Scalable Distributed Time-Travel Text Search

Masterarbeit im Fach Informatik
Master’s Thesis in Computer Science
von / by

Thaer Samar

angefertigt unter der Leitung von / supervised by

Dr. Klaus Berberich

begutachtet von / reviewers

Dr. Klaus Berberich
Prof. Dr. Gerhard Weikum

December 2011
Statement

I, Thaer Samar, hereby confirm that this thesis is my own work and that I have documented all sources used.
Saarbrücken, December 01, 2011

______________________________
Signature (Thaer Samar)
Acknowledgement

First and foremost, I would like to especially thank my supervisor Dr. Klaus Berberich, for his guidance, invaluable ideas and technical support.

I would also like to thank prof. Weikum, for giving me the opportunity to pursue my thesis in the “Database and Information Systems” group at the Max Planck Institute for computer science.

I would also like to thank my friends for their support and for the interesting moments that I spent with them during my master.

I am also thankful to the German Academic Exchange Service (DAAD), for supporting me during my Master’s study.

Finally, I would like to dedicate this thesis to my father, my mother, my brothers, my sisters, and my wife. I am deeply grateful to them for their support and encouragement.
Abstract

Web archives play an important role in preserving born-digital contents, archiving data is important for future generations, researchers, historians, and for the public. Time-travel text search addresses the limited access to web archives by extending regular text search with time-travel functionality. Time-travel text search combines Boolean queries (e.g., mpi AND saarland) and keyword queries (e.g., mpi saarland) with a time-point (e.g., 2011/02/01) or time-interval of interest (e.g., [2010/01/01, 2010/12/31]). Only documents that match the query and whose valid-time interval overlaps with the given query time-interval should be retrieved in response to the query. Time-travel text search has to be highly scalable to cope with huge size of web archives. Hadoop and HBase, as open-source implementations of Google’s MapReduce and BigTable, have recently become popular as tools to deal with massive datasets in a distributed environment.

In this thesis, we build scalable distributed time-travel text search on top of Hadoop and HBase. We devised schemas for storing the indexed data in HBase to support our queries. We build an index for New York Times (NYT) and the Revision History of English Wikipedia (WIKI) collections in a scalable way using MapReduce for indexing and HBase as storage for the index according to our devised schemas. We process our queries using different approaches on the index that has been stored in HBase tables using different schemas. Through comprehensive experiments, we study the performance and scalability of our indexing approaches on different collections. We experimentally evaluate the query processing performance for the different approaches.
# Contents

1 Introduction ........................................ 13  
1.1 Motivation ..................................... 13  
1.2 Problem Statement ............................. 14  
1.3 Contributions .................................. 14  
1.4 Outline ....................................... 15  

2 Technical Background ............................... 17  
2.1 MapReduce and Hadoop ......................... 17  
2.1.1 MapReduce ................................ 19  
2.1.2 Hadoop ................................... 22  
2.2 BigTable and HBase ........................... 25  
2.2.1 BigTable .................................. 25  
2.2.2 HBase ..................................... 31  
2.3 Information Retrieval ......................... 33  
2.3.1 Retrieval Model ............................ 34  
2.3.2 Inverted Index .............................. 36  
2.3.3 Query Processing ........................... 36  
2.3.4 Evaluation ................................ 37  
2.4 Temporal Databases ......................... 39  
2.4.1 Indexing Techniques ....................... 40  

3 Related Work ..................................... 45  
3.1 Time-Travel Text Search ....................... 45  
3.1.1 Model .................................... 45
## Contents

3.1.2 Time-Travel Inverted Index ........................................ 46  
3.1.3 Query Processing .................................................. 48  

3.2 Indexing in MapReduce .................................................. 51  
3.2.1 Per-token indexing .................................................. 52  
3.2.2 Per-term indexing .................................................... 52  
3.2.3 Per-document indexing .............................................. 52  
3.2.4 Per-posting list indexing .......................................... 52  

3.3 Text Search with Hadoop and HBase .............................. 53

### 4 Scalable Distributed Time-Travel Text Search

4.1 Model ................................................................. 57  
4.1.1 Data Model ......................................................... 57  
4.1.2 Query Model ....................................................... 58  
4.1.3 Retrieval Model .................................................... 58  

4.2 Data Layout ........................................................... 58  
4.2.1 Tid_Did ............................................................. 60  
4.2.2 Tid_DidTs .......................................................... 60  
4.2.3 TidDid_Ts .......................................................... 61  
4.2.4 TidDidTs_Tf ........................................................ 62  
4.2.5 TidTs_Did ........................................................... 63  

4.3 Indexing ............................................................... 64  
4.3.1 Mapper ............................................................... 64  
4.3.2 Reducers ............................................................. 64  

4.4 Query Processing ....................................................... 70  
4.4.1 Retrieving Postings List .......................................... 72  
4.4.2 Boolean Queries ................................................... 79  
4.4.3 Keyword Queries .................................................. 80

### 5 Prototype Implementation

5.1 System Architecture .................................................. 83  
5.2 Hadoop Setup ........................................................ 84  
5.3 HBase Setup .......................................................... 86
1 Introduction

1.1 Motivation

The Word Wide Web has become the most important resource of information and an effective medium of communication. It witnessed unprecedented growth in the last two decades.

In the last decades many attempts have been undertaken to make this knowledge discoverable, the most important attempts are full-text search engines such as Bing and Google. The user can use the keyword search interface and get documents that satisfy his query ranked from most relevant to the least. In order to process a user’s query in acceptable time, the query will be executed over a pre-computed index. Index size will increase as the size of web increases, so that we need an efficient scalable way for indexing and storing indexed data.

The current and transient nature of the Web means that new information replaces older information constantly without keeping the old version of the same information, so that many works have been done to keep the old data by archiving the Web. The user looking for information from the past can use the standard web search engines, and probably he will find nothing, because the recent contents replace the relevant old contents. The solution for this problem is to keep everything in the Web, then the user can find old information, and this is exactly the objective of web archives (e.g., the Internet Archive [IA]). Web archives provide limited access functionality. The Wayback Machine, for example, requires knowledge of precise URL or extensive browsing of the archived contents. The limited access on web archives has been addressed by adding the temporal di-
1 Introduction

mension to the inverted index build on the collection of data; an idea has been coined time-travel text search.

Time-travel text search addresses limited access of web archives by extending regular text search with time-travel functionality, so that users can issue queries, as they are used to from search engines such as Google, and also specify a time-point or time-interval of interest. Only versions of web pages that exist at the specified time-point or whose life time overlaps with the specified time-interval will be returned.

The implementation of the time-travel text search have to be highly scalable to cope with increasing size of the web archives, we have recently many tools for dealing with massive datasets in distributed environment, among those tools, Hadoop and HBase the open source variants of Google’s MapReduce and BigTable respectively are considered the most popular tools.

1.2 Problem Statement

The goal of this thesis is to devise a scalable implementation of time-travel text search on top of Hadoop and HBase. More specifically, the implementation should support Boolean queries and Keyword queries combined with a time-point (e.g., 20110201) or time-interval of interest (e.g., [20100101, 20101231]. Only documents that match the query and whose valid-time interval overlaps with the given query time-interval should be retrieved in response to the query.

1.3 Contributions

We make the following contributions:

1. Schemas to store indexes in HBase to support our queries. In doing so, we take into account the characteristics of HBase, especially the way how data
is stored and retrieved from HBase tables, and also the built-in timestamps associated with each cell in the HBase table.

2. Scalable indexing of versioned document collections (e.g., New York Times (NYT), and the Revision History of English Wikipedia (WIKI)). This includes tokenizing and inversion. We used the Hadoop implementation of MapReduce for indexing document collection, the resulting indexes are stored in HBase using different schemas.

3. Query processing based on the stored indexes, taking into account the schema that has been used for storing data in HBase table.

4. Experiments that evaluate our approaches and demonstrate their scalability.

1.4 Outline

The rest of the thesis is organized as follows: In Chapter 2, we provide technical background about tools that we used such as Hadoop MapReduce and HBase. In Chapter 3, we explore some related work. In Chapter 4, we present our approaches for storing data in HBase, indexing techniques that we used, as well as the query processing. In Chapter 5, we describe the implementation details of the techniques introduced in Chapter 4. In Chapter 6, we present the experimental evaluation of our approaches. Finally, in Chapter 7 we conclude our work and propose directions for future work.
2 Technical Background

In this chapter we give some technical background about the systems and tools that we use in our work, as well as a brief introduction to information retrieval.

2.1 MapReduce and Hadoop

Motivation One way to speed up the processing of massive data is by dividing the entire task into small tasks, that can then be executed in parallel using several machines. Running jobs in parallel creates the following problems:

1. Dividing work into equal-sized pieces is not an easy or obvious job. If we have different files of different sizes, then the nodes processing long files need more time than the nodes processing shorter files, so it is better to divide each file into chunks and assign chunks to nodes.

2. Combining results from different nodes needs further processing.

3. Coordination and reliability problems. So how to assign tasks to nodes, how to be sure that the node gets the required data for accomplishing its task, how to share results between nodes, and how to accomplish all of these in face of software errors and hardware faults.

The aforementioned problems have been solved by the MapReduce framework. We will discuss the important concepts behind MapReduce, before discussing MapReduce itself. MapReduce is not the first to introduce these concepts; some of them have been introduced in the literature decades ago. However, MapReduce developers put these concepts together in a very interesting way that was not known previously. We present these concepts in the following points:
2 Technical Background

1. **Scale “out”, not “up”**. In data-intensive workloads, a large number of commodity machines (i.e., scaling “out”) is preferred over a small number high-end servers (i.e., the scaling “up”).

2. **Fault-tolerance**. Assumes that failures are common, so that a system must cope with them as an intrinsic aspect.

3. **Move processing to data**. Instead of moving data to the processing machines as in traditional high-performance computing which has separate storage machines and processing machines, it is better to have data on cheap commodity machines, and then move the running code to the data nodes. We can take advantage of data locality by running the code on the processor attached with same node, that contains the data.

4. **Process data sequentially**. Data-intensive processings process large datasets which are held on disk, since they are too large to fit in memory. Seek times for random disk access are limited by the mechanical nature of the devices. So that it is better to process data sequentially and avoid random access.

5. **Hide system-level details from developers**. *MapReduce* maintains a separation between which computations to be performed and how these computations are actually performed on a cluster of machines. This separation will isolate the programmer from low-level details. The programmer does not care about splitting data into blocks, dividing large task into small tasks, assigning data blocks to tasks, and handling errors.

6. **Scalability**. We can define scalability along the following two dimensions. First, in terms of data, given twice the amount of data, the same algorithm should take at most twice the running time. Second, in terms of resources, if the cluster size has been doubled, the same algorithm should take at most half the time to finish. An ideal algorithm should have is these two properties without any modification or tuning.
2.1 MapReduce and Hadoop

2.1.1 MapReduce

MapReduce [DG10] is a programming model and an associated implementation for processing massive datasets. It was initially developed at Google as a platform for parallel processing of large datasets. To be precise, MapReduce can refer to three distinct but related concepts. First, MapReduce is a programming model, consisting of two main user-defined functions; map and reduce. Second, MapReduce the execution framework ("runtime"). Third, and MapReduce can refer to the software implementation of the programming model and the execution framework, such as the open-source implementation provided by Hadoop.

MapReduce has become very popular, because of its attractive properties. The most attractive qualities of programs written using MapReduce model are:

- High scalability: processing petabytes of data on hundreds or even thousands of cheap commodity machines.
- MapReduce takes care of retrieving data, partitioning input data, outputting data, scheduling parallel execution, coordinating network communication, and handling machine failures.

The MapReduce computation takes a set of input key-value pairs, and produces a set of output key-value pairs. It has two user-defined functions, the map and reduce functions. The map function takes the key-value pair, and generates a set of intermediate key-value pairs. The reduce function takes in all intermediate key-value pairs associated with a particular key, and emits a final set of key-value pairs. The input pairs to the map function and the output pairs of the reduce function are placed in the underlying distributed file system (DFS). Their interfaces are as follows:

\[
\text{map} : (k_1, v_1) \rightarrow [(k_2, v_2)] \\
\text{reduce} : (k_2, [v_2]) \rightarrow [(k_3, v_3)]
\]
2 Technical Background

Figure 2.1: MapReduce job execution stages (taken from [DG10])

MapReduce Execution Framework Figure 2.1 shows the execution of a MapReduce program using Google’s implementation. The execution process is done in steps, each step has a number, as shown in the Figure 2.1, we refer to these numbers as we discuss the execution process. A MapReduce program, which is referred to as a job consists of the map and reduce functions packaged with configuration parameters such as the input path and the output path. To execute the job, it will be sent to the submission node (called JobTracker in Hadoop). The first step is splitting the input data into \( M \) equal splits and starting many copies of the program on the cluster (Step 1). The number of mappers (workers which execute map function) is equal to the number of splits. The number of reducers (workers which execute reduce function) can be specified by the user as a configuration parameter (\( R \)).
2.1 MapReduce and Hadoop

One of the nodes in the cluster will be selected as a master node, all other nodes are workers. The master node selects the mapper and reducer workers (Step 2). A mapper reads the content of the corresponding split, and executes the map function on the content. Intermediate key-value pairs output of the mappers are buffered in memory (Step 3). Buffered intermediate key-value pairs will be written to disk periodically (Step 4). After all mappers have completed, the intermediate data on disk will be partitioned into equal-size partitions, the number of partitions is equal the number of reducers (R). The locations of intermediate pairs are sent to the master, which then forwards these locations to the reducers. Reducers use remote procedure calls (RPC) to read the data from the mappers (Step 5). Each reducer sorts its input data, aggregates by key, executes the reduce function, and then writes the output key-value pairs into a separate output file (Step 6). When all reducers finish their tasks, the master notifies the user program about job completion.

The execution framework is responsible for taking care of all aspects of distributed code execution, these responsibilities include:

- **Scheduling.** The scheduler divides the MapReduce job into smaller tasks, each task is responsible for a part of the input. A mapper is responsible for processing a certain block of key-value pairs. A reducer is responsible for processing a portion of intermediate key-value pairs. The scheduler also keeps track of the progress of running tasks and maintains a queue of pending tasks and assign them to worker nodes when they become free.

- **Data/code co-location.** One of the main ideas of the MapReduce is moving code to data not data to code. The scheduler takes this into account when assigning tasks to workers, so it will run a task on a node which contains the needed data. If this is not possible, then the scheduler will run the task on a node in the same rack of the node which contains the data, so cost of moving data over the network will be reduced.

- **Synchronization.** Synchronization is needed between the map and reduce
phases, when moving the intermediate key-value pairs from mappers to reducers. Copying the data over the network is called *shuffling*.

**Partitioners and Combiners** To give a complete overview of the *MapReduce* execution framework, we have to discuss the two other components; *partitioners* and *combiners*.

**Partitioners** partition the intermediate key-value pairs based on the keys resulting from the mappers into a number of partitions equal to the number of reducers. Thereafter, assign resulting partitions to the reducers. The easiest way is to compute a hash value of the key and then take the mod of this value with the number of reducers (R). The result of this computation gives the reducer number to which this key should be assigned.

**Combiners** aggregate the mappers’ intermediate output key-value pairs by key, before the shuffle and sort phases. this will reduce the amount of data transferred over the network.

**Example** The common *MapReduce* example is word count. Algorithm 2.1 gives pseudo-code for word count. It is a *MapReduce* task to count the occurrences of each word in a collection of documents. The map function emits each word plus an associated count of occurrences, the simple count is one, so each time the map worker finds the word in the collection documents it emits one for this word. The reduce function sums all counts emitted for a particular word. Figure 2.2 shows the execution of Algorithm 2.1 to count word frequencies in two documents.

### 2.1.2 Hadoop

For our work, we use the *Hadoop* [had] implementation of *MapReduce*. *Hadoop* is a popular implementation of *MapReduce*. *Hadoop* is an open-source Java application. It is a software framework that supports data-intensive distributed applications running on large clusters of commodity hardware. It was inspired by
2.1 MapReduce and Hadoop

Google’s MapReduce and Google File System (GFS). It consists of a distributed file system called Hadoop Distributed File System (HDFS) and a MapReduce execution framework on top of HDFS. HDFS meta data is kept in a single node called NameNode and raw data is stored in many nodes called DataNodes. The node which takes the responsibility of co-ordinating workers (TaskTrackers) is called JobTracker. Normally, the NameNode acts as a JobTracker, and DataNodes are the TaskTrackers.

Comparison between Hadoop MapReduce and Google’s MapReduce. There are some differences between Hadoop’s implementation of MapReduce and Google’s implementation. We summarize them in the following:

- In Google’s implementation, the reducer’s output key must be exactly the same as the reducer’s input key. In Hadoop, the reducer can emit an arbitrary number of key-value pairs with different keys.

