Distributed Indexing of Web Scale Datasets for the Cloud[*]

Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos and Nectarios Koziris
Computing Systems Laboratory
School of Electrical and Computer Engineering
National Technical University of Athens
{ikons, eangelou, dtsouma, nkoziris}@cslab.ece.ntua.gr

ABSTRACT

In this paper, we present a distributed architecture for indexing and serving large and diverse datasets. It incorporates and extends the functionality of Hadoop, the open source MapReduce framework, and of HBase, a distributed, sparse, NoSQL database, to create a fully parallel indexing system. Experiments with structured, semi-structured and unstructured data of various sizes demonstrate the flexibility, speed and robustness of our implementation and contrast it with similarly oriented projects. Our 11 node cluster prototype managed to keep full-text indexing time of 150GB raw content in less than 3 hours, whereas the system's response time under sustained query load of more than 1000 queries/sec was kept in the order of milliseconds.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems

General Terms

Design, Performance

Keywords

Cloud Computing, NoSQL, Hadoop, HBase, MapReduce.

1. INTRODUCTION

The cloud computing paradigm has been receiving an increasing amount of attention from both the industry and academia. On-demand and pay-as-you-go access to computational and storage resources that reside in distant data centers is a very attractive business model, especially for small to medium sized enterprises (SMEs) or start ups that need a quick, cheap and scalable access to hardware and software infrastructure. According to a recent survey[1], more than half of SMEs are going to use cloud computing services this year, compared to a mere 22% last year.

Our era is marked by what is referred to as the "data explosion": Increasing volumes of data that need to be stored, indexed and queried for every company (such as e-mail and web logs, historical data, click streams, etc). Cheaper storage and bandwidth enables the growth of publicly available datasets: sites like the *Internet Archive* offer access to petabytes of content such as web pages, books, etc. Another example is Amazon offering public datasets (almost) for free through its cloud infrastructure[2]. Effective indexing is very important to enable the dataset usability, but this is a very difficult task for such data volumes: even Internet Archive does not allow full text search in one of its most interesting services, the WayBackMachine[3] with a dataset of 4.5 PB.

The requirement to perform compute-intensive analytics on (semi) structured bulk datasets has pushed sql-like centralized databases to their limits [3]. This fact, along with the highly parallel nature of these tasks, has lead to the development of horizontal scalable, distributed non-relational data stores, called NoSQL databases [1]. Google's Bigtable [6], Amazon's Dynamo [8], Facebook's Cassandra [10], and LinkedIn's Voldermort [2] are a representative sample of such systems. In favor of scalability and high availability, NoSQL systems relax classic ACID guarantees made by typical DBMS, allowing, for instance, only eventual consistency. NoSQL systems serve a dual purpose: they can efficiently store and index arbitrarily big data sizes while enabling a large amount of concurrent user requests. NoSQL systems are perfect candidates for cloud infrastructures, as their shared nothing architecture enables them to scale by simply acquiring more computational and storage resources from a cloud vendor.

Our contribution: In this work, we present a distributed processing platform suitable for indexing, storing and serving large amounts (in the orders of TB and more) of content data under heavy request loads. Indexing rules of variable granularity, relative to both type of data and user-input, along with the raw content are processed by our framework. The augmented information is extracted in the form of a distributed index served to an arbitrarily large number of concurrent users. In order to speed up indexing, the compute and storage-intensive index creation and maintenance leverages the innovative *MapReduce* framework. To achieve low response times in high query load using commodity-node clusters, user requests are served through an *HBase* database. In this paper, we present an initial prototype of the platform that aims to achieve the following goals:

Support of almost any type of data: Support for various types of content, including unstructured (e.g., log entries, HTML files, etc), semi-structured (XML files) and

[*]This work was partly supported by the European Commission in terms of the GREDIA FP6 IST Project (FP6-34363).

[1]http://bit.ly/Cloud2010

[2]http://aws.amazon.com/publicdatasets

[3]http://www.archive.org/web/web.php

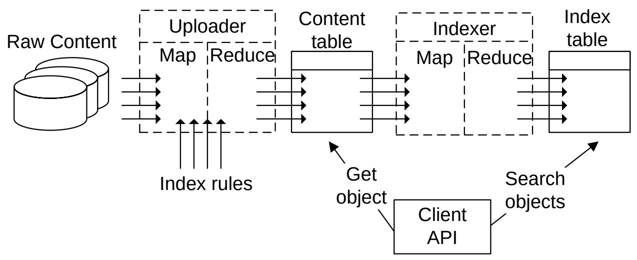

Figure 1: System Architecture

structured data (sql databases).

