Indexing Methods for Web Archives

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Avishek Anand
Max-Planck-Institut für Informatik

Saarbrücken
2013

Dekan – Dean

Prof. Dr. Mark Groves	Universität des Saarlandes	Saarbrücken, Germany

Betreuender Hochschullehrer – Supervisor

Dr. Klaus Berberich	MPI Informatik	Saarbrücken, Germany

Gutachter – Reviewers

Dr. Klaus Berberich	MPI Informatik	Saarbrücken, Germany
Prof. Dr. Gerhard Weikum	MPI Informatik	Saarbrücken, Germany
Prof. Dr.-Ing. Kjetil Nørvåg	NTNU	Trondheim, Norway

Kolloquiums – Defense

Datum – Date

September 06, 2013, in Saarbrücken

Vorsitzender – Head of Colloquium:

Prof. Dr. Dietrich Klakow	Universität des Saarlandes	Saarbrücken, Germany

Prüfer – Examiners:

Dr. Klaus Berberich	MPI Informatik	Saarbrücken, Germany
Prof. Dr. Gerhard Weikum	MPI Informatik	Saarbrücken, Germany

Protokoll – Reporter:

Dr. Pauli Mietinnen	MPI Informatik	Saarbrücken, Germany

Avishek Anand
Max-Planck-Institut für Informatik
Campus E14
D-66123, Saarbrücken
aanand@mpi-inf.mpg.de

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, den 02.07.2013

(Avishek Anand)

Acknowledgements

First and foremost, I would like to thank Prof. Gerhard Weikum for giving me the opportunity to pursue this thesis under him and his timely and valuable inputs. I would like to express my sincere gratitude for his support and guidance.

A special note of thanks to Dr. Klaus Berberich and Prof. Srikanta Bedathur. Their contribution over the past years has been invaluable and I have learned a lot from them. I thank them for their persistent support and patience through many discussions we had through the course of the thesis.

I am very thankful to my parents and friends for the continued emotional support which in the past few years has often proven to be the deciding factor for my successes. Last but not the least I would like to thank my wife, Megha, for always being there and putting up with me during good times and bad. Her being there has made this a journey to cherish.

Abstract

There have been numerous efforts recently to digitize previously published content and preserving born-digital content leading to the widespread growth of large text repositories. Web archives are such continuously growing text collections which contain versions of documents spanning over long time periods. Web archives present many opportunities for historical, cultural and political analyses. Consequently there is a growing need for tools which can efficiently access and search them.

In this work, we are interested in indexing methods for supporting text-search workloads over web archives like *time-travel queries* and *phrase queries*. To this end we make the following contributions:

- Time-travel queries are keyword queries with a temporal predicate, e.g., "mpii saarland" @ `[06/2009]`, which return versions of documents in the past. We introduce a novel index organization strategy, called *index sharding*, for efficiently supporting time-travel queries without incurring additional index-size blowup. We also propose index-maintenance approaches which scale to such continuously growing collections.

- We develop query-optimization techniques for time-travel queries called *partition selection* which maximizes recall at any given query-execution stage.

- We propose indexing methods to support phrase queries, e.g., "to be or not to be that is the question". We index multi-word sequences and devise novel query-optimization methods over the indexed sequences to efficiently answer phrase queries.

We demonstrate the superior performance of our approaches over existing methods by extensive experimentation on real-world web archives.

Kurzfassung

In der jüngsten Vergangenheit gab es zahlreiche Bemühungen zuvor verffentlichte Inhalte zu digitalisieren und elektronisch erstellte Inhalte zu erhalten. Dies führte zu einem weit verbreitenden Anstieg groer Textdatenbestände. Webarchive sind eine solche Art konstant ansteigender Textdatensammlung. Sie enthalten mehrere Versionen von Dokumenten, welche sich über längere Zeiträume erstrecken. Darüber hinaus bieten sie viele Möglichkeiten für historische, kulturelle und politische Analysen. Infolgedessen gibt es einen wachsenden Bedarf an Werkzeugen, die eine effiziente Suche in Webarchiven und einen effizienten Zugriff auf die Daten erlauben.

Der Fokus dieser Arbeit liegt auf Indexierungsverfahren, um die Arbeitslast von Textsuche auf Webarchiven zu unterstützen, wie zum Beispiel time-travel queries oder phrase queries. Zu diesem Zweck leisten wir folgende Beiträge:

- Time-travel queries sind Suchwortanfragen mit einem temporalen Prädikat. Zum Beispiel liefert die Anfrage “mpii saarland” @ `[06/2009]` Versionen des Dokuments aus der Vergangenheit als Ergebnis. Zur effizienten Unterstützung solcher Anfragen ohne die Indexgröe aufzublasen, stellen wir eine neue Strategie zur Organisation von Indizes dar, so genanntes *index sharding*. Des Weiteren schlagen wir Wartungsverfahren für Indizes vor, die für solch konstant wachsende Datensätze skalieren.

- Wir entwickeln Techniken zur Anfrageoptimierung von time-travel queries, nachstehend *partition selection* genannt. Diese maximieren den Recall in jeder Phase der Anfrageverarbeitung.

- Wir stellen Indexierungsmethoden vor, die phrase queries unterstützen, z. B. “Sein oder Nichtsein, das ist hier die Frage”. Wir indexieren Sequenzen bestehend aus mehreren Wörtern und entwerfen neue Optimierungsverfahren für die indexierten Sequenzen, um phrase queries effizient zu beantworten.

Die Performanz dieser Verfahren wird anhand von ausführlichen Experimenten auf realen Webarchiven demonstriert.

Contents

1 Introduction

1.1. Motivation

With the popularity and ubiquity of the Internet our digital presence and footprint has been growing at a rapid pace. This has resulted in numerous digital curation efforts which are believed to lead to cultural preservation [Con10]. Curation in the form of preservation of digital information deals with archiving born-digital content like Web archives and news archives. Web archives such as *Internet Archive* [http://www.archive.org] and *Internet Memory* [http://www.internetmemory.org] have been involved in periodically archiving websites for over 17 years with collection sizes amounting to several terabytes of data. Similarly, many news companies such as *The New York Times* [NYT13], *Wall Street Journal* [WSJ13], *The Times* [TIM13] archive both their published digital content as well as digitize non-digital articles. Digitization for preservation [KUL], apart from newspapers, also targets other content generated before the Web era in form of digitized books [Coy06], a prominent example being the Million Books Project [MBP13]. All these efforts have given rise to huge repositories of text data which span considerable time periods. In this work we collectively refer to these collections as *web archives*.

Cultural preservation is the first step, but the true potential of these collections can be realized by enabling efficient searching and mining tasks. Web archives present many opportunities for various kinds of historical analyses [SSU08, HER13], cultural analyses [MSA+10], and analytics for computational journalism [CHT11, CLYY11]. A case in point is data-driven journalism where the user resorts to various search tasks, visualization and longitudinal analyses on large amounts of data for obtaining useful insights in order to construct stories. As an example, *The Weyeser Explorer* [http://www.data-art.net/weyeser_explorer] provides visualizations of word clusters from news articles from *The Guardian* [http://www.guardian.co.uk/]. In this work we are interested in indexing methods for supporting various text-search workloads over web archives.

With the popularity of search engines, full-text search has evolved into a powerful and popular way of searching text collections. However, search is no longer limited to humans typing queries. It is increasingly being used as a "primitive" operation in preprocessing steps in a longer pipeline for various extraction, mining and analytics tasks. Entity extraction systems like [ACCG08, KN10] commonly employ techniques based on keyword search as the first step. Similarly, many text mining and analytics applications like [MTB$^+$10, SBBW10] use text search as a filtering step to focus on a smaller and hence more usable document collection. To realize these benefits of search over archives it would be important to support keyword search functionality with additional temporal predicates.