Algorithm 2.1: Pseudo-code for word count MapReduce algorithm

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Collection of documents</td>
<td><strong>Output:</strong> Number of occurrences for each term in the documents collection</td>
</tr>
<tr>
<td><strong>map</strong> (key, value)</td>
<td></td>
</tr>
<tr>
<td><strong>begin</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>key is the document identifier</td>
</tr>
<tr>
<td></td>
<td>value is the document content</td>
</tr>
<tr>
<td></td>
<td>for each term ( t \in \text{document } d ) do</td>
</tr>
<tr>
<td></td>
<td>emit( (t, \text{&quot;1&quot;}) )</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td></td>
</tr>
<tr>
<td><strong>reduce</strong> (key, set of values)</td>
<td></td>
</tr>
<tr>
<td><strong>begin</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>key is the term</td>
</tr>
<tr>
<td></td>
<td>values is the list of counts, list of ones</td>
</tr>
<tr>
<td></td>
<td>sum ( \leftarrow 0 )</td>
</tr>
<tr>
<td></td>
<td>for each value ( v \in \text{values} ) do</td>
</tr>
<tr>
<td></td>
<td>sum ( \leftarrow \text{sum + v} )</td>
</tr>
<tr>
<td></td>
<td>emit( (\text{key, sum}) )</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td></td>
</tr>
</tbody>
</table>
Figure 2.2: Illustration of the execution of Algorithm 2.1 to count words frequencies using two documents as input

- In Google’s implementation, the reducer’s input data could be sorted by specifying a secondary sort key for ordering values. In Hadoop, the reducer gets as input a key and an iterator over all values associated with that key.

- In Google’s implementation, master failures are handled by making the master periodically write checkpoints. However, it is assumed that master failures are unlikely to happen. Therefore if the master fails, the MapReduce...
computation will be aborted. In Hadoop, there is a Secondary NameNode, which communicates periodically with the NameNode to take snapshots of the HDFS metadata.

There are naming differences between Google’s MapReduce and Hadoop implementation of MapReduce. Table 2.1 summarizes these differences.

<table>
<thead>
<tr>
<th>Google’s MapReduce</th>
<th>Hadoop’s MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFS</td>
<td>HDFS</td>
</tr>
<tr>
<td>Master storage</td>
<td>NameNode</td>
</tr>
<tr>
<td>Slaves storage</td>
<td>DataNodes</td>
</tr>
<tr>
<td>Master execution framework</td>
<td>JobTracker</td>
</tr>
<tr>
<td>Slave (workers) execution framework</td>
<td>TaskTrackers</td>
</tr>
<tr>
<td>Backup task</td>
<td>Speculative copy</td>
</tr>
</tbody>
</table>

Table 2.1: Naming differences between Google and Hadoop implementation of MapReduce

### 2.2 BigTable and HBase

#### 2.2.1 BigTable

**Motivation** Nowadays, there are a variety of existing and newly-emerging applications, whose problems can’t be solved by traditional relational database management systems (RDBMS). RDBMS appeared in the early of 1970’s, and since that time they met the needs of many companies and organizations. Yet, using the RDBMS model makes sense for some applications, but also there are many applications that don’t fit this model. Ideally, one database system would be capable of serving the needs of very different customers. Stonebraker and Çetintemel claimed that the time of this “one size fits all” has come and gone in their paper [Sto08].

Nowadays, many companies realized the value of their data, and they need to keep huge amount of data. These companies started to find solutions to meet
their needs. For example, Google published a series of technical papers, that described scalable storage and processing system based on commodity machines. In 2003, Google published their file system in [GGL03], abbreviated by GFS. It uses a cluster of commodity machines to store a huge amount of data on a distributed file system. The file system handles data replication, for availability reasons. The file system supports stream reading, so that data could be read for later processing.

After that, Google published MapReduce in [DG10]. MapReduce and GFS forms the backbone for processing massive amounts of data. This processing lacks the ability to access data randomly. Access time is not close to real-time. GFS is good with a few very, large files, but not as with millions of tiny files, because the more files, the higher the pressure on the master’s memory.

Google started to find a solution that could support interactive applications, such as Mail. They designed BigTable, which relies on the GFS for replication and availability. The stored files are small blocks. The system takes care of aggregating them into large files, and provide some sort of index on the data to answer users queries with low latency.

BigTable [CDG⁺08] was designed to be a distributed storage system for managing structured data at Google, it is based on the proprietary Google File System, which gives BigTable the ability to scale across hundreds or thousands of commodity machines that collectively can store petabytes of data. BigTable is designed to be scalable both in terms of storage size and number of machines. BigTable has achieved several goals: wide applicability, high scalability, high performance, and high availability. It is used by many products and projects in Google, including Google Analytics, Google Finance, and Google Earth.

BigTable does not support a full relational data model, instead it provides clients with a simple data model that supports dynamic control over data layout, and allows the client to reason about data locality in the underlying storage and have control on the data locality through careful choices of their schemes.

Data Model
2.2 BigTable and HBase

**Table** Each table is a sparse, distributed, persistent multidimensional sorted map. The map is a collection of keys and a collection of values, where each key is associated with one value.

*BigTable* is persistent meaning that the data stored in the map persists after the program that created or accessed it finished. It is distributed because it is built on *Google File System (GFS)*, which is a distributed file system, so that the underlying storage could be spread out among multiple independent machines. Key-value pairs are kept in lexicographic order. Sorting is very important for *BigTable*, because it tends to be so huge and distributed, so sorting makes it easy to scan table and find related rows near each other. The table is sparse, since each row can have an arbitrary number of columns in each column family.

In contrast to traditional database which requires that all rows have the same number of columns. Storing *NULL* values is free of any cost in *BigTable*, if there is no value then we can omit the whole column.

*BigTable* is multidimensional, we think of it as map of maps, it has row key mapping and column family mappings. The table consists of rows and columns, and each cell has a timestamp, so each data cell in the table is organized into three dimensions: row, column, and timestamp, these three dimensions compose a triple \( <\text{row},\text{column},\text{timestamp}> \) key for key-lookup, insert, and delete.

We use the Webtable example in Figure 2.3 taken from the original *BigTable* paper [CDG+08] to illustrate *BigTable*’s data model. Webtable stores the URLs as row keys, various aspects of web pages as column families. The *contents* column family is used to store the content of the web pages under the timestamps when they were fetched. The *anchor* column family contains several columns, each column represents the text of any anchor that has reference to the row key URL.

**Rows** *BigTable* maintains data in lexicographical order by row key. Keys are arbitrary strings. Rows with consecutive keys are grouped in tablets, which is the unit of distribution and load balancing. Sorting rows and grouping consecutive keys in the same tablet gives the user the ability to locate data in the table. For
2 Technical Background

![Webtable Diagram](image)

Figure 2.3: Part of the **webtable** that stores Web pages. Row keys are reversed URLs. The **contents** column family contains the web page contents, and the **anchor** column family contains the text of any anchor that references the page. CNN’s page is referenced by two anchors `anchor:cnnsi.com` and `anchor:my.look.ca`, so the row contains column for each one of them. Each anchor has one version, and the contents family has three versions, (taken from [CDG+08])

example, in Webtable pages illustrated in Figure 2.3, the row key is a reversed URL, so that pages from the same domain are stored together into contiguous rows. Data from `www.cnn.com` are stored under the `com.cnn.www` key.

**Columns** Each row could have an arbitrary number of columns. Column keys are grouped into column families, a column family forms the unit of access control. Column families of the table have to be created before storing data. Data could be stored under any column key in the column family. **BigTable** has a small number of column families (a few hundreds at most), but an unbounded number of columns. For example, Webtable in figure 2.3 contains two column families **contents** and **anchor**, they are different. Anchors that reference the page are stored in different columns in the **anchor** column family, which could contain many columns depending on the number of anchors.

**Timestamps** A cell in the table can contain multiple versions of the same data. The timestamps are 64-bit integers, they can be assigned implicitly by **BigTable** and represent real time in microseconds. Multiple versions of a cell are stored in decreasing order of timestamps, so the recent versions are read first. Users can specify the number of versions to be read. The **webtable** in Figure 2.3 keeps the
2.2 BigTable and HBase

most recent three versions for every page.

**BigTable Building Blocks**  
BigTable is built on *Google File system (GFS)* [GGL03], Chubby Lock Service [Bur06], SSTable, and other Google programs. In the following, we will discuss these components.

**Google File System (GFS)**  
BigTable uses GFS to store log and data files. GFS is a distributed file system that maintains multiple replicas of each file for reliability and availability reasons.

**Chubby Service**  
A highly-available and persistent distributed lock service, it consists of five active replicas, one of them is elected as master and serves requests, the service is live when a majority of replicas are live and communicate with each other. Chubby provides a namespace that consists of directories and files, each one of them can be used as a lock, reads and writes to a file are atomic. Each Chubby client maintains a session with a Chubby service, the session expires if the client is not able to renew the session in the lease expiration time, once the session expires, the client loses the lock. BigTable uses Chubby service for the following tasks:

- To ensure that there is only one active master at any time.
- To store the bootstrap location of BigTable data.
- To store BigTable schemas.
- To discover tablet servers.
- To store access control list.

ZooKeeper is an open-source implementation of Chubby service.
2 Technical Background

**SSTable** Is an immutable-file format used to store *BigTable* data files, it provides a persistent, ordered immutable map from keys to values. Both keys and values are arbitrary byte strings. *SStable* contains a sequence of block (the default size is 64KB, but the size can be configured). At the end of *SStable* there is a block index used to locate blocks, the index block is loaded into memory when the *SStable* is opened.

**Implementation** *BigTable* has three major components: one master server, many tablet servers, and client linked library.

**Master server** is responsible for:

- Assigning tablets to the tablet servers.
- Detecting the addition and expiration of tablet servers.
- Balancing load between tablet servers.
- Handling garbage collection.
- Handling schema changes.

**Tablet servers** Each tablet server manages a set of tablets, typically the number of tablets is between ten to a thousand tablets per tablet server. A tablet server handles the reads and writes requests for the tablets that it contains. It takes care of the tablet size, so any tablet grows too large will be splitted by the tablet server. Each table in the *BigTable* cluster initially consists of one tablet, and as the table grows, the number of tablets increases, because of automatic splitting of tablets.

**Tablet Location** *BigTable* uses a three-level hierarchy analogous to B\(^+\)-tree [Com79] to store tablet locations; the root tablet, METADATA tablets, and the
user tables. The Root tablet is stored in Chubby and contains the locations of all tablets by storing the locations of the METADATA tables, each METADATA stores the locations of the tablets in the user tables, which contains the data. Figure 2.4 shows the tablets locations hierarchy.

![Figure 2.4: Tablet locations hierarchy, (taken from [CDG+08])](image)

### 2.2.2 HBase

HBase [hba] is an open-source implementation of Google’s BigTable. It is built on the Hadoop Distributed File system (HDFS) which is used as underlying storage, and the Apache ZooKeeper, a reliable, highly available, persistent, and distributed coordination service like Chubby in BigTable. The major components of HBase are: one master server, many region servers instead of tablet servers in BigTable, and a client library. The HBase table properties are similar to those for table in BigTable system. HBase is distributed, persistent, and strictly consistent¹ storage system. Table 2.2 summarizes the differences in terminology between HBase and BigTable.

The HBase implementation is very close to the BigTable implementation, and

¹Consistency is the guarantee that a database is always appears truthful to its clients. Strict is the highest form of consistency; changes to the data are atomic and take effect instantaneously.
2 Technical Background

<table>
<thead>
<tr>
<th>BigTable</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablet</td>
<td>Region</td>
</tr>
<tr>
<td>Tablet Server</td>
<td>RegionServer</td>
</tr>
<tr>
<td>Minor Compaction</td>
<td>Flush</td>
</tr>
<tr>
<td>Merging Compaction</td>
<td>Minor Compaction</td>
</tr>
<tr>
<td>Major Compaction</td>
<td>Major Compaction</td>
</tr>
<tr>
<td>Commit Log</td>
<td>Write-Ahead Log</td>
</tr>
<tr>
<td>GFS</td>
<td>HDFS</td>
</tr>
<tr>
<td>MapReduce</td>
<td>Hadoop MapReduce</td>
</tr>
<tr>
<td>memtable</td>
<td>MemStore</td>
</tr>
<tr>
<td>SSTable</td>
<td>HFile</td>
</tr>
<tr>
<td>Chubby</td>
<td>ZooKeeper</td>
</tr>
</tbody>
</table>

Table 2.2: Naming Differences between Bigtable and HBase

has all BigTable features, that we presented in 2.2.1, but there are some implementation differences, because HBase relies on other open-source projects like HDFS and ZooKeeper. We will present these differences in following points:

1. Timestamps

   Timestamps HBase are in milliseconds, but BigTable uses microseconds timestamps. This is attributed to timer resolution of C language used by BigTable, and Java language used by HBase.

2. Compression algorithm

   HBase uses compression algorithms supplied in Java and can use LZO², while BigTable uses two-phase compression using BMDiff and Zippy.

3. Underlying Storage

   HBase uses Hadoop Distributed File System (HDFS) primarily, while BigTable uses Google File System (GFS). HBase can work on other file systems such as Amazon Simple Storage Service (S3)³ and Amazon Elastic Block Store

---

²LZO is a data compression library which is suitable for data de-/compression in real-time. This means it favors speed over compression ratio.

³Raw or emulated HDFS
2.3 Information Retrieval

(EBS)\(^4\). This feature is an advantage for *HBase* resulted from the plug-gable FileSystem class provided by *Hadoop*.

4. **Data locality**

In *BigTable* clients can group columns together under the same column family, and also group specific column families together which is called locality groups, while *HBase* has only the column family concept which includes the column family and locality group concepts.

5. **Splitting/ Merging**

Splitting regions in *HBase* and tablets in *BigTable* are the same for both, but merging is handled differently. Merging is handled manually in *HBase*, while it is handled automatically by the master in *BigTable*.

6. **Garbage collection**

In *HBase*, cleanup is done by the region server, but in *BigTable* this is done by the master.

7. **Memory map**

*HBase* has this option per column family, while *BigTable* can map the entire storage files in the memory, and then use them for lookups.

We have presented the major differences between *HBase* and *BigTable*. There are also some minor implementation differences, such as the compaction algorithm and the naming strategy used for *BigTable* tablets and *HBase* regions, and where names are stored.

### 2.3 Information Retrieval

Information Retrieval (IR) is a well known academic field, that has traditionally concentrated on finding whole documents consisting of written text; many IR researches focuses on text retrieval. But there are many other interesting

\(^4\)provides block level storage for use with Amazon EC2 instances
areas, for example, image retrieval, speech retrieval, music retrieval, and cross language retrieval. The primary goal of IR in the past was the indexing of text document collections and search for useful documents in a collection. Nowadays, IR research includes modeling, document classification and categorization, system architecture, user interface, and data filtering and visualization. The following is a definition of IR taken from the textbook by Manning et al. [MRS08]

“Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).”

2.3.1 Retrieval Model

According to the definition of information retrieval, we have to decide how a document satisfies an information need. In the literature, there are many retrieval models; in the following we will discuss two of them, which we used in our work.

**Boolean Retrieval:** An information retrieval model, which is among the earliest retrieval models. The model views documents as sets of terms. For each collection there is a vocabulary \( \nu \), and each document in the document collection consists of a set of terms from the vocabulary set. Queries in the Boolean model are boolean expressions, which means that terms in the query are combined using boolean operators; the conjunction \( \land \), the disjunction \( \lor \), and the negation \( \neg \). The relevant document is the one that satisfies the query’s boolean expression. Relevance in Boolean model is binary, each document is evaluated as relevant or irrelevant to the given query.

**Ranked Retrieval:** The Boolean model assesses the relevance of the document depending on whether the query term is present or not in the document, but this is a binary relevance. The problem of this model is that it evaluates a document as relevant or irrelevant. But it could be that one document is more relevant than another, so we need some kind of weighting. One way is to take the term frequency into account. Evaluating the document to be relevant using only the
2.3 Information Retrieval

term frequency is not fair, because we consider that all terms are equally important. In fact some terms are more important than other terms. For example, a collection of documents about cats is likely to have the term cat in almost every document. We can not depend on the occurrence of the term, which is not important in the query for assessing relevancy, since this term is not discriminative. So weight of a term should be proportional to its importance. The term importance is measured by the document frequency ($df_t$). We need to scale the weight of term considering it’s importance. This is done by the inverse document frequency ($idf_t$). The tf-idf model combines the term frequency ($tf$) and the inverse document frequency ($idf$). Term frequency is the number of occurrences of term $t$ in document $d$, which is denoted as $tf_{t,d}$, with the subscripts denoting the term and the document in order. The inverse document frequency of a term is the total number of documents in the collection over the document frequency of the term as defined in the following equation:

$$idf_t = \log \frac{N}{df_t} \quad (2.1)$$

The $idf$ of a frequent term is low, while the $idf$ of a rare term is high. One of the common tf-idf weighting scheme, that assigns a weight for term $t$ in document $d$, is given by

$$tf-idf_{t,d} = tf_{t,d} \times idf_t \quad (2.2)$$

This weight will thus be highest when $t$ occurs many times in a small number of documents, the weight will be lower when $t$ occurs few times in a document; or occurs in many documents, and the weight will be lowest when $t$ occurs in all documents. Using this model, we can assess the relevance of the document by calculating the score. The score of document $d$ is the sum of the weight of all query terms, as given by

$$score(q,d) = \sum_{t \in q} tf-idf_{t,d} \quad (2.3)$$
2 Technical Background

2.3.2 Inverted Index

Building an inverted index is required to enhance query processing time. When using inverted indexes, query processing can be restricted to documents that contain at least one of the query terms. The inverted index keeps track of collection terms, and a list of documents that contain each term. The item of the list is called posting and the list of postings is called posting list. So the inverted index consists of two main parts: the dictionary (or lexicon) which contains all terms from the collection, and a posting list for each term in the lexicon. The lexicon is usually kept in main memory, and organized using hash table. It could also be stored on disk, and organized using a B-tree [Com79]. For more details on the lexicon implementations we refer to Witten et al. [ZM06]. Posting lists are stored on disk, and fetched at query processing time. The structure of posting lists depends on the query to be supported. For the Boolean retrieval model, postings contain only document identifiers which is enough to answer Boolean queries, as shown in figure 2.5. For keyword query in the ranked retrieval model, we need scalar payloads that contain, for example, the term frequency of the term in the document. Figure 2.6 shows an inverted index which stores the term frequency in the postings. Finally, for supporting phrase queries, we store the position of the term in the document. Posting lists can be sorted in ascending order of their document identifiers (document-ordered), or can be sorted in descending order of term frequency (frequency-ordered).

