Leveraging a Scalable Row Store to Build a Distributed Text Index

Ning Li Jun Rao Eugene Shekita
IBM Almaden Research center ({ningli,junrao,shekita}@us.ibm.com)

Abstract

Many content-oriented applications require a scalable text index. Building such an index is challenging. In addition to the logic of inserting and searching documents, developers have to worry about issues in a typical distributed environment, such as fault tolerance, incrementally growing the index cluster, and load balancing. We developed a distributed text index called HIndex, by judiciously exploiting the control layer of HBase, which is an open source implementation of Google's Bigtable. Such leverage enables us to inherit the support on availability, elasticity and load balancing in HBase. We present the design, implementation, and a performance evaluation of HIndex in this paper.

1. Introduction

Content-oriented applications such as hosted email, online shopping, and document archiving rely on text indexes for searching their data. Unlike text indexes built for the web, those applications require the index to be maintained incrementally, rather than rebuilt periodically. Similar to web indexes, many of those applications need to partition the index and serve it in a cluster. Therefore, in addition to the logic specific to the index, one has to worry about issues critical in a distributed environment. (1) Availability: how to failover when one of the nodes in the cluster dies? (2) Elasticity: how to grow/shrink the cluster incrementally without having to redistribute all existing data? (3) Load balancing: how to dynamically balance the workload across the cluster? All those issues make it hard to build a good distributed text index.

Over the last few years, in order to manage the large amount their internal data, Internet companies have built several scalable "row" stores, including Google's Bigtable [6], Amazon's Dynamo [9], and Yahoo's PNUTS [7]. These systems support a simple get/put interface by row keys and are used to manage data sets such as web crawls, hosted emails, and online shopping carts. They are designed to run on a cluster of hundreds to thousands of nodes, and are capable of serving data ranging from hundreds of terabytes to petabytes. Compared with traditional relational databases, those systems offer less querying capability and weaker consistency guarantees, but provide better support on availability, elasticity, and load balancing. If we build a text index by leveraging the distributed control layer in those systems, we may be able to reuse their technology for availability, elasticity, and load balancing, and therefore make the task of building a distributed text index much easier.

We developed a distributed index called *HIndex* by exploiting *HBase* [12], an open-source implementation of Bigtable. HIndex is incrementally updatable and is expected to be as scalable as HBase itself. We make the following contributions. First, we discuss critical design choices and the implementation of key components in our prototype. Second, we present an experimental evaluation of HIndex on a real data set, to measure the impact of certain design choices.

The rest of the paper is organized as follows. We provide an overview of Bigtable in Section 2. We describe the high-level design of our scalable index in Section 3. The implementation of key components in HIndex is covered Section 4. The results of a performance evaluation are explained in Section 5. We summarize related work and conclude in Section 6.

2. Bigtable Overview

Bigtable is a highly scalable row store. Each row is identified by a key and can have an arbitrary number of columns. Rows in a Bigtable are range-partitioned into tablets by the key. A Bigtable cluster has one *master* and a number of *tablet servers*. Each tablet server manages one or more tablets assigned by the master. The master maintains the metadata of each tablet and the mapping between tablets and their servers. To access a row, a client first contacts the master for the server of the tablet that contains the row key. The client then sends a request to the identified tablet server. Rows in each tablet are stored in a replicated distributed file system (*DFS*) called *GFS* [10]. Figure 1 depicts the overall architecture. Each box in a tablet server corresponds to a tablet and the text in the box specifies the key range covered by the tablet.

The distributed control layer in Bigtable provides the following functionalities: (1) Availability: When a tablet server fails, the master reassigns the tablets on the failed node to other tablet servers. Since the tablet data is stored in a DFS, it can be accessed from any tablet server. If the master fails, a new one is quickly re-elected on another live node. (2) Elasticity: As new tablet servers are added to the cluster, the master will move

