Scalable Inverted Indexing on NoSQL Table Storage

Xiaoming Gao
School of Informatics and Computing, Indiana University
201H Lindley Hall, 150 S. Woodlawn Ave.,
Bloomington, IN 47405
1-812-272-6515
gao4@indiana.edu

Judy Qiu
School of Informatics and Computing, Indiana University
201D Lindley Hall, 150 S. Woodlawn Ave.,
Bloomington, IN 47405
1-812-855-4856
xqiu@indiana.edu

ABSTRACT
The development of data intensive problems in recent years has brought new requirements and challenges to storage and computing infrastructures. Researchers are not only doing batch loading and processing of large scale of data, but also demanding the capabilities of incremental updates and interactive analysis. Therefore, extending existing storage systems to handle these new requirements becomes an important research challenge. This paper presents our efforts on IndexedHBase, a scalable, fault-tolerant, and indexed NoSQL table storage system that can satisfy these emerging requirement in an integrated way. IndexedHBase is an extension of the cloud storage system HBase. Modeled after Google's BigTable, HBase supports reliable storage and efficient access to terabytes or even petabytes of structured data. However, it does not have an inherent mechanism for searching field values, especially full-text field values. IndexedHBase solves this issue by adding support for an inverted index to HBase, and storing the index data with HBase tables. Leveraging the distributed architecture of HBase, IndexedHBase can achieve reliable index data storage, fast real-time data updating and indexing, as well as efficient parallel data analysis using Hadoop MapReduce. Exploiting the inverted index, IndexedHBase employs three different searching strategies to support interactive data analysis. In order to evaluate IndexedHBase in large scale HPC systems, we extend the MyHadoop framework and provide MyHBase, which can dynamically build a one-click HBase deployment in an HPC job, and automatically finish related tasks. We test the performance of IndexedHBase with the ClueWeb09 Category B data set on 101 nodes of the Quarry HPC cluster at Indiana University. The performance results show that IndexedHBase not only satisfies the requirements for fast incremental data updating, but also supports efficient large scale batch processing over both text and index data. Moreover, by intelligently selecting suitable strategies, searching performance for interactive analysis can be improved by one to two orders of magnitude.

Categories and Subject Descriptors
H.3.1 [**Content analysis and Indexing**]: Indexing methods – *full-text indexing in NoSQL databases.*

General Terms
Algorithms, Design, Experimentation, Performance, Measurement, Reliability.

Keywords
HBase, Inverted Index, Data Intensive Computing, Real-time Updating, Interactive Analysis.

1. INTRODUCTION
Data intensive computing has been a major focus of scientific computing communities in the past several years, and the development of data intensive problems has brought new requirements and challenges to storage and computing infrastructures. Researchers nowadays are not only doing batch loading and processing of big data, but also demanding capabilities of incremental updating [] and interactive searching and analysis [] from the data storage systems. Distinctly from batch loading, incremental data updating handles relatively small pieces of data, and finishes in real-time. Similarly, interactive searching and analysis also targets a relatively small portion of data, and requires a response time of seconds or minutes. For example, social network researchers may want to dynamically collect data from Twitter or Facebook and save them in real-time, and then issue queries like "what is the age distribution of all the people who have talked about Los Angeles Lakers in their status in the last 6 months?", and expect to get an answer in seconds or minutes.

While many existing systems for data intensive problems can handle data loading and processing in large batches very well, adding support for real-time updating and interactive analysis to them remains a research problem. Inspired by previous developments in the fields of information retrieval and database technologies, we believe indexing is the key towards efficient search and interactive analysis. Specifically, in order to create a suitable and powerful indexing mechanism for data intensive systems, we need to resolve the following research challenges:

(1) In case of large data size, how can we support reliable and scalable index data storage, as well as high-performance index access speed?

(2) How can we achieve both efficient batch index building for existing data and fast real-time indexing for incremental data?

(3) How do we design and choose proper searching strategies that can make good use of the indices to support interactive analysis?

(4) While functionalities of real-time updating and interactive analysis are added, how can we retain the existing capability of large scale data processing, and extend it to analysis over both original data and index data?

(5) How can we evaluate our solutions for these issues with large data sizes and on large-scale systems?

This paper presents our efforts towards addressing these challenges. Our work is based on a well-known cloud storage system, HBase []. Modeled after Google's BigTable [], HBase can support scalable storage and efficient access to terabytes or even petabytes of structured data. Besides, HBase is naturally integrated with the Hadoop [] MapReduce framework, thus it can support efficient batch analysis through large scale parallel processing. However, it does not provide an inherent mechanism for searching field values, especially for full-text field values. Searching and selective analysis can only be done by scanning the whole data set and finding the target data records, which is obviously inefficient and not suitable for interactive analysis. There are existing efforts about building indices to facilitate field value search in HBase, but they either do not consider full-text field values [], or do not have enough support for efficient batch index building and large scale index data analysis [].

In this paper, we focus on the issue of full-text value search in HBase, and propose to solve it by involving the usage of inverted index. Figure 1 shows an example fragment of an inverted index. For a given set of text documents, where each document is composed of a set of different terms (or words), an inverted index records for each term, the list of documents that contain it in their text? Specifically, it contains information about the frequencies and positions of terms in documents, as well as (in some cases) the degree of relevance between terms and documents.

"cloud" -> doc1, doc2, …
"computing" -> doc1, doc3, …

Figure 1. An example fragment of inverted index.

The inverted index technology has been widely used in information retrieval systems for searching text data, and the most well known implementation is the Apache Lucene library []. However, most existing Lucene-based systems, such as Solr [], maintain index data with files, and thus do not have a natural integration with HBase. Therefore, we propose a novel framework that can build inverted indices for text data in HBase, and store inverted index data directly as HBase tables. We call this framework IndexedHBase. Leveraging the distributed architecture of HBase, IndexedHBase can achieve reliable and scalable index data storage, as well as high performance for index data access. Moreover, by choosing proper searching strategies based on inverted indices, IndexedHBase can improve searching performance by several orders of magnitude, and therefore supports interactive analysis very well.

We use the ClueWeb09 Category B data set [] to test the effectiveness and performance of IndexedHBase, and carry out our experiments on 101 nodes of the Quarry HPC cluster [] at Indiana University. The following sections will explain, analyze, and verify our design and implementation choices towards solving the abovementioned research challenges. Section 2 gives a brief introduction about HBase. Section 3 describes the system design and implementation of IndexedHBase. Section 4 presents and analyzes the performance experiments of IndexedHBase in terms of parallel index building, real-time updating, distributed index access, and searching. Section 5 demonstrates the advantage of IndexedHBase in parallel data analysis with a synonym mining application. Section 6 compares IndexedHBase with related technologies, and Section 7 has our conclusion and outlines our future work.

2. HBASE

HBase is an open-source, distributed, column-oriented, and sorted-map datastore modeled after Google's BigTable. Figure 2 illustrates the data model of HBase. Data are stored in tables; each table contains multiple rows, and a fixed number of column families. For each row, there can be a various number of qualifiers within each column family, and at the intersections of rows and qualifiers are table cells. Cell contents are uninterpreted byte arrays. Cell contents are versioned, and a table can be configured to maintain a certain number of versions. Rows are sorted by row keys, which are also implemented as byte arrays.

