REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
	Technical Report	-

4. TITLE AND SUBTITLE

Comparing IndexedHBase and Riak for Serving Truthy: Performance of Data Loading and Query Evaluation

5a. CONTRACT NUMBER
W911NF-12-1-0037

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHORS

Xiaoming Gao, Judy Qiu

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES

Indiana University at Bloomington
Trustees of Indiana University
509 E 3RD ST
Bloomington, IN 47401 -3654

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
61766-NS-DRP.32

12. DISTRIBUTION AVAILIBILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT

This report summarizes our performance evaluation of IndexedHBase and Riak in their support of the Truthy system. We used data from June 2012 to test the performance of these platforms in two aspects: data loading and query evaluation. The total data size is 352GB. Queries tested can be found online. For IndexedHBase, we conducted scalability tests for data loading using more nodes on Alamo in FutureGrid (see appendix) while basic tests where done on 8 nodes of the FutureGrid

15. SUBJECT TERMS

performance evaluation, distributed database, noSQL, HBase, indexing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Alessandro Flammini
UU	UU	UU	UU		19b. TELEPHONE NUMBER 812-856-1830

Standard Form 298 (Rev 8/98)
Prescribed by ANSI Std. Z39.18

Report Title

Comparing IndexedHBase and Riak for Serving Truthy: Performance of Data Loading and Query Evaluation

ABSTRACT

This report summarizes our performance evaluation of IndexedHBase and Riak in their support of the Truthy system. We used data from June 2012 to test the performance of these platforms in two aspects: data loading and query evaluation. The total data size is 352GB. Queries tested can be found online. For IndexedHBase, we conducted scalability tests for data loading using more nodes on Alamo in FutureGrid (see appendix) while basic tests where done on 8 nodes of the FutureGrid Bravo cluster.

Comparing IndexedHBase and Riak for Serving Truthy

Performance of Data Loading and Query Evaluation

Xiaoming Gao and Judy Qiu
August 1, 2013

Digital Science Center, Indiana University
http://salsahpc.indiana.edu/

1.Introduction

This report summarizes our performance evaluation of IndexedHBase and Riak in their support of the Truthy system. We used data from June 2012 to test the performance of these platforms in two aspects: data loading and query evaluation. The total data size is 352GB. Queries tested can be found online. For IndexedHBase, we conducted scalability tests for data loading using more nodes on Alamo in FutureGrid (see appendix) while basic tests where done on 8 nodes of the FutureGrid Bravo cluster.

Compared with Riak, IndexedHBase provides a 6 or more times faster data loading speed for historical data. On the Bravo cluster with 8 nodes, IndexedHBase can load one month's data in less than 2 days (45.47 hours) and handle streaming data at a rate that is 6 times faster than the current daily incoming data rate. Riak is better at query evaluation for queries involving short time windows (shorter than 1 day) and small result sizes, while IndexedHBase is significantly faster for queries with time windows at the week or month level. For queries that can be completed by only accessing the index data, IndexedHBase can finish within seconds. For queries that need an extra MapReduce phase, IndexedHBase can complete evaluation processes involving analysis of millions of tweets within minutes. Adding improvements such as use of the lightweight Twister MapReduce, IndexedHBase will be able to finish queries with smaller result sizes with comparable speed to Riak so that it is always 1-6 times faster than Riak.

2.Methods

2.1 IndexedHBase

IndexedHBase (http://salsaproj.indiana.edu/IndexedHBase) is a research project that extends the HBase system with dynamic inverted indices to support incremental data updating and interactive analysis. It incorporates a high performance indexing framework that allows users to flexibly define the most suitable customized index structures to facilitate query evaluation. This framework is further extended with a two-phase parallel query evaluation strategy that can make the best use of user-defined customized index structures and executes complicated queries using MapReduce.

