UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Versioned index data structures for time travel text search

Laureando:
Andrea LANGELI

Relatore:
Dr. Ing. Gianmaria SILVELLO

Anno Accademico 2016/2017

Abstract

Normally, search engines do not allow users to query a versioned collection of documents, since as soon as a new document version is indexed, the older one is lost. This prevents us from accessing useful information in a wide range of applications. For instance, within prior art search, it is important to establish if an idea is actually new or if it has been already patented as well as if there exist any older document versions concerning that idea. Thus, we need to time travel across document versions in order to find all possible instances of an idea or a piece of evidence.

Hence, in this work we develop a system for letting users search versioned documents – i.e., collections containing multiple versions for the same document – specifying their validity range by means of time intervals. To this end, we decide to enhance the widely-used Terrier open-source IR system by means of two strategies for index versioning: (i) the Baseline Approach (BA) and, (ii) the Mapping Approach (MA). The BA strategy filters out data related to documents whose time intervals do not satisfy a given temporal predicate. Whereas, the MA strategy speeds up query evaluation by taking advantege of additional data structures maintaining the positions of document versions in the index.

We evaluate these approaches on two test collections: the CLEF Adhoc monolingual collection for the Italian language and a temporlly versioned subset of the English Wikipedia. The evaluation process shows the correctness of the proposed BA and the MA approaches and that MA is more efficient than BA, even though it requires more central memory.

Contents

1 Introduction

Over the last two decades, the research community began debating the importance of preserving the human generated content, which is in part available on the web. This content, which is evolving at increasing rate, is not captured by the actual search engines, because they focus on enabling users to access the latest information available. This implies that as soon as we index a newer version of a document, this version replaces the previous one, thus involving a loss of potentially useful data. In recent years there has been a growing awareness that search limited to the latest snap-shot of evolving data, results in losing access to content which could be important in a wide range of different applications. However, it is important to remark that over the last decades, there have been several efforts to preserve the born digital contents available on the web. In particular, the Internet Archive [1] in an attempt to preserve such data, for future generations, has collected more than 150 billion of web pages since 1996. While in web archiving there have been several advances, the same is not true with regard to the access to the collected information. In fact, current web archives focus on the preservation of content, rather than on enabling users to inspect it. For instance, the Internet Archive lets us only look at the version history of a document, once we have submitted its URL, without allowing us to consult the content of the document versions. Current web archives are far from providing a feature which permits users to answer queries by evaluating them only over the document versions satisfying a specified temporal constraint. Such a functionality would allow us not only to exploit the data contained in the document versions retrieved for a query, but also to get useful information about the evolution of the documents over time, which ultimately lead us to think about the causes bringing about such changes.

1.1 Purpose and structure of this work

Motivated by these ideas, we examine several existing proposals in the literature, concerned with time-travel text search. Time-travel text search is an extension of the standard text search which enables users to bound the scope of a search only to documents which match a temporal predicate. Therefore, users can couple a classical text query with a time constraint to locate only the document versions that are relevant with respect to the specified keywords and the temporal constraint of interest. A topic enriched with a time constraint is referred to as time-travel query. In order to evaluate a time travel query on a set of documents, we have to provide such documents with temporal data, thus obtaining a temporally versioned collection. The goal of the existing strategies is to allow users to carry out search over temporally versioned collections, by organizing the data collected during the indexing process so that the retrieval procedure is as efficient as possible. The idea underlying all the approaches we analyze is to exploit the temporal data of the collection documents to speed up query evaluation. Such strategies propose different organization schemes for the Inverted index, which, is the most efficient index structure for query evaluation. We can organize the existing solutions in four categories. The first group aims at speeding up the retrieval process by slicing the Inverted index along the time axis, while the second group deals with slicing such structure along the temporal dimension. The third group involves performing query evaluation by exploiting the similarities between adjacent versions of the same document. Finally, the four group trades off some speed in the retrieval process with the exactness of the results. After analyzing the ideas proposed in the literature so far, we aim at enhancing Terrier, an open source information retrieval system, with a functionality which lets us to search temporally versioned collections. Among the available open source platforms letting users conduct information retrieval tasks, we focus on Terrier because of the increasing interest of the information retrieval community towards such system and because it allow us to carry out the indexing and the retrieval processes incrementally, thus letting us perform the indexing procedure on some documents after answering some given queries. Moreover, we select such system because of the growing community of users collaborating to the support and the development of the project, with the ideal goal of implementing a functionality which can be used by the maximum number of people. On the minus side, Terrier limits the boundaries within which to implement the described feature, thus preventing us from exploiting any of the ideas proposed in the literature, concerned with time-travel text search. This is due to the fact that implementing one of such strategies would require us to completely change

how the indexing and retrieval processes are carried out in Terrier, thus compromising the possibility of allowing other users to adopt the developed strategies. To accomplish the goal specified above we devise two strategies, the Baseline Approach and Mapping Approach for index versioning, which will be referred to as BA and MA strategies, respectively. The Basic Approach for index versioning consists of a baseline approach for index versioning which enables search over temporally versioned collections by ignoring the data related to documents whose time intervals do not satisfy the one we provide the query of interest with. On the other hand, the MA approach deals with exploiting the positions assigned to the data of the document versions in the inverted index stream of bytes, for speeding up the retrieval process. With the dual objective of verifying the correctness of the implemented strategies and assessing the time we need to evaluate a given query, we test the BA and MA strategies on two collections. The first one is the Adhoc monolingual test collection for the Italian language, while the second collection is obtained by putting together some web pages about cooking we download from the Wikimedia Foundation Project.

This work is organized as follows: in Chapter 2 we give a background on Information Retrieval, particularly focusing on the indexing and retrieval processes. In Chapter 2 we present Experimental Evaluation and the fundamental concepts of relevance and test collection. Moreover, we describe the inverted index, the most efficient index structure proposed so far for text retrieval purposes. We will come back to the model we report for the inverted index throughout this work, with the aim of illustrating how we have to change the reference model to implement the BA and MA approaches. We conclude the chapter by giving some details on three open source systems, commonly used in conducting information retrieval tasks. In Chapter 3 we illustrate some strategies which have been designed to allow users to search temporally versioned collection, underlining for each of them their strengths and weaknesses. In Chapter 4 we present how the real-time indexing and retrieval processes are carried out in Terrier. In Chapter 5 we delineate the BA and MA strategies, the strategies we have developed as of Terrier real-time indexing and retrieval processes, for handling time-travel text search. In Chapter 6 we provide the results we have obtained by testing the BA and MA strategies on two collections, namely the CLEF Adhoc monolingual test collection for the Italian language and one we have generated starting from the English Wikipedia. In Chapter 7, after summarizing the results of our experimental evaluation, we provide some final remarks and some future works.

2 Information Retrieval

2.1 Information Retrieval

"Information retrieval is a field concerned with the structure, analysis, organization, storage, searching, and retrieval of information" [2]. This definition was proposed by Gerald Salton, one of the leading researchers of *Information Retrieval* from the 1960s to the 1990s. We start our discussion by analyzing the elements of this definition. Firstly, we take the "information retrieval" statement into account. As indicated in the textbook *Readings in Information Retrieval* [5], written by Spark Jones and Willett, the term was proposed by Mooers in 1952. Broadly speaking, it refers to the possibility for a user to obtain some information, which he should consider useful, in relation to a *query*, which, can be thought of as an actualization of an *information need*. The type of information that a user can obtain mostly depends from the type of documents comprising the collection in which the research is carried out. Since 1950s the Information Retrieval researchers has focused on text collections, but the technological advancement of the last decades, above all by increasing data storage capability of various orders of magnitude, led to a new interest towards other types of documents: audio, video, images. A discussion on how such documents are managed during the *indexing* and *retrieval processes* is beyond the scope of this thesis, in which we exclusively take into account text collections. Some examples of text collection documents are web pages, books, scholarly papers, company reports. Turning to the main topic, it is worth saying that, as suggested by *Lancaster*, the term "information retrieval" is somewhat erroneous. In fact, the retrieval process does not directly provide new knowledge to a user who issues a query, but it merely informs him about the existence of some documents, which should be about the topic the examined query is related to. In order to respond to some *queries*, the collection documents

must go through some processing phases. For each document in the collection, we carry out the following operations:

- *Extraction of the document content.*
- *Analysis of the document content to obtain a document representation.*
- *Storage of the document representation into some data structures that make the subsequent search of useful data as efficient as possible.*

Lastly, we consider the topic of "structure of information". The documents in a corpus can have a certain amount of structure, for instance the title of a scientific paper, the author, a section which provides a summary of the document content, but most of the document information is in the form of text, which is mostly unstructured. This feature of text collection documents has a great impact on the retrieval phase. Differently from responding a query in *Data Retrieval*, in which only matching items are searched for, in Information Retrieval we would like to get back all, but only the documents which have an high probability of being relevant with respect to a given query. Note that a document could be relevant in regard to a query even if it does not contain any of the *query terms*, namely the terms composing the query. Furthermore, since the relevance of a document is strictly dependent on the user issuing the query, it also depends on some aspects which are completely independent from the query. Some examples are the user capability of expressing his information need, the user awareness about the content of the documents, the query is matched with. We conclude this part by proposing an historical remark. As stated by van Rijsbergen in [4], when high speed computers began to be used to carry out information retrieval tasks, many researchers thought that computers would have been able to partially solve the problem of detecting all the relevant documents with respect to a given query, due to their calculating capacity. Soon, Information Retrieval researchers realized that the only way to exploit the new available computing capability was to represent the concept of being relevant in mathematical terms. They started to conceive what will be known as *information retrieval models*.

The remainder of this section is organized as follows. In 2.1.1 we give a background on the indexing process, placing particular emphasis on its meaning. In particular, we briefly present the different processing phases it generally consists of. Additionally, we provide a brief description of the *inverted index*, the most common index structure used for carrying out the indexing and retrieval processes. In Section 2.2 we analyze on the retrieval process,

particularly focusing on the concept of relevance which is crucial in Information Retrieval. All the topics we address in this section present some key concepts on which we will come back in the following parts of this work.

2.1.1 Indexing process

The purpose of the indexing process is to represent the content of a document in such a way that the information in the document is efficiently retrieved. After the indexing process, we obtain a representation of the documents we took into account. These documents are processed during the indexing procedure by means of techniques of different flavors, depending on the type of queries the information system, we look at, aims at satisfying. During the retrieval phase, an user query goes through the same indexing processing as the documents. Then the representation of the query is matched with the representation of the indexed set of documents, to detect the documents which more likely satisfy the user need. At this stage, it is important to remark that the procedures the indexing process consists of greatly influence the set of documents which are considered as relevant with respect to a query. The indexing process can be performed manually, automatically, or in any way between the extremes. In this work we focused only on automatic indexing process. For the indexing process, we can adopt a linguistic approach or a statistical one. Adopting a linguistic approach means representing a document content by using both some information taken from the document itself, and some others which are external to the document. We only mention some reasons, detailed by Van Rijsbergen in [4], which have led many researchers to focus on the statistical approach.

- Linguistic approach is quite difficult to put in practice because it is very expensive to implement.
- There were little improvements in the field over the past years, and to date, it is not clear what are the benefits provided by the adoption of such an approach.
- The statistical approach, whose theoretical underpinnings were Luhn's studies, has been studied for some decades and proved itself moderately successful.

The statistical approach is based on Luhn's teaching which can be summarized by proposing his own words: "It is here proposed that the frequency of word occurrence in an article furnishes a useful measurement of word significance. It is further proposed that the relative

position within a sentence of words having given values of significance furnish a useful measurement for determining the significance of sentences" [7]. The mode of operation of all the modern search engines is based on this principle. In fact, according to Luhn's lesson, much of the meaning of a text is captured by counting the words occurrence and co-occurrence. Taking english text documents into account, Luhn also noted that the distribution of words within a text is skew: there are some words which occur frequently, whereas there are others which occur rarely. In particular, he noted that the words "the" and "of" account for 10% of all word occurrences, while the six most frequent words account for 20% of all occurrences. These observations were expressed in mathematical terms by George Zipf, a linguistic researcher who formulated and proved many empirical laws. His law $\big($formula $2.1\big)$ relates the frequency with which a word occurs in a text with its meaning and structure.

$$r * f = k \tag{2.1}$$

The formula above states that the *rank* of a word times its frequency is approximately equal to a constant. An equivalent formulation of Zipf's law is given below.

$$r * P_r = c \tag{2.2}$$

In 2.2, the frequency with which a word occurs in a text is replaced by its occurrence probability. This is simply obtained by dividing the word frequency by the overall count of word occurrences. For the English language $c \approx 0.1$.
In Figure 2.1, we depict the relation between the *rank* and the *probability of occurence* of a word, for the English language.
Although Zip's law turned out to be quite trustworthy, some discrepancies between the predicted and computed values have been detected for high and low ranks. Some modifications of the Zipf's law were proposed so far. The most significant one is reviewed in [8]. Luhn has futher extended Zipf's contribution by defining two thresholds, in terms of rank values, which determine the words that can be considered as useful for the representation of the informative content of a set of documents. These thresholds were respectively referred to as *upper cutoff* and *lower cutoff*. The terms with a rank greater than the upper cutoff are common; they do not help us to discriminate one document from another; they occur in most of the documents we take into account. Some examples of these words are articles and prepositions. On the contrary, the terms with a rank lower than the lower cutoff occur rarely, hence they do not contribute to describe the informative content of the whole collection of

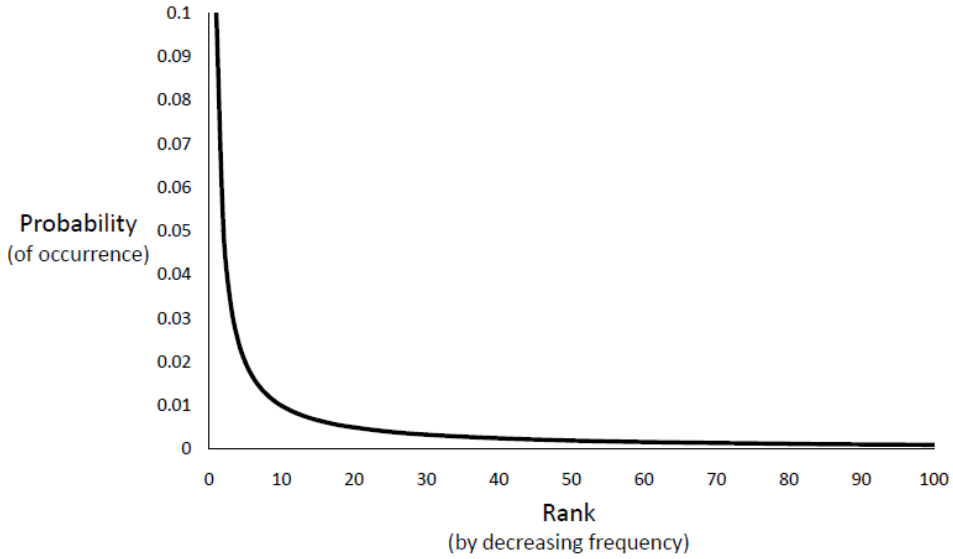

Figure 2.1. *Occurrence probability of a word as a function of its rank. In this figure we assume that constant c is equal to 0.1. The depicted trend for the occurrence probability is obtained from the formula 2.2. This figure has been proposed by Croft and al. in [6].*

documents. The capability of a term of being both descriptive and discriminating has been defined as *resolving power*. Looking at the frequency trend as a function of rank values, illustrated in Figure 2.2, the resolving power has its peak at a rank half way between the upper cutoff and lower cutoff, while it is almost equal to 0 at the cutoffs values. According to Luhn, words have to be considered as significant in relation to their *resolving power*: all the words with a rank between the two cutoffs are significant. The upper cutoff and lower cutoff thresholds must be determined by using a trial and error strategy.

So far, we have defined the single elements composing the documents of interest as words. Although this nomenclature is valid in case we consider textual documents, since we could examine documents of various type, the basic elements conveying information could not be words. Hereafter, instead of "word" we will use the term *token*, aiming at the maximum generality. A token could be a date, a number, a phrase, an image, a musical note. Turning to the purpose of the indexing process, we can redefine it more precisely. Looking at textual documents, the goal of the indexing process is to represent the tokens $\big($words in case of textual documents$\big)$ occurring in the collection documents by means of more consistent *index terms*. The index terms constitute the effective representation of the content of the corpus documents which is actually used during the retrieval phase. As mentioned above, the index terms are obtained by processing the documents of interest through a set of procedures. The procedures, documents go through, depend both on their type and the queries we want to

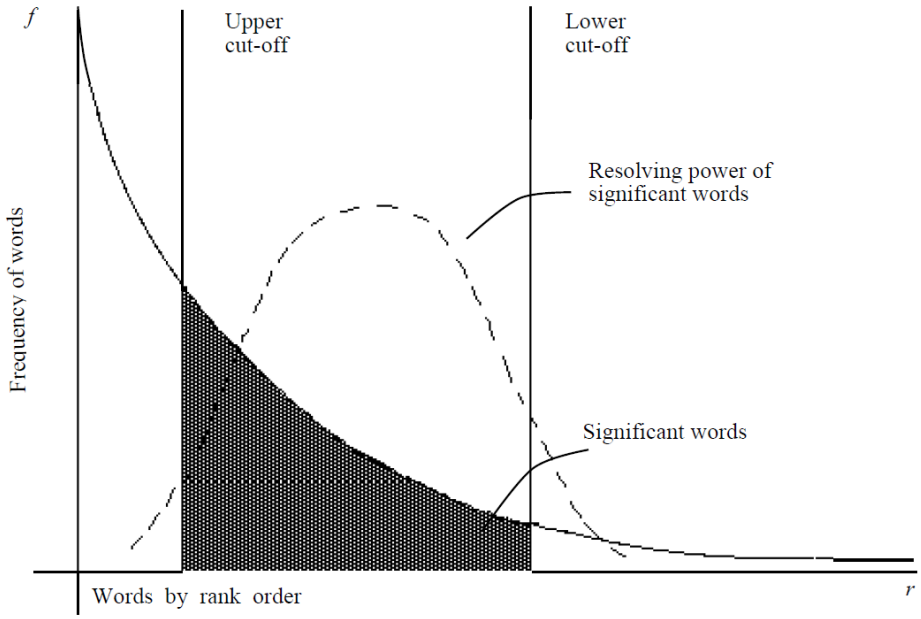

Figure 2.2. *Representation of the resolving power trend as a function of the rank values. In this figure we can also observe how the lower cutoff and the upper cutoff can be graphically used to select all, but only the significant words. This figure has been created by Van Rijsbergen and inserted in [4].*

respond.

The processing phases the documents generally go through are:

- Tokenization (Lexical analysis)
- Removal of the *stopwords*.
- Stemming.
- Term composition.

In Section 2.2 we place particular emphasis on the first three processing phases, since we actually perform them on the documents we consider during the experimental phase. It is worth noting that we do not have to apply all the presented procedures to the documents of a corpus. In fact, we believe that many commercial search engines do not actually do it.

2.1.2 Indexing process phases

In this section we provide an overview of the procedures which are commonly applied to textual documents, during the indexing process.

2.1.2.1 Lexical Analysis

The *lexical analysis* aims at converting a stream of characters into a stream of tokens. A text can be easily divided in a set of tokens, by exploiting the separating characters, as spaces and punctuation marks. The actual implementation of this processing phase strongly depends on the language of the documents. In fact, the language actually determines the set of characters we can use to separate the words composing the textual documents of interest. Carrying the lexical analysis out requires us to deal with some practical issues, including:

- Choosing an encoding to encrypt the characters.
- Deciding if to process the text either as a stream of characters or as a stream of bytes.
- Deciding if to use some buffers to speed up the reading operation.

Addressing these issues is of paramount importance, since we must execute these processing phases on several tens of thousands of documents.

2.1.2.2 The *stopwords* removal

The stopwords are words which have no meaning without being paired with other words. These words not only do not contribute to describe the content of the documents in which they are contained, but also generally occur in most of the documents we look at, hence they do not help us to discriminate one document from another. Looking at Figure 2.2, once the lower cutoff and the upper cutoff have been fixed, on the basis of the occurrence of the words within the examined documents, the stopwords are likely to have a rank greater than the upper cutoff. Aiming at not including the stopwords into the representation of the documents, we could use some lists containing these words during the indexing process. These lists are referred to as *stop lists*. After finding a word in a document, in case a stop list is adopted, we search for the word in the stop list. If it occurs there, we simply discard it. As for the previous phase, removing the stopwords involves tackling some practical issues. The main drawback consists of having to compile a stop list for each language we are faced with. On the plus side, using a stoplist greatly reduce the index size. At this stage, the index can be simply thought of as the collection of all the data related to the documents of interest.

2.1.2.3 Stemming

Stemming, also called *conflation*, is the indexing process phase which aims at capturing the relationships between different forms of the same word. Stemming allows the vocabulary mismatch between queries and documents issue to be solved, by reducing variant word forms to a common root. This processing phase is underpinned by the fact that, in many languages, group of words which share a substring are very often semantically related. The algorithm which actually performs the stemming of the words contained in the examined documents is called stemmer. In order to design a stemmer, two different approaches can be followed:

- Algorithmic approach: the matches between words and their *stems* is accomplished by an algorithm, which may use some statistics and data evaluated on the given corpus.

- Dictionary based approach: the matches between words and their *stems*, which are kept in some dictionaries, are prepared manually by some experts, on the basis of some etymological studies.

In this work, the emphasis is placed on the algorithmic approach. For what regards the dictionary based approach, we limit ourself to saying that it requires some experts to update the dictionaries over time. Additionally, we need to develop a different list of matches between words and their stems for each language we deal with. With respect to the algorithmic approach, stemmers can mainly be subdivided into two categories: *rule-based* and *statistical* ones. The first family adopts language-specific rules to group morphologically related words, while the latter transforms word variant forms into their *stems*, on the basis of a given corpus. In addition, there is a third *stemmer* group, namely *hybrid stemmers*, which combines some rule-based and statistical stemmer features in order to improve the performance of the stemming procedure. For a more detailed description of the various aforementioned stemmer families, refer to [9]. However, irrespective of the family a stemmer belongs to, its usage generally entails two benefits:

- Improving the retrieval performance.

- Reducing the number of words being used for representing the corpus documents, because of the stemming process.

2.1.2.4 Term composition

The term composition procedure originated from the wish of increasing the score of documents which exactly contain the *query terms*. From a theoretical perspective, considering some adjacent words as a phrase accounts for the fact that a phrase is much more precise in describing a topic than the single words comprising it. From a practical point of view, term composition involves dealing with some issues, including the following:

- Specifying the meaning of "phrase".
- Deciding if to identify phrases during the indexing process or to take them into account at a later time.

For what concerns the latter issue, it is worth stating that the second choice could be much expensive from a computational perspective, since we could consider many different definitions of "phrase". The way in which we handle the phrases during the indexing and retrieval processes could significantly affect the retrieval performance, since most of the queries submitted by users are made up of two or three words.

2.1.3 Inverted index

In the following parts of this thesis work, we will refer to the model we describe here, which is therefore taken as reference, to illustrate how the functionalities we have developed during the experimental phase are actually implemented. In particular, we place emphasis on the change we need to bring about in the way in which data are organized in the typical inverted index implementation to develop the added features. Query evaluation is carried out through an index, which informally can be thought of as a data structure that simply maps terms to the documents they occur in.
Over the years, the most efficient index structure for textual query evaluation has been found to be the inverted index. It consists of two main components:

- The *vocabulary*.
- The inverted lists associated with the vocabulary entries.

The *vocabulary*, for each token gathered during the indexing process, includes the following data:

term t	f_t	Inverted list for t
`and`	1	$\langle 6,2\rangle$
`big`	2	$\langle 2,2\rangle$ $\langle 3,1\rangle$
`dark`	1	$\langle 6,1\rangle$
`did`	1	$\langle 4,1\rangle$
`gown`	1	$\langle 2,1\rangle$
`had`	1	$\langle 3,1\rangle$
`house`	2	$\langle 2,1\rangle$ $\langle 3,1\rangle$
`in`	5	$\langle 1,1\rangle$ $\langle 2,2\rangle$ $\langle 3,1\rangle$ $\langle 5,1\rangle$ $\langle 6,2\rangle$
`keep`	3	$\langle 1,1\rangle$ $\langle 3,1\rangle$ $\langle 5,1\rangle$
`keeper`	3	$\langle 1,1\rangle$ $\langle 4,1\rangle$ $\langle 5,1\rangle$
`keeps`	3	$\langle 1,1\rangle$ $\langle 5,1\rangle$ $\langle 6,1\rangle$
`light`	1	$\langle 6,1\rangle$
`never`	1	$\langle 4,1\rangle$
`night`	3	$\langle 1,1\rangle$ $\langle 4,1\rangle$ $\langle 5,2\rangle$
`old`	4	$\langle 1,1\rangle$ $\langle 2,2\rangle$ $\langle 3,1\rangle$ $\langle 4,1\rangle$
`sleep`	1	$\langle 4,1\rangle$
`sleeps`	1	$\langle 6,1\rangle$
`the`	6	$\langle 1,3\rangle$ $\langle 2,2\rangle$ $\langle 3,3\rangle$ $\langle 4,1\rangle$ $\langle 5,3\rangle$ $\langle 6,2\rangle$
`town`	2	$\langle 1,1\rangle$ $\langle 3,1\rangle$
`where`	1	$\langle 4,1\rangle$

Figure 2.3. *Set of inverted lists making up the inverted index. For each token, we maintain f_t, namely the total count of documents in which the token occurs in and the corresponding inverted list. Each inverted list is composed of a list of pairs, each of which consisting of a document identifier d and the count of the occurrences of the token of interest in d.*

- f_t: the count of the documents the token t occurs in.
- p_t: a pointer which points to the *inverted list* associated with the token t.

The latter component of the index is a set of inverted lists, one for each token composing the *vocabulary*. An inverted list is a list containing the identifiers of the documents in which the corresponding token occurs in. Formally, an inverted list is a sequence of pairs of the type $\big(d, f_{d,t}\big)$. The first element d is simply a document identifier, whereas $f_{d,t}$ is the count of the occurrences of the term t in the document d. The inverted list is also denoted as posting list, in that each entry forming it is denoted as *posting*. In the remainder, we assume that the inverted lists composing the inverted index are stored in contiguous locations, instead of being stored as a set of different blocks. As suggested by *Zobel et al.* in [12], this

assumption has a range of implications. Firstly, a list can be written or read in a single operation. Secondly, no additional space is required to store pointers to the first position of each block that would compose the index structure. Additionally, by arranging the inverted lists into blocks, the index update procedures would manage variable-length fragments that would vary enormously in size, from tiny to vast. This could involves some inefficiencies. The last topic we address in this section is related to the efficiency issue. Instead of storing the identifiers of documents as they are, we store them by using *d-gaps*. For instance, the sequence:

$$11, 15, 18, 20, 25, 30 \tag{2.3}$$

is represented by

$$11, 4, 3, 2, 5, 5 \tag{2.4}$$

It is useful to remark that although this strategy does not reduce the magnitude of the stored values, it significantly decreases the mean value, thus significantly reducing the amount of space we need to maintain the index structure on disk.