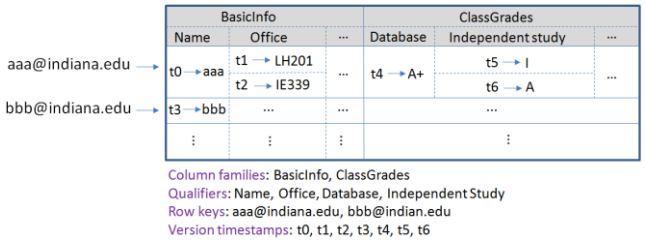

Figure 2. An example of the HBase data model.

Figure 3 shows the architecture of HBase. At any time, there can be one working HBase master and multiple region servers running in the system. One or more backup HBase masters can be set up to prevent single point of failure. Apache ZooKeeper [] is used to coordinate the activities of the master and region servers. Tables are horizontally split into regions, and regions are assigned to different region servers by the HBase master. Each region is further divided vertically into stores by column families, and stores are saved as store files in HDFS. Data replication in HDFS and region server failover ensures high availability of table data. Load balance is done through dynamic region splitting, and scalability can be achieved by adding more data nodes and region servers.

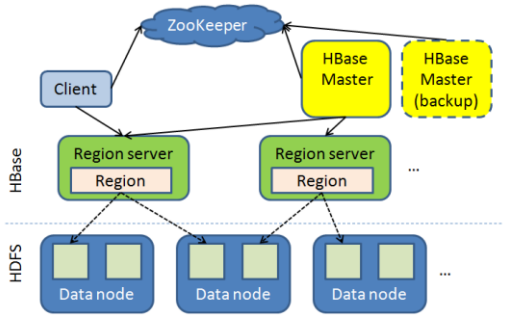

Figure 3. HBase architecture.

Based on this distributed architecture, HBase can support efficient access to huge amounts of data, and can be considered as a good candidate for meeting our identified requirements for supporting data intensive applications. However, it does not provide a native mechanism for searching field values, and thus cannot satisfy the requirement for interactive analysis. There are existing projects and work on building indices to facilitate field value search in HBase, but they either do not target full-text field values, or do not provide efficient solutions for batch index building and large scale index data analysis. Therefore, to solve this problem, we suggest building an inverted index for full-text data in HBase, and storing this index data in HBase tables. The next section will present and discuss the details about the design and implementation of our new system IndexedHBase.

3. INDEXEDHBASE DESIGN AND IMPLEMENTATION

3.1 Design of Table Schemas

In order to store text data and index data in HBase tables, proper table schemas are needed. Figure 4 illustrates major table schemas in IndexedHBase. Since the ClueWeb09 data set is composed of HTML web pages crawled from the Internet, we design the first table schema in Figure 4 to store the text contents of these web pages. For convenience of expression, we also call these web pages "documents". This table is named "CW09DataTable". Each row in the table contains data of one document, and the row key is a unique document ID. There is only one column family in this table, named "details". Each row has two columns in this column family. The "URI" column records the URI of each web page, and the "content" column contains the text data extracted from its HTML content.

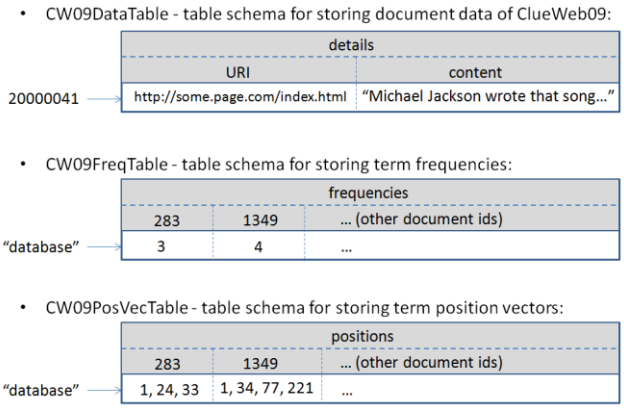

Figure 4. Major table schemas in IndexedHBase.

Inverted indices normally contain two types of information about terms' appearances in documents: their frequencies and positions. Correspondingly, we design two table schemas to store them in IndexedHBase, as illustrated by the second and third schema in Figure 4. Term values are used as row keys in both schemas, so each row contains inverted index information for one unique term. The CW09FreqTable contains one column family named "frequencies". Under this column family, each row has a different number of columns. Each column records one document containing the corresponding term as specified by the row key: the column name is the document ID, and the cell value is the frequency of that term in that document. The CW09PosVecTable also contains only one column family, named "positions". The columns for each row in this table are similar to CW09FreqTable; the only difference is that the cell values are vectors that record terms' positions in documents, instead of frequencies.

Using these tables to store index data brings the following advantages to IndexedHBase:

(1) Leveraging the distributed architecture of HBase, IndexedHBase can provide high availability for index data storage, and high performance for distributed index data access. Our performance evaluations in section 4 will verify this expectation.

(2) Although the information in CW09FreqTable can be totally reconstructed by scanning the CW09PosVecTable, we are still keeping a separate table for it. This is because these two tables may be needed in different searching context or data analysis applications. As will be demonstrated in section 5, in many cases only the frequencies information is needed. Since the row size of CW09PosVecTable is mostly much larger than CW09FreqTable, keeping a separate CW09FreqTable can help reduce the size of data transmission by a large portion.

(3) Since rows are sorted by row keys in HBase, it is easy to do a complete range scan of terms in index tables. This can be very helpful for evaluating queries containing wild characters, such as "ab*". Besides, since the qualifiers (document IDs) in each row are also sorted internally by HBase, it is easy to merge the index records for multiple terms.

(4) Since HBase is designed for efficient random access to cell data in tables, IndexedHBase can support very fast real-time document updates. The insertion, update, or deletion of a document only involves random write operations to a limited number of rows in these tables, and has little impact on the overall system performance, because HBase supports atomic operations at row level. According to our performance tests in section 4, real-time document updates can be completed at the level of milliseconds. Therefore, although temporary data inconsistency can happen during a document update, eventual consistency can be guaranteed within a very short time window.

(5) Based on the original support for Hadoop MapReduce in HBase, we can develop efficient parallel algorithms for building inverted indices. Furthermore, researchers are also able to implement MapReduce applications to complete large scale analysis using both text data and index data.

These advantages of IndexedHBase can help address research challenges (1), (2), and (4) as discussed in section 1.

3.2 System Workflow and Experiments

To testify the effectiveness and efficiency of IndexedHBase, we need to carry out a series of experiments on a large enough test bed and with a large enough data set. Moreover, we need a experimental environment where we can flexibly change testing parameters such as scale of system and data, number of clients, etc. Considering these factors, we choose to use the Quarry HPC cluster at Indiana University to launch our experiments. Since resource allocations in Quarry are completed at the level of HPC jobs, we need to organize our experiments into a proper workflow within a job, as illustrated in Figure 5.