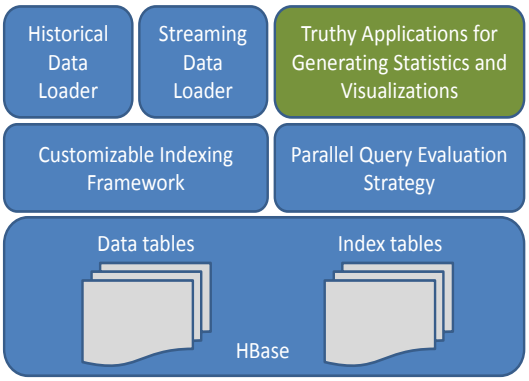

Figure 1. System Architecture of IndexedHBase

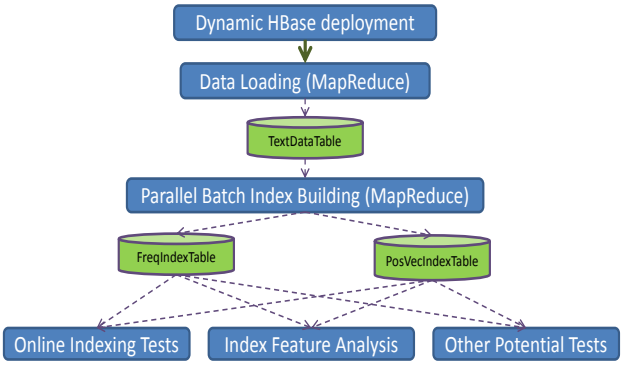

Figure 2. Parallel Data Loading and Index Building

IndexedHBase is used to support the Truthy project (http://truthy.indiana.edu/) to build a public social data observatory as an efficient and scalable storage solution to host TB-level large-scale structured social datasets containing both historical files and real-time streams. Furthermore, the storage systems must provide efficient evaluation mechanisms for its unique type of query, which could potentially involve analysis of hundreds of millions of social updates. For example, the Truthy social data observatory hosts data from Twitter, and a query may be required to extract all the

retweet edges from all tweets containing common hashtags created during a given time window. With the purpose of finding a solution for these challenges, we evaluate NoSQL databases such as Solandra, Riak, and MongoDB, which support text search using distributed inverted indices. However, our investigation shows that traditional inverted indices used in these systems are designed for text retrieval purposes with unnecessary storage and computation overhead for the use case of a social data observatory.

For IndexedHBase, we created two tables to host the original data contained in the .json.gz files, along with a series of index tables to facilitate query evaluation. Table schemas are illustrated in Figure 1. The user table employs a concatenation of user ID and tweet ID as the row key in order to keep track of changes in user metadata associated with each tweet posted. One set of tables is created for each month. 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.

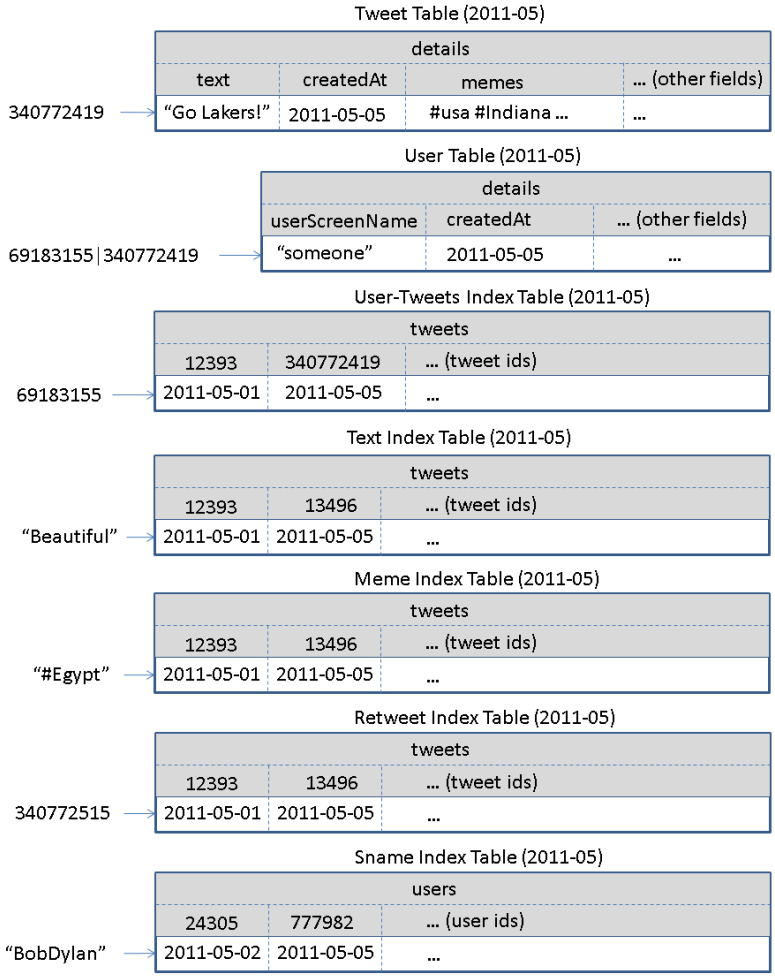

Figure 1. Table schemas used in IndexedHBase.

