Efficient Updates for Web-scale Indexes over the Cloud

Panagiotis Antonopoulos [#1,‡], Ioannis Konstantinou [*2], Dimitrios Tsoumakos [†3], Nectarios Koziris [*4]

[#] *Microsoft Corp, Redmond, WA, USA*
[1] panant@microsoft.com

[*] *Computing Systems Laboratory, School of Electrical and Computer Engineering*
National Technical University of Athens, Greece
[2] ikons@cslab.ntua.gr
[4] nkoziris@cslab.ntua.gr

[†] *Department of Informatics, Ionian University, Greece*
[3] dtsouma@ionio.gr

***Abstract*— In this paper, we present a distributed system which enables fast and frequent updates on web-scale Inverted Indexes. The proposed update technique allows incremental processing of new or modified data and minimizes the changes required to the index, significantly reducing the update time which is now independent of the existing index size. By utilizing Hadoop MapReduce, for parallelizing the update operations, and HBase, for distributing the Inverted Index, we create a high-performance, fully distributed index creation and update system. To the best of our knowledge, this is the first open source system that creates, updates and serves large-scale indexes in a distributed fashion. Experiments with over 23 million Wikipedia documents demonstrate the speed and robustness of our implementation: It scales linearly with the size of the updates and the degree of change in the documents and demonstrates a constant update time regardless of the size of the underlying index. Moreover, our approach significantly increases its performance as more computational resources are acquired: It incorporates a 15.4GB update batch to a 64.2GB indexed dataset in about 21 minutes using just 12 commodity nodes, 3.3 times faster compared to using two nodes.**

I. Introduction

Our era is characterized by what is referred to as "the data explosion": The amount of digital information worldwide will exceed 1.8 zettabytes by the end of 2011, expected to increase 44-fold within this decade [1]. Semi or fully unstructured data (created through social networking applications, blogs, etc.) is growing at a rate of 80% each year with over 2 billion internet users sharing, creating and updating their content. This has proved to be an overwhelming trend for traditional data management systems [2]. As data explodes in size, businesses face the challenge of storing, managing and analyzing this bulk of information efficiently. Industry leaders have dealt with this problem utilizing clusters of commodity hardware combined with scalable software.

The MapReduce framework [3] has been widely used for resource-demanding, batch-processing operations. Its ability to scale to a large number of commodity hardware nodes allows its extensive use for speeding up highly parallel operations at a reasonable cost. Similarly, a new generation of data stores, *NoSQL databases* [4], has been developed to enable the storage and serving of large semi-structured datasets. The term NoSQL is used to describe non-relational, horizontally scalable data stores. In favour of scalability and high availability, these systems relax classic ACID guarantees. 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.

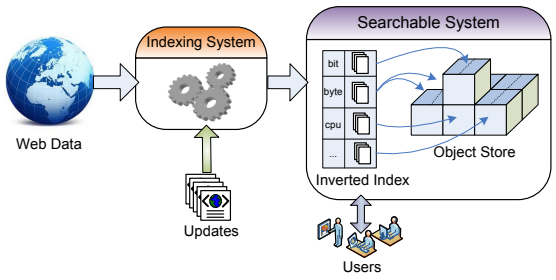

Fig. 1. A general model for indexing, updating and searching web-scale information.

In order to take advantage of the available information, huge datasets (many of which are freely available, either from public APIs or published online [5]) have to be effectively indexed (see Fig. 1). This is an extremely demanding task, considering the volume and the diversity of the stored data. At the same time, especially with the explosion of user-generated content, the information available on the Web is constantly increasing and changing. In Facebook, an average user creates 90 pieces of content each month while more than 30 billion pieces of content are shared each day [6]. In Twitter, the average number of Tweets people send per day is over 140 millions with a peak of about 7K Tweets per second [7]. Finally, each of Wikipedia's about 20 million articles is edited over 30 times a month, with over 8K new articles added to the collection per day [8].

This means that the content of the existing indexes becomes

[‡] Work done while the author was with the CSLAB, School of ECE, NTUA

frequently outdated and, as a result, users do not have access to the new information. Together with the size of the input data, this stresses the need for (possibly frequent) update operations on the created indexes. Thus, the indexing scheme should provide innovative techniques that will allow processing the new and modified data and updating the existing index structures without rebuilding them from scratch. As the volume of the modified data is usually orders of magnitude less than the already indexed dataset, updating should ensure that users have fast access to the new information on the Web. To the best of our knowledge, works in [9], [10] are the only open-source systems that offer distributed indexing and serving of web-scale datasets. Yet, none of these approaches allows for updates, requiring a rebuild of the Inverted Index in order to provide the updated content to the users.

