Elsevier Editorial System(tm) for Future Generation Computer Systems
Manuscript Draft

Manuscript Number:

Title: SciPDFindexer: Distributed Information Retrieval system using MapReduce

Article Type: Full Length Article

Keywords: distributed system; indexing; MapReduce; scientific articles; information rerieval

Corresponding Author: Mr. Sangyoon Oh, Ph.D.

Corresponding Author's Institution: Ajou University

First Author: Aziz Murtazaev, M.S.

Order of Authors: Aziz Murtazaev, M.S.; Geoffrey C Fox, Ph.D.; Sangyoon Oh, Ph.D.

Abstract: Indexing allows converting raw document collection into easily searchable representation. Bigger scale indexing poses some challenges how to distribute indexing computation efficiently on a cluster of nodes. MapReduce framework promises to be an effective tool for parallelizing computation, especially for divide-and-conquer type of problems, such as inverted index construction. We propose SciPDFindexer, distributed information retrieval system for scientific articles in PDF. Information retrieval (IR) of scientific papers is not much researched domain as general IR and differs from general one mainly by structure of documents and the necessity for converting them into indexable representation. In our system, our main focus is distributed indexing scheme using MapReduce. Given large collection of scientific articles in PDF our system parses and extracts metadata from articles, and then indexes extracted content using our proposed scheme. We also parallelized querying by using distributed database HBase for holding indices. We evaluated our indexing scheme by comparing with alternative approaches and showed that our indexing scheme performs equal or better than the compared ones. In the other experiment, we also investigated impact of different MapReduce parameters to indexing performance.

Suggested Reviewers: George Fletcher Ph.D.
Assistant Professor, Information Systems W&I, Eindhoven University of Technology
g.h.l.fletcher@tue.nl
Dr. Fletcher is actively researching on the DB, RDF, and parallelism.

C. Lee Giles Ph.D.
Professor, College of Information Sciences and Technology, Pennsylvania State University
giles@psu.edu
Dr. Giles directed the CiteSeerX project.

Harshawardhan Gadgil Ph.D.
Software Design Engineer, Microsoft
hsgadgil@gmail.com
He has been researched and published papersin parallel and distributed computing.

Sangwon Lee Ph.D.
Associate Professor, Information and Communication Engineering, Sungkyunkwan University

wonlee@ece.skku.ac.kr
He has published over over 50 international journal and conference proceedings and one of them is cited 168 times. He could be a good reviewer for papers in DB area.

Highlights

> We propose a distributed information retrieval system for scientific articles in PDF

> Our system parses and extracts metadata from articles, and then indexes extracted content using our proposed scheme

> We solve the problem of bigger scale indexing and querying by apply parallelism, using MapReduce and HBase

> Evaluation results shows that our indexing scheme by comparing with alternative approaches and showed that our indexing scheme performs equal or better than the compared ones

SciPDFindexer: Distributed Information Retrieval system using MapReduce

Aziz Murtazaev[1], Geoffrey C. Fox[2] and Sangyoon Oh[1]
[1]School of Information and Communication, Ajou University
Suwon, South Korea
[e-mail: {muraziz, syoh}@ajou.ac.kr]
[2]Community Grids Lab, Indiana University, Bloomington, Indiana, USA
[e-mail: gcf@indiana.edu]
*Corresponding author: Sangyoon Oh

Abstract

Indexing allows converting raw document collection into easily searchable representation. Bigger scale indexing poses some challenges how to distribute indexing computation efficiently on a cluster of nodes. MapReduce framework promises to be an effective tool for parallelizing computation, especially for divide-and-conquer type of problems, such as inverted index construction. We propose *SciPDFindexer*, distributed information retrieval system for scientific articles in PDF. Information retrieval (IR) of scientific papers is not much researched domain as general IR and differs from general one mainly by structure of documents and the necessity for converting them into indexable representation. In our system, our main focus is distributed indexing scheme using MapReduce. Given large collection of scientific articles in PDF our system parses and extracts metadata from articles, and then indexes extracted content using our proposed scheme. We also parallelized querying by using distributed database HBase for holding indices. We evaluated our indexing scheme by comparing with alternative approaches and showed that our indexing scheme performs equal or better than the compared ones. In the other experiment, we also investigated impact of different MapReduce parameters to indexing performance.

Keywords: distributed system, indexing, MapReduce, scientific articles, information rerieval

1. Introduction

When performing search over the whole contents of a set of documents, scanning them one-by-one could be suitable only for very small collections. However, for larger collections this way of searching is inefficient due to considerable response time. Usually larger collections are scanned, analyzed and *indexed* before making any query on them. This approach greatly reduces response time of searching.

For large-scale indexing we usually use distributed set of nodes, as a single node can take intolerable long time for such scale tasks. Large-scale indexing poses some challenges of how to perform index construction efficiently using distributed systems. Google performs very well in indexing enormously large data; the number of indexed web pages is estimated over around 45 billion according to [1], and still it is able to give sub-second query response time.

Indexing job can be efficiently performed in distributed system as it complies to divide and conquer style processing. That is, large collection of documents can be split to smaller chunks and processed independently in a set of nodes, and finally the results can be merged to produce final index structures. One of parallel processing techniques suitable for divide and conquer type of problems is MapReduce programming model which was introduced by Google [2]. MapReduce showed excellent scalability and performance by sorting 1 Tb data in 68 seconds using 1000 machines and sorting 1 Petabyte data in 6 hours and 2 minutes using 4000 machines [3]. MapReduce provides simple interface to programmers in the form of map() and reduce() functions, and underlying framework handles parallelization issues, such as splitting input data, moving intermediate data to corresponding nodes, sorting, grouping intermediate keys.

Even though MapReduce can automatically scale, the way we choose key-value pairs and how we process them in map and reduce phases affects overall job performance. In case of indexing, there are a few indexing schemes which does the same job, but the way the documents are indexed differs from one another. We believe that choosing the right scheme for indexing is important in terms of indexing efficiency measured by indexing throughput.

Information Retrieval (IR) in general is a well researched area. Up to now researchers proposed various indexing schemes, ranking models, index structures, index storages targeting indexing efficiency, increased query results relevance, and most of researches are related to indexing text and web-sites, verifying their approaches with such collections as TREC [4]. Scientific papers information retieval is not as much researched domain and it has some specifics which requires it to be considered separately. Specifics in this case is that usually scientific papers are given in PDF, in various layouts. In general case in indexing we deal with analyzing text by chopping text into tokens, normalizing them, removing very frequently used words (stop-words). However, when we deal with scientific papers indexing requires these papers to be converted to proper textual format before the procedures described above. And scientific papers have specific structures, and each part of the structures need to be treated separately.

There exist researches about IR of scientific papers and there are few which considers this as a whole system, which describes all from parsing to indexing to querying in detail. Considering IR of scientific paper as a whole system is important as each part of that is interrelated, that is, how we parse, what structure we obtain from parsing affect how we index documents; the way we index the documents and the index structures we choose affect querying performance. Among the few, there are notable ones like NEC's CiteSeer (and its decendant CiteSeer^{x}), Google Scholar and MS Academic Search. All of them are indexing system that index academic literature in electronic format (e.g. Postscript files) [25].

We propose an IR system, called *SciPDFindexer*, for parsing, indexing and querying scientific articles in PDF. Given large corpora of scientific articles in PDF our system parses and extracts article contents with additional metadata, such as title and abstract, and then indexes extracted content using MapReduce framework in a distributed system. Our querying system, which is also parallelized using distributed database, enables free text querying on the resulted indices. Even though our proposed system, *SciPDFindexer* is not as much large-scale and not provising as many features as major ones (e.g. 'group of' and 'cited by' of Google Scholar and a Object of MS Academic Search), our contribution is still noteworthy because we propose a new indexing and query scheme that improves the performance with parallelism and focuses on the service for the medium size academic or research organizations. Our contribution is twofold: first, we propose distributed indexing scheme suitable for our domain and second, the design and implementation SciPDFindexer system.

