Migrating a legacy web-based document-analysis application to Hadoop and HBase: An Experience Report

Double-blind Review

Abstract

Migrating a legacy application to a more modern computing platform is a recurring software-development activity. This chapter describes our experience with a contemporary rendition of this activity, migrating a web-based system to a service-oriented application on two different cloud software platforms, Hadoop and HBase. Using the case study as a running example, we review the information needed for a successful migration and examine the trade-off between development/re-design effort and performance/scalability improvements. The two levels of re-design, towards Hadoop and HBase, require notably different levels of effort, and as we found through exercising the migrated applications, they achieve different benefits. We found that both redesigns led to substantial benefit in performance improvement, and that expending the additional effort required by the more complex migration resulted in notable improvements in the ability to leverage the benefits of the platform.

1 Introduction

Migrating applications to cloud-computing environments is a software-engineering activity attracting increasing attention, as cloud environments become more accessible and better supported. Such migrations pose questions regarding the changes necessary to the code and to the architecture of the original software system, the effort necessary to perform these changes, and the possible performance improvements to be gained by the migration. The software-development team undertaking a migration-to-the-cloud project needs to address the following questions.

- What types of software (i.e., components and/or libraries) can developers expect when undertaking a migration project?
- What are the typical modifications required to the to-be-migrated application in order to better leverage the potential of the target cloud platform? What are the implications of the various platforms to the architectural and detailed design of the systems deployed on them?
- Will the particular software application benefit from its migration to a cloud environment? How may the trade-off between the costs of the planned modifications vs. the improvements anticipated of the application post-migration be assessed?

The term *cloud computing* characterizes the perspective of end users, who are offered a service (which could be in the form of a computing platform or infrastructure) while being agnostic about its underlying technology. The implementation details of the service are abstracted away, and it is consumed on a pay-per-use basis, as opposed to being acquired as an asset. In principle, one distinguishes among three different types of cloud-based services. When *infrastructure* is offered as a service (IaaS), end users are able to procure virtualized hardware. When a software *platform* is offered as a service (PaaS), end users consume a software platform, i.e., a combination of an

operating system, basic tools and libraries. Finally, when a *software* application is offered as a service (SaaS), end users consume as clients a specific application that is independently deployed and managed. Of course, these offerings can be combined into a stack of service offerings.

All three above scenarios promote improved scalability albeit through different mechanisms. The first scenario eliminates the need for users to acquire, manage and replace hardware since any number of appropriately configured virtual machines can be easily procured (and abandoned), for example, through Amazon Web Services[1]. The second scenario promises improved scalability with novel tools and computational metaphors, such as those of the Hadoop ecosystem for storing and manipulating "big data". Finally, when a software system is offered as a service, such as SalesForce[2], its consumers are offered state-of-the-art functionality, regularly maintained and extended, with guaranteed quality, at negotiable costs.

In this chapter, we report on our experience migrating a legacy application, TAPoR, to take advantage of IaaS (using AWS) and PaaS (in two scenarios, Hadoop and HBase). The original version of TAPoR had severe performance limitations; the promise of scalability through its migration "to the cloud" motivated our study. The original application ran on a single machine, in a single thread, within a single process. Taking advantage of the IaaS model, we modified it to incorporate a load-balancing component to distribute incoming requests to multiple identical processes, running on multiple virtual machines (Smit, Nisbet, Stroulia, Iszlai, & Edgar, 2009). This change however did not address the fundamental inability of the application to scale to large documents. To that end, we investigated the advantages of an architectural shift to exploit the advantages of (two variants of) the Hadoop ecosystem as a platform. To summarize, we have performed three types of modification to the original system:

- No architectural changes; deploy the software (with a load balancer) to multiple machines (on Amazon EC2, for instance);
- Rearchitecting towards the MapReduce paradigm; modify the architecture and implementation to make use of the distributed computation features of Hadoop; and
- Rearchitecting to use a NoSQL database; further change the implementation to also make use of the distributed database feature of Hadoop, HBase.[3].

Each of these scenarios required an increased level of development effort and a deeper conceptual understanding of the expected use cases of the software (*i.e.,*costs). Each offers varying performance, flexibility, and scalability (*i.e.,*benefits). To investigate the impact of these varied degrees of modification, we identified a set of standard text-analysis tasks and we compared the performance of the original version to the three modified versions on these tasks. We migrated four TAPoR operations to the Hadoop ecosystem, and comparatively evaluated the effort involved in making the transition to the performance benefit achieved. We found that expending additional effort did result in corresponding performance increases; we were not able to find the point at which further effort no longer resulted in appreciable gains. We believe that our results are, to some extent, generalizable to other similar software-migration projects, particularly those related to text analysis, business analytics, search, and similar. The degree to which a shift in functionality is practical and will achieve similar performance benefits will vary.

The rest of the paper is organized as follows. Section 2 describes MapReduce as a computational paradigm, its implementation in Hadoop, and HBase, a no-SQL database in the Hadoop family. Section 3 describes previous studies of application migration to Hadoop and HBase. Section 4 describes TAPoR, the legacy application which is the subject of our study. Section 5 discusses the development tasks involved in the three TAPoR migration studies. Section 6 describes the

experiments we set up to evaluate this case study migration and the lessons we learned from the experience. Finally, Section 7 presents the conclusions we can draw from our work to date and lays out some ideas for future work.

2 Platform as a Service: The Hadoop Ecosystem

In this section, we discuss the key concepts underlying the computational model supported by the two platforms to which we migrated TAPoR: *Hadoop*, the open-source implementation of the MapReduce paradigm, and *HBase*, a no-SQL database relying on *HDFS*, the Hadoop file system.

2.1 MapReduce and Hadoop

The now seminal MapReduce paper (Dean & Ghemawat, 2008) sparked an ongoing growth in data analytics using this paradigm. MapReduce is a scalable and straightforward approach to processing large amounts of data (up to petabyte scale) in parallel, on a cluster of commodity machines. It is not a silver bullet for all kinds of big-data problems, rather it is conceived to address specifically workloads that are data-parallel in nature. It proposes the design of applications in terms of a *map* and a *reduce* function: the map process consumes raw data as key-value pairs and generates intermediary key-value pairs; the reduce process consumes this output and somehow "aggregates" it to produce the final desired output. The processing of the map phase relies on the data-locality concept, with computation performed at the data nodes (nodes where data is residing). The MapReduce paradigm had already been part of functional languages in general (e.g., LISP); however, its usage in data-parallel big-data applications, where individual map tasks can be completed independently, has been a key factor in its popularity. It is being used in large production systems, where multiple terabytes of data are processed on a daily basis. The Apache Hadoop project offers an open-source implementation of MapReduce.

The main advantage of this model is the inherent support for execution in a distributed environment. Using Hadoop, developers do not need to take care of inter-process communication among the nodes. This helps in using this framework on a cluster of commodity machines, possibly virtualized by an IaaS offering, instead of requiring high-end shared-memory machines.

Given the general availability of IaaS offerings, one can easily leverage the potential of the MapReduce paradigm using a utility computing model from a provider such as Amazon, Rackspace, etc. Some providers, including Amazon, offer MapReduce through a Platform as a Service (PaaS) approach, assuming the overhead of setting up a Hadoop system. In this environment, software developers can rather easily program distributed software on a virtual cluster, by implementing the two functions. Platform libraries take care of the rest, including RPCs, distributing the workload among nodes, using data locality, handling node failures, etc.

One important consideration for MapReduce is that it provides a high throughput when run in a cluster environment. Though there has been some work of using it in multi-core shared-memory machines, its real advantage is using it for a distributing-computing application, on a cluster of machines. Its core features, namely handling node failures and re-running of failed tasks, work on the assumption of using a distributed file system. We discuss the essential features of such file systems in the next section.