To load the Truthy dataset into these tables, we propose two separate data-loading strategies in regards to historical data and streaming data. The historical data-loading strategy is implemented as a MapReduce job. Each job can be launched to load one month of data, with multiple jobs running

concurrently in the system, (available resources permitting). This will create multiple mappers with, each trying to load one .json.gz file for the corresponding month. Every mapper continues reading the next tweet from the .json.gz file to create tweet table records, user table records, and all related index table records based on the tweet content. Afterwards it inserts the records into corresponding tables on the fly.

The streaming data-loading strategy is illustrated in Figure 2. To make the loading process more efficient, we suggest running multiple loaders concurrently in the system, each responsible for a portion of the stream. In a simple prototype implementation, all loaders are assigneda unique loader ID and separately connected to the Twitter gardenhose stream using the Twitter streaming API. Upon receiving a tweet, the loader decides whether to process this tweet by way of a modulus operation of the tweet ID over the total number of loaders, then checking if the result matches its loader ID. If the answer is yes, it will create all the related table records and insert them into the tables.

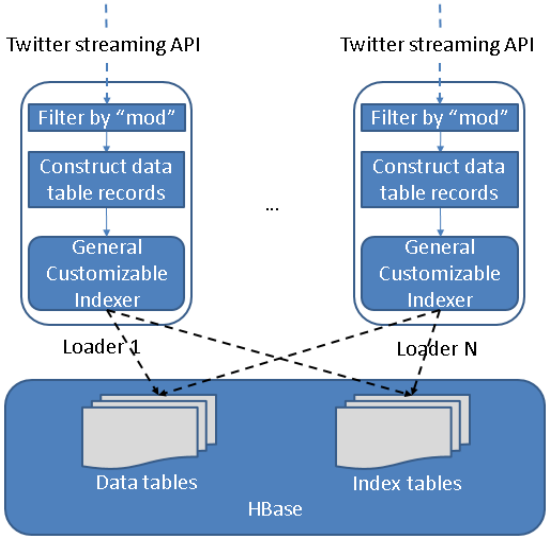

Figure 2. Streaming data-loading strategy on IndexedHBase.

Based on the data tables and index tables, we design a two-phase parallel query evaluation strategy on IndexedHBase, as illustrated in Figure 3. 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. The second phase launches a MapReduce job to process tweets in parallel and extract the necessary information to complete the query.

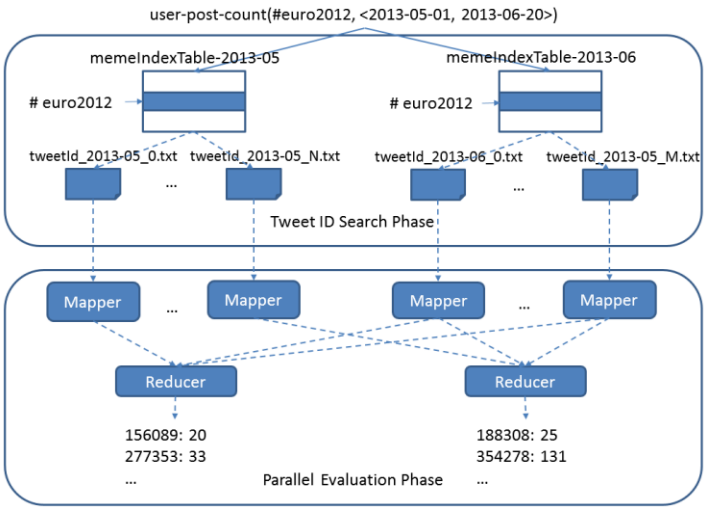

Figure 3. Two-phase parallel evaluation process for an example "user-post-count" query

2.2 Riak

Our implementation on Riak is an extension of Karissa and Clayton's previous work. Specifically, we use different buckets to manage data in different months. Within each bucket, <key, value> pairs are employed to directly store the tweet ID and JSON string of each tweet. Afterwards an extra "truthy_memes" field is added which contains all the hashtags, user-mentions, and URLs in the tweet, separated by a '\t' character.