Near real-time query response times: Query execution times should be in the order of milliseconds, to enable users to perform fast content searches. We are not considering "on the fly" heavy analytical tasks to execute as user queries (as Pig [13] and Hive [14] frameworks do), since these tasks are not suitable for "live" searches, due to their long execution time (from minutes to hours).

Scalability: This is a requirement both in terms of storage space and concurrent user requests. The system must be able to scale (preferably in an elastic way) by simply adjusting the number of participating server nodes.

Ease of use: Users should be able to define simple rules to declare the scope and type of the required index and perform meaningful searches over the data (e.g., find conferences whose title contains `cloud` and were held in `California`).

2. ARCHITECTURE

Our system is built on top of *Hadoop*, an open source java implementation of the MapReduce paradigm that has been widely adopted by a large number of organizations worldwide[4]. Hadoop consists of a distributed file system called HDFS (a GFS [9] - like distributed file system) and a MapReduce [7] executing framework on top of it. HDFS file metadata is being kept in a single node called the NameNode and raw data is stored in many DataNodes. MapReduce consists of a job scheduler called JobTracker which co-ordinates many worker nodes called TaskTrackers. In a typical setup, the NameNode acts as a JobTracker, and DataNodes are also TaskTrackers.

As our NoSQL storage substrate we are using *HBase*, which is an open source implementation of Google's Bigtable [6], since it is tightly coupled with Hadoop (it uses HDFS as its storage backend and provides I/O hooks to Hadoop's MapReduce framework). An HBase table consists of a large number of sorted rows indexed by a row key and columns indexed by a column key (each row can have multiple different columns). Actual content is stored in HBase cells: an HBase cell is defined by a combination of a row and a column key, in the same way an (x,y) value defines a point in a 2-dimensional space. The primary key of an HBase table is the row key. HBase supports two basic lookup operations on the row key: exact match and range scan. HBase consists of a single master (HMaster) that keeps track of numerous nodes that serve actual content called RegionServers.

Overview: In Figure 1, we present an overview of the system components along with their interactions. The main idea is the following: The raw content (in the form of large XML files, HTML files, SQL database dumps, logfile directories, etc) is submitted to HDFS. The content along with some instructions (the indexing rules) is fed to the Uploader, which is a MapReduce program. The Uploader creates an HBase table with the content in a record oriented view. A second MapReduce task (Indexer) takes as input the Content table, and extracts the Index, which is also stored as an HBase table. Users perform queries using the client API. The API contacts the Index table to perform searches, and the Content table to serve objects. In the following, we give a detailed presentation of the system components.

Index rules: According to the content type, users provide the system with instructions of what to index. In this phase, there is also a need to specify what are the boundaries of a single "record". Records are used to split the entire dataset in a number of distinct entities that will act as processing units. For instance, in the XML case, record boundaries can be considered some specific tags. In the unstructured content situation (e.g. in HTML files) a specific record can be a single HTML document, whereas in the database case, records are table rows. It is important to note that our system can adjust the granularity of the "record" according to the user/application requirements. Apart from this, users also select what specific content regions (attribute types) they want to index (such as text contents of a specific XML tag, contents of an HTML table or paragraph, data from a specific table or column from a database, etc). Index rules are used from every component (Uploader, Indexer and the client API).

Uploader: The Uploader class reads bulk datasets previously uploaded to HDFS, and creates the Content table. The Content table acts as a content hashmap, where every row contains a single record item and the row key is the MD5Hash of the record content. The Uploader class reads data input from HDFS and creates the content table using the MapReduce paradigm as follows: In the Map phase, mappers read from HDFS and emit a list of <key, value> objects where, the key is the MD5Hash of the record, and the value is an HBase cell containing the content. The reduce phase lexicographically sorts the incoming <MD5Hash, HBase cell> key values according to the MD5Hash, stores the results in HFiles (HBase data file format), and informs HBase of the location of the new table data files. The use of the content table is twofold: first, it allows fast random access reads during successful index searches. Second, it enables easy content manipulation in the case of new item additions or deletions of old records.