What has also evolved with the wide usage of search engines is the manner in which users interact with it. The user behaviour relies on multiple query reformulations and expansions to refine query intentions. A quick approximation of the query results often serves as feedback for further reformulations. Thus, to enhance the current day user behaviour it is important to provide functionality by which a large subset of the query results can be determined quickly.

Along with keyword queries, phrase queries are another important query type in text search. Although a small fraction of the queries issued to search engines are phrase queries, a fairly large number are implicitly invoked, for instance, by means of query-segmentation methods [HPBS12, LHZW11]. Beyond their usage in search engines, phrase queries increasingly serve as a building block for other applications such as (a) *plagiarism detection* [Sta11] (e.g., to identify documents that contain a highly discriminative fragment from the suspicious document), or (b) *culturomics* [MSA$^+$10](e.g., to identify documents that contain a specific n-gram and compute a frequency time-series from their timestamps).

1.2. Research Challenges

In this section, motivated from the scenarios presented above, we introduce the three different workloads we address in this thesis : **(A)** time-travel queries, **(B)** approximate queries and **(C)** phrase queries. We illustrate these by providing potential usecases for each of the workloads and consider the corresponding research challenges in designing efficient indexing and query processing methods.

(A) In spite of the progresses in preservation, search capabilities over archives have been limited. A Naïve adaptation of the indexing infrastructure used for text retrieval is expensive and does not capture the temporal dimension inherent to such archives. With this in mind, *time-travel queries*, which combine temporal predicates with keyword queries, e.g., "fifa world cup" @ `[06/2006 - 07/2006]`, were proposed in [BBNW07] as an effective way of searching collections.

This combination of keyword queries with a temporal context could be an attractive construct in various analytical and comparative longitudinal analyses. Consider a journalist interested in views about the recent ponzi scheme in 2008. A keyword query "ponzi scheme" without the temporal predicates over an archive might result in articles about various ponzi schemes spanning the entire time-line of the archive. With the proper temporal constraints she would be able to restrict the search to time-intervals of her interest.

To answer time-travel queries [BBNW07] propose indexes that incur an index-size blowup, by replicating parts of the index, for better query performance. This motivates :

RC I : *How do we build index structures which eliminate wasteful replication so as to have smaller index sizes and support time-travel queries efficiently?*

Current indexing techniques are agnostic to index-access costs which can make a considerable difference to retrieval efficiency.

RC II : *Can we devise index-tuning methods which take into account index-access costs rather than abstract cost measures ?*

Web archives are usually in a state of change where content is continuously added. The index needs to be in a fresh state and consistently reflect these changes. The current state-of-art indexing techniques are limited to static collections and do not consider updates. We need index-maintenance strategies which do not compromise query efficiency.

RC III : *Can we design indexes which can be efficiently maintained ?*

(B) Web archives are often associated with redundant information due to periodic recrawls. Also in many application domains like news articles, the same information is available from different documents, so missing a few of them could be acceptable. Similarly, a subset of the true results is usually good enough for quickly checking if content or temporal predicates of the query need to be adapted. Consider a historian interested in the former US president "George Bush" in 2005. He might go through a series of reformulations to clarify his intent from "bush" @ `[2005]` to "george bush" @ `[2005]` to "george bush senior" @ `[2005]`. After the first query, he might have results relating to the infamous bush fires in 2005 prompting him a second round of reformulation. The second round might be unsuccessful due to mentions of the son of the actual entity he is interested in, thus necessitating a third reformulation. A quick review of the results (partially computed thus far) in each of these rounds would allow him for a productive reformulation experience.

Due to the sheer size of archives, processing the temporal queries is expensive. This hampers the user experience when a user does not necessarily require the exact query result, but often would be satisfied with a good approximation that is determined quickly. Hence the need for query-optimization techniques to provide support for quick approximate results.

RC IV : *Can we support query optimizations given current indexes for temporal queries for quick approximate results ?*

(C) Consider a company that is interested in the product reviews for a camera model “canon eos 1100d” which was released in 2008 and is interested in all its mentions for planning for the next product release. This specific model is referred to as “canon rebel xs”, “eos rebel t3” and “eos kiss x50” in different contexts. In order to determine the documents which mention this specific model, these surface forms can be used as *phrase queries* along with the year of its release. The output documents can then be fed into a more elaborate extraction system for further analysis.

Traditional indexing methods for processing phrase queries use inverted indexes with positional information. Phrases are processed by intersecting posting lists corresponding to the query terms and additionally checked for positional proximity. Phrase-query processing in such a setting becomes expensive for large collections because (i) one cannot employ standard retrieval techniques like *stop-word elimination* common to standard keyword retrieval and (ii) additional checks for positional adjacency.

The above problem can be addressed by indexing phrases or multi-word sequences, but explicitly indexing all possible sequences is prohibitive. However, not all phrases are frequently queried and not all combinations of words make valid multi-word sequences. Promising sequences can be mined from the document collection and from workloads. By considering their selectivities in the document collections and frequency in the query workload, practical indexing and efficient phrase querying solutions are possible.

RC V : *How can we index multi-word sequences to improve phrase-query efficiency ?*

RC VI : *How can query processing efficiently answer phrase queries using indexed multi-word sequences ?*

1.3. Contributions and Publications

We now present a synopsis of our contributions made in this work in addressing the research questions posed above. We also mention the key publications in which these contributions appear.

(I) **Scalable and Efficient Support for Time-travel Queries :** We present a novel index organization scheme, called *index sharding*, that results in an almost zero increase in index size. This practically efficient index organization method reconciles the costs of random and sequential accesses, hence minimizing the cost of reading index entries during query processing.

We also describe index maintenance strategies based on which the proposed index can be updated incrementally as new document versions are added to the web archive. Our solution bounds the number of wasted read index entries per posting-list and can be maintained using small in-memory buffers and append-only operations. This work is published in:

- **[ABBS11]:** Avishek Anand, Srikanta Bedathur, Klaus Berberich, Ralf Schenkel. *Temporal Index Sharding for Space-Time Efficiency in Archive Search* in ACM Conference on Research and Development in Information Retrieval, SIGIR 2011.
- **[ABBS12]:** Avishek Anand, Srikanta Bedathur, Klaus Berberich, Ralf Schenkel. *Index Maintenance for Time-Travel Text Search* in ACM Conference on Research and Development in Information Retrieval, SIGIR 2012.

(II) **Supporting Approximate Time-travel Queries :** Building on the state-of-art index partitioning scheme proposed in [BBNW07] we present a framework for efficient approximate processing of time-travel queries and present practical algorithms for the query-optimization problem. We derive a query plan for each time-travel query ensuring that the number of results obtained at each stage of the execution is maximized. Our experiments with three diverse, large-scale text archives reveal that our proposed approach can provide close to 80% recall even when only about half the index is allowed to be read. This work is published in:

- **[ABBS10]:** Avishek Anand, Srikanta Bedathur, Klaus Berberich, Ralf Schenkel. *Efficient Temporal Keyword Queries over Versioned Text* in ACM International Conference on Information and Knowledge Management, CIKM 2010.