Riak search is enabled on the buckets to facilitate query evaluation. In the search schema, the "user_id", "truthy_memes", "text", "retweeted_status_id", "user_screen_name", and "created_at" fields are indexed. The "created_at" field is set to "inline only", 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.

To load historical data to Riak, we created one bucket for June 2012, and distributed the 30 files for that month among all 8 nodes of the cluster. The end result of this was that each node had 3 or 4 files. Then an equal number of threads per node were created to load all the files concurrently. Threads continue receiving the next tweet, apply preprocessing with the "created_at" field and "truthy_memes" field and then send the tweet as an object of mime type "JSON" to the Riak server, with the tweet ID as the key. Once the Riak server receives the object, the Riak search module will automatically index all the fields as defined in the search schema before adding the object into the bucket.

Riak supports MapReduce over Riak search results, which means the Truthy queries can be implemented in a similar way to the two-phase evaluation strategy on IndexedHBase. Figure 4 shows an example query in Riak. When this query is submitted, it will first use the index on "truthy_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":"truthy_memes:'#euro2012'",
            "filter":"created_at:['2012-06-08' TO '2012-06-20']"
          },
 "timeout": 72000000,
 "query":[
            {"map":{
                    "language":"javascript",
                    "source":"
                        function(v) {
                            ...
                        }"
                   }
            },
            {"reduce":{
                        "language":"javascript",
                        "arg":{"reduce_phase_only_1":true},
                        "source":"
                            function(v) {
                                ...
                            }"
                      }
            }
          ]
}
```

Figure 4. A sample Truthy query implementation on Riak

3.Experimental Results

3.1 Historical Data Loading

Table 2 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 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
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 a retweeted status, and many retweeted statuses are retweeted multiple times. This means they appear in the JSON string of numerous tweets. In IndexedHBase, the original tweet and the retweet are stored in two separate rows. Therefore, even if a tweet is retweeted repeatedly, only one record is kept for it in the tweet table. In Riak, such a “popular” original tweet will be stored together with every corresponding retweet.

Riak uses inverted indices to index the fields specified in the search schema. However, traditional inverted indices are mainly designed for information retrieval purposes – given a query composed of a set of keywords, they are used for finding the top-K most relevant documents. To achieve this target, information such as frequency and position of keywords is stored in the index, and documents are scored during query evaluation time. This is pure overhead , since queries in Truthy do analysis about all related tweets instead of the top-K most related. This issue is avoided in