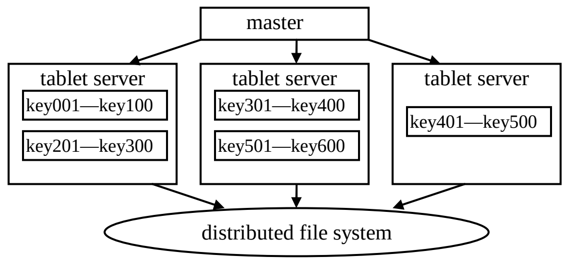

Figure 1. Bigtable Architecture

certain tablets from existing tablet servers to the new ones over time. (3) Load balancing: When a tablet becomes too large, a tablet server automatically splits it into two smaller ones and then lets the master assign them to tablet servers appropriately. Compared with some other row stores, Bigtable is unique in that it is an ordered table and relies on a DFS for fault-tolerance and availability. GFS is not a true POSIX file system. For example, files can't be updated in-place and can only be appended.

3. HIndex Design

A typical text index maintains a list of *terms* (a typical term a tokenized word). Each term points to a *posting list* that includes an ordered list of the document *IDs* (*docid*) of documents containing that term. A directory structure built on top of the term list is used for quick term lookups. To perform a search, the text index first locates the search terms, and then merges the posting lists of those terms to compute the matching document set. Modern text indexes are extremely efficient in merging posting lists (for any combination of unions and intersections), through zigzag-style joins [14]. Next, we discuss the design choices that we made for HIndex.

Index partitioning: There are typically two ways of partitioning a text index, by terms (*PBT*) or by docid (*PBD*). In PBT, each node serves a subset of the index terms and the full posting list of a term is stored on a single node. In PBD, each node serves a subset of the documents and a posting list in a node contains only documents in that node. PBT does not work well for us because we want to be able to update the index incrementally. To insert a single document, PBT requires updating a large number of terms, potentially stored in many nodes. This introduces significant communication overhead. A similar deficiency exists when evaluating a multi-term query. The posting lists for different search terms are likely stored in different nodes and intersecting them requires shipping at least one of the posting lists over the network. Thus, we adopt the PBD approach. To insert a document, we pretend to insert a row whose key is the docid and whose column value is the content of the document. Inside each tablet, instead of storing rows, we maintain a local text index for documents whose docid falls into the key range of this tablet. Since the terms for any given document are always stored on the same node, index terms can be updated locally and term intersections can be performed without shipping any posting list across the network.

When using PBD, we normally have to broadcast a search query to all nodes in the cluster. This may incur too much overhead in a large cluster. By taking advantage of the ordering property of Bigtable, we avoid broadcasting queries for certain applications with a careful design of docid. Suppose that we want to index all hosted email messages in a single distributed index. The docid for each email message can be chosen as the concatenation of the user ID and the message ID. This way, all emails for a particular user are clustered together and are likely stored in only a few nodes. When a particular user searches his emails, the queries need to be sent to only those nodes that have his data. Such a strategy is applicable to other multi-tenant systems. As another example, if an application prefixes each docid with a timestamp of when the document is created, then typical queries including a range constraint on creation time can be directed to a subset of nodes too.

Coupled or decoupled index: If a text index is built for data in a row store, our design allows the index to be either coupled or decoupled with the row store. In the coupled design, each tablet maintains a local row store as well as a local text index. The index is partitioned exactly the same way as the rows. In the decoupled design, the text index is maintained separately. While the coupled design enables tighter consistency between the data and the index, the decoupled design allows us to scale the text index differently from the row store. The rest of the paper focuses on the decoupled design.

Leveraging DFS: Like Bigtable, we choose to store the index data in a DFS. While DFS provides data redundancy and availability, we also need to consider its impact on performance. To that end, it helps to understand how a text index is organized. The storage model of a text index is very similar to that of a tablet in Bigtable. A text index typically maintains multiple index segments, each responsible for a set of documents. Segment files are written once and are never updated in-place. When new documents are inserted or existing documents are updated or deleted, a new index segment is created for the changes. All index segments are searched for querying and the results from each segment are merged to form the answer. Over time, smaller segments are merged into bigger ones to reduce search overhead. Segments from the previous generation are garbage-collected over time. There are a couple of reasons why it is appealing to store the index data in a DFS: (1) Since segment files are never updated in-place, caching in the DFS becomes simple. (2) Because each index partition is managed by a single tablet server, segment files are always written and read from the same node. This makes it is easy for the DFS to deal with read/write consistency.