2.3.3 Query Processing

Now we will discuss how we process a given query, and compute the documents relevance to the given query. For our work we use the term-at-a-time processing technique. We next describe this technique.

Term-At-A-Time (TAAT) processes query terms one by one by retrieving a list of candidate documents for each term, and accumulates partial document scores as the contribution for each query term is computed. Algorithm 2.2 gives the pseudo-code for TAAT conjunctive query processing.
2.3 Information Retrieval

Figure 2.5: Simple inverted index, which keeps for each term in lexicon list, a list of documents that contain this term

Figure 2.6: Inverted index, which keeps list of postings. Postings contain documents identifiers and the payload for the corresponding term

2.3.4 Evaluation

After we build our information retrieval system, we have to discuss how we measure the effectiveness of our IR system. The two most common and basic measures for information retrieval effectiveness are: precision and recall. In the following we define both of them.
Algorithm 2.2: Pseudo-code for TAAT conjunctive query processing

Input: Given query q, and inverted index
Output: List of scored relevant documents

begin
1 double Scores [N]
2 N is number of documents
3 for each term t ∈ query q do
4 fetch postings list for t
5 for each posting in postings list do
6 Score[d]+ = new score
7 tf-idf model could be used to calculate the new score
8 return components of Score[] in descending order
9 end

Precision \( P \) is the fraction of retrieved documents that are relevant, i.e.:

\[
P = \frac{|\text{Relevant} \cap \text{Retrieved}|}{|\text{Retrieved}|} \quad (2.4)
\]

Recall \( R \) is the fraction of relevant documents that are retrieved, i.e.:

\[
R = \frac{|\text{Relevant} \cap \text{Retrieved}|}{|\text{Relevant}|} \quad (2.5)
\]

F-measure measures the trade off between precision and recall, which is the weighted harmonic mean of precision and recall, it is given by

\[
F_{\text{measure}} = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha) \frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \quad (2.6)
\]

where \( \beta^2 = \frac{1-\alpha}{\alpha}, \alpha \in [0, 1], \) and \( \beta^2 \in [0, \infty] \).

Values of \( \beta < 1 \) emphasizes precision, while values of \( \beta > 1 \) emphasizes recall.

For further information about retrieval models and evaluation we refer to [MRS08].
2.4 Temporal Databases

Temporal databases provide special support for handling data with a temporal component. In a temporal database, objects can have different values at different times. Therefore at a specific point of time, we have a specific snapshot of the database. One of the important things that has to be addressed when creating a temporal database is the representation of time. Model of time can be considered to be either discrete or continuous. The discrete time model represents time with natural numbers, a set of equally spaced and ordered time-points. Continuous time model represents time with real numbers, so the result will be an infinite set of time-points.

There are two different notions of temporal information stored in the temporal database, namely valid time and transaction time.

The valid time denotes the time period during which a fact is true with respect to real word. The transaction time is the time period during which a fact is stored in the database. For example if a temporal database stores data about the 17th century, then the valid time is somewhere between 1600 and 1699, while the transaction time starts when we insert data into database. The transaction time databases store time when data was recorded, time in the past, and it is assumed that they are evolving linearly, so there is no way to modify the recorded data. But in valid time databases, it is possible to modify the time associated with records, and also to add time for future.

Many researches have dealt with different perspectives of temporal databases, such as temporal data models, temporal retrieval and query languages, inter-relationships between temporal databases and other database technologies (e.g., spatial databases), and indexing techniques. The most related perspective to our work is the indexing. In the next we describe some of the indexing techniques.
2 Technical Background

2.4.1 Indexing Techniques

In this section we discuss two indexing techniques for temporal databases; the Multi-Version B-Tree and the Log-Structured History Data Access Method.

Multi-Version B-Tree

Many approaches have been proposed in the literature to deal with multi-version data. Next we describe some of the attempts that deal with time-evolving data by adapting the B-tree [Com79], and then describe the Multi-Version B-Tree.

Indexing Historical Databases

Maintaining multiversion data has been described in 1989 by Kolovson and Stonebraker [KS89]. The main concern for them was to use a big and less expensive storage media to meet the huge storage amount of data for the temporal databases. Since temporal databases keep current and historical data, Kolovson and Stonebraker suggested the use of two storage media, magnetic disk which keeps current data, and optical disk for historical data versions. The historical data may reside on magnetic disk before transferring it into optical disk, data transfer process is called vacuuming. They proposed two approaches, both using R-tree index [Gut84] for organizing records. The first approach organizes records according to a record key value, while the second approach organizes them according to their life spans. The main difference between the two approaches is how the vacuuming process is triggered. In the first, if the index size reaches a specific threshold, then the vacuuming process moves part of the oldest data into the optical disk. The second approach maintains two R-trees, one in the magnetic disk completely, and the upper levels of the other tree on the magnetic and lower levels on the optical. In the second approach vacuuming process is triggered when the size of the R-tree on the magnetic disk reaches the maximum, then all data versions will be transferred to the optical disk except the root.
Time Index

Elmasri, Wuu, and Kim have proposed the time index (TI) [EWK90] for maintaining historical data. TI uses only valid time information, thus it indexes historical databases. Data records in the index time are organized using B+-tree according to versions. For each version, a bucket is maintained to store all records valid for that version. Using this organization strategy causes high redundancy. Several improvement on the TI basic approach have been done. The first modification in [EKW91] was to use two buckets for each leaf node, one for added versions and the other for deleting versions. The leaf entry has a separate incremental buckets. They uses more storage to improve the search performance. A further improvement, instead of having the complete bucket at each entry point in a leaf node, is to keep only versions that still valid from the last bucket in the previous leaf node. This improvement appeared in [KKEW94]

Write-Once-B-Tree

Easten proposed the Write-Once-B-Tree (WOBT) [Eas86], which is a variation of the B+-tree. WOBT is stored in optical storage. All data versions are kept forever, because of the write once property. Multiversion queries are answered in a straightforward way, if the version numbers are assigned to the index and data records. An overflow of data or index block is treated by the version split and key split. In WOBT, the storage space is inefficiently used, because it keeps everything forever, and when the data is split, the old data can’t be changed due to write-once policy.

Time-Split-B-Tree

The Time-Split-B-Tree (TSBT) proposed by Lomet and Salzberg [LS89], is a modification of the WOBT. It solved the problem of inefficient use of storage, by
using two storage medias. Magnetic disk which keeps current data, and WORM\textsuperscript{5} or WMRM\textsuperscript{6} to keep historical data. There are two types of TSBT splitting, the key-split and the time-split. In key-split, TSBT behaves the same as B\textsuperscript{+}-tree. In contrast to WOBT, time-split could be done at any arbitrary time-point. When a node splits at time \( t \), records that are no longer valid at \( t \) are moved to historical node on a WORM media, records valid after \( t \) go to new node, and records valid at \( t \) are stored in both nodes (historical and new).

**Multi-Version B-Tree (MVBT)**

\( MVBT \) proposed by Becker \[ ? \] modifies the B-tree to store multiple versions. It supports insertions and deletions of data items at the current version, and answers range queries and exact match queries of any version in the current or historical data. It stores records in leaf nodes, each record in the leaf node is denoted by \( \langle key, [t_b, t_e] \rangle \). It is a directed acyclic graph of B-tree nodes that results from an incremental changes to the initial B-tree, so that it has a number of B-tree root nodes, and each root node represents an interval of versions.

For each insert or delete (update) operation a new version is created; \( i \)-th update creates version \( i \). \( MVBT \) keeps two invariants for each node. The weak version condition requires that for each version \( i \), the number of entries of each version except the roots is either zero or at least \( d \), where \( b = k \cdot d \) for block capacity \( b \) and constant \( k \). The weak version condition may be violated if an entry has been deleted from a non-root block which has exactly \( d \) entries (block underflow). An insertion into a block which has \( b \) entries causes block overflow. Violation of the weak version condition and the block block overflow will trigger an operation called a version split. A version split copies all current entries into a new node. If the produced copy is almost full, then with few subsequent insertions the version split will be triggered again. Such situations will be avoided by using the strong

\textsuperscript{5}Write Once Read Many such as optical disks, data in such a storage can’t be modified after written
\textsuperscript{6}Write Many Read Many
version condition, which demands that, after a version split, at least $\epsilon \cdot d + 1$ updates (insert or delete) are required before triggering a new version split for some constant $\epsilon$. With this constraint, the number of entries on the new node resulting from the first version split must be between $(1 + \epsilon) \cdot d$ and $(k - \epsilon) \cdot d$. A strong version underflow occurs when a number of entries is less than $(1 + \epsilon) \cdot d$, then a merge with a copy of a sibling block is attempted. If the merge causes block overflow, then the block is split using a key split. Similarly, if a version split leads to more than $(k - \epsilon) \cdot d$ entries; strong version overflow occurs, then also keysplit is performed.

Log-Structured History Data Access Method

There are many types of temporal databases, some of these databases keep recent data as well as historical data. Such databases are needed for numerous applications (e.g., banking systems). Keeping all data makes indexing such a temporal databases more challenging, because of the huge amount of data to be indexed, and the support different types of time-travel queries. An access method of temporal databases should be able to so sustain high insert rates.

The Log-Structured History Data Access Method (LHAM) proposed by Muth et al. [MOPW00] introduced new access method for transaction-temporal data, in which multiple versions of a record are kept. LHAM was designed to support high insert/update rate. The begin timestamp for each record is the time when the record was inserted into database, and the end timestamp is the begin timestamp of the next version of the same record. All accesses to the database, including insert, update, and delete are all implemented as insert. Delete operation is implemented by inserting a special record version which marks the end of the record’s lifetime. LHAM solved the problem of high insert rate, and huge size of the data, by partitioning the data into successive components $C_0,..., C_m$ of increasing size based on records timestamps. Component $C_i$ resides in main memory, components $C_2,..., C_k$ reside in secondary storage, and components $C_{k+1},...,
C_m reside in archive media. LHAM splits the time domain into successive intervals, and assigns each one of them into one of the components. Each component has a low-time boundary, low_i, and high-time boundary, high_i, and thus contains record versions whose timestamps fall between its time-boundaries. Components with lower subscripts in the successive components contain more recent data, so C_i is more recent than C_{i+1}. Access rate differs components. Access rate of component C_0 which contains the most recent data is expected to be higher than access rate of any other component, so LHAM stores components in hierarchical storage. Component C_0 is kept in the main memory to support high fast access. Remaining components are kept in a secondary storage or archive media.

New records are inserted into C_0. When the component C_i becomes full, its content will be migrated to component C_{i+1}. Migrating data from more recent to older data is accomplished by rolling merge process. Rolling merge from component C_i to C_{i+1} migrates data from C_i to C_{i+1} and updates time intervals of the components, rolling merge will be triggered each time a component becomes full, so if the rolling migration of C_i to C_{i+1} causes C_{i+1} to be full, then C_{i+1} will be migrated to C_{i+2}. When the last component C_k which contains the oldest data on the secondary storage becomes full, then rolling migration doesn’t migrate C_k to C_{k+1} on the secondary storage, but rolling migration builds a new archive component, named C_{k+1}, and the previous archive components C_{k+1},...C_m are renamed into C_{k+2},..., C_{m+1}
3 Related Work

3.1 Time-Travel Text Search

The concept of time-travel text search has been introduced by Berberich et al. [BBNW07]. They addressed time-travel text search over temporally versioned document collections. Many versioned document collections are available, for example, web archives (e.g., the Internet Archive), news archives (e.g., New York Times Annotated Corpus), or revision histories of wikis (most notably Wikipedia). The time-travel search concept supports answering queries combined with time-point or time-interval. To accomplish this, they extended the inverted index, by adding the temporal dimension to it. The extended inverted index is called Time-Travel Inverted Index.

3.1.1 Model

To achieve the objective of this work in building time-travel text search, the collection, the query, and the retrieval models should be time-aware. They use a collection which contains versioned documents, so each document version has an associated timestamp, denoted by $d^t$. Any modification to a version results in the addition of new version with new timestamp, since they employ a discrete definition of time, the new version will be denoted as $d^{t+1}$. Then the valid-time interval of old version is $[t_i, t_{i+1})$, and for the new version is $[t_{i+1}, now)$. The query model used in this work, is a query $q$ which could be a Boolean query, keyword query, or phrase query, associated with time-point (time-point query) denoted by $q^t$, or time-interval (time-interval query) denoted by $q^{[t_b,t_e]}$. For retrieval model, they adapted the Okapi BM25 to make it time-aware. To measure the relevance
3 Related Work

of document version \(d^t\) to time-travel keyword query \(q^{[t_b, t_e]}\); the authors adapted Okapi BM25 which is given by

\[
w(d^t, q^{[t_b, t_e]}) = \sum_{t \in q} w_{tf}(t, d^t) \cdot w_{idf}(t, q^{[t_b, t_e]})
\]  

(3.1)

The \(w_{tf}(t, d^t)\) is given by

\[
w_{tf}(t, d^t) = \frac{(k_1 + 1) \cdot tf(t, d^t)}{k_1 \cdot ((1 - b) + b \cdot \frac{dl(d^t)}{avdl}) + tf(t, d^t)}
\]  

(3.2)

where \(dl(d^t)\) is the document version length, \(avdl\) is the average document length, \(b\) is the length normalization parameter, and \(k_1\) is the term-frequency saturation parameter.

The \(w_{idf}(t, q^{[t_b, t_e]})\), which is the inverse document frequency of the term \(t\) in the collection, is defined as

\[
w_{idf}(t, q^{[t_b, t_e]}) = \log \frac{N([t_b, t_e]) - df(t, [t_b, t_e]) + 0.5}{df(t, [t_b, t_e]) + 0.5}
\]  

(3.3)

Where \(N([t_b, t_e])\) is the number of document versions in the collection valid at any time in \([t_b, t_e]\), and \(df(t, [t_b, t_e])\), is the number of document version valid at any time in \([t_b, t_e]\) and contain term \(t\).

3.1.2 Time-Travel Inverted Index

In order to support time-travel text search [BBNW07] extended the standard inverted index, that we have introduced in 2.4.1, by adding explicitly the time functionality to both lexicon and posting lists. The extended inverted index is called Time-Travel Inverted Index (TTIX). Postings are extended by adding a valid-time interval denoting when the term payload was valid. Postings in TTIX have the following form

\((d, [t_i, t_j], p)\)
3.1 Time-Travel Text Search

Instead of maintaining one posting list $L_v$ per term as the standard inverted index does, TTIX maintains multiple posting lists and supports temporal partitioning. Posting lists have an associated interval of time $[t_k, t_l)$, and posting list $L_v: [t_k, t_l)$ contains all postings whose valid-time interval overlaps with $[t_k, t_l)$, i.e.,

$$L_v: [t_k, t_l) = \{(d, p, [t_i, t_j]) \in L_v | t_i < t_l \land t_j > t_k\}, \quad (3.4)$$

$L_v$ alone can be used to retrieve all postings for term $v$ in the query whose time-interval $[t_b, t_e) \subseteq [t_k, t_l)$. The set of time-intervals associated with posting lists for term $v$ in the index represents the temporal partitioning $P_v$ for term $v$. Posting lists in $P_v$ must cover $L_v$ completely.

In order to speed up query processing time, the posting list $L_v$ is physically stored in two separate posting lists $L^p_v$ and $L^+_v$ on disk. $L^p_v$ represents a list of postings already exist before time $t_k$, and $L^+_v$ list contains all postings created during $[t_k, t_l)$. $L^p_v$ and $L^+_v$ are disjoint and their union represents all postings from $L_v$, $L^p_v$ and $L^+_v$ are defined as follow

$$L^p_v: [t_k, t_l) = \{(d, p, [t_i, t_j]) \in L_v | t_k \leq t_i \land t_j \leq t_l\}, \quad (3.5)$$

Figure 3.1: Time-Travel Inverted Index
posting lists in the TTIX is not restricted to any sort order, it could be sorted by document, score, or valid-time interval boundaries, but the sort order should be consistent for all posting lists.

3.1.3 Query Processing

Time-travel text search presented in [Ber10] supports time-interval queries \( q^{[t_b, t_e]} \). Now we will describe how the authors of this paper process these queries.