2.2 Retrieval process

An *information need* is a lack of information that an user wants to fill. An user expresses an information need by means of a query, which, in the context we took into account, is made up of words, but it can generally consist of musical notes, drawings or data of any kind. Therefore, in the following, we refer to the single elements composing a query as *terms*. The type of terms strongly depends on the type of documents the user is interested in. A document satisfies an information need of a user if he perceives it as relevant. It is worthwhile to note that a document can be considered as relevant although it does not contain any of the query terms. Conversely, an user may evaluate a document as not relevant even if it contains all the query terms. This is an important difference between Information Retrieval and Data Retrieval. In Data Retrieval, we consider as relevat all the records matching a defined pattern, while in Information Retrieval, the relevance of a document strictly depends on the user submitting the query, as we mentioned above. Worth saying that when it comes to relevance, we distinguish between *topical relevance* and *user relevance*. A document is said to be topically relevant with respect to a query if it is about the same topic the query

is. Instead, the user relevance deals with factors which are more dependent on the user assessment, including:

- The language of a document. A document which is intended for a specialized audience could be evaluated as not relevant by a casual user.
- The novelty of a document. A document proposing new ideas on a topic could be perceived as more relevant than one addressing well-known matters.
- The level of detail of a document.

Since the success of the retrieval process also relies on some aspects a user can not be completely aware of, it may require several cycles before finishing, during which the initial formulation of the information need may be refined, due to the results returned by the information system. Additionally, the user can also specify more accurately his information need by using some advanced functionalities, as the possibility of restricting the search domain only to documents containing all the query terms. For information system we mean the totality of software and hardware components whereby the indexing and retrieval processes are carried out. The result of the retrieval process consists of a sorted list, containing the identifiers of the documents which are more likely to be considered as relevant by the user. The *sorted list* is denoted as *ranked list*. In the ranked list, the identifiers are listed in decreasing order with respect to the scores assigned by the information system, reflecting the similarity between the query and the documents. Informally, an information system is said to be effective if many of the identifiers on the top of the ranked list are related to documents which turn out to be relevant. Since the beginning of the Information Retrieval, researchers made a significant effort trying to establish quantitative methods for evaluating the effectiveness of an information system on the basis of the relevant documents being retrieved. The two commonest measures which have been adopted with this purpose are respectively *precision* and *recall*. Precision is the fraction of retrieved documents which are relevant, whereas recall is the fraction of relevant documents which are retrieved. Although recall and precision have been proposed in the 1970s, they are currently in use and form the building blocks on which many more complex measures rely on. Precision and recall were defined by Cyril Cleverdon, one of the leading figures of the field in the 1960s and 1970s. His experiments, known respectively as *Cranfield1* and *Cranfield2* constitute a milestone in Information Retrieval and significantly contributed to the development of the evaluation

topic. The goal of *Cranfield1*, which was carried out between 1958 and 1962, was to analyze four manual indexing techniques in order to establish which of the methods were the most efficient. The results of *Cranfield1* constituted the bases on which *Cranfield2* was designed. It was carried out between 1962 and 1966. In *Cranfield2*, Cleverdon aimed at studying various manual indexing methods more accurately than it was done previously, with a particular emphasis on the retrieval and evaluation phases. The best results, in term of retrieval performance, were obtained by taking single words into account during the indexing process and by using as query terms the words actually occurring in the corpus documents. These results, together with Luhn's lesson, proving that the most effective retrieval techniques are those which rely on the actual content of the corpus documents, which in turn can be adequately represented by the words occurring in them, paved the way for the evolution of Information Retrieval towards approaches and methodologies which are typical of the computer science. We conclude this section by giving a brief overview of two well-known *information retrieval models* with the purpose of addressing some issues arising during the retrieval process. A significant effort has been made over the past decades to comprehend and formalize the process through which an user perceives a document as relevant in regard to an information need. Although accomplishing this task is a tall order and would require developing a complete understanding of how human brain works, some steps in the right direction have been taken through the information retrieval models. Formally, an information retrieval model is a mathematical model which is designed to allow the representation of the content of documents and the queries to be carried out consistently. Over the decades, many information retrieval models of different flavors were defined. One of the first information retrieval model being devised was the *Boolean model*. This model gives back a document only if it exactly matches the query specification, which is defined by using operators from the Boolean logic. Since a document can be only retrieved or discarded, during the retrieval process, there are only two relevance grades. The binary nature of relevance which is enforced by the Boolean model implies that identifiers given back to the user posing the query are not ordered. Actually, later contributions, introduced some criteria which allow the similarity between queries and documents to be computed, thus allowing the identifiers returned to the user to be listed in some order. One of the main benefits of the Boolean model, in comparison with other retrieval models, is the efficiency of the retrieval procedure which accounts for the fact that is it still used, despite its simplicity. Another information retrieval model was the *vector space model*, which was proposed in 1975 by Gerald Salton. The details are omitted

here, but can be found in [11]. By using this model, we assume that documents and queries belong to a vectorial space consisting of t dimensions, where t is the count of the unique terms comprising the corpus documents. Both a corpus document and a query can be represented in the following way:

$$D_i = \big(d_{i1}, d_{i2}, ..., d_{ij}, ..., d_{it}\big) \tag{2.5}$$

$$Q = \big(q_1, q_2, ..., q_j, ..., q_t\big) \tag{2.6}$$

The score being assigned to each document, in regard to an issued query, can be conceived as a distance between that document and the query of interest. In fact, the score is evaluated on the basis of a similarity measure, which actually measures the similarity grade between the document and the query. One of the commonly used similarity measure is known as *cosine correlation*:

$$Cosine\big(D_i, Q\big) = \frac{\sum_j \big(d_{ij} q_j\big)}{\sqrt[2]{\sum_j d_{ij}^2 \sum_j q_j^2}} \tag{2.7}$$

Given a document and a query, the cosine correlation similarity measure is equal to the cosine of the angle between the vectors associated with them. Supposing that the computed vectors have the same number of elements, the cosine correlation is equal to 1 if the two vectors coincide, while it equals 0 if the two vectors have not got any element in common. Looking at cosine, we can note that d_{ij} has not been specified yet. This term accounts for the relevance of the document D_i on the grounds that it contains the query term q_j. Over the past years, many different weighting schemes have been proposed. Most of these are variations of the *tf-idf* weighting scheme. The *term frequency* component, which is generally indicated as *tf*, reflects the importance of a term in a document $\big(D_i\big)$. It is generally computed as:

$$tf_{ik} = \frac{f_{ik}}{\sum_{j=1}^{t} f_{ij}} \tag{2.8}$$

where D_i is the examined document, f_{ik} is the number of occurrences of the term k in the document D_i and tf_{ik} is the term frequency weight associated with the couple $\big(q, D_i\big)$. Instead, the *idf* term accounts for the importance of a term in relation to the whole corpus. The more documents a term occurs in, the less discriminating the term is between documents, and consequently the less useful it will turn out to be, during the retrieval phase. The idf form which is commonly used is:

$$idf_k = \log \frac{N}{n_k} \tag{2.9}$$

where N is the total number of corpus documents, n_k is equal to the number of documents containing the term q_k, while idf_k is the *inverse document frequency* of the term k. As suggested by Robertson in [20], the idf of a term can be thought of as the actual amount of information carried by that term. Finally, putting all the pieces together, we can compute d_{ik} as:

$$d_{ik} = \log\big(f_{ik} + 1\big)\log\big(\frac{N}{n_k}\big) \tag{2.10}$$

The last topic we discuss in this section is *text collection*. Firstly, we define text collection C of Information Retrieval the set

$$C = \Big\{D, T, RJ\Big\} \tag{2.11}$$

where D is a corpus of documents, T is a set of topics, while RJ is a set of relevance judgements, namely a list specifying the documents which are relevant for each topic t in T. As we mentioned above, a topic can be thought of as the actualization of an information need of a user, who ultimately submits the query to an information retrieval system to satisfy his need. From an historical point of view, as indicated by Harman in [30], the concept of test collection derives from a research work culminated in the Cranfield 2 experiment. In Cranfield 2, Cleverdon and his staff developed a paradigm for evaluation that constitutes one of the cornerstones on which Information Retrieval relies on. This paradigm, in particular, is grounded in two key concepts, that we briefly give below:

- The text collection must model a concrete research task of a specific class of users. This involves a range of implications. Before creating the text collection, we need to clearly point out the users for which the text collection is intended. In other words, we must indicate the class of users who can be really interested in the documents that will compose the corpus of the text collection. Note that specifying the class of users automatically means fixing some properties of the corpus documents, such as the level of detail and the language used to present the topics covered in the documents. Then, we must formulate a set of queries. These topics must reflect the information needs of the indicated users. In fact, they must be such that they could be actually formulated by the users taken into account in designing the text collection.

- The *text collection* must be prepared before carrying out the experimentation phase,

following the criteria specified above. This allows Information Retrieval researchers to reuse the text collection for performing their experiments at a later time than that at which we have actually developed the text collection.

It is worth stating that generating a test collection, following the Cranfield paradigm, is very expensive, because of the need of grouping together some experts, which must formulate the topics, according to the defined class of users. Moreover, they have to point out the relevance judgements, on the basis of the purpose for which we intend to create the text collection. In the experimental phase, for experimenting with the developed strategies, we have considered the CLEF Adhoc monolingual test collection for the Italian language, which has been developed in 2003. Such test collection has been generated within the CLEF evaluation campaign. Before going on to illustrate the different experiments we have carried out, we give a brief background on the evaluation campaigns. For evaluation campaign we mean an international initiative intended for putting together human and economic resources of research and industrial groups, with the purpose of allowing the assessment of information retrieval systems to be carried out in a trasparent and reproducible way. In particular, the CLEF evaluation campaign, which was instituted in 2000, in the footstep of the TREC evaluation campaign, is an european evaluation campaign. For what concerns the text collections, it is focused on analyzing multi-language collections. In the latest years, great emphasis was placed on collections containing multimedia documents.

2.3 Open source information retrieval systems

In this section we provide an overview of three well-known information retrieval systems, which are Terrier, Lucene and Indri. In particular, we focus on the first one, which we have used during the experimental phase. Open source information retrieval systems are of paramount importance for the information retrieval research community, as they provide state-of-art technologies which enable researchers to carry out their tasks without "reinventing the wheel". This is particularly important in a experimental field such as Information Retrieval, where the size of the test corpora has grown 500-fold over the last decade and where research goals are constantly evolving. Furthermore, the open source nature of these tools makes them act as corner-stones, upon which both the academic community and casual users may develop their own functionalities. The information retrieval systems we take into account in this section share some features. All of them allow us to

carry out the indexing and retrieval processes on the basis of some user-defined parameters. Furthermore, each of them let us specify the information retrieval model which is used for computing the scores of the documents in the collection of interest. Terrier, Indri and Lucene are being developed by tackling issues related to:

- Effectiveness: the system should be effective in the sense that it should enable researchers to access state-of-art technologies, while requiring minimal effort.
- Efficiency: the system should be designed to ensure that the *indexing* and retrieval processes are performed efficiently, regardless the examined data collection and the available resources.
- Scalability: over the last decade there was a shift in the scale of the information retrieval tasks researchers are required to perform. Therefore the system should provide the means through which these tasks can be conducted.
- Adaptability: user needs have evolved at increasing speed over the past years. The system should allow users to customize its functionalities as much as possible to let them tackle the new challenges they are faced with.

Below, we provide a brief description of the aforementioned *information retrieval systems*.

- *Terrier*: it stands for *Terabyte RetrIEveR* and it is a project being currently developed by the Department of Computing Science of the University of Glasgow. It is being developed in java. The project started in 2000 and the first version of the code has been available to the general public since November 2004. Initially, the project purpose was to provide a common framework for students to use for their research, but over the years its goal became more ambitious. Currently, its purpose consists of implementing a flexible and extensible platform which can be used for carrying out experiments in the information retrieval field, in a transparent, robust and reproducible way. With the aim of being as extensible as possible, Terrier adopts a modular design. This entails that many procedures, making up the indexing and retrieval processes, can be customized according to the user needs. For instance, Terrier provides implementation of several information retrieval models, including BM25 [20] and TF-IDF [15] as well as others from the divergence from randomness framework. Moreover, aiming at efficiency both in terms of the disk space used during the indexing process and in terms of the time we

need to answer a query, Terrier implements several compression techniques, such as the codings of the JavaFastPFOR package. This feature makes Terrier particularly suitable for being used to manage large-scale collections. Over the last years, the interest of the research community towards the new challenges of real-time search tasks has led to the implementation of the real-time indexing process. This feature which is available in Terrier since version 4.0 allow users to resume the indexing process at later time points, after we have responded some queries.

- *Indri*: this platform has been developed since 2004 with the purpose of creating a system capable of evaluating complex queries on large data collections. Initially, it has been released as a small modification of a bigger project, the Lemur project, which has arisen from a collaboration between the University of Massachusetts and the Carnagie Mellon University. The issues which emerged in the attempt of meeting the aforementioned requirements of flexibility and extensibility made the development team build almost an entirely new project. From the start, the Indri project was developed placing a particular emphasis on the distributed mode of operation, making the platform particularly suitable for performing information retrieval tasks on large data collections. As Terrier, for adaptability sake, Indri let users customize many components of the platform. The Indri project is mainly used for carrying out research experiments. The code is entirely written in C++.

- *Lucene*: Apache Lucene is an open-source java based library providing interfaces for performing tasks such as indexing, querying, language analysis. It is supported by a group of contributors of the Apache Software Foundation. Lucene functionalities are principally divided in four packages, which are about

  - The analysis of documents content.
  - The indexing and storage procedures.
  - The searching functionality.
  - The ancillary modules.

  The first three packages form the core of the library, while the last one provides additional functionalities used to perform search-related tasks. Lucene architecture has evolved over the years, reflecting the change in the project extent. Lucene is designed

to ensure that users can benefit from the maximum efficiency, while making use of one of its functionality.

In the experimental phase, among the aforementioned information retrieval systems, we have chosen Terrier. This choice is justified by the fact that it implements the indexing process in a real-time context, thus letting us resume the indexing process at later time points, without restarting the process from the scratch each time we take into consideration a new document. This functionality has enabled us to enrich the documents being part of the collection of interest with temporal data, and with them to keep up with the change in their content as they evolved over time. In fact, during the experimental phase we have indexed several versions of the same document, each of them enriched with a temporal interval, defining the time range in which it effectively represented the content of the document it belonged to. During the retrieval process, we have used the added information to narrow the research scope only to the versions whose temporal data satisfied the time predicate of the query of interest. Additionally, our choice has also been encouraged by the growing user community collaborating to support the development of this project as well as by the growing interest of the information retrieval research community towards Terrier, which ultimately aims at becoming the European Search Engine.

3 Related works

In this chapter, we briefly analyze some strategies which have been devised for addressing text search over temporally versioned document collections. We start this section by reporting some nomenclature. A versioned document collection is a collection in which each document occurs in multiple versions, which overall represent the evolution of the document over time. Searching such collections means to match a query, enriched with a temporal predicate, against the collection documents, and to retrieve all, and only the documents whose time interval overlaps with the query interval. In the following, we refer to the queries, composed of a bags of words and a temporal predicate, as *time-travel queries*. Note that in the simplest case, the time intervals of the time-travel queries reduce to single time points. As reported in [23], temporally versioned document collections are often much larger than standard ones, because of the multiple versions for each document occurring in the collection. It is worthwhile to note that consecutive versions of the same document are often very similar to each other. We can exploit this similarity for reducing the index size, by taking advantage of suitable compression techniques. So far there has been some work on indexing and searching of versioned document collections. We can broadly divide the strategies proposed so far in three groups, each of them referring to a specific layer of the search architecture for such collections.

- The first set is about index compression techniques which, by exploiting the similarity between adjacent versions of the same document, lead to a reduction of the index size, while speeding up query evaluation. This derives from the fact that we can maintain some inverted lists in the main memory, because of the decrease of the amount of space required for their storage.

- The second group deals with index organization and index traversal schemes. The proposals falling into this category, delineate some strategies for organizing the index data efficiently, with the purpose of skipping the postings whose validity time interval does not adhere to the time constraint of a given time-travel query.

- The third group is concerned with the development of higher-level temporal operators, such as the Stable *top-k* [25]. The idea underlying these works is that we can be interested in retrieving all the documents being relevant throughout the *query* time interval.

In this section, we focus on the middle layer, thus analyzing some index architectures, which are intended for achieving the highest query evaluation performance. The approaches we take into account principally attempt to achieve this goal in three different ways: (i) by partitioning the inverted lists along the time-axis (ii) by partitioning the inverted lists along the document dimension (iii) by relaxing the constraint of getting exact results. We explore these matters in greater detail in the remainder of the section, after presenting the model we refer to throughout this section.

We represent each document $\mathbf{d} \in \mathbf{D}$ of a temporally versioned collection through a sequence of its versions:

$$\mathbf{d} = \{d^{t_1}, d^{t_2}, d^{t_h} \therefore\} \tag{3.1}$$

Each document version d^{t_i} comes with a time interval $[t_i, t_{i+1})$, specifying when we must consider the version as the actual version of $\mathbf{d}$. In the remainder, we hypothesize that the timestamps t_j are non-negative integers assuming increasing values starting from 1. We point out that we use the value 1 for representing the actual time in which a search is carried out. Therefore, we denote a version d^{t_i} as the *current version* of a document $\mathbf{d}$, if its validity time interval is

$$[t_i, 1) \tag{3.2}$$

where t_i is a non negative integer value. The *active version* of a document turns into an *archive version* as soon as a new version of the same document go through the indexing process. Note that in the framework we describe here, the removal of a document from the collection consists of the insertion of a new version of the same document, which does not contain any term. The time-travel queries consist of bags of words $Q = \{q_1, \therefore, q_n\}$ provided with time-intervals $[b, e]$, where both b and e are integer values, with $b \geq 1$ and $e \geq b$. We

recall that, if b and e coincide, the time interval reduces to a single time point. During query evaluation, we analyze all the document versions that exist at any time during the examined time interval. With the aim of performing time-travel text search over a temporally versioned collection, we have also to extend the posting lists, the inverted index consists of, by adding temporal data to each posting. In detail, a time range is included into each posting, indicating when the payload information is valid. Therefore, each posting is composed of a set of three elements, outlined below:

$$\big(d, f, [t_i, t_{i+1}]\big) \tag{3.3}$$

where d is a document identifier, f is the count of the occurrences of the examined term in d, whereas $[t_i, t_{i+1}]$ is a time validity interval.

3.1 Sublist materialization

The first approach we analyze in this chapter is known as *sublist materialization*. This strategy, devised by *Berberich et al.*, is reviewed in [24]. Firstly, we point out that the baseline strategy for carrying out query evaluation involves scanning all the the posting lists of the inverted index and filtering out entries which do not satisfy the query temporal predicate. Note that this strategy entails some postings are taken out from the on-disk data structures and they are moved to the main memory, before being first decoded and then analyzed, even if their validity time interval does not adhere to the query time constraint. *Berberich et al.* address this problem by presenting the idea of *materialized sublists*. We obtain the materialized sublists by slicing the inverted index along the time-axis. Each *materialized list* is provided with a time-interval and contains all the postings whose validity time interval partly overlap with it. Performing the evaluation of a time-travel query reduces to scanning any materialized list, whose time interval spans across the temporal boundaries of the query time range. We use the toy example in Figure 3.1 to better explain the general idea behind this approach. The posting lists shown in Figure 3.1 contain 10 postings, belonging to three documents which are $d1$, $d2$ and $d3$. We assume the depicted postings are identified through the integer values they are labeled with. We further suppose that we are given with a query q with temporal constraint $[t_1, t_2]$. We assume of posing the given time-travel query q to an information retrieval system, whose inverted index implements the strategy described here. Moreover, we suppose that the time-axis is split in such a way that the first three postings

(whose labels are 1, 5 and 8) belong to a materialized sublist, whereas the remaining postings lie in an other one.

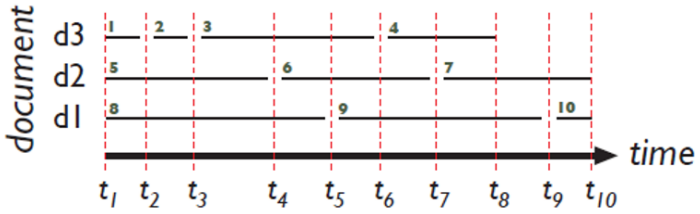

Figure 3.1. *Application of the sublist materialization approach to the postings of an inverted index, whose entries come from three documents, which are respectively d_1, d_2 and d_3. In this figure we can observe the validity temporal interval associated with each posting, identified by means of an integer value. This figure has been created by Berberich et al. and has been inserted in [24].*

In this case, processing the time-travel query q only requires us to read those postings which are part of the first materialized list, since it is the only materialized sublist whose validity interval overlap the one of the query. Choosing the number of materialized sublists, we must divide the inverted index in, results from a trade-off between performance and space. One approach to address the issue is to create a materialized sublist for each elementary time interval, thus achieving the best possible performance, but at the cost of significantly increasing the amount of space required for maintaining the inverted index on disk. In contrast to this strategy, we can generate an unique materialized sublist, into which we include all the postings. This approach would require minimal space, since each posting would occur exactly once in the inverted index.
The approach outlined here has some drawbacks which we present below:

- A posting whose validity time-interval partially span the one of a sublist is added to that sublist. This involves inserting the same posting in several materialized sublists, ultimately resulting in blowing up the index size.
- Although the outlined strategy permit to significantly reduce the number of invalid postings, analyzed during the retrieval process, inefficiencies may arise in query evaluation if the time range of the time-travel query is very long.

3.2 Performing query evaluation by using KMV synopses

To work around the drawbacks outlined above, *Berberich et al.* conceived an other strategy, which aims to trade some result quality off for improved retrieval time. Firstly, we start this

section by proposing the definition of *partition*. For a term v, we define the partition $\rho_{v,j}$ as the set of postings, associated with v, which occur in the materialized sublist with time range $[j, j+1)$. Therefore, each term is associated with a set of such *partitions*, which is denoted as ρ_v. The principle underlying this approach is that if we knew that a partition is mainly composed of postings, occurring in partitions already analyzed, we could avoid taking into account the partition at hand without significantly compromising the final result quality. The goal of *Berberich et al.* was to delineate an algorithm whereby selecting a set of partitions, which can generate the best approximation of query result, without exceeding a given cost. This can be expressed either in terms of the number of partitions opened for reading or the number of postings which can be analyzed during the query evaluation process. The algorithm presented in [27] heavily rely on obtaining high quality estimates of the cardinality of the intersection or the union of two sets, which are composed of document identifiers. With the purpose of getting these estimates, KMV synopses are used. A dissertation on this mathematical tool can be found in [26].

The main limit of this approach is that it allows query evaluation performance to be improved by degrading the quality of the result. Relaxing the constraint of result exactness is not feasible in most of the application contexts. This argument as well as the high computational cost of the algorithm for selecting the optimal set of partitions to evaluate a query, discouraged us from using this strategy in the experimental phase.

3.3 Temporal index sharding

Another proposal which attempts to speed up time-travel search over temporally versioned collections by dividing the *inverted index* in several components is reviewed in [28]. This approach has been devised by the same researchers who conceived the strategies described above and it can be actually considered as an improvement over the solutions which has been previously presented. This strategy involves dividing the index in *shards*, so that any time-travel query can be evaluated without sequentially reading the postings which do not qualify for the query time range. Indeed, the division of the inverted index in shards is performed by taking into account the different computational costs of sequential and casual accesses we incur onto during the query evaluation process. *Temporal index sharding* aims at addressing the problem we are faced with in the materialized sublists approach. In detail, instead of partitioning the posting lists along the time-axis, it involves slicing them along the document

identifier dimension. The index resulting from this operation is referred to as *sharded index*, in that the different components in which the postings are arranged in are denoted as shards.

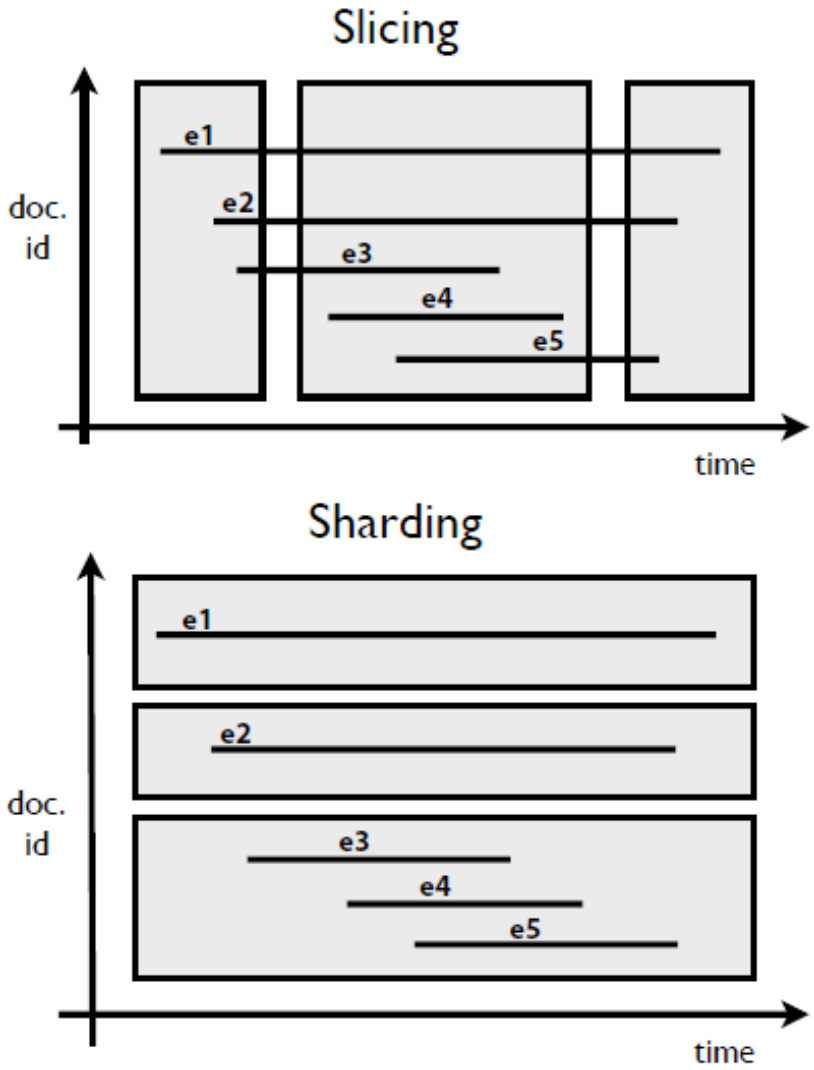

Figure 3.2. *Comparison on the way in which the entries of an inverted list are handled in the sublist materialization approach and in the temporal index sharding strategy. In the first image of the figure, the entries are subdivided into different sets in regard to their validity temporal interval, while in the latter image they are distributed among the depicted partitions with respect to their identifiers. This figure has been drawn from [28].*

In Figure 3.2 we can see a comparison between *sublist materialization* approach and *temporal index sharding* strategy. Looking at the entry e_1, we can note that sublist materialization approach implies it is replicated in all the depicted materialized sublists, since its time interval overlap with the ones the depicted materialized lists are attached with. Conversely, the

temporal index sharding strategy, entailing the inverted index is sliced over the document identifier dimension, involves including e_1 into a unique shard.
Each shard is provided with a data structure, defined *impact list*. For every possible begin boundary of a query interval b_q, it keeps the position, within the corresponding shard, of the first entry whose time interval contains b_q. The impact lists are maintained in the main memory. The *postings* of a shard are ordered by the begin boundary of the time interval they are attached with. Given a time predicate, after matching its begin boundary against the impact lists, we only need to analyze each shard composing the index, starting from the position the corresponding impact list points to. Therefore, for each involved shard, processing a time-travel query implies carrying out the following operations:

- *Opening the examined shard.*
- *Matching the left boundary of the query interval against the impact list of the shard.*
- *Seeking the position detected at the previous point.*
- *Accomplishing sequential reads of the postings in the shard until all the entries with a valid time interval have not been read. Note that as the elements composing the shard are arranged in begin-time order, this stage ends as soon as we encounter a posting whose begin-time exceeds the query end-time.*

Therefore, by using impact lists we can avoid reading postings whose time interval is before the one attached with an issued query. Nevertheless, wasted reads can still occur. Suppose that we are given with two postings, which are indicated below as A and B, whose time intervals are respectively $[A_{begin}, A_{end}]$ and $[B_{begin}, B_{end}]$. We further suppose that:

$$A_{begin} \leq B_{begin} \quad \wedge \quad A_{end} \geq B_{end} \tag{3.4}$$

Note that responding a time-travel query q, with time interval $[q_{begin}, q_{end}]$, involves reading the posting A even if its time interval does not qualify for the query time range. Formally, reading a posting corresponds to a wasted read if it is accessed during query evaluation even if its time interval does not span the query time interval. Let us introduce the definition of *staircase property*.