Indexer: The Indexer module calculates an inverted list of index terms and document locations and stores it in the Index table. The row key of the index table (primary key) is the keyword term followed by the attribute type on which this keyword was encountered (e.g., if the keyword `google` was found in a <revision> tag, then the row key will be `google_revision`). Every row stores a list of MD5Hashes of the records that contain this specific keyword (e.g., `google`) in a specific attribute (e.g., revision). This list actually links the Index to the Content table. The Indexer is a MapReduce task that works as follows: in the Map phase, mappers process the Content table, and emit a <keyword_attribute, MD5Hash> key-value pair for every keyword they encounter. Reducers receive all emitted key-values for a specific key and aggregate them in one key-value pair of the type <keyword_attribute, list(MD5Hash)>. These key-value pairs are finally stored in HDFS (HFile format), and HBase is informed

[4]http://wiki.apache.org/hadoop/PoweredBy

about the new table.

The reason for bypassing the HBase API during initial table creations is that it is many times slower for bulk insertions, as it is explicitly stated by the HBase developers. This happens because insertions are first written to a persistent write-ahead-log and are afterwards accumulated in a memstore (a sorted, in-memory buffer). When the memstore is full, they are flushed to the disk in HFiles. By directly storing HFiles, we avoid expensive intermediate interactions with the write-ahead-log and the memstore for each new object.

Client API: The client API provides the basic search and get operations and it is built on top of HBase's client API. Our index design allows us to perform google-style freetext queries over multiple indexed attributes. Users are able to search content in a specific attribute type (e.g., find all documents that contain the keyword `google` in the <title> tag) or in any attribute type. In the first case, the user query is translated in an HBase point query with the parameter "google_title", whereas in the second case it is translated in a range scan in all the HBase rows that start with the prefix "google_" (one for each attribute type), which by default are lexicographically adjacent. More complex range queries of the type: "find all documents that have a creation date in 2009" or prefix queries such as "find all the documents that contain a keyword goo*" are also supported and are translated in an HBase range scan. In any case, the client contacts HBase once for every query, even if this is a range scan, and the network overhead between the client and HBase is only one round-trip message per query. Query results consist of a list of MD5Hashes of the matching documents that can be retrieved with a simple lookup from the Content table. Our system also enables `AND`-ing and `OR`-ing of queries through client side processing: queries are executed for each dimension and query results are merged by the client to provide the final list of the matching documents.

3. EXPERIMENTAL RESULTS

Our experimental setup consists of 11 worker nodes and a single machine in the role of HDFS, MapReduce and HBase master. The worker nodes have 2 Quad-Core E5405 Intel Xeon®CPUs @ 2.00GHz, 8 GB of RAM (with disabled page swapping) and a 500GB disk (for a total of 88 CPUs, 88 GB RAM and 5.37 TB of disk space), while the master has similar CPUs and disk, but only 2 GB RAM. The version of Hadoop is the latest 0.20.1, re-compiled from source to suite our setup. HBase version is 0.20.2.

The Hadoop MapReduce framework is configured to take full advantage of the available resources. Hadoop and HBase are each given 1 GB of memory in every running machine, and each Mapper or Reducer task is given 512 MB of RAM. Each worker node can spawn 6 Mappers and 2 Reducers running concurrently, for a total cluster capacity of 66 Mappers and 22 Reducers. We have disabled Hadoop's speculative execution, where every task is executed 3 times for redundancy, as this would drop the cluster's effective capacity to one third. HBase is managing its own Zookeeper (an Apache project providing a distributed consistency system) instance with 3 quorum nodes. HDFS was configured with a replication factor of 2.

During MapReduce execution, the framework automatically sets the number of Map tasks. To enable full cluster CPU utilization, Reducers for all jobs were manually set to 100. Any number beyond the cluster concurrent capacity of 22 would be sufficient, but a larger number minimizes the effect of failed reduce tasks on job completion time.

Table 1: Content table creation time for various dataset sizes and types

XML		HTML		DB		TXT	
size GB	time min	size GB	time min	size GB	time min	size GB	time min
5	7	5	9	1	3	1	3
10	44	10	42	6	9	5	15
50	192	50	202	12	12	10	33
150	576	150	601	23	20	20	70

Our datasets were downloaded from Wikipedia's dump service[5] and from project Gutenberg's custom DVD creation service[6]. Our structured data comprises of a 23GB MySQL database dump of the latest English version of Wikipedia. The structured dataset was obtained from the current MediaWiki XML dump available at the Wikipedia download site with the use of `mwdumper`[7] and uploaded to a local MySQL 5.0.51 database instance, from which a new SQL dump was obtained to form the basis for our experiments. The reason for this process was our desire to have a dataset consistent with actual MySQL dumps.