(III) **Efficient Indexing and Phrase-Query Processing :** We propose a phrase indexing solution and query-optimization techniques to improve the performance of phrase queries. Our solution augments the existing word level index by a phrase-level index which is tunable by index size and is optimized for phrase-query performance. We study how arbitrary phrase queries can be processed efficiently on such an augmented inverted index. Moreover, we develop methods to select multi-word sequences to be indexed so as to optimize query-processing cost taking into account characteristics of both the workload and the document collection. Experiments on two real-world document collections demonstrate the efficiency and effectiveness of our methods. This work is under peer review:

- Avishek Anand, Ida Mele, Srikanta Bedathur, Klaus Berberich. *Efficient Phrase Indexing and Querying*, under review.

1.4. Outline

This thesis is organized in three parts, each focused on a different workload type. Before we describe our contributions in detail, Chapter 2 introduces the required background and necessary technical foundations on which we build.

Chapter 3 explores a novel space-time efficient index partitioning technique called *index sharding*. It addresses issues surrounding index organization, tuning and maintenance along with extensive experimental evaluation. Chapter 4 introduces the problem of supporting approximate queries by proposing query-optimization techniques over the state-of-art vertically-partitioned index. It discusses the different approaches to this problem along with detailed experimental results. Chapter 5 deals with the last workload type, namely *phrase queries*. It introduces novel indexing and query processing techniques for efficient phrase-query processing. We finally conclude in Chapter 6 by revisiting the research challenges sketched before and discussing our contributions. In addition, we give an outlook on future research directions.

2 Foundations and Technical Background

In this chapter, we introduce the technical background necessary to understanding the contributions presented in this thesis. We start with a brief description of web archives and recent efforts relating to their acquisition and preservation. Next, we give an overview of information retrieval with focus on text retrieval. In particular, we focus on the efficiency aspects of text retrieval and describe issues concerning indexing and query processing over large text collections. Finally, we conclude with the state-of-art techniques relating to indexing archives.

2.1. Web Archiving

The Web is in a continuous state of change [ATDE09, NCO04, FMNW03]. Existing pages are modified, new content is added, and old pages are removed resulting in a change of state of the Web. Web archives prevent this loss by attempting to capture and preserve this knowledge before it disappears. Efforts to preserve web contents have been undertaken both by governments [WAR13, ARK13, BNF13, PAN13], non-profit organizations [IA13, IM13], and companies [HAN13]. Undoubtedly, the most popular and large-scale effort in archiving the web has been the *Internet Archive* [IA13] which has an estimated size of 500 Terabytes and is growing at 100 Terabytes a year.

Crawling in Web Archives Crawling is the most common method of acquiring content for web archives. A web crawler systematically requests and stores content from web pages. It starts visiting pages from a seed set of pages and traverses the website in a breadth-first or depth-first manner. The key difference from crawling strategies in standard web search is that all files require to be fetched as opposed to only the ones which would be later indexed. The challenges in crawling for archiving is usually in capturing a consistent snapshot of a website at a given instance. Usually, due to po-

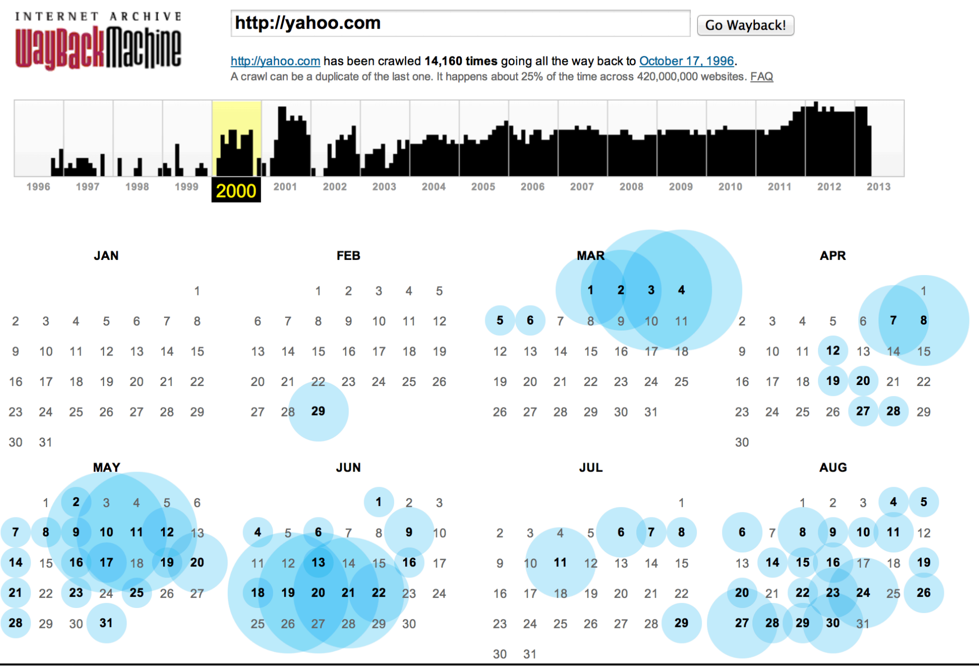

http://yahoo.com has been crawled **14,160 times** going all the way back to October 17, 1996.
A crawl can be a duplicate of the last one. It happens about 25% of the time across 420,000,000 websites. FAQ

Figure 2.1.: Website preserved by *Internet Archive*

liteness requirements, it takes a long time to crawl an entire website. In the meanwhile many portions of the website might undergo changes and modifications which are not captured. Figure 2.1 shows how often and when `yahoo.com` has been crawled and preserved by the Internet Archive.

Denev et al. [DMSW09] propose a model for assessing the quality of the crawled data for web archives. They define *blur* as a stochastic measure of the quality of capture. The longer it takes to capture an entire website the more blur the gathered data is said to have. Further, *coherence* is a deterministic quality measure which determines the accuracy of a snapshot, i.e., number of pages which did not change during the crawl. They further propose effective crawling techniques which optimize for blur and coherence.

The Internet Preservation Consortium (IIPC) [IIP13] and other preservation organization are involved in establishing standards necessary for effective web preservation. IIPC is also involved in the development of open source software, most prominently Heritrix [HER13], for crawling for web archives. For other issues and best practices in web archives please refer to [Mas06].

Although there has been progress in content acquisition for web archives; access methods and search support over them have been limited. Most of the archives either do not have an explicit search interface or provide limited search functionality based on open-source tools like Nutch [NUT13] or Solr [SOL13].

2.2. Data Management

In this section we look at various data management principles and techniques which are either related or used in our work. We briefly discuss the work done in temporal databases which also deals with managing temporal data. Next, we give a detailed account of distinct value estimators and KMV synopsis which we require in Chapter 4 in our work on query optimization for time-travel text search.

2.2.1. Temporal Databases

Temporal databases deal with data management issues pertaining to data associated with temporal aspects. Specifically these temporal aspects include the notion of *transaction* and *valid times*. Transaction time refers to the time when a certain fact was stored in the database. Valid time, on the other hand, refers to the time when the fact existed or was active in the real world. As an example if we enter a fact about the great depression of 1930's into the database, the valid time would refer to a time-interval [1930 - 1940], while the transaction time would be when the entry was made, for example March 2013. Transaction times evolve linearly and are immutable to change in the past unlike valid times where changes in the past are allowed. Research in temporal databases has proposed models, query languages and indexing techniques over arbitrary data with such temporal aspects.

We now discuss in brief about notable index structures and indexing methods for transaction time data. Early approaches, like the Time-Split B-trees (TSBT) [LS93] and Multi-Version B-trees (MVBT) [BGO$^+$96], adapt B$^+$-trees for time-evolving data. The Time-Split B-Tree was aimed at reducing storage costs and improving query performance over current records but did not provide good worst case guarantees. MVBT overcame this problem and also provided the same asymptotic space and time complexity as the B-tree when a single version per record is managed.