We designed indexing system as consisting of two parts: Preprocessing, where scientific document collection in PDF is converted into textual representation with corresponding scientific article structure, and Text-indexing, where inverted index structures are built by analyzing parsed documents. Both of these parts run as consequitive MapReduce jobs, the

output of one goes as input to another. We designed special parsing algorithm which parses title, abstract and body text from given scientific article in PDF.

We provide our proposed distributed indexing scheme along with similar alternative two schemes, and provide details how they differ from one another. For ranking model, we use TF-IDF weighting scheme, where the weight of the term is proportional to number of times a term appears in document and inverse proportional to frequency of term in a whole collection. We use HBase to hold index data and it enables to have term-partitioned indices. This allows HBase to perform querying in parallel, as the terms along with their posting-lists are distributed in a cluster of nodes.

We conducted several evaluations of our system. First, we showed that our indexing scheme performs better than alternative schemes. Second, we discovered how MapReduce parameters affect the performance of indexing. And the last one, we showed that our querying system gives sub-second response time for various length of queries.

The rest of this paper is organized as follows. In Section 2, we talk about MapReduce programming model and other related works. In Section 3, first, we give overall architecture of SciPDFindexer, then describe indexing system and querying system separately. We present our system implementation in Section 4. We evaluate our system and present results in Section 5, and finally, in Section 6 we conclude our work show future directions of our research.

2. Background and Related Work

Our research is related to such disciplines as parallel computing with MapReduce framework, distributed indexing schemes and information retrieval of scientific papers. We provide overview of MapReduce framework and mention other works related to indexing and scientific papers' IR.

A. MapReduce framework

MapReduce is a programming model introduced by Google which enables specifying two user functions, *map* which processes key/value pair and generates another intermediate key/value and *reduce* which merges all intermediate values related with the same intermediate key [2]. This model came into being after realizing that developing parallel applications became very tough because of parallel computing issues, such as providing fault tolerance, replication, availability, scalability. MapReduce framework allows programmers focusing on key components, while infrastructure management logic, such as fault tolerance, scheduling, tracking jobs are done by the underlying framework.

MapReduce framework has become popular in both academic and business area. Open source counterpart of Google's MapReduce framework named Hadoop [5] written in Java has been developed later and is currently being used by Yahoo, Facebook, Amazon and many others'. MapReduce model is suitable for solving data-intensive and compute-intensive tasks with good performance. Within Google MapReduce model is used for such tasks as large-scale machine learning problems, clustering problems, large graph computations, web-pages properties extraction [2]. Besides that, MapReduce can be used for such tasks as inverted index construction, matrix multiplication, string matching, KMeans, linear regression and many others [6]. MapReduce is designed to work in a cluster of commodity hardware where the storage and computation are effectively used by having multiple replicas of data over the cluster and utilizing data-locality by moving computation to data.

We used Hadoop implementation of MapReduce in our system to parallelize parsing and indexing processes. Hadoop Distributed File System (HDFS) [7] is used as storage for collection of PDF documents which is used as input to MapReduce jobs.

B. Indexing schemes

MapReduce framework provides a convenient interface for construction of inverted index, as input data can be processed as a series of key-value pairs and framework functionalities as

splitting, sorting and grouping can be utilized. Different indexing schemes were proposed already which uses MapReduce to distribute processing among cluster of nodes. Original MapReduce paper shows inverted index construction as one of examples suitable for their framework [2]. The algorithm they described is very simplified and we adapted their algorithm to our domain to be suitable for comparison.

J. Lin *et al.* in [8] proposed distributed retrieval system named *Ivory* using Hadoop and one of their focuses is distributed inverted indexing algorithm. Main difference of their scheme from ours is that their postings are *document-sorted*. They changed the structure of intermediate key-value pairs to utilize MapReduce sorting functionality to obtain document-sorted postings. In our case we upload posting-lists to HBase, hence, we do not need to sort document IDs. We present Ivory scheme in more details in our comparisons.

Another indexing scheme which uses MapReduce is incorporated into Nutch, open-source Web search engine [9]. In that scheme, map processes a set of documents and instead of outputting postings, it outputs analyzed document. This results in *document-partitioned* index in each of reducer nodes, where each node will hold posting-lists of independent subset of documents. This is different from ours, as we build *term-partitioned* index (with HBase), where each node holds a subset of all terms with posting-lists covering whole document collection.

McCreadie *et al.* [10] proposed distributed indexing scheme utilizing both single-pass indexing algorithm and MapReduce framework. In their scheme, instead of emitting terms per each term in each document, they build up posting-list in memory and emit them only when memory is exhausted or when all the data is processed. This approach is similar to ours, as in our proposed scheme with combiner we also emit aggregated posting-lists instead of individual postings. Their in-memory accumulation of postings could give some little performance advantages over other implementations, however, they need to take care about low-level details as memory requirements, memory checks, while in our case we offload these low-level computations to framework itself, hence making our code simpler.

C. Information retrieval of scientific papers

There have been several works related with information extraction from scientific papers. S. Lawrence et al. [11] proposed an Autonomous Citation Indexing (ACI) system named CiteSeer (and updated system - CiteSeerX [12]). Their main goal is to organize scientific literature openly available in the Web by automating creation of citation indices. Their system crawls scientific articles from the Web, extracts citations, and indexes full-text articles as well. Their system can identify the same citations given in different formats and tracks them in citation database. And users can query these articles where the resulted documents can be sorted by number of citations to that document. Their focus is on linking articles by citations and keeping citation statistics for their collection of documents, while our system focuses on indexing with MapReduce and parallel querying in a cluster of nodes. Also, we work only with pre-defined repository, while CiteSeer tries to cover whole Web.

Verstak and Acharya release Google Scholar, its free accessible search engine for scientific papers in 2004. Along with its vast amount of indexed articles and its unique ranking algorithm, Google Scholar provides many convenient features like ‘group of’ and ‘cited by.’

Developed by Microsoft Research Asia, Microsoft Academic Search is also one of most popular free search engine for scientific papers which is focused on computer science, electrical engineering and physics [26]. Unlike Google Scholar which discloses the list of coverage, it lists publishers online.

Briscoe *et al.* [13] propose a search engine for scientific literature allowing sentence-level search. One of the main features of their system is that they integrated text and image search which enabled fine-grained search for scientific papers. While they focus on improving search experiences for users, we focus on providing scalable indexing system. And they use Lucene [14] for indexing and retrieving results while we build our own.

Karamuftuoglu *et al.* [15] discusses their experimental retrieval system of scientific articles based on a probabilistic model. They discuss many issues related with field searching, query language and terms, query expansion. They propose retrieval system based on probabilistic model, called Okapi, which uses similar steps as our indexer: pre-processing, parsing, normalizing text. In their system, they receive relevance judgment feedback from users and use this information in further expanding queries. Despite that their application domain is same with ours, we focus on indexing performance rather than providing complex type of searching.

3. Design of SciPDFindexer system

In this section we describe the design of our proposed system. First, we show overall architecture of our system describing system components and their interactions. Then we describe indexing component of the system, and particularly show the reasons why we separated indexing job into two parts: preprocessing job and text-indexing job. In preprocessing description, we explain how documents in PDF are converted to textual format maintaining structure of scientific papers. After that we discuss about various distributed indexing schemes and propose ours showing the differences from alternative indexing schemes. Finally we discuss querying system design, particularly, the ranking model used in our system.

3.1 System overview

The overall architecture of our system is depicted on **Fig. 1**. SciPDFindexer accomplishes two tasks: indexing documents and querying on resulted indices. As we can see from this figure, those two tasks correspond to the two major components: *Indexer* and *QueryParser*. Those two components are central components of indexing and querying tasks respectively. There are several subcomponents, which are used either by *Indexer* component or *QueryParser* component or shared by both. The workflow of the system is numbered separately for indexing and querying processes in this figure.