Our semi-structured dataset comprises of one XML and one HTML dataset: the XML dataset is a 150GB part of a 2.55TB uncompressed XML dump of every English Wikipedia page along with its revisions up till May 2008. The HTML dataset in turn is a 150GB dump containing a static version of Wikipedia from June 2008.

Our unstructured data is a full dump for all languages of Gutenberg's text document collection. The dataset comprises of approximately 46,300 text files, that take 20 GB of hard disk space.

To locate attribute types during indexing creation, in the structured (database dump) and the unstructured (text files) case we utilized simple regular expressions, whereas in the semi-structured (XML and HTML) case Xpath expressions were used. In the case of HTML, all documents were filtered through `TagSoup`[8], a parser that converts HTML to well formed XHTML documents.

In order to create query traffic, we utilized a publicly available dataset from AOL that contains twenty million search keywords for over 650,000 users over a 3-month period[9] to calculate a zipfian query frequency distribution. During our experiments, clients were generating both point and prefix queries based on that distribution. The advantages of the AOL keyword dataset compared to a random keyword generator is that it follows a real-life, non-uniform skewed distribution, where popular keywords are requested more often. The experiments try to clarify the performance of our indexing system under heavy user load when using a reasonably large number of structured, semi-structured and unstructured data.

3.1 Content table creation

In this section, we present our findings during the content

[5]http://download.wikimedia.org/
[6]http://snowy.arsc.alaska.edu/pgiso/
[7]http://www.mediawiki.org/wiki/MWDumper
[8]http://home.ccil.org/~cowan/XML/tagsoup/
[9]http://techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data/

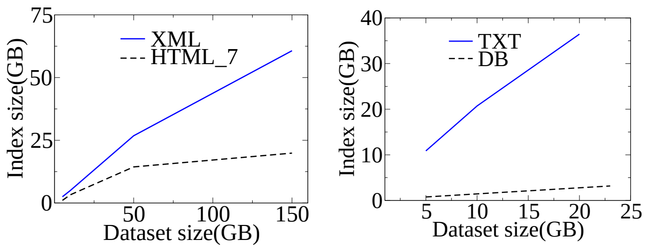

Figure 2: Index size for various datasets

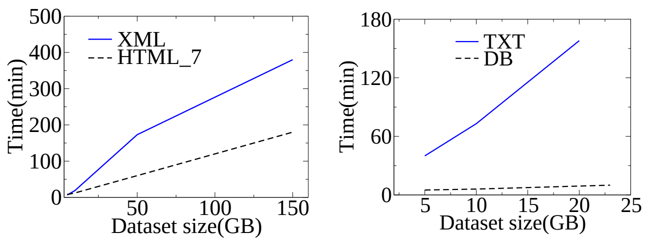

Figure 3: Index time for various datasets

Table 2: Index size and creation time for different numbers of attribute types (5GB HTML).

Iteration No	Indexed tags (count #)	size GB	time min
1	[table,li, p,b, i,u, title] (7)	1.049	7
2	1 + [h1, h2, h3, h4, h5, h6, big] (14)	1.097	6.5
3	2 + [blockquote, del, em, s, small] (19)	1.117	9.5
4	3+ [strong, sub, sup, tt, pre, dt, dd, font] (27)	1.296	11

Table 3: DB Index creation time vs # of nodes.

# of Nodes	time(min)
2	34
4	23
6	16
8	11
11	10

table creation procedure. In Table 1 we depict the time it took for the Uploader to create the content tables from raw HDFS data for bulk insertion into HBase. The tests were run using all the available nodes.

From Table 1 we can deduce that our system exhibits the expected behavior for bulk insertion of the dataset. The time to completion increases linearly with the size of the dataset and is comparable between different types of data. The diversion from the norm of the structured dataset can be explained by taking into account the reduced processing required for the SQL dataset. Similarly, the larger than expected time to completion for the plain text Uploader is justified, given the fact that it is composed of a large number of very small files. Thus, the initialization penalty for each Mapper is high compared to the processing, and though small (about 1 second), makes a significant difference given the average time to completion for each file (2-5 seconds).