The time-point query \( q^t \): For each term \( v \) in the query, all postings for \( v \) whose valid-time interval contains the query time \( t \) will be retrieved. Retrieving all postings could be done by retrieving the posting list \( L_v : [t_k, t_l] \) from the index, where \( t \in [t_k, t_l) \). If there are multiple qualified posting lists, then the shortest one will be chosen using the following optimization definition.

\[
\arg\min_{[t_k, t_l] \in P_v} |L_v : [t_k, t_l]| \quad \text{s.t.} \quad t \in [t_k, t_l)
\]

Retrieving the posting list \( L_v \) requires reading and merging the two posting lists \( L^+_v \) and \( L^-_v \). While reading, irrelevant postings, whose valid-time interval doesn’t include query time-point are filtered out. The filtering process will be done after transferring the postings from the disk to the main memory, and this causes significant I/O cost. In order to reduce the I/O overhead, the authors develop a partitioning strategies that determine partitions at the index creation time, we discuss them later.

The time-interval query \( q^{[t_b, t_e]} \): For each term \( v \) in the given \( q^{[t_b, t_e]} \), all postings whose valid-time interval overlaps with the query time-interval are retrieved. Because of temporal partitioning \( P_v \) structure, which consists of time-intervals
for which the index keeps a posting list for term \( v \), there may be no single list \( L_v : [t_k, t_l] \) such that \( [t_k, t_c] \subseteq [t_k, t_l] \). To cover the query time-interval a set \( \mathcal{L}_v \subseteq P_v \) of posting lists will be retrieved. It is possible to find a different sequence of posting lists which cover the query time-interval, the selection of posting lists will affect the number of postings being read and merged, and thus this will affect the query processing time. The result of time-interval query may have multiple versions per document, for query processing improvements, the authors develop the temporal coalescing techniques.

**Temporal Coalescing** attempts to reduce the index size in general and speed up query processing for all queries. The naïve (it keeps one posting per term per document version) way to index versioned document collection leads to a huge number of postings, especially for frequent terms which appear in many versions, and for highly-dynamic collection. Many of the postings that belong to one term in versions of the same document may have minor changes, e.g., spelling corrections. The idea of coalescing approach is to coalesce postings of versions of the same document belonging to one term which are temporally adjacent and have minor changes. The decision of what is minor changes are depends on the query type, e.g., for Boolean query, we do not keep payload in the postings. Thus, if we have two adjacent postings containing a term, then we can coalesce them in one posting whose time-interval covers both. However, for all query types, coalescing approach takes as an input a sequence of \( n \) temporally adjacent postings which represent a contiguous time period for term \( v \) in document \( d \).

\[
I = < (d, [t_1, t_2), p_1), (d, [t_2, t_5), p_2), \ldots, (d, [t_n, t_{n+1}), p_n) > \tag{3.7}
\]

and produces an output sequence which consists of coalesced postings

\[
O = < (d, [t'_1, t'_2), p'_1), \ldots, (d, [t'_m, t'_{m+1}), p'_m) > \tag{3.8}
\]

which covers the same time-interval as the input sequence \( I \), and has number of
3 Related Work

coalesced postings $|O| = m \leq n$

**Partitioning Strategies** The performance of processing a time-point query is influenced by the I/O overhead wasted for reading all postings for a term, and then filter-out irrelevant postings. The main objective of the partitioning strategies is to improve the query processing performance. Figure 3.2 shows 5 postings for term $v$ in versions of three documents.

![Postings for term $v$ to illustrate partitioning strategy](image)

Figure 3.2: Postings for term $v$ to illustrate partitioning strategy

Given a time-point query at time $t \in [t_1, t_2)$, which contains term $v$. If we have only a single list $L_v : [t_1, t_5)$ in the inverted index, then we need to read all 5 postings (this is the worst case). But if we have a list $L_v : [t_1, t_2)$ in the inverted index, then we read only two postings (namely, 1, and 2) and we achieve the optimal query processing performance (this is the best case). Thus, if we keep a posting list for every time-interval, then only postings that are valid at the time of the query will be read. Keeping postings this way will increase the size of the inverted index significantly. In the best case, performance in not good, but space consumption is the minimal (this strategy is called $S_{opt}$). In the best case performance is optimal, but it is not space-efficient (this strategy is called $P_{opt}$).

$S_{opt}$ and $P_{opt}$ are optimal in terms of space and performance, respectively. To trade off space and performance in a controlled manner, the authors came up
with another two partitioning approaches; the \textit{Performance-Guarantee} and the \textit{Space-Bound}.

\textbf{Performance-Guarantee approach.} The idea behind this approach, that it is possible to save a significant space, but with a guarantee on query processing. The guarantee is that the query processing cost is worst than optimal by at most a factor of $\gamma \geq 1$

\textbf{Space-Bound Approach.} The idea of this approach, that it is possible to optimize query processing performance, while not exceeding a given space bound.

\section*{3.2 Indexing in MapReduce}

Indexing is an important operation in Information Retrieval, which should be parallelized to support huge data collection size. Many works have been done to index data in a distributed manner. We will discuss indexing techniques that have been proposed using \textit{MapReduce}.

Indexing using \textit{MapReduce} has been described shortly in original \textit{MapReduce} paper [DG08]. They indexed data using the two user defined \textit{MapReduce} functions, map and reduce. The map function parses each document, and emits \langle \text{word, document ID} \rangle pairs. It emits for each word, the word itself and the document id containing the word. Reduce function reads all intermediate pairs of the map function, sort the document id’s emitted for every word, and emits a list of document id’s that contain this word in the form \langle \text{word, list(document id)} \rangle pairs. The output pairs of the reduce function forms a simple inverted index. The computation in the map and reduce function could be modified to emit the frequency of the word in the document, or even the position of the word in the document.

For the inverted index, it is better to store term frequencies, to implement it, there are two possible techniques: \textit{per-token indexing} and the \textit{per-term indexing}. Next, we discuss these techniques.
3 Related Work

3.2.1 Per-token indexing

The map function outputs a set of \( \langle \text{token, document id} \rangle \) for each token in the document. For each token, the reduce function aggregates document ids which contain this token. Using this strategy, the \( \langle \text{term, document id} \rangle \) will be emitted by the map function each time the token appears in the document. Therefore, the intermediate data of the map will be very large, and this will increase the map-to-reduce network traffic.

3.2.2 Per-term indexing

The map function in per-token indexing technique produces a huge output. To reduce this, we emit \( \langle \text{term, (document id, tf)} \rangle \). Emitting once per term is called per-term indexing strategy. In this strategy, the map function computes the frequency of the term in the document. This will reduce the number of “emit” significantly. The reduce function in this implementation becomes easier, it emits for each term a list of ids for documents which contain the term. This strategy of indexing has been implemented in the Ivory system [LMEW]

3.2.3 Per-document indexing

Instead of emitting per term or token as described above, this approach emits per document. The map function only tokenizes the document, and emits \( \langle \text{document id, analyzed document} \rangle \) tuples. the analyzed document contains the textual forms of each term with the corresponding frequencies. The reduce function takes the responsibility to build the index. This approach has been implemented by The Nutch platform [CC04]. This indexing approach emits less than per-token and per-term indexing approaches, but the value of each emit is larger.

3.2.4 Per-posting list indexing

This approach was proposed in [MMO09]. They adapted the single pass indexing strategy proposed by Heinz and Zobell in [HZ03] for MapReduce. It is also
employed by Terrier IR platform. In the single-pass indexing, the posting lists for each term are built in memory as the collection is scanned. When the memory becomes full, data will be flushed to disk. The final index is built by merging flushed data. The indexing process in this approach is split into multiple map tasks, each map task works on a subset of the data collection. The map task builds the posting lists for each term in memory, while processing documents. When the memory is exhausted, the partial index is flushed into disk by emitting \langle term, posting list \rangle pairs for all terms in memory. Flushed partial indexes are sorted by term, map and flush numbers before passing them to reduce tasks. Reduce task takes terms one by one, and for each term merges the posting lists into the full posting list. This approach has the positive effect of minimizing the size of the map output and the number of emits. Size of the postings themselves will also be small. Compared to per-token and per-term, the per-posting indexing emits less, but the size of the emitted value will be larger. Compared to the per-document indexing, the per-posting indexing emits less, while the emitted value size is smaller.

3.3 Text Search with Hadoop and HBase

In this section, we describe existing work that focused on implement text search using Hadoop and HBase

In [LRW11], the authors address one of the inefficient aspects of Hadoop-based processing, which is the need to full scan the entire dataset. A full text scan is not necessary all the time. The idea of this paper is to leverage a full-text index to optimize selection operations on the text fields within records. Therefore, the inverted index informs the Hadoop search engine about the blocks which contain the given query terms. Thereafter Hadoop engine fetch only those blocks, instead of scanning the whole blocks.

The work has been done in [KATK10b] is similar to our work in using MapReduce
3 Related Work

framework for processing and building inverted index, and using HBase table for storing inverted index. This paper incorporates and extends the functionality of Hadoop and HBase to create a distributed index on large datasets, that supports different types of data; structured, semi-structured, and unstructured. They process and index the data using MapReduce framework, and use HBase table for storing the indexed data, to achieve low response time for answering queries. Indexing is done in two stages using two MapReduce jobs. The first MapReduce job represents the work of what they called uploader. The uploader reads and analyzes the raw data from the HDFS. Later it stores the content in HBase table, called content table. The second MapReduce job is for indexing itself. This job is called the indexer. The indexer reads data from the content table, builds the inverted index, and stores the inverted index in HBase table. Query search API is build on top of the HBase’s client API for answering exact query and range query search.

The Ivory approach, described in [LMEW] uses the Hadoop implementation of MapReduce for building a distributed retrieval system. The authors of this paper use the MapReduce algorithm to build the inverted index. We described this algorithm in section 3.2.2 MapReduce indexing approach. The inverted index is stored in the Hadoop Distributed file system (HDFS). And The retrieval system is built directly on top of HDFS. They used Metzler’s SMRF (Search using Markov Random Fields) retrieval engine. The major modification, is that fetching postings is done from the HDFS, instead of the local disk. Data stored in the HDFS is split into partitions, the NameNode contains locations of partions (blocks), which are stored in the DataNodes. So fetching any posting needs two contacts: first contact the NameNode to find the location of the block which contains the posting, and then contact the DataNode that contains the posting. For answering a user’s query and returning postings from the HDFS, there should be a middle component. In the Ivory system, the server partition does this job. Server partition is implemented using a MapReduce job that runs only mappers. Inside each mapper, there is a server that handles query over TCP connection.
and accesses postings from \textit{HDFS}.

The \textit{HIndex} - a distributed index - has been presented in [LRST09]. The \textit{HIndex} exploits the \textit{HBase} control layer to inherit the support on availability, elasticity and load balancing. It was designed to support applications whose data index is maintained incrementally, such as shopping online application. \textit{HIndex} stores data in a Distributed File System (DFS). Storage model used by \textit{HIndex} is similar to that of the \textit{BigTable} tablet. \textit{HIndex} partitions the index, and distributes partitions on the cluster nodes. Partitioning is done by adapting the \textit{Partition By Document approach} (PBD). In PBD, each node serves a subset of documents. posting list in a node contains only documents in that node. Using such partitioning approach may incur an overhead for query search because of the broadcasting a given query to all nodes. This overhead could be avoided by exploiting the the advantage of the sorting property of \textit{BigTable} and carefully designing the row keys. With this property, it is possible to just forward the query to the node that contains the data.

The approach presented in [KATK10b] has a fully distributed architecture. Indexing using \textit{MapReduce}, and storing using the \textit{HBase} table, but Ivory [LMEW] and \textit{HIndex} [LRST09] approaches distribute only on part of the process, so it achieves significant reduction in time needed for both indexing and query response.
4 Scalable Distributed Time-Travel Text Search

In this chapter, we present our work on building a distributed time-travel text search, which supports Boolean queries and keyword queries combined with time-point or time-interval. We discuss our model of versioned document collection, our query model, and the retrieval models in Section 4.1. Then, we present schemas that we devised to store indexed data in HBase in Section 4.2. After that, we discuss our approaches for indexing versioned document collections in Section 4.3. Finally, we discuss how we process time-travel queries in Section 4.4.

4.1 Model

4.1.1 Data Model

In our work, we use a versioned document collection. Each document in the collection may have more than one version. When we build the inverted index, we keep a posting for every document version. The timestamp associated with every version is the document version timestamp. We do not index the valid-time interval for versions since we handle this at the query processing time while retrieving relevant documents for the given query. If there is only one version of a document $d$ at timestamp $t_i$, then this document is valid from its creation time until now $d^{t_i,[now]}$. In case we have a newer version of $d$ at timestamp $t_j$, then valid-time interval of $d^{t_i}$ is $[t_i, t_j)$, and valid-time interval of $d^{t_j}$ is $[t_j, now]$. 
4 Scalable Distributed Time-Travel Text Search

4.1.2 Query Model

Our inverted index supports answering both time-point queries \( q^{ts} \) and time-interval queries \( q^{[tb,te]} \), where \( ts \) is the timestamp of the given query, \( tb \) and \( te \) are the begin and end boundaries of the query, respectively. Query \( q \) can be a Boolean query or a keyword query.

\[
\text{Model: } \left\{ \text{Boolean keyword} \right\} \times \left\{ \text{time-point time-interval} \right\}
\]

4.1.3 Retrieval Model

For answering Boolean queries, we use the Boolean retrieval model. For keyword queries, we use the tf-idf model that we discussed in Section 2.3.1 to score retrieved documents relevant to a given query.

4.2 Data Layout

We use HBase for storing indexed data. HBase is tightly coupled with Hadoop. It uses the Hadoop Distributed File System (HDFS) as a storage and provides I/O hooks to Hadoop’s MapReduce. We exploit the features and data representation model in HBase that we introduced in Section 2.2 to devise different schemas for storing indexed data in HBase tables, which are called index tables. The differences between these schemas are summarized in the following points:

1. Selection of row key.
2. Column family.
3. Column qualifiers.
4. Timestamps. Each cell is either timestamped implicitly by the system or explicitly by the user.

In HBase, each cell has a triple key \(<\text{row key, column, timestamp}>\). User can specify the timestamp when insert in HBase table (explicit) timestamp, but if
the user does not specify it, then *HBase* will assign a timestamp equal to time of the machine at insertion time (*implicit*). Our schemas use explicit timestamp, since we are dealing with versioned document collections and we need the real timestamps attached with these versions. For some schemas that we have we do not use the time dimension provided by *HBase* and we compose the timestamp in the row key or column name (*explicit*). However, *HBase* will timestamped inserted cells and we just ignore them.

In the following, the name of the schema starts with the row key, followed by an underscore, then the column family name. Table 4.1 shows the shortcuts notations we use in naming our approaches.

<table>
<thead>
<tr>
<th>Tid</th>
<th>Term identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Did</td>
<td>Document identifier</td>
</tr>
<tr>
<td>Tf</td>
<td>Term frequency</td>
</tr>
<tr>
<td>Ts</td>
<td>Timestamp</td>
</tr>
</tbody>
</table>

Table 4.1: Shortcuts notations

For all schemas, table name and column family have to be specified when creating the table, but the column qualifiers are created when they are needed during data insertion.

Next we discuss our schemas of the *index* tables. We illustrate the data representation in *index* table using different schemas using the following example data that captures the occurrence of the term *cat*.

<table>
<thead>
<tr>
<th>document id</th>
<th>document timestamp</th>
<th>term frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_{12}$</td>
<td>$t_{s1}$</td>
<td>6</td>
</tr>
<tr>
<td>$d_{12}$</td>
<td>$t_{s5}$</td>
<td>9</td>
</tr>
<tr>
<td>$d_{13}$</td>
<td>$t_{s4}$</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4.2: Documents versions that contain term *cat*
4.2.1 Tid_Did

For this schema, we use an integer term \textit{id} as a row key. We use one column family named \textit{Did}. In this column family we add as many columns as needed. Each column in the column family represents a document \textit{id} which contains the term in the corresponding row key. Each cell holds the term frequency of the corresponding row key and can contain multiple versions under different timestamps. We use explicit timestamps stored in version dimension of \textit{HBase} table which represent timestamps of document versions. According to this schema, a row in the \textit{index} table contains all information related to one term. Documents that contain a given term can be identified as follows. We first retrieve the row corresponding to the given term from the table. The documents are then obtained as the names of columns in the \textit{Did} column family whose value are not NULL. Figure 4.1 shows the data representation of this schema using \textit{cat} information.

![Diagram of Tid_Did index table]

Figure 4.1: Data representation in Tid_Did index table.

4.2.2 Tid_DidTs

This schema also uses term \textit{id} as a row key. The column family name used here is \textit{DidTs}, which means that column qualifiers added under this column family will be composed of a document \textit{id} and a timestamp of the document version
that contains the term. We do not need HBase timestamps because we combine
the document version timestamp with the document id in the column qualifier.
Columns in the column family represent the document’s versions. If there are
multiple versions of a document, then number of column qualifiers will be larger
than number of columns in the Tid_Did schema. However, if there is one version
per document, then the number of columns will be the same as in the Tid_Did.
Figure 4.2 shows the data representation of this schema using cat information.

![Figure 4.2: Data representation in Tid_DidTs index table.](image)

### 4.2.3 TidDid_Ts

For this schema, we use a row key which is a composite of term id and docu-
ment id. The column family is Ts. Columns in the Ts column family are the
timestamps of a document versions. According to this schema, documents that
contain a given term will be found in contiguous rows. Since we store terms ids,
we can retrieve all documents by applying a range scan on the index table and
this range scan starts from the term id of a given term, and stops at the term
id plus one. Figure 4.3 shows the data representation of this schema using cat
information.
4 Scalable Distributed Time-Travel Text Search

4.2.4 TidDidTs_Tf

This schema uses a composite key, which is composed of term id, document id, and timestamp. This way the information about each term is encoded in the row key. The column family is tf, which contains only one column qualifier, and the cell value is the term frequency. We can obtain information about a given term in a specific document at a certain timestamp by composing the term identifier, the document identifier, and the timestamp together and scan the index table for this value. Figure 4.4 shows the data representation of this schema using cat information.

