Experimenting Lucene Index on HBase in an HPC Environment

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

Vaibhav Nachankar
School of Informatics and Computing, Indiana University
2612 Eastgate Lane, Apt. A, Bloomington, IN 47408
1- 812-606-1014
vmnachan@cs.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@cs.indiana.edu

ABSTRACT
Data intensive computing has been a major focus of scientific computing communities in the past several years, and many technologies and systems have been developed to efficiently store and serve terabytes or even petabytes of data. One important effort in this direction is the HBase system. Modeled after Google's BigTable, HBase supports reliable storage and efficient access to billions of rows of structured data. However, it does not provide an efficient searching mechanism based on column values. To achieve efficient search on text data, this paper proposes a searching framework based on Lucene full-text indices implemented as HBase tables. Leveraging the distributed architecture of HBase, we expect to get high performance and availability, and excellent scalability and flexibility for our searching system. Our experiments are based on data from a real digital library application and carried out on a dynamically constructed HBase deployment in a high-performance computing (HPC) environment. We have completed system design and data loading tasks of this project, and will cover index building and performance tests in future work.

Categories and Subject Descriptors
H.3.3 [**Information Search and Retrieval**]: Search Process – *full-text indexing, search in NoSQL databases.*

General Terms
Performance, Design, Reliability, Experimentation.

Keywords
HBase, Lucene index, HPC cluster.

1. INTRODUCTION
Many research projects have focused on data intensive computing in recent years, and various systems have been developed to tackle the problems posed by storage and access of huge amounts of data. Another issue to be resolved is how to complete efficient search of these data. One proposed solution is HBase [2], a distributed NoSQL database system modeled after Google's BigTable [7]. Built with a distributed architecture on top of the Hadoop Distributed File System (HDFS) [13], HBase can achieve high data availability, scale to billions of rows and millions of columns, and support fast real-time access of structured data. However, it does not provide an original mechanism for searching field values, especially for full-text values.

Apache Lucene [4] 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. The Apache Lucene library supports various features such as incremental indexing, document scoring, and multi-index search with merged results. However, most existing Lucene indexing systems, such as Solr [15], maintain index data with files and do not have a natural integration with HBase.

This paper proposes a searching framework for text data stored in HBase by creating Lucene indices for the data, and storing indices with HBase tables. Leveraging the distributed architecture of HBase, our framework is expected to have high performance and service availability, as well as excellent scalability in terms of data and index size. Furthermore, since index data are maintained as tables and are accessed at the granularity of rows, we expect our solution to have more flexibility for operations of adding and removing documents, which can be carried out online and only affect the rows and cells related to the corresponding documents.

We use data from a real digital library application to validate our solution. The experiments are carried out on a dynamically constructed HBase deployment in an HPC environment hosted at the Alamo site of FutureGrid. Section 1 provides introduction and motivation of our research. Section 2 covers briefly related technologies including eight state-of-the-art systems. We have extended descriptions of our system design including dynamic HBase deployment, data loading tasks and future work in section 3, followed by conclusion in section 4.

2. RELATED TECHNOLOGIES
2.1 HBase
HBase is an open-source, distributed, column-oriented, and sorted-map datastore modeled after Google's BigTable. Figure 1 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.

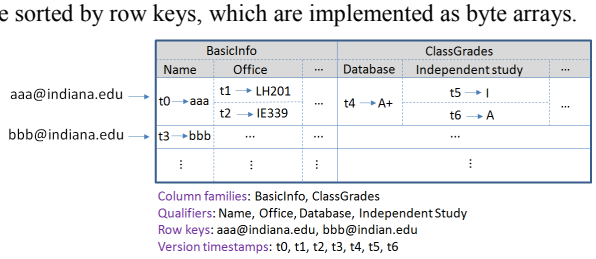

Figure 1. An example of the HBase data model.

HBase runs on top of HDFS, the architecture of which is shown in Figure 2. 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 ensures high availability of table data in HBase. During the runtime operations of the HBase, the ZooKeeper [6] is used to coordinate the activities of the master and region servers, and to save a small amount of system metadata.