After getting the required resources, the first task is to create a dynamic HBase deployment on the allocated nodes. We modify the MyHadoop [] software to implement this task. MyHadoop is a software package that can be used to dynamically construct a distributed Hadoop deployment in an HPC environment. It is mainly composed of two parts: a set of template Hadoop configuration files, and a set of scripts working with HPC job systems, which apply for HPC nodes, configure nodes as Hadoop masters and slaves, start Hadoop daemon processes on these nodes, and then launch MapReduce jobs on the constructed Hadoop system. The flow chart of the MyHadoop scripts is shown at the left side of Figure 6. We add template configuration files for HBase to MyHadoop, and then add operations in the scripts for configuring ZooKeeper, HBase master and region servers, and for starting HBase daemon processes and applications. We call our modified MyHadoop package "MyHBase", and the flow chart is shown at the right side of Figure 6.

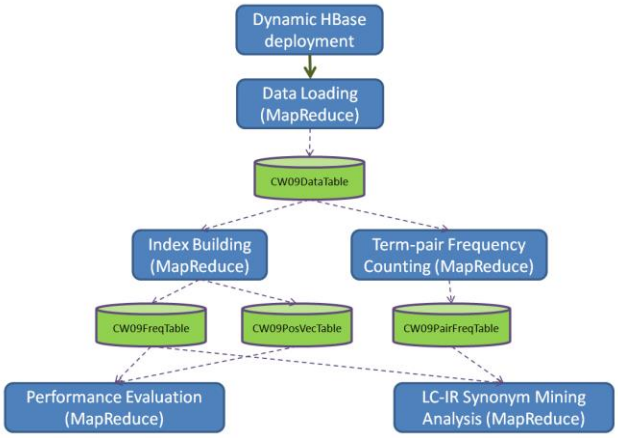

Figure 5. System workflow of IndexedHBase experiments.

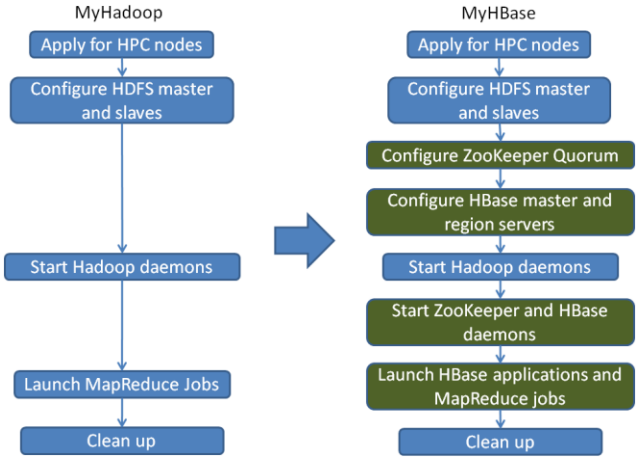

Figure 6. MyHadoop and MyHBase.

After the first task in the workflow is completed, HBase and Hadoop will be running and available for data storage and MapReduce job execution. The second task is a MapReduce program that loads data from the ClueWeb09 Category B data set to CW09DataTable in HBase. The ClueWeb09 data set is originally stored in the form of multiple .warc.gz files, so this program first splits all these files into different groups, then assigns these groups to a set of mapper tasks. Each mapper will read HTML web pages from the files of its groups, and then output HBase "Put" objects for each page, which will then be handled by HBase and inserted as rows to CW09DataTable.

After data are loaded to CW09DataTable, they can be used in two ways. On one hand, we can run MapReduce programs to generate CW09FreqTable and CW09PosVecTable, which will be accessed and tested in a series of performance evaluation experiments. On the other hand, text data in CW09DataTable and index data in CW09FreqTable and CW09PosVecTable can both be useful in various data analysis applications, such as the LC-IR synonym mining analysis [] in our workflow. Implementation of the index building program will be presented in section 3.3; details about the LC-IR synonym mining analysis will be discussed in section 5.

Our system workflow and tasks design address the research challenge (5) as mentioned in section 1.

3.3 Implementation of the Inverted Index Building Task

3.3.1 Overall Index Building Strategy

The index building task in the work flow takes the documents in CW09DataTable as input, builds inverted index for them, and then stores index data into CW09FreqTable and CW09PosVecTable. We use the HBase bulk loading strategy to finish this process, because this is the most efficient way to load data into HBase tables in large batches. The whole process consists of the following two steps:

(1) Run a MapReduce program to scan CW09DataTable, build inverted index for all documents, and write index data to HDFS files in the HFile format, which is the file format HBase internally uses to store table data in HDFS.

(2) Import the HDFS files generated in step (1) to CW09FreqTable or CW09PosVecTable using the "CompleteBulkLoad" tool provided by HBase.

Step (2) normally finishes very fast (in seconds), and the major work is done in step (1). For step (1), we build two MapReduce programs to separately generate data for CW09FreqTable and CW09PosVecTable. This section only explains the implementation of the program for CW09FreqTable, and the implementation for CW09PosVecTable is similar.

3.3.2 HFile Format

Since the MapReduce index building program generates HFiles as output, we need to give a briefly description to the HFile format first. Figure 7 illustrates the HFile format. As described in section 2, one HFile contains data for one column family in one region of a table. The major part of an HFile is composed of (key, value) pairs. A key is composed of four components: row key, column family, qualifier, and timestamp; it defines a specific position with an HBase table. A value is just the cell value at the specified position. All (key, value) pairs in an HFile are sorted in ascendant order by the combination of (row key, column family, qualifier, timestamp). In the Java implementation of HBase, (key, value) pairs are represented as objects of the KeyValue class.

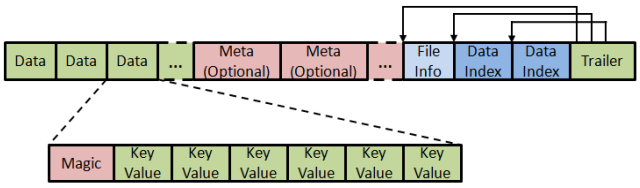

Figure 7. HFile format [].

3.3.3 Implementation of the MapReduce index building program

Before the Mapreduce index building program is launched, CW09FreqTable is created with a predefined number of regions, each having a different row key boundary. The MapReduce job is then configured with these regions' information, so that the job will launch the same number of reducers, each generating the HFile for one region. To generate qualified HFiles, reducers output sorted KeyValue objects, and rely on the HFileOutputFormat class to write them into correctly formatted HFiles.

The execution of the whole MapReduce job is illustrated in Figure 8. Inspired by Lin's work on Ivory [], our index building algorithm

also relies on the Hadoop MapReduce framework to sort the KeyValue objects during the shuffling phase.

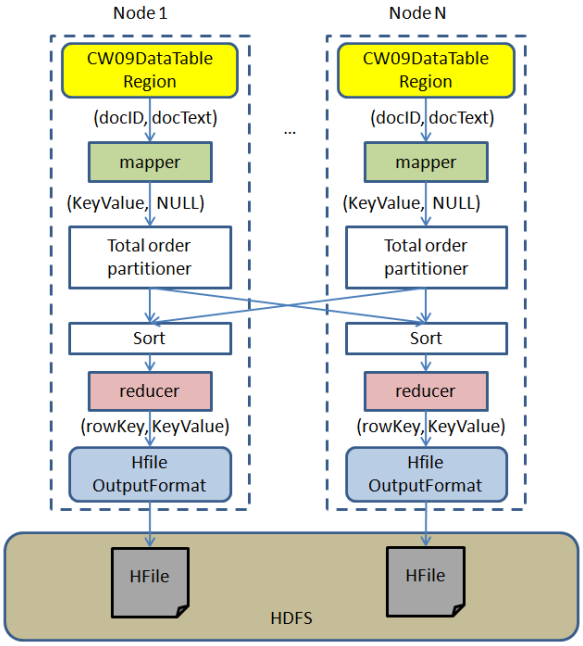

Figure 8. MapReduce job execution for index building.