Figure 4.3: Data representation in TidDidTs index table.

Figure 4.4: Data representation in TidDidTs.tf index table.
4.2 Data Layout

4.2.5 TidTs_Did

This schema uses a composite row key of term id and timestamp. The column family is Did, which contains columns qualifiers referred to documents ids. We use this schema for time partitioning. In all previous approaches we keep an entry for each document version in the index table. However, for this schema, we keep an entry for each month between the two timestamps. We use the month granularity and round the timestamps of the documents to the beginning of closest month. We have a term cat in two versions of document d_{12} at two timestamp ts_1 and ts_5. The valid-time interval of the first version is [ts_1, ts_5). Thus, for each time-point in this interval (ts_1, ts_2, ts_3, and ts_4) we insert an entry for cat in index table. Figure 4.5 shows the data representation of this schema using cat information. This partitioning strategy is similar to partitioning strategies in [BBNW07] that we discussed in Section 3.1. We are keeping an entry for each time-point in the life-time interval of the document version which contains term t the idea is similar to \( P_{opt} \) in [BBNW07]. If we increase the time granularity for partitioning, then we insert less entries in the index table. If we keep only one entry for the whole life-time interval of the document which contains t, then this similar to \( S_{opt} \) in [BBNW07].

![Data representation in TidTs_Did index table.](image)

Figure 4.5: Data representation in TidTs_Did index table.
4 Scalable Distributed Time-Travel Text Search

4.3 Indexing

We use Hadoop MapReduce to build a distributed inverted index. Our indexing algorithm is a MapReduce task, which consists of two phases map and reduce. In the map phase, mappers read the collection data from the HDFS, and emit \(<\text{term}, \text{term} - \text{info} >\). The emitted key is a term, and the emitted value is composed of the id and the timestamp of the document containing the term along with the term frequency. In the reduce phase, reducers read the intermediate key-value pairs from the mappers, extract the information from the emitted values, then insert data into an HBase index table. We have developed five MapReduce indexing algorithms which use the same mapper, but different reducers. Since reducers use index table as data storage, and we have different schemas for index tables, then we have different implementations for reducers. In the following sections, mapper and reducers are discussed in detail.

4.3.1 Mapper

Mapper reads data splits from the HDFS, parses the contents and extracts necessary information for indexing from the input file. This information is the document id, the timestamp, and the content. The mappers tokenize the document contents and calculates the number of occurrences for all terms. For each term, they combine the id and the timestamp of the document which contain the term along with the term frequency. Finally, it emits \(<\text{term}, (\text{did}, \text{ts}, \text{tf}) >\) pairs. The number of emitted pairs for each term depends on the number of documents and the number of versions of these documents that contain the term. Algorithm 4.1 gives the pseudo-code representation for the mapper.

4.3.2 Reducers

Reducers build the inverted index and store it in HBase tables. Reducers read the intermediate key-value pairs from the mappers, replace each term with an integer id, build the postings list for each term, and then insert postings into index table. Reducers insert data into two HBase tables; the lexicon table which contains the
Algorithm 4.1: Pseudo-code for indexing mapper

Input: document collection
Output: \(<\text{term}, (\text{did}, \text{ts}, \text{tf})>\) pairs

1 begin
2     parse input split
3     for each document in the split do
4         extract document id: did
5         extract document timestamp: ts
6         extract document’s content
7         normalize content (e.g., convert to lower case)
8         tokens = Tokenize(content)
9         for each term \(\in\) tokens do
10            compute the term frequency(tf)
11            combine \(<\text{did}, \text{ts}, \text{tf}>\)
12            emit(term, \(<\text{did}, \text{ts}, \text{tf}>\))
13     end
14 end

terms and their corresponding ids, and the index table which keeps the inverted index. Data in two tables using the same reducers. So that our algorithms do indexing in one-single pass. Reducers implement the TableReducer interface which takes the responsibility of inserting emitted key-value pairs in the HBase table, the key is the row key and the value is put statements to insert data in HBase. Using TableReducer alone is not sufficient, since we can only write in one HBase table while we need to insert data in two tables. Using MultiTableOutputFormat allows reducers to make two different emits. The key is the name of the table, and the value is the put instance. The multiple reducers result from the different index table schemas. Reducers take the responsibility to insert the inverted index into the index table. Thus, the reducer’s implementation depends on the schema of the index table. Algorithm 4.2 gives pseudo-code for the reducer in general. The inputs for the reducers are the intermediate \(<\text{term}, (\text{did}, \text{ts}, \text{tf})>\) pairs from the mappers and the output is two types of key-value pairs, first type is the \(<\text{“LexiconTable”}, \text{putInLexicon}>\) pairs, where the key is the lexicon table name and values are the put instances each put corresponds to one term and
it’s id. The second type is the <“LexiconTable”, putInLexicon> pairs, where keys are the index’s table name and values are the put instances for inserting inverted index into the index table.

This algorithm works as follows: We have multiple reducers working in parallel, and we want to avoid collisions between terms ids and give a unique id to every term. Thus, each reducers sets a counter to its id (Line 2). For every new term, it increments by the total number of reducers and assigns value to term id. (Line 9-10), prepare the put instance, for which we need to specify the key that is in our case the term, and specify the column family (ID) and the column qualifier (ID), the value is the term id. (Line 11), emits the <“LexiconTable”, putInLexicon> pair. Building and inserting inverted index in the index table depends on the table’s schema. We use the buildAndStoreIX() function which takes as an input a set of values associated with a term. From each value in the set, it extracts the document id, the timestamp, and the term frequency to create the postings list for the term. The output of this function is the <“IndexTable”, PutInIndex> pairs, where IndexTable is the name of the used index table, the value is a put instance to insert data in the index table.

The first add uses an explicit timestamp, while the second uses an implicit one of HBase.

Reducers share all steps in Algorithm 4.2, but they differ in building and storing inverted index which is the work of the buildAndStoreIX() function. Next, we discuss the implementation of this function in different reducers. The reducers name is the same as the schema of table used by this reducers. E.g., Tid_DidReducer is the reducers which uses an index table whose schema is Tid_Did.

Tid_DidReducer

Algorithm 4.3 shows the pseudo-code of the buildAndStore() function in the Tid_DidReducer. This reducer uses Tid_Did index table to store the inverted index. Row key in index table is the term id. We instantiate one put for each term. Then, we extract the document id, the timestamp, and the term frequency
Algorithm 4.2: Pseudo-code for indexing reducer

| Input:   | <term, (Did, ts, tf)> intermediate pairs |
| Output:  | <“IndexTable”, putInIndex> pairs, <“LexiconTable”, putInLexicon> pairs |

1 begin
2     N = number of reducers
3     counter = reducer id
4     reduce (key, set of values)
5         key: term
6         values: all emitted values for a term
7     counter += N
8     termID = counter
9     Put putInLexicon = new Put(key)
10    putInLexicon.add(“ID”, “ID”, termID )
11    emit (<“LexiconTable”, putInLexicon >)
12    buildAndStoreIX(term values set)
13 end

from each value in the set of values for this term. For every extracted did, ts, and tf we perform an add to the put instance (Line 7). Finally, we perform one insert in index table. So this reducer emits once per term.

Tid_DidTsReducer

Algorithm 4.4 gives the pseudo-code of the buildAndStore() function in the Tid_DidTsReducer. This reducer uses Tid_DidTs index table to store the inverted index. It uses the term id as a row key, so we need one put instance to add all postings. In the end we perform one insert for each term.

TidDid_TsReducer

Algorithm 4.5 shows the pseudo-code of the buildAndStore() in the TidDid_TsReducer which uses TidDid_Ts index table. Here row key is a composite of the term id and the document id, so we can not instantiate a single put for a term. Given a set of values for a term for each value we extract the document id, the timestamp, and the term frequency. Then, we instantiate a put with row key composed of
4 Scalable Distributed Time-Travel Text Search

Algorithm 4.3: buildAndStoreIX() implementation in Tid_DidReducer

Input: term values set
Output: <“IndexTable”, putInIndex > pairs

begin
1 Put putInIndex = new Put(term id)
2 for each value ∈ values do
3 extract did
4 extract ts
5 extract tf
6 putInIndex.add(“Did”, did, ts, tf)
7 emit (<“IndexTable”, putInIndex >)
end

Algorithm 4.4: buildAndStoreIX() implementation in Tid_DidTsReducer

Input: term values set
Output: <“IndexTable”, putInIndex > pairs

begin
1 Put putInIndex = new Put(term id)
2 for each value ∈ values do
3 extract did
4 extract ts
5 extract tf
6 putInIndex.add(“DidTs”, Bytes.add(did, ts), tf)
7 emit (<“IndexTable”, putInIndex >)
end

the term id and the document id (Line 7). Then, we perform add to column family Ts, with column qualifier equal to the real timestamp of the document in the row key. The value is the term frequency (Line 8). Finally, we emit this put instance, so emitting is done for every value in the set of values of a term. We emit once per value (multiple emits for each term). We don not specify a timestamp, since we add the real timestamps in columns keys.
4.3 Indexing

Algorithm 4.5: buildAndStoreIX() implementation in TidDid_TsReducer

*Input*: term values set

*Output*: <“IndexTable”, putInIndex> pairs

1. begin
2.   for each value ∈ values do
3.     extract did
4.     extract ts
5.     extract tf
6.     Put putInIndex = new Put(Bytes.add(term id, did))
7.     putInIndex.add(“TS”, ts, tf)
8.     emit (<“IndexTable”, putInIndex>)
9. end

**TidDidTs_tfReducer**

Algorithm 4.6 shows the pseudo-code of the buildAndStore() function in the TidDidTs_tfReducer which uses the TidDidTs_tf to store the inverted index. Row key is a composite of the term id, the document id, and the timestamp. Therefore, we cannot instantiate a single put for a term. Given a set of values for a term; for each value we extract the document id, the timestamp, and the term frequency. Then, we instantiate a put with row key composed of the term id, the document id, and the timestamp. We perform add to a column tf:tf, the value is the term frequency (Line 8). Finally, we emit the put instance, so we emit once per every value in the set of values for a term. We don not specify a timestamp, since we add the real timestamps in the row keys.

**TidTs_DidReducer**

Algorithm 4.7 gives the pseudo-code of the buildAndStoreIX() in the TidTs_DidReducer which uses TidTs_Did schema to store inverted index. We use this reducer for time partitioning. This means, if we have two versions of the same document containing a term t, the first existed at time t_2 and the second existed at time t_5, then the first version’s valid-time interval is [t_2, t_5), so we insert an entry for every time-point in this interval for term t. Thus, we have an entry at time t_2, t_3,
4 Scalable Distributed Time-Travel Text Search

**Algorithm 4.6:** buildAndStoreIX() implementation in TidDidTs.tfReducer

- **Input:** term values set
- **Output:** \(<\text{"IndexTable"}, \text{putInIndex}>\) pairs

1. **begin**
2.  
3.  
4.  
5.  
6.  
7.  
8.  
9. **end**

and \(t_4\). We use month granularity for partitioning. We combine the timestamp with the term \(id\) in the row key, so that all versions of a document containing the term \(t\) are stored in contiguous rows. Algorithm 4.7 works as follows: Given a set of values for term \(t\), create a postings list, where a posting has \((\text{did, ts}, \text{tf})\) structure (Lines 4-8). Postings in the list are ordered by the document \(id\). We determine the valid-time intervals of all postings. Each posting represents a version of a document containing the term \(t\). If there is only one version at time \(t_i\), then the end-time of this version is \(\text{now}\) and its valid-time interval is \([t_i, \text{now})\). We insert only one entry for this version (Lines 24-26). Otherwise, the end-time of the version is the begin-time of the next version. For each time-point, we use month granularity between begin-time and end-time of a version to make an insert in the index table (Lines 16-22).

### 4.4 Query Processing

The main objective of our work is to be able to process time-travel queries. We prepare the inverted index which is supported by time functionality, and we store it in *HBase*. In processing the given queries, we exploit range scan on row keys which *HBase* supports efficiently. Now we discuss how we process time-travel queries using the distributed inverted index in *HBase*. The query \(q\) can either be
Algorithm 4.7: buildAndStoreIX() implementation in TidTs_DidReducer

Input: term values set
Output: <“IndexTable”, putInIndex > pairs

begin
List< posting(did, ts, tf) > termPostingList
for each value ∈ values set do
  extract did
  extract ts
  extract tf
  termPostingList.add(new posting(did, ts, tf))
sort termPostingList in ascending order by timestamp
for each posting ∈ termPostingList do
  $t_b$ = timestamp of the posting
  if posting.did = nextPosting.did then
    $t_e$ = next posting’s timestamp
  else
    $t_e$ = Long.MAX_VALUE
  if $t_e$ ≠ Long.MAX_VALUE then
    partitions :partition time between $t_b$ and $t_e$ using month granularity
    for each ts ∈ partitions do
      Put putInIndex = new Put(Bytes.add(term id, ts ))
      putInIndex.add(“Did”, did, tf)
      emit (< “IndexTable”, putInIndex >)
    else
      Put putInIndex = new Put(Bytes.add(term id, $t_b$ ))
      putInIndex.add(“Did”, did, tf)
      emit (< “IndexTable”, putInIndex >)
end
a Boolean query (e.g., mpi AND saarland) or a keyword query (e.g., mpi saarland).

4.4.1 Retrieving Postings List

Now we discuss how data is retrieved from the index table. Retrieving data depends on the index table schema. We use different schemas to exploit HBase’s data representation in order to obtain a reasonable query response time. When we access an index table to retrieve relevant documents for a given query term, we retrieve all information needed to build the postings list. Recall that our posting’s structure is (did, ts, tf). HBase supports two lookup operations on the row key; exact match and range scan using the get and the scan objects, respectively. In our work, we use scan to retrieve data from the index table. Using scan, we can specify the start row key, the end row key, the column family, and the maximum number of versions to return. Next, we discuss the different methods to retrieve data from the index table. We name the used methods after the corresponding index table schema. For example Tid_Did, is the method which retrieves data from index table whose row key is the term id, and has Did as the column family.

Tid_Did

This method retrieves data from the Tid_Did index table. To build the postings for the relevant documents to a given query term, we need the document id, which is located in the Did column family which contains ids of the documents as column keys. The timestamp is the HBase timestamp, and the term frequency is the cell value. All this information is retrieved by fetching the row key which is equal to the given term id. We use the scan object to get data from the index table. The Scanner starts scanning from the row key equal to the given term id, and stops at the row key equal to this term id plus one. We build a three dimensional map on the returned result from the scanner to extract needed information. The map is <cf, map <cq, map <ts, value >>>. In the first map, the key is the column family Did, the value is the second map. In the second map, the key is the column qualifier (document ids), the value is the
third map. In the third map, the key is the timestamp, and value is the index’s cell value (tf). Algorithm 4.8 gives a pseudo-code for retrieving postings list from the Tid_Did index.

*Tid_DidTs*

This method uses the Tid_DidTs index table for processing time-travel queries. For the given term, we scan the index table from the term id (Tid), and stop at the row key equal to this term id plus one (Tid+1). The document id and its timestamp forms the column qualifier. The term frequency is the cell value. Thus using a simple map on the returned result from the scanner, we obtain all information that we need. We scan the whole column family DidTs which contains all did-ts combinations for the corresponding term at the row key. Algorithm 4.9 gives a pseudo-code for retrieving postings list from the Tid_DidTs index table.

*TidDidTs*

This method scans the TidDidTs index table. The document id is part of the row key, timestamps are columns in the TS column family. We need a map on the column family which maps timestamps to the term frequency in the cell value. The document id is extracted from the row key. Algorithm 4.10 gives pseudo-code for retrieving postings list from the TidDidTs index table.