In the indexing process, first, corpora of PDF files are provided to *Indexer* (1), and *Indexer* uses *PDFParser* subcomponent to parse PDF into plain text and extract context information as Document fields, such as Title, Abstract, Body (2,3). Then plain text is sent to *TextAnalyzer* subcomponent to chop the text into small meaningful units of text – *tokens* (4). Besides just breaking text into tokens, *TextAnalyzer* performs additional job, such as removing very frequently used words, which has little meaning in the text (articles, prepositions) and also it extracts basic morphological form of the words for the purpose of grouping words with the same morphological origin (‘jumping’, ‘jumped’ → ‘jump’). As a result of these normalization operations we obtain a list of *term*s which is ready to be saved to DB. *Indexer* receives this list of terms along with their document mappings and frequency of the term in that document (5) and saves them to the *Index Database* (6) where each entry will be represented as (term, posting-list) pairs.

In the querying process, user searching for some keywords (1), inputs them into *Search UI*, and it sends the query to *QueryParser* component (2). That query will be analyzed first by submitting it to *TextAnalyzer* (3,4), which breaks search text into tokens and then normalized, in the same way it was done in the indexing process. Using the same analyzing algorithm for both indexing and querying process is essential, as querying process will look up the DB which contains already analyzed bunch of terms. Next, those set of terms received from *TextAnalyzer* will be looked up in the *Index DB* containing (term, posting-list) pairs (5,6). For each term, posting-lists containing document IDs and frequency of occurrences of this term in this document will be merged and scores calculated for each document appearing in the set of posting-lists (7,8). This is done in the *Ranking* subcomponent, which scores the documents containing more search terms higher, and the results are displayed in the *Search UI* (9) as a set of document titles with the links to the actual documents.

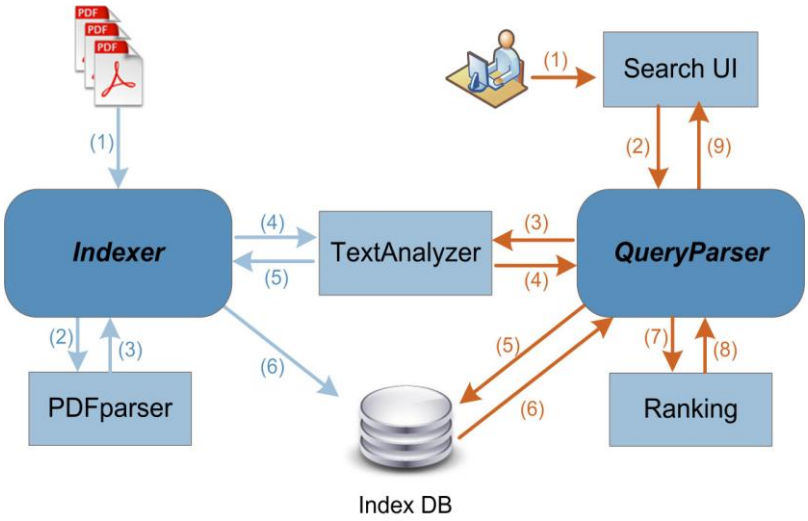

Fig. 1. Overall architecture of SciPDFindexer

This was high level overview of the SciPDFindexer system which we designed and implemented. In single node mode, the interaction among the components is exactly as described in the **Fig. 1.** However, our system is designed to run on distributed system and this architecture is not enough to understand how the system works in detail in large-scale clustered environments. We are going to parallelize both indexing and querying jobs, so that we can handle big scale in efficient manner. As the input to our system is a large number of PDF documents, we need to consider how to split them into smaller pieces and process them in parallel and how to combine the processed parts into single output file. MapReduce framework provides interface to tackle these types of problems.

Our ultimate design is affected by MapReduce programming model and ranking model described in [16]. Our system performs both indexing and querying tasks, but to make each part clearer, we describe indexing process and querying process separately. Indexing system parses PDF files first and then constructs index structures on parsed text. As it performs two different tasks, we split the indexing job itself to preprocessing and text-indexing steps. Next section describes these steps in detail and show the reasons of splitting indexing job.

3.2 Preprocessing and Text-indexing

We decided to split indexing process into two parts, *preprocessing* and *text-indexing* for two reasons. First, in MapReduce programming model map tasks are independent from each other and they do not share any information. But indexing requires global Document to DocumentID mappings of all documents, so that mapper can produce <term, docID> pairs from the documents it processed. This requires those mappings to be known in advance. Second reason is that we want to logically separate two different operations: PDF parsing from indexing. This allows us first to convert PDF into plain text and bind those text with document IDs, and then use these text files with only necessary information for indexing to analyze text, chop into tokens, normalize them into terms, create <term, posting-list> mappings. In this way our architecture becomes clearer and easier to implement.

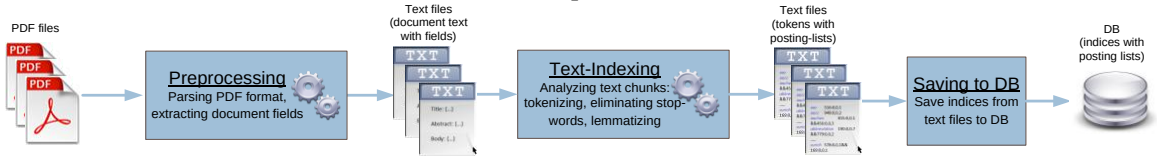

Fig. 2. Indexing process workflow

Even though we can see Indexing system and its interactions with other components in **Fig. 1**, **Fig. 2** gives clearer picture of how the indexing job is done in terms of job workflow in

distributed system. We designed Preprocessing and Text-indexing jobs as two consequent MapReduce operations, the output of first operation goes as input to the second operation. Preprocessing job parses PDF documents to text representation, text-indexing analyzes text chunks and create index structures, and finally index is saved to database. This additional step is necessary to avoid concurrency issues of databases, such as the case when several reducers attempt to insert data into database simultaneously. We use distributed database system in order to enable querying indices in parallel.

In the following part of this section, we describe preprocessing and text-indexing steps in detail.

A. Preprocessing

Preprocessing step is responsible for parsing scientific articles in PDF documents into text files and creating document structure given that PDF files usually do not contain any hints to recreate structure of scientific paper. We designed special parser algorithm, by assuming that scientific articles have some common layout which will help us to rebuild that structure.

We divide the document context of the scientific paper into three zones: *title*, *abstract* and *body*. This is necessary for two reasons: First, when we search for certain keywords in our system, search result should display titles of the documents rather than the whole document text. Also we can use paper abstract as a short description of that document in a search page. Second, we need those zones for ranking purpose. We score documents by assigning different weights for each zone. We consider that if keyword appears in title, it should have more relevance than if it appears in abstract or body. Title has more weight than abstract, and abstract has more weight than body text.

Given the PDF document we need to extract these zone texts separately. While extracting full text from PDF is not a big issue (as there exists a bunch of software libraries providing this functionality), extracting those zones separately is a challenging one. PDF structure does not contain any clues about the document structure and metadata. Even if the PDF document is created with providing the necessary metadata explicitly, we cannot fully rely on them, as filling this metadata information is not mandatory. Hence, we need some special techniques to extract the zones, without relying on PDF's own document metadata.

What we can do in this case is parsing zones using keyword texts. However, using only keyword may not be enough, so we use keyword in combination with font information in the PDF. Our parser algorithm assumes that the scientific paper in PDF has the following characteristics:

- Title text has the biggest font in the page where it appears;
- PDF contains the word "Abstract";
- Title of the first section of the paper is always "Introduction" OR "Motivation"
- All the papers has these keywords "Abstract" AND "Introduction"/"Motivation" AND "References"

Parsing PDFs from various sources is a painful task, as there are a numerous of scientific paper formats. Widely used two-column scientific paper format is shown in Fig. 3. However, there exist lots of other formats besides that. Things get worse when considering the papers published several decades ago, which are encoded into PDF in different ways than the recent ones. Building a single parser which correctly parses every possible scientific paper format is just practically impossible. Besides that, we depend on third-party parsing library, which may have some errors in parsing with particular documents. Hence, we should tolerate some errors in the parsing process.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

Fig. 3. Snapshot of scientific paper. Keywords and font information assist in extracting document fields: title, abstract and body text

The line of the text which font size is biggest in the first page is parsed as title. For extracting abstract, first, we locate the keyword "Abstract". Then, we parse line by line after this keyword until the font characteristics (size, weight, italic) change in the next line. We should note that we allow the paper to be indexed even if it does not contain keyword "Abstract". Because, we consider that tolerating the requirement of having description (abstract is used as description of a paper) is more important than excluding one more paper from indices. In that case, abstract text is simply assigned as empty string. Body text is parsed by locating keyword "Introduction" or "Motivation" and extracting the whole text after this keyword.

