# Finding needles in the haystack: experimenting inverted index in HBase

## 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. This paper presents our efforts towards a storage solution that can satisfy these emerging requirements in an integrated way. Our work is based on 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. We propose to solve this issue by adding support for inverted index to HBase, and storing the index data directly as HBase tables. Leveraging the distributed architecture of HBase, our solution can achieve reliable index data storage, fast real-time data updating and indexing, as well as efficient parallel data analysis through Hadoop MapReduce. Based on usage of inverted index, we design and investigate three different types of searching strategies to support interactive data analysis. We use the ClueWeb09 Category B data set to test the performance of our solution, and carry out our experiments on 101 nodes of the Quarry supercomputer at Indiana University. Evaluation results show that our system not only satisfies the requirements for fast real-time and incremental data updating, but also supports efficient large scale batch processing over both text data and index data. Moreover, by intelligently selecting proper searching strategies, searching performance can be improved by tens or even hundreds of times.

## 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 more requirements and new 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. 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, how to add support for real-time updating and interactive analysis to them remains a research problem. Inspired by the 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 amount of data and on large-scale systems?

This paper presents our efforts towards handling these challenges. Our work is based on a well known cloud storage system, HBase. Modeled after Google's BigTable, HBase can support scalable storage of 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 items, 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 (words), an inverted index records such information: for each term, which subset of documents contain it in their texts? Specifically, it contains information about the frequencies and positions of terms in documents, or even the degree of relevance between terms and documents.



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 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 HIIS (HBase with Inverted Index Solution) for short. Leveraging the distributed architecture of HBase, HIIS 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, HIIS can improve search performance by a factor of tens or even hundreds of times, and therefore supports interactive analysis very well.

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

## 2. HBase

HBase is an open-source, distributed, column-oriented, and sorted-map datastore modeled after Google’s BigTable. Figure 2 shows 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 both uninterpreted arrays of bytes and versioned. A table can be configured to maintain a certain number of versions for its cell contents. Rows are sorted by row keys, which are 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. The 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. Regions are further vertically divided 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 split, 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 researchers' requirements as discussed in section 1. However, it does not provide a native mechanism for searching field values, thus cannot fulfill users' requirements for interactive analysis over targeted subsets of data. There are existing projects and work about 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 inverted index for full-text data in HBase, and storing index data in HBase tables in HIIS. The next section will present and discuss the details about the design and implementation of HIIS.

## 3. HIIS Design and Implementation

### 3.1 Design of Table Schemas

Figure 4 illustrates major table schemas in HIIS. As mentioned in section 1, we use the ClueWeb09 data set to verify our solution. Since the 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, named "details", in this table, and each row has two columns in this column family. The "URI" column records the URI of each document, and the "content" column contains the text data extracted from the HTML web pages.



Figure 4. Major table schemas in HIIS.

Inverted index normally contains two types of information about terms' appearances in documents: frequencies and positions. Correspondingly, we design two table schemas to store them in HIIS, 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 information about the related documents 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 the terms' positions in documents, instead of frequencies.

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

(1) Leveraging the distributed architecture of HBase, HIIS 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 HBase is designed for efficient random access to cell data in tables, HIIS can support very fast real-time document updates. The insertion, update, or deletion of a document only involves random 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.

(4) 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.

### 3.2 System Workflow and Experiments

To testify the effectiveness and efficiency of HIIS, 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.



Figure 5. System workflow of HIIS experiments.

After getting the required resources, the first task is to create a dynamic HBase deployment on the allocated nodes. We modified the MyHadoop [12] 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 added template configuration files for HBase to MyHadoop and added 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 6. MyHadoop and MyHBase.

After the first task, 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 different groups to a set of different mappers. Each mapper will read HTML web pages from the files, 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 a MapReduce program 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.

### 3.3 Implementation of Inverted Index Building Task

The index building task 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.

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.

To achieve efficient index building, CW09FreqTable is created with a predefined number of regions, each having a different row key boundary. The index building program is then configured with the regions' information, so that the MapReduce 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 the HFileOutputFormat will take these objects and write them into correctly formatted HFiles.

The execution of the whole job is illustrated in Figure 8, and the pseudo codes for mapper and reducer classes are given in Figure 9. Inspired by Jimmy 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. Using CW09DataTable as input, the job assigns one mapper to build inverted index for the documents in each region. Each row in CW09DataTable is transformed as one (key, value) input to the mapper, where the key is a document ID and the value is the text content of the document. The mapper will process the text of the document, count the frequency of each unique term, and create one KeyValue object for each term; the row key in the KeyValue is the term, the column name is the document ID, and the value is the frequency of the term. Each mapper generates multiple (KeyValue, NULL) pairs as its output, and these pairs will be partitioned by a total order partitioner based on the row keys in the KeyValue objects, so that each partition will contain the right set of KeyValue objects for one reducer. Before each reducer is launched, the MapReduce framework will sort these (KeyValue, NULL) pairs by KeyValue objects. As a result, when a reducer receives these pairs as input, the KeyValue objects are already coming in order. Therefore, what each reducer does is to simply take the row keys from the KeyValue objects, and output corresponding (rowKey, KeyValue) pairs in the same order as its input. The HFileOutputFormat will take over these pairs, and write the KeyValue objects to the corresponding HFiles for each region of CW09FreqTable.