In the late nineties, the *log-structured history data access method* (LHAM) was proposed by Muth et al. [MOPW00] for data with high update rates. LHAM stores its data in successive components $C_0, \ldots, C_m$ of varying sizes and access costs. The core idea is that the most recent data is stored in the component with the least access cost C_0 thereby improving update performance. The older records are gradually merged into components with high access costs C_{i+1} whenever a component C_i is full. The partitioning of records into components entails a partitioning of the entire time duration into non-overlapping time intervals. Consequently, each component C_i is associated with a time interval. Typically the component sizes follow a geometric progression and whenever a C_i is full it potentially triggers a series of merges via a *rolling merge* operation. A detailed survey of access methods for temporal databases can be found in [ST99].

2.2.2. Synopsis Structures

Determining number of distinct values in large collections, termed as *distinct-value estimation* or *DV estimation*, is an important task in query optimization, data streams and network monitoring. While the exact count of distinct values is desired, determining them is both computationally expensive and does not scale well. To this extent, approximate methods have been proposed in the literature which efficiently estimate the result like Bloom filters [Blo70], Hash Sketches [FNM85] and KMV Synopsis [BHR+07]. These approaches are based on concise data-structures constructed over the input called *synopsis* or *sketches*. In this section we give brief background information on the synopsis structures, and specifically KMV synopsis, that are utilized by our algorithms.

Bloom Filters

It is a space-efficient probabilistic data structure used for answering set membership, set containment, or set-intersection queries with a high confidence. Bloom filters are concerned with sets and offer less than 10 bits per element for a 1% false positive probability. However, they cannot be employed for arbitrary multiset operations.

Hash Sketches

These are distinct-value estimation technique was proposed in [FNM85]. They rely on a pseudo-uniform hash function over the input to derive a sketch. These sketches allow for various multiset operations by counting distinct elements after allowing appropriately combining the sketches. Multiple intersections introduce high relative errors and has a high computational complexity.

KMV Synopsis

Beyer et al. [BHR+07] introduced *KMV synopses* as effective sketches for sets that support arbitrary multiset operations including union, intersection, and differences. They differ from hash sketches in having lower computational costs and more accurate DV estimation. In this section we first explain the principle of DV estimation and introduce the KMV synopsis data structure. We then explain how multiset operations like *union, intersection* and *set difference* can be applied to them and estimated.

Consider an input multiset S with D distinct values $\Theta(S) = \{v_1, v_2, \dots, v_D\}$ where $\Theta(S)$ represents the domain. Each of these values is mapped to a random location on an unit interval. Assuming that the points are distributed uniformly, the expected distance between any two neighboring points is $1/(D+1) \approx 1/D$. The expected position of the k-th point from the origin or the k-th smallest point, U_k, , is k/D, i.e, $E[U_k] \approx k/D$. The simplest estimator of $E[U_k]$ is simply U_k and yields the *basic estimator* as introduced in [BYJK+02]:

$$\hat{D}_k^{BE} = k/U_k.$$

Consider a hash function $h : \Theta(S) \mapsto 0, 1, \ldots, M$ which maps the input in the range $[0 - M]$ where $M >> D$. The hash function h is chosen such that mapped values $h(v_1), h(v_2), \ldots, h(v_D)$ resemble an independent and identically distributed sample from the discrete uniform distribution on $0, 1, \ldots, M$. Normalizing the range to a unit range, as above, the values U_i can be expressed as $U_i = h(v_i)/M, v_i \in \Theta(S)$. The value M regulates the number of collisions. The higher the value of M the lower the probability of collisions.

Building on this notion of the basic estimator, KMV synopsis is defined in [BHR+07] for a multiset S as follows. Given a hash function h, the k-smallest values of the mapping $h(v_i)$ where $v_i \in \Theta(S)$ constitute the KMV (k-minimum values) synopsis of S. Typically $M = \mathcal{O}(D^2)$ is chosen to avoid collisions and hence each of the synopsis values can be encoded in $\mathcal{O}(\log D)$ bits. Consequently, the number of bits required to store the KMV synopsis is $k\,\mathcal{O}(\log D)$. The KMV synopsis can be created by a single scan over S and only storing the k smallest hashed values implemented efficiently by a priority queue. The basic estimator is further extended to an unbiased estimator $\hat{D}_k$.

$$\hat{D}_k = (k-1)/U_k$$

The cardinality estimation of the multiset operations involving two or more synopses, like unions and intersections, can be carried out efficiently based on the above estimators.

2.3. Information Retrieval

Information retrieval (IR) deals with finding information from large collections of unstructured data to satisfy user information needs. Historically, the focus of IR has been on investigating concepts, models and computational methods for searching large text collections. However, the field has expanded its horizons to multimedia content [Cho10], facts and semi-structured content [Lal12, CMS10]. In this work we focus on text collections and deal with the core problem in IR, i.e.,

We are given a document collection $\mathcal{D}$ associated with a vocabulary of terms $\mathcal{V}$. For a given query $q \in \mathcal{V}$ our aim is to find the documents which satisfy the user information need expressed *by* q.

2.3.1. Retrieval Models

The question above gives rise to two key aspects which account for the *effectiveness* of IR systems – (1) How are documents and queries modelled ? (2) How is the relevance of a document to a given query modelled ?

Boolean Retrieval Many retrieval models have been proposed to model documents and queries and the similarity between them. We provide the key ideas of some of the fundamental retrieval models. The *Boolean retrieval model* is the earliest and simplest retrieval model. Queries in this retrieval model are Boolean expressions comprising of terms $v \in \mathcal{V}$ and connected by Boolean operators (OR, AND, NOT). The notion of relevance of a document d to a query q is binary, i.e., either d is relevant to q or not depending on the evaluation result of the Boolean expression.

Ranked Retrieval Salton introduced the *vector space model* [SWY75] to rank documents according to their perceived relevance to the query by modelling documents and the query as a vector of features. The features are typically *terms* in this case and the similarity value between a query and a document encodes the relevance of a document given a query. The similarity between q and d is given by the *cosine similarity* between the document and query vector cast in the feature space of terms.

Query Semantics There are two widely accepted query semantics – *conjunctive* and *disjunctive* query semantics. Conjunctive query semantics require all terms to be present in a document to qualify as a relevant result whereas disjunctive semantics allow any occurrence of the query terms in the document. For example, under conjunctive query semantics, the relevant documents for the keyword query “german soccer team” using boolean retrieval should contain all the constituent terms “german”, “soccer” and “team” to constitute as results. Although simple to implement, Boolean-retrieval models are limiting when a large number of documents qualify as results. It is hard for the user to evaluate all results and a notion of ranking is hence desirable.

We now come to the question of assigning feature values for vectors. To address this there are two aspects which are considered. The more a query term occurs in a document the more important it is. This is captured by the term frequency or $\mathtt{tf}(d, v)$ of a term v for a document d, which is defined as the number of times v is present in d. Note that in this model, documents are considered as *bags* of terms. However, certain terms frequently occur throughout the corpus, for instance articles, prepositions and stop words. These terms, although highly frequent, might not be *discriminative* from other query terms when it comes to capturing the true user intent. For example in the query "house of fraser", "fraser" is more discriminative than the other terms which appear frequently in the collection. To account for this, the second aspect or document frequency $\mathtt{df}(v)$ of a term v is taken into consideration. Using the value $\mathtt{df}$, the inverse document frequency is defined as

$$\mathtt{idf}(v) = \log \frac{|\mathcal{D}|}{\mathtt{df}(v)}$$

is construed as being directly correlated with the discriminativeness of a term.