2.2 Hadoop Distributed File System (HDFS)

MapReduce jobs impose a different set of requirements for reading/storing files as compared to traditional native file systems (POSIX-compliant). Below, we review the main requirements for

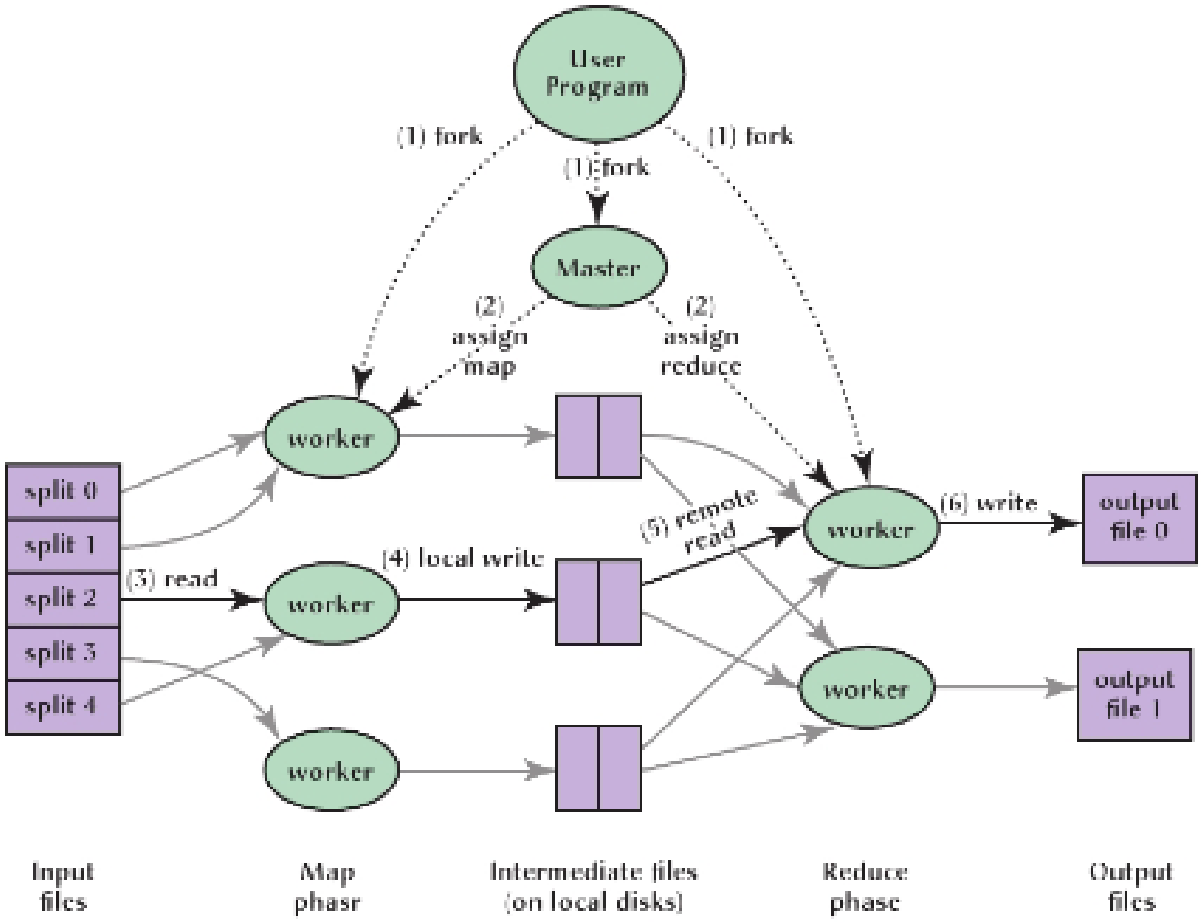

Figure 1: Workflow of a typical MapReduce application (Dean & Ghemawat, 2008)

such a file system.

- A distributed file system should run on inexpensive commodity machines, and it should continuously monitor itself and recover from inconsistencies.
- It should be optimized to store large files, with file sizes in the multiple of gigabytes being the common case.
- It should primarily support two kind of "read" workloads: large streaming reads (up to few MBs) and small random reads (up to few KBs).
- It should support large sequential writes; updates are relatively rare, primarily file appends.
- It should support concurrent updates and reads, as there may be more than one application using the file system.

Google designed and implemented the Google File System (GFS) (Ghemawat, Gobioff, & Leung, 2003) to provide the above functionalities. An open-source implementation, which we used in the work described in this chapter, is the Apache Hadoop Distributed File System (HDFS). The HDFS

architecture is based on a single master (*Namenode*) and multiple workers (*Datanodes*) nodes. Files are divided in blocks of configurable size, with 64MB being a common block size for Hadoop. The HDFS Namenode contains the meta data about mapping files to blocks, and blocks to addresses in datanodes; these worker nodes are also sometimes referred to as "chunk servers". A client interacts with the master Namenode only when it has to look for a chunk location (whether reading an existing or, writing to a new chunk); it caches that location thereafter and connects directly to the chunk server without any master Namenode lookup. Thus the stored data is never directly seen by the master.

The workflows in the file system are such that the interaction between the client and the master Namenode is minimal, to prevent the master from becoming a bottleneck of the system and enable high throughput and performance. Fault tolerance is supported by the replication of the blocks on multiple datanodes, considering intra-rack and inter-rack topology while deciding which nodes. For example, if the desired replication factor is 3, the second copy is made on the same rack, and the third copy is made on a different rack. This ensures data availability even in the case of a rack failure. As might be expected, the trade-off is performance as copies are made to different racks and the client is not given a write acknowledgment until all the copies are made.

There are some workloads where random reads are required. In HDFS, this entails seeking to the byte offset and reading the file, which can be a costly operation given the large block size. To reduce this cost for sorted data, Hadoop provides a special format, *MapFile*. It comprises of two separate files: a data file that contains the actual data, and an index file that has the byte offset of a sequence of sorting-key values, separated from each other by a configurable gap. For example, a file with 1k records will have 10 entries in the index file if the gap is defined as 100. When reading, a *binary search* on the index file is performed and the correct byte offset is located. We use this feature in our migration to store the indices for the re-designed TAPoR.

2.3 HBase, a NoSQL database

HBase is a NoSQL database based on Google's *BigTable* (Chang et al., 2008), which was intended to address the requirements of random-read workloads. BigTable is a distributed, column oriented database, built on top of GFS (Ghemawat et al., 2003) in order to support random access patterns on large data. Apache *HBase* is an open source Java implementation of BigTable, which stores its contents (i.e. tables) on HDFS.

The results of MapReduce computations can be stored using a variety of methods; for example, the machine native file system, HDFS, HBASE or a relational database. The choice depends on the access patterns of the target application and the expected latency range. If the computed data is large (in the order of terabytes), storing it in a RDBMS is usually not appropriate. HDFS is best-suited to write-once-read-many workloads, not for random read-write operations. Reading a record in HDFS involves a TCP connection to the Namenode (this information can be cached), a TCP connection to the datanode, a *seek* to reach the specific record of interest, and then the actual reading of the data and transmission to the client. HBase is intended to improve random read-write performance.