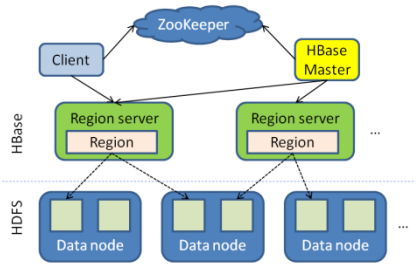

Figure 2. HBase architecture.

HBase supports efficient data access based on row keys, column families and qualifiers, but does not provide a native mechanism for searching cell data. There is existing work on building indices on HBase tables [8], but it is focused on indexing basic-type field values such as strings, numbers and dates. While supporting range search with these indices, it does not address full-text search. Our work in this paper concentrates on supporting full-text search on data stored in HBase by building Lucene indices as HBase tables.

2.2 Pig and Hive

Pig [5] 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 [3] is a data warehouse system for Hadoop [1] 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 Latin and HiveQL both have operators that complete search activities, but the search is done by scanning the dataset with a MapReduce program and selecting the data of interests. Pig and Hive are mainly designed for batched data analysis on large datasets. In comparison, our solution aims at supporting real-time search on data stored in tables based on use of indices.

2.3 Solr, ElasticSearch, and Katta

Solr is a widely used enterprise level Lucene index system. Besides the functionality provided by the Apache Lucene library, 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 [10] 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 [9] 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.

Contrary to our solution, SolrCloud, Katta, and ElasticSearch 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, ours leverages the distributed architecture of HBase to achieve load balance, high availability and scalability, and concentrates on choosing the right index table designs for the best search performance.

2.4 Cassandra and Solandra

Cassandra [12] 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 [14] is a Cassandra-based Lucene index system for supporting real-time searches. Similar to our solution, Solandra also maintains the index data in Cassandra tables, but the table schemas are different from ours. Compared with Solandra, our solution not only serves efficient real-time searches, but also provides better support for large scale parallel analysis on the index data because of the inherent MapReduce support in HBase.

3. SYSTEM DESIGN AND IMPLEMENTATION

3.1 Problem Definition

We use a real digital library application to demonstrate our Lucene index implementation based on HBase, and to test our system's design and performance. To clarify the problem that we are targeting, this subsection gives a brief description of the requirements and data organization of the application.

The basic requirements of the application are to store the digital library data and to serve queries from users. A user submits a query for books from a web interface, gets a list of information related to his/her query, and selects the books of interest to read their text or image content. A query is composed of a set of terms from one or more fields, and can be expressed in the form of {<field1>: term1, term2, ...; <field2>: term1, term2, ...; ...}. For example, a query expressed as {title: "computer"; authors: "Radiohead"; text: "Let down"} is looking for a book written by Radiohead which contains "computer" in the title and "Let down" in the text. A query result is a list of books that contain the specified terms in their corresponding fields.

The whole dataset currently contains ~50GB of data, composed of three parts: bibliography data in XML files, image data in .png

SuperComputing'11, Nov. 12–18, 2011, Seattle, WA, USA.

and .tif files, and text data in .txt files. The bibliography data contain metadata about books, including title, category, authors, publishers, keywords, etc. The image data contains images created by scanning pages in books, and the text data contains texts extracted from the images. We save all kinds of data in HBase tables, and we create Lucene indices for bibliography data and text data. The indices are also saved in HBase tables.

3.2 Architecture and Workflow

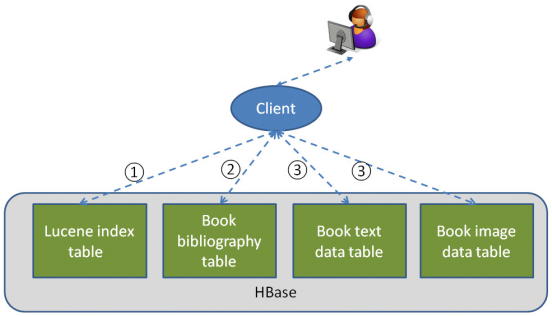

Figure 3. System architecture.

Figure 3 shows the system architecture of our digital library application. We create three tables in HBase to store the bibliography, text, and image data of the books, and one Lucene index table to store the index for bibliography and text data. The table schemas are given in Table 1, expressed in the format of "<row key> --> {<column family>: [<qualifier>, <qualifier>, ...]; <column family>: [<qualifier>, <qualifier>, ...]; ...}". The schemas of the first three tables are self-explainable. In the Lucene index table schema, each row corresponds to a different term value, and is composed of multiple column families, each containing information about the documents having that term value in a specific field. A "<doc id>" will be "<book title>-<seq>" at run time, and the cell data will be the position vector of the corresponding book for the term value.