Combining both the measures we arrive at the $\mathtt{tf}$-$\mathtt{idf}$ which has become one of the most central weighting scheme for the vector space model. Most of the other weighting schemes are often variations and refinements of $\mathtt{tf}$-$\mathtt{idf}$. A detailed account of these weighting schemes can be found in Manning et al. [MRS08]. The most popular refinement of the $\mathtt{tf}$-$\mathtt{idf}$ value is the Okapi BM25 model by Robertson et al. which is known to provide high retrieval effectiveness. Okapi BM25 takes into account the length of the document and also the average length of documents in the entire collection to normalize the term frequencies. Formally,

Definition 2.1 (Okapi BM25) *For a document* d *and a query* q*, the relevance of* d *to* q *is defined by Okapi BM25 as*

$$r(q, d) = \sum_{v \in q} w_{tf}(v, d).w_{idf}(v). \tag{2.1}$$

The $\mathtt{tf}$*-score* $w_{tf}(v, d)$ *is defined as*

$$w_{tf}(q, d) = \frac{(k_1 + 1) \cdot tf(q, d)}{k_1 \cdot ((1 - b) + b \cdot \frac{dl(d)}{avdl}) + tf(q, d)}\ , \tag{2.2}$$

where $0 \leq b \leq 1$ *and* $k_1 \geq 1$ *are tunable parameters. The length of* d *is denoted as* $|d|$ *and* $avdl$ *represents the average document length in the document collection.*

The $\mathtt{idf}$*-score is defined as*

$$w_{idf}(v) = \log \frac{N - df(v) + 0.5}{df(v) + 0.5}\ , \tag{2.3}$$

where N *is the collection size and* $\mathtt{df}(v)$ *has the aforementioned semantics.*

The $\mathtt{tf}$ component in Okapi BM25 is scaled by adjusting the parameter k_1 while b controls the effect of length normalization. Typical values for the two parameters which have been seen to work well are $k_1 = 1.2$ and $b = 0.75$.

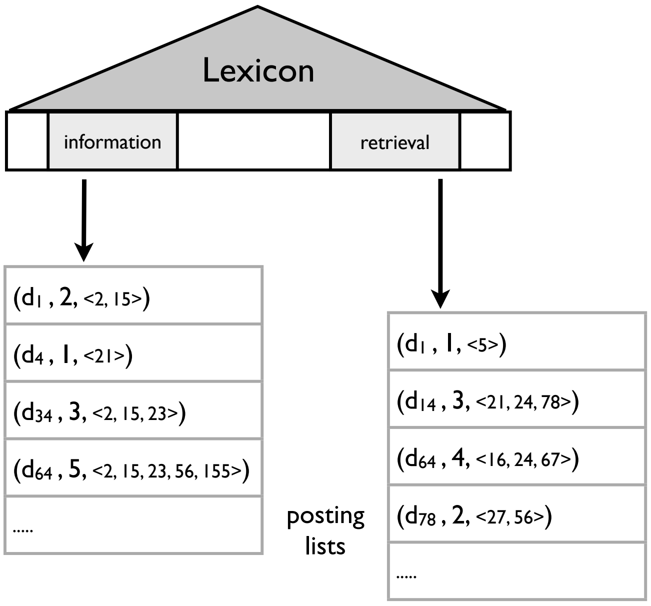


Figure 2.2.: Inverted index

The success of a retrieval system not only depends on the effectiveness of the retrieval models but also on how fast it can respond to queries. With the exponential growth of collection sizes the challenges for scalable and efficient search have accounted for a rich body of research over the past decade. The efficiency issues can be broadly classified into two major areas in the retrieval phases – (i) indexing the document collection and (ii) processing queries over the index. In the following, we introduce the common techniques employed for indexing text.

2.3.2. Indexing Text

Indexing methods are required in order to efficiently perform the retrieval process. The most popular choice to index text collections is the *inverted index*. It is the cornerstone for most commercial search engines and real-world information retrieval systems. The inverted index comprises of two components - (i) the *lexicon* and the (ii) collection of *posting lists* comprising the *inverted file*. Each of them can be implemented using different data structures and are materialized into separate files.

Lexicon The lexicon or the term dictionary contains the indexed terms and their corresponding statistics. These terms are typically words which are extracted and stemmed from individual documents. Each entry in the lexicon represents a term v and it must contain, but is not limited to, the following – (i) the string of the term v, (ii) document-frequency information $\mathtt{df}(\nu)$ and, (iii) pointer to the posting list of ν in the postings file.

During query processing the lexicon is consulted to check for the containment of the query terms in the collection. The look-up process results in providing further data, if needed, for the rest of the retrieval process. There are many strategies to improve lexicon look-up speeds. One straightforward approach, which is now common with the growth of memory capacities, is to load the entire lexicon into memory as a hash table. Other examples of dictionary layout include search trees and *dictionary-as-a-string* proposed by Witten et. al [WMB99]. For a detailed discussion we point the reader to Manning et al. [MRS08].

Postings File The second data structure is the postings file which contains information about occurrence of terms in documents. Since terms are the indexed units here, each term ν is associated with a list of postings L_ν, called the *posting list*, and each posting represents a document where ν is present. The postings file is a collection of posting lists, one for each term. A posting belonging to the posting list L_ν, contains the document identifier d_i where ν is present and information about the ν's occurrences. The occurrence information can be binary, if ν is present in the document or not, or more expressive, like the frequency of occurrences or even positions where ν was present in the document, i.e.,

$$\langle d_i,\ \mathtt{tf}(\nu, d_i),\ < pos_1, \ldots, pos_{\mathtt{tf}(\nu,d_i)} > \rangle.$$

For Boolean retrieval storing only document identifiers is sufficient. In case of ranked retrieval, which is indeed the more common scenario, a score s is stored which is usually the $\mathtt{tf}(\nu, d_i)$ or $\mathtt{tf}(\nu, d_i).\mathtt{idf}(\nu)$. If phrase or proximity queries need to be supported the posting also stores a list of positions where the term occurs in the document (see Figure 2.2). These positions are sorted for best compression and query processing benefits as we will see later. Posting lists are highly compressed and stored contiguously on disk for best space and cache effectiveness. However, for dynamic collections the contiguous placement might not always be desirable and we discuss later how such collections can be indexed. Additionally these lists can be augmented with *skip pointers* for faster list traversal during query processing [MZ96]. For the scope of this work we do not consider such optimizations, since they are orthogonal to the methods proposed.

The posting list can be *document-ordered* or *score-ordered*. In document-ordering the postings are ordered by increasing document identifiers. The benefit of such an organization is that it achieves high compression ratios. Additionally, document-ordered lists are easier to maintain than their score ordered counterparts. On the other hand score-ordered lists are ordered according to their scores. Score-ordered lists facilitate dynamic pruning of lists during query processing and do not require the entire list to be read into memory. Several top-k algorithms have been proposed which allow for early termination by leveraging the score order during query processing [ADGK07, FLN01, BMS+06, TWS04, YSL+12]. However, the compression factor achieved by such an ordering is not as high as document-ordered lists and index maintainance is also expensive.

Compression

The posting list accounts for the majority of the index size. Thus posting-list compression is a key problem in indexing text and has been subject to active research over the past two decades. The obvious advantage of list compression is their savings in overall index size. Since inverted indexes are typically stored on disk, compression can reduce the space consumption by 75% [MRS08]. Also, due to the use of caching for query processing, some of the frequently accessed index lists are stored in memory to increase query performance. Since, memory is a more expensive resource than disks, a smaller memory footprint is desirable. Finally, CPU performance has improved relatively more than hard-disk performance in recent years. Thus the aggregate benefit of transferring a highly compressed posting list to memory and decompressing it outperforms reading it in its uncompressed form. We outline the most widely used techniques for index compression.