3.2 Index table creation

In this section, we present our experiments during the index table creation. We are interested in the index size and creation time. Index tables are created from the content tables described in Section 3.1. In Figure 2 we present the growth of the index size as the content table increases. The first figure presents index growth for the HTML and XML dataset, whereas the second figure depicts index growth for the DB and TXT dataset. In Figure 3 we present the indexing time of the aforementioned four dataset types for various sizes. In the HTML_7 case, the indexed attributes are the first 7 HTML tags from Table 2, whereas in the XML case, <content> and <timestamp> tags are indexed.

In Figure 2 for the XML and HTML inputs we notice that in any case the growth of the Index table gets smaller with the increase of the dataset: this happens because, after a certain Content size, indexed terms size is not significantly increased. What is more, we notice that XML index is larger than the HTML index of the same dataset size: HTML dataset contains a lot of formatting code that gets stripped during keyword extraction, whereas in the case of XML there is (almost) clean text. In the case of DB and TXT, the variations between structured and unstructured data as to the size are justified, as the structured dataset is already indexed in a reasonable way. Moreover, the diversity between the different documents contained in the Gutenberg collection (in terms of language and topic) means that a lot more terms have to be indexed, increasing the size of the index because of the added metadata.

In Figure 3 for the XML and HTML we notice that the XML dataset is more demanding in terms of processing time compared to the HTML dataset, because of the HTML formatting code that gets stripped during indexing. In the TXT and DB case, the structured DB dataset is the most easily indexed, as the system does not perform much re-indexing except for re-arranging the data to fit its internal specifications. The TXT dataset obviously requires much more time, as more processing is needed to extract and process metadata. Both datasets however exhibit near linear scalability as the number of data increase, but with different angles as explained before. This is a desirable outcome for the Indexer, as it highlights its robustness and good behavior under these tests.

We now present experiments that show how our indexing mechanism responds when we vary the number of the attribute types. We utilize the 5GB HTML dataset and we increase the number of attribute types in every iteration, as we depict in Table 2: in every iteration, all the tags from the previous iterations are included. In Table 2 we present the growth in the indexing time and size for every iteration. As expected, both the size and the index time increases along with the attribute type number. Nevertheless, this increase rate remains relatively low.

In the following experiment, we measure the scalability of the indexing mechanism by varying the number of the cluster nodes while keeping a stable index input dataset size. We run the Indexer on the largest DB dataset and in Table 3 we present the index creation time variation. As expected by the fully distributed indexing nature of MapReduce, the speed is proportional to the number of processing nodes, something that proves the system's scalability during index creation. This property is a typical cloud application requirement, since extra nodes are acquired by a cloud vendor in an easy and inexpensive manner.

3.3 System performance under query load

In this section we measure our system's ability to respond to a large number of simultaneous user queries. We consider

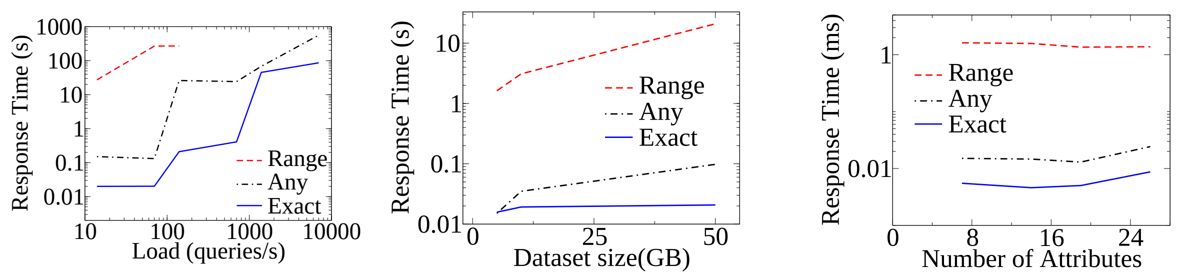

Figure 4: Response time of queries vs system load in queries/s, original data size and number of indexed attributes.

three types of queries on indexed attributes: free keyword search in a specific attribute, free keyword search in any attribute and prefix queries in any attribute type. Prefix queries were generated using the first four characters of randomly selected AOL keywords. The test obtains response times from HBase for these search types. Client instances were run concurrently on 14 machines, to ensure realistic measurements in terms of network load from different machines. Most of the tests were executed using a 14GB index table of a 50GB HTML dataset subset. For the response times vs data size, 5, 10 and 50 GB partitions of the HTML dataset were used with 7 indexed attributes (as seen in Table 2).