In this paper, we present HMR-Index-Updater[1], an open source, distributed processing architecture which allows updating existing Inverted Indexes when a part of the dataset is modified or new data is added. In order to speed up the compute and storage-intensive update process, we leverage the capabilities of MapReduce in combination with the horizontal scalability and loose-schema features of a distributed, NoSQL database, *HBase* [11]. By limiting the processing of the existing dataset and minimizing the changes needed to the index, the proposed technique significantly reduces the time required for the update, which is now almost independent of the size of the existing index and dataset. The updated index is, finally, stored in HBase to support a large number of concurrent users and achieve low response times under heavy query load, using a commodity hardware cluster.

Experiments using different Wikipedia snapshots (23.7 million pages with over 2 million updated documents), demonstrate the speed and robustness of our implementation: Our system scales linearly with the size of the updates and the degree of change in the documents; it significantly increases its performance as more computational resources are acquired (decreases the time required to incorporate a 15.4GB update batch to an indexed set from over 100 minutes to about 20 minutes when increasing the cluster size by a factor of 6); and, finally, demonstrates almost constant update performance regardless of the size of the underlying index, allowing very frequent update operations on large indexes.

II. Update Technique

An Inverted Index stores a mapping from every term included in the indexed dataset to a list of references to the documents that contain the corresponding term. This information is usually stored using tuples of the form $(term, list(doc\_ref))$, where "doc_ref" represents a reference, such as the document ID, to a document that contains this *term*. By the term *"index record"* we refer to a tuple of the form $(term, doc\_ref)$, which indicates that this specific *term* is included in the document referenced by doc_ref.

[1] `http://hmr-index-updater.googlecode.com`

The update technique presented in this paper applies the necessary changes *only* to the index records which refer to the updated documents. In this way, we aim to minimize the processing over the new and modified documents, performing the necessary tasks only on a small relevant subset of the whole dataset.

The technique focuses on achieving the following goals:

- The time required for the update has to be independent of the size of the existing dataset and index, in order to enable small and frequent updates on large Inverted Indexes.
- The index must remain consistent and its structure unaffected, in order to ensure the stability and the efficient operation of the system, even after a big number of updates.
- The update process must be able to be executed in a scalable, parallelizable way to reduce the required time by exploiting commodity cluster resources. In this way, the index can be frequently updated even when the volume of the updates is high.

A. Standard Update Procedure

The input of the proposed update technique is considered to be a collection of new or modified documents. The new documents simply need to be indexed and the created index records have to be added to the existing Inverted Index. On the other hand, the update process is more complicated for updating existing documents: in this case, the index records which refer to the old version of each document have to be deleted, before the records of the new version are added to the index. This requirement has two major implications. Firstly, the older version of the document has to be retrieved and processed in order to detect the records that need to be deleted from the index. Secondly, these records need to be located in the existing Inverted Index in order to be removed. Locating and deleting the old index records is a demanding task and the time required for this process is inevitably affected by the schema used for the Inverted Index.

B. Inverted Index Schema in HBase

In order to speed up the update process, it is essential to store the index using a structure that will allow fast record discovery and deletion. However, this choice must not sacrifice the index's performance to user queries. To achieve that, we leverage the unique characteristics of HBase. HBase has the ability to scale horizontally and store millions of columns for billions of table rows. In contrast with the traditional, relational database systems, the schema of a table does not need to be defined at the creation of the table, but can dynamically change while the data is inserted or deleted. Each row can have a different number of columns and their names may also vary among rows. HBase indexes the columns of each row and as a result ensures fast access to every table cell.

Based on these characteristics, we select a schema where each row represents a term included in the Inverted Index and

each column of a row represents an element of the corresponding list of document references: Each cell $(row, column)$ of the table represents a record $(term, doc\_ref)$ of the Inverted Index. To obtain instant access to every record, we choose to use the document ID as the column name. This means that we can locate and delete specific index records simply by deleting the corresponding cells, $(term, documentID)$, without having to scan the lists of references to identify the record that needs to be removed. Using the indexes maintained for the columns of each row, HBase allows fast deletion of specific cells and, therefore, the time needed for the deletions is only slightly affected by the size of the existing index.

C. Forward Index to Speed Up the Update

In order to identify obsolete records that have to be deleted, the older versions of the modified documents must be retrieved and processed. This is a demanding task in terms of computation and I/O, and can significantly increase the time required for the update. However, these documents have already been processed when the index was created or updated and, as a result, reprocessing can be avoided by storing the document terms that were included in the Inverted Index. To achieve that, when a document is indexed, its terms are also stored in a different data structure. This structure contains the terms of each document using tuples of the form $(doc\_ref, list(term))$ and is widely known as the *Forward Index*. The Forward Index is easily built while the documents are indexed and allows for instant access to the records which need to be deleted during the updates. At the same time, retrieving the Forward Index of a document is more efficient than retrieving the document itself, since the size of the indexed terms tends to be much smaller than the size of the document, due to the elimination of the duplicate words. The Forward Index technique achieves faster updates through an increase in the required storage space, certainly affordable given the decreasing storage costs.