After the job starts, it launches multiple mapper tasks; each mapper is responsible for building inverted index for the documents in one region of CW09DataTable. Each row in the table is given as one (key, value) input to the mapper, where the key is a document ID and the value contains the text content of the document. The mapper will process the text of the document, count the frequency of each unique term, and generate one index record for each term as a KeyValue object. These KeyValue objects will then be partitioned by a total order partitioner, so that each partition will contain the right set of KeyValue objects for one reducer. The MapReduce framework will then sort these KeyValue objects, and give them to reducers as input. Each reducer will simply pass these sorted KeyValue objects to the HFileOutputFormat, which will write them to the corresponding HFiles for each region of CW09FreqTable. The pseudo codes for the mapper and reducer classes are given in Figure 9.

```
1: class Mapper
2:  method Map(docID n, doc d)
3:      M = new HashMap()
4:      for all term t in doc d do
5:          M(t) = M(t) + 1
6:      for all term t in M do
7:          p = currentTimestamp()
8:          kv = new KeyValue(t, "frequencies", n, p, M(t))
9:          Emit(kv, NULL)

1: class Reducer
2:  method Reduce(KeyValue kv, NULL)
3:      r = kv.getRowKey()
4:      Emit(r, kv)
```

Figure 9. Mapper and reducer implementation for index building program.

Our efforts on the inverted index building task also address the research challenge (2) as mentioned in section 1. The solution for challenge (3) will be presented and analyzed in subsection 4.5 of the next section.

4. PERFORMANCE EVALUATIONS

4.1 Testing Environment Configuration

We use the ClueWeb09 Category B data set to test the performance of IndexedHBase in various aspects, including parallel index building, real-time document updating and indexing, index data access, and searching. The whole data set contains about 50 million web pages; its size is 232GB in compressed form, and about 1.5TB after decompression. Data are stored as files in gzip-compressed Web Archive (WARC) file format, so each file has an extension name of ".warc.gz".

We use 101 nodes in the Quarry HPC cluster of Indiana University to carry out our experiments, and the data set is initially stored in the Data Capacitor (Lustre) file system that is mounted to Quarry. We use a major part of the data set (about 93%) for batch data loading and index building tests, and the rest for real-time document updating and indexing tests.

Each node in the testing cluster has two Intel(R) Xeon(R) quad-core E5410 CPUs at 2.33GHz, 16GB memory, and about 85GB local disk storage under the /tmp directory. Each node has two 1GB Ethernet adapters, and all nodes are connected to the same LAN. The operating system running on each node is Red Hat Linux version 6 (RHEL 6), and we use Java 1.6, Hadoop 1.0.4, HBase 0.94.2, and MyHadoop 0.2a in our tests. Among the 101 nodes, one is used to run HDFS name node and Hadoop job tracker, one is used to run HBase master, and three are used to build a ZooKeeper quorum; the other 96 nodes are used to run HDFS data nodes and HBase region servers. In HDFS, each data node uses a sub-directory under /tmp as the local storage location. In HBase, gzip is used to compress data for all tables. The parallel index building tests are launched at the scale of 48, 72, and 96 data nodes to measure the scalability of the index building program. All the other tests are done with 96 data nodes. To avoid contention to local disk and memory, we set the maximum number of mappers and reducers to 4 and 2 on each data node.

4.2 Index Building Performance Test

This test measures the performance and scalability of our parallel index building algorithm. Figure 10 shows the time used for building CW09FreqTable at different cluster sizes, and the results for CW09PosVecTable are similar. In case of 96 data nodes, it takes about 68 minutes to load data into CW09DataTable, and 181 minutes to build the inverted index. So the index building time is only 2.66 times of the data loading time. Besides, our index building performance is also comparable to the performance of Ivory's index building program as reported in []. Considering the overhead of table operation handling, (key, value) pair sorting, and data replication from HBase, our index building algorithm proves to be very efficient in building inverted indices for text data stored in HBase. Moreover, the index building time gets shorter as the number of nodes in the cluster increases, and we get a nice speed up of 1.76 when the cluster size is doubled. This indicates that our index building program is scalable, and IndexedHBase can easily accommodate larger data sets by having more resources.

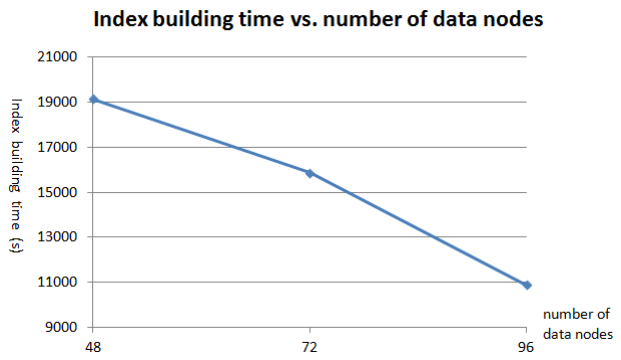

Figure 10. Parallel index building performance at different cluster sizes.

After the inverted index tables are built, some interesting characteristics about the index data are discovered. For example, one interesting feature is the document count of each indexed term, which means the number of documents containing each term. For the whole data set, a total number of 114,230,541 unique terms are indexed. However, 73,705,898 (64.5%) of them appear in only one document. Only 14,737 (0.01%) of them appear in more than 10,000 documents. Figure 11 illustrates the logarithmic distribution of document count for terms appearing in less than 10,000 documents, and Figure 12 shows the distribution of document count for all other terms. As will be demonstrated in section 4.5, this distribution is very useful for completing efficient searches.

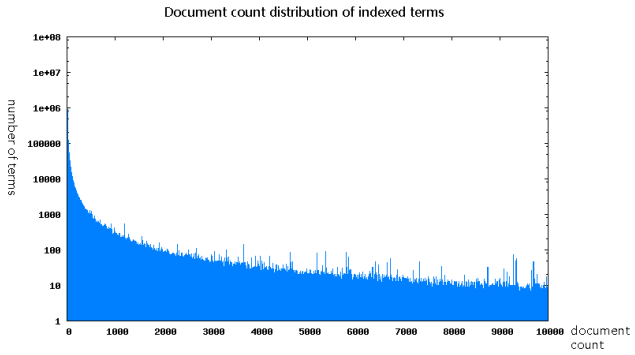

Figure 11. Logarithmic distribution of document count for indexed terms appearing in less than 10,000 documents.

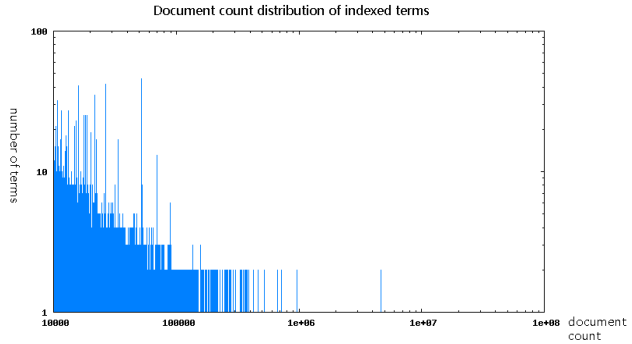

Figure 12. Distribution of document count for indexed terms appearing in more than 10,000 documents.