Table 1. HBase table schemas

Table name	Schema
Book bibliography table	<book title>-<seq> --> {md:[category, authors, createdYear, publishers, location, startPage, currentPage, ISBN, additional, dirPath, keywords]}
Book text data table	<book title>-<seq> --> {pages:[1, 2, ...]}
Book image data table	<book title>-<seq>-<page number> --> {image:[image]}
Lucene index table	<term value> --> {title:[<doc id>, <doc-id>, ...]; category:[<doc id>, <doc id>, ...]; authors:[<doc id>, <doc id>, ...]; keywords:[<doc id>, <doc id>, ...]; texts:[<doc id>, <doc id>, ...]}

This architecture has the following advantages for supporting a Lucene index system:

First, by leveraging the distributed architecture of HBase, the index table is split into regions, which are assigned across all region servers. Therefore, the total throughput of the index system scales with the number of region servers. Moreover, since table data are replicated in HDFS, the index data are of high availability, and its size scales with the capacity of HDFS.

Second, since Hadoop has original support for MapReduce jobs using HBase tables as input, we can develop MapReduce programs for building the index table in a parallel manner, leading to a very fast indexing process. Once the index table is built up, we can have data analysis applications implemented as MapReduce jobs running directly over the index table.

Third, since HBase provides atomic mutations at the level of rows, the index data are maintained at the granularity of term values. This suggests the feasibility of adding or deleting a document in real-time with little interference to the system performance. To add or delete a document, for each unique term in the document we only need to add a set of <qualifier, value> pairs or mark a set of <qualifier, value> pairs as deleted in its corresponding row, which only imposes a very small impact on concurrent accesses to the same row.

The whole system works as follows. A user submits a query through a client program. Using the terms in the query as row keys, the client program will first get a list of titles for the books which contain those terms in their corresponding fields. Then, using the book titles as row keys, the client will get the bibliography data for the books and return them to the user. Finally, the user will select books of their interests, and the client program will access the corresponding text or image data upon the user's requests.

Figure 4 shows the workflow of the major tasks that need to be executed to build and test the system. All tasks and experiments are carried out in the Amalo HPC cluster of FutureGrid, and before any task is executed, the whole dataset is first loaded into a distributed file system mounted on Alamo. The first task is to dynamically construct a distributed HBase deployment in Alamo. After that is done, the bibliography data are loaded from the XML files to the bibliography table in HBase. Since the bibliography table contains file system paths to the text and image data files, it is then used as input for the text and image data loading tasks, which will store the data into HBase tables. The bibliography and text data table will then be used by the Lucene index-building task, and the indices will be stored in an index table. Finally, we can either launch a performance evaluation task to test the efficiency and scalability of the index system, or complete further data analysis tasks on the text data table or index table.

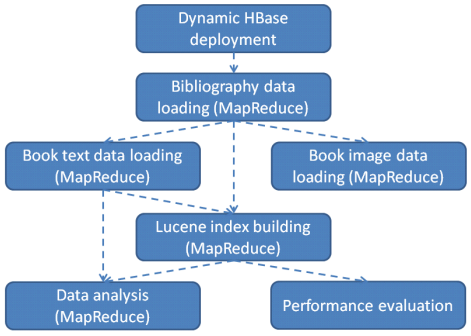

Figure 4. Workflow of major tasks.

3.3 Current Progress

We modified the MyHadoop [11] software to complete the dynamic HBase deployment 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 are shown at the left side of

Figure 5. 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 5.

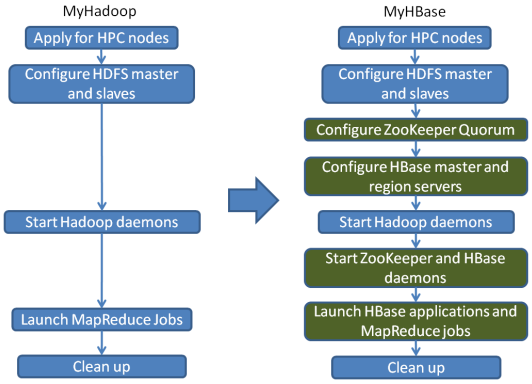

Figure 5. MyHadoop and MyHBase.