D. Minimizing Index Changes during the Updates

In the update process described so far, all the existing index records of a modified document have to be removed before the records of the newer version can be added to the index. This way, a large number of cells has to be deleted and re-inserted, even for a minor modification in the document's content. Considering that modifications in documents are limited [12], especially when the index is frequently updated, we present an update strategy which minimizes the number of changes required to the Inverted Index. More specifically, we compare the Forward Index entries of the newer and older version of each document in order to identify the terms which are not common. A record of the Inverted Index has to be deleted for each term which was included in the old version, but has been removed from the new one, whereas a new record must be added to the index for each term which is included in the new version, but not in the old one. The common terms do not require index changes. Although the comparison of the Forward Indexes increases the complexity of the process, the reduction in the number of the required changes significantly speeds up the index update, avoiding a large number of costly HBase deletions and insertions.

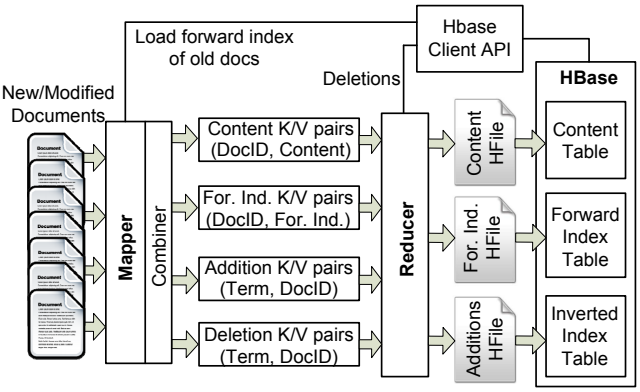

Fig. 2. The distributed update process

III. Distributed Index Update Processing

Although the proposed technique significantly speeds up the index update, a centralized system would need a lot of time to process the large volume of new and modified documents. For this reason, it is essential to execute the update in a distributed environment, in order to take advantage of the resources of a large cluster. During the update, each document can be processed independently and, then, the created index records have to be merged in order to be added to the Inverted Index. It is obvious that this application is perfectly suited to the MapReduce logic. Utilizing the MapReduce framework, we can easily distribute the processing across the nodes of the cluster to reduce the time needed for the update. Moreover, we can adaptively add or remove Mapper and Reducer processes according to the dataset needs and the available cluster resources.

Our system is built on top of *Hadoop* [13], an open-source implementation of Google's MapReduce. Hadoop consists of a distributed file system called *HDFS* and a MapReduce executing framework. Fig. 2 presents the workflow of our distributed indexing system. The new and modified documents are processed by a MapReduce job which updates the Inverted Index and loads the Forward Index and raw content of these documents to the corresponding HBase tables. When the update is completed, new content is accessed through the updated Inverted Index Table and Content Table. Considering that the documents are already stored in HDFS, storing them again in HBase seems redundant. However, in this way we can leverage the database characteristics of HBase(caching, etc.) to achieve lower response time to user queries.

A. The Map Phase

The new and modified documents dataset is split into chunks which are processed in parallel by independent map tasks. Each map task processes the documents of the corresponding chunk according to Algorithm 1, emitting four different types of key/value pairs. These are responsible for updating the Inverted Index and loading the raw content and Forward Index of the processed documents in HBase.

Algorithm 1 Processing the documents - Map phase

```
for all Document d ∈ myChunk do
  Emit a content key/value pair: (d.ID, d.content)
  Scan d to identify the terms that will be indexed
  for all Term t ∈ d do
    Insert t in Forward_Index new_fi
  end for
  Emit a Forward Index key/value pair: (d.ID, new_fi)
  Load from HBase the Forward Index old_fi of d
  if (old_fi exists) then
    Compare new_fi to old_fi
    for all Term t : t ∈ new_fi ∧ t ∉ old_fi do
      Emit an addition key/value pair: (t, d.ID)
    end for
    for all Term t : t ∈ old_fi ∧ t ∉ new_fi do
      Emit a deletion key/value pair: (t, d.ID)
    end for
  else
    for all Term t : t ∈ new_fi do
      Emit an addition key/value pair: (t, d.ID)
    end for
  end if
end for
```