HBase belongs to the general class of NoSQL databases. These databases are not a replacement of traditional relational databases, rather they should be considered their orthogonal counterparts, preferable in use cases where the data is unstructured. This is in contrast to the primary requirement for RDBMs, which is to define and model in the database schema the relations between the stored data entities. They are scalable to big data (petabytes), run in a clustered environment, provide fault tolerance, offer limited transaction support, and usually they do not support specific data types (other than byte arrays). The other main advantage of NoSQL databases is their theoretically

	CF1		CF2
RK1	RK1:CF1:CQ1:D:t9	RK1:CF1:CQ2:D:t4	
RK2	RK2:CF1:CQ1:D:t3 RK2:CF1:CQ1:D:t4 **RK2:CF1:CQ1:D:t5**		
RK3	RK3:CF1:CQ1:D:t1		RK3:CF2:CQ4:D:t11

Figure 2: HBase conceptual schema

unlimited scalability and elasticity; one just need to add/remove nodes to scale up/down. On the other hand, they do not support any standard querying language like SQL; they only have custom APIs to access the data, which is stored in de-normalized schemas. This proves good for unstructured data like text, logs, web pages etc, where meaningful relationships are implicit in the shared words and have to be inferred through processing, rather than explicit in the structure of the text.

There are many databases intended to provide cloud-type data management services, such as HBase[4], Cassandra[5], MongoDB[6], CouchDB[7], etc. Among these, Apache HBase is one of the most popular, having made its mark in academia and industry (Contributors, 2011; Zhang & De Sterck, 2010; Konstantinou, Angelou, Tsoumakos, & Koziris, 2010). Mendeley, an academic social network and a reference manager, has a collection of more than 99 million research papers and uses HBase at its backend (Contributors, 2011; Mendeley, 2011). Facebook, which invented Cassandra (Lakshman & Malik, 2010), the closest HBase competitor in its area, picked HBase as its data store for its recently launched messaging service (Muthukkaruppan, 2011). The official HBase clientele web page (Contributors, 2011) lists 38 companies that use HBase in their application stack, including Adobe, Facebook, HP, Meetup, Twitter, and Yahoo!.

The HBase Data Model HBase stores records sorted by a primary key; this is the only key in the entire schema, and the original HBase does not support secondary indices. After defining a table in HBase, and the *primary key* for each row, one has to define a (set of) column family(ies). As its name suggests, a *column family* represents a collection of columns which are accessed in a single transaction, like in one read/write call. Each column family is stored as a separate file on the file system. This helps to limit the number of disk I/O operations in one transaction. Column families are defined at table creation time, although from version 0.20.6 onwards, HBase supports adding column families at a later stage.

For the sake of flexibility, one need not define columns qualifiers before hand; they are appended to the given column family at run time. Each cell can be treated as an independent entity associated with its own *row key, column family, column qualifier, data, and creation timestamp.*

One can conceptualize HBase data to be stored in a three-dimensional table. Apart from length (number of rows) and breadth (number of column families and columns), the third dimension is the depth of the table (number of distinct cell values over time). One can access the previous value of a given *Rowkey:ColumnFamily:ColumnQualifier* cell by requesting a specific version number. Cell updates do not alter the cell value, rather a new value is appended onto the cell's *stack of values.*

Figure 2 represents a sample schema of a HBase table. In the figure, the cell value is depicted as *Rowkey:ColumnFamily:ColumnQualifier:Data:Timestamp*. There are two column families (`CF1` and `CF2`), and three rows. `Row 1` has two cell values in `CF1` and no value in `CF2`. `Row 2` has three

different values for the `CF1:CQ1` combination, the latest one with timestamp `t5`. `Row 3` has one cell in `CF2` with a qualifier `CQ4` (a row can have any qualifier for a given column family). There is no cost incurred by storing empty cells. It is possible to store sparse tables in HBase, expanding to millions of columns to billions of rows. It serves the exact requirement of low latency read and write (random and sequential) with large datasets in a clustered environment.

HBase API There are no datatypes in HBase; It stores all its data as byte arrays and provides the following API functions for accessing it:

1. *get(key)*: fetches a given byte[] row;
2. *getScanner(Scan)*: returns a *Scanner* object that is used to iterate on a subset of a table; the argument *Scan* defines the start/stop rows, column families, and other filters to be used while scanning the table;
3. *put(byte[] row)*: inserts a new row; and
4. *delete(byte[] row)*: deletes an existing row.

HBase tables are indexed based on a primary key, which enables fast access to the data when it is queried by row key, through a *get*. One can also sequentially access a range of table rows by providing the start and end row keys (through the *scan* function).

HBase Coprocessors Given the above APIs, the client program requests data through *get* (one row) or *scan* (multiple rows) and proceeds to process the collected rows. It is important to note here that the actual processing occurs at the client side, after the selected rows have been fetched from their respective *regions*, where a *region* is a subset of a table. HBase originally did not offer any support for deploying code at the nodes where the table regions are stored in order to perform computations local to the data and return results (instead of just table rows) to the client. This limitation makes the cost of several computations prohibitive. Consider, for example, a row-count process. In a very naive implementation, the client has to use the scan API to fetch the entire data and then count it locally. Alternatively, one may implement a MapReduce computation: using the scan API at the node level, the node rows are passed to a mapper process implemented at the node; the results (i.e. counts) are passed to the reducer process to do a global row count.

HBase *Coprocessors*, inspired by Google's BigTable coprocessors (Dean, 2011), are meant as a means of creating supporting functionalities to simplify the design of the main process and they are used to implement solutions for specific types of frequent workloads. In HBase, they are an arbitrary piece of software deployed per table region and can be used to act as an observer of any operation on the table, or perform a region-level computation.

When coprocessors are used as region observers one can compare them to relational database triggers. They can be used to observe any region activity, invoked either by a client using the APIs (get, put, delete, scan) or by a server-administration process (region split, memstore flush, compactions, etc).

When coprocessors are used for region-level computation one can compare them to stored-procedure objects in relational databases. They can be used to pre-compute results at the region level and feed these interim results to the client, instead of the raw table rows. This may result in reduced RPC traffic depending on the use case.

Consider for example the use case we described above: computing a row count on a subset of a table. Here, one can use a coprocessor to send the *local* row count in the target region back to the client, sums the individual results. The client library of coprocessor framework makes sure to execute all the calls to individual regions in parallel. Developers need to define their own coprocessor interface by extending the *CoprocessorProtocol* interface and instantiating a concrete implementation at the server side. The framework supports the invocation of any arbitrary coprocessor APIs from the client side and the retrieval of the coprocessor results by the client.

3 Legacy-System Migration to the Cloud

The potential advantages of HDFS, MapReduce, Hadoop and HBase have motivated migration efforts.

Shang *et al.* (Shang, Jiang, Adams, & Hassan, 2009) reported the use of MapReduce for software-repository mining. They deploy J-REX, an evolutionary code extractor for Java systems, on a four-machine Hadoop cluster. They report that the distributed version is four times faster than the single node version. Moreover, the migration to MapReduce required only a few hundred lines of code and it was easily adaptable to different cluster configurations. Their methodology is to create indices beforehand and render the result for subsequent queries using the pre-built indices.

Machine learning is an area which commonly involves computationally expensive tasks. MapReduce provides an interface to run such tasks in a clustered environment that hides complexities such as data partitioning, task scheduling, fault tolerance and inter-process communications. Panda *et al.* (Panda, Herbach, Basu, & Bayardo, 2009) used MapReduce for classification and regression tree learning on large datasets with their tool called PLANET. They used it to analyze click streams to predict the user experience following the click of a sponsored ad. Chu *et al.* (Chu et al., 2007) provide an overview of solving popular machine-learning algorithms (naive Bayes classification, Gaussian discriminative analysis, k-means, neural networks, support vector machines, etc.) in a MapReduce model using a shared-memory multi-processor architecture. Considering the popularity of MapReduce in machine learning, Apache has started Mahout[8], a project for implementing clustering, classification and batch collaborative filtering, on Hadoop.

On the HBase side, applications requiring continuous *editing* of the data prefer HBase as compared to plain HDFS because the later does not allow random file modifications. One can only append to an existing file in HDFS.