4.3 Real-time Document Updating and Indexing Test

The update test is done after the major part of the data set is loaded and indexed, and it measures the real-time document updating and indexing performance of IndexedHBase in practical situations where the system already has some preloaded data and multiple clients are concurrently updating and indexing documents in real-time. In this test, multiple clients are started concurrently on different nodes, and each client intensively processes all the documents of one .warc.gz file. For each document in the file, the client first inserts it into CW09DataTable, then creates inverted index records for all terms in the document, and finally inserts these records into CW09FreqTable. We vary the number of concurrent clients from 1 to 32, and measure the aggregate and per-client performance in each case.

Figure 13 shows the variation of average number of documents processed per second (Docs/s) by each client, and Figure 14 shows the system aggregate performance in this regard. Figure 15 shows the variation of aggregate throughput in KB/s. We can see that as the number of concurrent clients increase, per-client performance drops because of intensive concurrent write operations to HBase, but the aggregate system throughput and number of documents processed per second still increases sub-linearly. Even in the case of 32 distributed clients, it takes only 50ms for a client to insert and index one document. This proves that IndexedHBase can support dynamic real-time data updates from multiple application clients very well. Our system differs from the "Near Real-time Search" support in systems like Solr [], and document data and index data in IndexedHBase are persisted to hard disks as soon as they are written into HBase tables. HBase provides row-level atomic operations, so when a document or index record is inserted to a table, it only affects the related row and has little impact on the performance of the whole system. During the update of a document, there could be temporary data inconsistency before all index records are inserted, but eventual consistency can be guaranteed within a time window of milliseconds.

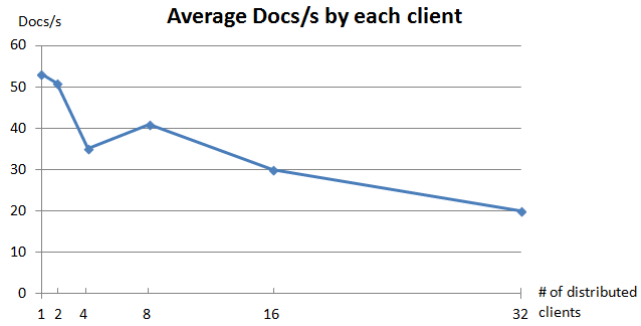

Figure 13. Average number of documents processed per second by each client.

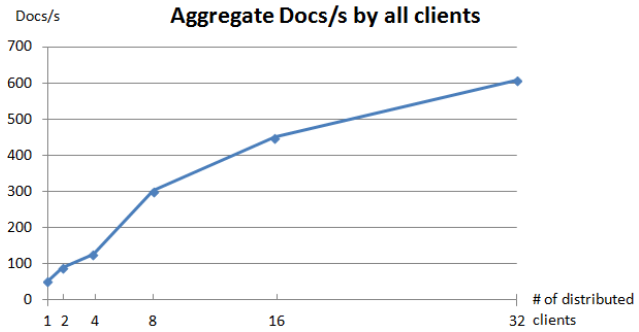

Figure 14. Aggregate number of documents processed per second by all clients.

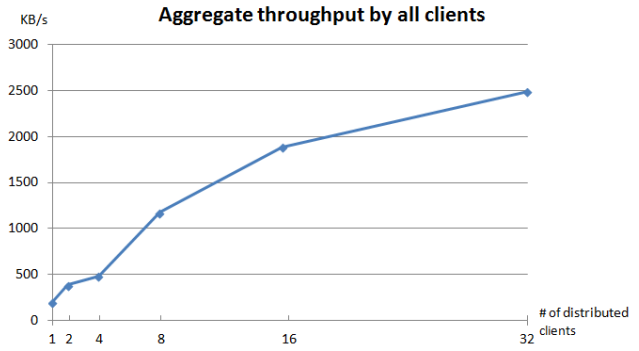

Figure 15. Aggregate data throughput in KB/s by all clients.

4.4 Index Data Access Test

The data access test measures random read performance to index tables, because this is the access pattern to index data that is relevant in most cases. In this test, we also start multiple testing clients on different nodes concurrently, and each client will randomly read 60000 rows from CW09FreqTable. We also measure both per-client performance and aggregate performance for the whole system, and the results are illustrated by Figure 16 and Figure 17. Results for CW09PosVecTable are similar.

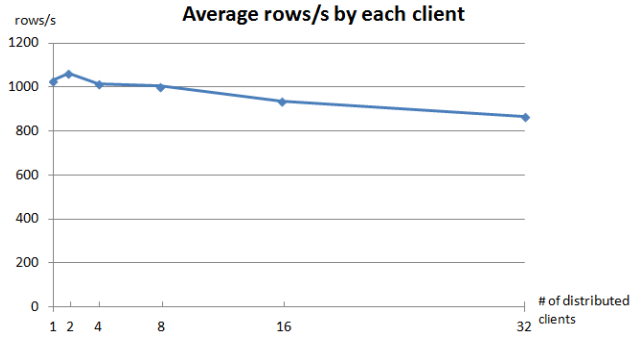

Figure 16. Average number of index rows accessed per second by each client.

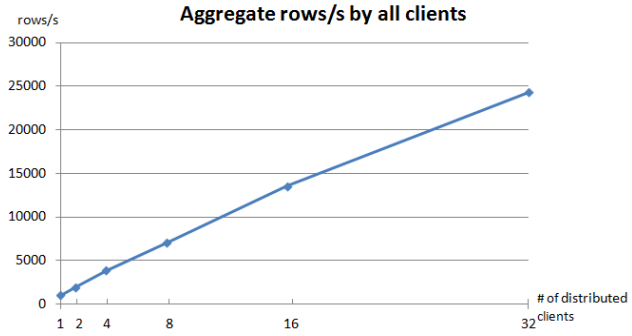

Figure 17. Aggregate number of index rows accessed per second by all clients.

We can observe from Figure 16 that as the number of distributed clients increases, the per-client performance only decreases slightly; Figure 17 shows that the aggregate number of rows accessed per second grows almost linearly. This indicates that IndexedHBase scales very well for distributed index access workload, and can potentially support high volumes of search evaluations in practice.

4.5 Searching Performance Tests

Section 4.1 to 4.4 show that IndexedHBase is efficient and scalable in inverted index creation and access. To support efficient search and interactive analysis, we need appropriate searching strategies that can make good use of the inverted index. We have designed and tested the following three different searching strategies for full-text search:

(1) **Parallel scan search (PSS).** This strategy does not use index at all and is included so we can evaluate the benefits of our new work. To search for a given term, it starts a MapReduce program to scan CW09DataTable with multiple mappers. Each mapper scans one region of the table, and tries to find the given term in the "content" column of each row. If a match is found, the row key (i.e., document ID) will be written to output.

(2) **Sequential index search (SIS).** To search for a given term, this strategy first accesses CW09FreqTable with the term as the row key, and then for each document ID recorded in the row, it sequentially accesses CW09DataTable to get the content of the document, and finally writes the document ID to output.