Algorithm 2 Processing the intermediate key/value pairs - Reduce phase

```
for all Key key do
  if (key refers to an addition) then
    for all Value v ∈ values do
      Insert v in DataStructure ds, in order to sort them
    end for
    Sort the elements of ds in ascending order
    /*The values are used as column names and they have
    to be sorted in order to be bulk loaded to HBase*/
    for all Value v ∈ ds do
      Emit a key/value pair (key, v) in the Additions output
    end for
  else if (key refers to a deletion) then
    for all Value v ∈ values do
      Emit a key/value pair (key, v) in the Deletions output
    end for
  else if (key refers to the Forward Index) then
    Emit a key/value pair (key, v) in the Forward Index
    output
  else if (key refers to the documents' content) then
    Emit a key/value pair (key, v) in the Content output
  end if
end for
```

B. The Combiner

In order to minimize the intermediate key/value pairs that need to be transferred to the reducers, we use a simple *combine* function which merges the intermediate key/value pairs of each map task. More specifically, the combine function merges the values of each key, creating a list of values, which is finally emitted in *only* one key/value pair of the form $(key, list(values))$.

C. The Reduce Phase

The intermediate key/value pairs emitted by the map tasks are grouped and sorted according to their key and then transferred to the reduce tasks. These key/value pairs are processed by the reduce tasks according to Algorithm 2, in order to produce the output of the MapReduce job. As seen in Fig. 2, the reducers have four independent output streams. One of these streams deals with the deletion of index records and uses the HBase Client API in order to access the HBase and delete the corresponding cells. The other three streams, which concern insertions to the Inverted Index, Forward Index and Content Table, write the output key/value pairs directly to HDFS using the HFile format. The HFiles are bulk loaded to the corresponding HBase tables, which finally contain all the updates.

The reason for bypassing the HBase API is that it is particularly slow for bulk insertions: in this case, all the insertions are first added to a write-ahead-log and then in a in-memory buffer (MemStore). MemStore is periodically flushed to the disk in HFiles. By directly creating the HFiles and bulk-loading them in HBase we avoid the expensive intermediate interactions with the write-ahead-log and the MemStore needed for each insertion.

D. Evenly Distributing the Reducers Load

In order to achieve maximum parallelism, the completion time for all the tasks of each phase has to be approximately the same. According to a recent study [14], the uneven distribution of the input data can significantly increase the completion time of a job, since a small number of map or reduce tasks may need significantly more time to complete its execution compared to the majority of the tasks. This usually happens because of the skewed distribution of the input or intermediate key/value pairs: few keys are extremely common, while the vast majority appear very rarely.

This problem is tackled in the Map phase, as the framework divides the input dataset in equally sized chunks, assigning each chunk to an independent map task. However, in order to ensure that the intermediate key/value pairs are evenly distributed to the reduce tasks, we have to design a custom *partitioning function*. The intermediate key/value pairs of our job have either a document ID or a term as their key. Since these types of keys are completely different, we prefer to manage them independently.

There always exist two key/value pairs with the same key-DocumentID: one for the content and one for the Forward Index. Therefore, we simply need to ensure that each reducer receives approximately the same number of keys. This can be easily achieved using a hash function.

On the other hand, the word occurrences in natural languages follow a Zipfian distribution. As a result, the number of

key/value pairs per key-term varies significantly. For instance, the term "the" appears in almost every document, whereas the term "democracy" is a much more rare case. Therefore, even if we ensure that all reducers receive the same number of terms, the number of key/value pairs that each reducer will have to process will vary, affecting the task execution time.

In order to overcome this problem, we have designed a sampling MapReduce job, similar to those used in [15], [16], which is executed before the update and partitions the range of terms in R approximately equally sized partitions, where R represents the number of reducers in our main job. The map tasks process a representative sample of the dataset, according to Algorithm 1, but emit simply a key/value pair $(term, 1)$ for every term that has to be added or deleted. The job has only two reducers. The first reducer receives *all* the key/value pairs which refer to additions to the Inverted Index, whereas the second one receives *all* the key/value pairs which refer to deletions. Each reducer counts the number of occurrences of each term and outputs a file containing the splitting points that divide the sample terms in R equally sized partitions.

The partitioning function of our main job loads the splitting points, for both the addition and the deletion key/value pairs, and chooses the reducer for each key using the function:

$$f(key) = \begin{cases} i & \text{if } key < SP[i], i \in [0, R-2] \\ R-1 & \text{if } key > SP[R-2] \end{cases}$$

where SP represents an array containing R-1 splitting points for the additions or the deletions depending on the key type.

IV. Experimental Evaluation

A. Experimental Setup

To evaluate the performance of our indexing system, we conduct experiments using publicly available datasets from Wikipedia. Wikipedia creates a snapshot of all its pages every month. As pages are constantly updated and new ones are created, using snapshots from different dates allows us to test the performance of our update system. In our experiments, we use the snapshots created on April 5, 2011 and May 26, 2011 and the changes between them as the update dataset. The size of the first dataset (default initial dataset) is 64.2 GB, containing 23.7 million documents[2], while the new/modified documents dataset (update-dataset) corresponds to 15.4 GB and 2.2 million documents respectively.