Another interesting family of HBase applications are those requiring high scalability and low latency workload, such as web applications. The BigTable paper (Chang et al., 2008) mentions that the BigTable infrastructure is being used for 90 Google products, the list including Google Maps, Gmail, etc. There has been similar workloads reported on the Apache HBase site by various users (Contributors, 2011).

In the research community, prior work of using HBase for document analysis has been reported. Konstantinou et al. (Konstantinou et al., 2010) used HBase for storing document indexes for a real time application. They used the existing APIs, and mentioned that their application has to make client-side merging of two queries before rendering the complete solution. It required two server trips before producing the end result. In a way, this case study underlined the need of an enhanced query support in HBase. (Vashishtha & Stroulia, 2011) used the coprocessors framework, in particular the endpoints variant, to provide an enhanced query support where the results are computed at the *region* level, and then all individual results are merged at the client side using the existing coprocessor callback feature.

In another work of creating and storing document indices, Li et al. (N. Li, Rao, Shekita, &

Tata, 2009) defined HIndex, that gets persisted on top of HBase and supports parallel lookup of target indexes. These indexes are fetched and the results are merged at the client side. In the work discussed in this chapter, we did use the coprocessor framework while creating indexes, but designed the schema such that the workload is limited to one row transaction and does not require any client side merging.

3.1 Migrating to the Cloud

In addition to Hadoop migration efforts, there has also been efforts in migrating software applications to other cloud platforms.

Frey et al. (Frey & Hasselbring, 2011) describe an approach for identifying the ways existing software would violate the constraints of a cloud service if deployed to it. Constraints include not writing to ephemeral storage, certain operations being disallowed in a PaaS model, etc. A profile of the cloud service constraints is created (they use Google PaaS as an example) and re-used. The OMG-standard Knowledge Discovery Meta-Model is used to model the appplication. Constraint violations are detected in the model and reported to the developer for repair prior to cloud deployment. They do not consider or address re-architecting an application for deploying to the cloud, nor do they actually migrate an application.

Li et al. (A. Li, Zong, Kandula, Yang, & Zhang, 2011) describe a tool to test the performance of an application on a variety of clouds prior to migration. They create a coarse-grained trace of an application's performance and emulate that on the cloud. They stop short of actually performing a migration.

REMICS (Mohagheghi & Sæther, 2011) is an EU FP7 research program focused on migrating legacy applications to the cloud. Their heavyweight model-driven approach is based on the OMG standard Architecture Driven Modernization and other existing models. Knowledge is extracted and a model constructed, model-driven engineering is used to produce a new system, and this system is validated against the existing system. At present they have not reported an implementation or evaluation of their approach. While their models are too comprehensive to reproduce here, we follow similar steps to achieve migration; our demonstration of a successful migration should be very relevant to their ongoing work.

Babar et al. (Babar & Chauhan, 2011) describe their process for migrating an application to the cloud. Their approach required simpler architectural changes and was performed at the IaaS level. Our PaaS approach offers different insights and tested various levels of re-architecting. While they do not explicitly consider the cost-benefit trade-off, they offer general observations on desirable properties of migrated application.

4 Text Analysis with TAPoR

Digital text is growing at an unprecedented rate, because of massive digitization efforts as well as because much of textual production these days is "born digital". A variety of analyses of this data can lead to the discovery of much meaningful knowledge. Such analyses can be done at the lexical level (that entails looking for most frequent words, their distribution across the documents and others) or at the semantic level (finding contextual meaning and relations among various entities in the data itself). In either case, the computation involved is intensive. In this chapter, we concern ourselves with a lexical-analysis tool, TAPoR, developed by Digital Humanists, who are frequently interested in analyzing and comparing different works of same author or works of different authors. This type of analysis can also vary in its scope, ranging from single document to a large corpus. For instance, an interesting use case may involve finding the most popular Latin word in either a single

Shakespeare novel, or in all of his collected works. Information gathered from these analyses can offer interesting insights. The typical workflow involves identifying a text corpus to analyze, then repeatedly executing a variety of queries on that corpus. Each query may require several iterations to adequately tune and filter the results.

The Text Analysis Portal for Research (TAPoR) is a web-based application that provides a suite of text-analysis tools to scholars and researchers in the Digital-Humanities (Rockwell, 2006). It includes a front-end portal and a back-end web service called TAPoRware. TAPoR has several deployments across the world and has an increasing number of users.

TAPoRware is a single web service with 44 operations, implemented in Ruby using the SOAP4R libraries[9]. Each operation runs in $O(n)$, and is bounded by CPU time relative to the size of the input. The tools covered in this paper are (a) listing the words of a document with their counts, (b) generating word clouds, (c) locating the use of a word in context (concordance), and (d) finding instances of two words located near each other in text (co-occurrence). These operations are described in (Table 1). In addition to accessing the functions via web services, they may also be accessed via CGI.

Table 1: The TAPoR operations (from `http://taporware.ualberta.ca`)

Operation	**Description**
List Words	Count, sort and list words of the source text in different orders. It can strip words user specified from the list, or common stop words.
Find Concordance	Find user specified word/pattern anywhere in the text with its context. Display the result in KWIC format.
Find Co-occurrence	Find user specified pattern and co-pattern in a given length of context. Highlight the patterns in the result.
Word Cloud	Using the top k frequent words reported by List Words, generate a visualization of those words and their relative frequency.

A typical TAPoR usage scenario begins with the end user identifying a piece of text to analyze with a given tool. Then, via a web-services client or the web front-end, the user selects the relevant parameters for the analysis. Common options include word stemming, excluding stop words, and defining the number of words/sentences/paragraphs to display results in-context. In addition, each operation has its own configuration options. The entire text to be analyzed and the configuration options are encoded in a SOAP request and transmitted to the web service.

The existing TAPoR implementation suffers several limitations that make the flexible experimentation of Digital Humanists with different collections a challenge. One major bottleneck is its lack of scalability to larger documents, due to its design. It was originally built to cater to small documents, processing the document in its entirety for each individual request, though a new request may have only a single different parameter from the previous one. So, if one does a concordance for a word "love" and now wants to do for "blood", it will process the entire document twice. This approach suffices when the document is small, but it is not scalable to larger documents (size over 5-10MB is sufficient to give a response time out error). This results in a poor end-user experience, as each request takes the same O(n) time, where n is proportional to the document size. An alternative to this approach is the standard way of creating indices of the document, and then using these indices for TAPoR operations. Index creation process is also costly and its cost is proportional to document size; but it pays off in the long run as this cost is incurred only once and the indices are

used over and over again. This is expected to be particularly useful for large corpora.

An initial step added the ability for TAPoRware to be load balanced over several processes (cores) or CPUs, or even over multiple physical or virtual machines (Smit et al., 2009). While this offered some basic scalability benefits, the cap on the size of text corpora still existed, and substantial computation resources were required to scale.

For our TAPoR-migration study, we chose four "recipes", i.e., either individual operations or compositions of operations, to support in the migrated version. We analyzed TAPoR access logs ranging for three years and profiled the workload. This showed that although TAPoR supports around 44 recipes, the four most popular ones cover 87% of the total requests over the past three years. We prioritized the operations to be implemented in the new service based on the frequency of their use in the old service (Table 2). The migration process benefits (in a way a new implementation would not) from making use of domain knowledge gathered during the lifetime of the original application. We found our decisions and understanding of the domain were heavily informed by the legacy application and its usage patterns.

Table 2: Percentage of requests for each operation, based on the original TAPoR deployment

Operation	%
Word Cloud	45.6
List Words	22.0
Concordance	16.3
Collocation	4.6
Co-occurrence	3.1
Pattern Distribution	3.0
Extract Text	3.0
Visual Collocation	1.8
Googlizer	0.6

5 The TAPoR Migration

In this section, we describe migrating TAPoR to Hadoop. We consider two closely related migrations. For the first case, we describe how we re-architected TAPoR to allow Hadoop to process the text, storing the indices on HDFS. For the second, we use the same architecture and Map-Reduce job for the text processing, but add improved functionality to store the indices in HBase.