IndexedHBase by using specially customized index tables. Note that IndexedHBase compresses table data using Gzip, which generally provides 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. So when the JSON string of each tweet is loaded to the system, it is parsed only once when it is read from the .json.gz files, upon which its content is converted to table records and inserted into different tables. 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 Riak server for extracting indexed field values.
(2) When building inverted indices, Riak not only uses more spaces to store the frequency and position information, but also spends more time collecting such information.

3.2 Scalable Data Loading

We also tested the scalability of historical data loading in terms of cluster size on IndexedHBase. Since we only have 8 nodes on the Bravo cluster, this test was done on the Alamo cluster of FutureGrid. In it we fix the data set to files for two months, May 2012 and June2012, and measure the total loading time at different cluster sizes with 16, 24, and 32 data nodes. The results are illustrated in Figure 5. 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. Still, our results here clearly demonstrate that we can get more system throughput and faster data loading speed by adding more nodes to the cluster.

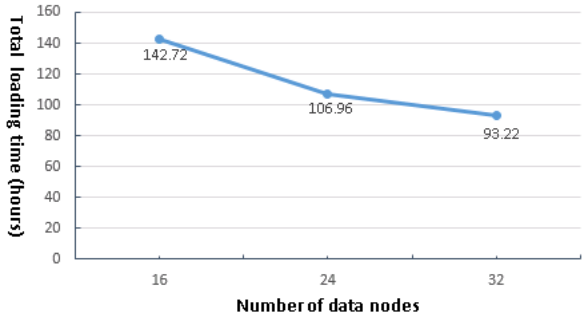

Figure 5. Historical data loading scalability to data size

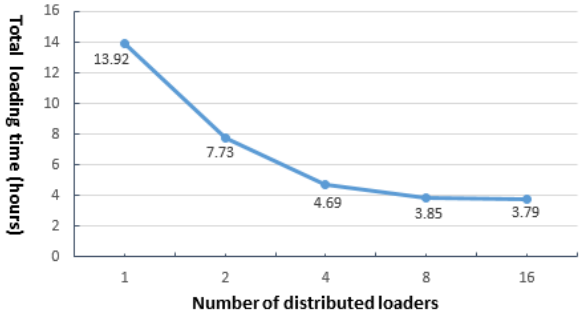

Figure 6. Results for streaming data loading test

3.3 Streaming Data Loading 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 rate coming from 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 throughput against different number of loaders. For each amount of distributed loaders, the whole 2013-07-03.josn.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 corresponding 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 streaming loader gets an extremely high incoming data rate that is equal to local disk I/O speed. For every case, we measure the total time for the loaders to finish loading the whole file, thereby allowing us to estimate the system's capacity accordingly.

Figure 6 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 region servers sees 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.

3.4 Query evaluation

We choose one popular meme "#euro2012" within the loaded dataset, as along with a time window whose length varies from 3 hours to 16 days. This is done to test the query evaluation performance of Riak and IndexedHBase for handling queries involving different number of tweets and different result sizes. 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 in Riak and IndexedHBase:

(1) No MapReduce on either Riak or IndexedHBase.

The "meme-post-count" query falls into this category. On IndexedHBase, query evaluation can be 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 the case of Riak this is accomplished by issuing an HTTP query for each meme, only fetching the "id" field of each tweet. There is no way to directly access the index data – we have to fetch at least one field to finish the query.