By default, our cluster consists of 8 worker nodes (increased to a maximum of 12 nodes) plus a single machine in the role of the HDFS, MapReduce and HBase Master. Each worker node has 2×Quad-Core Intel Xeon E5405 CPUs at 2.00GHz with 12MB L2 cache and 8 GB of RAM (6 GB RAM for the Master). All nodes are operating on 64bit Debian Linux and are interconnected with Gigabit Ethernet.

We utilized Cloudera's distribution for Hadoop and HBase, Hadoop v.0.20.2-CDH3 and HBase v.0.90.3-CDH3, to take advantage of their extra features. To achieve maximum resource utilization, we allow each worker node to spawn 6 map and 6 reduce tasks (creating a total of 48 map and 48 reduce tasks for the default runs) with a heap size of 1024 MB for the reduce tasks. We disabled Hadoop's speculative execution, which reduces the cluster's effective capacity, executing every task more than once for redundancy. Finally, we set the number of reducers in our jobs to 80, to increase load balancing during the Reduce phase. The rest of the settings for both Hadoop and HBase were left unaffected.

[2]We refer to the term document in contrast to a Wikipedia article. An article may link to multiple documents (e.g., due to redirects) which contain the actual content. Thus, we define documents to be the unit of processing in our evaluation.

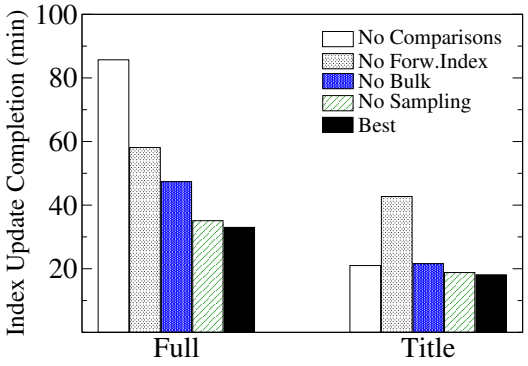

Fig. 3. The performance of the system for full-text and title-only indexing with different optimizations disabled. Best is our proposed method that includes all optimizations.

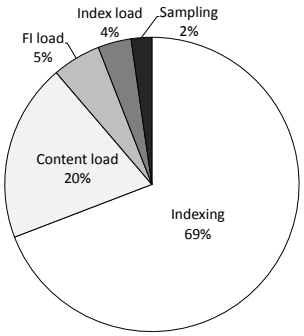

Fig. 4. Contribution of each processing step for the full-text indexing.

Before performing the update experiments, we have to create the Inverted Index from the initial dataset. Our system is capable of building a new Inverted Index without any change: If the Forward Index Table is empty, then every document is regarded as a new one and *all* its terms are added to the empty Inverted Index. Our system needs approximately three hours to process this dataset and load the documents' content and the Inverted and Forward Indexes to HBase.

B. Evaluating our Design Choices

As described in the previous sections, in order to speed up the update process, we decided to employ specific optimizations:

- Use the Forward Index of the existing documents,
- Bulk load the output in HBase,
- Run a sampling job to evenly distribute the load to the reducers,

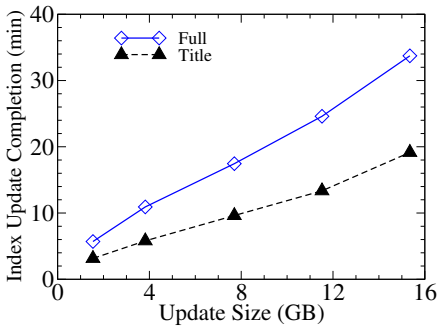

Fig. 5. Update time for different volumes of new and modified documents

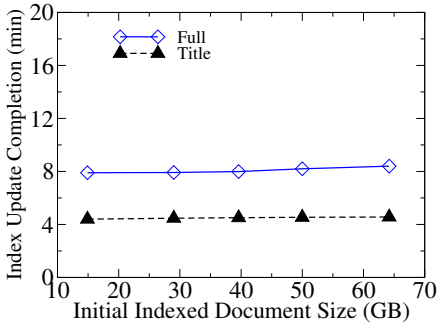

Fig. 6. Update time for different sizes of the existing index

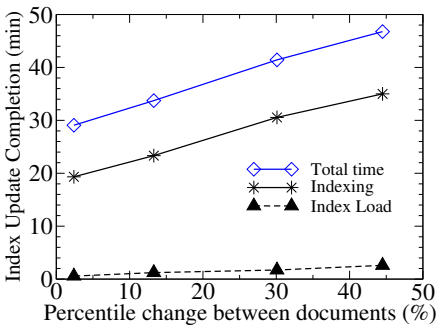

Fig. 7. Update time for various percentages of the documents' modification

- Compare the two different versions of each document to minimize the changes required to the index.