5.1 Migration to Hadoop and HDFS

In previous work (Vashishtha, Smit, & Stroulia, 2010), TAPoR was migrated to use Hadoop, including a substantial design shift. Hadoop's algorithmic strengths are index building and batch processing; ideally migrated software makes use of these strengths. There is a certain amount of fixed overhead involved with a distributed file system and the map and reduce phases, so the task should warrant increased computing power, a distributed file system, or a robust framework that works around failures.

A key step was to **identify the functional requirements of the original and new services**, examining what functionality the existing service and new platform offers, and how might it be

offered in the new service. In particular, MapReduce is designed for offline, batch-processing tasks like index building. It is not intended for on-line processing or request-response style interactions.

For TAPoR, we decided an index-and-query strategy would make best use of MapReduce. Counting every word in a document, for example, necessarily involves reading every word in the document. To achieve the kind of speed-up we needed, we had to begin with an index. We also identified new functional requirements that could not be met by the old implementation but that we could provide during the transition. Our requirements included generating indices of large text corpora that would facilitate querying the collection, or defined sub-collections, in an offline fashion; and querying indices on request.

The next step was to **design the new service**. One important design consideration is whether to wrap the old service or to re-implement, or something in between. Some applications can be moved to a MapReduce paradigm by applying the thin veneer of a map step and reduce step to existing APIs or functions. For other applications, the map or reduce steps can be implemented such that most of the code is re-used, though integrated into a new application. For yet others, the best way to draw on the power of Hadoop is to re-implement the functionality. Regardless of the wrap/re-implement choice, one must determine what happens in the map phase and what happens in the reduce phase.

In our case, the old service processed chunks of text one-time, without maintaining (or persisting) any intermediate state of its computation. Repeated analysis of the same text requires re-uploading and re-analyzing the text from scratch. We chose to re-implement, creating our own indexing algorithms, our own index formats, and new querying algorithms. A substantially similar WSDL is used to maintain syntactic compatibility, though there is a semantic change to require uploading the name of an indexed document instead of the document, and an added operation for uploading a document to index as well as one to list the indexed documents.

We designed our indices to allow operations to re-use the same index, to allow us to divide collections into sub-collections easily (e.g., to analyze the collected works of William Shakespeare one day, and only *Hamlet* the next), and to be quickly searchable and sortable. An index has, for each word, a count of its occurrences in the collection, a list of the files that word appears in, and the byte locations for each of those files. For us, a collection consists of a set of files; sub-collections are subsets of this file set.

The need to keep key-value pairs sorted by source file required us to avoid the default Hadoop key (the map operation takes name-value pairs from the documents and distributes them to reduce nodes based on the name). We used a combination of file name and word to keep these keys sorted together by file name. In the map phase, each word is emitted as a key and its byte location and the corresponding file id as values. In the reduce phase, we combine the indices for each word found in the corpus to make a collective index. By default, this index is sorted alphabetically. We created a separate index sorted by frequency of word; one of our functional requirements is to support the common use case of asking for the top k words.

With these design changes, we turned to **implementing the new service**. A key implementation question is how to get indices/queries/data that live entirely in the Hadoop virtualized environment into the "regular" environment that service requests will arrive from, such as a Tomcat/Axis environment. The HDFS can be queried using the appropriate libraries (available in many programming languages but most reliable in Java); in the index-query methodology, the query algorithms can talk to HDFS without running on MapReduce nodes. Alternatively, indices can be moved out of HDFS and onto a local file system (allowing index-building computation power to be rented on the cloud and a smaller local server to run queries).

The migrated web service infrastructure is shown in Figure 3. The left side shows the submission of a document, which is processed by the index builder MapReduce task. The index stored in the

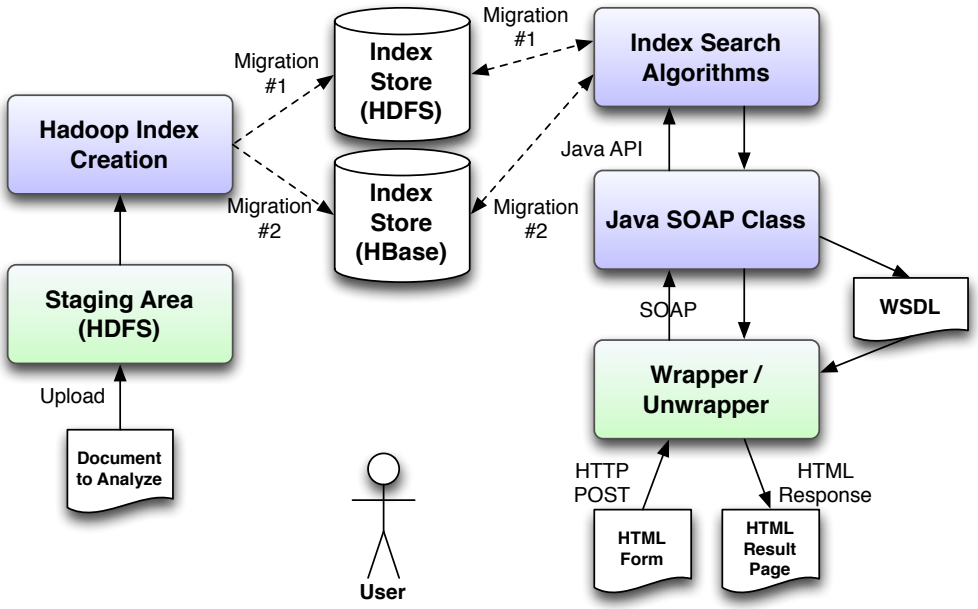

Figure 3: The architecture of the re-implemented TAPoR; either the HDFS index store (scenario 1) or the HBase index store (scenario 2) is used.

index store. For this scenario, the index is stored in HDFS. The various implemented operations need to access the indices in different ways. The List-Words and Word-Cloud operations need to access indices in a sequential manner, either sorted by frequency or alphabetically. The Concordance and Co-occurrence operations need to access indices specific to a key word. We use binary search for locating the required index entry. We used the MapFileFormat which provides the same functionality but with lower memory requirements.

The new web service runs in the Apache Axis SOAP stack[10] on Apache Tomcat 6[11], and is implemented in Java. The majority of the service was generated using the Axis `wsdl2java` tool. The WSDL for the new service is nearly identical to the old. The main technical change is a new URI for the service. The main semantic change is the "inputText" parameter is now the name of the text collection to query, instead of being the actual text to query. The current operations do not yet support all features of the old service.

Our SOAP server receives incoming SOAP requests and, based on the parameters in these requests, invokes the index-based query algorithms for the requested operations on the specified text collection. This invocation is done using the Java API made available by our index querying implementation. The API could be used by any mechanism, not just a web services server. The response is wrapped in an appropriate SOAP message and returned to the client.

Finally, we implemented a small front-end client and some command line tools that can be used to **test the new service**. Regression testing ensures the new service returns the same results as the old service, new functionality is tested, and the performance improvement is measured. A user visits a web page and selects a text collection to analyze, which operation to use, and the parameters for that operation. The web page submits via HTTP POST to our client. The form submission is converted into a SOAP message and transmitted to the server. The resulting response is extracted and converted to the appropriate output format before being returned to the user via a standard HTTP response.