Given a *shard* S

$$\forall A, B \in S : A = [A_{begin}, A_{end}] \quad B = [B_{begin}, B_{end}]$$

$$A_{begin} \leq B_{begin} \implies A_{end} < B_{end} \tag{3.5}$$

we say that S has the staircase property. In this case, S is denoted as *idealized shard*.
If we subdivided each posting list into one or more idealized shards, processing a time-travel query would not involve any wasted read. Since query evaluation basically consists of carrying out an open and several skip operations for all the shards of a query term, we would like to minimize the number of idealized shards composing the inverted index. In [28], *Berberich et al.* propose a greedy algorithm for solving the problem at hand, given the set of time intervals of the inverted index entries. As we argued above, given that this strategy entails all the shards have the staircase property, it might involve many idealized shards being generated, ultimately degrading query processing performance. If the cost of a casual access is very high, some benefits might derive from loosening the idealized sharding approach, thus allowing some wasted reads. For idealized sharding approach we mean the strategy delineated above, which aims at improving query evaluation performance, by subdividing the inverted index into a certain number of idealized shards. To ensure the highest query evaluation performance is achieved, the problem of finding the smallest set of idealized shards can be further bounded by adding a constraint related to the costs of random and of sequential reads. A greedy algorithm for solving the problem of interest has been proposed by *Berberich et al.* in [28].

Although the strategy delineated above proved itself quite successful when it has been tested on standard text collections, it can not be adopted for dealing with temporally versioned collections. If new versions of the documents are inserted, then the shards must be computed from scratch for later time points.

3.4 Incremental sharding

In the literature there exists a variant of the *temporal index sharding* approach, which supports time travel search over temporally versioned collections. This solution is referred to as *incremental sharding*. This strategy involves using a *sharded index*, which is actually divided in an *active index* and an *archive* one. The former only deals with *active versions*, while the latter only handles archive versions of the documents in the collection. The archive index is in turn subdivided into a in-memory and an on-disk index, both organized in shards. Each shard of the in-memory index is associated with both a buffer of size η and an impact list. η

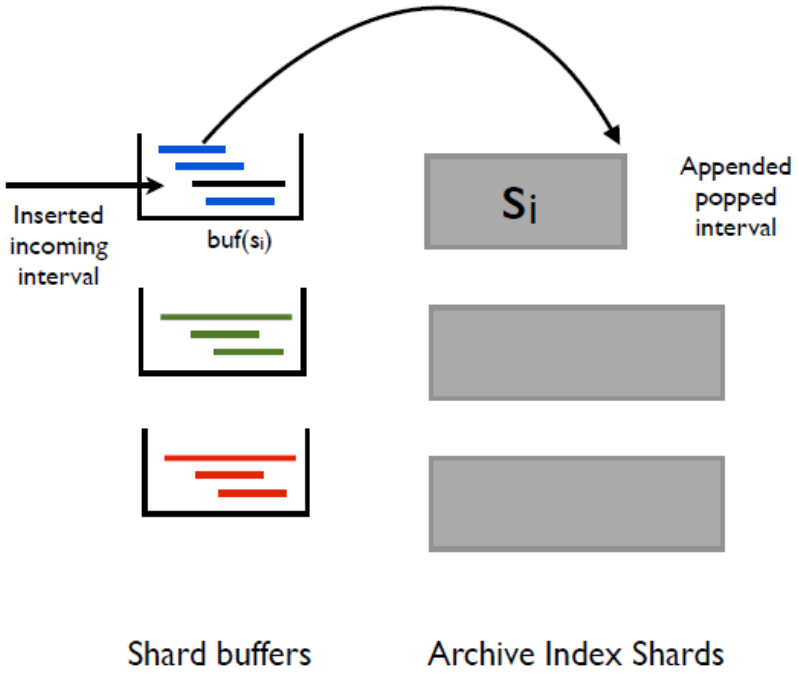

Figure 3.3. *Representation of the buffers and the in-memory shards, associated with the in-memory archive index. The size of each buffer is equal to the maximum number of postings we accept them being read during the query evaluation process for a specific query term, even if they do not qualify for the query time interval. As soon as one the depicted buffer fills up with postings, the one with the earliest time interval left boundary is taken out and inserted in the corresponding shard. We have drawn this image from [28].*

corresponds to the maximum number of wasted reads that are allowed per shard, during the evaluation of a single query term. If during the indexing process, we chance upon a document for which a version exists in the active index, it turns into an archive version and its data are moved to the in-memory data structure of the archive index. In detail, its postings, enriched with the temporal intervals, are included into the buffers associated with the in-memory index shards.

As depicted in Figure 3.3, as soon as the $\eta+1-th$ posting is included into a buffer, the element of that buffer with the smallest begin time is popped out and it is added to the corresponding shard of the in-memory archive index. When these shards are full, their data are merged together and included into the shards of the disk-based archive index.

It is worthwhile to note that this strategy, by using only append operations, has the benefit of building the up-to-date *archive index* incrementally, from the data used for building the previous versions of the same archive index, thus avoiding recomputation. Additionally, this

strategy ensures the query evaluation process is efficient, since the set of inverted index shards is evaluated taking into account the input-output characteristics of the storage infrastructure housing the index.

3.5 Approximate temporal coalescing

Approximate temporal coalescing technique strives to tackle the blow-up in the size of the inverted lists, by merging sequence of postings together, if the error incurred in such an operation is below a defined threshold. This strategy comes from the fact that in highly dynamic versioned document collections, frequent terms are associated with very long posting lists, each of them maintaining a posting for each document version, the corresponding term occurs in. This ultimately results in significantly degrading query evaluation performance. Moreover, successive versions of the same document are often very similar to one another, hence, only some of their postings differ, while the remaining ones are the same. *Approximate temporal coalescing* approach tackles the mentioned issue, thus reducing the number of postings in a inverted list, by merging sequences of postings which have almost the same payload. We apply this strategy to each inverted list, singularly taken. Let us introduce some nomenclature. Below we assume to take into account an index term v and its posting list, which will be denoted as L_v.

$$L_v = \Big\{ \big(d, f_i, [t_i, t_{i+1}\big), \therefore, \big(d, f_{n-1}, [t_{n-1}, t_n\big)]] \big) \Big\} \tag{3.6}$$

where d is a document identifier, f_i is the count of the occurrences of v in each version of d and $[t_i, t_{i+1}\big)]$ is a time interval. Note that, in 3.6 we have only considered the postings related to a single document, whose identifier is d. By carrying out the *Approximate temporal coalescing* approach on L_v we obtain the minimal length output sequence of *postings* O_v:

$$O_v = \Big\{ \big(d, f_j, [t_j, t_{j+1}\big), \therefore, \big(d, f_{m-1}, [t_{m-1}, t_m\big)\big) \Big\} \tag{3.7}$$

where d, f_j and $[t_j, t_{j+1}\big)]$ have the same meaning as in the formula 3.6. O_v is evaluated from L_v by enforcing the following constraints:

- L_v and O_v must cover the same time-range, hence $t_i = t_j$ and $t_n = t_m$.

- A sequence of postings of L_v can be turned into a single posting of O_v only if the error incurred in such transformation is below a given threshold ϵ.

For coalescing a sequence of postings of L_v into a single posting of O_v several error functions could be adopted. Looking at $\big(d, f_j, [t_j, t_{j+1}\big)$ and $\big(d, f_j, [t_j, t_{j+1}\big)$, generic *postings* of L_v and O_v respectively, we could employ as error function the relative error between their payloads, namely:

$$err\big(p_i, p_j\big) = \frac{\|p_i - p_j\|}{\|p_i\|} \tag{3.8}$$

Given an input posting list, evaluating an optimal solution for the problem at hand takes $O\big(n^2\big)$ time, but an approximation can be computed by using a linear time algorithm, which is based on the sliding window algorithm, given in [29]. In the experimental phase we haven't considered this strategy, because we are only interested in approaches allowing exact results to be computed.

3.6 A two-level index organization scheme

The alternative we look at here is focused on improving time-travel search performance by means of an index organization scheme ensuring the effectiveness of the low-level compression techniques is preserved. The idea underlying the proposed scheme is that techniques which enable time travel search over temporally versioned collections by partitioning the index, without taking into account the effects of partitioning on the effectiveness of the low-level compression techniques, can lead to significant increases of the index size. This ultimately may result in slowing down query evaluation process, in both memory and disk-based index architectures. The index organization scheme we take into account has been outlined in [23]. It involves using a two-level inverted index. The first level contains a posting list for each token composing the index Lexicon. Such a posting list maintains a posting for a document d if and only if exists at least a version of d in which the term t actually occurs in. Instead, the second level of the inverted index consists of two components:

- a *bit vector*: its size corresponds to the number of versions of d. It specifies which versions of d actually maintain t.

- a *frequency vector*: it keeps the number of occurrences of t within each version of d.

We remark that, with the aim of minimizing the amount of space required for storing index data structures, all the bit and frequency vectors are compressed by using suitable up-to-date compression techniques, such as the PForDelta [32] compression methods. As highlighted in [23], this index organization scheme permits time-travel query evalutation to be carried out very efficiently. Indeed, we first match the query against the first-level index, ruling out all the documents which do not contain any query term, then we traverse the second-level index, thus fetching and decompressing any necessary bit and frequency vectors. Moreover, with the goal of further improving query evaluation performance, in [23], *He et al.* propose to extend the inverted index with a memory-based table, containing the time ranges of the different collection document versions. By inspecting this table, after traversing the first-level of the inverted index, we can verify if the documents retrieved actually overlap the query time interval. This allows us to rule out documents which do not satisfy the query temporal constraint, without fetching the corresponding second-level data. Furthermore, in [23] *He et al.* delineate some methods which could be used for alleviating the disk-access costs, in case a disk-based index is used for performing the indexing process. These index organization schemes aim at improving query evaluation performance by placing the index postings so that those intersecting a given query range are located in one part of the inverted list. A description of these proposals is beyond the scope of this work. A comprehensive discussion of these matters is given in [23].

Although the outlined approach has several advantages and presents us with many opportunities for improving text search over temporally versioned collections, we have not taken it into account in the experimental phase. This decision is underpinned by the fact that, in order to implement it, we should have completely modified how the Terrier real-time indexing and retrieval processes are carried out. Moreover, if we had put the approach at hand into effect, by changing entirely the way in which real-time indexing and retrieval processes are actually performed in Terrier, we would have made a pointless effort, since the functionality we would have developed in that case could not been used by the information retrieval research comunity. In fact, in such case, in order to use the implemented feature, researchers should have adopted our system.

4 Terrier real-time indexing and retrieval processes

In this section we give an overview of how the indexing and retrieval processes are carried out in an actual information retrieval system. In particular, we will take into account the open source system called Terrier. Terrier implements a flexible open source search engine which allows us to use state-of-art indexing and retrieval functionalities.

This section is divided in two parts. In the first one, we report a background on the indexing and retrieval processes, as they are provided in Terrier $\left(\text{since version } 4.0\right)$, focusing on their mode of operation, while in the second one we report some implementation details.

4.1 Terrier real-time indexing and retrieval processes: mode of operation

4.1.1 Real-time indexing process

Along with traditional on-disk indices, Terrier allows us to carry out the indexing and retrieval processes by using either memory-only or hybrid memory and disk index structures. In fact, we can adopt the first approach only if the examined collection of documents is small, due to the limited amount of available central memory, but has the advantage of being very efficient. The second strategy involves moving data from the central memory to disk structures at regular intervals. In contrast with the first approach, the second strategy enables us to consider bigger document collections. Using this strategy requires us to point out how we want the data mantained on disk to be managed. In fact, for what concerns the disk-based data structures, Terrier presents two choices:

- Merging the data structures progressively getting a unique on-disk data structure which, in conjunction with the one kept in-memory, reflect the actual state of the index over time.

- Keeping the data structures separate. In this case, the index is made up of an in-memory data structure and several ones maintained on disk.

Note that what is generally denoted as index, is actually a wrapper of many indices, one of them is kept in-memory, while the others are maintained on-disk. In the remainder, we assume that the second strategy is adopted as well as that the data moved on disk at regular intervals are merged together resulting in a unique on-disk index. Therefore, in our context, a real-time index consists of one in-memory data structure and one data structure on disk. Before carrying out the indexing process, we have also to specify the document collection. In fact, documents could be divided in more than one collection. Since from a computational point of view, there is no significant difference between using only one or more collections, in the following, we assume that only one is used, to which all documents belong to. During the indexing process, each document is handled separately. Terrier let us specify which operations are applied to each document term. By default, these processing procedures are the removal of the stopwords and the stemming. The first step of the indexing process is the creation of the index structures which are designed to store the data collected from the documents. In Terrier, an index generally consists of a *Lexicon*, an *Inverted index*, a *Direct index*, a *Document index*, a *Meta index*. The *Direct index* basically associates each document in the collection with the terms composing it and their frequencies in that document. The *Direct index* is principally used for query expansion and since we do not take into account such functionality, in the remainder, we do not consider it. However it is worthwhile to note that the *Direct index* is not generated by default during the Terrier indexing process. Below, we give some background on the aforementioned data structures. They are:

- *Lexicon*: for each token occurring in the indexed documents, it contains an identifier, which is an unique integer value and some general statistics as the term and document frequency. If the *Lexicon* belongs to a disk-based index, it also maintains a reference to the initial position in which its postings are stored in the *Inverted index*. In Terrier, it basically consists of a sorted map which associates every token in the corpus with an entry containing the described fields.

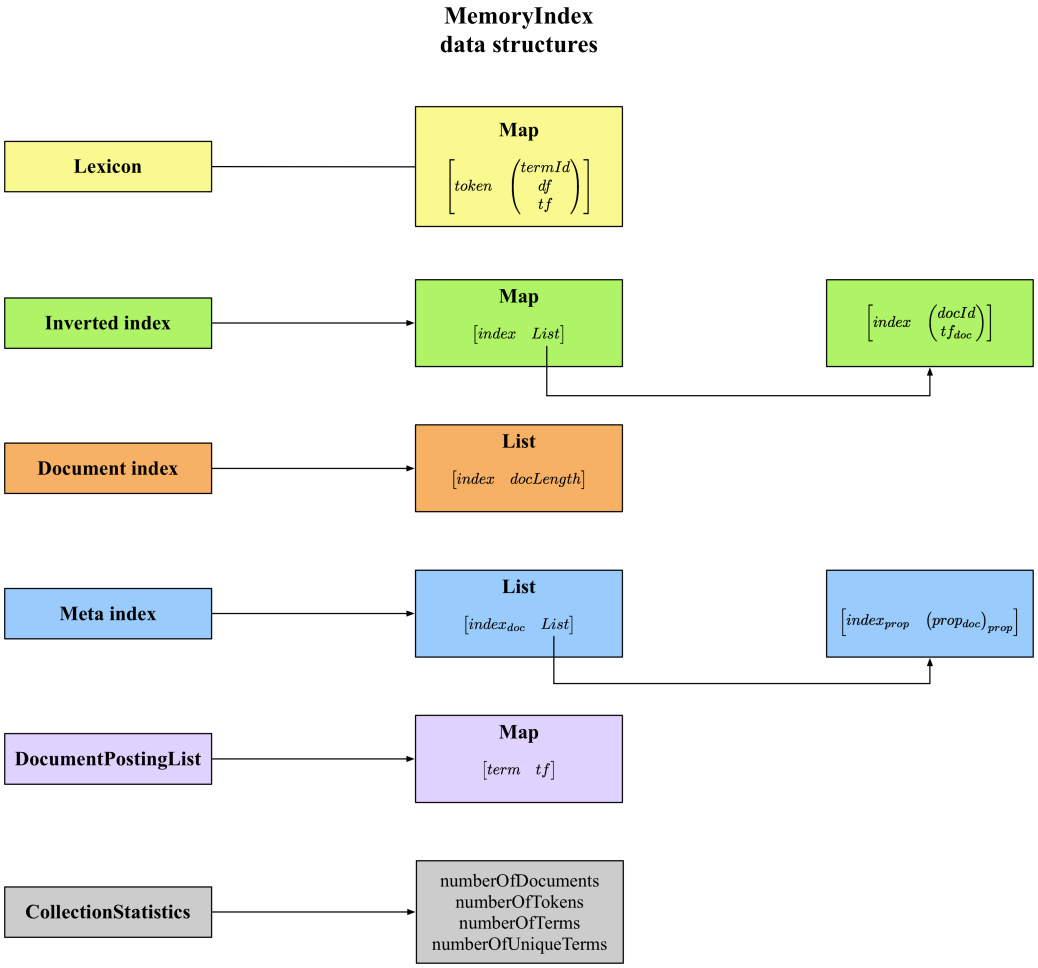

Figure 4.1. *In-memory data structures created during the indexing process.*

- *Inverted index*: it essentially corresponds to the set of *posting lists* described in Section 2.1.1. It is worth stating that despite the expression "Inverted index" is commonly used to refer to an index structure consisting of both a vocabulary and a set of posting lists, in the remainder, it will be used only to refer to a set of posting lists. In Terrier, the *Inverted index* is implemented by means of a map which connects each token with its posting list.

- *Document index*: for each document it stores the count of its tokens. It basically consists of a list, in which each position corresponds to a document identifier.

- *Meta index*: it includes some data for each document going through the indexing procedure. In our context, it maintains only the filenames of the indexed documents.

- *Collection statistics*: it is essentially a wrapper of a set of statistics about the indexed documents. They are the overall number of indexed documents, the number of unique terms in those documents, the overall number of pointers, which corresponds to the overall number of postings the *Inverted index* consists of, and the count of the tokens we analyze during the indexing procedure.

In Figure 4.1 we llustrate the memory-based index structures used for collecting the data taken out from the collection documents going through the indexing process. For each index structure, we provide the data structure it is essentially composed of.
In addition to these structures, the indexing process involves the *DocumentPostingList*, which stores token occurrences in a particular document before that it go through the indexing procedure. In Terrier, it consists of a map which connects each term occurring in a document to the count of its occurrences in that document. Each time we index a new document, a new map is initialized. In Terrier, the real-time indexing process is entirely carried out in memory. For each document going through the indexing procedure, we repeat these processing phases:

- *Generation of the document representation*: depending on the last part of the filename, which determines the extension of the file, we generate a different object. This element is responsible for representing the examined document during the indexing process. The type of the object is used to successfully decode the raw text composing the document and to convert the obtained stream of characters into a stream of terms. The generated object contains a map which keeps some properties about the examined document, which we include into the *Meta index*.

- *Extraction of the document terms*: this phase deals with the extraction of the terms composing the document. In this process, we perform different operations depending on the lexicon of the document. These are usually concerned with removing the punctuation marks as well as converting all the upper-case letters to lower-case ones.

- *Application of the procedures composing the indexing process to each token*: the procedures forming the indexing process compose a pipeline whereby we process each token of the document of interest. By default, this pipeline consists of two stages, the removal of the stopwords and the stemming process. The result of this stage, together with the *term frequency* of the token are inserted in the *DocumentPostingList*.

- *Insertion of the document properties in the Meta index*: we include all the document properties into the *Meta index*.

- *Inclusion of the document length into the Document index*: we add the count of the tokens occurring in the document of interest to the *Document index*.

For each token in the *DocumentPostingList*, we carry out the following operations:

- *Creation of a Lexicon entry and its insertion in the Lexicon*: first of all we generate a Lexicon entry related to the token. Its identifier is set to the value of a counter which is increased by one at each insertion in the *Lexicon*; its term frequency field is set to the the corresponding value of the considered token, kept into the *DocumentPostingList*, whereas its document field is set to one. For convenience sake, here we have supposed that the entry associated with the token has not been added to the *Lexicon* yet . If it is not the case, the corresponding entry is accordingly updated.

- *Generation of a posting and its insertion in the Inverted index*: we initialize a new posting regardless a new Lexicon entry has been created or not. In particular, its document identifier field is set to the number of documents indexed so far. The *posting* is then added to the posting list of the *Inverted index* associated with the identifier of the examined token. Note that, for each term, there is a posting for each collection document in which it occurs.

Finally, after taking into account all the tokens of a document, we accordingly update the index statistics.
In Figure 4.2 we schematically depict the different phases composing the indexing process.
After performing the indexing process for a document, two possibilities may occur depending on the strategy we have established for handling the data transfer from in-memory to disk-based index structures.
In the remainder, this policy will be denoted as *flush policy*. Setting this property means establishing the number of indexed documents, after which the data transfer to disk is carried out. Therefore, after carrying out the indexing process on a document, if we have not reached the threshold value of the flush policy, we go on applying the indexing procedure to the next document in the collection. Otherwise we initialize a new on-disk index and we move the data kept in memory to its disk-based structures. Despite on-disk index structures differ from those maintained in memory in the way in which they are actually implemented, their purpose and mode of operation are the same.

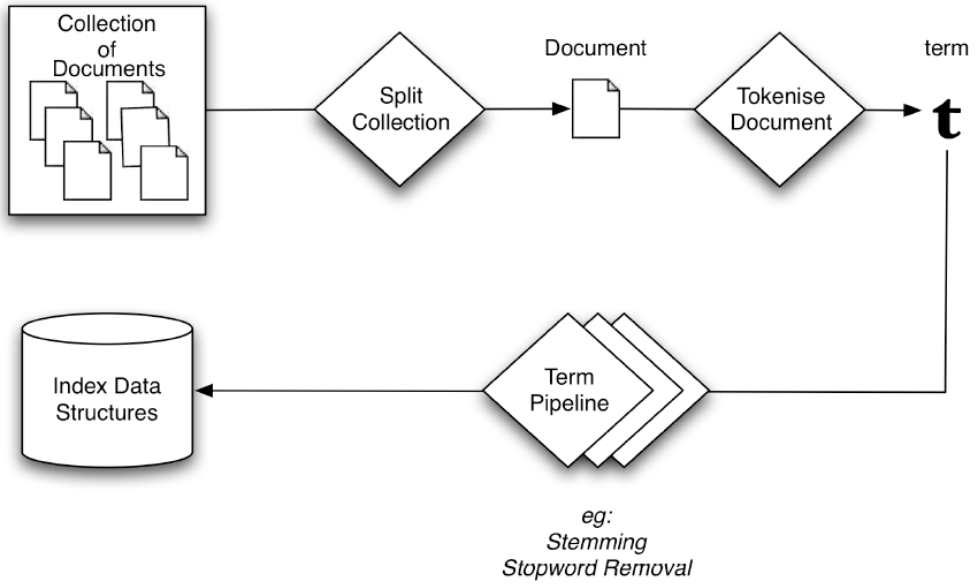

Figure 4.2. *Graphical representation of the indexing process. After a collection document has been taken into account, all its terms go through the pipeline procedures defined for the indexing procedure. Finally, the resulting data are stored into the in-memory data structures which have been depicted in Figure 4.1.*

For description sake, we will distinguish between the case in which we refer to the on-disk structures and the one in which we refer to the in-memory structures, by using the prefix "disk" for the first ones and the prefix "memory" for the second ones.
In what follows, we assume that the threshold associated with the flush policy has been reached, hence we need the data maintained in memory to be moved to disk-based data structures. To carry out the data transfer to the disk-based data structures, we carry out the following operations:

- *Generation of temporary in-memory Document index entries and their insertion in the disk-based Document index*: we generate an entry for each element occurring in the in-memory *Document index*. It specifies the number of tokens composing the corresponding collection document. Then we add this entry to the on-disk *Document index*.

- *Storage of the properties of each indexed document in the on-disk Meta index*: we add the filename of each collection document to the on-disk *Meta index*.

Then, for every token for which an entry exists in the in-memory *Lexicon*, we perform the following operations:

- *Taking out the token posting list*: we use the token as key for retrieving the *Lexicon* entry, the token is associated with. Then, we pull out its posting list from the identifier of the retrieved *Lexicon* entry.

- *Insertion of the postings comprising the posting list in the on-disk Inverted index*.

- *Generation of an entry for the disk-based Lexicon and its insertion in such data structure*: we initialize a new *Lexicon* entry from the one occuring in the in-memory index *Lexicon*. Specifically, this entry contains a pointer to the position of the posting list of the token of interest in the disk-based *Inverted index*.

Note that we take out the in-memory Lexicon entries according to the lexicographical order defined over the set of tokens, the Lexicon entries are associated with.

Once, the data are moved from in-memory to the on-disk data structures, we merge them with the ones already maintained in the on-disk data structures, with the aim of having a more compact index.
For completeness sake, we point out that the policy establishing how we want the data moved to disk to be managed is denoted as *merge policy*. After carrying out the data transfer from in-memory data structures to the corresponding disk-based ones, we generate another disk-based index. It will contain the data belonging to the two latest on-disk indexes. It is worthwhile to note that the latest on-disk index which was created, corresponds to the one initialized after the number of indexed documents has exceeded the threshold associated with the flush policy. Merging together the disk-based data structures, after generating a new on-disk index has the benefit of allowing us to access all the data maintained on-disk by inquiring an only disk-based index. Below, we review how the data included into two on-disk indexes are merged together. In what follows, we refer to the source indexes as "index1" and "index2", while we denote as "destination" index, the disk-based index to which we will add the incoming data. The merging procedure consists of two phases:

- *Merging the source Lexicon and Inverted index structures*: this phase deals with cyclically executing a sequence of operations, until we have inserted all the elements in index1 and index2 *Lexicon* data structures in the *Lexicon* of the destination index. The first step is concerned with picking up the first Lexicon entry of each source *Lexicon*. We remark that we pull out Lexicon entries according to the lexicographical order defined over the tokens associated with them. Then we compare the retrieved entries,

to establish which of them is the first that we must take into account. The second step involves storing the posting list associated with the examined Lexicon entry into the *Inverted index* of the destination index. Finally, we generate a new Lexicon entry from the one selected at the first step.