In Preprocessing MapReduce operation we have only map phase. In this case, the outputs of map tasks are written directly to distributed file system which is common storage type for most MapReduce implementations. Map tasks process <docID, pdf_byte_contents> key-value pairs as input. They read each file and parse the pdf_byte_contents into text in parallel as described previously, that is, it extracts three text chunks (title, abstract and body). Output of map task is <docID, contents_array> pair, where *contents_array* is the array consisting of three text elements, each corresponding to document fields (title, abstract and body).

B. Text-indexing

After scientific article PDF files are parsed into proper text structure, we can do actual indexing. One of the most common index implementations used in search engines is *inverted index*. Index construction is a memory intensive operation: while the documents are scanned, terms are collected and kept in memory until finally they are flushed to disk. In single node, memory limitation problem is solved by partially flushing postings to disk when memory is full and finally merging them to make a single posting-list. This indexing algorithm with single node is called *single-pass in-memory* [16].

However, large collection of documents cannot be indexed efficiently with single node. It may take intolerably long time to index such big collections. Distributed cluster of nodes is a proper place to perform large scale indexing jobs. MapReduce framework is especially suitable for inverted index construction job, where terms from each document are usually used

as key, and the keys are sorted and grouped by the framework itself and finally posting-lists for each term are collected in reduce phase.

There exist several distributed indexing schemes using MapReduce framework. Conceptually all of them perform the same thing, that is, finally outputting posting-lists from collection of documents. However, they differ by the way these postings are created, by the structure of posting-lists which need to be used later in querying and by performance as well. In next chapter, we introduce several indexing schemes and propose ours which is suitable for our domain.

3.3 Distributed indexing schemes with MapReduce

In this section, first we describe the distributed indexing scheme from original MapReduce paper. Then we show improvements to that scheme and describe two of our propose schemes. And finally we introduce indexing scheme from J. Lin *et al.* [8].

A. Basic scheme

Dean and Ghemawat in original MapReduce paper [2] shows inverted index as one of MapReduce usage examples. In their scheme each call to map function analyses document and emits <word, documentID> for each analyzed word. Reduce function collects all documentIDs for each term and emits <word, list(documentID)> pairs. They simplified their algorithm to give general information of how MapReduce can be used for inverted index, however, details matter, such as where term frequencies are collected, what structure is used as key-value pairs in map outputs and overall, details affect performance of indexing. Even though we believe that within Google their indexing scheme is much efficient than they described in their paper, we use their described scheme in our comparison. We adapted their algorithm to our document structure and pseudo-code is given in **Fig. 4**. We refer to this scheme as *basic*.

Distributed Indexing algorithm by Dean&Ghemawat

Map() function	***Reduce()*** function
Input: list of documents with identifiers (<*docID*, *contents_array*>)	*Input:* list of <*term*, (*docID, docField*)> pairs grouped by term
Output: list of <*term*, (*docID, docField*)> pairs for each term in each field of document	*Output:* list of <*term*, *posting-list*> pairs

foreach *docField_contents* **in** *contents_array*
  foreach *token* **in** *docField_contents*
  term ← normalize(*token*)
  emit(*term*, (*docID, docField*))
	Posting-list[] ← {}
	foreach (*docID, docField)* **in** list(*docID, docField*)
	  if (*docID* exists in *Posting-list*)
	  increment *docField.Frequency* of *docID* in *Posting-list*
	  else
	  add *docID* with *docField.Frequency*←1 to *Posting-list*
	emit(*term*, *Posting-list*)

Fig. 4. Pseudo-code for *basic* indexing scheme

In this algorithm, output of previous preprocessing job <docID, contents_array> is submitted to map task, and map function process each document field in contents_array and analyze the text to produce <term, (docID, docField)> tuples. *normalize()* function performs removing punctuation symbols, lowercasing, stemming or lemmatizing and removing the term if it appears in stop-words list. Here, in our adapted algorithm docField (which is enumeration of three values) is emitted together with docID, to indicate whether the given term appears in Title, Abstract or Body of the document. Reduce function process (docID, docField) pairs for each term, and produces <term, posting-list> pairs by aggregating occurences of each term in each docID and each of docFields. Posting-list has the following structure: array of (docID, title_freq, abstract_freq, body_freq) tuples. For example, <term, posting-list> key-value pairs can appear as "*throughput 578:1,2,46 && 601:0,0,55 && 1231:2,2,34 && ...*". In this format,

"*578:1,2,46*" notation means that the term "*throughput*" appears in document which ID equals to *578* and it appears *1* time in title, *2* times in abstract and *46* times in body text.

B. *Updated scheme*

In *basic* indexing scheme, frequencies for each document fields are counted on Reduce side. In our proposed indexing scheme, we move the computation of frequencies from Reduce side to Map side. That is, Map function instead of emitting <term, (docID, docField)> tuples aggregates the terms occurred within the same document and emits <term, posting> pairs.This way fewer key-value pairs will be emitted in map side, which means the shuffling phase should take less time, because less data will be copied to corresponding reducers. *posting* in that context is (docID, title_freq, abstract_freq, body_freq) tuple. Reduce function is simple; it only collects individual postings and aggregates them to form posting-list for each term. We show our proposed indexing scheme, which we refer as *updated*, in **Fig. 5**.

Proposed Indexing algorithm

Map() function
Input: list of documents with identifiers (<*docID*, *contents_array*>)
Output: list of <*term*, *posting*> pairs for each term in each field of document

Term_posting[] ← {}
foreach *docField_contents* **in** *contents_array*
  foreach *token* **in** *docField_contents*
    term ← normalize(*token*)
    if (*term* exists in *Term-posting*)
      increment *docField.Frequency* of *term* in *Term-posting*
    else
      add *term* with *docField.Frequency*←1 to *Term-posting*
foreach (*term*, *posting*) **in** *Term_posting*
  emit(*term*, *posting*)

Reduce() function
Input: list of <*term*, *posting*> pairs grouped by term
Output: list of <*term*, *posting-list*> pairs

Posting-list[] ← {}
foreach *posting* **in** list(*posting*)
  add *posting* to *Posting-list*
emit(*term*, *Posting-list*)

Fig. 5. Pseudo-code for *updated* indexing scheme

C. *Updated+combiner scheme*

There is still some space of improvement of our *updated* scheme. Before moving <term, posting> key-value pairs from mapper to reducer, we can do local reducing after map function is completed. As we have repetitive intermediate keys in map output, we can aggregate them before sending to reducer. This way we can transfer even less amount of data than in previous scheme. The aggregating function is referred as Combiner in MapReduce terminology [2] [17]. Compared to *updated* scheme, Map output is <term, posting-list> pairs, where posting-list contains only single posting. Reduce() function only aggregates posting-lists into single posting-list and it is used for both Combiner and Reducer functionalities. We refer to this scheme as *updated+combiner*.