*TidDidTs_tf*

This method scans the TidDidTs_tf which has row keys composed from term ids, document ids, and timestamps. It uses one column in the column family tf. The cell value is the term frequency of the term in the document version in the corresponding row key. For this method, we do not need any map on the scan result. The document id and the timestamp can be extracted from the row key, and the value of the result is the term frequency. Algorithm 4.11 gives pseudo-code for retrieving postings list from the TidDidTs_tf index table.
Algorithm 4.8: Pseudo-code for retrieving postings list from Tid_Did index

Input: Term id \( Tid \)
Output: postingslist

begin

postingsList: List< posting >

instantiate scan object

\( scan.addFamily(\text{Did}) \)

\( scan.setStartRow(Tid) \)

\( scan.setStopRow(Tid + 1) \)

create scanner on the index table using scan object

for each result \( r \in \text{scanner results} \) do

three level map

NavigableMap < byte[], NavigableMap < byte[], NavigableMap < Long, byte[] >> > map1 = r.getMap()

iterate over the first level of the map

for each entry1 \( \in \text{map1.entrySet()} \) do

key: is the column family
value: the second map

NavigableMap < byte[], NavigableMap < Long, byte[] >> > map2 = entry1.getValue()

iterate over the second level the map

for each entry2 \( \in \text{map2.entrySet()} \) do

key: column qualifier, represent document id’s

did: entry1.getKey()
value: the third map

NavigableMap< Long, byte[] > map3 = entry2.getValue()

iterate over the third level the map

for each entry3 \( \in \text{map3.entrySet()} \) do

timeStamp: entry3.getKey()
tf: entry3.getValue()

postingsList.add(new posting(did, ts, tf))

end
Algorithm 4.9: Pseudo-code for retrieving postings list from Tid_DidTs index

Input: Term id Tid
Output: postingslist

begin
  postingsList: List<posting>
  instantiate scan object
  scan.addFamily(“Did”)
  scan.setStartRow(Tid)
  scan.setStopRow(Tid + 1)
  create scanner on the index table using scan object
  for each result r ∈ scanner results do
    get the map on “DidTs:” column family
    NavigableMap< byte[], byte[] > familyMap =
    r.getFamilyMap(“DidTs”);
    key: column qualifier(composite of did and ts)
    value: term frequency
    for each entry ∈ familyMap do
      cq = entry.getKey()
      tf = entry.getValue()
      first 4 bytes from cq represents document id
      did: extract document id from cq
      last 8 bytes from cq represents timestamp
      ts: extract timestamp from cq
      postingsList.add(new posting(did, ts, tf))
  end
end

TidTs_Did

This method scans the TidTs_Did index table which we have designed to store partitioned document versions using month granularity. In all previous methods we set scan object to start scanning from the row key equal to a given term id (Tid) and to stop at (Tid+1). In this method we start the scan from the row key equal to a composite of the term id and the begin-time of the query rounded to previous month granularity, i.e., (Tid − q\text{tb}). We stop at the row key equal to a composite of the term id and the end-time rounded to the next month granularity, i.e., (Tid − q\text{te}). Algorithm 4.12 gives pseudo-code for retrieving
Algorithm 4.10: Pseudo-code for retrieving postings list from TidDid Ts-Ts index

Input: Term id Tid
Output: postingslist

begin
postingsList: List<posting >
instantiate scan object
scan.addFamily("Did")
scan.setStartRow(Tid)
scan.setStopRow(Tid + 1)
create scanner on the index table using scan object
for each result r ∈ scanner results do
rowKey: r.getRowKey()
did: extract document id from the rowkey
get the map on “TS:” column family
NavigableMap< byte[], byte[] > familyMap = r.getFamilyMap("TS"); key:column qualifier(ts)
value: term frequency(tf)
for each entry ∈ familyMap do
    ts: entry.getKey()
    tf: entry.getValue()
    postingsList.add(new posting(did, ts, tf))
end

postings list from the TidTs_Did index table.
After retrieving postings list for each term in the query, we need to filter out irrelevant postings from the postings list. Next, we discuss how to filter out irrelevant postings for time-point queries and time-interval queries.

Time-Point Query Processing Algorithm 4.13 shows the pseudo-code for sorting and filtering out irrelevant postings for time-point queries. We refer to the postings list which contains all postings relevant and irrelevant as the candidate postings list. We sort the candidate postings list by the document id and the timestamp in ascending order for approaches that need sorting, some of the approaches we have returned data sorted, e.g., TidDidTs tf. This way of sorting
Algorithm 4.11: Pseudo-code for retrieving postings list from TidDidTs tf index

Input: Term id Tid
Output: postingslist

begin
1   postingsList: List< posting >
2       instantiate scan object
3       specify column to scan, cf:cq
4       scan.addColumn("tf:tf")
5       scan.setStartRow(Tid)
6       scan.setStopRow(Tid + 1)
7       create scanner on the index table using scan object
8       for each result r ∈ scanner results do
9         document id and timestamp are part of the row key
10     row key is 16 bytes length
11     first 4 bytes forms the term id
12     second 4 bytes represents the document id
13     last 8 bytes represents the timestamp
14     rowKey: r.getRowKey()
15     did: extract document id from the rowkey
16     ts: extract timestamp from rowkey
17     term frequency is the value of r
18     tf: r.value()
19     postingsList.add(new posting(did, ts, tf))
20   end
end

is necessary to have the versions of the documents ordered by their timestamps, such that we can easily decide the valid-time intervals for versions. We modify the posting by adding the end-time boundary, thus the new structure of the posting will be (did, tk, tl, tf). From the candidate postings list, we keep postings whose valid-time intervals contain the given query’s timestamp ts, such that ts ∈ [tk, tl].

Time-Interval Query Processing  Algorithm 4.14 shows the pseudo-code for sorting and filtering out irrelevant postings for the time-interval queries. We sort candidate postingslist, and determine the valid-time intervals for the versions
Algorithm 4.12: Pseudo-code for retrieving postings list from TidTs_Did index

**Input**: Term id Tid

**Output**: postings list

begin

1. postingsList: List< posting >
2. instantiate scan object
3. scan.addFamily("Did")
4. scan.setStartRow(tid-q^b)
5. scan.setStopRow(tid-q^e)
6. create scanner on the index table using scan object
7. for each result r ∈ scanner results do
   8. timestamp is part of the row key
   9. row key is 12 bytes length
      first 4 bytes forms the term id
      last 8 bytes represents the timestamp
   10. rowKey: r.getRowKey()
   11. ts: extract the timestamp from the rowkey
   12. we need a map which maps column qualifiers(cq) to values (tf)
      key: cqs represent document ids
   13. NavigableMap< byte[], byte[] > familyMap =
      r.getFamilyMap("Did");
   14. for each entry ∈ familyMap.entrySet() do
      15. did: entry.getKey()
      16. tf: entry.getValue()
      17. postingsList.add(new posting(did, ts, tf))

end
Algorithm 4.13: Pseudo-code for processing time-point query

| Input: timestamp of the given query ts, and candidate postings list |
| Output: list of relevant postings list |

1 begin
2  List result
3  sort candidate by did and ts
4  for each posting ∈ candidate do
5    determine valid-time interval for every posting
6    modify posting(did, ts, tf) → posting(did, tk, tl, tf)
7  for each posting ∈ candidate do
8    if (ts ∈ [tk, tl)) then
9      result.add(posting)
10  return result
11 end

of the documents as we have done for the time-point queries. The relevant document version is the one whose valid-time interval overlaps with the given query time-interval, i.e., \{te >= tk | te <= tl\}.

4.4.2 Boolean Queries

Given a time-travel query, for each term in the query we retrieve the postings list of the relevant document versions. We use the term-at-a-time processing, meaning that we retrieve the postings list for the first term in the query, add the result items to a hash map, and iterate until the last term in the query. Keys in this hash map are compound of the document id and the timestamp. Thus it maintains one item for each document version. The values in the hash map are the postings (did, tb, te). For the evaluation of disjunctive Boolean queries, the relevant document version is the one which contains any of the query terms, and whose valid-time interval contains query timestamp (for time-point query) or overlaps with query time-interval (for time-interval query). Algorithm 4.15 gives a pseudo-code for the term-at-a-time processing of disjunctive Boolean queries. In (Line 7) a postings list is retrieved using one of the algorithms described in Section 4.4.1. The used algorithm depends on the schema of the index table.
Algorithm 4.14: Pseudo-code for processing time-interval query

**Input:** time boundaries of the given query \([t_b, t_e]\), candidate postings list

**Output:** list of relevant postings list

```
begin
List result
sort candidate by document id and timestamp
for each posting ∈ candidate do
  determine valid-time interval for every posting
  modify posting(did, ts, tf) → posting(did, t_k, t_l, tf)
for each posting ∈ candidate do
  if \((t_e ≥ t_k \&\& t_e ≤ t_l)\) then
    result.add(posting)
return result
end
```

4.4.3 Keyword Queries

The only difference between processing of a Boolean query and a keyword query, is the scoring. For keyword queries, while we process its terms one at a time, we accumulate the score for each candidate posting. Algorithm 4.16 shows the term-at-a-time processing for the time-travel keyword query. (Line 8), get postings list using one of the algorithms for retrieving postings list from index table that we discussed in Section 4.4.1. The algorithm employed depends on the schema of the index table.
Algorithm 4.15: Term-at-a-time processing for disjunctive Boolean queries

**Input:** time-travel query  
**Output:** list of relevant postings

```
begin  
  ResultItem: (did, tb, te)  
  ResultItemKey: (posting.did, posting.tb)  
  List< ResultItem > result  
  HashMap< ResultItemKey, ResultItem > candidates  
  for each term t ∈ the given query do  
    get postings list for t  
    for each posting (did, tb, te) ∈ postings list for t do  
      key = new ResultItemKey (posting.did, posting.tb)  
      if key /∈ candidates then  
        candidate = new ResultItem  
        (posting.did, posting.tb, posting.te)  
        candidates.put(key, candidate)  
      after finishing all query terms  
      for each ResultItem item ∈ candidates.values() do  
        result.add(item)  
    return result  
end
```
Algorithm 4.16: Term-at-a-time processing for keyword queries

**Input:** time-travel query

**Output:** list of relevant scored postings

begin

1. ResultItem: \((did, t_b, t_e, score)\)
2. ResultItemKey: \((posting.did, posting.t_b)\)
3. List\(< ResultItem >\) result
4. HashMap\(< ResultItemKey, ResultItem >\) candidates
5. get the collection size

   for each term \(t\) \in the given query do

   6. get postings list for \(t\)

   7. for each posting \((did, t_b, t_e)\) \\in postings list for \(t\) do

   8. key = new ResultItemKey\((posting.did, posting.t_b)\)

   9. if key \\notin candidates then

      10. candidate = new ResultItem\((posting.did, posting.t_b, posting.t_e, posting.score)\)

      11. candidates.put(key, candidate)

      12. get the document length of did in the new added posting

      13. update candidate’s score

      14. new score =\((\text{tf-idf})\) of new coming term which exist in candidate’s did

      15. candidate.score += new score

   after finishing all query terms

   16. for each ResultItem item \\in candidates.values() do

   17. result.add(item)

   18. sort result by score

   19. return result

end
5 Prototype Implementation

In this section, we describe the prototype implementation of our search engine, which uses a distributed inverted index stored in index table to return relevant documents for a given query. Index table is an HBase table designed using one of the schemas that we discussed in Section 4.2. Section 5.1 gives a brief view of the architecture of the system. Section 5.2 presents the Hadoop setup. Section 5.3 describes HBase setup, and how to create HBase tables that we used to store the inverted index. Section 5.4 describes the implementation of the indexing algorithm. Finally, Section 5.5 describes the implementation of query processing.

5.1 System Architecture

Figure 5.1 shows a high-level view of the architecture of our system. It is composed of the following components:

1. Hadoop/MapReduce, we discuss the installation and the configuration of Hadoop in Section 5.2 in detail.

2. Hadoop/HDFS is used to store our document collections. It is used by HBase as a storage backend.

3. HBase is used to keep the inverted index in the index as well as the lexicon table. Section 5.3 presents the installation and the configuration of HBase.

4. Indexing, is responsible for building an inverted index of the input document collection using a MapReduce algorithms which we discussed in
Section 4.3 in detail. Section 5.4 presents the implementation of this component in detail.

5. *Query processing*, is responsible for processing Boolean queries and keyword queries combined with time-point or time-interval. In Section 4.4 we discussed the processing of a given time-travel query. Section 5.5 gives the implementation of this component.

6. *document collection*, which represents the input to the *indexing* component.

---

**5.2 Hadoop Setup**

The Apache *Hadoop* project is an open-source project, which develops an open-source software for reliable, scalable, and distributed computing. It includes the following subprojects:

---
5.2 Hadoop Setup

- **Hadoop Commons** including FileSystem, Remote Procedural Call (RPC), and serialization libraries.

- **Hadoop Distributed File System (HDFS):** Hadoop applications use HDFS as a primary storage for the input and output.

- **Hadoop MapReduce:** A framework for distributed processing of large datasets on clusters.

Hadoop can be configured to run in *non-distributed* or *distributed* modes. For *non-distributed* modes, Hadoop is configured on a single-node as a single Java process (*Standalone mode*) which is the Hadoop default mode, or run on a single-node where each Hadoop daemon runs in a separate Java process (*Pseudo distributed mode*). In distributed modes, Hadoop is configured to run on a cluster (*Fully distributed*). To run Hadoop on Linux and Windows, it is required to have Java\textsuperscript{TM} 1.6.x and ssh installed on the machine. Additional requirement for Windows is the installation of Cygwin. A stable release of Hadoop can be downloaded from Hadoop website [had]. The first step is to download a Hadoop stable release. Then, unpack the downloaded release. Hadoop environment variables are defined in the `conf/hadoop-env.sh` which is found in the unpacked distribution folder. The important thing to be defined in this file is the `JAVA_HOME` to inform Hadoop where Java is located. Configuration of Hadoop is made using the following XML files:

- `conf/core-site.xml` contains core properties.
- `conf/hdfs-site.xml` contains the HDFS properties.
- `conf/mapred-site.xml` contains MapReduce properties.

In earlier versions of Hadoop before Hadoop-0.20.0 release, instead of these files, there was only one configuration file called `hadoop-site.xml`. Table 5.1 shows the main properties need to be configured for Pseudo-distributed and Fully-distributed modes. Standalone mode configurations are not included as they are the default.
5 Prototype Implementation

<table>
<thead>
<tr>
<th>configuration file</th>
<th>property</th>
<th>Pseudo-distributed</th>
<th>Fully-distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>core-site.xml</td>
<td>fs.default.name</td>
<td>hdfs://localhost/</td>
<td>hdfs://namenode/</td>
</tr>
<tr>
<td>hdfs-site.xml</td>
<td>dfs.replication</td>
<td>1</td>
<td>default is 3</td>
</tr>
<tr>
<td>mapred-site.xml</td>
<td>mapred.job.tracker</td>
<td>localhost:port#</td>
<td>jobtracker:port#</td>
</tr>
</tbody>
</table>

Table 5.1: Configuration properties for Pseudo and Fully-distributed modes

After downloading and configuring Hadoop we run it by starting Hadoop daemons:
$bin/start-all.sh, which starts the HDFS and MapReduce.

The input of the MapReduce job should be in the HDFS. To copy an input file into the distributed filesystem we can use the put command:
$bin/hadoop dfs -put inputfile location

To copy files from the distributed filesystem, we use the get command:
$bin/hadoop dfs -get filename localLocation

Hadoop daemons can be stopped using:
$bin/stop-all.sh

5.3 HBase Setup

Documentations and HBase versions are found in the HBase website [hba]. We download the HBase stable release from the list of Apache Downloads Mirrors. Then, we decompress the downloaded release into an unpacked directory. HBase can be configured using conf/hbase-site.xml. One of the configurations is to inform HBase about the directory to use for writting. After that, HBase is ready to be used. We start HBase with:
$./bin/start-hbase.sh

We connect to the running HBase via the shell
$./bin/hbase shell

We create a table, insert data in it, scan it, and get data from it.
5.3.1 HBase Tables

HBase allows HBase client to connect to HBase master using an instance of HBaseAdmin class. HBaseAdmin needs an instance of HBaseConfiguration class. The HBase client can create, modify, and delete tables and column families. Column families must be specified when creating the table. To create a column family, an instance of HColumnDescriptor is used. For example, Figure 5.2 shows how to create the Tid_Did index table we discussed in Section 4.2.1. We create an instance of HBaseConfiguration (Line 1) required for HBaseAdmin (Line 2). Then, we make an instance of HTableDescriptor which takes the name of table as an input (Line 3). Later, we use the HColumnDescriptor to make an instance of the column family (Line 4). Next, we add the instantiated column family to the instantiated HTableDescriptor (Line 5). At last, we create the table (Line 6).

```
Line1: Configuration conf = HBaseConfiguration.create()
Line2: HBaseAdmin hbase = new HBaseAdmin(conf)
Line3: HTableDescriptor desc = new HTableDescriptor("Tid_Did")
Line4: HColumnDescriptor columnFamily = HColumnDescriptor("Did")
Line5: desc.addFamily(columnFamily)
Line6: hbase.createTable(desc)
```

Figure 5.2: Create HBase table Tid_Did with one column family Did

5.4 Indexing

Figure 5.3 shows the data flow during the indexing process. It consists of the following components:

1. *Document collection*, which represents the input data to be indexed.

2. *Mapper*, which is responsible for parsing and analyzing input document collection. A mapper is composed of implementations of three components: inputFormat, parser, and the analysis component. 5.4.1 presents them in detail.
5 Prototype Implementation

3. **Reducer**, which is responsible for building the *inverted index* and storing it in the *index* table. We have five different implementations for the reducer, we discussed them in Section 4.3.2.

4. **Index table**, where *inverted index* is kept.

5. **Lexicon table**, where terms with their corresponding ids are kept.

![Figure 5.3: Data flow of indexing](image)

Recall that in Section 4.3 we mentioned that we have five *indexing* algorithms. Each one of them is a MapReduce, consists of *mapper* and *reducer*. We also need a driver class. In which, we add the job configurations such as, an input file, an output file, a *mapper* class, a *reducer* class and other necessary configurations. Figure 5.4 shows the class diagram of *indexing*.

**5.4.1 Mappers**

Figure 5.5 shows the data flow of the *mapper*. It is composed of the following components:

1. *Data collection* which is stored in the *HDFS*. We use two document collections, the *New York Times Annotated Corpus (NYT)* and the *Revision...*
5.4 Indexing

Figure 5.4: Class diagram of indexing
5 Prototype Implementation

*History of English Wikipedia WIKI.* We discuss both collections in Section 6.2.