- *Merging the source Document and Meta index structures*: we sequentially include the *Document index* entries into the corresponding data structure of the destination index, before repeating the same procedure for the entries of the source Meta index structures.

4.1.2 Retrieval Process

In what follows, we suppose we are provided with a *query*, consisting of different terms, along with an index, obtained by applying the indexing process to a corpus of documents.
The retrieval process can be divided in three phases, which are:

- *Extracting the index data associated with the query terms.*
- *Computing the scores of the collection documents.*
- *Generating the ranking list from the scores assigned to the documents.*

In Figure 4.3, we can see a graphically representation of the retrieval process. Taking into account a query q, we match such query against the content of the index structures, to locate the documents which are likely to be regarded as relevant with respect to the given query q. Then, we generate a ranking list on the basis of the scores assigned to the collection documents.
In the remainder, we consider a document as "relevant", in regard to a query, if its score is greater than 0.

Firstly, with the aim of fullfilling the retrieval procedure, we have to specify which information retrieval model we must use for the score computation. In fact, more than one model can be indicated. At this stage, we only mention that they represent the mathematical framework whereby the mental process trough which a user decides whether a document is to be regarded as relevant or not has been conceived. Secondly, we process each term in the query by the pipeline defined for the indexing process. Then, the retrieval procedure go on using the resulting terms for retrieving the corresponding data from the index. The structures which are involved in this process are the *Lexicon*, the *Inverted index*, the *Document index*

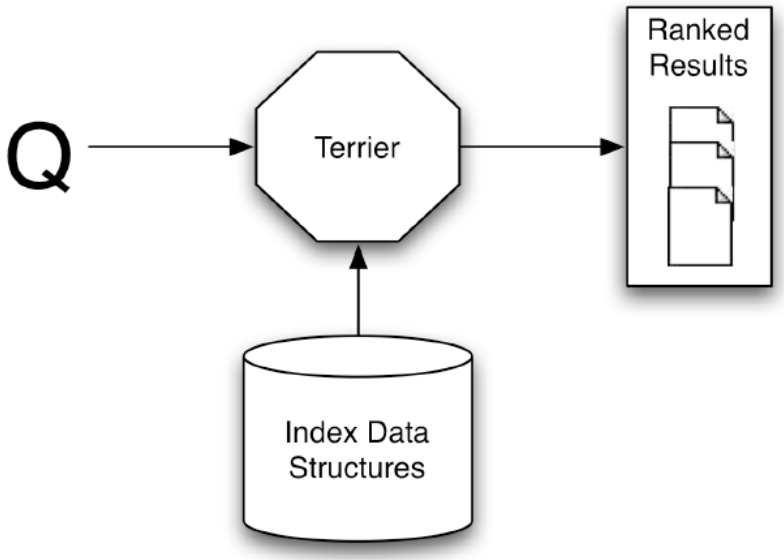

Figure 4.3. *Scheme of the retrieval process. The query terms are matched against the data stored into on-disk and in-memory data structures. From this comparison, a ranked list of the documents which are more likely to be regarded as relevant is obtained. The documents in the ranked list are sorted in descending order of their scores.*

and the *CollectionStatistics*. More specifically, we use each query term as key to pick up the *Lexicon* entry it is coupled with. We remark that for an index term the corresponding *Lexicon* entry specifies its identifier, the number of documents in which it occurs in, denoted as document frequency, the count of the occurrences of that term we refer to as term frequency and a pointer to the position of the first posting belonging to its posting list in the *Inverted index*. We remind that the index term is simply the result of the application of the indexing process to a token. Thirdly, we use the data associated with the retrieved Lexicon entries to pull out the posting lists associated with the query terms. Then, we evaluate a score for each posting occurring in one of the retrieved posting lists, accounting for the relevance of the document, the identifier of the posting refers to. For computing the score of a posting, in addition to the count of the occurrences of the query term in the document of interest, we also take into account some statistics about the examined collection as well as others about the considered document. They are: i) the document frequency of the query term ii)the number of indexed documents iii) the number of tokens comprising the examined document. In case, we specify more than one information retrieval model, we compute the overall score of a posting by adding up the values computed for each of the specified information retrieval models. The score of a document is given by the sum of the scores associated with the query terms

occurring in that document. For the score computation, Terrier provides implementations of many information retrieval models, including: i) BM25, the BM25 probabilistic model [20], ii) IFB2, the Inverse Term Frequency model for randomness, iii)TF-IDF, the tf-idf weighting function, where tf is given by Robertson's tf [21] and idf is given by the standard Sparck Jones' idf [22]. We point out that most of the implementations of the information retrieval models, available in Terrier, contain some parameters we can adjust according to our needs.

In the experimental phase, we have adopted the TF-IDF information retrieval model. Computing the score with this model involves multiplying two components. The former deals with the relevance of a query term for the document of interest, while the latter accounts for the query term capability of discriminating the analyzed document from the others in the collection.

4.2 Indexing and retrieval processes: some implementation details

This purpose of this section is to give some information on how the real-time indexing and retrieval processes are implemented in Terrier. Since we reported a general description of these topics in Section 4.1.1, below we only provide some details on the main phases comprising the examined processes. Additionally, to better explain how the indexing and retrieval processes are actually carried out in Terrier, we give a complete example of the issues presented in Section 4.1.1 and 4.1.2, highlighting the key ideas underlying the described processes. In the example we take into account throughout this section, the documents in the collection are pdf ones. It is useful to remark that for what concerns the indexing and the retrieval processes nothing would have changed if we had considered documents of a different type.

4.2.1 Real-time indexing process

The first thing we need to do in order to index some documents is to initialize an *IncrementalIndex*. This object is responsible for performing the indexing procedure. Using such an object means especially specifying the flush policy and the merge policy. It is useful here to remind their meaning.

- *flush policy*: it defines the number of documents which are indexed before the data maintained in-memory are moved to disk-based data structures.
- *merge policy*: it establishes how the disk-based indexes, generated during the indexing process, are handled.

The first phase deals with the creation of a document as well as the insertion of its terms in a *DocumentPostingList*. The several stages, this phase consists of, are shown in Figure 4.4. Firstly, the filenames of the documents in the collection, we are interested in, are added to a *SimpleFileCollection*. This object basically implements the concept of collection of documents, in which each document is stored in a different file. Note that the documents of a *SimpleFileCollection* may be of different types. Futhermore, the *SimpleFileCollection* associates each filename with a class which will be used for parsing the corresponding document. Specifically, it allows us to encapsulate the document content in an object of a specific type. In our context, the *SimpleFileCollection* basically let us iterate over the filenames of the collection documents and for each of them we create an instance of a class extending the *Document* one, through the *getDocument* method. Note that this class represents the concept of generic document, whereas all the classes extending it provide the object representing a collection document with features related to a specific class of documents. Then, for each *Document* we sequentially take out its terms through the *getNextTerm* method, before processing them through the pipeline defined for the indexing process, we reviewed in Section 2.2. We specify that each stage of the pipeline is actually implemented by a *TermPipeline*, an object being responsible for performing the operations of a specific processing stage on each term. The processing steps generally involved in the indexing process are: i) lexical analysis, ii) stopwords removal, iii) stemming process, iv) term composition. We provide the result of each pipeline stage as input to the next one. Note that, if a term does not constitute a valid input for a pipeline procedure, we discard it. The last step of the pipeline deals with including all the resulting terms into a *DocumentPostingList*. We remind that such an object basically consists of an map which associates the terms resulting from the application of the pipeline to the document words with their frequencies in that document. In Figure 4.5 we illustrate the implementation of the indexing process on a document composed of the statement “Hello World! I am Andrea and I am twenty-three years old”. The numbers, the rectangles are labeled with, specify the order with which the memory-based structures presented in Figure 4.1 are involved in the indexing process. The entry $[hello, 1]$ occurs in the rectangle *occurrences*. The content of such shape

reflects the state of the *DocumentPostingList* after all the tokens of the document go through the procedures of the pipeline, established for the indexing process. Looking at the entry $[hello, 1]$, it is linked with the element whose token is equal to $hello$ in the rectangle *map*. Such shape corresponds to the memory-based Lexicon. We can observe that its *df* variable is equal to 1, since the term "hello" exists in the only document of the collection. The value of the field *tf* equals the value of the corresponding field of $[hello, 1]$, whereas the *termId* variable is equal to 0, since the $[hello, 1]$ is the first element of the *DocumentPostingList* analyzed. Note that the values of the *termId* and *tf* fields, along with the number of indexed documents, define the fields of the posting associated with the examined Lexicon entry.

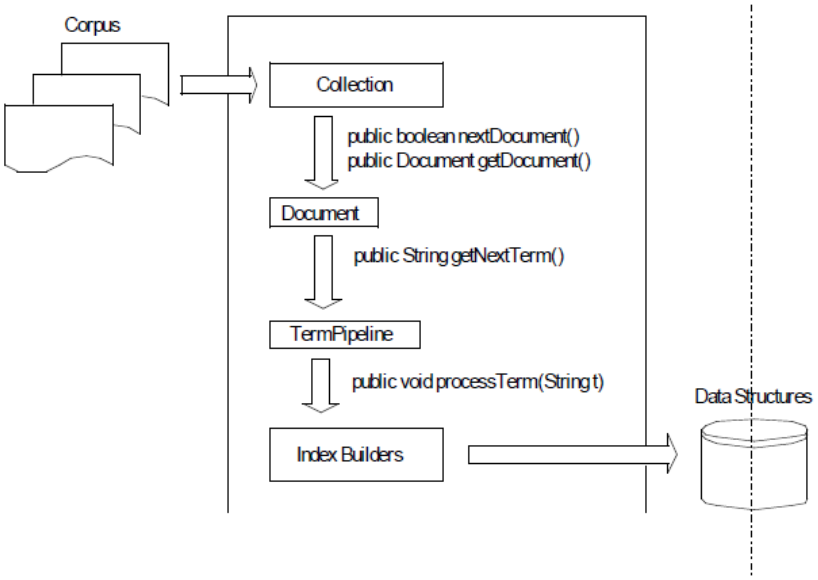

Figure 4.4. *Interaction between the main components taking part in the indexing process.*

Below, we present how the *Lexicon* and *Inverted index* entries are moved from the in-memory to the disk-based index structures, because of the implementation of the flush policy. It is useful to remind that after the number of indexed documents has exceeded the threshold defined by the flush policy, we initialize a new disk-based index, into which we include all the data mantained in the central memory at that time. First of all, we point out that we handle the in-memory and on-disk data structures through instances of two classes, which are named *MemoryIndex* and *IndexOnDisk*. They actually implement the concept of index maintained in the central memory and on disk, respectively. For each entry in the *MemoryIndex Lexicon*, we perform the following operations:

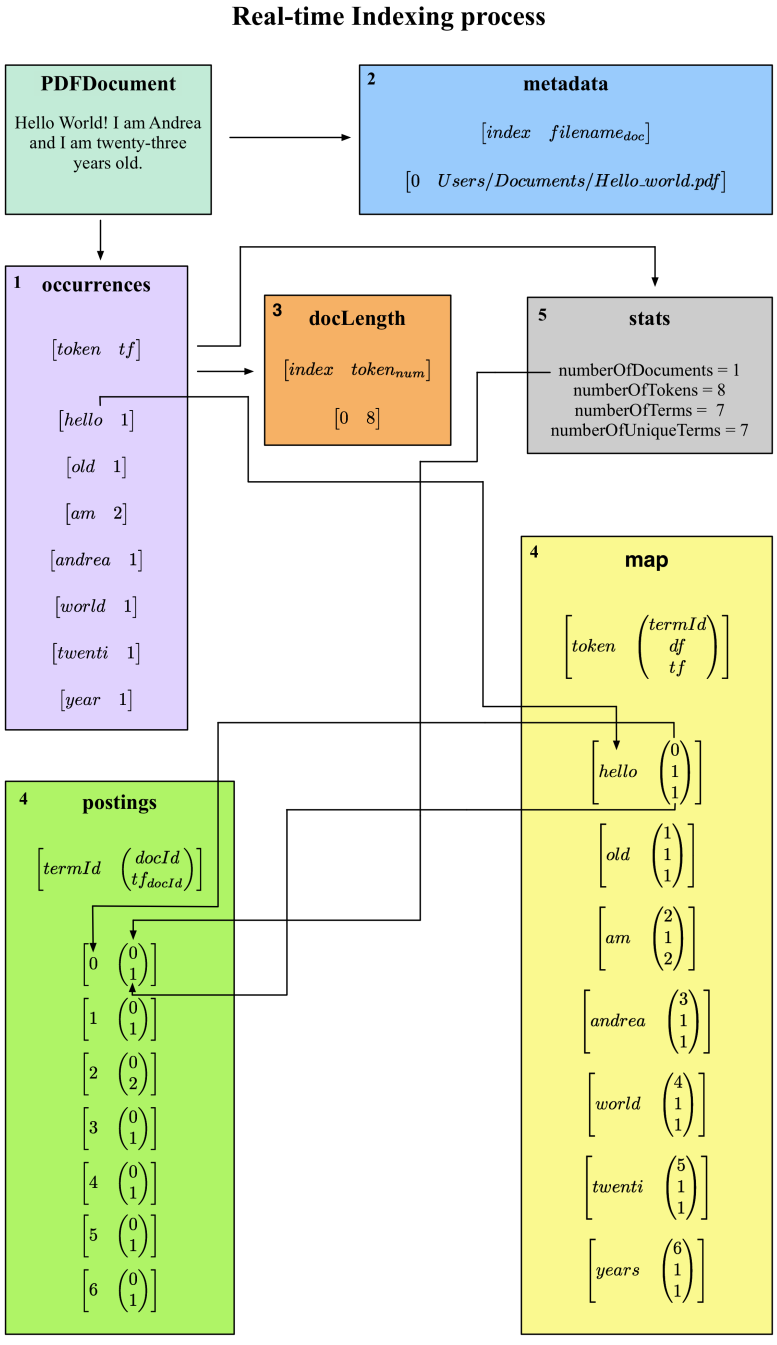

Figure 4.5. *Representation of the operations composing the first phase of the Terrier real-time indexing process, which is concerned with the insertion of the data of the collection documents in the in-memory data structures. In this figure, a simple example is analyzed. It consists of the execution of the indexing process on a document which actually contains a single statement: "Hello World! I am Andrea and I am twenty-three years old".*

- *Taking out the posting list related to the entry*: we use the entry identifier as key to pull out the posting list of the entry from the in-memory *Inverted index*. This data structure essentially consists of an map of posting lists, each of them coupled with an index term.

- *Storage of the elements of the posting list in the disk-based Inverted index*: we sequentially add the data of the *postings* contained in the posting list to the stream of bytes the disk-based *Inverted index* basically consists of. Specifically, the document identifiers are encoded by using the *Elias Gamma coding* [16], while the term frequencies are encrypted by using the *Unary coding* [18]. With the aim of decreasing as much as possible the amount of disk space we need to maintain the whole posting list on disk, we save the identifiers of the postings by storing *d-gaps* instead of the straight document identifiers. We point out that a d-gap is simply the difference between the identifiers of the documents to which two adjacent postings refer to. Note that to put this strategy into practice, we need the postings of a posting list to be arranged in ascending order with respect to their identifiers. In fact, during the indexing process we assign increasing integer values to the collection documents, thus ensuring that the above property is retained.

- *Generation of a new Lexicon entry and its insertion in the IndexOnDisk Lexicon data structure*: we generate a new *Lexicon* entry. We set its identifier, tf $\big($the term_frequency$\big)$ and n_t $\big($the document_frequency$\big)$ fields to the corresponding values of the fields of the analyzed *MemoryIndex Lexicon* entry. Futhermore, we enrich it with a pointer referring to the position in the disk-based *Inverted index*, in which the first posting associated with the examined entry is stored. The pointer is implemented by means of a long and a byte fields.

The way in which we handle the *Document index* and *Meta index* entries can be easily deduced from the overview of the indexing process we have provided in Section 3.1.
In Figure 4.6, the data which in Figure 4.5 have been stored in the memory-based index structures are moved to the corresponding on-disk ones. As in Figure 4.2.1, the numbers the shapes are labeled with indicate the order with which the index structures are involved in the flushing operation. Looking at the in-memory Lexicon entry, whose token is am, which appears on the left side of the figure, we can note that its data are used for extracting the postings included into the in-memory *Inverted index*, associated with the term am. The data

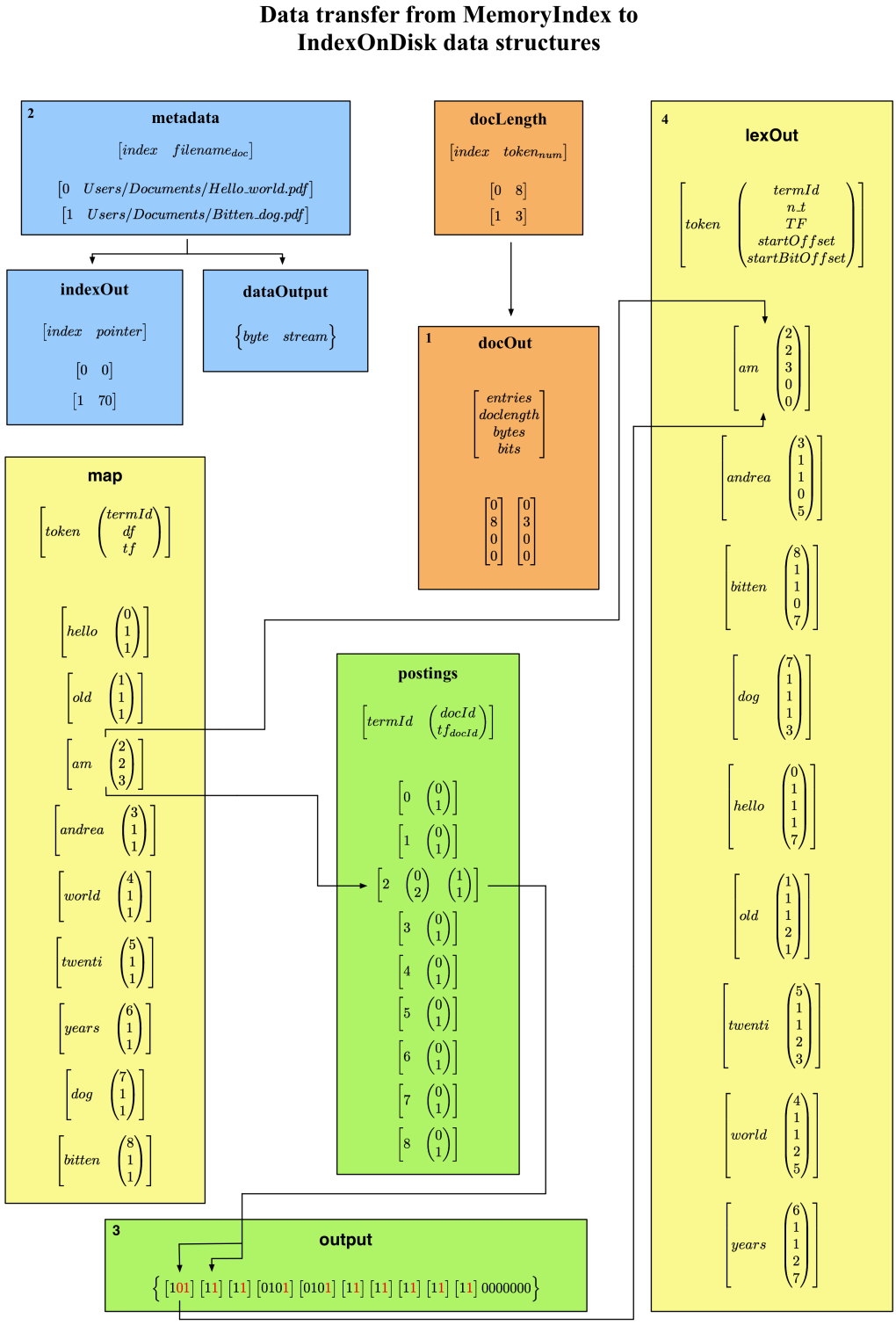

Figure 4.6. *Transfer from the in-memory data structures to the disk-based ones of the data collected by applying the indexing procedure to a couple of documents, both composed of an only statement. The first one consists of the statement "Hello World! I am Andrea and I am twenty-three years old", while the second one is "I was bitten by a dog".*

comprising such postings are encoded by means of the Unary [18] and Gamma [16] encoding, before being added to the disk-based *Inverted index*.
Here, we proceed to illustrate how the data of two *IndexOnDisk* are merged together into a new one. Hereafter, the same nomenclature which we used in the previous section is adopted: we refer to the former *IndexOnDisk* as "index1", while to the latter as "index2". Finally, we indicate the *IndexOnDisk* which is intended for storing the data coming from the first ones, as "destination" index.
The merging procedure of two disk-based indexes is principally composed of two phases. The former deals with merging the source *IndexOnDisk Lexicon* and *Inverted index* data structures, while the latter is concerned with putting together the data coming from the source *Document index* and *Meta index* structures. Here, we give some details of the first phase. Since many operations performed during this stage are very similar to those we have described above, we focus on the elements which have not been presented yet, by only mentioning the issues we have already taken into account. Therefore, to merge together the "index1" and "index2" Lexicon and Inverted index structures, we carry out the following operations:

- *Extraction of the first entry from each of the source IndexOnDisk Lexicon data structures*.

- *Comparison of the Lexicon entries to establish which is the first entry we must insert in the destination IndexOnDisk Lexicon*. We carry out the comparison by using the lexicographical criterion over the terms the examined entries are related to.

- *Taking out the posting list, associated with the Lexicon entry, selected at the previous step*. We load the inverted list in the central memory, before encapsulating it by means of a *BasicIterablePosting*. This object is responsible for letting us access the *postings* forming the analyzed posting list. In particular, the *BasicIterablePosting* consists of a buffer, storing the raw data associated with the posting list and an integer variable, whose value is set to the corresponding value of the *n_t* field of the *Lexicon* entry. Note that this field specifies the number of postings comprising the analyzed inverted list. We use this variable to read all, but only the postings associated with the examined *Lexicon* entry.

- *Storage of the inverted list postings in the Inverted index data structure of the destination IndexOnDisk*. We sequentially add the postings composing the inverted list

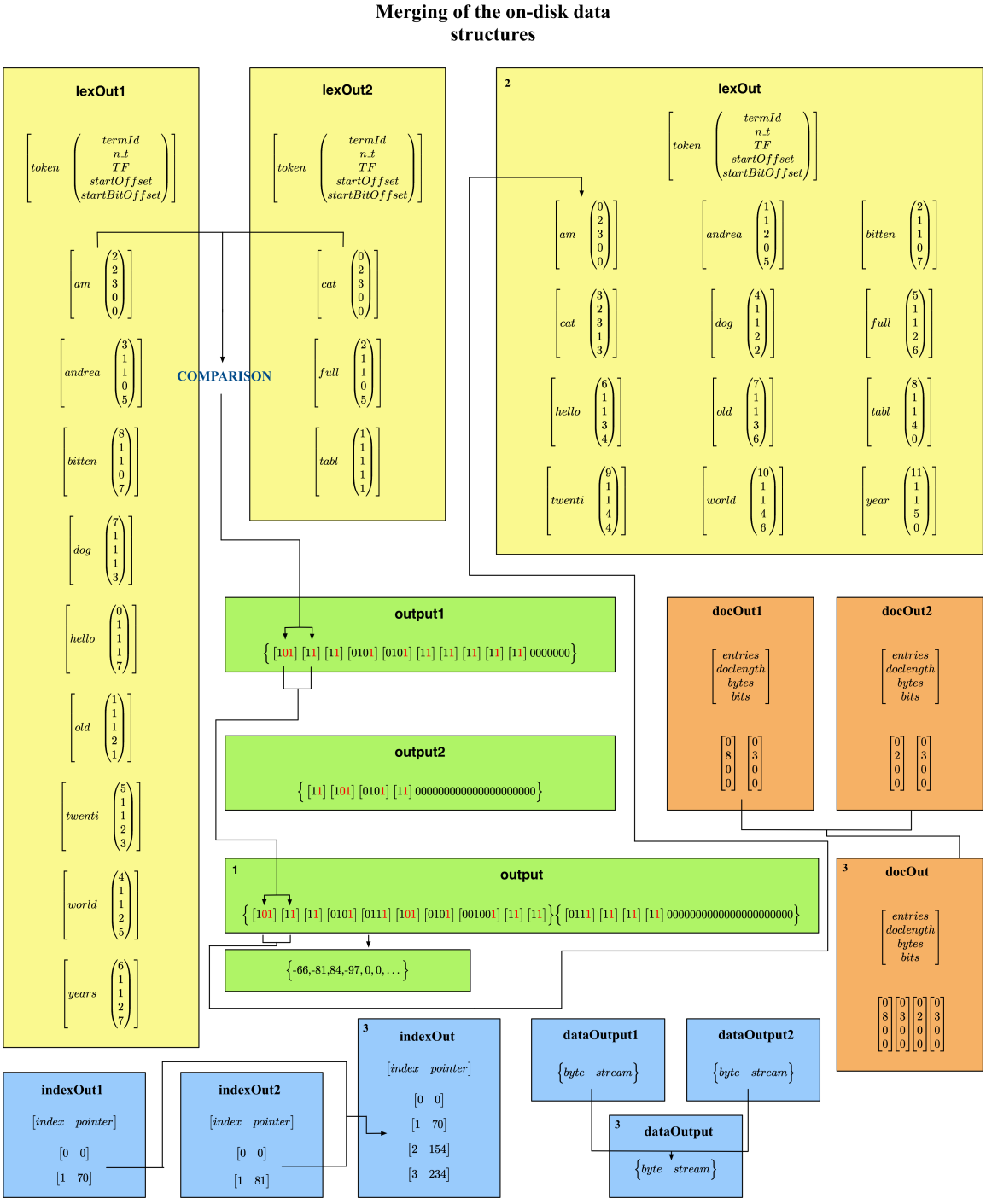

Figure 4.7. *Merging process of the data structures of two on-disk indexes. Both the examined indexes have been obtained by carrying out the operations depicted first in Figure 4.5, and then in Figure 4.6. The index structures of the IndexOnDisk of interest are distinguished from each other through a numerical identifier.*

of interest to the byte stream the destination *IndexOnDisk Inverted index* consists of. We point out that, before being stored in the byte stream, the identifiers in the postings are adjusted to avoid that more than one posting have the same identifier. To achieve this goal, we increase the identifiers of the postings maintained in the index2 *Inverted index* by the number of documents whose data are stored in the index1 *IndexOnDisk*.

- *Generation of a new Lexicon entry and its insertion in the corresponding data structure of the destination IndexOnDisk.*

In Figure 4.7, we show how the merging operation of two disk-based indexes is performed. We remind that such process results in generating a new on-disk index, to which we add all the data retrieved from the indexes involved in the merging procedure. In the upper part of the figure, we can note that the first entry of the index1 and index2 Lexicon structures are compared with respect to the lexicographical order, to establish which is the first entry whose data must be included in the newly created disk-based index. Since the term am precedes the term cat in the lexicographical order, we take into consideration the corresponding Lexicon entry. Specifically, its data are used for pulling out the postings associated with the term am from the index1 *Inverted index*. These data are loaded in the central memory and then encoded again before being stored in the *Inverted index* of the generated disk-based index.