Exploiting Bigtable: There is a list of critical features that we obtained for free from Bigtable. These functionalities include automatic failover of tablet servers, incrementally adding new nodes, and load balancing.

4. Implementation of Key Components

We developed HIndex by modifying HBase as of May, 2008. HBase has been under development for about 2 years and has a fairly active community. HBase stores all its data in the *Hadoop* [11] DFS (*HDFS*), an open source implementation of GFS. Our implementation also exploits *Lucene* [15] (an open source text index library) for building the local text index.

Document Insertion: We implemented a new API for inserting documents one at a time. The client submits a <docid, Document> pair. Our document model is extensible. A document can be plain text, <field, value> pairs or a JSON [13] object. We treat the docid as the row key in HBase and use it to route the document to the right tablet. We extended HBase to support a new type of tablet for indexing (called *index* tablet). An index tablet maintains an on-disk Lucene index as well as an in-memory one. A new document is first appended to a write-ahead log, stored in HDFS. The document is then tokenized and inserted to the in-memory index. Periodically, the in-memory index is flushed to HDFS as a new index segment, which becomes part of the on-disk index. While flushing the current in-memory index, we open up another one to accommodate concurrent insertions.

Over time, segments of the on-disk index have to be merged. A simple way is to let each local Lucene index manage segment merging independently. However, since typically there are multiple index tablets on a tablet server, different Lucene indexes may decide to start merging at about the same time. This can cause resource contention and slow the whole tablet server down. The same contention can happen if we flush each index tablet on the same tablet server independently. Instead, we manage segment merging and flushing at the tablet server level. If an index tablet wants to merge segments or flush the in-memory index, it adds a request to a queue. A separate thread in the tablet server processes the requests in the queue one at a time in the order of their arrival. Since the tablet server has the global view of the node, it can do a better job at scheduling the request based on the available resources. Note that segment merging does not reduce the availability of the index since it happens in the background.

Region Splits: As an index tablet grows too big, it is split into two smaller ones. When a split happens, the tablet being split is taken offline temporarily. To increase the availability of the system, we employ a strategy called *lazy-split* to complete a split quickly. We split an index tablet I (call it the *parent* tablet) in the following steps: (1) Take I offline and flush the in-memory index of I to HDFS. (2) Create two empty child index tablets I_1 and I_2, each responsible for half of the range of the docids in I. (3) Make two copies of I, one for I_1 and one for I_2. We achieve this quickly by creating two symbolic links for every index segment file in I, and adding each link to I_1 and I_2, respectively. (4) We then logically remove those documents not in the range of I_1 (I_2), by applying a proper filter over docid on every search result. (5) Make I_1 and I_2 online by letting the master assign them to tablet servers (which can be different from the node serving I). As one can see, the only step in the above process that involves real data movement is step (1). The rest of steps can all complete instantaneously. Therefore, the unavailability window mostly depends on the maximal in-memory index size, which can be configured according to an application's requirement. Lazy-split is possible because of the exploitation of a DFS. It trades performance for availability by deferring the real cost of a split to a later time.

After a child tablet becomes available, it starts to receive new documents falling into its key range. We pay the real cost of a split when a segment merge on the tablet is triggered. During the merge, we physically delete all documents out of its range, generate merged new segments, and remove the symbolic links and the filter. Merging does not reduce availability of the index tablet because it is non-blocking. However, it does affect performance and we study this in Section 5. To simplify the splitting logic, an index tablet cannot be split again until all symbolic links have been removed from its segment files. After a split, the parent tablet is eventually deleted after neither child tablet references it. This is handled by a thread in the master that periodically scans through the metadata of all tablets and garbage-collects any "dangling" tablets that are no longer referenced. We note that the idea of lazy-split is inspired by a similar design in HBase for regular tablets.