Integer Compression We first talk about integer-compression techniques useful in compressing document identifiers and positional information. Storing integers in fixed-size bit sequences or arrays wastes space for small integers. *Variable byte encoding* uses integral number of bytes to encode an integer. An example is the *7-bit encoding* in which the most-significant bit of a byte is the *continuation bit* and the remaining seven bits represent the payload. The 7-bit encoding can be decoded by reading a sequence of bytes until the continuation bit is zero. The sequence read concatenates the payloads of each byte, disregarding the continuous bit values of each byte, to decode the integer. As an example, $n = 129$ is encoded as $\langle\, 00000001 \quad 11111111 \,\rangle$ and $n = 30$ is $\langle\, 00011110 \,\rangle$.

Unlike variable byte encoding, which is byte aligned, we can consider directly working with bits. The simplest bit-level encoding technique *unary encoding* stores $n - 1$ one bits followed by a zero as a delimiter to store the positive integer n. Although, it is beneficial for small n it soon becomes infeasible for higher values of n. For more efficient representations of integers *Elias-γ encoding* is used which splits the positive integer n into two parts. The first part encodes $1 + \lfloor \log n \rfloor$ in unary encoding. The second part encodes $n - 2^{\lfloor \log n \rfloor}$ using binary encoding. As an example $n = 10$ would be encoded as $\langle\, 1110 \quad 10 \,\rangle$.

Elias-δ encoding is slightly different from *Elias-γ encoding* in which the first part, $1 +$

$\lfloor \log n \rfloor$, is encoded using *Elias-γ encoding* instead of unary encoding. Thus, $n = 10$ would now be encoded as $\langle\, 110 \;\; 00 \;\; 10 \,\rangle$.

Compressing d-gaps Let us for simplicity view a posting list as an array of document identifiers. The obvious choice in compressing a posting list is to use one of the above techniques to compress each document identifier did. In case of document-ordered lists, we can exploit the ascending order of dids by encoding the difference between consecutive did values or so called *d-gaps*. In this scheme, the original integer value of only the first or minimum integer is stored. The remaining integers are represented by their d-gap value. All integers can be reconstructed in the course of the linear scan from begin of the list from the d-gaps. For example $\langle\, d_1, d_2, d_2, d_4 \,\rangle$ can be represented as $\langle\, d_1 \,, d_2 - d_1 \,, d_3 - d_2 \,, d_4 - d_3 \,\rangle$.

The primary advantage of storing d-gaps is that they tend to be small integers and can be compactly represented using the techniques discussed above. The gains are significant especially for long posting lists, corresponding to frequent terms such as articles or preposition, where the difference between dids is very small. Apart from using d-gaps for dids it can also be employed for positional information. A more comprehensive survey of other compression methods for d-gaps can be found in [WMB99].

Query Processing

After having discussed retrieval models and indexing we now turn to query processing: *Given a query,* q*, and an inverted index how can we process queries efficiently ?* Depending on the index organization there are two major query processing techniques. We discuss query processing techniques for document-ordered lists - *Term-at-a-Time* and *Document-at-a-Time*.

In *Term-at-a-time* (TAAT), the posting list L_v for each query term $v \in q$ is processed one after the other. A set of accumulators, one for each document, is maintained which store partial scores for the results determined thus far. In each iteration, the partial scores in the accumulators are updated. Finally, after all the terms have been processed the scores in the accumulators are sorted in descending order.

In order to reduce the number of operations and accumulators, numerous methods and heuristics have been proposed. For conjunctive query semantics, the retrieval of the posting list is scheduled in increasing order of their lengths. Since the result set is always a subset of every posting list such a scheduling facilitates reduction in number of accumulators initialized. Dynamic pruning strategies have also been proposed for memory-limited conditions, a detailed account of which is presented in [BYGJ+08].

Document-at-a-Time (DAAT) processing accesses all the posting lists in parallel. In this strategy the final scores of result documents are not incrementally computed, as in TAAT using accumulators, but determined on the fly. The posting lists are merged efficiently using a priority queue of postings. A cursor is kept for each posting list L_v corresponding to each term in the query, $v \in q$, and a min-heap of document identifiers is maintained. In each iteration, the cursors are advanced to the minimum document identifier, say d_{min}, in the heap. Next, the final score for d_{min} is computed and the priority queue is populated with the minimum document identifier from the current cursor pointers. For processing phrase queries, an additional check is done utilizing the positional information in the postings if the occurrence of the v in d_{min} is a part of the query phrase. For example, consider a query "information retrieval" on an inverted index represented in Figure 2.2. Although document d_1 contains both query terms, "information" at positions 2, 15 and "retrieval" at position 5, but they are not present next to each other. Hence d_1 does not qualify as a result to the phrase query. On the other hand d_{64} contains them one after another starting at positions 23.

Depending on the query semantics, further optimizations can be performed. For instance when conjunctive query semantics are required, a max-heap instead of a min-heap is maintained to avoid scoring of documents not containing all query terms.

Index Maintenance

The indexing methods discussed so far concern static document collections. However, document collections change over time. New documents are inserted and old documents are either deleted or modified. These changes should appropriately be reflected in the index to keep it in sync with the document collection. In this section, we discuss *index maintenance strategies* which have been proposed to address this problem. The index can be updated either in a *batched* manner or *incrementally*.

Batched Index Updates In batched updates, the index maintenance is carried out periodically while accumulating the changes to the document collections in the meantime. When the index is finally updated, these changes are incorporated into building a fresh index which is consistent with the collection. A straightforward strategy to accomplish this is to *re-build* the entire index from scratch on the current state of the document collection. Although easier to implement, such a strategy works well only if more than 60% of the changes involve deletions and updates [BCC10]. When insertions dominate, which is the case in most situations, re-building the index from scratch often becomes expensive. The other, more practical alternative is to build a partial index on the accumulated updates and merge it with the existing index. This update strategy is referred to as *re-merging*. Re-merging avoids the inversion of the older documents into postings and creates posting list by merging the posting lists, for each term, of the partial and existing index. Deletions are managed by keeping a list of documents which are no longer present in the collection. This list is used as a filter during query processing to remove non-existing documents from the result set.

Incremental Index Updates In many applications it is critical that the index always reflects the current state of the collection. Batched updating of indexes is undesirable for such requirements. Most of the literature which deals with incremental updates assumes a strict *incremental* nature of the updates, i.e., only new documents can be added and existing documents can never be modified or removed.

The underlying principle is to create multiple in-memory partial indexes, similar to the partial indexes in re-merging, and allow query processing on them. The partial indexes are of fixed sizes depending on memory limitations. Once a partial index exhausts the space budget, it is materialized to disk and a new index is started. Over a period of time and numerous updates we have multiple small disk-resident indexes. When queries are issued, they are routed to all the partial indexes. For a query q, the posting-list fragments for each term $v \in q$ are fetched from say m partial indexes. All m fragments, including the in-memory index, are concatenated to form the term's posting list. Following this, the corresponding query processing method is invoked for computing the final result set.

Partitioning the entire index into such smaller partial indexes might be desirable for indexing but is clearly expensive for query performance. In this scenario, the number of random seeks for fetching the fragments are $|q|.m$. The query performance can be improved by selectively merging these partial indexes into a smaller set of partial indexes. Taken to the extreme, the performance is at its best when there is one consolidated index thereby eliminating fragmented lists. In such a strategy, called *immediate merge*, only a single disk-resident index is maintained apart from the in-memory partial index. Once the in-memory index reaches the size threshold it is merged into the main index.