D. *Ivory scheme*

Next indexing scheme is named Ivory proposed by J. Lin et al. [8]. Its main difference from other indexing schemes is that it is designed to produce *document-sorted* posting-lists. MapReduce framework does not make any sorting guarantees about the intermediate values (but it sorts intermediate keys). Hence, instead of sorting document ids in-memory in reduce tasks they incorporate documentIds to key structure to utilize sorting functionality of MapReduce framework itself. Map function instead of <term, (docID, termFrequency)> tuples outputs <(term, docID), term_frequency> tuples, where (term, docID) became key instead of docID. This way MapReduce framework guarantees that docIDs arrive to reducer in

sorted order (they implemented custom partitioner and custom comparator to achieve that behaviour). We refer to this scheme as *ivory*, and the pseudo-code adapted to our case is shown in **Fig. 6**.

We provide comparison of the mentioned distributed indexing schemes' performances in Evaluation Section.

Ivory Indexing algorithm

```
Map() function
Input: list of documents with identifiers
   (<docID, contents_array>)
Output: list of <(term, docID), FieldFrequency>
   pairs for each term in each field of document

Term_posting[] ← {}
foreach docField_contents in contents_array
  foreach token in docField_contents
    term ← normalize(token)
    if (term exists in Term-posting)
      increment docField.Frequency of term
       in Term-posting
    else
      add term with docField.Frequency←1
       to Term-posting
foreach (term, posting) in Term_posting
   emit((term, docID), FieldFrequency)
```

```
Initialize() function
   term_prev ← ""
   Posting-list[] ← {}

Reduce() function
Input: list of <(term, docID), FieldFrequency>
   pairs grouped by term
Output: list of <term, posting-list> pairs

   if (term_prev is not empty and
     term ≠ term_previous)
       emit(term, Posting-list)
       Posting-list ← {}
   add (docID, FieldFrequency) to Posting-List
   term_prev ← term

Close() function
   emit(term, Posting-list)
```

Fig. 6. Pseudo-code for *ivory* indexing scheme

3.4 Querying system design

Querying system enables querying on given keywords and returning list of documents ranked with relevance to the keywords. We provide so called *free-text queries* to the users. Users can provide search words without providing some Boolean operators as AND or OR, and also, without providing which part of the document to locate given keywords. When user submits search text, it is analyzed to get list of terms as described in Section 3.1. Querying system looks up posting-lists for each term from Index database and obtains statistics of this term in each document fields. Then it calculates score for each document, ranks them by the scores and occurrences and returns the ones with highest scores.

We decided to use database system for holding indices because random-access in a filesystem usually has longer latencies. Traditional relational database management systems (RDBMS) could be used, however, in multi-user environment RDBMS could be bottleneck because of using only single node. We are targeting large-scale indexing and to parallelize querying we need to use some parallel techniques. MapReduce is not suitable for querying, because only job configuration and setup may take several seconds, and real-time access is necessary for querying to enable positive user experience. Hence we used HBase [18], distributed database system modeled after Google's BigTable [19], where we can query indices in term-partitioned nodes (in HBase terminology they are called regionservers).

A. Ranking model used

There exist number of ways in scoring documents relative to given term. We used one of the weighting schemes described in [16], called *TF-IDF* weighting. The simplest way is to count number of occurences of the term *t* in the document *d*, also known as *term frequency* (denoted as $tf_{t,d}$). In our case, we have three fields and weights are different for each of them. $tf_{t,d}$ is calculated with the following formula:

$$tf_{t,d} = title\_f_{t,d} \cdot w_t + abstract\_f_{t,d} \cdot w_a + body\_f_{t,d} \cdot w_b \quad (1)$$

where $title\_f_{t,d}$, $abstract\_f_{t,d}$, $body\_f_{t,d}$ are frequency of term t in the title, abstract, body of the document d correspondingly, and w_t, w_a, w_b are corresponding field weights.

However, some terms may appear too frequently over the whole collection which devaluates its importance and relevance. For example, collection of documents on nuclear physics possibly contains the word 'atom' in almost every document. Hence, to alleviate this effect inverse document frequence (denoted as idf_t) is used:

$$idf_{t,d} = log(N/df_t) \quad (2)$$

where N is number of documents in the collection, and df_t is number of documents in the collection that contain a term t.

And finally, score is given to each document d for the query q which made of a set of terms t is calculated by:

$$score(q, d) = \sum_{t \in q} (tf_{t,d} \cdot idf_{t,d}) \quad (3)$$

where $tf_{t,d}$ is calculated by (1) and $idf_{t,d}$ by (2).

We order the results first by occurences of the search terms, then by scores. For example, if we search '*cloud computing*', the results are obtained in the following way: First, the list of documents which contain both terms '*cloud*' and '*computing*' are sorted by their scores separately and added to results. Then other documents containing either '*cloud*' or '*computing*' are sorted separately as well and appended to the result list. Hence, a document which contains both terms will be ranked higher than the one containing only one of them, even if the score of document containing only one term is higher than the score of the one containing both. We consider that documents containing all search terms are more relevant than the ones which contain only part of search terms, regardless from their term frequencies.

4. SciPDFindexer implementation

We fully implemented SciPDFindexer system as described in System Design section (Section 3). In this section we provide some implementation issues related with preprocessing and text-indexing, and also we show querying implementation with distributed database system.

4.1 Indexing system implementation

We used Hadoop MapReduce [5] to parallelize both parsing and indexing processes. The input files to preprocessing MapReduce job are a large number of PDF files. One of the problems we encountered during this stage is that we had to process large number of binary files separately. In simplest case, we could assign one PDF file per one map task, but that would be very inefficient, taking into account map initialization overhead. Instead, we used special serialization format provided by Hadoop – *SequenceFile* [17] to package large number of small PDF files into small number of large SequenceFiles. By this, we can ensure that single map proceeds with approximately single HDFS block (in our case - 64Mb) amount of PDF files, which greatly reduces required total number of map tasks, which in turn reduces execution time.

Before parallelizing parsing process, we need to obtain global Document to DocumentID mappings, because in MapReduce framework no context is shared among map tasks. We cannot assign DocumentID to processed documents in each map task separately due to possible DocumentID collisions with documents processed by other map tasks. In our case DocumentID is unsigned integer incremented with each additional document.

We implemented small Java application which performs both of the above tasks. It packs a big amount of PDF files into small number of SequenceFiles and outputs global Document-DocumentID mappings which will be used in later processes. Besides that, instead of first producing SequenceFiles then moving them to HDFS, packing application accumulates contents of several PDF files located locally into memory and flushes

SequenceFiles directly into HDFS.

Preprocessing step utilizes MapReduce only to parallelize parsing process. Hence only only map phase is used. We used PDFTextStream library [20] to extract text from PDFs. PDFTextStream library provides API for detailed PDF content extraction, such as page-level access, hierarchical document model (Page, Block, Line, TextUnit), and specific information as font, line height. We used their API to extract title, abstract and body text.

In text-indexing step, text contents are analyzed in the following way: First, text contents are chopped into tokens. Then, tokens are filtered to remove ones, which carry little semantics taken separately, also known as *stop-words*. We accumulated common stop-words in English, such as articles, prepositions, pronouns, numericals, etc. After filtering, tokens are normalized to get basic morphological forms of the words by using English Lemmatiser library from Dragon toolkit [21].

4.2 Querying system implementation

We implemented querying part in HBase [18], non-relational distributed database system used for real-time read/write access. Table schema design in HBase is cardinally different from traditional RDBMS. We specify only column families, any columns related with given column family can be added dynamically at any time. As table rows are sorted, it is important to design row keys of the tables to fit specific application requirements.

In our case, we have two tables: *document* and *term*. In 'document' table, row key is document ID, and it has fixed number of columns. It has one column family 'info' and three column qualifiers belonging to it: 'title', 'abstract' and 'filename'. We do not store 'body' part of the document, because users can access the body from the actual document.