Bulk Load: When an HIndex is created, initially there is only a single index tablet responsible for the full range of docid. Therefore, all insertions will be hitting on a single tablet server. On a large cluster, it may take a while before enough tablets are created through splitting. To increase the scalability of bulk loading, we added a new interface so that when creating an HIndex, a user can provide a list of consecutive docid ranges. We then allocate an empty index tablet for each of the specified ranges. To use this feature, a client can sample the docid range from the documents to be bulk-loaded and request a list of index tablets to be pre-allocated. After the index is created, insertions can be done in parallel using multiple threads, each inserting documents to a different index tablet. The same idea can also be used for appending a large number of documents with docid larger (or smaller) than any document in an existing index. Bulk-loading rows into an ordered table has been recently investigated in [18]. Our approach, although simpler, works quite well in practice, since tokenization during index building is CPU intensive.

Search: We added an interface to search documents stored in HIndex. In addition to a set of search terms, the client can provide a list of docid ranges to be searched. Once a search is submitted, our client library first identifies a set of tablets that cover the specified ranges. It then sends the search query in parallel to the identified tablet servers. Each tablet being contacted issues a local search request to the Lucene index that it manages, and returns a list of matching docids. The client library accumulates the matches from each of the tablet and returns the final result. This interface is potentially useful for any multi-tenant system. For example, in the hosted email application mentioned earlier, to search emails for a particular user, say "u1". A client can specify a docid range ["u1", "u2"). This way, we avoid the overhead of broadcasting the query to all index tablets. Instead, the query is directed to only those tablets that store at least some email messages for "u1".

We have to deal with the rare but possible situation when a tablet splits in the middle of a search. If we are not careful, a search may not return the complete set of matches. To that end, for each index tablet searched, the client library remembers its docid range before sending the search request. Each index tablet, in addition to the list of matches, also returns its range when the local search is actually performed. The client library then compares the returned range with the remembered one. If they don't match (the returned range should be a subset of the remembered one), the tablet must have split between the time that the client sends the search and the time that the local search is performed. When this happens, the client library issues a compensation search on a subset of the remembered range not yet covered. This process continues until all returned ranges match the previously remembered ones.

5. Experimental Results

We designed a set of experiments to show: (1) the impact of the design choices that we made including, the use of a DFS to store and serve a text index, lazy-split, pre-allocating index tablets, and partitioning by docid; (2) certain scalability features that we inherited from HBase. Our experiments were performed on a cluster of 10 Linux nodes, each with 8 2.1GHz cores, 16GB of memory, and 8 directly attached SATA drives. All 10 nodes were connected to a 1Gb Ethernet switch. To analyze the impact of storing the index data in a DFS, we experimented with HDFS 0.17.1 and GPFS v3.2.0.1. GPFS [19] is more complicated than HDFS since it supports true POSIX semantics, and is also more mature than HDFS. For both HDFS and GPFS, we set the replication factor to be 2. Both HDFS and GPFS were set up using all 10 nodes in the cluster. Each block in HDFS was 64MB and was striped across 8 disks on a single node. Data in GPFS was widely striped across all 80 disks in units of 4MB. For comparison, we also tested HIndex's performance on a local file system (referred to as *local*) when running a single tablet server. The local file system was striped across all 8 disks using software raid 0. The data set that we used was the text description of 6 million U.S. patents [17]. Each patent was treated as a document and had an average size of 50KB. The patent number was used as docid and it contained the issue date as prefix. We configured HIndex so that each index tablet stored about 200K documents and each tablet server ran on a separate Linux node.

Insertion: We first present the insertion results. All insert clients read data from a parallel database in a separate cluster. The database was never the bottleneck in our tests. In the first test, we started HIndex with a single tablet server and inserted a total of 200K documents into an empty index. No splits occurred during the insertion and the final index tablet was about 2GB. Table 1 reports the insertion time (in thousands of seconds) on different file systems and using a various number of clients. The insertion time was not affected much by the choice of the underlying file system. The reason is that the insertion cost was dominated by tokenization, which was extremely CPU intensive. The performance on GPFS and HDFS were comparable, and were slightly worse than the local file system because of the replication overhead. When writing an index file, HDFS stored one copy of every block in the local node

	local	hdfs	gpfs
1 client	4.09	4.29	4.21
3 clients	2.84	3.19	3.29
6 clients	2.62	2.98	2.91

1. 200K inserts to 1 tablet (K secs)

	with split	w/o split
hdfs	8.95	4.48

2. 3 clients (200K inserts each) , 1 tablet server (K secs)