Figure 7 shows the query evaluation time for "meme-post-count" on Riak and IndexedHBase. As the time window gets longer, the query evaluation time increases for bothHowever, the absolute evaluation time is much shorter for IndexedHBase, because Riak has to spend extra time to retrieve the "id" field.

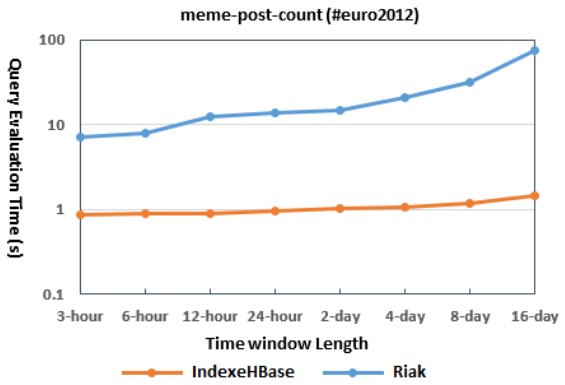

Figure 7. Query evaluation time for "meme-post-count"

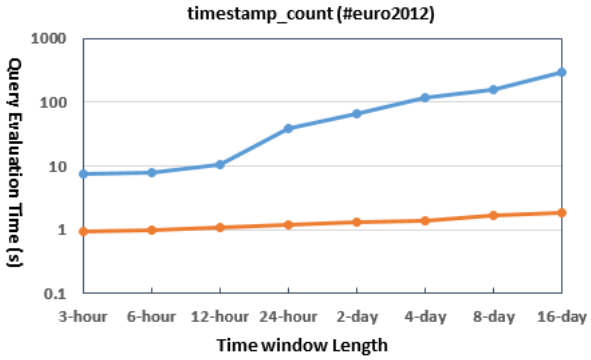

Figure 8. Query evaluation time for "timestamp-count"

(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. Riak, implements itwith MapReduce over Riak search results, where the MapReduce phase completes the timestamp counting based on the content of the related tweet.

Figure 8 shows the query evaluation time for "timestamp-count" on Riak and IndexedHBase. 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. Figure 9 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 3 lists the results sizes (number of unique tweets, users, memes, edges, etc.) for these queries.

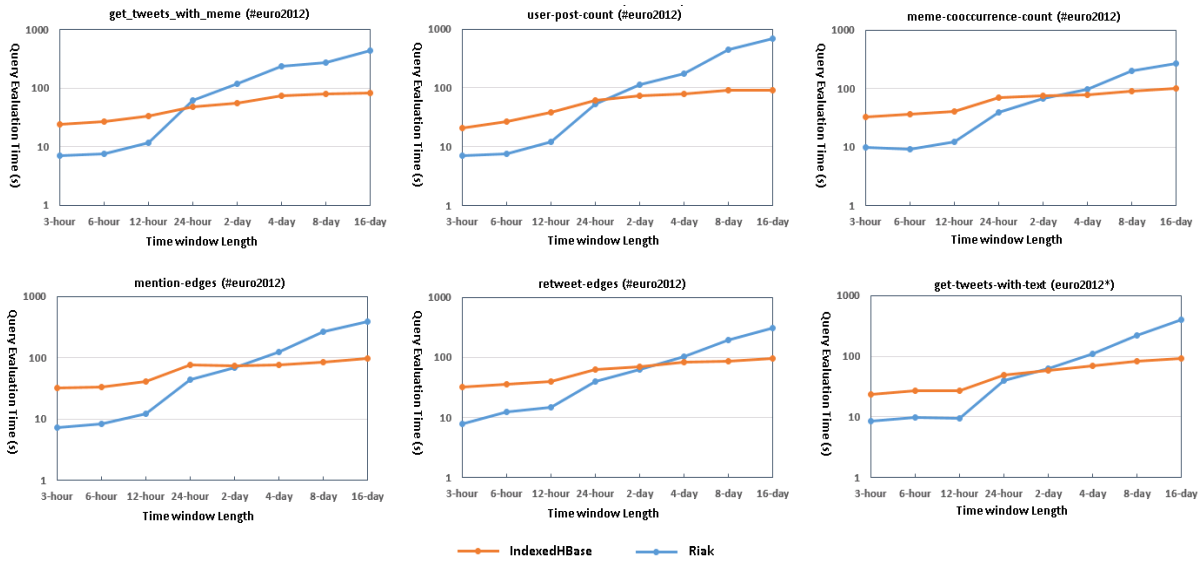

Figure 9. Query evaluation time for queries requiring MapReduce on both platforms

Table 3. Result sizes for queries tested

	get-tweets-with-meme	user-post-count	meme-cooccurrence-count	mention-edges	retweet-edges	get-tweets-with-text
3 hours	1287	1155	372	673	356	1547
6 hours	2539	2292	698	1367	642	3009
12 hours	9342	8476	2190	4885	2112	11707
24 hours	87596	69788	10017	31330	16884	94877
2 days	144575	106679	15414	49265	26143	154506
4 days	234643	162743	23911	80547	43012	249126
8 days	434043	262952	39522	145498	79701	458140
16 days	606062	341840	51775	207783	115788	639386

The main reason for the performance difference observed is the different manner in which the MapReduce model is implemented 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 kind of heavyweight initialization process for setting up the runtime environment for MapReduce tasks on the worker nodes (such as distributed cache). Hadoop MapReduce uses local disks on worker nodes to save intermediate data, and does grouping and sorting of intermediated data based on their keys before they are passed to reducers for final processing. A job can be configured to use zero or multiple reducers.

By comparison, the MapReduce framework on Riak is more lightweight, designed for use cases where users can just write some simple query logics with JavaScript and get them running on the data nodes quickly without a complicated initialization process. There is always a single reducer running for each MapReduce job. Intermediate data is not saved on local disks of the worker nodes, but transmitted directly from mappers to the reducer. The format of intermediate data in Riak MapReduce is a list of values instead of list <key, value> pairs, so no sorting or grouping is done with the intermediate data. The whole chunk is passed to the reducer directly through the memory stack, so the reducer will crash for large intermediate data sizes. Furthermore, the default timeout of reducer is set to 5 seconds, which also demonstrates the lightweight use cases for which Riak MapReduce is designed. We actually had to change this parameter in the source codes and recompile Riak to get some of the above queries working.