The Bibliography data loading task is completed by a modified version of the "SampleUploader" MapReduce program. The "SampleUploader" takes a comma separated text file as input, and passes each line as a value to a mapper task. The mapper parses each line in the format of "<row>, <column family>, <qualifier>, <value>", and converts it to a "Put" object, which is emitted as an intermediate input value for reducers. The reducer takes over the "Put" objects and stores them into the bibliography table. We implemented a pre-processing program to read in all the XML files containing bibliography data, and generate a big text file as input to the "SampleUploader". Since many values of the bibliography data contain commas, we used "###" as a separator and modified the "SampleUploader" to adapt to this change.

The book text data loading task and image data loading task are completed by two MapReduce programs. These programs take the bibliography table as input, and pass each row as a <key, value> pair to their mappers. The mapper gets the file system path of the text or image data, and emits a "Put" object containing the data as an intermediate input value for reducers. The reducers then store these "Put" objects into the text and image data tables.

3.4 Prospected Future Work

First, our immediate next step will be to implement MapReduce programs for building the Lucene index table. The programs will take rows of the bibliography table and text data table as input to mappers, which generate partial index data for each term, and the reducers will then collect the results from different mappers and generate the complete index table. The index table schema may be adjusted depending on performance considerations, and we will try to add scores to the index, either by adding new qualifiers to the existing table schema, or by creating a separate table for it.

Second, we will implement a distributed benchmarking program to evaluate the performance of the index system. The program will imitate the activities of multiple distributed clients, and generate queries based on samples of real use cases. Besides query efficiency, we will also evaluate the scalability of the index system as the number of HBase region servers increase, as well as the flexibility in terms of adding or removing documents.

Third, we will develop some MapReduce programs that perform data analysis on Lucene index data to demonstrate the advantages of our solution in MapReduce support. We plan to carry out these further steps in the next three months.

4. CONCLUSIONS

HBase is an open-source implementation of the BigTable system that originated from Google. Built with a distributed architecture on top of HDFS, HBase can support reliable storage and efficient access of a huge amount of structured data. However, it does not provide an efficient mechanism for searching field values, especially for full-text search. This paper proposes a distributed Lucene index solution to support interactive real-time search for text data stored in HBase. By hosting Lucene indices directly with HBase tables, we expect to leverage the distributed architecture of HBase and achieve high performance and better scalability and flexibility for the indexing system. By modifying the MyHadoop software package we developed MyHBase, which can dynamically construct a distributed HBase deployment in an HPC environment. Our experiments were conducted on the Alamo HPC cluster of FutureGrid, using data from a real digital library application. We have completed the data loading processes, and expect to continue with the index building and performance evaluation parts of our research within the next couple of months.

5. REFERENCES

[1] Apache Hadoop, http://hadoop.apache.org/.

[2] Apache HBase, http://hbase.apache.org/.

[3] Apache Hive, http://hive.apache.org/.

[4] Apache Lucene, http://lucene.apache.org/.

[5] Apache Pig, http://pig.apache.org/.

[6] Apache ZooKeeper, http://zookeeper.apache.org/.

[7] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating System Design and Implementation* (Seattle, WA, USA, November 06-08, 2006).

[8] Dumon, B. Building indexes using HBase: mapping strings, numbers and dates onto bytes. http://brunodumon.wordpress.com/2010/02/17/building-indexes-using-hbase-mapping-strings-numbers-and-dates-onto-bytes/.

[9] ElasticSearch, http://www.elasticsearch.org/.

[10] Katta, http://katta.sourceforge.net/.

[11] Krishnan, S., Tatineni, M. and Baru, C. MyHadoop – Hadoop-on-Demand on Traditional HPC Resources. San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego, 2011.

[12] Lakshman, A. and Malik, P. Cassandra - A Decentralized Structured Storage System. In *Proceedings of The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (Big Sky, MT, USA, October 10-11, 2009).

[13] Shvachko, K., Kuang, H., Radia, S. and Chansler, R. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies* (Incline Village, Nevada, May 03 - 07, 2010).

[14] Solandra, https://github.com/tjake/Solandra.

[15] Solr, http://lucene.apache.org/solr/.