First, we measure the impact in performance and the respective gains that each of these features provides. Specifically, we measure the update completion time of our system with and without each of these optimizations for two modes of document indexing: Index the whole document (*Full*) and index only the title (*Title*). Results are presented in Fig. 3.

Comparing the different versions of each document is the most important optimization for full-text indexing, boosting performance by a factor of 2.6. Deleting and re-inserting all the terms of the updated documents creates a large volume of intermediate key/value pairs, which has to be transferred across the network and processed by the reducers, but, also, a large number of HBase deletions and insertions. Comparing allows us to minimize these volumes, speeding up the update process. However, the effect of this optimization is not as prominent in the title-only indexing (1.2X faster). In this case, the number of indexed terms is orders of magnitude smaller and, as a result, the time required for processing the documents and loading them in HBase dominates the total time for the update.

Using the *Forward Index* of the older version of the documents improves the performance of our system in both cases (1.8 and 2.4 times faster respectively). This optimization enables us to avoid retrieving the whole document from HBase and rescanning it, which results in decreasing the I/O and computational cost. On the contrary, the significance of *Bulk Loading* is affected by the volume of the indexed terms. As a result, it considerably reduces the update time for full-text indexing (over 30%), but not for title-only indexing (about 16%). Finally, *sampling* the input dataset to balance the load between the reducers does not significantly improve the performance of our system (4-6%). Because of the comparison, the amount of the intermediate key/value pairs is relatively small and, therefore, their uneven distribution does not have an important impact on the update time.

Fig. 4 depicts the portion of time spent during the different steps of our update process (full-text indexing). The dominating task, taking up about 70% of the time, is the indexing process itself. Thus, the most advantageous optimizations in our system (i.e., comparisons and Forward Index method) point to this step. The loading of the content itself takes up 20% of the time, with the Bulk Load optimization contributing to minimize this portion. Sampling the input and loading the updated Forward and Inverted Index records take up about 10% of the time, with sampling being the least time-consuming stage.

C. Evaluating the Performance of our System

One of our main goals is to provide efficient updates over web-scale datasets, with minimum deterioration over the size of the processed data. Fig. 5 presents the performance of our system for different update sizes over an existing Inverted Index created from our default initial dataset. In both full-text and title-only indexing, the time required for the update is linear to the size of the input, which is consistent with the theoretical analysis. Our results show that increasing the size of the applied update by a factor of 10 increases the update time by a factor roughly equal to 6. This linear growth pattern allows efficient processing of large datasets.

An equally important characteristic is the system's ability to complete the update in time almost independent of the existing index size. Fig. 6 shows the performance of our method for different sizes of the existing Inverted Index. The update dataset is the same in all experiments, containing approximately 400K documents (≈5 GB). According to Fig. 6, the update time is almost constant for all index sizes, as for over 4 times larger existing index, the update time increases by less than 6%. The results of these experiments prove that by utilizing the horizontal scalability of HBase and the schema described in Section II-B, we are able to frequently update large Inverted Indexes.

The results in Fig. 3 show that comparing the different versions of the modified documents significantly speeds up the update process. This optimization, as described in Section II-D, is based on the fact that, although a large number of documents gets updated, the changes in the content of these documents are minimal. For this reason, we would like to examine the behaviour of our system with a variable amount of changes in the modified documents. Fig. 7 presents the update completion time in relation to the percentage of the documents' content that has been modified. An increase in the content changes results in a proportional increase in the number of intermediate key/value pairs as well as in the