Many factors determine the efficiency of the merge operation like – assignment of document identifiers, choice of compression algorithm and allowing for in-place merging. Typically, in a dynamic collection, document identifiers are assigned incrementally, i.e., every new document is assigned an identifier greater than previously assigned identifiers. Consequently the partial indexes always have document identifiers greater than those indexed in the primary index. Thus the merging operation of the two posting lists reduces to an append operation. Indexes constructed with a different document-identifier assignment, which does not exhibit the above property, can be difficult to maintain due to expensive de-serialization and sorting operations.

The second factor which affects merge efficiency is the choice of the compression method. As discussed in the section earlier *d-gaps* are computed and compressed using integer-compression methods. If we employ compression algorithms which are independent of global parameters, like collection-wide statistics, one does not have to de-compress the already compact posting list from the primary index. For example, the merging of posting list of term v using 7-bit encoding of *d-gaps* would require us to read L_v from the primary index and determine the last document identifier in the list. The last document identifier, say d_{last}, can be identified on the fly or can be explicitly stored. The posting list from the partial index is compressed and appended to L_v by computing d-gaps from d_{last}.

In-Place Merging After the updated posting list, say L'_v is determined, the new copy has to be written to a new location. However, *in-place* merge techniques have been proposed which allow for over-allocation of space after each posting list. This allows L_v to be updated in-place and avoids expensive relocations. There are two kinds of in-place merge strategies. One requires the *entire posting list* to be contiguous, as proposed by Lester et. al [LZW06], while the other does not [TGMS94]. Contiguity of posting lists improves query performance but sometimes involves expensive relocations. On the other hand, allowing for non-contiguous lists improves update performance at the expense of query performance. To reconcile these two approaches hybrid methods have also been proposed in [BCL06a] which take into account the posting-list length and update characteristics for over-allocation.

We have discussed two techniques which are on the extremes of the trade-off between index maintenance and query performance, i.e., the no-merge strategy which has no maintenance overhead and immediate merging which has the best query performance. There are other approaches which explore this trade-off to selectively merge partial indexes to balance maintenance and query performance namely *logarithmic merge*, *geometric merge*, etc. A detailed account of these can be found in [BCC10].

2.3.3. Indexing Archives

In this section we look at text search over versioned text collection with focus on web archives. Before we discuss the details of the index organization we outline the document, collection and query model and formally define the *time-travel retrieval task*.

Document and Collection Model We adopt a discrete notion of time and assume that a time-stamp t is a positive integer and is computed periodically, with a fixed granularity, from a reference point in the past where $t = 0$. For instance, a widely accepted time of origin in computer systems is the *Unix Epoch* – 00:00:00 UTC on 1 January 1970. The granularity of measurement could be milliseconds, hours, days, weeks or years.

We define the overlap of time-intervals as follows.

Definition 2.2 (Overlapping intervals) *An interval* $I_1 = [b_1, e_1)$ *is said to overlap with another interval* $I_2 = [b_2, e_2)$*, denoted as* $I_1 \sqcap I_2$*, if the following relation holds:*

$$I_1 \sqcap I_2 \iff b_1 < e_2 \ \wedge \ b_2 < e_1$$

and I_1 *does not overlap with* I_2*, denoted as* $I_1 \not\sqcap I_2$*, if the following holds:*

$$I_1 \not\sqcap I_2 \iff b_1 \geq e_2 \ \vee \ b_2 \geq e_1.$$

We operate on a versioned document collection $\mathcal{D}$. Each document $d_i \in \mathcal{D}$ is associated with a sequence of its versions $\langle d_i^1, d_i^2, \ldots \rangle$. Each version d_i^k is drawn from a vocabulary of terms $\mathcal{V}$, i.e. $d_i^k \subseteq \mathcal{V}$. Furthermore, each document version d_i^k has an associated valid-time interval $valid(d_i^k) = [begin(d_i^k)\,,\ end(d_i^k))$ when d_i^k existed in the real world. We also make a natural assumption that versions of the *same* document do not have overlapping valid-time intervals, i.e.,

$$\forall d_i^j, d_i^k : \ valid(d_i^j) \not\sqcap valid(d_i^k).$$

A document version is active, or has not yet been superseded by a new version if $end(d_i^j) = \infty$. Note that this is similar to the notion of *transaction time* in temporal databases as introduced earlier.

Query Model Berberich et al. in 2007 [BBNW07] introduced the *time-travel* queries as an important query type to search archives and other time-stamped corpora. Formally, a time-travel query Q, as considered in this work, consists of a set of terms $keywords(Q) = \{q_1, \ldots, q_m\}$ and a time interval denoted by $interval(Q)$. The begin and end of the query time-interval are represented by $begin(Q)$ and $end(Q)$, i.e.,

$$interval(Q) = [begin(Q)\,,\ end(Q)].$$

As an example, for a time-travel query Q = "bhopal gas tragedy" @ `[11/1984 – 01/1985]` – $keywords(Q) = \{bhopal,\ gas,\ tragedy\}$, $begin(Q) = 11/1984$ and $end(Q) = 01/1985$. With the model in place we can now formally define the time-travel retrieval task.

Definition 2.3 (Time-travel Retrieval Task) *Given a document collection* $\mathcal{D}$ *with versioned documents (each document version associated with a time-interval), and a time-travel query* Q*, the objective is to find all documents versions* $R(Q)$ *whose valid-time interval overlaps with the query time-interval and which contains all the query keywords.*

$$R(\,Q\,) = \\ \quad \left\{\, d_i^k \in \mathcal{D} \mid \forall q \in \mathit{keywords}(Q)\ :\ q \in d_i^k \wedge \mathit{valid}(d_i^k) \mathbin{⧉} \mathit{interval}(Q) \,\right\}\ .$$

Queries for which $\mathit{begin}(Q) = \mathit{end}(Q)$ holds, so that the query time-interval collapses into a single time point, will be referred to as *time-point queries*.

Time-Travel Index

Berberich et al. proposed the time-travel inverted index as an adaptation of the inverted index for efficient processing of time-travel queries. It transparently extends the standard inverted index and proposes novel compression and index partitioning schemes. We start by looking at the extensions with respect to the postings and posting-list organization. Firstly, the posting representing a version d_i^k is extended by addition of a valid-time interval $\mathit{valid}(d_i^k) = [\mathit{begin}(d_i^k)\,,\ \mathit{end}(d_i^k))$, i.e.,

$$\langle d_i^k, [\mathit{begin}(d_i^k), \mathit{end}(d_i^k)), s\rangle.$$

Secondly, it was proposed that each posting list L_v may be be partitioned along the time dimension into a set of partitions. Each partition has an associated time-interval $\mathit{span}(\phi_{v,j}) = [\mathit{begin}(\phi_{v,j})\,,\ \mathit{end}(\phi_{v,j}))$. It contains postings which are present in the unpartitioned list L_v, $\phi_{v,j} \subseteq L_v$, and those which represent document versions whose valid-time interval overlaps with $\mathit{span}(\phi_{v,j})$, i.e.,

$$\phi_{v,j} = \left\{\ d_i^k \in \mathcal{D} \mid \mathit{valid}(d_i^k) \mathbin{⧉} \mathit{span}(\phi_{v,j})\ \right\}\ .$$

Further, we assume that partition-spans for a given term v are disjoint, i.e.,

$$\forall i\ \forall j\ :\ \mathit{span}(\phi_{v,i})\ \not\mathbin{⧉}\ \mathit{span}(\phi_{v,j})\ .$$