Table 3: The TAPoR HBase schema

Row Key	**bl:foo**	**bl:bar**	**bl:sports**	**spl:Top100**
1	3123, 4223, #2	553, 643, 5544, #3		hello:105, world:56, love:45, blood:40
2			434, 423, 545, 646, #4	games:10, soccer:5, sports:4

5.2 Migration to HBase

In the previous section, we discussed migrating TAPoR to a HDFS based architecture. As we will see in the experimental results, it improved both scalability and performance. This section describes a further migration to using HBase, which is claimed to be a better alternative for an online querying type of system such as TAPoR because it provides a better random read performance; and since it uses HDFS as its backing file system, it also inherits its advantages.

Schema design The first step in using HBase is to create a schema for the data storage. The overall performance of an application using HBase is dependent on its schema, which should be designed with the expected workload in mind. Our design focused on the top four operations we are supporting (list words, word cloud, concordance, co-occurrence). We examined all four operations, in particular the expected reads and writes to the files. We considered the same indexing strategy used for the HDFS implementation when estimating reads and writes.

- List Words is computed from an index that includes a list of all the words in the document and the number of times they occur. This involves reading one index and not reading the source document.
- A Word Cloud is constructed using the output of a List Words operation, so the pattern will be the same.
- Concordance shows the presence of a target word in a dataset along with a configurable context. It is computed using a *position-based inverted index* that associates a word with all its byte locations in the document. For a given target word, the byte locations are retrieved from the index, then the document is read from those locations. This requires one read to fetch the index, and then a series of random reads from the byte locations named by the index.
- Co-occurrence shows how two target words are occurring together in a document. A user can provide a threshold to define the degree of closeness such as within 10 words or 5 sentences. For a given pair of target words, the algorithm fetches the index for each (as in concordance) and then read both indices, searching for pairs of byte locations likely to be within the given threshold. The document is read from these locations and further filter the candidate pairs based on the threshold (word and sentence distance cannot be determined by byte locations alone). This requires two reads (one per index) and a series of random reads from the selected byte locations of two documents.

We use HBase to improve the performance of the indexing; the actual files are still read from their locations in HDFS. The goal in schema design is to ensure data locality and to limit the

number of transactions. Recalling the nature of HBase column families (Section 2.3), we created the schema shown in Table 3. The HBase table (named "docIndex") has two *column families*, "bl" and "spl" ("byte location" and "special keywords" respectively). The row key is equivalent to a document ID.

This design has only one row per document. Each unique word becomes a *column qualifier* in the "bl" *column family*, and each cell value is a list of the byte offset locations in the document, followed by its frequency. In Table 3, we see the word "foo" occurred twice in Document 1, at byte offsets 3123 and 4223. We store the top K words in the "spl" column family; to read the top K words, one can simply read the "topK" column qualifier in the "spl" family. To read the byte offset of a word, one can retrieve the "bl" family with the target word as the column qualifier: "bl:word". All of these index operations are achieved in a single transaction.

Index Building The index creation part is done with the same Hadoop job that was used in the HDFS version, to compute the byte offsets for each word. These byte offsets are inserted in a HBase table in the Reduce phase. Once the reduce phase is complete, a coprocessor endpoint is used to compute the frequency index at the server side, by reading the frequencies of all the words in the document and sorting them in decreasing order. Using a coprocessor for computing the frequency index helped in distributing the computation among the datanodes instead of it all happening at the requesting machine. The inverted index of a document is substantially large, sometimes even larger than the document itself; with a coprocessor, the byte-offset indices do not need to be transferred to the client side. The coprocessor endpoint computes the top K words and stores them in the "spl" column family.

6 Experiment and Results

The existing legacy TAPoR is not capable of handling large files. In our testing, depending on the timeout settings, it struggled with files above 1MB. Ccomparing it to the migrated implementations, where we used a 4,000MB file, is not possible. As the scalability benefits are evident, we will primarily focus on the performance gains by the move to HDFS versus the further migration to HBase.

6.1 Environment

Where applicable, the original application was tested in a virtual machine that was allocated a single core of a Core2 Duo 1.86 GHz processor with 1GB of memory. It was tested using a 1 MB file from Project Gutenberg, Mary Shelley's *The Last Man*.

The testing of the migrated versions was conducted on a four-node Hadoop cluster. Each node had 8GB of RAM and two AMD Opteron 800 MHz processors. The nodes shared access to a 4 TB RAID 0 volume. Network latency was negligible. We used Hadoop version 0.20.3 and our modified version of HBase 0.90. The four nodes all served as task nodes and data nodes (running jobs and storing HDFS files), with one additionally assigned to serve as the master and run all of the coordination/dispatch services.

We used a dataset of 54 million tweets from Twitter for our experiments. The file was constructed by pseudo-randomly sampled tweets (chosen by Twitter in response to our API calls) from 46 non-sequential days over a one year period. Non-English tweets were removed. There are 46 documents in the dataset, one per day, for a total size of 4 GB. (The exact numbers are 53,958,269 tweets, average 11.4 words per tweet, average 71.6 characters per tweet).

6.2 Development Effort

Effort estimation is a non-trivial problem (Jorgensen & Shepperd, 2007). We report effort very simply, in terms of a) an objective metric, source lines of code (SLOC), and b) a subjective description of the conceptual effort required. For each, we excluded UI-related code, focusing on the implementation of the code necessary to answer the questions of the TAPoR users, and not on the formatting or transmission of the answers. We do not quantitatively assess effort.

Both implementations required an understanding of the MapReduce paradigm, as well as domain knowledge. We did approach this problem with an existing level of understanding of MapReduce, which helped when deciding on an indexing strategy capable of leveraging the strengths of MapReduce. Though we had access to the legacy application, several years of logs, and the client, we did not have access to the developer of the legacy application. We went through an informal requirements elicitation phase where we tried to understand the expected use of TAPoR and tools like it, including several meetings and attending a digital humanities conference.

Our implementation using HDFS required 650 SLOC. Implementing the HDFS version required familiarity with the basic concepts of a file system and the conceptual step to apply these concepts in a distributed fashion.

The HBase-driven version is a mostly independent code base, with only 70 SLOC overlapping (related to reading the context from the files stored in HDFS, which is the same for both). The TAPoR-specific code is 670 SLOC (including the overlap), with an additional 130 SLOC related to modifying the HBase coprocessors. Conceptually, this version required the same knowledge of HDFS, plus the additional understanding of HBase. Designing the schema required a solid understanding of how HBase wrote and retrieved information, and applying the domain knowledge of access patterns. There are additional HBase services (RegionServers, Master, ZooKeeper) that must be run. The changes to coprocessors required not only understanding their use but synthesizing an extension to perform text analysis computations locally.

Measured by SLOC alone, the two are fairly close. The domain knowledge required was also similar, and it was available from the legacy application and the client. In terms of conceptual effort, HBase required substantial additional effort to achieve the performance gains. This knowledge is not specific to the application; it is applicable to future migration and maintenance activities within the Hadoop ecosystem.

6.3 Methodology

The two variations differ primarily in creating, storing, and accessing the indices. We tested the index building process and two representative operations: List Words (which uses the same indexing strategy as Word Cloud) and Concordance (which uses the same indexing strategy as Co-occurrence).

For List Words, we read the top K words. In the HDFS version, the frequency indices are stored in a file and we simply read the top K lines; in the HBase version, the frequency indices are stored as columns in the "docIndex" table. This presents a comparison between HDFS and HBase read performance.