Figure 8. MapReduce job execution for index building.



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

## 4. Performance Evaluations

### 4.1 Testing Environment Configuration

We use the ClueWeb09 Category B data set to test the performance of HIIS 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, and its size is 232GB in compressed form, and about 1.5TB after decompressed. 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. 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 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, we can observe that the index building time gets shorter as the number of nodes in the cluster increases, and we can get a nice speed up of 1.76 when the cluster size is doubled. This indicates that our index building program is scalable, and HIIS 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 a 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. As will be demonstrated in section 4.5, this distribution is very useful for completing searches.



Figure 11. Logarithmic distribution of document count for indexed terms.

### 4.3 Real-time Document Updating and Indexing Test

This test is done after the major part of the data set (216 GB) is loaded and indexed, so it tries to measure the real-time document updating and indexing performance of HIIS 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 processes one .warc.gz file. For each document in the .warc.gz 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 or CW09PosVecTable. We vary the number of concurrent clients from 1 to 32, and measure the aggregate and per-client performance in each case.

Figure 12 shows the variation of average number of documents processed per second (Docs/s) by each client, and Figure 13 shows the system aggregate performance in this regard. Figure 14 shows the variation of aggregate throughput. We can see that as the number of concurrent clients increase, although per-client performance drops because of intensive concurrent write operations to HBase, 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 HIIS can support dynamic real-time data updates from multiple application clients very well. Different from the "Near Real-time Search" support in systems like Solr [], document data and index data in HIIS 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 12. Average number of documents processed per second by each client.



Figure 13. Aggregate number of documents processed per second by all clients.



Figure 14. Aggregate data throughput in KB/s by all clients.

### 4.4 Index Data Access Test

This test measures random read performance to index tables, since this is the access pattern to index data 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 15 and Figure 16. Results for CW09PosVecTable are similar.



Figure 15. Average number of index rows accessed per second by each client.



Figure 16. Aggregate number of index rows accessed per second by all clients.

We can observe from Figure 15 that as the number of distributed clients increases, the per-client performance only decreases slightly; Figure 16 shows that the aggregate number of rows accessed per second grows almost linearly. This indicates that HIIS scales very well for distributed index access workload, and can potentially support high volumes of search evaluations in practice.

### 4.5 Search Performance Tests

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

(1) Parallel scan search. This strategy does not use index. 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. 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. To search for a given term, this strategy also first access 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, and 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 results for these tests. Green cells mark the fastest strategy for searching each term.

Table1 performance comparison for searching strategies

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| term | document count (%) | parallel scan search time (s) | sequential index search time (s) | parallel index search time (s) | longest / shortest |
| all | 30237276 (65.31%) | 2335 | 25208 | 904 | 28 |
| copyrights | 4022026 (9.98%) | 2365 | 3579 | 155 | 23 |
| continental | 435901(1.08%) | 2394 | 961 | 208 | 12 |
| youthful | 64409 (0.16%) | 2384 | 282 | 173 | 14 |
| pairwise | 6011 (0.01%) | 2427 | 32 | 50 | 76 |
| instituitional | 90 (< 0.01%) | 2413 | 3 | 31 | 804 |

We have the following observations from Table 1:

(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 return in about 15 minutes. This proves HIIS to be a good fit for researchers' requirement for interactive analysis.

These observations suggest that by wisely choosing the proper searching strategy, HIIS can save searching time by tens or even hundreds of fold, and thus support interactive analysis very well. Furthermore, when making a 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. LCIR Synonym Mining Analysis

### 5.1 LCIR Synonym Mining Analysis

Performance results in section 4.3 and 4.5 show that HIIS is efficient for real-time updates and interactive analysis. Using the LC-IR synonym mining analysis [] as an example application, this section demonstrates the capability and efficiency of HIIS 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 11:



Figure 11. 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 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 accessing CW09FreqTable.

### 5.2 A Naive LCIR Synonym Mining Algorithm

Based on the similarity formula in Figure 11, it is straightforward to come up with a naive 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 the document data table 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 threshold value, and output the pairs with similarities higher than the threshold. 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 naive algorithm turned 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 not 0 only if both Hits("w1 w2") and Hits("w2 w1") are not 0. Since most word pairs appear only in one order in the documents, 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 naive 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 eliminates the unnecessary total hits recalculation in the naive 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 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 12 and Figure 13 illustrate the performance comparison between the naive algorithm and the optimized algorithm for 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 synonym scoring 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 12. Synonym mining performance comparison for sample data set with 14641 documents.



Figure 13. 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 2 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 2. Example synonyms mined

|  |  |  |  |
| --- | --- | --- | --- |
| Word 1 | Word 2 | 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. This section compares these existing systems with our solution.

### 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.

Our solution 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, our solution 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 search performance. Secondly, these systems are oriented towards document storage and search, but not designed for completing large scale data analysis. In comparison, our solution 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 our system, it differs in the following major aspects:

Firstly, while our solution 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 our system 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 our solution, since index data are directly stored in HBase tables.

### 6.1.3 Ivory

Ivory is an information retrieval system developed by Jimmy 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 our system 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 our system, 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, our solution 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 our solution, 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. Different 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 our solution 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 ours. 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 our solution. 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 our solution.

## 7. Conclusions and Future Work