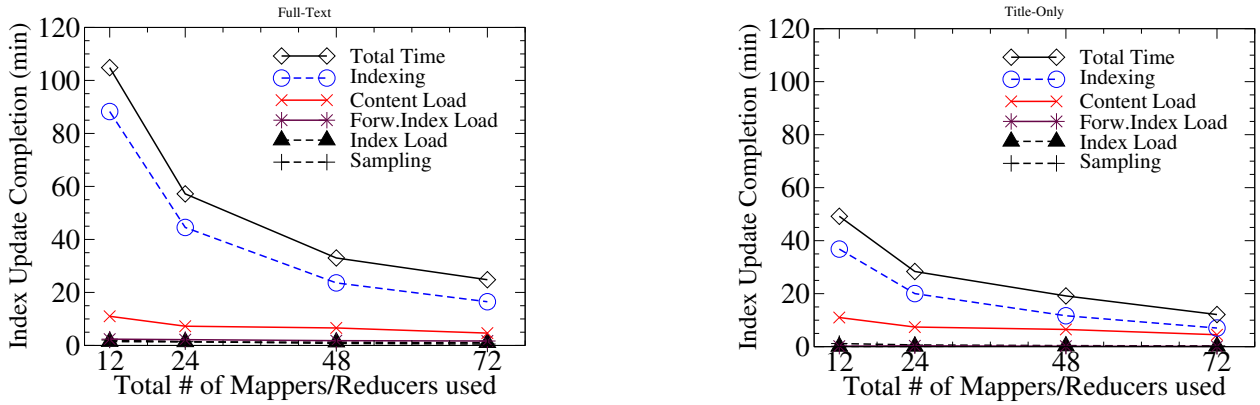

Fig. 8. Index Update times for the full-text and title-only update tasks under variable number of available Mappers and Reducers.

number of deletions and insertions in HBase. The results presented in Fig. 7 are consistent with the theory, since the time required for the update is linear to the volume of the intermediate key/value pairs and the HBase operations. This diagram shows that for an 18-fold increase in the amount of modifications per document, the update time increases by a factor of 1.6 (significantly lower than the time required without this optimization as shown in Fig. 3). This trend is followed by the indexing curve which increases linearly with the amount of changes. On the other hand, loading the index updates to HBase increases at a much slower pace, exhibiting times between 0.5 and 2.5 minutes.

D. Varying the number of cluster resources

A major requirement from modern systems is the ability to take advantage of the large number of hardware resources available in a cluster and increase their performance when more processing power, storage, network throughput, etc. are provided. The tools our system utilizes have inherent scalability features: Hadoop enables any task to be undertaken by a variable number of Map and Reduce tasks which work in parallel. Increasing the number of Mappers and Reducers results in a decrease in the size of the data each task is required to process independently and, therefore, in a decrease in the completion time of the MapReduce job. Similarly, HBase enables horizontal partitioning of a large dataset among the cluster's nodes, thus distributing both upload and query load.

To demonstrate the scalability of the update process in relation to the available hardware resources, we measure the update time over the default initial and update datasets using a variable number of worker nodes, ranging from 2 to 12. Since each node spawns 6 mappers and 6 reducers, the respective number of processes ranges from 12 to 72 (mappers and reducers separately). Fig. 8 presents the results. Our system is able to greatly increase its performance, when more resources are added: For both full-text and title-only indexing, the total update times are reduced by 70%. This is an important cloud application requirement, since extra nodes are acquired by a cloud vendor in an easy and inexpensive manner.

Results presented in Fig. 8 allow us to draw some additional conclusions: The most time-consuming task is the indexing part, which demonstrates excellent scalability (5 times faster from 12 to 72 tasks) to the number of nodes. Because of the highly parallel nature of the update-indexing process, the amount of gain is not significantly reduced even when the number of available Mappers/Reducers gets higher. Content load also gets faster, but to a smaller extent (2.3X). The increase in the number of HBase nodes does not seem to have a proportional increase in the performance of bulk loading operations. Finally, sampling the input and loading the index updates (for the Inverted and Forward Indexes) take up a small amount of time and, therefore, their performance does not significantly contribute.

V. Related Work

Researchers have long realized the limitations of centralized systems to process the huge volume of data available on the Web. As a result, research has often focused on designing distributed architectures that speed up the processing of Web datasets, exploiting the resources of multiple machines [17]. In 2004, Google presented MapReduce [3], a distributed platform that simplifies the implementation of distributed applications on large commodity hardware clusters. The MapReduce framework allowed a significant speed up in the creation of Web content indexes.

A large number of content analysis systems has been built on top of Hadoop, some of which specifically deal with the distributed creation or serving of Inverted Indexes. Ivory [18], for example, distributes the index creation through a MapReduce job, while HIndex [19] and Sphinx [20] support distributed search on their indexes. In [21], various distributed indexing approaches are compared. Moreover, this work examines whether the MapReduce model is suitable for Inverted Index creation. To the best of our knowledge, the only open source systems which perform both index creation and serving in a distributed way are [9] and Katta [10]. Katta combines Apache Lucene [22] with Hadoop in order to distribute the index creation and partitions the created index into "shards" which are served through independent Katta Nodes, controlled by a Katta Master. The system described in [9] distributes the index creation using Hadoop and stores the Inverted Index through HBase. Although all these systems significantly speed

up the creation of the Inverted Index, the index has to be periodically rebuilt in order to include the updates made to the indexed dataset. Considering the size and the rate of change of Web documents, search engines cannot afford rebuilding their indexes from scratch.