Our first experiments evaluate the maximum load measured in queries/s that our index table can support in HBase. We have observed the limit of HBase when serving queries by running 14 concurrent clients, each sending queries with a delay that follows an exponential distribution probability function. Our experiments (results seen in Figure 4a) were run with values higher than the previously reported limit for sustained queries per second against HBase on HDFS[11]. Although high response times were measured, the system remained stable and kept serving requests even under heavy load, highlighting the robustness of HBase. Range query loads above 140 queries/s however failed to complete and in most cases the clients had to be manually terminated. This is reasonable, as such queries request a large number of data and demand processing on both the server and client, increasing exponentially the load on available resources. For reasonable load on the server, in the order of 14 queries per second from different clients, we have measured response times close to 20 ms for point queries, 150 ms for any attribute queries and 27 seconds for range queries.

We believe that the observed behavior of the system in Figure 4 is a consequence of HBase caching: up to 100 queries/s there are available channels to accommodate the clients. Beyond this point, the response time increases because of the increased client requests, who now have to wait in line to be served. Between roughly 100 to 1000 queries/s, HBase caching is significant: each popular keyword is loaded only once in memory and then served as is. This leads to a significant decrease of the average response time of the system because of the skewed distribution of the requests. This tactic fails for load above 1000 queries/s and the average response time for queries increases exponentially.

Running queries for datasets of different sizes, shown in Figure 4b, the response times follow an expected pattern. The tests were run with an average load of 14 queries per second for the system, i.e. 14 clients issued on average one query per second. The choice was made to ensure that range queries would be included in our results. Range queries are the most expensive, and therefore response times are larger, as are searches for a specific keyword in any indexed attribute. Response times for point queries (exact matches of keyword and attribute) remain relatively constant, irrespective of dataset size, while response time for range and any attribute queries increases slowly as the number of records containing them increases with the datasize.

An increase in the number of attributes (Figure 4c) has no effect on the response time of point queries, and this highlights the efficiency of the system. Similarly, times for range queries and queries in any attribute scale almost linearly, keeping the overhead small even for large indices. This might seem unexpected, as for most range queries, it would mean a theoretical increase of factor 27. This is not the case due to the structure of HTML: since the expected nesting depth of HTML tags is low, the number of query results returned by such queries rarely reaches this theoretical maximum. However, since the selection of indexed attributes is left to the user, this behavior could theoretically be simulated by choosing consistently nested tags (e.g. table, tr and td in HTML). Given the limited usefulness of such a selection though, we believe that it is not a concern for practical system use.

4. RELATED WORK

The distributed cooperation of a large number of computational and storage resources is a challenging task. Application specific requirements of large scale data management tasks (e.g. the need to push computation near the data) prohibit the use of typical general purpose job schedulers. To cope with these requirements, "data-aware" distributed data management frameworks have been proposed, with Google's MapRreduce [7] as the most prevalent. MapReduce is inspired by the typical "map" and "reduce" functions found in Lisp and other functional programming languages: a problem is separated in two different phases, the Map and Reduce phase. In the Map phase, non overlapping chunks of the input data get assigned to separate processes, called mappers, which process their input and emit a set of intermediate results. In the Reduce phase, these results are fed to a (usually smaller) number of separate processes called Reducers, that "summarize" their input in a smaller number of results that are the solution to the original problem. For more complex situations, a workflow of map and reduce steps is followed, where mappers feed reducers and vice versa.

On top of MapReduce, a number of frameworks have been proposed to facilitate the management and execution of data warehousing tasks. Yahoo's Pig [13], Facebook's Hive [14] and HadoopDB [4] are a representative subset of such frameworks. In these frameworks, users write their analytical jobs in a declarative scripting language, they are translated in a chain of MapReduce steps and they are executed on Hadoop. Microsoft's SCOPE is a similar approach but it is deployed on top of Cosmos, a proprietary distributed storage and execution engine.