4.2.2 Retrieval process

This section presents some details on how the retrieval process is actually implemented in Terrier. In what follows, we will suppose that a query consisting of different terms as well as an *IncrementalIndex* storing data associated with some documents have been provided. Firstly, a *Manager* object is initialized. It is responsible for carrying out all the operations which are somehow related to the processing of the examined *query*. In particular, the *Manager* specifies the procedures which are to be executed on the query terms. After an initialization phase, the same pipeline which was used for processing the tokens, during the indexing process, is taken into account. The only difference is in the last stage: in the indexing process it consists of the insertion of the resulting term in a *DocumentPostingList*, while in the retrieval procedure, it deals with assigning the result of the performed operations to a *Query* object. It represents the single query term a query is composed of. Therefore, a query is implemented by means of a set of *Query* objects, which is encapsulated in a *MultiTermQueryObject*. A *MultiTermQueryObject* is basically an iterator over the *Query*

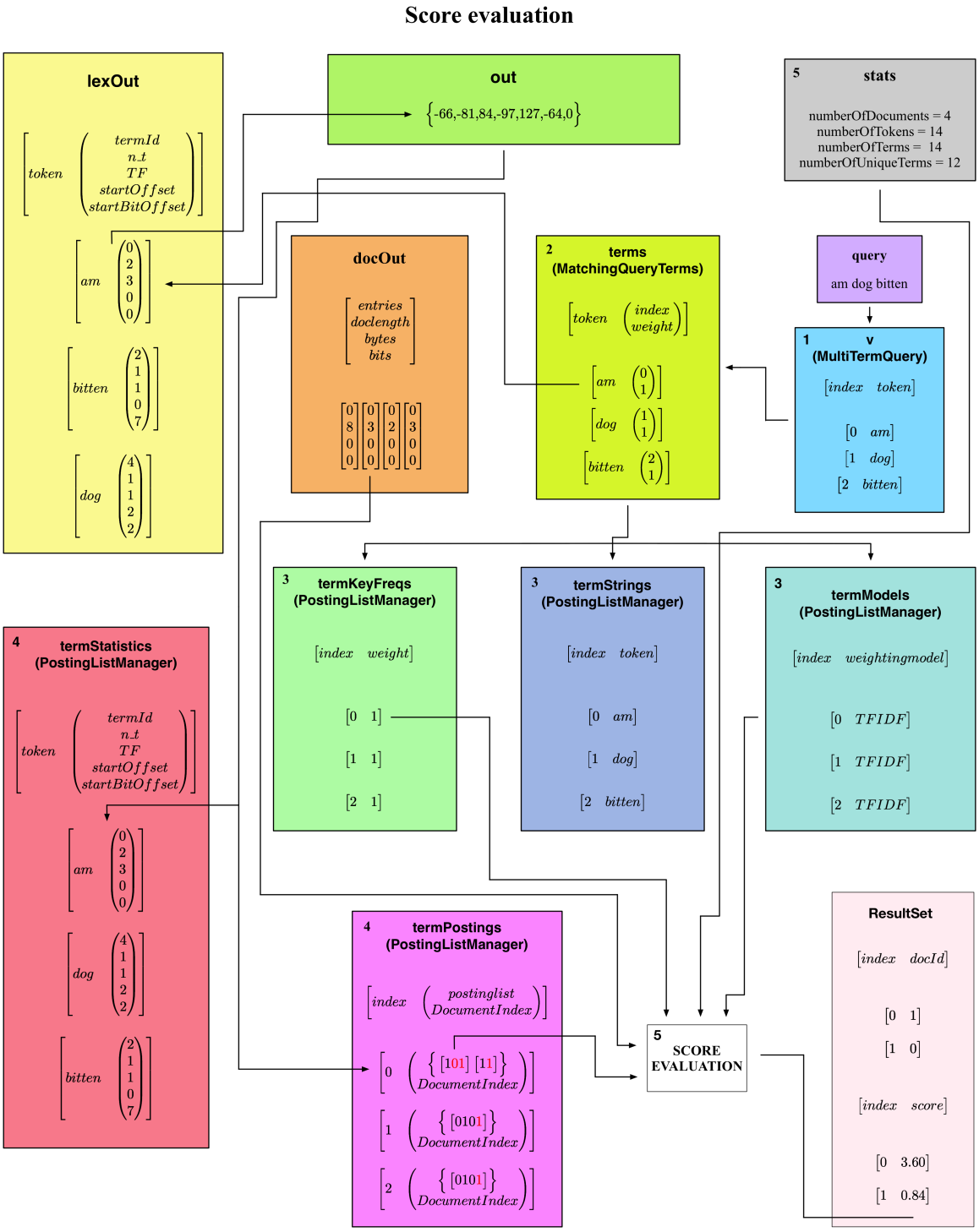

Figure 4.8. *Representation of how the retrieval process is performed in Terrier, assuming that a query, consisting of the statement "am dog bitten", is provided. The Lexicon, Inverted index, Document index structures as well as the CollectionStatistics have been obtained by applying the operations which has been graphically presented in Figures 4.5, 4.6, 4.7. This figure depicts the score computation for the posting* $\begin{pmatrix} 0 & 1 \end{pmatrix}^T$ *associated with the index term* am.

objects it is made up of. Before the given query is evaluated, a *Request* object is generated. It contains the controls which determines the operations which have to be performed on the given query as well as a *ResultSet* object. It will include the ranking list of the collection documents, once the examined query has been evaluated. The retrieval process is performed by the aforementioned *Manager* object. It is made up of four procedures which are listed below in the same order with which they are actually carried out.

- Pre-processing
- Matching
- Post-processing
- Post-filtering

The pre-processing stage is concerned with the application of the procedures which are determined by the *Request* object. In particular, this phase deals with the application of the pipeline, defined for the indexing process, to each of the query terms. As we pointed out beforehand, for each resulting term a Query is generated. The entire query is represented by means of a *MultiTermQuery*. Additionally, this phase involves the initialization of a *MatchingQueryTerms* object, which keep for each query term, its number of occurrences within the given query, its weight $\big($unless otherwise specified it coincides with its number of occurrences within the query$\big)$ as well as the information retrieval model which must be adopted to evaluate the contribution to a document score provided by the examined term. The matching phase is the most important stage of the retrieval process. In it, the computation of each collection document score in regard to the query, we took into account, is actually carried out. In fact, the score is evaluated only for the documents which contain at least one of the query terms. First of all, a *PostingListManager* is generated. It is responsible for gathering the data coming from the *InvertedIndex* and *MatchingQueryTerms* together. In detail, for each query term it keeps its weight, its statistics data, made up of the *Lexicon* entry belonging to the *Lexicon* data structure and finally its posting list. Since, we took real-time indexing and retrieval processes into account, it is worth noting that the data, *PostingListManager* data structures are composed of, may derive both from on-disk data structures and in-memory ones. It is useful to specify that the inverted lists are managed in such a way that at each step of the score evaluation procedure, the posting list, whose next posting has the lowest identifier is taken into account. Note that if the next postings of two posting lists have the same identifier,

is analyzed the one whose query term comes first in the query. The way in which score evaluation process is carried out involves that the scores are computed sequentially, avoiding that the same document is considered more than one time during its execution. The process ends when all the postings of all the posting lists have been considered. At this point, all the scores and the identifiers of the documents they are associated with are maintained in the *ResultSet*, related to the *Request* object. Post-processing phase goes on to process the *ResultSet*, obtained in the previous phase, by sorting the identifiers of the relevant documents in descending order with respect to their scores. Instead, the post-filtering phase is responsible for filtering out the obtained results. A detailed description of these stages is beyond the scope of this section. However, further information on this topic can be found in [14].
In Figure 4.8, we can observe how query evaluation is carried out. As shown in Figure 4.8, for each of the terms composing the query, we generate a *SingleTermQuery*, an object responsible for mantaining all the information related to the score evaluation for that term, as the number of occurrences of such term in the query and the model we have to take into consideration in computing the scores of the documents in which that term occurs in. The information associated with all the query terms are then gathered together in the *MatchingQueryTerms*. Looking at the entry $[am \quad \binom{0}{1}]$ in the *terms* rectangle, we can note it is linked with the disk-based Lexicon entry related to the index term am. The fields of such Lexicon entry are used to extract the posting list of the index term am from the Inverted index, which in Figure 4.8 corresponds to the rectangle out. We include such inverted list into *TermPostings*, in the position corresponding to the one of the term am within the query "am dog bitten". Note that the postings composing the inverted list are added to the *TermPostings* as raw data. The data contained in the *MatchingQueryTerms*, are moved to the *termKeyFreqs*, *termStrings* and *termModels* structures of the *PostingListManager*, represented by the green, blue and light blue shapes, depicted in the center of the figure. At each stage of the retrieval process, after decoding the raw data associated with the posting of the inverted list taken into account, the data maintained in the *PostingListManager* structures related to the query term of interest are used for computing the score of the examined posting. In fact, we point out that to evaluate the score of a posting we also take into account the collection statistics and the length of the document the posting refers to, which is included into the Document index, depicted in the figure 4.8 by the rectangle *docOut*.

5 BA and MA strategies

5.1 Basic Approach and Mapping Approach: mode of operation

5.1.1 Indexing process

This section outlines the changes we needed to make to the Terrier real-time indexing process in order to handle versioned collections. Terrier does not allow us to take into account different versions of the same document. In fact, during the indexing process these are managed as if they were completely different collection documents. Another problem which arises by considering temporally versioned collections is how to deal with temporal data, we enrich collection documents with. Moreover, to support search over such a collection, we need to establish a policy for computing and storing statistics data about the collection, which are used for score computation. First of all we explain the ideas underlying the strategies we have conceived. The Basic Approach (BA) for index versioning consists of a baseline approach for index versioning which enables search over temporally versioned collections by filtering out postings which do not adhere to a certain temporal constraint. This strategy implies that during the retrieval process we scan the postings lists related to the query terms, searching for valid postings. A posting is considered as valid if the temporal range attached with it satisfy the query temporal predicate. In the case a posting is not valid, it is ruled out and its payload does not contribute to the score of the corresponding document. MA stands for Mapping Approach for index versioning. Compared to the BA strategy, it deals with exploiting the position assigned to each document within the stream

of bytes of the inverted index, for speeding up the retrieval process. Note that, in order to carry out this approach, we need to store the postings related to different versions of the same document in contiguous positions within the inverted index. Before explaining how we countered the aforementioned issues, we proceed to describe the context we took into account for implementing our strategies. We remind that in Terrier, the real-time indexing process is divided in three different phases:

- *Indexing phase*: it deals with indexing the collection documents. The data collected during this phase are temporary stored into in-memory data-structures.
- *Flushing phase*: after the number of indexed documents exceeds a defined threshold, which is referred to as flush policy, data within the in-memory data structures are transferred to the corresponding disk-based data structures.
- *Merging phase*: once the Flushing phase has been carried out, on-disk data structures are merged together to maintain an only index, through which to access all the data of the collection documents. This operation is actually performed according to a policy which is denoted as merge policy.

It is worth noting that the data transfer from in-memory to on-disk data structures and the subsequent merging of the on-disk data structures are unavoidable operations, since we need to perform them for practical needs. In fact, for each collection document, the indexing process deals with generating the representation of its content, which is used during the retrieval process for computing the document score. This generally results in the generation of a huge volume of data which can not be maintained in the central memory. This holds true especially for temporally versioned collections, since we take into account the evolution of the collection documents over time, thus generating an even larger amount of data. Carrying out the indexing process by using in-memory data structures allow users to extend the collection taken into consideration after some queries have been evaluated. This functionality lets us analyze a new version of a document occurring in a collection of interest, thus keeping up with the evolution of such document over time. Formally, the goal of the implemented approaches, in regard to the indexing process, is to organize the data collected during the indexing process in such a way that they can be retrieved efficiently during the retrieval phase. Since in the experimental phase we have looked at temporally versioned collections, we exploited the temporal data of the documents to better organize the index data, with the purpose of speeding-up query evaluation. The first issue

we have addressed was to allow different versions of the same document to be recognized as successive versions of such document. To achieve this goal, we have assigned two values to each document version, which we have referred to as *realId* and *tempId*. The former consists of the identifier assigned to the document, the examined version belongs to, while the latter indicates the number of document versions indexed before the analyzed one. With the purpose of assigning these values to each document version, we have defined two objects: *MemoryMetaName* and *MemoryVersionDocId*. The *MemoryMetaName* essentially consists of a map which associates each document name with an integer identifier, while the *MemoryVersionDocId* is responsible for connecting each *tempId* with the identifier of the corresponding document. Therefore, as a version of a document goes through the indexing process, both the BA and MA approaches involve inserting its name, namely the last part of its filename in *MemoryMetaName*, obtaining the identifier related to the corresponding document. Then we include the retrieved value into *MemoryVersionDocId*, getting the actual identifier of the version of interest.

The second issue we have tackled was about with the evolution of the collection statistics data over time. We remind that during the execution of the Terrier real-time indexing process we compute some statistics about the collection of interest, which we use during query evaluation for computing the scores of the collection documents. These statistics are: (i) the number of indexed documents, (ii) the number of postings in the inverted index, (iii) the sum of the number of terms comprising each collection document, (iv) the overall number of different terms comprising the lexicon. During the Terrier real-time indexing process, we adjust the values of these variables as soon as a new document is indexed. In particular, while we update the first three statistics during the indexing phase, by analyzing the statistics singularly computed for each document, adjusting the count of the different terms in the collection requires us to regularly evaluate the number of entries comprising the index Lexicon. In the case we take into consideration a temporally versioned collection, keeping the collection statistics updated, as the documents in the collection evolve over time, is more challenging. During the experimental phase, we have provided each document version with an integer value reflecting the time point in which we took into account such version to index it. In fact, we expect to have different values for the collection statistics for each of the following cases:

- *Indexing a new collection document*: this case requires us to update all the collection statistics, incrementing them by the values computed for the new document.
- *Indexing a new version of a previously examined document*: we must adjust the

collection statistics by taking into account the differences between the statistics evaluated for the current and the previous version of the document.

- *Removing a document from the collection*: this case actually reduces to the previous one. It requires us to adjust the collection statistics, by decreasing them by the values computed for the previous version of the document.

Note that, for what regards the update of the collection statistics, both performing the indexing process on a new version of a collection document and removing an existing document from the collection involves maintaining the statistics computed for the previous versions of the collection documents over time. In detail, both the BA and MA strategies involve including the statistics computed for the actual versions of the documents into a list. In the remainder of this section, it will be referred to as *CSList*. Therefore, adjusting the collection statistics, after a new version of a collection document goes through the indexing process, firstly requires us to pick up the values related to the previous version of the document from the *CSList*. Then we match these data against the ones associated with the new version of the document, before updating the collection statistics. Finally, we insert the new data in *CSList*, thus replacing the ones associated with the previous version of the examined document. It is interesting to note that by using this incremental strategy, we can not index out-of-order document versions. In other words, to correctly save the collection statistics, while the collection documents evolve over time, we can not take into account a document version after indexing a more recent version of the same document. Although at a first glance this constraint can be thought of as rather restrictive, it is not. Indeed, the indexing process should represent the evolution of the collection documents as they change over time. Therefore, it is quite plausible to analyze the versions of the collection documents on date basis. At this stage, we consider useful to propose two observations. The former is about the time interval each document version is attached with. The duration of the time interval of a document in a temporally versioned collection strongly depend on the collection we are interested in. Specifically, it depends on the frequency with which newer versions of the documents in the collection go through the indexing process. Therefore, two consecutive versions of the same document could represent the change of that document occurring in a time range of an year, a month, a day, of only few hours, depending on the goal for which the collection is maintained. The latter observation is about the meaning of "collection statistics". In fact, even if in Chapter 4, we have supposed all the documents of a corpus belong to the same collection, it is not the general case: we could handle the documents coming from more than one collection. We remark that the term

"collection statistics" actually does not refer to the stastics computed for the documents of a specific collection, as the term would seem to suggest us, but it refers to the statistics data evaluated taking into consideration all the documents we have indexed, regardless the number of collections we have adopted during the indexing process.

The last issue we have to deal with in this section is the managing of the postings composing the inverted lists. Firstly, it is useful to propose again the theoretical model we refer to in explaining how postings are handled in the BA and MA strategies. It entails each posting is represented by means of a set of three elements. They are:

$$\big(d, f, [t_i, t_{i+1}]\big) \tag{5.1}$$

where d is the integer value associated with a document version, f is the count of the occurrences of the term, the examined posting is associated with, in d, whereas $[t_i, t_{i+1}]$ is a time validity interval. After a new version of a collection document undergoes the revised indexing process, the BA and MA strategies are constituted of, the right boundary of the time intervals of their postings is set to 1. We remind that in our context the usage of the value 1 is twofold. It constitutes the lowest value we can assign to a document version, but as right boundary of a time interval, it indicates that the posting the examined time interval is attached with, refers to the latest version of a collection document. This means that before these postings are matched against the ones, we have previously generated, their validity interval ranges from t_i to ∞. The BA and MA approaches significantly differ in the way in which postings are managed during the Indexing phase as well as the Flushing and Merging ones. We first illustrate how we handle postings in the BA strategy, before moving on to the MA strategy. The BA approach involves that postings, collected during the Indexing phase, are stored into a memory-based list, according to their arrival time. Then, during the Flushing phase, we add them to the on-disk inverted index, which we have generated after the number of indexed documents has exceeded the threshold related to the flush policy. Instead of storing the identifiers of the document versions as they are, we adopt the d-gap technique. Specifically, instead of encrypting the identifiers of the document versions as they are, we prefer encoding the differences between identifiers of adjacent versions. This results in significantly reducing the amount of space required for storing the disk-based inverted index. We point out that the identifiers of the document versions are encrypted by using the Elias Gamma coding [16]. The same encoding is used for storing the two boundaries of the time interval, the postings are associated with. It is worth noting that before being encoded,

the time interval right boundary is checked for verifying if a newer version of the document of interest exists in the collection. If this case occurs, the right boundary is set to the value of the left boundary of the interval associated with the version we have found. In fact, this check is performed also in the Merging phase and during the retrieval process. This ensures that regardless the time point of a query, issued to information retrieval system implementing the BA approach, all the postings analyzed during the retrieval process are valid. Note that the time interval spanned by the union of the time ranges associated with all the versions of a document ranges from 1 and $\infty]$. The frequency value f is encoded by using the Unary [18] encoding. In the BA strategy, with the exception of the time interval boundaries, the data comprising the posting are stored by exploiting the same techniques used in the real-time Terrier indexing process. The Figure 5.1 shows how the postings data are first encoded and the inserted in the inverted index, with reference to the BA approach.

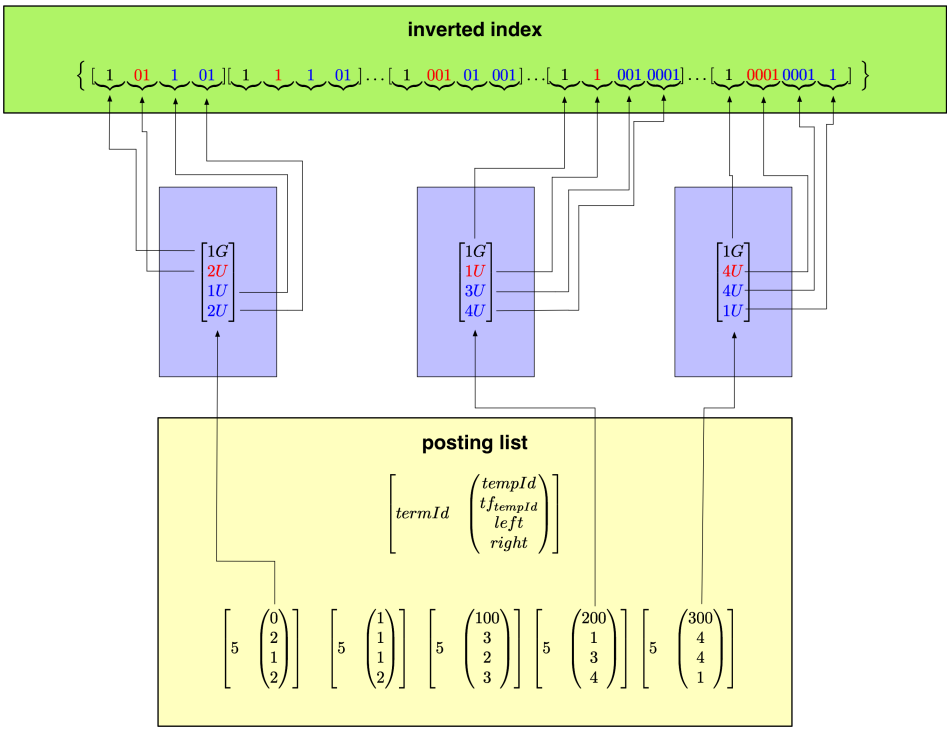

Figure 5.1. *Representation of how the postings data are encrypted and stored in the inverted index, with respect to the BA strategy. The letter "U", stands for Unary coding, while the letter "G" stands for Gamma coding.*

Now we turn our attention to the MA approach. We recall that the purpose of the MA strategy is to speed-up query evaluation by using a map which would enable to efficiently access the data stored in the inverted index stream of bytes. In detail, for an index term it should allow

us to access the first posting of each document in which the examined term occurs in. To implement this approach we have modified how the postings are arranged in the on-disk inverted index, so that the postings related to successive versions of the same document are stored in adjacent positions in the inverded index. In this way, if a query term occurs in a specific document, we can use the map to seek the first posting related to the examined document. Therefore, we could scan the inverted index stream of bytes starting from the position indicated by the map, thus ignoring some postings which certainly do not contain the examined query term. To implement this strategy, before being flushed to disk, the postings of an inverted list are sorted by taking into account two elements: i) the *realId* identifier, ii) the left boundary of the time interval. Carrying out the sorting operation ensures that the postings associated with different versions of the same document occupy adjacent positions in the on-disk inverted index. Although from a computational perspective performing this sorting operation is expensive, we actually need it only if we take into account more than one version of the same document between two consecutive executions of the flushing procedure. Furthermore, a single document version often contains an extremely reduced number of the terms composing the index Lexicon, hence such operation actually involves only a restricted number of posting lists. For what concerns the Merging phase, ensuring that the posting lists deriving from the actualization of the merging operations on the data maintained on-disk, show the aforementioned property, requires us to make a significant effort. We look at the case in which the merging procedure is applied to two inverted lists, related to the same index term. To achieve the goal we can exploit the fact that the examined inverted lists are singularly ordered in the right way with respect to the aforementioned property. Therefore, we can accomplish the objective by comparing the elements which time after time constitute the first entries of the two posting lists. At each stage, we add the posting, which from the comparison turns out to be the lesser, to the resulting stream of bytes. To compare the postings composing the inverted lists, we take into consideration their identifiers and their time intervals. It is worth stating that as for the BA approach, the MA strategy involves updating the time intervals of the postings after newer versions of the collection documents go through the indexing process. We perform the update operation both during the the indexing process and during the retrieval process. This ensures that in computing the scores of the collection documents, we only consider valid postings. In the remainder, the map which is responsible for maintaining the positions of the documents in which an index term occurs in, which constitutes the cornerstone, on which the MA strategy relies on, will be denoted

as *mapTermDocuments*. We create it for the first time after the first activation of the flush policy and then we reinitialize it each time we enter the Merging phase. This ensures that the *mapTermDocuments* keep up with the evolution of the inverted index over time. Now we proceed to illustrate how the data composing a posting are encrypted in the MA approach. The identifiers of the document versions are stored by using the d-gap technique, which was outlined in Chapter 4. Note that differently from the BA approach, in the MA strategy, the difference between the identifiers of two adjacent postings of a posting list is generally quite large. This is due to the fact that adjacent postings contain data related to two consecutive versions of the same document, rather than data of versions which go through the indexing process one after the other, as occurs in the BA strategy. It is useful to recall that the fact that two versions are indexed one after the other entails that their identifiers only differ by a unit. We encrypt the frequency field by using the Unary [18] coding, while for the the boundaries of the time interval the postings are attached with, we use the Gamma [16] encoding. At this stage, it is useful to specify that we do not apply the d-gap technique, which is used for encoding both the identifiers and the boundaries of the time intervals, to all the entries of an inverted list, considered as a whole, but we singularly implement it on the postings related to the single documents, occurring in the posting list, individually analyzed. In other words, after encoding all the data of the postings of a document, we encrypt the fields of the first version of the next document as they are, before again taking into account the d-gap technique for storing the data related to the following versions of the same document.

5.1.2 Retrieval process

In this section we detail how the BA and MA strategies allow us to carry out query evaluation, once we are given with a time-travel query. As in the previous section, we only mention the differences between the developed approaches and Terrier real-time retrieval process, without explaining how the process is performed in its entirety. Firstly, looking at the theoretical model we gave in Section 5.1.1, we point out that in the experimental phase we have only focused on time-travel queries enriched with a time point rather than a time interval. Therefore, in this work, a time-travel query q can be thought of as a bags of words $q = \{q_1, \dot{..}, q_n\}$, provided with a time-point t_q. We consider a posting p, associated with one of the query terms, as valid if its validity time interval contains t_q. Formally, given a posting p

$$\big(d, f, [t_i, t_{i+1}]\big) \tag{5.2}$$

it is valid if $t_i \leq t_q < t_{i+1}$ or $t_i \leq t_q$ and $t_{i+1} = 1$, since 1 is adopted to indicate that the posting is related to the latest version of the corresponding document. We recall that d is the identifier of a document version, f the number of occurrences of the examined term in d, $[t_i, t_{i+1}]$ is a time validity interval. Below, we briefly report how the retrieval process is carried out in Terrier, before proposing the main differences between it and the developed approaches. Given a query, the Terrier retrieval process involves taking out the inverted lists, associated with the query terms, from the inverted index. We pull out each posting list by using the information about its position within the stream of bytes, the inverted index consists of. These information are obtained by matching the corresponding query term against the terms of the index Lexicon and retrieving the data associated with it, which include the position of its posting list in the inverted index as well as the number of postings associated with the examined term. We have denoted this value as document frequency in Chapter 3. Then we scan the retrieved posting lists computing at each stage the contribute of a query term to the score of a document. The score is basically evaluated from three components: i) the statistics of the Lexicon entry related to the query term Lexicon entry, ii) the collection statistics, iii) the number of occurrences of the query term in the document of interest. It is worthwhile to mention that the documents in which the query terms occur in, are sequentially analyzed during the retrieval process. This means that after we have assigned the score to a document, we do not consider it anymore, during the retrieval process. We state that both in the BA and in the MA strategy, we can not give a score to a document version, before entirely analyzing all the posting lists of the query terms. This results from the fact that to compute the contribute of a query term to the score of a document, we must know the document frequency of that term with respect to the time point of the query q of interest. Roughly speaking, we must know the number of documents for which exists a version containing the query term whose time interval includes t_q. We divide the retrieval process in two phases:

- *Retrieving phase*: it deals with scanning the posting lists searching for valid postings. As soon as we encounter a valid posting, we increment the document frequency of the corresponding query term by a unit. Then we insert the posting in a list associated with the examined query term. Specifically, we carry out this phase by cyclically performing a sequence of operations. In each step, as in the Terrier retrieval process, we take into consideration a single query term. Both for the BA and the MA strategy, a step ends when either we have found a valid posting or we have visited all the postings of the inverted list of the analyzed query term.