For Concordance, we read the byte offsets of the target word, and then do a read on the file at the selected byte offsets. We limited our experiments to the first portion of request processing, specifically excluding the step of accessing the original documents on HDFS, which is identical for both variations. As mentioned in Section 5.1, we stored the indices in a MapFile in the HDFS version, which supports binary search on its content. The index-reading phase presents a comparison for random reads in HDFS vs. HBase, because the MapFile is a large sorted file, and

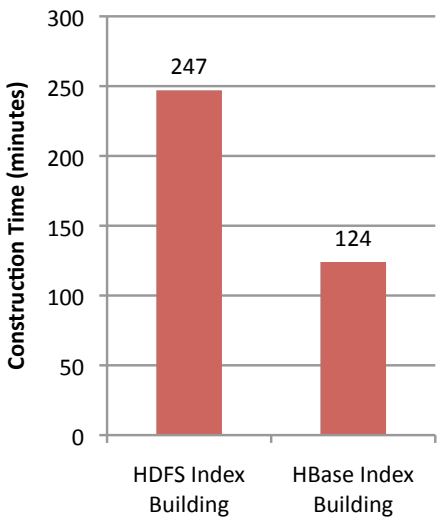

Figure 4: Time required to construct the index for the two methods.

we are reading the words at random.

We used YCSB (Yahoo! Cloud Serving Benchmark) (Cooper, Silberstein, Tam, Ramakrishnan, & Sears, 2010) to generate load requests similar to List Words and Concordance requests. We created a workload of 500 requests and measure the throughput of both the approaches. The workload consisted of List Words requests for randomly selected documents, and Concordance requests for randomly selected documents and randomly selected words from a common set of the 50 most frequent words. There was only one client for all the 500 requests. This is done to make it more realistic where a client caches the file blocks location (in case of HDFS), or *Region* location (in case of HBase).

6.4 Results

The migrated versions show substantial performance improvements on a per-request basis. In earlier work (Vashishtha et al., 2010), we showed that on the 1MB test file, the legacy application took 29.75 seconds and 7.27 seconds for List Words and Concordance, respectively. The migrated application (HDFS) indexed the 1MB test file in 60 seconds (mostly overhead) but responded to requests in .4 and .6 seconds, respectively. These numbers include UI processing, network lag, and opening the files in HDFS to obtain context. The cost of indexing is saved within 2-8 requests.

Turning to comparing the two flavors of the Hadoop migration, the HBase version was able to index the 4GB dataset in half the time of the HDFS version (Figure 4), reducing the time between submitting a dataset and being able to query that dataset by almost 2 hours in this case. This can be attributed to two main reasons. First, in the case of the HDFS version, we needed to run two MapReduce jobs, where the second one was to compute the frequency index. In HBase, the frequency index is computed using a coprocessor endpoint. The Hadoop MapReduce framework has a high starting cost, making the overall cost higher. The second reason is that the write rate of HBase is inherently higher than that of HDFS.

For the List Words service, the HBase approach more than doubled the throughput of the HDFS approach (Figure 5). For Concordance, the HBase throughput was almost 80 times that of the HDFS approach. Though the end-user impact will be reduced as they still have to wait for

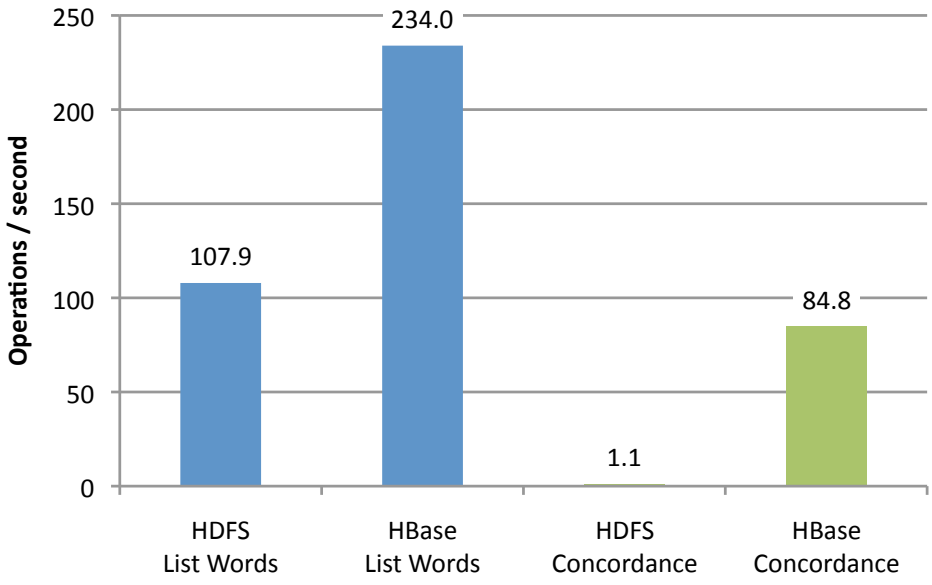

Figure 5: The number of operations per minute achieved for each service, for each migration flavor.

the HDFS retrieval of context from the original document, the performance improvement of the index process in HBase is substantial. A charitable pro-rating of the original application shows a 4GB file, if it could be processed leveraging the same computing power, would have a throughput of .0002.

Turning to **scalability**, the original application was limited to several megabytes per request. Though we have not explored the limitations of the two approaches, we know that HDFS has been successfully used for similar tasks at the Terabyte scale, and we have demonstrated fast response times at the Gigabyte scale. When comparing the two migrations for scalability, HBase has a more extensible indexing approach, where additional indices can be added as columns to the existing HBase table. We suspect that HBase will support the migration of additional TAPoR services with smaller index size and better scaling than HDFS. We also know we can add additional index information to the HBase table in other column families without increasing the processing time of the existing indices. In the HDFS approach, however, we need to have separate files for each index type, or, possible, design a creative representation to use the same index files for multiple purposes.

6.5 Lessons Learned

While migrating TAPoR to index based HDFS/HBase stack greatly reduces the latency as compared to the original implementation, it becomes increasingly difficult to maintain it as the number of users and documents increased. This is because each of the indexes were kept in a separate directory, which resulted in a *directory explosion*. This is an alarming situation for HDFS because the Namenode keeps the filesystem metadata in memory and each directory increases its memory footprint.

Another suboptimal usage of HDFS in the initial design is the TAPoR workload is more about random reads rather then large sequential reads, while the HDFS is specially designed for the later kind of workload. These reasons led us to explore other options such as HBase. With HBase, one can map the one directory per index design to a row(s) in a table. And since HBase stores its data

in HFile, one does not need to care about the problem of small documents increasing *Namenode* memory footprint. HBase also provide much better random reads as compared to plain HDFS as shown by the experiments in Section 6.

One key consideration while using HBase is the right schema, which along with meeting the application-specific workload needs, should also be optimized from the HBase perspective. In case of TAPoR, our analysis suggests two level of indexes, one at the document level (for list words and word cloud), and other at the document-word level (for concordance and co-occurrence). We initially designed a schema, with each row corresponding to a word, containing as its values the byte-offset locations of the word in the document. To distinguish a word for a document, each such word is prefixed with an auto generated document ID. Table 4 gives an example of such a schema for a document, with its ID as #1. The row key of a document level index in such a schema will be special keyword, like "Top100" for the top 100 words, as shown in the table. The document with ID 2 also has the word *foo* but it is in a separate row, as shown in the last row of the table.

Table 4: An alternative schema for TAPoR on HBase

Row Key	**CF:byteLoc**
doc#1,foo	3123, 4223,#2
doc#1,bar	553,643,5544#3
doc#1,...	....
doc#1,Top100	hello:105, world:56, love:45, blood:40
doc#2,foo	909, 656,6786#3

This schema meets all the requirements for looking a specific word (one just need to add the document ID as the prefix), and also for the List-Words functionality where the *keyword* is given as input. The only drawback with this design is that it results in a tall table, as there is one row for each unique word in the document. This is a drawback because of how HBase stores and reads its data. As mentioned above, HBase stores its data in a special file, called HFile. The data in a HFile is arranged in data blocks, followed by an index block at the end of the file. The index block keeps the index of the starting element of each of the data block. While reading a specific record, a binary search on the index block can tell whether that target record is in the HFile or not. For faster look-ups, this index block is kept in memory of the hosting RegionServer. The above schema results in a large number of rows (a 1 MB document has on average 220,000 words), and it increases the index block size in the HFile. This index block size is directly proportional to the number of rows in the table; as a result this is not an optimal schema as it increases the memory footprint for the application. So, even though all our the required functionalities are met by this schema, it is still suboptimal. We believe that this experience with Hadoop and HBase will help others while designing/migrating their application to the Hadoop system.