'term' table is designed to store the posting-lists of terms. 'term' table's design is totally different from typical table schemas in RDBMS, because there can be any number of columns in our case. We use string representation of the term as a row key. We have two column families: 'info' and 'postingscore'. Column family 'info' contains only 'df' column, which represents document frequency of the term, that is, how many documents contain this term. And 'postingscore' column family may contain as much columns as the number of documents in the collection. The format of each column is like this: *postingscore:<doc-id>*, that is, *postingscore:1*, *postingscore:2*, …, *postingscore:i*, … We can imagine 'term' table as a big sparse matrix with dimensions (*number_of_total_terms* x *number_of_total_documents*). Each column *postingscore:<doc-id>* will carry $tf_{t,d}$ value from Equation (1). Instead of storing three values seperately (title frequency, abstract frequency, bodytext frequency) we pre-calculate scores with given weights for each zones. This is done to improve performance, as instead of calculating $tf_{t,d}$ in each querying, it is calculated once when the data is inserted to the DB, and then only that score is used to determine rankings of documents. You can see these table structures in HBase in **Fig. 7**.

Document

row	column family
	info:
<doc-id>	info: title info: abstract info: filename

Term

row	column families	
	info:	*postingscore:*
<term>	info: df	postingscore: <doc-id>

Fig. 7. Table structure with HBase

We created simple user interface for searching document base and the results show the documents in the order of relevance score we calculated in the previous step. For each such document we display title of the document as a hyperlink to that document, and abstract as a description of the document. **Fig. 8** shows the screenshot of SciPDFindexer QueryParser GUI.

Fig. 8. SciPDFindexer Search UI snapshot

5. Evaluation and Analysis

In this section we evaluate the performance of the indexing and querying system on Hadoop environment.

5.1 Experiment Environment

Our cluster consists of five server nodes connected via Gigabit Ethernet: one master node and four slave nodes. Master node has more processing power than the other four identical slave nodes. The parameters of the system are given in **Table 1**.

Table 1. Evaluation system parameters

Hardware and software specifications:		
	Master node	*Slave nodes*
CPU type	Intel Xeon	Intel Xeon
CPU clock	1.6 GHz	2.4 GHz
Number of cores	8	4
Memory	8Gb	4Gb
Network speed	1 Gbps local, 100 Mbps internet	
OS	Fedora 13 64-bit (Linux kernel version: 2.6.33)	
Java Version	Java 2 platform, Standard Edition (1.6.0_18)	

Cloudera distribution of Hadoop (CDH2) [22] (Hadoop version 0.20.1) which includes both MapReduce and HDFS software is installed on all the nodes. Also HBase software (version 0.90.2) is installed on the same set of nodes. All four slave nodes run *HDFS DataNode, TaskTracker* and *HBase Regionserver* processes and Master node runs *HDFS NameNode, JobTracker* and *HBase Master* processes. We set the replication factor to 4 to

ensure that all the map tasks are data-local. We use default HDFS block size – 64Mb.

5.2 Experiment Setup

In the Experiment Setup sub-section we provide Experiment Objectives first and then the way we gathered input data.

A. Experiment Objectives

We want to evaluate our SciPDFindexer system by addressing three research questions:

1) Can our proposed indexing scheme perform better compared to other alternative schemes?
2) How different MapReduce parameters affect indexing throughput?
3) Can we achieve reasonable response time for the querying large-scale indices?

For the first part, we want to compare performance of distributed indexing schemes mentioned in Section 3.3. For each number of nodes case and for all the indexing algorithms we evaluate throughput of indexing process *T(m)* defined as

$$T(m) = doc\_coll\_size / time\_to\_complete \quad (4)$$

where *doc_coll_size* is the size of document collection processed in parallel in Mb in textual format after being parsed from PDF, *time_to_complete* is the time taken to complete the process in seconds, and *m* is number of nodes. As a document collection size we take textual format of the documents after being parsed, as the size of text will directly affect the indexing performance. We provide comparisons of all four algorithms and reasons why one is performing better than another.

For the second part, we want to evaluate such MapReduce parameters as number of maps and number of reducers. If we have too many maps (reducers), it will result in additional overhead of map (reducer) initialization, setup and then shutting down. In Hadoop framework, we cannot explicitly specify total number of maps, as total number of maps is determined by input size and the way how the input is split. Instead, we investigate the impact of number of concurrent maps per node to indexing performance. But for reduce tasks, we can specify total number of reduces, hence we investigate the impact of total number of reducers to indexing throughput. We will conduct experiments by varying number of concurrent maps and total number of reducers to see their impact to indexing throughput.

And in the third part, we will measure response time of querying. Providing small response time is crucial from the perspective of positive user experience. As we are dealing with large scale of documents, the database holding the indices and posting-lists could become bottleneck of the system. Using more query words means retrieving posting-lists for each of the query words, and merging them together to calculate final scores. We will evaluate how much the increase in query words affects response time.

As a response time, we are interested only in server processing time, that is, the time server looks up indices database and calculates scores for documents, because other parts of response time as sending HTTP request and receiving HTML page depends on network bandwidth between user and server. To evaluate the response time as indicated above, we produced a set of queries containing from 1 to 10 different keywords related to Computer Science. We produced five different queries for each number of keywords and we ran each of five different queries 5 times, estimate response time and take average of them. Then we average the resulted averages of each query word/phrase in each category.

B. Obtaining input data

For our experiments we need a large number of scientific papers in PDF as an input. To get this amount of input we built a specific crawler application in Java which crawls articles in PDF from ACM Digital Library [23]. Our school has been granted access to most of full-text articles in PDF in this library.

Our crawler works in the following way: First we make a list of keywords related to

Computer Science. Crawler visits ACM Digital Library's web-site and puts the first keyword in our list to search textbox and click the "Search" button. Then it finds links to article PDF files by XPath query, downloads them and then proceeds to the next result page. When it reaches the end of result page, it gets next keyword from our list and performs above-mentioned procedures again. File name and URL of the downloaded articles are written to log file which is used to filter out duplicates when the crawling is finished. We used *HtmlUnit* API in Java [24] to emulate browsing web-sites. With this crawler we crawled 27,822 PDF articles which totals 20.5 Gb.

5.3 Experiment Results

Here we will provide the results of parsing PDF files to get the textual format for indexing. After that we show results of three experiments mentioned in Experiment Objectives section. Each case of the experiment is conducted 10 times and the result is averaged.

A. Parsing PDF files into textual format

This stage is also known as Preprocessing stage in previous sections. As mentioned in Section 3.2, in this stage scientific articles in PDF are parsed and text files with predefined structure are extracted from them.

We packed 27,822 PDF files obtained from crawling into 21 files in SequenceFile format of around 1Gb size each. As a result, instead of having 27,822 maps, we had 330 maps (i.e. 330 HDFS blocks, 20.5Gb/64Mb≈329.4). Not all of the PDF files were parsed, because some of the files were not in scientific research paper format and some of them could not be parsed because of specifics of third-party parsing library we used (PDFTextStream). Out of 27,822 files 21,256 (76.4%) were correctly parsed. Most of non-parsed articles come from very old articles with specific encodings which our parsing library could not parse. Total parsed text size was 883Mb.

B. Comparison of indexing schemes at larger scale

In this part of the experiment we compare the throughput of our proposed indexing scheme with alternative schemes by varying number of nodes. In each node case, the number of reducers was set equal to number of nodes.

We can see the result of the experiment in **Fig. 9**. The order of indexing schemes based on their performance from top to bottom is the following: *updated+combiner*, *updated*, *ivory*, *basic*. In MapReduce framework, when map phase is finished, it will have partitioned data chunks in local disks, the number of which corresponds to the number of reduce tasks. TaskTrackers running reduce tasks copy their partitions from each map task outputs from each node to their local disk, known as *copy phase*, before proceeding to sorting and reducing phases. This copy phase has considerable impact to running time, and consequently to throughput. In *basic* scheme, map phase emits too many key-value pairs, one for each analyzed term in each document. This results in lowest performance for that scheme.

updated and *ivory* schemes calculate frequencies of term in each document field in map phase and as a result they emit only unique terms per document. Therefore, they emit fewer key-value pairs than *basic* which is reflected in their throughput.