- *Scoring phase*: it is concerned with evaluating the scores of the document versions, from the values we have computed for each valid posting, which we have encountered in the Retrieving phase. In detail, we scan the list associated with each query term, assigning a value to each posting occurring in such a list. We evaluate the score of each version by summing up the values computed for the postings referring to that version.

Below, we focus on the Retrieving phase, since the Scoring phase simply involves giving a score to each posting retrieved during the first phase and obtaining the final scores of the versions in the collection of interest. We firstly concentrate on the BA strategy, before turning our attention to the MA approach. In the BA strategy, each step of the Retrieving phase involves analyzing all the postings of an inverted list. For each posting, we execute the following operations:

- *Taking out the posting from the disk-based inverted index and loading its data into the main memory*.
- *Decoding the posting fields*.
- *Verifying if the posting is valid*. If it is valid, the stage of the Retrieving phase ends. Otherwise the Retrieving phase goes on repeating the same operations on the next posting of the posting list.

It is useful to highlight that before checking if a posting is valid, we must verify if a newer version of the document, the posting refers to, has been indexed, after we have stored the posting into the disk-based inverted index. Actually, we do not need to perform this check for each posting, but only for those related to the latest versions of the collection documents. In the MA strategy, we speed up the research of valid postings by skipping some of them, which certainly do not contribute to the final score of a document version. To do so, we use a map, which we have previously referred to as *mapTermDocuments*. It essentially connects each index term with the positions in the inverted index of the first postings of the documents in which the term actually occurs in. Additionally, for each term, we keep track of the position of the last posting of its posting list. Here, we proceed to explain how the Retrieving phase is actually performed in the MA approach. In the following, for simplicity sake, we suppose that the posting list of the index term we look at, is entirely maintained on disk. The general case, which entails some postings are maintained in the in-memory inverted index, can be simply inferred from the proposed one. Furthermore, we assume that we are given with a

time-travel query q, enriched with a time point t_q. Looking at an iteration of the Retrieving phase, after selecting the posting list related to the examined query term, it involves searching for the position of the next document containing the term. We seek that position in the stream of byte of the inverted index and we resume searching for valid postings from there. As soon as we encounter a posting, whose time interval boundaries are later than t_q, we jump to the following document containing the term. Therefore, by exploiting the information kept in *mapTermDocuments*, for a given document, we avoid analyzing all the postings whose time interval left boundary is later than t_q. We conclude this section by proposing two final remarks.

- Firstly, it is worth stating that although the outlined strategy significantly reduce the number of not valid postings, which we take into account, it does not nullify it. In fact, during the Retrieving phase, for each document containing the query term of interest, we load in the main memory each posting whose time-interval boundaries are earlier than t_q, before decoding the raw data composing its fields. We perform these operations also for the documents that although they contain the query term, they have not got a valid version for it. This results from the fact that in developing the MA strategy we have chosen to keep track of the documents containing a given index term, rather than the single versions in which it occurs in.

- Secondly, the MA strategy allow us to take advantage more of the benefits delineated above, when the time-point of the query of interest is closer to the time instant associated with the first version of the collection documents. Indeed, in this case, by exploiting the information kept in *mapTermDocuments*, we can skip most of the postings composing the inverted lists for the examined query terms.

5.2 BA and MA strategies: implementation details

In this section we give some details on how we have implemented the BA and MA strategies. We first provide some details on the indexing process, before focusing on the retrieval process. In doing so, we take into account both the developed strategies and we underline both the differences between them and those between the implemented approaches, individually considered, and the Terrier real-time indexing and retrieval processes.

5.2.1 Indexing process

The first challenge we have faced taking into account a temporally versioned collection was to allow different versions of the same document to be recognized as successive versions of the same document. To accomplish this goal, before carrying out the indexing procedure on a document version, we assign to it the name of the document, it belongs to. We specify that for "name", we mean the last part of a document filename. Therefore, we associate each document with an unique alphanumeric identifier, which we use for identifying all its versions. As we mentioned above, Terrier by default index different versions of the same document as if they were different documents. We have addressed this shortcoming by providing each document version with two identifiers:

- *realId*: an integer value which determines the real identifier of the document, the version belongs to.
- *tempId*: an integer value specifying the number of versions we have indexed before the examined one.

Note that *tempId* corresponds to the identifier we assign to each document going through the Terrier real-time indexing process. In the developed approaches, in order to assign the *realId* and *tempId* identifiers to each document version, we use two objects, which we have denoted as *MemoryMetaName* and *MemoryVersionDocId*. These basically encapsulate two data structures, which are a map and a list, respectively. In particular, for each document version we perform the following operations:

- *Assigning the realId identifier*: we use the last part of the filename of the document version as key to verify if we have already indexed a previous version of the corresponding document. If this case occurs, the representation of the document version, obtained by parsing the file it is included into, is provided with the integer identifier associated with the detected entry. Otherwise, we insert in *MemoryMetaName* a couple consisting of the name of the document version and the number of elements contained in such object at this time.
- *Assigning the tempId identifier*: we associate the representation of the document version with the number of documents we have indexed so far, which constitutes the *tempId* identifier. Finally, we add an entry composed of the two identifiers of the document version to the *MemoryVersionDocId*.

Here, we describe how we have addressed the issue of adjusting the time interval of a posting referring to the current version of a document, when we analyze a newer version of the same document. In detail, given a posting which is related to the current version of a document, we must ensure that its time interval is properly updated as soon as a more recent version of the same document goes through the indexing process. This ensures that the retrieval process is properly carried out, regardless the time a query is issued to the information retrieval system implementing one of the described strategies. To address this issue, in the experimental phase, we have defined an object which we have referred to as *MemoryDocVersions*. For each collection document, such object is responsible for maintaining the right boundary of the time interval its versions are enriched with, while the collection evolves over time. It basically consists of a wrapper of a variable-size matrix. The y-axis is the document dimension. Looking at a document, going along the x-axis involves finding the time interval right boundary of the versions which we have indexed for that document. Given a document, we use the values included into the instance of *MemoryDocVersions* to update the validity interval of the postings referring to the version that time after time constitutes the latest version of the examined document. We perform the check operation, which deals with verifying if a time interval of a posting must be adjusted, both during the indexing process and the retrieval procedure. We point out that we actually need to check only the postings whose time interval right boundary equals 1, namely the postings which refer to the latest versions of the corresponding documents. We remind that the usage of the value 1 in the time intervals, the postings are enriched with, is two-fold. If the left boundary of a time interval is equal to 1, it means that the related posting refer to the first version of the corresponding document. Instead, if the right boundary of a validity interval equals 1, it means the corresponding posting is associated with the latest version of the document, that posting refers to. Below, with the purpose of explaining how we perform the check operation, we suppose we are provided with a posting, whose time interval right boundary is equal to 1. For such a posting, verifying if we have indexed a newer version of the corresponding document corresponds to searching for the first value greater than its time interval left boundary in *MemoryDocVersions*, in regard to the document of interest. If we find such a value, we update the right boundary of the posting by setting it to the detected value. It should be noted that, looking for the latest version of a document could involve scanning all the values stored in the *MemoryDocVersions* for that document. To reduce the computational cost of such operation, after carrying out the Merging phase, we reinitialize

the *MemoryDocVersions* used during the indexing process. In fact, we point out that after the execution of the Merging phase, all the postings of the inverted index are properly updated, regardless the fact that they are maintained on-disk or in the in-memory inverted index.

Another problem we have faced during the experimental phase was to manage the update of the collection statistics. Since, we have given a background on this matter in Section 5.1.1, below we only provide some details of the implemented solution, without giving any further explanation. With the aim of keeping the collection statistics, as the documents in the collection evolve over time, we have defined an object, referred to as *collectionCS*. It is basically a list of *CollectionStatistics*. We recall that a *CollectionStatistics* is a wrapper of four statistics, which we have been presented in Chapter 3. Below, we briefly repeat them.

- *numberOfDocuments*: the number of documents indexed so far.
- *numberOfTokens*: the count of the terms composing the documents occurring in the collection.
- *numberOfTerms*: the number of *postings* forming the *inverted index*.
- *numberOfUniqueTerms*: the number of distinct elements comprising the *Lexicon*.

Note that after indexing a document version, we expect a new *CollectionStatistics* is initialized, reflecting the change in the collection statistics due to the execution of the indexing process on the examined version. In the experimental phase, we have assigned a *CollectionStatistics* to each document, responsible for maintaining the values of the proposed variables for the version that time after time is the latest version of the document. We include these objects into *collectionCS*, at the positions corresponding to the *realId* identifiers of the associated documents. In addition to them, we generate an other *CollectionStatistics*, which in the following will be denoted as *summaryCS*. *summaryCS* accounts for the actual values of the collection statistics as the index evolves over time. Therefore, while each *CollectionStatistics* mantained in *collectionCS*, keeps up with the evolution of the of a single collection document, *summaryCS* specifies the values of the aforementioned statistics for different time-points, taking into account all the documents comprising the corpus of interest. In fact, we evaluate the *summaryCS* fields from the corresponding ones of the *collectionCS* elements. Below, we explain how we keep the collection statistics updated, after a document version goes through the indexing process. Firstly, we must check if a *CollectionStatistics* exists in *collectionCS* at the index corresponding to the *realId* identifier of the document. If

this case does not occur, we generate a new *CollectionStatistics*, whose fields are evaluated on the basis of the content of the document version. Then, we add the newly generated object to *collectionCS*. Finally, we update the *summaryCS* fields by summing up the corresponding values of the *collectionCS* entries. Instead, if the condition presented above arises, two cases can occur depending on the content of the document version. If it is blank, after setting to 0 all the fields of the *CollectionStatistics* associated with the document, we adjust the *summaryCS* statistics. In detail, we decrement its variables by the difference between the current and the previous version of the document. Additionally, we further decrement the *numberOfDocuments* variable by 1. This accounts for the fact that inserting a blank version of a document means eliminating that document from the collection. Instead, if the document version is not blank, with the exception of the update of the *numberOfDocuments* field, we perform the same operations delineated above. For what concerns the *numberOfDocuments* variable, we increment it by 1 if the previous version of the document was blank, otherwise we do not change its value.

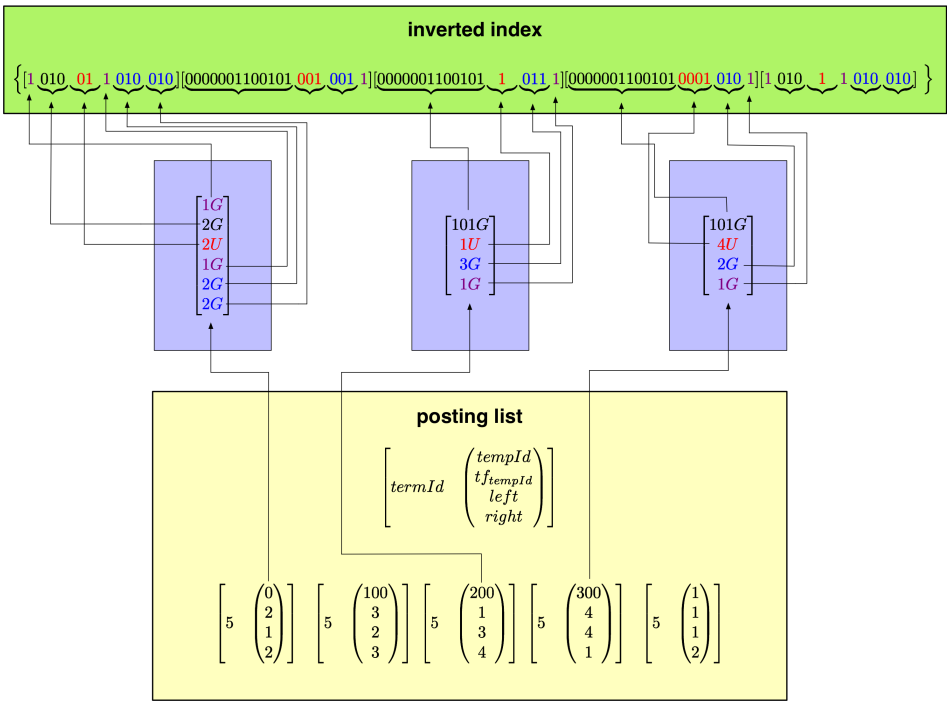

Figure 5.2. *Representation of how the postings data are encrypted and stored in the inverted index, with respect to the MA strategy. The letter "U", stands for Unary coding, while the letter "G" stands for Gamma coding.*

After giving some implementation details on elements which are common to both the developed strategies, we conclude this section by providing some details of the MA strategy.

Specifically, we focus on the way in which postings data are stored during the indexing process, after briefly repeating some key points underlying the considered strategy. We recall that in order to implement the MA strategy we must arrange the postings of a posting list so that the ones related to successive versions of the same document occupy adjacent positions in the inverted list. In Section 5.1.1 we have also noted that in order to reduce the size of the inverted index as much as possible, we encrypt the identifiers and the time interval data of the postings by using the d-gap technique. We remark that the usage of this approach is restricted to all the postings related to the versions of the same document. In other words, the identifier and the temporal interval left boundary of the first posting related to a document are stored as they are. At this stage, we turn our attention to the manner in which we have actually carried out this strategy, thus giving some particulars on how we must interpret the data related to a posting, after pulling it out and loading its raw data in the central memory. Precisely, with the purpose of putting this approach into effects, we have adopted some flag variables. In detail, each time we take into consideration a new document, we precede the identifier of the examined version with the value 1. It is worth mentioning that this value actually constitutes the lowest value that we can encrypt by using the Gamma [16] coding, the technique we have basically adopted for storing the identifiers and the temporal interval boundaries of the postings in the stream of byte composing the inverted index. This implies that, for two postings related to the same document, we can not directly include the difference between their identifiers into the stream of bytes, in that it could be equal to 1. In such a situation, the following value in the stream would be interpreted as the identifier of a new document. To tackle this issue, before storing the difference between the identifiers associated with successive document versions, we increase it by 1. Therefore, by using the described approach, 2 becomes the lowest value we can encrypt in the disk-based inverted index, for representing the difference between identifiers of successive versions. It is useful to note that although by using the Gamma coding, we need 3 bits to represent the value 2, compared to the 1 required to store the value 1, implementing the described approach does not result in significantly increasing the inverted index size. This derives from the fact that by employing the Gamma coding for representing a value x, we need $2\lfloor \log_2 x \rfloor + 1$ bits. Therefore, incrementing the difference by 1, results in increasing the number of bits required for storing the computed value only if the difference is equal to $2^m - 1$, with $m \in \mathbb{N}_{\not\vdash}$. Note that going along with the value of m, the probability that such case occurs rapidly goes down. With respect to this issue, we remark that in the MA approach, the difference between the

identifiers of two adjacent postings of a posting list is generally quite large, because they are generally related to versions which do not go through the indexing process one after the other. A procedure similar to the one described above is used for storing the time interval boundaries in the inverted index stream of bytes. For a generic version of a document, with the exception of the first one, we store into the stream of bytes the difference between the two extremes, incremented by 2. Note that we need to pay particular attention to handling the postings which refer to the latest version of a document, because their time interval right boundary is equal to 1, hence the difference between the extremes of their time interval is negative. We have addressed this issue by using an other flag variable. In detail, for the postings related to the current versions of the documents in the collection, instead of encrypting the difference between the two boundaries of their time interval, we include the value 2 into the stream. We ends the storage of the data related of a posting by adding the value 1 to the inverted index, again encrypted by using the Gamma coding. It is worth stating that we use this value to handle correctly also the case in which both the boundaries of the time interval are set to 1. The ideas we have presented above are illustrated in Figure 5.2. Specifically, the Figure 5.2 depicts the way in which posting fields are encrypted and their data are included into the inverted index, in regard to the MA strategy. Looking at the figure 5.2, to better explain how the different data composing a posting are encoded, we use different colours for different types of information. In detail, we use the black for the data related to the document identifiers, the red for the ones associated with the frequency field, whereas the blue is used for the data related to the time intervals the postings are associated with. Finally, the violet bits do not convey any information. They are used as flag variables for correctly interpreting the useful data included into the inverted index.

5.2.2 Retrieval process

In this section we provide some implementation details on how we have carried out the retrieval process in the MA approach. In particular, we focus on the change we have brought about to the Matching phase of the Terrier retrieval process, to allow us to respond a time-travel query. Firstly, we have extended the *PostingListManager* class. We remind that an instance of that class is used in the Matching phase of the Terrier retrieval process for maintaining all the data related to the query terms. We have enhanced this class in such a way that the object used during the retrieval process keep also the *CollectionStatistics* related to the time instant attached with the given query. Below, we assume that a query is submitted

to the information retrieval system, implementing the strategy taken into account, right after the Merging phase has been carried out. In such a situation all the *CollectionStatistics* which we have generated during the indexing process are stored in a disk-based list, which will be referred to as *CSList*. During the retrieval process, we access the entries of the*CSList*, by means of a map, entirely maintained in memory, which associates each time-point considered during the indexing process with an index of such list. Secondly, the MA retrieval process involves instantiating two objects, which we have denoted as *postingScore* and *documentFrequency*. The former basically is responsible for maintaining the valid postings which we find by scanning the posting lists related to the query terms. It is useful to specify that each *postingScore* entry is actually made up of two elements: i) a valid *posting* we have encountered by scanning the inverted lists of the query terms, ii) the *realId* identifier associated with the version the corresponding posting refers to. The *documentFrequency* is an array which keeps the document frequency of each query term. In the following, as we have done in Section 5.1.2, for description sake, we suppose to divide the retrieval process in two different phases, which we have referred to as Retrieving phase and Scoring phase. The former deals with analyzing the posting lists associated with the query terms searching for valid postings. The latter is concerned with the actual computation of the document scores. In each iteration of the Retrieving phase we carry out the following operations:

- *Selecting the posting list whose first posting has the lowest identifier*. In order to decide which posting list we must take into consideration, we compare the *realId* identifiers associated with the first posting of the posting lists of interest.

- *Searching for the next valid posting in the inverted list, selected at the previous pointè. In this section, we do not give any detail about how this stage actually takes place, since we have already delved into this matter in Section 5.1.2*.

- *Inserting the detected posting and the corresponding realId identifier in postingScore*.

- *Increasing by one the document frequency of the query term whose inverted list has been taken into consideration at the previous point*.

In the Scoring phase, we attribute a score to each posting composing a *PostingScore* entry. In particular, for each list the *postingScore* is made up of, we compute the scores of the corresponding postings by taking into account i) the collection statistics related to the time-point of the given query, ii) the document frequency of the query term associated with the

examined list, iii) the data composing the analyzed postings. During the *Scoring phase*, we use the *realId* identifiers to associate the scores computed for each valid posting occurring in *PostingScore* with the corresponding documents.

6 Experimental results

This chapter presents the results obtained in the experimental phase, where we tested the presented approaches on two collections: the CLEF Adhoc monolingual test collection for the Italian language and one we have generated from the English Wikipedia. After describing the collections we have used in the experimental phase, we accurately present the different experiments we have carried out on such collections, before presenting the obtained results.

6.1 Test collection

In the experimental phase we have actually tested the BA and MA strategies on two collections. The first one is the CLEF Adhoc monolingual test collection for the Italian language, while the second one is composed of web pages of the English Wikipedia we have downloaded from the Wikimedia Foundation project. Below, we give some details on these collections before going on to describe the experiments we have conducted. In the following, the CLEF Adhoc monolingual test collection for the Italian language will be referred to as AH-MONO-IT. The documents forming the corpus of such collection are of two different types. The former group, LASTAMPA94, contains articles from the LASTAMPA newspaper, published in 1994. The latter group AGZ-94, AGZ-95 includes news from the Swiss news agency, related to the years 1994 - 1995, translated in Italian. The overall number of documents in the corpus is 157558. The topics of the AH-MONO-IT test collection follow the TREC structure, including three fieds which are title, description and narrative, respectively. The title field presents the argument the query is about, while description and narrative extend such field, by providing contextual information. In the experimental phase, we have only taken into account the title and description fields, thus ignoring the others. The

AH-MONO-IT test collection includes 51 topics. Below, in Table 6.1, we summarize the specifications of the examined test collection.

AH-MONO-IT	
Number of documents	157558
Numer of topics	51

Table 6.1. *Features of the CLEF Adhoc monolingual test collection for the Italian language.*

Here, we take into account the second collection we have actually used in the experimental phase. In the remainder, it will be referred to as COOKING-MONO-EN, since it is made up of English web pages about cooking. Some of the topics covered by these pages are chefs, typical dishes, cooking techniques, cooking utensils, cookbooks, cooking competitions. Hereafter, we describe how we have actually generated the COOKING-MONO-EN corpus of documents, starting from the pages we have downloaded from the English Wikipedia. These pages, offered by the Wikimedia Foundation project, are multi-licensed under the Creative Commons Attribution-ShareAlike 3.0 License (CC-BY-SA) and the GNU Free Documentation License (GFDL). For each of the downloaded pages, we have stored part of its hystory into an xml file, up to a maximum of 1000 versions. The versions are enriched with a date indicating when they have been made available on the web. In order to ensure that the collection of documents we wanted to create had a large number of versions per page, we have selected the 1000 files with the highest size, thus ignoring the others. Starting from the selected files, we have generated the temporally versioned collection used in the experimental phase, by carrying out the following operations:

- *Extracting the first* 300 *versions of each page.*
- *Assigning each version with an integer identifier indicating the position of that version with respect to all the versions of that page, we have actually taken into account.*
- *Inserting the versions associated with the same numerical identifier in the same xml file.*

By performing the procedure given above, we have generated a temporally versioned collection containing up to a maximum of 300 versions for the 1000 web pages we have selected. Note that in carrying out the operations given above, we have ignored the temporal information related to the versions comprising the web pages. The topics we have generated

```
<topics>
<topic>
<identifier>1</identifier>
<title>Vegan Cooking</title>
<description>
Retrieve information about vegan cooking. Find out if vegan food is more expensive than that of gluten-free cooking or oil-free cooking.
Additionally, find some vegan recipes for preparing delicious dishes.
</description>
</topic>
<topic>
<identifier>2</identifier>
<title>Alessandro Stratta</title>
<description>
Retrieve information on Alessandro Stratta, the celebrity chef who won the James Beard Foundation award for Best Chef Southwest in 1998.
He was the executive chef and owner of "Tapas by Alex Stratta" in Las Vegas, until its closure in 2015.
</description>
</topic>
<topic>
<identifier>3</identifier>
<title>White Chocolate</title>
<description>
Retrieve information about white chocolate, a derivative of the chocolate.
The main ingredients being used for preparing it are cocoa butter, sugar, milk solids.
</description>
</topic>
<topic>
<identifier>4</identifier>
<title>Cooking techniques</title>
<description>
Retrieve information about cooking techniques. For cooking techniques we generally mean a set of procedures for preparing, cooking and presenting food.
Some of them are: boiling, barbecuing, deglazing, frying.
</description>
</topic>
<topic>
<identifier>5</identifier>
<title>Typical dishes</title>
<description>
Retrieve information about the typical dishes of the Turkish cuisine.
Many experts believe the main elements of the Turkish cuisine derive from Circassian, Caucasian, Balkan, Greek cuisines.
</description>
</topic>
</topics>
```

Figure 6.1. *Topics of the COOKING-MONO-EN collection.*

for the COOKING-MONO-EN collection follow the TREC structure, as the AH-MONO-IT test collection. Specifically each query is made up of two fields, which are title and description, respectively. The title defines the topic of the query, while the description extend this field by giving additional information. The COOKING-MONO-EN collection includes 5 topics, which are presented in Figure 6.1. The Table 6.2 displays some summary data about the COOKING-MONO-EN, we have discussed above.

COOKING-MONO-EN	
Number of web pages	1000.00
Number of topics	5.00
Maximum number of versions per page	300.00
Overall number of versions	179386.00
Average number of versions per page	179.39

Table 6.2. *Features of the COOKING-MONO-EN collection*

6.2 Evaluation framework

In the experimental phase we have compared the BA and MA strategies to the Terrier real-time indexing and retrieval processes. The experiments we have carried out on the AH-MONO-IT and the COOKING-MONO-EN text collections fall into two different categories: i) correctness tests, ii) performance evaluation. The former group of tests aims at ensuring that the developed strategies allow us to obtain the same results we would have obtained by using the Terrier real-time indexing and retrieval processes with no time queries. The latter group of tests is concerned with computing the time we needed to evaluate a time-travel query as well as the amount of memory that each of the developed strategies requires to temporally store the data collected during the indexing and retrieval processes.
In Sections 6.2.1 and 6.2.2 we report the results we have obtained by testing the BA and MA strategies on the collections we presented in Section 6.1. Although we have performed the same tests on both the examined collections, for description sake, we report the results of the correctness test only for the AH-MONO-IT test collection.

6.2.1 Experimental results: AH-MONO-IT

Before experimenting with the BA and MA strategies, we have divided the corpus documents in four groups, each of them containing about the same number of documents. In detail, we have included the files corresponding to the collection documents into a single folder, before arranging them according to the lexicographical order. In Table 6.3, with the aim of making our tests reproducible, we specify the set of documents composing each of the groups.

	Corpus documents
Group 1	AGZ.940101.0001 - AGZ.940925.009
Group 2	AGZ.940925.0010 - AGZ.950715.0075
Group 3	AGZ.950715.0076 - LASTAMPA94-017451
Group 4	LASTAMPA94-017452 - LASTAMPA94-058051

Table 6.3. *Groups of documents in which we have actually divided the AH-MONO-IT test collection.*

During the experimental phase, we have handled the four groups of documents indipendently from one another, as if they constituted different set of documents that we have taken into account for the indexing process at different time-points. Below, we present the different tests we have conducted on the AH-MONO-IT test collection.

```
<topics>
<topic>
<identifier>141</identifier>
<title>Lettera Bomba per Kiesbauer</title>
<description>
Recupera le informazioni relative all'esplosione di una lettera bomba nello studio della presentatrice della rete televisiva PRO7.
</description>
<narrative>
Una lettera bomba di un gruppo di estrema destra spedita al personaggio televisivo di colore Arabella Kiesbauer è esplosa in uno
studio del canale TV PRO7 il 9 Giugno 1995. Un'assistente è rimasta ferita. Sono rilevanti tutte le informazioni sull'esplosione
e le indagini della polizia dopo l'accaduto. I documenti che si riferiscono ad altri attacchi con lettere bomba non sono pertinenti.
</narrative>
</topic>
<topic>
<identifier>142</identifier>
<title>Christo impacchetta il Parlamento Tedesco</title>
<description>
Trova i documenti che parlano dell'impacchettamento del Parlamento tedesco a Berlino ad opera dell'artista Christo.
</description>
<narrative>
L'artista dell'impacchettatura Christo ha impiegato due settimane nel giugno del 1995 per avvolgere il Reichstag tedesco a Berlino
in chilometri di stoffa. Trova i documenti che parlano di questo evento artistico. È rilevante qualsiasi informazione sia sulla
preparazione sia sull'esecuzione dell'opera, inclusi i dibattiti politici, le decisioni prese in merito e la preparazione tecnica.
</narrative>
</topic>
<topic>
<identifier>143</identifier>
<title>La Conferenza di Pechino sulle Donne</title>
<description>
Le posizioni controverse di alcuni delegati hanno significato che la Conferenza di Pechino sulle donne ha rischiato di fallire.
</description>
<narrative>
I documenti rilevanti discuteranno qualsiasi problema o disaccordo sorto alla conferenza sulle donne di Pechino.
Di interesse particolare sono le posizioni sostenute dai rappresentanti del Vaticano, delle comunità musulmane
e del partito comunista cinese.
</narrative>
</topic>
<topic>
<identifier>145</identifier>
<title>Importazioni Giapponesi di Riso</title>
<description>
Recupera i documenti che discutono i motivi e le conseguenze della prima importazione di riso in Giappone.
</description>
```

Figure 6.2. *Topics of the AH-MONO-IT test collection.*

- Correctness tests related to the insertion of the documents of the aforementioned groups in the collection we have used for carrying out the indexing process, at successive time instants.