Google Caffeine [23] is a new indexing scheme which constantly updates the existing index with the new content found on the Web and provides the users with fresher results. Yet, Google Caffeine is a proprietary system used only by the Google search engine. Apache Solr [24], on the other hand, is an open source enterprise search platform which supports distributed search and allows updates. However, index creation and updating are centralized and, therefore, cannot scale. To overcome this shortcoming, Lucid Imagination developed LucidWorks [25], a free but closed source system built on top of Solr. LucidWorks distributes documents among the nodes of a cluster, making each node responsible for indexing, updating and serving a subset of the dataset. Both Solr and LucidWorks do not utilize the comparison optimization, simply deleting the old version of each modified document before indexing the new one.

VI. Conclusions

The data available on the web is growing at an amazing rate. By utilizing modern distributed architectures we are able to combine the hardware resources of large clusters in order to cope with this huge volume of data. However, speeding up the processing is not always sufficient. A large number of applications demand incremental processing of the updates over the existing datasets whose volume is usually orders of magnitude larger than the newly added data. Our system combines incremental and distributed processing in order to enable fast and frequent updates on web-scale Inverted Indexes. Our update technique allows us to process only the new or modified documents and minimize the changes required to the index, reducing the update time. Moreover, applying this method through the MapReduce framework and using HBase as the storage layer, we manage to parallelize the update process incurring significant gains as the number of concurrent tasks increases.

Through our experimental evaluation, our system exhibits update times linear to the size of the applied updates and almost independent of the existing index size. Consequently, large Inverted Indexes can be frequently updated to provide the users with fresher results. At the same time, utilizing Hadoop and HBase, our system has the ability to scale to a large number of commodity machines, achieving great performance at a reasonable cost. Using 8 worker nodes, our system is capable of updating a full-text index of all Wikipedia documents, applying over 15 GB of updates, in only 34 minutes, when building the index from scratch requires more than 3 hours. Through an increase in the number of participating nodes, performance improved by a factor of up to 3.3.

References

[1] J. Gantz and D. Reinsel, "Extracting value from chaos," *IDC research report, Framingham, MA, June*, vol. 19, 2011.

[2] D. J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," *IEEE Data Eng. Bull*, vol. 32, no. 1, pp. 3–12, 2009.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] Nosql databases. [Online]. Available: http://nosql-databases.org

[5] Public data sets on aws. [Online]. Available: http://aws.amazon.com/publicdatasets/

[6] Facebook demographics revisited – 2011 statistics. [Online]. Available: http://www.kenburbary.com/2011/03/facebook-demographics-revisited-2011-statistics-2/

[7] #numbers. [Online]. Available: http://blog.twitter.com/2011/03/numbers.html

[8] Wikipedia statistics. [Online]. Available: http://stats.wikimedia.org/EN/TablesWikipediaZZ.htm

[9] I. Konstantinou, E. Angelou, D. Tsoumakos, and N. Koziris, "Distributed Indexing of Web Scale Datasets for the Cloud," in *MDAC '10*, 2010.

[10] Katta - lucene and more in the cloud. [Online]. Available: http://katta.sourceforge.net/

[11] Hbase. [Online]. Available: http://hbase.apache.org/

[12] Z. Wang, "Incremental Web Search : Tracking Changes in the Web," Ph.D. dissertation, New York University, 2006.

[13] Hadoop. [Online]. Available: http://hadoop.apache.org/

[14] J. Lin, "The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce," in *LSDS-IR*, 2009, p. 5762.

[15] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on Processing Spatial Data with MapReduce," in *Scientific and Statistical Database Management*, Berlin, Heidelberg, 2009, vol. 5566, pp. 302–319.

[16] O. OMalley, "TeraByte Sort on Apache Hadoop," 2008.

[17] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina, "Building a Distributed Full-text Index for the Web," *ACM Trans. Inf. Syst.*, vol. 19, no. 3, p. 217241, July 2001.

[18] J. Lin, D. Metzler, T. Elsayed, and L. Wang, "Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search," in *Proc. of TREC*, 2009.

[19] N. Li, J. Rao, E. Shekita, and S. Tata, "Leveraging a Scalable Row Store to Build a Distributed Text Index," in *CloudDB*, 2009, pp. 29–36.

[20] Sphinx, open source search server. [Online]. Available: http://sphinxsearch.com/

[21] R. McCreadie, C. Macdonald, and I. Ounis, "Comparing Distributed Indexing: To MapReduce or Not?" in *LSDS-IR*, 2009.

[22] Apache lucene. [Online]. Available: http://lucene.apache.org/

[23] Google caffeine. [Online]. Available: http://googleblog.blogspot.com/2010/06/our-new-search-index-caffeine.html

[24] Apache solr. [Online]. Available: http://lucene.apache.org/solr/

[25] Lucidworks. [Online]. Available: http://www.lucidimagination.com/products/lucidworks-search-platform