#clients	3	15	30
#tablets/#servers	3/1	15/5	30/10
hdfs	4.48	7.81	10.28

3. 200K inserts per client, no split (K secs)

	local	hdfs	gpfs
cold	39	9	63
warm	253	10	193

4. search 1 tablet (queries/sec)

	with filter	w/o filter
cold	44	73
warm	207	227

5. search 1 tablet on gpfs (queries/sec)

#tablets	1	5	10	20
warm (gpfs)	291	281	199	69

6. search 1 to 20 tablets concurrently (queries/sec)

and the other copy remotely. In contrast, GPFS striped an index file to all 80 disks and thus most blocks were remote to the tablet server. As we increased the number of concurrent clients, the insertion time decreased. However, the decrease was sub-linear in the number of clients because of the overhead of synchronization among the insertion, flushing, and merging threads.

Next, we measured the overhead of splitting an index tablet and the benefit of tablet pre-allocation. In Table 2, we compare the time to insert a total of 600K patents using 3 clients into an HIndex with a single empty tablet and another with 3 pre-allocated tablets. During insertion, the former split 3 times and the latter had no split. Insertion with splitting was twice as expensive as that without splitting. Splitting a tablet itself was reasonably fast because of lazy-split. In our test, the average time to split a tablet was less than 2 seconds. However, subsequent segment merging became more expensive since it did the "real" work of a split, such as removing out-of-range documents. The longer the merge, the more likely it would compete with ongoing insertion for resources. We then tested the scalability of insertion with no splits. As we increase the number of insertion clients (each inserting 200K patents) from 3 to 30, we pre-allocate index tablets proportionally. Table 3 shows the result. Had the insertion scaled perfectly linearly, we would expect the numbers in Table 3 to be almost unchanged. In reality, there was a difference of about a factor of two between inserting 600K and 6 million patents. Such a difference was mainly due to variance in document sizes.

Search: We now present our experimental results on the search performance. We generated a set of random search queries, each returning up to 128 matches per index tablet. Every query had two required search terms. To evaluate a query, the text index had to zigzag through two posting lists, each starting from a random location in the segment file, to perform the intersection. In each test, a single client issued a certain number of search queries to a pre-loaded HIndex. Both cold and warm numbers were measured. In the first test, we directed all queries to a single index tablet. The tablet didn't contain any symbolic links or filters. Table 4 shows the search result in number of queries completed per second. Let's start with the cold numbers. Searching on HDFS was more than 70% slower than that on local. Both HDFS and local read the index data from 8 local disks. In the case of HDFS, it supported read affinity and always preferred serving data from a local copy over the remote one. The overheads of HDFS include (1) serving data from a socket (to localhost); (2) reopening a socket for every random read (no socket caching); (3) verifying checksums (a checksum of 4 byte is computed for every 512 bytes). Most overheads were from the first two items. Currently, HDFS is designed primarily for running MapReduce [8] jobs and isn't optimized for random accesses. GPFS performed better than local. This is mainly because GPFS served data from all 80 disks (since it did wide striping). Therefore, for the same amount of data, disk seeks on GPFS were shorter than that on local. For the warm test, the HDFS number was much worse than local. This is because HDFS does not have client-side caching and warm data was still served through sockets. All overheads listed above still existed in the warm test and they became more significant when there was no disk I/O overhead. GPFS has client-side caching and its number was about 25% worse than local. Most overheads in GPFS came from locking and token validation for maintaining cache consistency among all nodes in the cluster. Such overheads are unnecessary for HIndex since each index tablet is always written and read from a single node.

Next, we measured the search overhead because of lazy-split. We issued a set of queries to two index tablets containing the same set of documents, one with symbolic links and filters because of a recent lazy-split, and one without. The numbers are reported in Table 5. Lazy-split added about 30% and 10% performance penalty for cold and warm searches, respectively. This is what we pay for higher availability. Note that such overhead is temporary and disappears after the next merge.