- Correctness tests related to the removal from the collection of the documents of the groups described above, at successive time points.

Note that performing the tests above has involved associating each group with a time point specifying the time instant in which we have applied the indexing process to each document of that group. It is worthwhile to note that we have supposed that documents belonging to the same group went through the indexing process at the same time.

For the first test set, we have provided the groups of documents with successive integer values, starting from 1. In giving such values, we have sequentially taken into account all the groups in ascending order with respect to their identifiers. Hence, we have assigned the time-point 1 to the first group, the time-point 2 to the second group and so on. Then we have applied

Rank	Terrier	BA	MA	Terrier	BA	MA
1	8242	8242	8242	23.9064	23.9064	23.9064
2	18958	18958	18958	15.6858	15.6858	15.6858
3	689	689	689	12.0537	12.0537	12.0537
4	28831	28831	28831	11.9604	11.9604	11.9604
5	1623	1623	1623	11.7723	11.7723	11.7723
6	4740	4740	4740	11.7216	11.7216	11.7216
7	24209	24209	24209	11.5932	11.5932	11.5932
8	26068	26089	26068	11.4307	11.4307	11.4307
9	8953	8953	8953	11.3431	11.3431	11.3431
10	18969	18969	18969	11.2440	11.2440	11.2440
11	34670	34670	34670	11.2170	11.2170	11.2170
12	16239	16239	16239	11.1094	11.1094	11.1094
13	28417	28417	28417	10.7622	10.7622	10.7622
14	33096	33096	33096	10.5554	10.5554	10.5554
15	32989	32989	32989	10.3686	10.3686	10.3686
16	34332	34332	34332	10.3077	10.3077	10.3077

Table 6.4. *Results obtained for the time-travel query* 142*, enriched with the time-point* 1*.*

the BA and MA strategies to all the groups, taken into account in the order given by the time-points their documents were enriched with.

Rank	Terrier	BA	MA	Terrier	BA	MA
1	75038	75038	75038	30.9096	30.9096	30.9096
2	76666	76666	76666	27.6471	27.6471	27.6471
3	74664	74664	74664	27.1126	27.1126	27.1126
4	76536	76536	76536	24.4940	24.4940	24.4940
5	54290	54290	54290	24.0311	24.0311	24.0311
6	46514	46514	46514	23.5022	23.5022	23.5022
7	8242	8242	8242	23.3289	23.3289	23.3289
8	75519	75519	75519	23.3205	23.3205	23.3205
9	73779	73779	73779	22.8103	22.8103	22.8103
10	73852	73852	73852	22.4341	22.4341	22.4341
11	74055	74055	74055	22.2322	22.2322	22.2322
12	73924	73924	73924	20.2291	20.2291	20.2291
13	74082	74082	74082	19.0127	19.0127	19.0127
14	76784	76784	76784	18.9759	18.9759	18.9759
15	18958	18958	18958	15.4837	15.4837	15.4837
16	62064	62064	62064	12.0416	12.0416	12.0416

Table 6.5. *Results obtained for the time-travel query* 142*, enriched with the time-point* 2*.*

To check the correctness of the BA and MA strategies, we have evaluated all the queries of the AH-MONO-IT text collection, after performing the indexing process on each group of documents. It is useful to point out that at each iteration of the retrieval process, we have provided the time-travel queries with the time-point associated with the latest group we have taken into account before carrying out the evaluation of such queries. This allowed us to obtain a ranked list per topic for each of the four time-points we have actually enriched the corpus documents with. Finally, we have compared the obtained results with the ones we have

Rank	Terrier	BA	MA	Terrier	BA	MA
1	75038	75038	75038	32.5173	32.5173	32.5173
2	76666	76666	76666	29.8480	29.8480	29.8480
3	74664	74664	74664	28.7344	28.7344	28.7344
4	75536	75536	75536	26.1637	26.1637	26.1637
5	54290	54290	54290	25.1273	25.1273	25.1273
6	46514	46514	46514	24.7787	24.7787	24.7787
7	8242	8242	8242	24.5534	24.5534	24.5534
8	75519	75519	75519	24.4014	24.4014	24.4014
9	73779	73779	73779	24.3351	24.3351	24.3351
10	74055	74055	74055	24.0261	24.0261	24.0261
11	73852	73852	73852	23.7364	23.7364	23.7364
12	73924	73924	73924	21.7132	21.7132	21.7132
13	76784	76784	76784	20.2709	20.2709	20.2709
14	74082	74082	74082	20.0410	20.0410	20.0410
15	18958	18958	18958	16.7846	16.7846	16.7846
16	79017	79017	79017	16.4970	16.4970	16.4970

Table 6.6. *Results obtained for the time-travel query* 142*, enriched with the time-point* 3*.*

gotten by applying the Terrier real-time indexing and the retrieval processes to the collection documents, we have considered in testing our approaches.

In Figure 6.2, we propose an overview of some of the topics composing the AH-MONO-IT test collection. We remind that in the experimental phase, we have only considered the title and description fields . Note that in Figure 6.2 the queries are enriched with an identifier. In the remainder, we employ such identifiers to refer to the queries such identifiers are associated with.

Rank	Terrier	BA	MA	Terrier	BA	MA
1	75038	75038	75038	33.9806	33.9806	33.9806
2	76666	76666	76666	31.8366	31.8366	31.8366
3	74664	74664	74664	30.2113	30.2113	30.2113
4	76536	76536	76536	27.7675	27.7675	27.7675
5	54290	54290	54290	26.0280	26.0280	26.0280
6	46514	46514	46514	25.9204	25.9204	25.9204
7	74055	74055	74055	25.8370	25.8370	25.8370
8	73779	73779	73779	25.7723	25.7723	25.7723
9	8242	8242	8242	25.6406	25.6406	25.6406
10	75519	75519	75519	25.2870	25.2870	25.2870
11	73852	73852	73852	24.8936	24.8936	24.8936
12	152094	152094	152094	24.8323	24.8323	24.8323
13	73924	73924	73924	23.1376	23.1376	23.1376
14	76784	76784	76784	21.4062	21.4062	21.4062
15	74082	74082	74082	20.9427	20.9427	20.9427
16	18958	18958	18958	17.8518	17.8518	17.8518

Table 6.7. *Results obtained for the time-travel query* 142*, enriched with the time-point* 4*.*

In Tables 6.4, 6.5, 6.6, 6.7, we propose excerpts of the ranking lists we have obtained by responding the time-travel query 142, time after time enriched with one of the time-instants we have provided with the documents in the collection. For each of the documents occurring

in the proposed ranking lists, we also report the corresponding score. We remind that each ranking list at most contained 1000 identifiers. This means that given a time-travel query, we consider as relevant only the first 1000 documents with the highest score. Looking at the Tables 6.4, 6.5, 6.6, 6.7, we can observe how the scores of the documents changed as soon as we have applied the developed strategies to a new group of documents. This is due both to the added information and the fact that at each stage of the retrieval process, we compute the scores of the documents from scratch, by taking into consideration the statistics related to the indexed documents at the time corresponding to the time predicate of the query of interest.

Rank	**Terrier**	**BA**	**MA**	**Terrier**	**BA**	**MA**
40	34597	34597	34597	6.6722	6.6722	6.6722
41	24838	24838	24838	6.6451	6.6451	6.6451
42	10445	10445	10445	6.6210	6.6210	6.6210
43	23999	23999	23999	6.5481	6.5481	6.5481
44	31442	31442	31442	6.5445	6.5445	6.5445
45	24414	24414	24414	6.5291	6.5291	6.5291
46	34661	34661	34661	6.5064	6.5064	6.5064
47	17152	17152	17152	6.5037	6.5037	6.5037
48	13605	13605	13605	6.4947	6.4947	6.4947
49	34313	34313	34313	6.4942	6.4942	6.4942
50	36982	36982	36982	6.4757	6.4757	6.4757
51	**13167**	**13166**	**13166**	**6.4716**	**6.4716**	**6.4716**
52	**13166**	**13167**	**13167**	**6.4716**	**6.4716**	**6.4716**
53	3472	3472	3472	6.4643	6.4643	6.4643
54	9772	9772	9772	6.4480	6.4480	6.4480
55	20270	20270	20270	6.4469	6.4469	6.4469

Table 6.8. *Results obtained for the time-travel query* 143*, enriched with the time-point* 1*. We can observe that the documents with identifiers* 13166 *and* 131667*, despite the fact that they have the same scores, they are placed in reverse order in the ranked lists associated with the developed strategies, compared to the one obtained by applying the Terrier retrieval process.*

Note that in Table 6.8 althought the documents, whose indentifiers are 13166 and 13167, have the same score, they appear in reverse order in the ranking lists associated with the developed approaches, compared to the one obtained for the Terrier retrieval process. This results from the fact that in the Terrier retrieval process, the algorithm we have actually adopted for sorting the documents identifiers, with respect to their scores, is not stable.

In order to make sure the ranked lists obtained by using our strategies were the same as those acquired by using the Terrier real-time indexing and retrieval processes, we have indirectly

compared them by means of the Kendall rank correlation coefficient [31], a statistic measure proposed by Kendall, in 1938. The Kendall rank coefficient is also denoted as Kendall's Tau. It is a non parametric measure reflecting the relationships between columns of ranked data. It assumes values ranging from -1 and 1. 1 indicates that the two column are perfectly the same, 0 that the rankings are indipendent, while -1 denotes a negative correlation, namely one ranking is the inverse of the other. Given two sequences of data, we associate each element of the second sequence with the position of that element in the first sequence, which we suppose to take as reference. At this stage we take into account the positions of the elements in the two sequences. In particular, in the following, we denote the first list of positions as X, while we refer to the second list as Y. Let $\big(x_i, y_i\big)$ a generic couple, where x_i is the i-th element of X, whereas y_i is the i-th element of Y. Looking at the couples $\big(x_i, y_i\big)$ and $\big(x_j, y_j\big)$, with $i \neq j$, we define them *concordant* if either $x_i > x_j$ and $y_i > y_j$ or $x_i \leq x_j$ and $y_i \leq y_j$. On the contrary, we denote them as *discordant* if either $x_i > x_j$ and $y_i < y_j$ or $x_i < x_j$ and $y_i > y_j$. We compute the Kendall rank coefficient through the formula given below:

$$\tau = \frac{nc - nd}{n\big(n-1\big)/2} \tag{6.1}$$

where n is the actual lenght of X, nc is the number of concordant pairs, while nd is the number of discordant pairs.

In the experimental phase, we have matched each of the ranking lists, resulting from the application of the BA and MA strategies, against the corresponding ranking list, acquired by performing the Terrier real-time indexing and retrieval processes. The results we have obtained for the time-point 1 and all the queries of the test collection are proposed in Table 6.9. In that table we can observe that all the values we have computed for the Kendall's Tau coefficient are very close to 1, indicating that the ranking lists we have obtained for the developed strategies are very similar to the one acquired by employing the Terrier indexing and retrieval processes. Although the results depicted in Table 6.9 are very encouraging, we can note that there are many values different from 1. In fact, these values are justified by the occurrence of the situation we have outlined above. It entails that two documents can be placed in reverse order in the ranking lists obtained for the developed approaches in comparison with the one related to the application of the Terrier real-time indexing and retrieval processes even if they have the same score. To make sure that the values of the Kendall's Tau coefficient different from 1 were a consequence of the presented circumstance,

Query	BA	MA	Query	BA	MA
141	1.0000	1.0000	174	0.9999	0.9999
142	1.0000	1.0000	176	0.9999959	0.9999
143	0.9999	0.9999	177	0.9999	0.9999
145	1.0000	1.0000	178	0.9999	0.9999
147	1.0000	1.0000	179	1.0000	1.0000
148	1.0000	1.0000	180	1.0000	1.0000
149	1.0000	1.0000	181	0.9999	0.9999
150	1.0000	1.0000	182	1.0000	1.0000
151	1.0000	1.0000	183	1.0000	1.0000
152	0.9999	0.9999	184	0.9999	0.9999
153	1.0000	1.0000	185	0.9999	0.9999
154	1.0000	1.0000	186	0.9999	0.9999
155	0.9999	0.9999	187	1.0000	1.0000
156	0.9999	0.9999	188	1.0000	1.0000
157	0.9999	0.9999	189	0.9999	0.9999
159	0.9999	0.9999	190	1.0000	1.0000
161	0.9999	0.9999	192	0.9999	0.9999
162	1.0000	1.0000	193	0.9999	0.9999
163	1.0000	1.0000	194	1.0000	1.0000
164	1.0000	1.0000	195	0.9999	0.9999
165	0.9999	0.9999	196	1.0000	1.0000
166	0.9999	0.9999	197	0.9999	0.9999
167	0.9999	0.9999	198	1.0000	1.0000
168	0.9999	0.9999	199	0.9999	0.9999
171	0.9999	0.9999	200	0.9999	0.9999
173	0.9999	0.9999			

Table 6.9. *Kendall's Tau coefficient values computed for the ranked lists obtained by applying the BA and MA strategies to the first group of documents in which we have divided the AH-MONO-IT corpus of documents.*

we have compared the scores of the documents for which we have found a mismatch in the positions of the corresponding ranking lists. This has confirmed the hypothesis presented above.

Now, we focus on the second group of tests we have conducted. As we mentioned above, they are designed to check that the developed strategies correctly work, even when we modify the documents in the corpus, by including newer versions of them into the collection used for carrying out the indexing process. Specifically, with the aim of comparing the results with those we have obtained for the Terrier indexing and retrieval processes, we have tested the

BA and MA strategies on a set of documents obtained from the AH-MONO-IT corpus of documents by removing time after time one of the groups of documents, given above. Note that, to let the results be matched against the ones we have gotten for the Terrier processes, we have removed the groups of documents from the collection in reverse order to the one we have adopted for the first set of tests. We remind that removing a document from a collection, in our context, means including into it a blank version of the document we want to eliminate. Therefore, to carry out the second set of tests, we have performed the following operations:

- *Applying the BA and MA strategies in the same way as we have done for the first set of tests*. This required us to provide each of the four groups presented above with an integer value starting from 1. Then we have performed the MA and BA strategies on the different groups of documents, in the order given by the time-points, we have enriched them with.

- *Creating a set of blank documents for each of the groups of documents taken into account during the indexing process*. For each group of documents that went through the indexing process, we have generated a new one containing empty versions of the documents occurring in it. To ensure that each version were recognized as a newer version of the corresponding document, we have assigned it with the same name of the related document.

- *Performing the BA and MA strategies on the new groups of documents*. Before carrying out one of the developed strategies on each of the newly-created group of documents, we have enriched them with time-points, starting from 5. We state that we have taken into account the newly generated groups of documents in reverse order to the one we have considered at the first point. After performing the indexing process on a group of documents, we have evaluated all the queries of the AH-MONO-IT test collection with regard to the time point associated with the examined group.

In Tables 6.10, 6.11, 6.12, we provide the results we have obtained for the time-travel query 142, enriched with the time-points 5, 6, 7, 8, respectively. Note that we do not show the results we have gotten by responding the examined query in regard to the time-point 8, since at that time the collection used for performing the indexing process did not contain any valid version of the documents composing the corpus of the AH-MONO-IT test collection.

Below, before reporting some information related to the time we needed to carry out the indexing process on the entire corpus of documents, as well as to the memory usage during

Rank	Terrier	BA	MA	Terrier	BA	MA
1	75038	75038	75038	32.5173	32.5173	32.5173
2	76666	76666	76666	29.8480	29.8480	29.8480
3	74664	74664	74664	28.7344	28.7344	28.7344
4	75536	75536	75536	26.1637	26.1637	26.1637
5	54290	54290	54290	25.1273	25.1273	25.1273
6	46514	46514	46514	24.7787	24.7787	24.7787
7	8242	8242	8242	24.5534	24.5534	24.5534
8	75519	75519	75519	24.4014	24.4014	24.4014
9	73779	73779	73779	24.3351	24.3351	24.3351
10	74055	74055	74055	24.0261	24.0261	24.0261
11	73852	73852	73852	23.7364	23.7364	23.7364
12	73924	73924	73924	21.7132	21.7132	21.7132
13	76784	76784	76784	20.2709	20.2709	20.2709
14	74082	74082	74082	20.0410	20.0410	20.0410
15	18958	18958	18958	16.7846	16.7846	16.7846
16	79017	79017	79017	16.4970	16.4970	16.4970

Table 6.10. *Results obtained for the time-travel query* 142*, enriched with the time-point* 5*.*

Rank	Terrier	BA	MA	Terrier	BA	MA
1	75038	75038	75038	30.9096	30.9096	30.9096
2	76666	76666	76666	27.6471	27.6471	27.6471
3	74664	74664	74664	27.1126	27.1126	27.1126
4	76536	76536	76536	24.4940	24.4940	24.4940
5	54290	54290	54290	24.0311	24.0311	24.0311
6	46514	46514	46514	23.5022	23.5022	23.5022
7	8242	8242	8242	23.3289	23.3289	23.3289
8	75519	75519	75519	23.3205	23.3205	23.3205
9	73779	73779	73779	22.8103	22.8103	22.8103
10	73852	73852	73852	22.4341	22.4341	22.4341
11	74055	74055	74055	22.2322	22.2322	22.2322
12	73924	73924	73924	20.2291	20.2291	20.2291
13	74082	74082	74082	19.0127	19.0127	19.0127
14	76784	76784	76784	18.9759	18.9759	18.9759
15	18958	18958	18958	15.4837	15.4837	15.4837
16	62064	62064	62064	12.0416	12.0416	12.0416

Table 6.11. *Results obtained for the time-travel query* 142*, enriched with the time-point* 6*.*

the application of such process, we give some details of the hardware whereby we have actually performed the tests, outlined above. These data are reported in Table 6.13.

In Figure 6.3 there are two histograms representing the execution time and the memory consumption of the developed strategies along with the Terrier real-time indexing process on the AH-MONO-IT test collection. Comparing the data riported in Figure 6.3 and in particular the ones related to the memory consumption, we can conclude that the greatest contribution to the memory consumption is given by the *mapTermDocuments*, the map the MA strategy is based on. Looking at the matter from another perspective, we can state that maintaining the name of the documents and their identifiers in the central memory only marginally contributes to the memory usage.

The last topic we address in this section is about the time required to answer the topics of

Rank	Terrier	BA	MA	Terrier	BA	MA
1	8242	8242	8242	23.9064	23.9064	23.9064
2	18958	18958	18958	15.6858	15.6858	15.6858
3	689	689	689	12.0537	12.0537	12.0537
4	28831	28831	28831	11.9604	11.9604	11.9604
5	1623	1623	1623	11.7723	11.7723	11.7723
6	4740	4740	4740	11.7216	11.7216	11.7216
7	24209	24209	24209	11.5932	11.5932	11.5932
8	26068	26089	26068	11.4307	11.4307	11.4307
9	8953	8953	8953	11.3431	11.3431	11.3431
10	18969	18969	18969	11.2440	11.2440	11.2440
11	34670	34670	34670	11.2170	11.2170	11.2170
12	16239	16239	16239	11.1094	11.1094	11.1094
13	28417	28417	28417	10.7622	10.7622	10.7622
14	33096	33096	33096	10.5554	10.5554	10.5554
15	32989	32989	32989	10.3686	10.3686	10.3686
16	34332	34332	34332	10.3077	10.3077	10.3077

Table 6.12. *Results obtained for the time-travel query* 142*, enriched with the time-point* 7*.*

the AH-MONO-IT test collection. In the Tables 6.14 and 6.16 we report the times we have obtainined for the BA and MA strategies, during the experimental phase. Specifically, they are the times needed for performing the retrieval process for the BA and MA strategies, in regard to all the time points we have actually enriched with the documents of the AH-MONO-IT corpus.

Looking at the values in Tables 6.14 and 6.16, it should be noted that those presented in 6.16 are greater than those occurring in 6.14. As we remarked above, the MA strategy generally requires us to analyze a lower number of postings than the BA strategy, thus decreasing the time we need to evaluate a query. This case mostly occurs when we match a time- travel query against a versioned collection containing several versions for each document. In fact, the advantages brought by the MA strategy compared to the BA approach, in terms of the number of postings analyzed during the retrieval process, are more significant when the time-point of interest refers to the first versions of the documents in the collection. Note that, this case does not occur here. Actually, even if we have divided the AH-MONO-IT corpus of documents

Component	Name	Characteristics
CPU	Intel(R) Core(TM) i7 CPU 2600	*@ 3.40GHz, 4 core*
		CACHE L1: 128 KB
		CACHE L2: 1024 KB
		CACHE L3: 8192 KB
Ram	2x MT16JSF51264HZ-1G4D1	*Total: 8,00GB Ram*
		DDR3 1333 Mhz
Drive	WDC WD20EARX-00PASB0	*Capacity: 1863 GB*

Table 6.13. *Hardware components description.*

(a) (b)

Figure 6.3. *On the left we give the execution time of the indexing process for the strategies we have discussed in this work as well as for the Terrier real-time indexing process. On the right, we indicate the maximum amount of memory we need for carrying out the indexing process for the examined approaches.*

in 4 groups and we have managed these groups as if they formed indipendent document sets, going through the indexing procedure at different times, these groups actually do not contain multiple versions of the same documents. For this reason, the collection we have created from the AH-MONO-IT corpus of documents can be only considered a particular case of versioned collection. Turning our attention to the data reported in Tables 6.14 and 6.16, the argument given above accounts for the fact that indipendently for the time point we consider, the execution times given for the BA strategy are lower than the corresponding values reported for the MA approach. Specifically, we justify the results proposed in Tables 6.14 and 6.16 by considering the fact that differently from the BA strategy, in the MA strategy the difference between the identifiers related to consecutive versions of the same document is very large. In our context such difference equals the number of the corpus documents, which is 157558. This entails that decoding the data corresponding to a posting in the MA strategy takes more time than that we need to perform the same operation for the BA strategy.

Additionally, in carrying out the MA approach we must also interpret the values we have included into the inverted index stream of bytes to allow the postings fields to be correctly decoded.

Tables 6.15 and 6.17 display the mean and standard deviation values for the execution times we have outlined in Tables 6.14 and 6.16, respectively.

In Tables 6.18 and 6.20 we outline the execution times required to evaluate the queries of the AH-MONO-IT test collection, evaluated after indexing all the groups of documents we have

Query (id)	1	2	3	4	Query (id)	1	2	3	4
	(ms)					(ms)			
141	42.47	74.33	112.80	154.83	174	26.03	52.30	79.10	108.23
142	44.10	85.10	128.33	179.43	176	38.27	75.17	117.67	162.53
143	57.57	118.97	175.27	238.93	177	22.23	44.63	71.20	101.60
145	39.13	77.30	119.33	163.50	178	33.70	65.80	100.07	134.67
147	41.40	79.13	119.70	160.63	179	59.03	120.20	181.70	245.60
148	47.20	93.90	144.83	199.40	180	27.67	55.27	85.67	117.77
149	72.30	139.93	220.60	315.37	181	35.20	69.53	107.20	146.67
150	37.60	74.63	114.93	158.13	182	72.43	142.53	223.63	305.23
151	53.20	106.97	162.60	219.80	183	30.10	61.87	93.90	127.07
152	23.17	45.83	69.67	94.33	184	26.77	52.80	83.70	102.23
153	38.20	76.33	117.43	162.00	185	62.50	125.33	190.80	256.87
154	31.73	62.63	96.83	132.50	186	50.33	101.60	160.57	222.57
155	30.57	60.57	94.60	132.10	187	45.30	90.13	138.17	185.17
156	35.83	70.73	114.37	165.13	188	44.20	87.83	133.17	179.33
157	41.83	83.03	127.60	174.70	189	36.53	73.40	113.07	156.70
159	57.53	116.07	177.63	239.60	190	54.23	109.97	166.97	228.03
161	28.03	54.97	84.00	113.63	192	63.00	127.57	202.30	284.13
162	39.67	78.23	123.60	174.87	193	36.60	70.47	111.97	154.63
163	35.00	69.53	107.37	148.67	194	36.13	71.60	115.23	163.37
164	32.27	64.30	100.77	142.23	195	23.03	44.50	71.40	101.47
165	37.50	75.83	122.47	177.67	196	63.00	126.73	194.30	262.77
166	65.43	132.93	201.43	266.87	197	43.90	88.07	133.30	177.23
167	42.97	85.93	132.83	180.77	198	41.70	83.30	128.73	178.63
168	23.77	47.17	73.13	96.00	199	38.67	77.87	118.17	160.07
171	37.77	76.57	123.00	178.50	200	22.73	45.53	70.40	95.27
173	63.47	126.73	194.67	266.70					

Table 6.14. *Execution times of the retrieval process for the BA strategy, with respect to all the queries composing the AH-MONO-IT test collection. Note that the queries are evaluated taking into account each of the time-points we have provided the corpus documents with. The execution times are expressed in milliseconds. The values we have reported in this table have been obtained by repeating each test for* 50 *times and then averaging the computed execution times.*

	1	2	3	4
μ	41.82	83.15	128.47	176.71
σ	13.21	26.66	40.79	55.64

Table 6.15. *Mean value and standard deviation for the execution times referring to each of the time points we have taken into account in Table 6.14.*

divided the AH-MONO-IT corpus in. Specifically, in Table 6.18 we focus on the BA strategy, while in Table 6.20 we take into account the results obtained for the MA strategy. Looking