MapReduce framework after sorting map outputs groups values with the same key and submits to reducer. In *ivory* scheme no grouping happens, because all the keys map tasks outputted are unique (as a key (term, docID) pair is used). They do not take advantage of grouping by terms in order to achieve another goal of getting document-sorted posting-lists. Hence, reduce phase takes more time in *ivory* compared to *updated*, because *ivory* processes each key-value pairs separately, while in *updated* values associated with the same key are grouped together and processed in once.

Our proposed *updated+combiner* scheme utilizes combiner (also known as local reducer) to reduce amount of map output. After map phase outputs key-value pairs and before sending them to reducer, combiner merges map outputs with the same key. As a result, only unique

terms per a set of documents which a single map task processed are copied to reducer. This is one more step forward in reducing the amount of data moved from mappers to reducers.

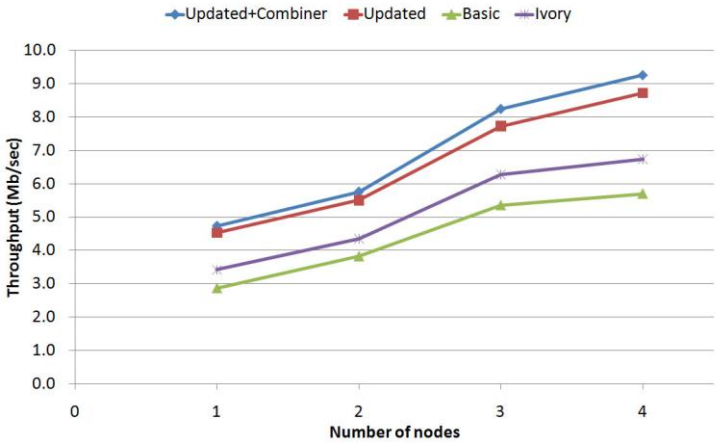

Fig. 9. Throughput comparison per number of nodes

In this experiment we showed that our proposed schemes *updated* and *updated+combiner* performs better than other two schemes. Even though *ivory* performed not as good as ours, main reason is that they have specific requirement for its output, that is, posting-lists should be document-sorted while we do not have such requirement. We can say that our scheme works well for our domain and our case.

C. Impact of MapReduce parameters to indexing performance

In this part, we show experiments with investigating impact of MapReduce parameters, such as number of maps and reduces to the throughput of indexing job. Preprocessing stage produced 330 output files, and we will vary number of concurrent maps per node to process total 330 map tasks.

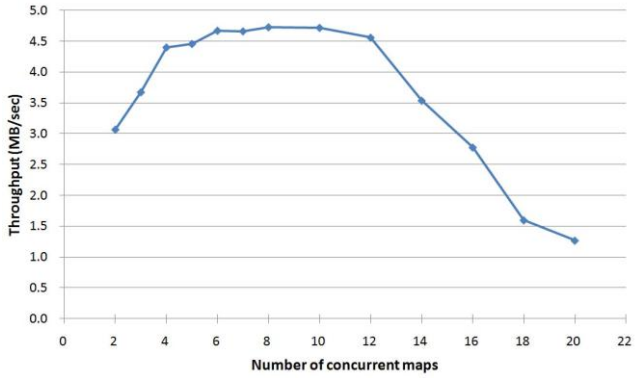

Fig. 10. Throughput per number of concurrent maps per node.

Fig. 10 shows that throughput is sharply increasing when we range number of concurrent maps from 2 to 4. This happens due to the fact that each node has 4 CPU cores and until we use 4 concurrent maps, some of the cores are underutilized. From 4 to 6, we can observe slight increase in throughput, which means that we still can gain from parallelization. From 6 to 12 the throughput is steady and still effectively parallelized and in this range the highest throughput is achieved. From 12 throughput starts decreasing. This happens because too many map processes start contending for the same CPU core. Besides CPU, memory also becomes bottleneck when there are many maps running in parallel resulting in many map task failures. Contention for both CPU and memory causes many map task failures, which have to be rerun, which in turn increases completion time. Result of this experiment shows that there exists a range of values for concurrent maps number when the highest throughput is achieved.

Next, we conduct experiment evaluating throughput by varying number of reduces. Number of reduces determines how many partitions each map task will produce, how many

files will be copied from map side to reduce side and how many files will be outputted finally. In **Fig. 11** we can see results of this experiment. It shows that throughput is not very sensitive to total number of reducers. This can be explained by the fact that reducer in our proposed scheme does only simple job: aggregating postings related to the same term. Hence, its impact to total indexing execution time, correspondingly to throughput is minimal. When we increase number of total reducers from 2 to 10, throughput is gradually increasing, which shows that CPU cores are better utilized. Then it is stable for some range and gradually decreasing. The maximum throughput is achieved with total 14 reducers (from 3 to 4 reducers per node) giving 9.6 Mb/sec throughput.

We can improve the throughput of indexing by packaging 330 small input files into several large SequenceFile formats in a similar way as we did in Preprocessing stage. By this way we can reduce total number of maps from 330 to 15, each map processing the amount of text roughly equal to HDFS block size. As a result, we can improve average indexing time of the job run in 4 nodes with 8 concurrent maps and total 8 reduces from 195 sec to 93 sec. We used packaged input in indexing schemes comparison and throughput per number of total reducers experiments.

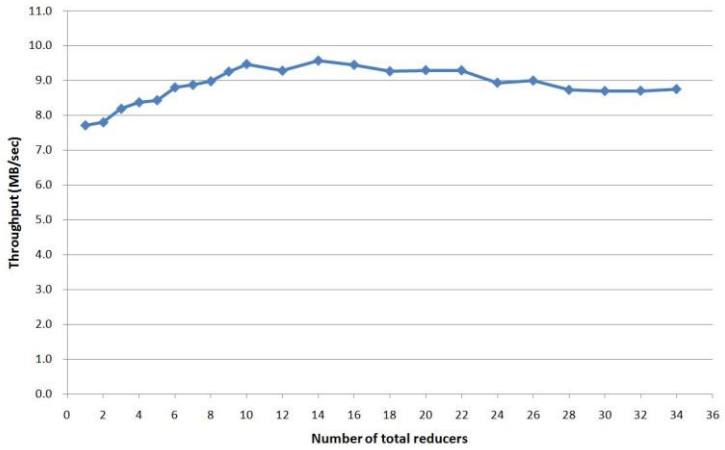

Fig. 11. Throughput per number of total reducers.

D. Evaluating response time of querying indices

The output of indexing job is deserialized and loaded into the HBase database. As a result, 943,626 rows were loaded into 'term' table. We can see the results of this experiment in **Table 2**.

Table 2. Querying response time

Number of query words in single query	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Response time (in sec)	0.04	0.10	0.18	0.30	0.28	0.33	0.60	0.56	0.55	0.64

When we submit several query words, the posting lists are retrieved for each of them and then they are merged by aggregating scores for each document. Then the total set of documents with aggregate scores is sorted first by occurrence then by scores. As we can see, generally, more query words involve more computations in merging, sorting, therefore taking more time to respond to user. Even though the response time is increasing while we are providing more query words, the absolute maximum value of response time is less than 1 second. The longest response time taken in this experiment is 636 milliseconds and with up to 5 query words 298 milliseconds which is quite fast response time for large scale systems. In this experiment, we showed that our querying system provides sub-second response time for various length of queries.

6. Conclusion and Future Works

In this work we addressed the information retrieval problem of scientific articles and provided our solution for that. We proposed indexing scheme, which can efficiently index a large corpus of scientific articles in parallel with MapReduce framework. We designed and implemented full IR system for scientific articles in PDF - *SciPDFindexer* which uses the mentioned indexing scheme. Our system is designed to perform indexing job and run query both in parallel using distributed set of nodes to deal with large scale.

We believe that IR of scientific articles differs from general IR, hence, it needs to be treated differently. General IR mostly deals with text or web-sites, but in our case we need to extract text content from articles in PDF and not only extracting it but recreating structure of scientific paper, such as title, abstract, body. And for each of these document fields we assigned different weights in ranking which we believe improves relevancy of query results.