(3) **Parallel index search (PIS).** To search for a given term, this strategy also first accesses CW09FreqTable with the term as the row key, and gets all the document IDs recorded in the row. It then splits these document IDs into multiple subsets, and starts a MapReduce program to get the content of all these documents. Each mapper in the program will take one subset of document IDs as input, then fetch the content for each ID from the CW09DataTable, and finally write these IDs to output.

It should be noted that all these searching strategies fetch the content data of related documents, although they are not written into output. So the following tests measure their performances for getting the full document data, instead of just the document IDs. Taking the document count distribution in Figure 11 into account, we test the performance of all these strategies for searching 6 terms with different document counts. Table 1 presents the information about the terms, and Table 2 records the results for these tests. Green cells in Table 2 mark the fastest strategy for searching each term.

Table 1. Terms used in searching performance tests

term	document count	document count / total number of documents
all	30237276	65.31%
copyrights	4022026	9.98%
continental	435901	1.08%
youthful	64409	0.16%
pairwise	6011	0.01%
institutional	90	< 0.01%

Table 2. Performance comparison for 3 searching strategies

term	PSS search time (s)	SIS search time (s)	PIS search time (s)	longest / shortest
all	2335	25208	904	28
copyrights	2365	3579	155	23
continental	2394	961	208	12
youthful	2384	282	173	14
pairwise	2427	32	50	76
institutional	2413	3	31	804

We have the following observations from Table 2 for each of the three methods introduced above:

(1) Sequential index search is especially efficient for searching infrequent terms. For terms appearing in a large number of documents, it quickly becomes impractical because of the long time spent on sequentially getting documents' content data.

(2) While parallel scan search reads document data by scanning, parallel index search reads document data by random access. Although scanning is much faster than random access in HBase, the performance of parallel scan search is still not comparable to parallel index search, mainly due to its intensive computation for matching the target term with the document data.

(3) Even for searching the most frequent term "all" in the whole data set, parallel index search can complete in about 15 minutes. This proves IndexedHBase to be a good fit for researchers' requirement for interactive analysis.

These observations suggest that by wisely choosing the appropriate searching strategy, IndexedHBase can save searching time by orders of magnitude as seen in ratio of SIS and PIS (using the inverted index) to PSS (that is the older technology), and thus support interactive analysis very well. Furthermore, to make the choice about searching strategies in practice, multiple factors should be considered, including terms' document count distribution, random access speed of HBase, number of mappers to use in parallel scan search and parallel index search, etc.

5. EXPERIMENTS WITH LC-IR SYNONYM MINING ANALYSIS

5.1 LC-IR Synonym Mining Analysis

Performance results in section 4.3 and 4.5 show that IndexedHBase is efficient for real-time updates and interactive analysis. Using the local context–information retrieval (LC-IR) synonym mining analysis [] as an example application, this section demonstrates the capability and efficiency of IndexedHBase in large scale batch analysis over text and index data. LC-IR is an algorithm for mining synonyms from large data sets. It discovers synonyms based on analysis of words' co-appearances in documents, and computes similarity of words using the formula in Figure 18:

$$Similarity_{\text{LC-IR}}(w_1, w_2) = \frac{\min(\text{Hits}(\text{“}w_1\ w_2\text{”}), \text{Hits}(\text{“}w_2\ w_1\text{”}))}{\text{Hits}(w_1) \times \text{Hits}(w_2)}$$

Figure 18. Similarity calculation in LC-IR synonym mining analysis.

In this formula, Hits("w1 w2") is a function that returns the frequency of word combination "w1 w2", which is the number of times w1 appears exactly before w2 in all documents. Hits("w1") is a function that returns the frequency of word "w1" in all documents. Obviously, these kinds of information can be generated by scanning CW09DataTable and accessing CW09FreqTable.

5.2 A Simple LCIR Synonym Mining Algorithm

Based on the similarity formula in Figure 11, it is straightforward to come up with a simple algorithm for mining synonyms from the ClueWeb09 Category B data set. This algorithm consists of the following steps:

(1) **Word pair frequency counting step.** Scan CW09DataTable with a MapReduce program, and generate a “pair count” table for all word pairs in the documents. Here a "word pair" means two adjacent words in any document.

(2) **Synonym scoring step.** Scan the “pair count” table with a MapReduce program, and calculate similarities of word pairs. Single word hits are calculated by first looking up each single word in CW09FreqTable, and then adding up its frequency in each document it appears.

(3) **Synonym filtering step.** Filter the word pairs with a similarity value above a threshold, and output these. This step is actually carried out on-the-fly by the MapReduce program in step (2).

5.3 An Optimized LCIR Synonym Mining Algorithm

The performance of the simple algorithm of Section 5.2, turns out poor, mainly because step (1) generates a huge number of word pairs, which leads to a huge number of random accesses to CW09FreqTable; moreover, since a single word may appear in many word pairs, there is a lot of repeated calculation for the hits of single words in step (2).

To improve this algorithm, we observe from the formula above that similarity of (w1, w2) is non zero only if both Hits("w1 w2") and Hits("w2 w1") are non zero. Since most word pairs appear only in one order in a given document, we can reduce the number of word pairs to be checked in step (2) by only generating pairs that appear in both order in step (1). Based on this principle, we applied the following optimizations to the simple algorithm:

Firstly, in step (1), local combiners and global reducers were added to filter the word pairs, so that a pair (w1, w2) is generated only if Hits("w1 w2") > 0 and Hits("w2 w1") > 0.

Secondly, before step (2) is executed, a word count table is generated to only record the total hits of each word in the data set. The total hits information is intensively used in step (2), and addition of this table not only makes access to such information faster, but also eliminates the unnecessary total hits recalculation in the simple algorithm. Furthermore, since a large portion of words (40% - 50%) appear only once in all documents, we choose

not to store these words in the word count table. Therefore, if we cannot find a word in this table, we know its frequency is 1. At the same time, we apply a bloom filter to the word count table to efficiently identify words that are not recorded in the table.

Finally, in the synonym scoring step, a memory buffer is added for storing word total hits information, so that repeated access to the frequency of the same word can be completed in local memory.

Figure 19 and Figure 20 illustrate the performance comparison between the naive algorithm and the optimized algorithm on two sample data sets. It is clear that the optimizations improved the performance of both step (1) and step (2). Moreover, the improvement is more significant for larger data sets. The number for the synonym scoring step before optimizations in the 408454 data set is not available because it ran for more than 11 hours, which caused our job to be killed because of wall time limit.

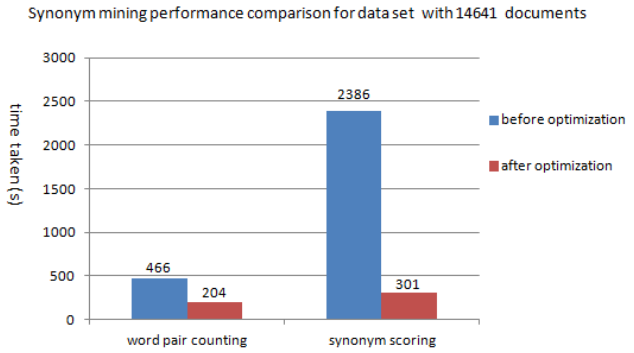

Figure 19. Synonym mining performance comparison for sample data set with 14641 documents.

