, where optimal query evaluation performance is achieved at lower cost in storage and computation overhead. Reliable parallel processing frameworks such as Hadoop MapReduce can handle large intermediate data and results involved in the query evaluation process.

To the best of our knowledge, IndexedHBase is the first effort in developing a totally customizable indexing framework on a distributed NoSQL database. In the future we will add failure recovery to the distributed streaming data loading strategy. The efficiency of parallel query evaluation can be further improved with data locality considerations. Spatial queries will be supported by inferring and indexing spatial information contained in tweets. Thanks to the batch index building mechanism in IndexedHBase, adding spatial indices can be done efficiently without completely reloading the original dataset. Finally, we will integrate IndexedHBase with Hive [3] to provide a SQL-like data operation interface for easy implementation in social media observatories such as Truthy.

**Acknowledgments**

We would like to thank Onur Varol, Alessandro Flammini, Geoffrey Fox, and other colleagues and members of the Center for Complex Networks and Systems Research (cnets.indiana.edu) at Indiana University for helpful discussions and

contributions to the Truthy Project and the present work. We gratefully acknowledge partial support from the National Science Foundation (grant CCF-1101743), DARPA (grant W911NF-12-1-0037), and the J. S. McDonnell Foundation. We would also like to personally thank Koji Tanaka and the rest of the FutureGrid team for their continued help. FutureGrid is supported by National Science Foundation (NSF) under Grant No. 0910812 to Indiana University for "An Experimental, High-Performance Grid Test-bed." IndexedHBased is in part supported by National Science Foundation grant OCI-1149432 for CAREER Award.

## References

[1] Apache Hadoop. http://hadoop.apache.org/.
[2] Apache HBase. http://hbase.apache.org/.
[3] Apache Hive. http://hive.apache.org/.
[4] Lilian Weng, Jacob Ratkiewicz, Nicola Perra, Bruno Gonçalves, Carlos Castillo, Francesco Bonchi, Rossano Schifanella, Filippo Menczer and Alessandro Flammini: *The role of information diffusion in the evolution of social networks*. Proc. 19th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), 2013.
[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R. (2006). *Bigtable: A Distributed Storage System for Structured Data*. Proceedings of the 7th Symposium on Operating System Design and Implementation, (OSDI 2006).
[6] Conover, M., Ratkiewicz, J., Francisco, M., Goncalves, B., Flammini, A., Menczer, F. (2011). *Political Polarization on Twitter*. Proceedings of the 5th International AAAI Conference on Weblogs and Social Media, (ICWSM 2011).
[7] Conover, M., Davis, C., Ferrara, E., McKelvey, K., Menczer, F., Flammini, A. (2013). *The Geospatial Characteristics of a Social Movement Communication Network*. PLoS ONE 8(3): e55957.
[8] Conover, M., Ferrara, E., Menczer, F., Flammini, A. (2013). *The Digital Evolution of Occupy Wall Street*. PloS ONE, 8(5), e64679.
[9] DataStax. http://www.datastax.com/.
[10] Derczynski, L., Yang, B., Jensen, C. (2013). *Towards Context-Aware Search and Analysis on Social Media Data*. Proceedings of the 16th International Conference on Extending Database Technology, (EDBT 2013).
[11] DiGrazia, J., McKelvey, K., Bollen, J., Rojas, F. (2013). *More Tweets, More Votes: Social Media as an Indicator of Political Behavior*. Working paper, Indiana University.
[12] Graefe, G. (1993). *Query evaluation techniques for large databases*. ACM Computing Surveys (CSUR), 25(2): 73-169.
[13] Hall, A., Bachmann, O., Büssow, R., Gănceanu, S., Nunkesser, M. (2012). *Processing a Trillion Cells per Mouse Click*. Proceedings of the 38th International Conference on Very Large Data Bases, (VLDB 2012).
[14] McKelvey, K., Menczer, F. (2013). *Design and Prototyping of a Social Media Observatory*. Proceedings of the 22nd international conference on World Wide Web companion, (WWW 2013).
[15] Melnik, S., Gubarev, A., Long, J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T. (2010). *Dremel: Interactive Analysis of Web-Scale Datasets*. Proceedings of the 36th International Conference on Very Large Data Bases, (VLDB 2010).

[16] Padmanabhan, A., Wang, S., Cao, G., Hwang, M., Zhao, Y., Zhang, Z., Gao, Y. (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] Peng, D., Dabek, F. (2010). *Large-scale Incremental Processing Using Distributed Transactions and Notifications*. Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, (USENIX 2010).

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

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

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

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

[22] Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I. (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).

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