Our indexing system consists of two parts: preprocessing (or parsing) and text-indexing, and this splitting simplifies our task and logically separate two different jobs. Both of these jobs are batch type jobs and we use MapReduce programming model to execute both of them in parallel. MapReduce model provides convenient interface for divide-and-conquer type problems and it is especially suitable for batch jobs. Two parts of indexing system run as two consecutive MapReduce jobs, output of first going as input to second. For preprocessing job, we designed special parsing algorithm to parse scientific articles from raw PDF content. This allows extracting title, abstract, and body fields of an article which is used later for ranking and in Search web-UI.

There exist several indexing schemes using MapReduce programming model. While all of them conceptually perform the same job, that is, creating inverted index, they differ by map and reduce key-value structures, how these key-values are processed and how much they utilize underlying MapReduce framework functionalities. We proposed our distributed indexing scheme along with alternative schemes and showed how different is ours from other implementations.

We used TF-IDF weighting scheme in our ranking model which scores documents based on term frequency, document frequency and which zone the term appeared. While smarter ranking model could be researched and used, we left it as future work. Traditional RDBMS except specialized complex parallel database systems does not take advantage of parallelism in distributed nodes. We used HBase to hold inverted index, because it provides real-time access to database which is partitioned in a cluster of nodes. Response time of querying is important from user's point, and HBase allows querying distributed nodes in parallel.

We evaluated our system by conducting several experiments. First, we compared our indexing scheme with alternative ones and showed that our schemes outperform others in terms of indexing throughput. Second, we investigated the impact of MapReduce parameters into the indexing throughput. And lastly, we showed that our querying system gives sub-second response time for various length of queries.

Our contributions include first, distributed indexing scheme, second, design and implementation of SciPDFindexer system, and third, special parsing algorithm to parse scientific papers in PDF and extract their structure. Our system can be used as a search engine for the repository of scientific papers in research institutes, in universities or in any organization which deals with large collection of scientific papers.

As a future work, we intend to extend our system to support dynamic indexing, such as when new documents are regularly added to the collection and indices need to be up-to-date. This poses some new challenges: how to manage new indices and how to merge them with old ones.

Acknowledgements

This research was jointly supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency" (NIPA-2011-(C1090-1121-0011)) and R&D program of MKE/KEIT (10024119). We would like to thank Minkyung Kim and Hayeon Jung for helping us in the initial phase of this research. Also we would like to thank *Snowtide Informatics* company for providing us free one-year license for the PDF text extraction library *PDFTextStream* for academic use.

References

[1] WorldWideWebSize.com. http://www.worldwidewebsize.com/ [cited in 2011]
[2] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, Dec. 2004.
[3] E. Lai, Google claims MapReduce sets data-sorting record, topping Yahoo, conventional databases, ComputerWorld, Nov 28, 2008. Available at: http://www.computerworld.com/s/article/9121278/Google_claims_MapReduce_sets_data_sorting_record_topping_Yahoo_conventional_databases [cited in 2011].
[4] Text REtrieval Conference (TREC) Data. http://trec.nist.gov/data.html [cited in 2011]
[5] Hadoop MapReduce. http://hadoop.apache.org/mapreduce [cited in 2011].
[6] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. Proceedings of the 13th International Symposium on High Performance Computer Architecture, 2007.
[7] Hadoop Distributed File System. http://hadoop.apache.org/hdfs/ [cited in 2011].
[8] J. Lin, D. Metzler, T. Elsayed, L.Wang. Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search. Proceedings of the 2009 Text REtrieval Conference (TREC 2009), 2010.
[9] D. Cutting. Nutch: an Open-Source Platform for Web Search. Workshop on Open Source Web Information Retrieval (OSWIR), 2005.
[10] R. McCreadie, C. Macdonald, I. Ounis. Comparing Distributed Indexing: To MapReduce or Not? 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009.
[11] S. Lawrence, C. Lee Giles, K. Bollacker. Digital Libraries and Autonomous Citation Indexing. IEEE Computer, vol. 32, no. 6, 1999.
[12] H. Li, I. Councill, W. Lee, C. Lee Giles. CiteSeerx: an architecture and web service design for an academic document search engine. Proceedings of the 15th International Conference on World Wide Web 2006 (WWW '06).
[13] T. Briscoe et al. Intelligent Information Access from Scientific Papers. The Information Retrieval Series, Vol. 29, Part 5, 2011.
[14] B. Goetz. The Lucene search engine: Powerful, flexible, and free. Javaworld. http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene.html [cited in 2011], 2002.
[15] M. Karamuftuoglu, S. Jones, S. Robertson, F. Venuti and X. Wang. Challenges posed by web-based retrieval of scientific papers: Okapi participation in TIPS. Journal of Information Science, vol. 28 no. 1, Feb 2002.
[16] C. D. Manning, P. Raghavan, H. Schütze. An Introduction to Information Retrieval, Cambridge University Press, 2008.
[17] T. White. Hadoop: The Definitive Guide. Published by O'Reilly Media, 2009.
[18] Apache HBase. http://hbase.apache.org/ [cited in 2011].
[19] F. Chang et al. BigTable: A Distributed Storage System for Structured Data. Seventh Symposium on Operating System Design and Implementation, 2006.
[20] PDFTextStream. PDF Text Extraction library for Java, .NET, Python. http://snowtide.com/PDFTextStream [cited in 2011].
[21] Zhou, X., Zhang, X., and Hu, X., "Dragon Toolkit: Incorporating Auto-learned Semantic Knowledge into Large-Scale Text Retrieval and Mining," In proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), October, 2007.
[22] Cloudera Distribution of Hadoop. http://www.cloudera.com/ [cited in 2011].
[23] ACM Digital Library. http://portal.acm.org/ [cited in 2011].
[24] HtmlUnit API. http://htmlunit.sourceforge.net/ [cited in 2011].

[25] C. Lee Giles, K. D. Bollacker, S. Lawrence, CiteSeer: An Automatic Citation Indexing System, Proceedings of the 3rd ACM Conference on Digital Libraries, New York, pp. 89-98, 1998.
[26] M. Thelwall, Extracting accurate and complete results from search engines – Case Study Windows Live, Journal of the American Society for Information Science and Technology, 59(1), pp. 38-50, 2008

***Biographies (Text)**

Aziz Murtazaev (82-10-5114-1323 muraziz@ajou.ac.kr)

Murtazaev received B.A. in Economics from the National University of Uzbekistan in 2007 and M.S. in Computer Engineering in Ajou University, South Korea in August, 2011. He is currently working for Samsung Electronics. After his B.A., he worked 2 years for Muranosoft, outsourcing software solution provider, as a software engineer. His research interests include distributed systems, cloud computing, information retrieval and large-scale software system.

Geoffrey Charles Fox (1-812-219-4643 gcf@indiana.edu)

Fox received a Ph.D. in Theoretical Physics from Cambridge University and is now professor of Computer Science, Informatics, and Physics at Indiana University. He is director of the Community Grids Laboratory of the Pervasive Technology Laboratories at Indiana University. He previously held positions at Caltech, Syracuse University and Florida State University. He has published over 550 papers in physics and computer science and been a major author on four books. Fox has worked in a variety of applied computer science fields with his work on computational physics evolving into contributions to parallel computing and now to Grid systems. He has worked on the computing issues in several application areas – currently focusing on Earthquake Science.

Sangyoon Oh (82-10-3129-7022 syoh@ajou.ac.kr)

Oh received Ph.D. in Computer Science Department from Indiana University at Bloomington, U.S.A. He is an associate professor of School of Information and Computer Engineering at Ajou University, South Korea. Before joining Ajou University, he worked for SK Telecom, South Korea. His main research interest is in the design and development of web based large scale software systems including Cloud Computing, Virtualization Technology, Parallel Computing, Grid Computing and Service Oriented Architecture (SOA).

murtazaev_biophoto.gif

Click here to download high resolution image

fox_biophoto.gif

Click here to download high resolution image

oh_biophoto.gif

Click here to download high resolution image