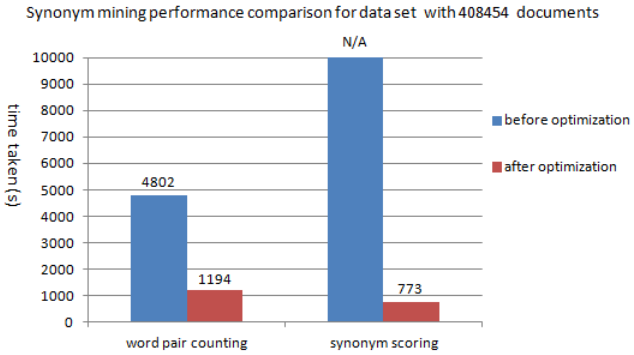

Figure 20. Synonym mining performance comparison for sample data set with 408454 documents.

With the optimized algorithm, we were able to efficiently complete the LC-IR synonym mining analysis over the whole data set. In a configuration with 48 data nodes, step (1) finished in 4 hours and 42 minutes, and step (2) finished in 1 hour and 42 minutes. Setting similarity threshold to 0.1, we were able to find many unusual synonyms that do not even appear in traditional vocabulary. Table 3 shows some example synonyms from our results. In summary, our synonym mining experiments demonstrate that with the right storage and access solution, inverted index data can be useful for not only search, but also large scale data intensive analysis.

Table 3. Example synonyms mined

synonyms	synonym score	meaning
ablepharie, ablephary	0.17	German and English words for the same eye disease
AbsoftProFortran, PGIFortran	0.11	two fortran compilers
abzuyian, bzypian	0.5	two dialects of the Abkhazian language
acamposate, acomposate	0.14	two drugs for curing alcoholism
accessLinkId, idAccessLink	0.13	variable names meaning the same thing

6. RELATED TECHNOLOGIES

Existing technologies similar or related to our project fall into three categories: search oriented systems, analysis oriented systems, and hybrid systems that support both search and data analysis to a certain degree. Table 4 presents a brief comparison between many of these systems and IndexedHBase on some major features. Due to space limit, we use the acronym "PBC" to represent "possible but complicated". The following subsections will the differences in detail.

Table 4. A brief comparison between related technologies and IndexedHBase

Type	System	inverted index storage	MapReduce over text data	MapReduce over index data
Search Oriented	Solr	File	No	No
	Elastic Search	File	No	No
	Katta	File	No	No
	HIndex	File	Yes	PBC
	Ivory	File	No	PBC
Analysis Oriented	Pig	N/A	Yes	N/A
	Hive	N/A	Yes	N/A
Hybrid	MongoDB	File	Yes	No
	Solandra	Table	Yes	PBC
	Indexed HBase	Table	Yes	Yes

(PBC: possible but complicated)

6.1 Search Oriented Systems

6.1.1 Lucene, Solr, ElasticSearch, and Katta

Apache Lucene [] is a high-performance text search engine library written in Java. It can be used to build full-text indices for large sets of documents. The indices store information on terms appearing within documents, including the positions of terms in documents, the degree of relevance between documents and terms, etc. Lucene supports various features such as incremental indexing, document scoring, and multi-index search with merged results. The Lucene library is employed as a core component in many commercial document storing and searching systems, including Solr, Katta, ElasticSearch, etc.

Solr [] is a widely used enterprise level Lucene index system. Besides the functionality provided by Lucene, Solr offers an

extended set of features, including query language extension, various document formats such as JSON and XML, etc. It also supports distributed indexing by its SolrCloud technique. With SolrCloud, the index data are split into shards and hosted on different servers in a cluster. Requests are distributed among shard servers, and shards can be replicated to achieve high availability.

Katta [] is an open-source distributed search system that supports two types of indices: Lucene indices and Hadoop mapfiles. A Katta deployment contains a master server and a set of content servers. The index data are also split into shards and stored on content servers, while the master server manages nodes and shard assignment.

ElasticSearch [] is another open-source distributed Lucene index system. It provides a RESTful service interface, and uses a JSON document format. In a distributed ElasticSearch deployment, the index data are also cut into shards and assigned to different data nodes. Furthermore, there is not a node in a master role; all nodes are equal data nodes and each node can accept a request from a client, find the right data node to process the request, and finally forward the results back to the client.

IndexedHBase differs from SolrCloud, Katta, and ElasticSearch in two respects. Firstly, these systems all manage index shards with files and thus do not have a natural integration with HBase. While each of these systems has its own architecture and data management mechanisms, IndexedHBase leverages the distributed architecture of HBase to achieve load balance, high availability and scalability, and concentrates on choosing the right index table designs for excellent searching performance. Secondly, these systems are oriented towards document storage and search, but not designed for completing large scale data analysis. In comparison, IndexedHBase not only works for efficient search of document data, but also supports large scale parallel analysis over both text and index data based on the MapReduce framework of Hadoop.

6.1.2 HIndex

HIndex [] is also a project that tries to leverage HBase to build distributed inverted index. While the general concept of HIndex is similar to IndexedHBase, it differs in the following major aspects:

Firstly, while IndexedHBase uses HBase as an underlying storage layer and stores index data directly in HBase tables, HIndex modifies the implementation of HBase and maintains inverted index directly with a modified version of HBase region server. This introduces more complexity in terms of system consistency and fault tolerance.

Secondly, document data updates and index data updates are logically coupled in HIndex. Each region server maintains index data for a certain range of document IDs, and whenever a document is inserted, it is indexed by the corresponding region server. Therefore, HIndex is suitable for real-time document insertion and updates, but it is hard to build indices in batches for document data that already exist in HBase tables. Besides, HIndex partitions index data by document IDs, while index data in IndexedHBase are partitioned by terms, since the index tables are using terms as row keys.

Finally, HIndex builds inverted index using the Lucene library, and stores index data as files in Hadoop Distributed File System (HDFS). Therefore, it is also possible to process index data with Hadoop MapReduce in HIndex, but a certain amount of preprocessing and proper input format implementation are necessary. On the other hand, doing parallel analysis over index data with MapReduce is straightforward in IndexedHBase, since index data are directly stored in HBase tables.

6.1.3 Ivory

Ivory [] is an information retrieval system developed by Lin's group at University of Maryland. Ivory uses HDFS to store document and index data, and integrates an information retrieval layer by running "Retrieval Broker" and "Partition Servers" directly as MapReduce jobs on Hadoop. Ivory also uses Hadoop MapReduce to build inverted indices, but it differs from IndexedHBase in two major aspects. Firstly, Ivory stores both document and index data as files on HDFS, and completes index building in batches with MapReduce jobs. It does not consider real-time document insertion and indexing as a requirement. Secondly, Ivory focuses on information retrieval, and does not take data analysis as a major concern. Doing parallel analysis over document and index data with MapReduce is possible in Ivory, but not as straightforward and flexible as in IndexedHBase, since it takes some extra effort and configuration to deal with its specific file formats.

6.2 Analysis Oriented Systems

6.2.1 Pig and Hive

Pig [] is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, and an infrastructure for evaluating these programs. With its "Pig Latin" language, users can specify a sequence of data operations such as merging data sets, filtering them, and applying functions to records or groups of records. This provides ease of programming and also provides optimization opportunities.