7 Conclusion

Migrating this legacy application required substantial domain knowledge and knowledge about the target platforms, HDFS, Hadoop and HBase. We believe that our experience is relevant to the migration of document-manipulating applications in general, since many of those manipulations are bound to refer to words, their locations and their relations. In the first migration phase, the HDFS indices, we achieved substantial improvements over the previous approach in both performance (over time) and scalability. The demonstrated difference in scale is three orders of magnitude, with

the limit not established but believed to be at least another 6 orders of magnitude higher. The demonstrated difference in performance is estimated at 4 orders of magnitude.

The second migration phase, moving the indices to HBase, required the additional understanding of HBase. Modifying the source code of the computing platform is a substantial development investment, and adding something like co-processor support to the cloud computing platform requires deep conceptual understanding. In terms of SLOC, the effort was not much more than the first phase; in terms of expertise required, there is a substantial increase. In exchange, there was an increase in performance - the indexing time was halved, the per-lookup index performance improved by about two orders of magnitude, and the bound on index size relative to document size was halved. We do expect that migrating additional TAPoR services along the same path will require substantially less effort now that the modifications are in place, and we believe the index will be more efficient and more extensible even with these additional services.

The increased effort in migrating did achieve notable performance improvements. The initial move to a scalable, distributed computing platform did achieve the highest gains, for even lower overall cost. However, there was still return on investment for additional changes and optimizations to leverage that platform.

The next step of this work is to measure the bounds of scalability of this approach to see if we can achieve in practice what is promised by theory. We would also like to examine more closely the impact this has on user experience. Will TAPoR users actually run different analyses if they know repeated requests while fine-tuning parameters will return quickly? At least some of the benefits achieved here were due to the redesign, and we would have improved performance and scalability even in the absence of large-scale computing platforms. The next step in the evaluation is to run the index-based method outside of Hadoop, on a bare-metal machine, and compare the performance and scalability to the versions running on the Hadoop ecosystem. Finally, applying our approach to migrating other applications would show the extent to which our approach and results are generalizable.

Acknowledgments

The authors received funding from AITF (formerly iCore), IBM, and NSERC.

References

Babar, M. A., & Chauhan, M. A. (2011). A tale of migration to cloud computing for sharing experiences and observations. In *Proceedings of the 2nd international workshop on software engineering for cloud computing* (pp. 50–56). New York, NY, USA: ACM.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., et al. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, *26*(2), 1–26.

Chu, C., Kim, S., Lin, Y., Yu, Y., Bradski, G., Ng, A., et al. (2007). Map-reduce for machine learning on multicore. In *Advances in neural information processing systems 19: Proceedings of the 2006 conference* (p. 281). Cambridge, MA, USA: The MIT Press.

Contributors, H. W. (2011). *Hbase/poweredby - hadoop wiki.* `http://wiki.apache.org/hadoop/Hbase/PoweredBy`.

Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st acm symposium on cloud computing* (pp. 143–154). New York, NY, USA: ACM.

Dean, J. (2011). *Designs, lessons and advice from building large distributed systems.* http://www.odbms.org/download/dean-keynote-ladis2009.pdf.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.

Frey, S., & Hasselbring, W. (2011, march). An extensible architecture for detecting violations of a cloud environment's constraints during legacy software system migration. In *Software maintenance and reengineering (csmr), 2011 15th european conference on* (p. 269 -278).

Ghemawat, S., Gobioff, H., & Leung, S. (2003). The Google file system. In *ACM SIGOPS operating systems review* (Vol. 37, pp. 29–43). New York, NY, USA: ACM.

Jorgensen, M., & Shepperd, M. (2007). A systematic review of software development cost estimation studies. *Software Engineering, IEEE Transactions on*, *33*(1), 33 -53.

Konstantinou, I., Angelou, E., Tsoumakos, D., & Koziris, N. (2010). Distributed indexing of web scale datasets for the cloud. In *Proceedings of the 2010 workshop on massive data analytics on the cloud* (pp. 1–6). New York, NY, USA: ACM.

Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, *44*(2), 35–40.

Li, A., Zong, X., Kandula, S., Yang, X., & Zhang, M. (2011). Cloudprophet: towards application performance prediction in cloud. In *Proceedings of the acm sigcomm 2011 conference* (pp. 426–427). New York, NY, USA: ACM.

Li, N., Rao, J., Shekita, E., & Tata, S. (2009). Leveraging a scalable row store to build a distributed text index. In *Proceeding of the first international workshop on cloud data management* (pp. 29–36).

Mendeley, L. (2011). *Free reference manager and PDF organizer.* http://www.mendeley.com/.

Mohagheghi, P., & Sæther, T. (2011). Software engineering challenges for migration to the service cloud paradigm: Ongoing work in the REMICS project. In *Services (services), 2011 ieee world congress on* (p. 507 -514).

Muthukkaruppan, K. (2011, Nov.). *The underlying technology of messages.* http://www.facebook.com/note.php?note_id=454991608919.

Panda, B., Herbach, J. S., Basu, S., & Bayardo, R. J. (2009). PLANET: massively parallel learning of tree ensembles with MapReduce. *Proc. VLDB Endow.*, *2*(2), 1426–1437.

Rockwell, G. (2006). TAPoR: Building a portal for text analysis. In R. Siemens & D. Moorman (Eds.), *Mind technologies; humanities computing and the canadian academic community* (p. 285-299). Calgary, AB: University of Calgary Press.

Shang, W., Jiang, Z. M., Adams, B., & Hassan, A. E. (2009). MapReduce as a general framework to support research in mining software repositories (MSR). In *Msr '09: Proceedings of the 2009 6th ieee international working conference on mining software repositories* (pp. 21–30). Washington, DC, USA: IEEE Computer Society.

Smit, M., Nisbet, A., Stroulia, E., Iszlai, G., & Edgar, A. (2009, Nov). Toward a simulation-generated knowledge base of service performance. In *Mwsoc '09: Proceedings of the 4th international workshop on middleware for service oriented computing.* New York, NY, USA: ACM.

Vashishtha, H., Smit, M., & Stroulia, E. (2010). Moving text analysis tools to the cloud. In *Ieee congress on services* (p. 107-114). Los Alamitos, CA, USA: IEEE Computer Society.

Vashishtha, H., & Stroulia, E. (2011). Enhancing query support in hbase via an extended coprocessors framework. In *Proceedings of the 4th european conference on towards a service-based internet* (pp. 75–87). Berlin, Heidelberg: Springer-Verlag.

Zhang, C., & De Sterck, H. (2010). Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase. *Proc. of Grid2010*.

Notes

[1] http://aws.amazon.com/

[2] http://www.salesforce.com

[3] It should be noted here that this approach relied on our own modifications to HBase and that these modifications were sufficiently general to be contributed and accepted to core HBase, and will be generally available in version .92.

[4] http://hbase.apache.org/

[5] http://cassandra.apache.org/

[6] http://www.mongodb.org/

[7] http://couchdb.apache.org/

[8] http://mahout.apache.org

[9] http://rubyforge.org/projects/soap4r/

[10] http://ws.apache.org/axis2/

[11] http://tomcat.apache.org/