By taking advantage of PBD, we don't have to broadcast every query. We tested the scalability of search by varying the number of index tablets to be searched from 1 to 20. The tablets were evenly spread around 10 nodes. Since warm performance is more crucial for search, we only report the warm number on GPFS. Table 6 shows the result in queries per second. For ideal scalability, we should expect queries per second remains constant as we increase the number of tablets to be searched. The drop in performance as we increase the number of tablets was due to (1) variance in matches from different index tablets, and (2) the overhead of merging the matches from all index tablets.

Availability: In the last experiment, we measured the availability of HIndex when failure occurred. We killed one of the tablet servers and measured the time for all 3 index tablets on the killed server to be available again. The elapsed time was about 2 minutes and can be broken into 3 parts: (1) 67 seconds for the master to expire the lease on the killed tablet server; (2) 37 seconds for the master to start processing the killed server from a delayed queue; (2) 2 seconds for all index tablets to be reassigned and made available at the new servers. Note that all three parts can be shortened by customizing certain configuration properties. This quick failover scheme is one of the features we inherited from HBase.

We summarize our experimental results as follows: (1) the file access pattern in HIndex makes it possible to build a DFS that is much simpler than a POSIX file system such as GPFS, and offers performance close to that of a local file system; (2) lazy-split trades short-term performance for availability; (3) pre-allocating index tablets improves performance and scalability.

6. Related Work and Conclusion

There exists a rich set of work on building distributed text index for web search [2][3][4][16]. Those systems use sharding for scalability and replication for fault tolerance. However, shards are not incrementally maintained and are rebuilt in batches. Google's App Engine [1], built on Bigtable, only supports secondary value indexes [20], not text index. The value index is built using another instance of Bigtable whose row keys are used as index keys. In addition to Bigtable, there are a few other distributed row stores with different design points. *Dynamo* [9] is a highly available and scalable key-value store developed and deployed at Amazon. It employs an eventual consistency model to achieve high availability. *PNUTS* [7] is a distributed system that manages structured data for Yahoo!'s web applications. Its data model is similar to that of Bigtable and it supports per-row timeline consistency. PNUTS uses the Yahoo! message broker, a publish/subscribe system, for both redo logs and replication. Facebook open-sourced *Cassandra* [5], which combines the distributed architecture used in Dynamo and the data model used in Bigtable. It is possible to leverage those systems too.

In this paper, we described our experience of building a distributed text index by leveraging the scalable control layer in a row store. Such leverage enabled us to reuse the logic on availability, elasticity, and load balancing. Our experimental study demonstrated the tradeoffs of the design choices that we made in HIndex.

REFERENCES

[1] AppEngine. http://code.google.com/appengine/

[2] A. Arasu, et al: Searching the Web. ACM Transactions on Internet Technology, Vol. 1

[3] L. Barroso, et al: Web Search for a Planet: The Google Cluster Architecture. In IEEE Micro, 2003.

[4] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks, 1998

[5] http://code.google.com/p/the-cassandra-project/

[6] Fay Chang, et al: Bigtable: A Distributed Storage System for Structured Data, OSDI 2006

[7] Brian Cooper, et al: PNUTS: Yahoo!'s Hosted Data Serving Platform, VLDB 2008

[8] Jeffrey Dean, et al: MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004

[9] Giuseppe DeCandia, et al: Dynamo: amazon's highly available key-value store. SOSP 2007

[10] Sanjay Ghemawat, et al: The Google File System. SOSP 2003

[11] Hadoop. http://hadoop.apache.org/core/

[12] HBase. http://hadoop.apache.org/hbase/

[13] JSON. http://www.json.org/

[14] Xiaohui Long, et al: Optimized Query Execution in Large Search Engines with Global Page Ordering, VLDB 2003

[15] Lucene. http://lucene.apache.org/

[16] S. Melnik, et al: Building a Distributed Full-text Index for the Web. ACM Trans. Inf. Syst

[17] Patent dataset: http://www.nber.org/patents

[18] Adam Silberstein, et al : Efficient Bulk Insertion into a Distributed Ordered Table, SIGMOD 2008

[19] Schmuck, Frank, et al: GPFS: A Shared-Disk File System for Large Computing Clusters, FAST 2002

[20] http://snarfed.org/space/datastore_talk.html