Hive [] is a data warehouse system for Hadoop that facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in Hadoop compatible file systems. Hive also provides a language, HiveQL, for data operations, which closely resembles SQL.

Pig and Hive are mainly designed for batched data analysis on large datasets. Pig Latin and HiveQL both have operators that complete searches, but searching is mainly done by scanning the dataset with a MapReduce program and selecting the data of interests. Hive started to support indexing in its later versions, but not including inverted indices for full-text search. In comparison, IndexedHBase not only supports batched analysis via MapReduce, but also provides an interactive way of searching full-text data in real-time based on use of inverted indices.

6.3 Hybrid Systems

6.3.1 MongoDB

MongoDB [] is an open-source document-oriented NoSQL database. It stores structured data in BSON format, a file format similar to JSON with dynamic schemas, and can also be used as a file system. MongoDB supports index on all kinds of document fields, including inverted index on full-text field values, and can evaluate multiple types queries, such as range queries and regular expression queries. MongoDB implements its own data replication and sharding mechanisms to achieve high data availability, scalability, and load balancing. MongoDB also supports MapReduce for batch processing and aggregation operations, with map and reduce functions written in JavaScript.

Compared to IndexedHBase, MongoDB is similar in that it also works as a NoSQL database, and supports inverted index and search for full text data. The difference is that MongoDB stores

inverted index data in as files instead of tables, and does not support batch processing over the index data with MapReduce jobs. MapReduce in MongoDB aims mainly at aggregation operations, and is not as expressive and rich as Hadoop MapReduce. For example, there is no way for a map function written in JavaScript to directly access the index data in MongoDB, which is necessary in our LC-IR synonym mining analysis.

6.3.2 Cassandra and Solandra

Cassandra [] is another open-source NoSQL database system modeled after BigTable. Differently from HBase, Cassandra is built on a peer-to-peer architecture with no master nodes, and manages table data storage by itself, instead of relying on an underlying distributed file system.

Solandra [] is a Cassandra-based inverted index system for supporting real-time searches. The implementation of Solandra is an integration of Solr and Cassandra. It inherits the IndexSearcher, IndexReader, and IndexWriter of Solr, and uses Cassandra as the storage backend. Although Solandra is similar to IndexedHBase in that it also stores index data in tables (in Cassandra), it is different in the following ways:

Firstly, the table schemas used by Solandra are different from IndexedHBase. Similar to Solr, Solandra splits documents into different shards, and the row key of the index table in Solandra is a combination of shard ID, field name, and term value. Therefore, the index data storage is partitioned not only by term, but also by shard ID and field name. Besides, Solandra stores term frequency information and term position vectors in the same table. This may lead to unnecessary data transmission in cases where position vectors are not needed for completing searches.

Secondly, since HBase supports efficient range scan of rows, it is easy to finish range scan of terms in IndexedHBase. In contrast, range scan of rows is not supported very well in Cassandra. As a result, Solandra has to rely on an extra term list table to complete range scan of terms, which is not as efficient as in HBase.

Finally, Cassandra started to integrate with Hadoop MapReduce in its later versions, but the implementation is still not mature enough and the related configuration is not as straightforward as in HBase. Therefore, doing parallel analysis over document and index data in Solandra is not as convenient and efficient as in IndexedHBase.

7. CONCLUSIONS AND FUTURE WORK

The development of data intensive problems in recent years has brought new requirements and challenges from researchers to storage and computing infrastructures, including incremental data updates and interactive data analysis. In order to satisfy these emerging requirements, it is necessary to add proper indexing mechanisms and searching strategies to existing data intensive storage solutions. Moreover, after the addition of these new capabilities, the storage system should still be able to support large scale analysis over both original and index data.

This paper presents our work on IndexedHBase, a scalable, fault-tolerant, and indexed NoSQL table storage system, that addresses these research challenges. In order to support efficient search and interactive analysis, IndexedHBase builds inverted indices for HBase table data, and uses a combination of multiple searching strategies to accelerate the searching process. Moreover, by storing inverted indices as HBase tables, IndexedHBase achieves several advantages, including reliable and scalable index data storage, efficient index building mechanisms for both batch loading and incremental updating, as well as support for large scale parallel analysis over both original and index data.

Performance evaluations show that IndexedHBase is efficient and scalable in batch index building, real-time data updating and real-indexing, and random index data access. Furthermore, by choosing the appropriate optimized searching strategies, IndexedHBase can improve the searching performance by orders of magnitude as shown in table 2. Our experiments with the LC-IR synonym mining analysis demonstrate that inverted index data are not only useful for boosting search, but also valuable for efficient large scale data analysis applications.

There are several directions that we can continue to explore in our future work:

Firstly, our current experiments demonstrate that IndexedHBase is efficient at the scale of 100 nodes. Based on the distributed architecture of HBase and Hadoop, we expect IndexedHBase to scale to a much larger size. Therefore, we plan to carry out experiments at the level of thousands of nodes in the future to further verify the scalability of IndexedHBase.

Secondly, our results in section 4.4 suggest that in order to choose the right searching strategy, multiple factors about the searched term and system environment should be considered. As part of our future work, we will try to build a searching mechanism that can take all these factors into account and make dynamic choices of optimal search strategies to get the best searching performance.

Finally, our current searching strategies can only handle queries as simple combinations of terms. So another major concern in our future work is to develop a distributed search engine that can handle more complicated queries by making and executing distributed query evaluation plans.

8. REFERENCES

[1] Bowman, M., Debray, S. K., and Peterson, L. L. 1993. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.* 15, 5 (Nov. 1993), 795-825. DOI= http://doi.acm.org/10.1145/161468.16147.

[2] Ding, W. and Marchionini, G. 1997. *A Study on Video Browsing Strategies*. Technical Report. University of Maryland at College Park.

[3] Fröhlich, B. and Plate, J. 2000. The cubic mouse: a new device for three-dimensional input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (The Hague, The Netherlands, April 01 - 06, 2000). CHI '00. ACM, New York, NY, 526-531. DOI= http://doi.acm.org/10.1145/332040.332491.

[4] Tavel, P. 2007. *Modeling and Simulation Design*. AK Peters Ltd., Natick, MA.

[5] Sannella, M. J. 1994. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Doctoral Thesis. UMI Order Number: UMI Order No. GAX95-09398., University of Washington.

[6] Forman, G. 2003. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.* 3 (Mar. 2003), 1289-1305.

[7] Brown, L. D., Hua, H., and Gao, C. 2003. A widget framework for augmented interaction in SCAPE. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology* (Vancouver, Canada, November 02 - 05, 2003). UIST '03. ACM, New York, NY,

1-10. DOI= http://doi.acm.org/10.1145/964696.964697.

[8] Yu, Y. T. and Lau, M. F. 2006. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *J. Syst. Softw.* 79, 5 (May. 2006), 577-590. DOI= http://dx.doi.org/10.1016/j.jss.2005.05.030.

[9] Spector, A. Z. 1989. Achieving application requirements. In *Distributed Systems*, S. Mullender, Ed. ACM Press Frontier Series. ACM, New York, NY, 19-33. DOI= http://doi.acm.org/10.1145/90417.90738.