Query (id)	1	2	3	4	Query (id)	1	2	3	4
	(ms)					(ms)			
141	47.43	94.40	138.54	193.37	174	30.90	64.81	97.80	133.16
142	52.69	107.43	161.53	220.00	176	46.48	94.98	144.26	204.64
143	73.42	146.45	225.15	308.48	177	26.61	55.43	85.21	123.86
145	47.49	97.89	149.59	202.88	178	40.16	84.65	123.93	170.09
147	51.38	99.10	151.78	201.20	179	74.46	155.80	234.63	319.97
148	56.97	117.86	180.76	253.90	180	32.86	70.14	105.55	145.03
149	93.24	179.94	280.98	392.04	181	42.31	88.44	132.58	184.58
150	45.53	96.06	144.42	196.57	182	96.58	186.74	289.46	394.72
151	66.74	136.27	206.34	282.66	183	36.67	77.36	114.80	157.09
152	27.01	57.18	85.34	116.69	184	31.74	65.96	102.53	148.97
153	46.17	96.79	146.13	207.05	185	82.68	165.36	247.49	337.16
154	38.03	79.82	119.16	163.17	186	65.63	130.84	206.22	290.24
155	43.35	75.72	115.99	165.11	187	56.22	115.81	173.31	234.61
156	42.98	90.59	141.53	211.44	188	53.40	111.67	166.88	227.60
157	51.25	109.79	158.89	221.49	189	44.90	96.08	140.38	194.83
159	75.61	150.39	228.91	313.05	190	69.05	139.11	214.77	296.40
161	34.15	70.11	103.58	140.68	192	84.45	167.43	262.11	364.11
162	51.81	99.36	157.34	224.68	193	45.81	89.20	140.40	195.27
163	42.13	87.58	132.45	185.55	194	43.89	90.51	142.57	206.22
164	39.22	82.23	124.74	178.72	195	27.09	55.97	86.15	123.64
165	46.62	97.10	151.47	226.01	196	83.66	165.38	250.48	343.71
166	85.77	174.95	261.88	347.85	197	53.89	112.08	168.46	225.03
167	52.94	108.73	167.80	228.85	198	52.96	105.63	161.18	226.23
168	27.97	59.11	87.83	117.98	199	47.37	97.79	149.56	202.03
171	45.53	96.94	151.99	27.63	200	26.50	56.81	85.47	116.19
173	84.54	165.76	254.05	349.42					

Table 6.16. *Execution times of the retrieval process for the MA strategy in regard to all the queries of the AH-MONO-IT test collection. The queries have been evaluated by adopting the approach we have actually described in Section 6.2.The values we have reported in this table have been obtained by repeating each test for* 50 *times and then averaging the computed execution times.*

at the Table 6.18, we can observe that the execution time increases as we take into account increasing values for the time-points reported in that table. At a first glance, this could seem a

	1	2	3	4
μ	52.28	106.30	161.85	224.35
σ	18.41	35.39	54.73	74.20

Table 6.17. *Mean value and standard deviation for the execution times referring to each of the time points we have taken into account in Table 6.16.*

Query (id)	1	2	3	4	Query (id)	1	2	3	4
	(ms)					(ms)			
141	74.34	100.72	124.27	155.32	174	48.70	67.82	88.15	110.28
142	82.68	111.44	143.99	175.27	176	74.23	101.98	131.56	165.50
143	109.83	149.66	196.01	239.84	177	44.91	62.64	80.07	102.84
145	76.10	103.36	135.13	162.76	178	65.00	90.23	111.43	138.73
147	79.63	104.32	135.76	162.76	179	108.52	157.02	200.92	247.41
148	89.42	124.59	160.79	201.71	180	52.18	73.77	94.10	119.35
149	134.87	181.75	240.27	303.05	181	68.01	93.75	118.04	148.13
150	74.18	101.53	130.94	158.74	182	137.98	186.02	245.45	302.85
151	102.02	139.64	180.90	222.15	183	58.11	80.88	103.29	127.95
152	42.91	60.47	77.88	96.13	184	54.70	72.93	95.85	121.73
153	73.05	102.99	129.89	164.96	185	118.79	163.15	210.36	263.82
154	61.07	84.89	107.40	133.89	186	103.03	135.58	181.83	226.21
155	58.00	80.90	105.22	133.66	187	88.67	120.29	156.18	188.84
156	70.53	98.81	126.99	167.45	188	83.08	116.16	148.92	180.24
157	80.87	112.43	141.43	176.41	189	69.41	97.81	126.16	157.44
159	111.04	151.25	197.94	241.98	190	107.19	143.65	188.99	231.22
161	53.80	73.29	94.31	115.14	192	126.83	168.25	225.23	282.16
162	81.82	106.47	142.05	176.80	193	74.19	96.12	126.59	156.99
163	69.00	93.98	120.64	149.73	194	72.71	98.55	129.30	166.65
164	66.36	89.82	113.50	143.97	195	47.54	62.77	80.94	103.32
165	76.28	105.58	137.73	180.04	196	121.29	166.24	213.99	264.59
166	125.65	170.71	222.53	269.18	197	84.90	114.91	149.15	179.03
167	85.49	114.29	151.54	183.91	198	82.55	111.68	144.25	179.82
168	44.58	63.05	79.65	97.63	199	76.75	103.14	133.51	162.45
171	77.02	106.45	137.72	181.34	200	42.98	60.96	77.52	96.21
173	122.49	166.33	218.76	268.56					

Table 6.18. *Execution times of the retrieval process for the BA strategy with respect to all the queries of the AH-MONO-IT test collection. The queries have been evaluated after the four groups of documents given in Table 6.3 have gone trough the indexing procedure. The values we have reported in this table have been obtained by repeating each test for* 50 *times and then averaging the computed execution times.*

	1	2	3	4
μ	68.57	110.69	143.43	178.18
σ	13.70	33.65	44.92	54.96

Table 6.19. *Mean value and standard deviation for the execution times referring to each of the time points we have taken into account in Table 6.18.*

bit strange, since answering a time-travel query by using the BA strategy involves analyzing all the postings related to the query terms, indipendently from their time intervals. In fact, even if evaluating a time-travel query requires us to take into consideration all the postings composing the inverted index, regardless the time-point of the query of interest, as shown in Table 6.22, taking into account a lower time point allows us to avoid executing a large amount of operations, such as the selection of the posting lists, we have to consider in each step of the Scoring phase. An argument supporting this reasoning is that the values reported for the time-point 4 in Table 6.20 are very similar to the ones provided in Table 6.16 for the same time-point. For what concerns the Table 6.20, we can note that, with the exception

Query (id)	1	2	3	4	Query (id)	1	2	3	4
	(ms)					(ms)			
141	103.97	135.25	162.46	200.11	174	68.51	90.92	111.64	134.52
142	117.68	154.38	190.64	226.94	176	105.43	139.98	171.12	212.41
143	158.16	208.53	262.77	313.15	177	61.80	82.35	100.42	126.33
145	108.83	142.58	175.53	209.01	178	90.72	119.53	144.16	175.87
147	110.93	142.10	176.94	206.97	179	159.28	215.99	270.37	326.43
148	129.92	174.33	215.80	263.32	180	73.89	98.29	120.85	147.41
149	192.05	250.35	319.54	396.14	181	95.45	126.41	156.23	192.09
150	104.85	137.03	169.63	202.87	182	198.74	260.56	328.60	398.17
151	147.85	195.85	244.22	289.74	183	81.79	107.80	131.70	159.53
152	59.43	79.99	96.99	117.34	184	75.98	98.78	123.21	154.43
153	105.57	142.11	173.55	217.20	185	169.80	225.48	280.83	343.30
154	86.44	113.45	137.36	165.77	186	147.85	190.73	243.05	294.97
155	82.39	109.85	135.68	169.80	187	125.58	164.87	205.62	241.86
156	102.38	137.47	171.17	219.60	188	121.19	160.48	197.08	234.52
157	115.81	153.02	186.69	226.95	189	99.81	134.11	163.60	200.96
159	159.06	210.27	265.47	316.31	190	154.45	200.99	252.93	302.93
161	74.84	99.07	119.24	141.44	192	180.34	233.51	298.74	366.18
162	115.88	153.02	186.69	226.95	193	103.86	130.59	166.73	200.22
163	96.56	127.85	157.18	191.75	194	103.12	134.57	169.53	213.71
164	93.98	124.43	148.54	180.79	195	63.28	82.14	101.31	126.38
165	111.45	147.64	184.11	236.30	196	174.69	229.95	288.08	350.12
166	179.67	238.25	299.32	353.78	197	121.76	159.68	198.12	231.08
167	121.14	157.37	199.25	236.28	198	118.38	156.82	193.60	234.54
168	61.39	82.71	99.67	118.56	199	107.50	141.24	175.05	207.69
171	109.92	146.90	182.99	236.39	200	59.08	80.10	97.37	118.76
173	178.08	233.15	293.94	353.00					

Table 6.20. *Execution times of the retrieval process for the MA strategy with respect to all the queries of the AH-MONO-IT test collection. The queries have been evaluated after the four groups of documents given in Table 6.3 have gone trough the indexing procedure. The values we have reported in this table have been obtained by repeating each test for* 50 *times and then averaging the computed execution times.*

	1	2	3	4
μ	115.50	152.10	189.15	229.72
σ	36.89	48.26	62.31	74.81

Table 6.21. *Mean value and standard deviation for the execution times referring to each of the time points we have taken into account in Table 6.20.*

of the time-point 4, the execution times are significantly greater than those given in Table 6.16. This is due to the fact that evaluating a query after indexing all the group of documents, we have outlined above, requires us to analyze a greater number of postings than those we have to consider by adopting the strategy we have described in this section. We recall that it involves evaluating all the queries of the examined test collection after incrementally indexing the group of documents we have divided the corpus of the AH-MONO-IT test collection. In order to support this argument, in Table 6.23 we report the number of postings analyzed during the evaluation of some of the queries of the examined test collection, for both the cases we have considered above. Note that the values presented in Tables 6.16 and 6.20 for

Query (id)	1	2	3	4	Query (id)	1	2	3	4
	(number of postings)					(number of postings)			
141	552446	552446	552446	552446	141	552446	552446	552446	552446
145	597110	597110	597110	597110	145	597110	597110	597110	597110
152	316778	316778	316778	316778	152	316778	316778	316778	316778
163	521871	521871	521871	521871	163	521871	521871	521871	521871
174	376769	376769	376769	376769	174	376769	376769	376769	376769
183	451460	451460	451460	451460	183	451460	451460	451460	451460
192	927569	927569	927569	927569	192	927569	927569	927569	927569
200	306751	306751	306751	306751	200	306751	306751	306751	306751

Table 6.22. *Number of postings analyzed during the retrieval process for the BA and MA strategies, respectively. The queries, whose identifiers are reported in the first column of the table have been evaluated after indexing all the documents of the AH-MONO-IT corpus, On the left we propose the results for the BA approach, while the ones related to the MA strategy are reported on the right.*

the time-point 4 are very similar with one another, since the evaluation of the AH-MONO-IT queries for such time-point involves analyzing all the postings of the inverted index for both the cases we have considered in such tables.

Query (id)	1	2	3	4	Query (id)	1	2	3	4
	(number of postings)					(number of postings)			
141	552446	552446	552446	552446	141	123520	258905	398441	552446
145	133736	280283	430504	597110	145	597110	597110	597110	597110
152	316778	316778	316778	316778	152	70798	148350	227995	316778
163	521871	521871	521871	521871	163	113890	238404	370296	521871
174	376769	376769	376769	376769	174	82270	172344	267960	376769
183	451460	451460	451460	451460	183	103634	216587	329535	451460
192	927569	927569	927569	927569	192	203773	426266	660915	927569
200	306751	306751	306751	306751	200	64881	138215	216583	306751

Table 6.23. *Number of postings examined during the retrieval procedure for the MA strategy. On the left we propose the values we have obtained by answering the queries, whose identifiers are given in the first column, after we have indexed all the documents of the AH-MONO-IT corpus. On the right we show the execution times we have obtained by adopting the incremental strategy described in Section 6.2*

.

6.2.2 Experimental results: COOKING-MONO-EN

The Table 6.24 reports the execution times we have obtained by evaluating the five topics presented in Figure 6.1 after indexing all the documents of the COOKING-MONO-EN corpus. Looking at the values reported in Table 6.24, we can observe that the MA strategy outperforms the BA approach with respect to almost all the examined time-points. Specifically, for low values of the time-points we have provided with the queries of the COOKING-MONO-EN collection, the MA approach allows us to perform the retrieval

Query	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
(time-points)			(ms)					(ms)		
1	55.36	44.13	29.76	49.96	24.73	14.67	12.18	8.31	12.28	7.11
12	48.36	40.93	28.90	49.10	24.30	14.70	14.38	7.70	13.11	7.64
24	47.56	39.73	28.46	47.4	25.1	14.96	14.48	4.80	16.74	5.31
36	49.73	39.73	28.80	47.56	24.33	20.16	13.90	9.61	17.63	5.69
48	47.83	39.73	29.16	47.6	24.13	19.21	17.73	12.64	18.59	9.41
60	47.43	39.60	29.43	48.26	24.13	19.81	24.80	16.74	15.69	16.32
72	48.86	39.83	28.26	47.83	25.33	20.51	23.17	18.66	22.04	14.67
84	47.43	38.06	28.60	49.63	24.46	23.99	22.18	18.51	22.81	14.58
96	47.43	38.60	29.86	48.78	24.33	28.07	22.31	18.47	30.80	15.95
108	48.60	40.43	28.53	48.30	24.46	30.18	22.70	21.01	31.08	17.10
120	47.60	38.36	28.33	49.20	24.50	31.75	23.01	19.75	33.70	16.80
132	47.33	39.13	28.86	49.40	24.60	33.41	29.17	19.34	36.10	17.40
144	47.50	40.83	28.93	48.93	24.53	36.80	30.00	18.18	37.41	18.42
156	47.76	40.03	28.9	48.16	24.40	37.63	33.27	19.82	38.50	18.63
168	48.33	39.66	28.73	48.10	24.60	39.96	30.56	24.17	40.61	18.92
180	48.93	40.00	28.63	48.16	24.36	40.97	32.17	26.28	39.40	21.51
192	48.4	40.0	28.63	48.16	24.36	42.93	32.85	26.23	43	21.81
204	47.76	39.03	28.70	49.66	25.06	45.70	33.77	23.84	46.11	21.60
216	47.33	41.5	28.46	48.36	24.53	46.20	36.04	26.04	45.98	24.04
228	47.73	39.63	28.90	49.76	24.73	47.72	35.96	29.44	46.55	23.87
240	48.30	40.46	28.93	48.70	25.13	50.35	38.77	28.28	52.10	22.10
252	49.10	39.43	28.73	48.11	25.10	50.96	37.65	33.38	47.30	23.03
264	48.43	40.16	28.93	48.30	25.40	50.48	37.05	31.60	51.51	29.41
276	48.43	40.73	28.80	48.23	24.86	53.08	39.48	32.98	53.89	24.91
288	48.3	39.93	29.03	48.23	26.03	53.98	38.02	33.42	55.55	25.51
300	47.36	40.43	29.43	48.16	24.36	54.18	43.57	32.40	54.61	30.11

Table 6.24. *On the left we propose the execution times for the retrieval process with respect to the BA strategy, while on the right we give the execution times in regard to the MA approach. We point out that the results reported in this table have been obtainined by repeating each measure* 50 *times and then averaging the computed values.*

process in much less time than that required for the BA strategy. This results from the fact that the MA strategy, as we stated several times throughout this section, speeds up query evaluation by skipping some postings which do not match for the query time-predicate. As shown in Figure 6.4, once we have established the bag-of-words composing the query, the number of postings analyzed during the retrieval

process rapidly grows as we take into account increasing values for the time-point we enrich the query with. In fact, answering the first query (blu curve in Figure 6.4), requires us to analyze 5167 postings with respect to the time-point 1 and 601969 in regard to the time-point

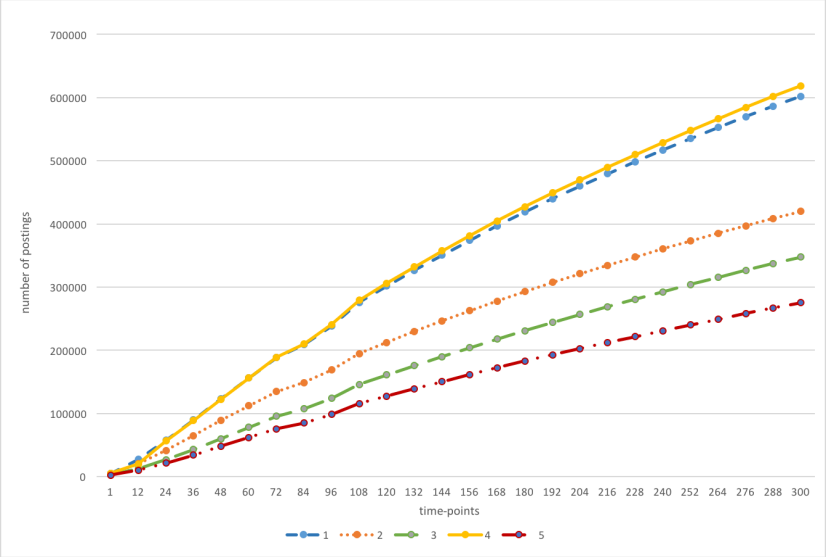

Figure 6.4. *Representation of the number of postings examined during the retrieval process as a function of the time-points associated with the COOKING-MONO-EN queries. Here, we assume that the topics are evaluated after performing the indexing process on all the documents of the COOKING-MONO-EN corpus.*

300. It is worthwhile to note that the fact that the trend of the curves depicted in Figure6.4 is nearly linear exclusively depend on the documents of the corpus and the queries of interest. It indicates that the query terms occur uniformly in the versions of the COOKING-MONO-EN corpus. Finally, turning again our attention to the Table 6.24, we observe that going along with the time-points, the difference between the execution times reported for the BA and MA strategies decreases, until the BA approach outperforms the MA approach. This derives from the fact that, considered the same number of postings, carrying out the retrieval procedure for the BA strategy takes less time than performing the same process for the MA strategy, because of the different techniques we adopt for encoding the posting data. Therefore, taking into account the retrieval process, as soon as the number of postings analyzed for the MA strategy gets closer to the number of postings examined for the BA strategy, the difference in the execution times between the implemented strategies approaches tends to 0.

7 Conclusions

7.1 Conclusions

The objective of this work was to develop a functionality which supported time-travel text search over temporally versioned collections. In the experimental phase, we have implemented such feature on Terrier, an information retrieval system being currently developed by the Department of Computing Science of the University of Glasgow. First of all, we have throughly examined some ideas proposed in the literature, intended for letting users match standard queries enriched with time predicates against documents we have provided with temporal data. Such queries have been referred to as time-travel queries. In the experimental phase, we have considered collections made up of multiple versions of the same document, each of them enriched with a time interval specifying when it actually represents the related document. Throughout this work such collections of documents have been referred to as temporally versioned collections. The strategies we have examined aim at speeding up the retrieval process by arranging the data in the inverted index in such a way to take advantage of the temporal data related to the documents in the collection. The most promising approaches involved slicing the inverted index either along the time-axis or along the temporal dimension, to minimize the number of postings analyzed during the retrieval process, by ignoring the postings which certainly do not contribute to define the final scores of the document versions occurring in the collection. Although the ideas underlying the examined approaches were promising, in the experimental phase we could not exploit such ideas, because implementing such strategies would have required us to completely change how the real-time indexing and retrieval processes are performed in Terrier. Therefore, we have devised two strategies, referred to as the BA and the MA strategies. In designing such

approaches we have exclusively focused on queries enriched with time points rather than time intervals. BA stands for Basic Approach for index versioning. It carries out the evaluation of a time-travel query by ruling out all the postings whose time interval does not satisfy the time predicate of the query. In fact, we point out that this strategy corresponds to the baseline approach, since using this strategy for query evaluation requires us to analyze all the postings of the posting lists related to the query terms, regardless the time point of the query. MA stands for Mapping approach for index versioning and it allows us to speed up the retrieval process, by exploiting a map associating each index term with the first posting of a document in which that term occurs in. In order to take advantage of this map, we have to ensure that the postings associated with successive versions of the same document, occupy adjacent positions in each of the inverted lists composing the inverted index. It is worthwhile to point out that in designing the BA and MA approaches, we had to satisfy all the requirements related to the Terrier real-time indexing and retrieval processes, such as the need of flushing the data collected during the indexing process at regular intervals or the managing of the update of the collection statistics as the documents of the collection evolved over time. These aspects have significantly limited the range of possibilities we could consider in implementing the BA and MA strategies. To verify the correctness of the BA and MA strategies as well as to evaluate the retrieval efficiency, we have experimented with them on two collections. The former was the CLEF Adhoc monolingual text collection for the Italian language, referred to as AH-MONO-IT. The second collection has been generated starting from the English Wikipedia. In the experimental phase we have denoted such collection as COOKING-MONO-EN. Both the corpora of the collections we have taken into consideration have been divided in several groups of documents in order to generate temporally versioned collections. Therefore, the AH-MONO-IT test collection contains 4 document sets, each of them containing almost the same number of documents. The COOKING-MONO-EN collection consists of 1000 documents, each of them occurring in multiple versions, up to a maximum of 300 versions. It is worth remarking that the AH-MONO-IT test collection is a special case of versioned collection, since the collection contains an only version for each document occurring in it. The tests we have performed on the examined collections fall into two categories. The former group of tests aims at evaluating if the developed strategies give back the same raking lists we would have obtained by evaluating the queries of interest by adopting the Terrier real-time indexing and retrieval processes. The latter group deals with evaluating the time we actually need for query evaluation as well as the amount of memory

which is required to perform the indexing process for both the developed strategies. We conclude that the MA strategy achieves a better time performance than the BA approach when the number of postings analyzed during the retrieval process for the MA strategy is lower than the one taken into accout for the BA strategy. As the difference between the number of such postings decreases, the execution times obtained for the BA and MA strategies get closer. We remind that for the MA strategy the number of postings we encounter during the retrieval process strictly depends on the time point we enrich with the query of interest. In fact, given a time-point, evaluating a query for the MA approach involves decoding not only the postings whose time interval contains the time point of interest, but also the ones whose time interval boundaries are earlier than the examined time-point. Moreover, the results proposed in Chapter 6 demonstrate that taking into account the same number of postings, the BA strategy outperforms the MA approach, because of the longer time required for decoding the posting data for the MA strategy, compared to the BA approach. This is due to the fact that, both for the BA and the MA strategies, we store document identifiers by adopting the d-gap technique. This involves including in the inverted index the difference between the identifiers of adjacent postings rather than storing their identifiers as they are. At this stage, we remind that while for the BA strategy the difference between the identifiers of adjacent postings in an inverted list is always 1, the same does not hold true for the MA strategy, where the difference between the identifiers of adjacent postings is large. As future works we envision the possibility to analyze the change in the performance of the described strategies we could obtain by modifying the compression techniques used for encrypting the data of the postings in the inverted index. Indeed, implementing the BA and MA strategies, we have learnt that in order to significantly improve the execution time we need to answer a given query, by conceiving a new index organization scheme, we can not ignore how this scheme actually would affect the efficiency of the compression techniques used for storing the data of the collection documents.

Bibliography

[1] Anand, A. and Bedathur, S. and Berberich, K. and Schenkel, R. (2012). Index maintenance for time-travel text search. In Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval (SIGIR '12). ACM, New York, NY, USA, 235-244.

[2] Salton, G. (1968). Automatic Information Organization and Retrieval. McGraw-Hill, New York.

[3] Lancaster, F. W. (1968). Information Retrieval Systems: Characteristics, Testing and Evaluation, Wiley, New York.

[4] van Rijsbergen, C. J. (1975). Information Retrieval, Butterworths.

[5] Spark Jones, K. and Willett, P., eds. (1997). Readings in Information Retrieval. Morgan Kaufmann, San Mateo, CA.

[6] Croft, W. B. and Metzler, D. and Strohman, T. (2010). Search Engines: Information Retrieval in Practice, Addison-Wesley.

[7] Luhn, H. P. (1958). The automatic creation of literature abstracts, IBM Journal of Research and Development, 2, 159-165.

[8] Mandelbrot, B. (1966). Information theory and psycholinguistics: a theory of words frequencies P. Lazafeld, N. Henry (Eds.), Readings in Mathematical Social Science, MIT Press, Cambridge, MA.

[9] Singh, J. and Gupta, V. (2016). Text Stemming: Approaches, Applications, and Challenges. ACM Computing Surveys, 49(3):1–46.

[10] Robertson, S. E. (2004). Understanding inverse document frequency: On theoretical arguments for IDF. Journal of Documentation, 60, 503–520.

[11] Salton, G. and Wong, A. and Yang, C. S. (1975). A vector space model for automatic indexing. Commun. ACM 18, 11, 613-620.

[12] Zobel, J. and Moffat, A. (2006). Inverted files for text search engines. ACM Comput. Surv. 38, 2, Article 6.

[13] Ounis, I. and Amati, G. and Plachouras, V. and He, B. and Macdonald, C. and Lioma, C. (2006). Terrier: A High Performance and Scalable Information Retrieval Platform. In Proceedings of ACM SIGIR'06 Workshop on Open Source Information Retrieval. Seattle, Washington, USA.

[14] Terrier home: Terrier IR Platform - Homepage. Available: `http://terrier.org/Documentation`.

[15] Hiemstra, D. (2000). A probabilistic justification for using tf-idf term weighting in information retrieval.

[16] Elias, P. (1975). Universal codeword sets and representations of the integers. Trans. Info. Theory 21.

[17] Catena, M. and Macdonald, C. and Ounis, I. (2014). On Inverted Index Compression for Search Engine Efficiency. In: Proc. ECIR '14.

[18] Williams, H. E. and Zobel, J. (1999). Compressing integers for fast file access. The Computer Journal 42.3: 193-201.

[19] Porter, M. F. (1980). An algorithm for suffix stripping, Program, Vol. 14 Issue: 3, pp.130-137.

[20] Robertson, S. and Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond, Foundations and Trends in Information Retrieval. Vol. 3: No. 4, pp 333-389.

[21] Robertson, S. E. and Walker, S. (1994). Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In Proceedings of the 17th Annual ACM SIGIR Conference on Research and Development in Information Retrieval, Dublin, Ireland, W. B. Croft and C. J. van Rijsbergen, Eds. ACM, New York, 232–241.

[22] Spark Jones, K. (1972). A Statistical Interpretation Of Term Specificity and its application in retrieval, Journal of Documentation , Vol. 28 Issue: 1, pp.11-21.

[23] He, J. and Suel, T. (2011). Faster temporal range queries over versioned text. In: Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information - SIGIR.

[24] Berberich, K. and Bedathur, S. and Neumann, T. and Weikum, G. (2007). A time machine for text search. In Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '07). ACM, New York, NY, USA, 519-526.

[25] Hou, L. U. and Mamoulis, N. and Berberich, K. and Bedathur, S. (2010). Durable top-k search in document archives. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 555-566.

[26] Beyer, K. and Haas, P. J. and Reinwald, B. and Sismanis, Y. and Gemulla, R. (2007). On synopses for distinct-value estimation under multiset operations. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07). ACM, New York, NY, USA, 199-210.

[27] Anand, A. and Bedathur, S. and Berberich, K. and Schenkel, Ralf. (2010). Efficient temporal keyword search over versioned text. In Proceedings of the 19th ACM international conference on Information and knowledge management (CIKM '10). ACM, New York, NY, USA, 699-708.

[28] Anand, A. and Bedathur, S. and Berberich, K. and Schenkel, R. (2011). Temporal index sharding for space-time efficiency in archive search. In Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval (SIGIR '11). ACM, New York, NY, USA, 545-554.

[29] Keogh, E. J. and Chu, S. and Hart, D. and Pazzani, M. J. (2001). An Online Algorithm for Segmenting Time Series. In Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM '01), Nick Cercone, Tsau Young Lin, and Xindong Wu (Eds.). IEEE Computer Society, Washington, DC, USA, 289-296.

[30] Harman, D. (2011). Information Retrieval Evaluation. Morgan Claypool Publishers.

[31] Kendall, M. G. (1938). Biometrika, Volume 30, Issue 1-2, Pages 81–93.

[32] Heman, S. (2005). Super-scalar database compression between RAM and CPU-cache. MS Thesis, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands, July.