2. **InputFormat** is responsible for deciding the key-value pairs format. For *NYT* collection we use the *TextInputFormat* which breaks plain text files into lines, using carriage-return to signal end of line. The output key-value pairs form the inputs for the *mapper*. Keys are the positions of the file’s lines. Values are the text contents in lines. For *WIKI*, we use an *XMLInputFormat* which extends the *Hadoop* InputFormat. It breaks the input XML file into blocks. The block boundaries depend on the specified start-tag and end-tag. These blocks represent the input values for the map function. For example, if the start-tag and end-tag are `<page>` and `</page>`, then the input values for the map function are the *WIKI* pages with all revisions for each page.

3. **Parser** is responsible for obtaining the following information from each input document for the map function: document *id*, timestamp, and the content itself. For *NYT*, the input values for the map function are lines, where each line represents a document from the *NYT*. Each line consists of a document *id*, a timestamp, and content; they are separated by a tab. In the *WIKI*, the input values to the map function are the pages, where each page may contain multiple revisions. The revisions are the documents. The document *id* is in the `<id>`, `</id>` tags. The document’s timestamp is between the `<timestamp>`, `</timestamp>` tags. Finally, the content is between the `<text>`, `</text>` tags. The parser for *WIKI* document uses regular expressions to extract these information.

4. **Data analysis** tokenizes the content of the document and counts the frequency of terms. An *ImmutableBytesWritable* is composed from: a document *id*, a timestamp, and a term frequency of each term in the document. Document *id* is a 4 bytes integer. Timestamp is an 8 bytes long while term frequency is a 4 bytes integer.
5.4 Indexing

5. Intermediate key-value pairs which are the output of the map functions. Keys are terms, and values are the composed values from component 4.

![Data flow of the mapper](image1)

**Figure 5.5: Data flow of the mapper**

5.4.2 Reducers

Figure 5.6 shows the data flow in the reducers. It is composed of the following components:

1. Intermediate key-value pairs which are the mapper’s output.

2. Build and store inverted index is responsible for building the inverted index and storing it in the index table. We discussed different implementations of this component in Section 4.3.2.

3. Index table and lexicon table are the outputs of the indexing process. They are located in the HDFS.

![Data flow of the reducer](image2)

**Figure 5.6: Data flow of the reducer**
5 Prototype Implementation

5.5 Query Processing

Figure 5.7 shows a high-level view of the architecture of query processing. It is composed of the following main components:

1. **Query parser** parses the given query. For each term in the query, the parser obtains the term *id*, the begin-time, and end-time of the query. The parser forms an object from the obtained data.

2. **Query processor** is responsible for answering the given query. It picks the query object from the query parser. For each term in the query, the processor retrieves all candidate *postings* which contain this term from the *index* table. Any retrieved *posting* is relevant if its time-interval contains the query’s time-point or overlaps with the query’s time-interval for point-of-time queries and interval-of-time queries, respectively. For keyword queries, the processor calculates the scores of retrieved documents. Finally, it returns the list of relevant documents to the given query. We discussed query processing in Section 4.4. Figure 5.8 shows a class diagram of query processing.

![Query processing architecture diagram](image)

Figure 5.7: Query processing architecture
Figure 5.8: Query processor class diagram
6 Experimental Evaluation

This chapter presents an experimental evaluation of our approaches to build a
scalable distributed time-travel text search. In Chapter 4 we discussed our ap-
proaches for indexing versioned document collections using different schemas of
index table. We examine these different approaches with regard to index size,
indexing performance, and query-processing performance on two real-world ver-
sioned document collections. Section 6.1 presents hardware and software spec-
fications of the cluster that we use for experiments. Section 6.2 describes the
real-world versioned document collection that we run our experiments on. Sec-
tion 6.3 presents the results of the different approaches.

6.1 Setup

6.1.1 Hardware Specifications

Our cluster contains 10 machines. 1 master node (Dell PowerEdge 410, 2x Intel
Xeon X5650 6-Core CPU at 2.66Ghz, 64GB RAM and 2x2TB disk configured as
RAID-1) and 9 slave nodes (Dell PowerEdge 410, 2x Intel Xeon X5650 6-Core
CPU at 2.66 Ghz, 64GB RAM and 4x2TB disk configured as Bunch Of Disks
BOD). The machines are in a single rack and connected via 10Gbit Ethernet.

6.1.2 Software Specifications

We used Cloudera CDH3 that is based on Hadoop 0.20.2 and HBase 0.90.1. Ev-
ery slave node is configured to run at most ten map tasks and ten reduce tasks
6 Experimental Evaluation

in parallel. The replication level of the distributed file system is set to 2.

6.2 Data sets

New York Times Annotated Corpus (NYT)  It contains a total of 1,830,592 documents and its size is 5.68 GB. The documents are articles published by New York Times in the period between January 1, 1987 and June 19, 2007. NYT contains one version per document. The valid-time interval of each document ranges from publication time until the current time now. We used the maximum long value to represent the current time.

Revision History of English Wikipedia (WIKI)  It can be freely downloaded as a single XML file. It contains articles from 2001 to 2005. Documents in the XML file are pages. Each page has different revisions at different timestamps. These revisions are the multiple versions of the document. The begin-time of the time-interval for each document version is the creation time, while the end-time is the creation time of a newer version. Figure 6.1 shows a sketch of the Wikipedia page structure.

In our experiments, we use a sample from the WIKI. The sample consists of 64 compressed text files stored as a sequence file. Each file contains multiple articles and each article consists of multiple revisions. In total, the sample contains 968,832 articles and 3,200,000 versions. The size of the sample is 58 GB, while the size of the original collection is 280 GB.

6.3 Results and Comparisons

Now we present an experimental evaluation of our distributed inverted index which supports time-travel text search. We designed four approaches for indexing, storing, and retrieving data. We compare them with regarding to indexing performance, index size, and query-processing performance.
6.3 Results and Comparisons

Figure 6.1: Wikipedia page structure. For each version, the revision begin and end tags will be repeated

6.3.1 Indexing Performance and Scalability

The first part of our experiments is to examine the performance of our proposed indexing approaches. We run these approaches on NYT and WIKI collections.

NYT

Figure 6.2 shows the performance of our indexing approaches on the NYT collection. We measure the time needed to build the inverted index using these approaches using different number of nodes. The NYT input file is split into 91 splits. Therefore, the JobTracker initiates 91 map tasks and number of reducers is equal to the number of reducers we configure in the job configuration. For performance and scalability testing, we run our experiments using different number of nodes. We can set number of reducers in the job configuration, but we can not set number of mappers. However we can run our job on a pool of machines in the cluster using the fairscheduler. For example when we want to run the indexing approach on 16 nodes then we submit this job to a pool which consists of 16 nodes. That means at most 16 of them will run at the same time.
Comparing the performance of the indexing approaches, we find that \text{Tid\_Did} and \text{Tid\_DidTs} perform better than \text{TidDid\_Ts} and \text{TidDidTs\_tf}, because the first two have fewer inserts into their index tables. They insert one entry for a term with all documents containing this term. But \text{TidDid\_Ts} inserts once per \text{tid-did} combination which means for every term in any document there is an insert (number of inserts for any term is equal to number of documents containing this term). \text{TidDidTs\_tf} approach inserts once for each \text{tid-did-ts} combination meaning that it inserts an entry for a term in every document version. Comparing \text{Tid\_Did} with \text{Tid\_DidTs}, the first approach performs better because it has less information per inserted entry. Comparing \text{TidDidTs\_tf} with \text{TidDid\_Ts}, number of inserts for \text{TidDid\_Ts} is equal to that for \text{TidDidTs\_tf}, since in the NYT collection we have one version per document. However, \text{TidDidTs\_tf} performs better than \text{TidDid\_Ts}, because the latter has more information in each insert.

![Performance of indexing approaches on NYT collection without pre-partitioning index tables](image)

Figure 6.2: Performance of indexing approaches on NYT collection without pre-partitioning index tables

In Figure 6.2, we observe a decreased running time while increasing number of
nodes. Using 32 nodes yields a running time equal or larger than that when we run using 16 nodes. We do not get full advantage of running using large number of nodes. All nodes are doing insert to an index table which is empty and thus stored on a single regionserver. Therefore, all nodes will start inserting to a single regionserver. The regionserver splits when it reaches the maximum size. To verify this conclusions, we re-run our indexing approaches. This time indexing approaches insert into a pre-partitioned index table. We can pre-partition the table manually, since we know the range of keys to be inserted in the index table. Another choice is to use the index table resulted from indexing approaches. Therefore, index tables already contain the inverted index and consist of multiple regionservers. This is an optimal pre-partitioning. Figure 6.2 shows the performance of indexing approaches using a pre-partitioned index tables. For all approaches, we get an improved performance using any number of nodes compared to the results in Figure 6.2. Thus, the increase of indexing time is due to the difference between number of workers in the cluster and number of regionservers.

WIKI

The second versioned document collection that we used is a sample from the Revision History of English Wikipedia (WIKI). The sample collection consists of 64 XML files, each file contains multiple articles, and each article contains multiple revisions. For indexing, the Hadoop JobTracker initiates 64 map tasks, each task takes as an input one of the 64 XML files. Figure 6.4 shows the performance of the different indexing approaches on the WIKI sample collection. We measure the performance of the different indexing approaches using different number of nodes. We observe same behavior of the indexing approaches using WIKI sample to the indexing of NYT collection we presented in Section 6.3.1. More precisely, Tid_Did and Tid_DidTs inserts fewer number of rows into index table than Tid_Did_Ts and TidDidTs_tf. Thus they perform better. Tid_Did and Tid_DidTs insert an equal number of rows in the index table. However, Tid_Did performs better because it inserts less information per row. TidDidTs_tf performs better than
6 Experimental Evaluation

![NYT Indexing approaches with pre-partitioning tables](image)

Figure 6.3: Performance of indexing approaches on NYT collection using pre-partitioned index tables

We see from Figure 6.4, as we increase the number of nodes the indexing time is decreased for all approaches until we have 32 nodes. There is a little decrease in indexing time between 16 nodes and 32 nodes which is not compared to the decrease between 8 nodes and 16 nodes. The decrease in indexing time between 16 nodes and 32 nodes differ from one approach to another depending on the number of inserts in index table. The little decrease in time using 32 nodes is due to the reason, that we increase number of processing nodes, which insert data to the same region in the index table.

6.3.2 Index Size

This part of our experiments examines how the selection of the indexing approach will impact the index size. For all approaches we store term frequency in the cell.
6.3 Results and Comparisons

Figure 6.4: Performance of indexing approaches on WIKI collection using pre-partitioned index tables

The difference between them is in the selection of the row key, the column family, and column qualifiers. The information we keep in the row key and columns are the term \( id \), the document \( id \), and the timestamp with different representations depending on the schema of the index table, we presented the different schemas in Section 4.2. The term \( id \) is a 4 bytes integer value, the document \( id \) is also a 4 bytes integer, and the timestamp is an 8 bytes long value. Next, we present the size of index tables resulting from the different indexing approaches for both NYT and WIKI versioned document collections.

**NYT**

Figure 6.5 shows the relation between indexing approaches and the size of the index table (in GB). The figure clearly shows differences in the size of index table between indexing approaches. We summarize the reasons behind the differences in the index tables size in the following points:
• Number of rows. Tid_Did and Tid_DidTs approaches have fewer rows compared to TidDidTs and TidDidTs_tf approaches.

• Number of columns. All index tables except TidDidTs_tf should have equal number of columns, because NYT collection contains only one version per document. Thus using document id or document timestamp as columns in the index table will result in the same number of columns in the index table. TidDidTs_tf has only one columns.

• Timestamp. HBase will add a timestamp for every inserted cell. This timestamp is by default the machine time in milliseconds at insertion time. User can specify the timestamp of the cell explicitly. For Tid_Did, we use HBase timestamps explicitly. It does not store any time information in the row key or in the columns. Other approaches store time information in row key or in columns. Nevertheless, HBase assigns a timestamp for every added row using these approaches. Thus we have two timestamps, the first is the real timestamp that we add in the row key or the column name depending on the indexing approach. The second timestamp from HBase which is the machine time at insertion.

Tid_Did has the smallest index size. Its index has fewer number of rows compared to TidDid_Ts and TidDidTs_tf. It has the same number of rows and columns compared to Tid_DidTs, however it contains less information in column names. The index size of Tid_DidTs approach is the largest because it keeps more columns per each row. The index of Tid_Did_Ts and TidDidTs_tf approaches contains one column per each row. TidDid_Ts approach stores timestamp of the document version of the document in the column name. As there is only one version per document in the NYT collection there is only one column per row. The index of TidDidTs_tf keeps one column per each row. The indexes of TidDid_Ts and TidDidTs_tf approaches have the same number of rows and same number of columns. But The index of Tid_Did_Ts is smaller than the index of TidDidTs_tf as the first stores less information per row.
6.3 Results and Comparisons

Figure 6.5: Size of index table using different approaches on NYT

**WIKI**

Figure 6.6 shows the relation between indexing approaches and the size of the index table (in GB). The WIKI collection contains multiple versions per document. Thus the approach which uses a composite of the term id and the document id as row key has fewer rows than the one uses a composite of the term id, the document id, and the timestamp. In the first case number of rows in the index table depends on the number of documents in the collection which contain the term. In the second case, the number of rows depends on the number of versions which contain the term.

In the figure, the index of Tid_Did is the smallest. It contains fewer number of rows compared to TidDid_Ts and TidDidTs_tf. Also it has the same number of rows as TidDidTs but TidDidTs has larger number of columns. Moreover, the timestamps in the index of Tid_Did is the HBase timestamps which we use explicitly for this approach. We do not compose timestamp neither in the row
key nor in the columns as we do in other approaches. The index size of Tid_DidTs approach is the largest, because it has the largest number of columns compared to all other approaches. TidDid_Ts and TidDidTs_tf have the same index size. The index of TidDid_Ts has smaller number of rows compared to the index of TidDidTs_tf. The number of rows in the first depends on the number of documents which contain the term, while in the second it depends on the number of versions. However, the index of TidDidTs_tf has only one column while number of columns in the index of TidDid_Ts depends on the number of versions of the document in the corresponding row key.

![Figure 6.6: Size of index table using different approaches on WIKI](image)

### 6.3.3 Query-Processing Performance

In this section, we evaluate the query-processing performance that can be achieved on the different types of indexes. For measurement, we employ query-processing wall-clock times of query-processing performance. Times are reported in two digits rounded seconds (s). We measure time of query-processing performance
on warm caches. So we execute each query for times; the first to warm caches, the subsequent three executions to obtain a more stable measurements of the query-processing time.

Next, we discuss the query-processing performance of Boolean queries and keyword queries using different indexes. Since we are interested in time-travel text search, we enrich each query type with time-point or time-interval. For time-point queries, the query is combined with a timestamp selected randomly that exists in the considered document collection life-time. For time-interval queries, we consider five granularities. Namely, milliseconds (MS), a single day (D), a week (W), a month (M), and a year (Y). For each time-interval query, the begin-time is randomly chosen and the end-time is selected by adding the corresponding granularity, e.g., if we use month granularity, then we pick the begin-time randomly and the end-time will be the begin-time plus one month. The entire time-interval falls into the time-interval of the considered document collection.

**Query Workloads** We use the same 150 queries that were employed in [BBNW07].

The complete query workloads of Boolean and keyword queries for the NYT and WIKI are given in Appendix A. As we use a sample from WIKI, and query workload in A.2 is for the original collection, processing these queries on the sample leads to get many empty results. Thus we only consider queries that have a non-empty result on our WIKI collection. After filtering out queries that lead to empty results, we get 82 queries shown in A.3

Next, we report our findings regarding processing performance for Boolean queries and keyword queries on the NYT and WIKI versioned document collections. For each query type, we report the processing performance of the five different granularities. We evaluate Boolean queries and keyword queries on the index tables whose information is shown in Tables 6.1 and 6.2 for NYT and WIKI, respectively. For query-processing performance comparison, we compute the mean time $\mu$, the standard deviation $sd$, and different percentiles, more precisely at
6 Experimental Evaluation

5%, 25%, 50%, 75%, and 95%. All measurements are in seconds.