Since most of queries in Truthy use time windows at the level of weeks or months, IndexedHBase is more suitable for the queries above. Further we can use a lighter MapReduce such as our own Twister with HBase.

Riak is especially fast for queries with small result sets, such as get-tweets-with-user and get-retweets, as illustrated in Figure 10 and 11. Specifically, for cases where “get-retweets” returns no results, a naïve implementation on IndexedHBase still runs for ~15 seconds, which gives an estimate of the pure job initialization overhead of Hadoop MapReduce. In order to improve the performance of IndexedHBase for such queries, we modified the query implementations, so that for smaller number of results (less than 30,000), the query client will try to retrieve the tweets sequentially. After this improvement, IndexedHBase can achieve query evaluation speed at the same level as Riak for queries with smaller result sizes.

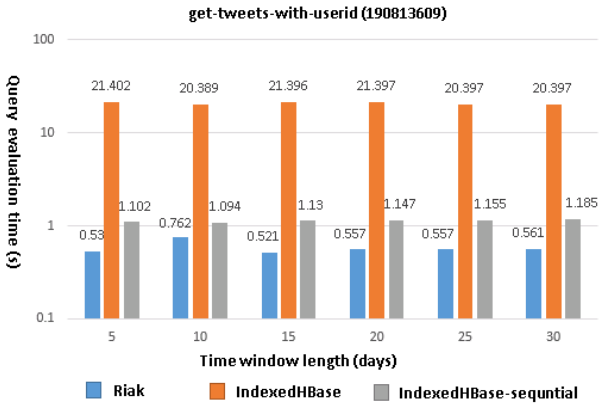

Figure 10. Query evaluation time for "get-tweets-with-userid"

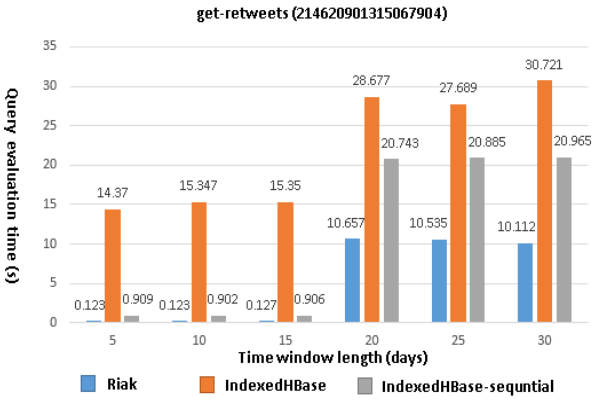

Figure 11. Query evaluation time for "get-retweets"

3.4 Changing Index Schemas for Faster Evaluation

One advantage of IndexedHBase is that it can accept dynamic changes to the index structures to achieve more efficient query evaluation. After the index structures are changed, we can use an efficient MapReduce algorithm to rebuild the changed index table based on the existing data tables, without reloading the whole data set. To verify this, we extended the meme index table to also include user IDs of tweets in the cell values, as illustrated in Figure 12. Using this new index structure, IndexedHBase is able to evaluate the "user-post-count" query by only accessing index data, which could be dramatically faster than the current implementation.

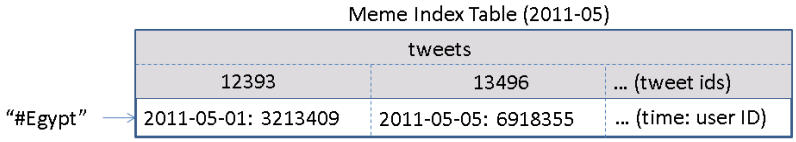

Figure 12. Extended meme index table schema

We tested this schema change on the tables for the 2012-06 dataset. The time taken for rebuilding the meme index table was 3.89 hours. The table size increased from 14.23GB to 18.13GB, which is 27.4% larger. Figure 13 illustrated the query evaluation time comparison. In cases where the "user-post-count" query is frequently used, the query evaluation speed improvement is definitely worthy the storage overhead.

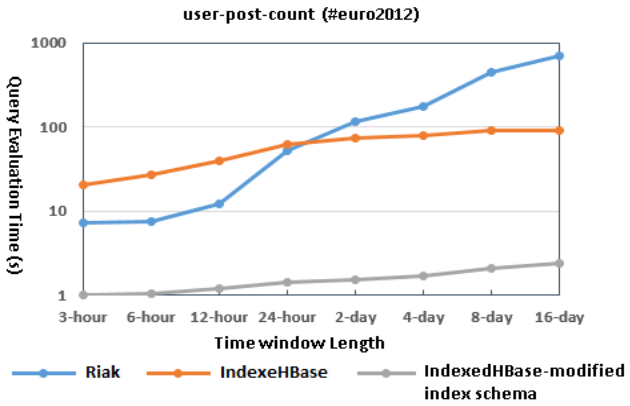

Figure 13. Query evaluation time comparison with modified meme index table schema

Appendix

Each node's configuration is listed in Table 4. For IndexedHBase, 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. For Riak, all eight nodes are used to construct a Riak ring. The data replication level is set to 2 on both platforms. IndexedHBase table data is compressed using Gzip, while Riak uses LevelDB as the storage backend and data is compressed with Snappy.

Table 4. Per-node configuration used in the testing environment

CPU	RAM	Hard Disk	Network
8 * 2.40GHz	192MB	2TB	40Gb InfiniBand

Acknowledgement

We would like to thank Evan Roth for his help on performance evaluation and the Truthy group for their continued assistance.