A number of content analytic platforms built on top of Hadoop have been proposed recently. In [5], IBM presents a preliminary version of their work on analyzing large datasets using Hadoop, where they do not deal with serving the created content. Distributed index creation frameworks that exploit the parallelism of Hadoop and HBase have been recently announced. Ivory, [12] an indexing framework implementation on top of Hadoop, distributes only the index creation through a MapReduce job, whereas the index is served through a centralized repository. On the other hand, HIndex [11] serves indices through HBase, but the index creation is centralized. In our system, both the index creation and serving is done in a distributed way.

5. DISCUSSION

The indexing system described above is an attempt to leverage the abilities of Hadoop and HBase over similar distributed approaches. In essence, we complement the MapReduce framework's ability to distribute processing over a large number of nodes with HBase's flexible and high performance architecture. Similar solutions (see Pig[13] or Hive[14]) deal with the same problems by running MapReduce jobs for each query submitted. While this allows for complex queries, it also requires significant processing time on the servers for each query and knowledge of the way the data are physically stored. In comparison, our solution aims for speed in simple queries, while most of the processing required on complex queries is performed by the client. This keeps network traffic and CPU load on the data servers at a minimum, allowing for more concurrent client connections. HadoopDB[4] extends Hive with support for SQL queries but suffers from the same issues, although it does allow for data partitioning, which would be comparable with our indexing process. However, this requires explicit knowledge of the HadoopDB architecture, while alternative partitioning rules (i.e., views in database terminology) cannot be enforced.

In contrast to other approaches (such as Ivory[12] and HIndex[11]) that distribute only one part of the process, our implementation has the benefit of a fully distributed architecture. This ensures full utilization of the available physical infrastructure and increased scalability. Moreover, our approach significantly reduces the time needed for both index creation and query responses.

Considering that data are already stored in HDFS, storing them again in HBase seems redundant: one could use the offsets of an HDFS file as record identifiers. Yet, such an approach makes insertion of records difficult, as all offsets need to be re-calculated. HBase can perform single imports and requires few updates in the index table per new record. Instead of index updates, if consistency between the context table and its index is relaxed, periodic reruns of the Indexer can be used. An evaluation of the trade-off between these two approaches is left for future work.

At the moment, the content and index creation is a serial procedure that could be otherwise pipelined: the output of an Uploader reducer could be directly fed to the Indexer mapper and written down to HDFS as HFiles at the same time using a chain of MapReduce steps. Nevertheless, this operation is new and, in our opinion, relatively unstable in the current Hadoop version.

The need for complex row keys in the index table (e.g., *google_revision*) was imposed from the diversity of our datasets. Different data types (structured, unstructured, semistructured) constrain the granularity of the index in different ways. With such a complex row key, a single index table that accommodates all these constraints can be used, while making client lookups for specific attributes from different data types fast and straightforward. This structure can also overcome HBase limitations on the dimensionality of the data stored in its tables.

In the future, our system will be extended to support more complex queries, such as SQL-like joins, and complex table views for the content and index tables. This would make our system more functional while preserving its main advantages, speed and ease of use. Another useful improvement to our implementation would be support for secondary indices to speedup custom searching on different dataset dimensions. This would be implemented using the same basic design as the index table and would immediately increase the usability and lower response times for complex, multidimensional queries. We are also considering the deployment of our system to an actual cloud vendor such as Amazon.

6. REFERENCES

[1] Nosql databases. http://nosql-databases.org.

[2] Project Voldemort. http://project-voldemort.com.

[3] D. J. Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Eng. Bull*, 32(1):3–12, 2009.

[4] A. Abouzeid et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[5] K. S. Beyer et al. Towards a Scalable Enterprise Content Analytics Platform. *IEEE Data Eng. Bull.*, 32(1):28–35, 2009.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, page 220, 2007.

[9] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):43, 2003.

[10] A. Lakshman and P. Malik. Cassandra-A Decentralized Structured Storage system. In *LADIS*, 2009.

[11] N. Li, J. Rao, E. Shekita, and S. Tata. Leveraging a scalable row store to build a distributed text index. In *CloudDB*, pages 29–36, 2009.

[12] J. Lin, D. Metzler, T. Elsayed, and L. Wang. Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search. In *TREC*, 2010.

[13] C. Olston et al. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110, 2008.

[14] A. Thusoo et al. Hive - a Warehousing Solution over a Map-Reduce Framework. *PVLDB*, 2:1626–1629, 2009.