<table>
<thead>
<tr>
<th>index</th>
<th>size(GB)</th>
<th># of rows</th>
<th># of regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tid_Did</td>
<td>16</td>
<td>1,469,740</td>
<td>158</td>
</tr>
<tr>
<td>Tid_DidTs</td>
<td>20.6</td>
<td>1,469,740</td>
<td>158</td>
</tr>
<tr>
<td>TidDid_Ts</td>
<td>19.2</td>
<td>491,453,936</td>
<td>201</td>
</tr>
<tr>
<td>TidDidTs_tf</td>
<td>20.1</td>
<td>491,453,936</td>
<td>204</td>
</tr>
</tbody>
</table>

Table 6.1: Index table information of different schemas for NYT collection

<table>
<thead>
<tr>
<th>index</th>
<th>size(GB)</th>
<th># of rows</th>
<th># of regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tid_Did</td>
<td>30.6</td>
<td>1,264,036</td>
<td>190</td>
</tr>
<tr>
<td>Tid_DidTs</td>
<td>33.3</td>
<td>1,264,036</td>
<td>191</td>
</tr>
<tr>
<td>TidDid_Ts</td>
<td>32.7</td>
<td>1,820,000,000</td>
<td>209</td>
</tr>
<tr>
<td>TidDidTs_tf</td>
<td>32.7</td>
<td>3,469,000,000</td>
<td>284</td>
</tr>
</tbody>
</table>

Table 6.2: Index table information of the different schemas for WIKI sample collection

Boolean Queries

Tables 6.3 and 6.4 show the query-processing performance for Boolean queries using different retrieving methods to retrieve data from different indexes for the NYT and WIKI collections. As the tables reveal, the schema of the index table has an impact on the query-processing performance across the two collections. Processing queries using the index tables which contain fewer rows gives better performance. More precisely, using the index which keeps one row per term in the collection performs better than Tid_Did and Tid_DidTs schemas. Using thes two schemas, given a term, we can access the index table and get one row whose row key is the term and all the documents containing this term in the columns. These indexes have an equal number of rows, and an equal number of columns for NYT and Tid_DidTs has more columns for WIKI because NYT contains one version per document, while WIKI contains multiple versions per document. We obtained similar query-processing performance using different granularities because the NYT collection, it contains one version per document. For the WIKI,
we take a sample as we described in Section 6.1, we see from the number of documents and versions of our sample that we have 3 versions per document on average, and this will not affect the processing time for the different granularities.

We use 150 queries for the NYT, these queries contain different terms, some of these terms are more frequent than others. For frequent terms we retrieve more data, for example, processing the *patios* query takes a 0.014 s for the time-point query, which is fast because it contains only an infrequent term, while processing the *washington post* query takes 1.3 s. Moreover the processing time for the *the* *washington post* takes 10.4 s, because it contains the stop word “the”. In the Tables 6.3 and 6.4 we present different percentiles to show the differences between the processing time of the different queries. From the standard deviation and the different percentiles for processing Boolean queries on NYT, we conclude that we have a clear difference of the query-processing time for the different queries. The 5% percentile shows the shortest processing times. The 95% percentile gives clue about how many queries are far away from the mean value and take large processing time.

The processing time for queries over NYT indexes is larger than that over the WIKI. The first covers a long interval of time, which means that for the given term, retrieved documents will be much larger, especially because they do not have a big difference in terms of the number of versions per document. Another main reason behind the processing time difference between them is that we use a sample from WIKI, the complete query workload set for the entire WIKI collection is shown in A.3. From this set, we filtered out queries that will return empty result when executed on the WIKI sample. The complete set of selected queries for the sample is shown in A.3. When we process these queries, we retrieve few documents for many of them. This observation is clear when we look at the different percentiles. From Table 6.6, we observe that around 75% of the WIKI queries have query-processing time less than the mean time for the different schemas with different granularities.
6 Experimental Evaluation

Keyword Queries

Tables 6.5 and 6.6 show the query-processing performance for keyword queries on NYT and WIKI collections, respectively. The tables compare the query-processing performance for the different approaches. The schema of the index table has an impact on the query-processing performance across the two collections. Query-processing on \texttt{Tid\_Did} and \texttt{Tid\_DidTs} indexes is faster than \texttt{TidDid\_Ts} and \texttt{Tid\_DidTs\_tf} indexes, because the first two contain fewer rows for each term, they contain exactly one row per term, while the number of rows for the other two indexes is larger and depends on how many documents contain the term for \texttt{Tid\_Did\_Ts}, or depends on number of versions for \texttt{TidDidTs\_tf}.

For NYT, by looking at the different percentiles in Table 6.5, we conclude that the keyword queries that we process on NYT collection have different processing time across the different indexes. Some of the queries take long processing time, this is clear when we look at the difference between the 75% percentile and the 95% percentile. For example, \texttt{Tid\_Did} has 0.86 s and 16.99 s for the 75% percentile and 95% percentile, respectively.

The query-processing performance depends on the query’s term, a query with frequent terms or stop words like “the” takes more processing time than one with an infrequent terms. \texttt{patios} takes 0.018 s to be processed, it is fast because it contains an infrequent term. Processing the \texttt{washington post} takes 1.46 s, moreover processing of the \texttt{washington post} takes 11.4 s, because it contains a stop word.

For WIKI, we use the same queries that we use in the Boolean queries. We selected them from the set of queries which is processed on the entire WIKI collection. From the different percentiles, we observe that most of the queries have a short processing time. For example, in Table 6.6 \texttt{Tid\_Did} has 0.19 s and 0.54 s
at the 75% and 95% percentiles, respectively.

6.3.4 Summary

We devised different schemas for storing our indexed data in HBase having different schemas leads to having different indexing approaches and different retrieving methods for query-processing. We study the indexing performance and the query-processing performance for these approaches through comprehensive experiments. We conclude that schemas which contain fewer rows in the index tables perform better in terms of both indexing performance and query-processing performance. Tid_Did and Tid_DidTs keep one row per each term in the index table, thus their indexing performance is better than TidDid_Ts and TidDidTs_tf. TidDid_Ts keeps one row per each document containing the term, and TidDidTs_tf keeps one row per each document version containing the term. For better indexing performance, it is good practice to pre-partion the HBase table which stores the indexed data. We find that the query-processing performance is independent of the size of the table because when we process the query we do not need to scan the entire index table, we limit our scan on it for specific term.
Table 6.3: Impact of the data layout of the index table on the processing performance of Boolean queries on NYT with different granularities; where µ is the mean time, sd is the standard deviation, and % is different percentiles, all measurements are in seconds.
Table 6.4: Impact of the data layout of the index table on the processing performance of Boolean queries on WIKI with different granularities; where $\mu$ is the mean time, $sd$ is the standard deviation, and $%xx$ is different percentiles, all measurements are in seconds.
Table 6.5: Impact of the data layout of the index table on the processing performance of keyword queries on NYT with different granularities; where \( \mu \) is the mean time, \( sd \) is the standard deviation, and \( %xx \) is different percentiles, all measurements are in seconds.
### Table 6.6: Impact of the data layout of the index table on the processing performance of keyword queries on WIKI with different granularities; where $\mu$ is the mean time, sd is the standard deviation, and $%xx$ is different percentiles, all measurements are in seconds
6 Experimental Evaluation
7 Conclusions and Future Directions

Nowadays, there are many available versioned document collections, for example, web archives (e.g., the Internet Archive), news archives (e.g., New York Times Annotated Corpus), or revision histories of wikis (most notably Wikipedia). With these available versioned document collections, the implementation of time-travel text search addressed the access limitations of web archives. Inverted index resulted from indexing these version document collections has a very large size. Our objective in this thesis, is to implement a distributed time-travel text search using Hadoop MapReduce for processing and HBase to store the inverted index.

7.1 Conclusions

In this thesis, we presented a distributed time-travel text search using Hadoop MapReduce for indexing, and using HBase for distributed storage to store indexed data. We exploit the data representation in HBase and devised different schemas to store indexed data in HBase. We implemented time-travel text search in a distributed fashion, it is distributed in terms of processing and storage. We use Hadoop MapReduce for indexing, so indexing is done using multiple machines. We use HBase to store indexed data, so the indexed data is distributed among different machines. We provide experimental evaluation to compare our devised approaches in terms of indexing performance, index size, and query-processing performance.
7 Conclusions and Future Directions

7.2 Future Directions

Our work has some space for further improvements. We list them in the following points:

- **Queries**: In our work, we support only disjunctive Boolean queries and keyword queries, our work can be extended to support other query types, e.g., conjunctive Boolean queries and richer query types like phrase queries.

- **Versioned document collections**: In the current setting of our system, we indexed two versioned document collections; the *New York Times Annotated Corpus (NYT)* and the *Revision History of English Wikipedia (WIKI)*. Our indexing approaches can be extended to support other versioned document collections like the *European Archive Crawls of U.K. Governmental Websites (UKGOV)*. To extend our indexing approaches to index a new collection, one needs to modify the mapper, but the reducer function does not depend on the collection.

- **HBase table pre-partitioning**: We conclude from our experiments that it is good practice to pre-partition the *HBase* table at the creation time. One can study the effect of the pre-partitioning of the *HBase* table on indexing performance.

- **Experiments**: In our work, we devised 5 different data layouts to represent indexed data in the *index* table. We provide experimental evaluation for 4 approaches in terms of indexing performance, index size, and query-processing performance. As a continuation, we can do these experiments to the *TidTs_Did* which we introduced for time partitioning. Further work could be done for this approach to study the effects of different partitioning time granularities on the indexing performance, index size, and the query-processing time.
A Query Workloads

| washington post, weather, cnn news, washingtonpost ltd, lou dobbs, flowers, time magazine, msnbc, wall street journal, sports illustrated, larry king, cnmoney, nancy grace, cnn money, horoscopes, news, political cartoons, funny videos, johnny cash, cnssi, war in iraq, larry king live, anderson cooper, decks, princess diana, time, the washington post, cartoons, redskins, florists, fortune magazine, cw network, world news, gmt, miss manners, patios, money magazine, don knotts, pope john paul ii, stocks, money, oil prices, retirement calculator, jackie robinson, chris penn, enron, fingernails, dixie chicks, fortune 500, dana reeve, videos, washington redskins, the wall street journal, crosswords, lou dobbs tonight, opus dei, september 11 2001, numerology, dana reeves, wnyt, london weather, washington dc metro, stock quote, drywall, headline news, interest rates, mutual funds, cnnfn, weed, sports, sudoku, storm shutters, sat scores, greenwich mean time, hurricane katrina, camorders, time zones, preteen models, bathrooms, cold war, eleanor roosevelt, florist, send flowers, e pierce marshall, iraq war, laptops, child safety, mother teresa, retaining walls, hurricane shutters, scanners, fortune, drive time, luther vandross, cars, sandra dee, world time, cost of living, aids in africa, henry ford, playground equipment, greenwich time, william kennedy smith, nfl, colonics, patrick kennedy, norma mccorvey, bed bath and beyond, rosa parks, drivetime, girls gone wild, political cartoon, soundboards, lifetime products, fica, dubey, zulu time, o j simpson, katharine graham, chandra levy, seven wonders of the world, nba, barbaro, dennis weaver, civil rights movement, video, hometime, bill cosby, twin towers, lucille ball, home improvements, watergate, dc metro, pheromones, kimberly dozier, wall street, chore charts, local news, immigration bill, health, bruising, thrush, dudley moore, chanel sunglasses, retirement, flight 93, bowflex, deal or no deal, john ritter |

Figure A.1: Boolean/keyword queries used for NYT dataset
Figure A.2: Boolean/keyword queries used for WIKI dataset
deaths in 2006, hurricane katrina, playboy, ask jeeves, randy orton, google, sudoku, wwe, cocaine, scientology, vagina, vietnam war, hanso foundation, columbine, pete wentz, french revolution, world war ii, elmo s world, american idol, david blaine, the last supper, truman capote, mozart, kkk, led zeppelin, alexander the great, june carter, agnostic, appendix, penthouse, rosa parks, hotmail, germany, chuck norris jokes, spain, acre, roe v wade, morphine, domino harvey, existentialism, palm sunday, spleen, sociopath, priory of sion, serotonin, karma, gloria vanderbilt, high school musical 2, the beatles, fidel castro, andy milonakis, randy jackson, maya angelou, john adams, june carter cash, the cold war, abraham lincoln, rome, nudity, watergate, manhunt net, kama sutra, bees, julius caesar, dna, dominican republic, manifest destiny, truman doctrine, korean war, concentration camps, liver, abortion, beethoven, oxycodone, the great depression, george w bush, charmed, imperialism, robert e lee, marijuana, the da vinci code, sportsnet new york

Figure A.3: Boolean/keyword queries used for WIKI sample dataset
A Query Workloads
# List of Figures

2.1 MapReduce job execution stages (taken from [DG10]) 20

2.2 Illustration of the execution of Algorithm 2.1 to count words frequencies using two documents as input 24

2.3 Part of the **webtable** that stores Web pages. Row keys are reversed URLs. The **contents** column family contains the web page contents, and the **anchor** column family contains the text of any anchor that references the page. CNN’s page is referenced by two anchors `anchor:cnnsi.com` and `anchor:my.look.ca`, so the row contains column for each one of them. Each anchor has one version, and the contents family has three versions, (taken from [CDG+08]) 28

2.4 Tablet locations hierarchy, (taken from [CDG+08]) 31

2.5 Simple inverted index, which keeps for each term in lexicon list, a list of documents that contain this term 37

2.6 Inverted index, which keeps list of postings. Postings contain documents identifiers and the payload for the corresponding term 37

3.1 Time-Travel Inverted Index 47

3.2 Postings for term $v$ to illustrate partitioning strategy 50

4.1 Data representation in $Tid\_Did$ index table 60

4.2 Data representation in $Tid\_DidTs$ index table 61

4.3 Data representation in $TidDid\_Ts$ index table 62

4.4 Data representation in $TidDidTs\_Tf$ index table 62

4.5 Data representation in $TidTs\_Did$ index table 63
List of Figures

5.1 System architecture ............................................ 84
5.2 Create HBase table \textit{Tid,Did} with one column family \textit{Did} ...... 87
5.3 Data flow of indexing ........................................... 88
5.4 Class diagram of indexing ....................................... 89
5.5 Data flow of the mapper ......................................... 91
5.6 Data flow of the reducer ......................................... 91
5.7 Query processing architecture ................................... 92
5.8 Query processor class diagram ................................... 93

6.1 Wikipedia page structure. For each version, the revision begin
and end tags will be repeated ........................................ 97
6.2 Performance of indexing approaches on \textit{NYT} collection without
pre-partitioning index tables ......................................... 98
6.3 Performance of indexing approaches on \textit{NYT} collection using pre-
partitioned index tables .............................................. 100
6.4 Performance of indexing approaches on \textit{WIKI} collection using pre-
partitioned index tables .............................................. 101
6.5 Size of index table using different approaches on \textit{NYT} .......... 103
6.6 Size of index table using different approaches on \textit{WIKI} .......... 104

A.1 Boolean/keyword queries used for \textit{NYT} dataset ............... 117
A.2 Boolean/keyword queries used for \textit{WIKI} dataset ............... 118
A.3 Boolean/keyword queries used for \textit{WIKI} sample dataset ........ 119
List of Tables

2.1 Naming differences between Google and Hadoop implementation of MapReduce ........................................... 25
2.2 Naming Differences between Bigtable and HBase ............... 32
4.1 Shortcuts notations .................................................... 59
4.2 Documents versions that contain term cat .......................... 59
5.1 Configuration properties for Pseudo and Fully-distributed modes 86
6.1 Index table information of different schemas for NYT collection 106
6.2 Index table information of the different schemas for WIKI sample collection ............................................. 106
6.3 Impact of the data layout of the index table on the processing performance of Boolean queries on NYT with different granularities; where $\mu$ is the mean time, $sd$ is the standard deviation, and $\%$ is different percentiles, all measurements are in seconds ............ 110
6.4 Impact of the data layout of the index table on the processing performance of Boolean queries on WIKI with different granularities; where $\mu$ is the mean time, $sd$ is the standard deviation, and $\%xx$ is different percentiles, all measurements are in seconds ............ 111
6.5 Impact of the data layout of the index table on the processing performance of keyword queries on NYT with different granularities; where $\mu$ is the mean time, $sd$ is the standard deviation, and $\%xx$ is different percentiles, all measurements are in seconds ............ 112
6.6 Impact of the data layout of the index table on the processing performance of keyword queries on WIKI with different granularities; where $\mu$ is the mean time, sd is the standard deviation, and $%xx$ is different percentiles, all measurements are in seconds . . . . . . . 113
List of Algorithms

2.1 Pseudo-code for word count MapReduce algorithm .......... 23
2.2 Pseudo-code for TAAT conjunctive query processing .......... 38
4.1 Pseudo-code for indexing mapper .......................... 65
4.2 Pseudo-code for indexing reducer .......................... 67
4.3 buildandStoreIX() implementation in Tid_DidReducer .......... 68
4.4 buildandStoreIX() implementation in Tid_DidTsReducer .......... 68
4.5 buildandStoreIX() implementation in TidDid TsReducer .......... 69
4.6 buildandStoreIX() implementation in TidDidTs fReducer .......... 70
4.7 buildandStoreIX() implementation in TidTs_DidReducer .......... 71
4.8 Pseudo-code for retrieving postings list from Tid_Did index .......... 74
4.9 Pseudo-code for retrieving postings list from Tid_DidTs index .......... 75
4.10 Pseudo-code for retrieving postings list from TidDid Ts index .......... 76
4.11 Pseudo-code for retrieving postings list from TidDidTs f index .......... 77
4.12 Pseudo-code for retrieving postings list from TidTs_Did index .......... 78
4.13 Pseudo-code for processing time-point query .................. 79
4.14 Pseudo-code for processing time-interval query ............... 80
4.15 Term-at-a-time processing for disjunctive Boolean queries ........ 81
4.16 Term-at-a-time processing for keyword queries ............... 82
Bibliography


Bibliography


[HG] Erik Hatcher and Otis Gospodnetic. Lucene in Action. MANNING.


[KATK10a] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, and Nectarios Koziris. Distributed indexing of web scale datasets for the cloud. MDAC 10, pages 1–6, New York, NY, USA, 2010. ACM.

Bibliography


[MMO] Richard M. C. McCreadie, Craig Macdonald, and Iadh Ounis. Comparing distributed indexing: To mapreduce or not?


Bibliography