For processing a time-travel query, we must first retrieve for each query term, the set of postings which overlap with the query-time interval. Once retrieved, standard query processing techniques discussed before can be applied on this subset of postings. To reduce the retrieval cost, only those partitions which overlap with the query-time interval are fetched, i.e., for a retrieved partition $\phi_{v,j}$ with respect to a time-travel query Q the following holds

$$\mathit{interval}(Q) \mathbin{⧉} \mathit{span}(\phi_{v,j}).$$

Following that, all partitions for the same term are merged and irrelevant postings, whose valid-time interval do not overlap with the query-time interval, are filtered out

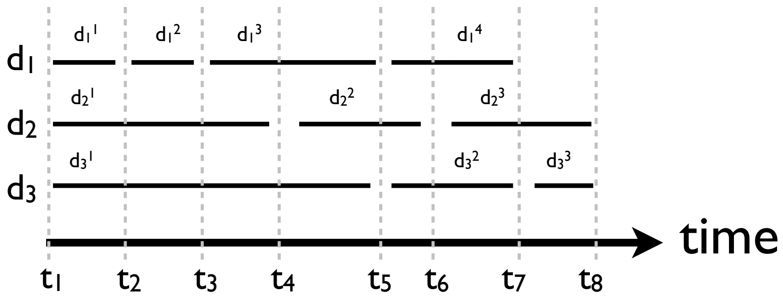


Figure 2.3.: Partitioning a posting list

in the process. Note that identifying if a postings is irrelevant in a partition is only possible once the partition is retrieved. If most of the postings in a partition are irrelevant then an entire partition must be wastefully read to determine a few relevant postings. An approach to reduce the cost of filtering out such postings, thus improving query performance, the granularity of partitioning can be reduced. As one extreme, called the *performance-optimal* approach or P_{opt}, partition boundaries are placed at time-points that occur as boundaries of valid-time intervals as shown in the Figure 2.3. Such a partitioning eliminates irrelevant postings in a partition.

On the other hand, since each partition is associated with a time span, postings whose valid-time intervals overlap with multiple partitions are replicated in all of them. For instance, in Figure 2.3, the version d_3^1 is replicated multiple times for the performance-optimal approach. Replication of postings in multiple partitions results in an index blowup thus making the performance-optimal approach infeasible in practice. The other extreme case, referred to as *space-optimal* partitioning or S_{opt}, ensures that there is no index blowup by disallowing replication of postings.

Partitioning Strategies There is a natural trade-off between query performance and index-size blow and the above two approaches represent the two extremes of this. In practical situations the desired *partitioning strategy* lies between these two strategies. To explore this middle ground two approaches of partitioning strategies are proposed - the *performance guarantee* approach and the *space bound* approach.

In the performance guarantee approach a bounded loss of performance over P_{opt} is tolerated. The reduction of performance, by the performance-guarantee approach, for a given query is measured by the excess of postings processed over P_{opt}. The performance reduction is bounded by a user specified parameter γ and a partitioning is desired which adheres to this bound for all possible queries. The objective, under such a performance guarantee, is to minimize the overall index-size blowup.

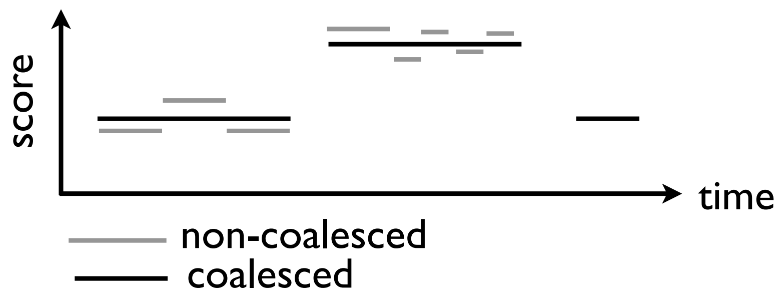


Figure 2.4.: Temporal coalescing

The space-bound approach is proposed for situations when storage space is at a premium and a bound is desired on the overall index size. Given a user specified index-size bound, κ, the objective here is to minimize the reduction of performance relative to P_{opt}. The size bound on the entire index can be translated to a bound on the size of each posting list. The size bound for partitioning posting list for each term v is given by $\kappa.|L_v|$. The space-bound approach also allows for a query distribution determining the probability of a term given a time-point $P(t)$. An optimization problem is formulated which intends to partition a posting list L_v to minimize the expected query processing cost, given κ and $P(t)$, while adhering to a space bound $\kappa.|L_v|$. We refer to these partitioning strategies as *vertical partitioning* in the rest of the thesis.

Temporal Coalescing Indexing all the versions is expensive in terms of storage costs. However, archives are characterized by a high degree of redundancy. Often times, changes to documents are minor resulting in a high overlap of text content between consecutive documents versions. Temporal coalescing techniques exploits this redundancy to substantially decrease the overall index size.

A naïve implementation of the posting list L_v would create a posting for each document version term pair. To exploit the redundancy between consecutive versions temporal coalescing in conjugation with the time-travel index. Temporal coalescing methods aim to coalesce postings belonging to consecutive versions of the same documents when the information contained between them is not significantly different. The techniques are applied on a per-term basis. They also have different semantics for different types of payloads, i.e., boolean, scalar or positional.

For boolean payloads of the form $\langle d_i^k, \mathtt{valid}(d_i^k)\rangle$, multiple postings for consecutively occurring versions can be replaced by a single posting. The single posting represents that the term was present in a document for a contiguous period of time encoded in the posting.

For scalar payloads, scores of consecutive versions with low variance can be captured and coalesced into a single posting. Figure 2.4 illustrates that coalescing postings with small variation in scores results in reducing the number of postings, in this example, from 9 to 5. The degree of relaxation allowed is specified by the user and captured by the parameter η. This is formulated as an optimization problem. Given an input sequence of postings of the same document, ordered by their begin times, the objective is to generate a minimal number of coalesced postings adhering to the user specified relaxation.

3 Efficient Indexing and Maintenance for Time-Travel Text Search

3.1. Motivation and Problem Statement

In this chapter we address two major indexing challenges. Firstly, given a versioned document collection we address how to efficiently identify documents which are solutions to the *time-travel retrieval* task. To this end we propose novel index organization and tuning approaches. Secondly, given a dynamically growing versioned collection we propose strategies for efficient index maintenance.

Archives are collected over a period of time and hence have an implicit temporal dimension. Consequently, time-travel text search is important in searching archives as it limits the search to a specified time of interest in the past. We motivate this by two use cases. A political analyst is interested in statements made by presidential candidates during the recent financial crisis (in 2008). Constraining the search to document versions that existed in the year 2008 filters out versions during previous instances of such events thus preventing dilution of content.

Consider a business analyst looking for web pages predicting and analyzing upcoming releases of tablet computers. If only keyword queries are used to retrieve pages from a web archive, irrelevant information about earlier releases of tablet computers may corrupt the analytics results. By including a temporal constraint on the publication or discovery time of web pages, such undesirable results can be eliminated.

Archives are dynamically increasing collections with content being added over time. Almeida et al. [AMC07] report how Wikipedia has grown exponentially since its inception. News archives grow continuously and are progressively becoming accessible online. Research and improvements in archive crawling technology [Den12, HER13] also suggest that more content will be amassed in the future. To keep up with the dynamically growing archives it is essential that our indexes have to be efficiently maintain-able. Naïve approaches like periodically rebuilding indexes are prohibitively expensive. Hence there is a need for strategies for index organizations which can be updated easily while still being competitive in query performance.

3.1.1. Approach

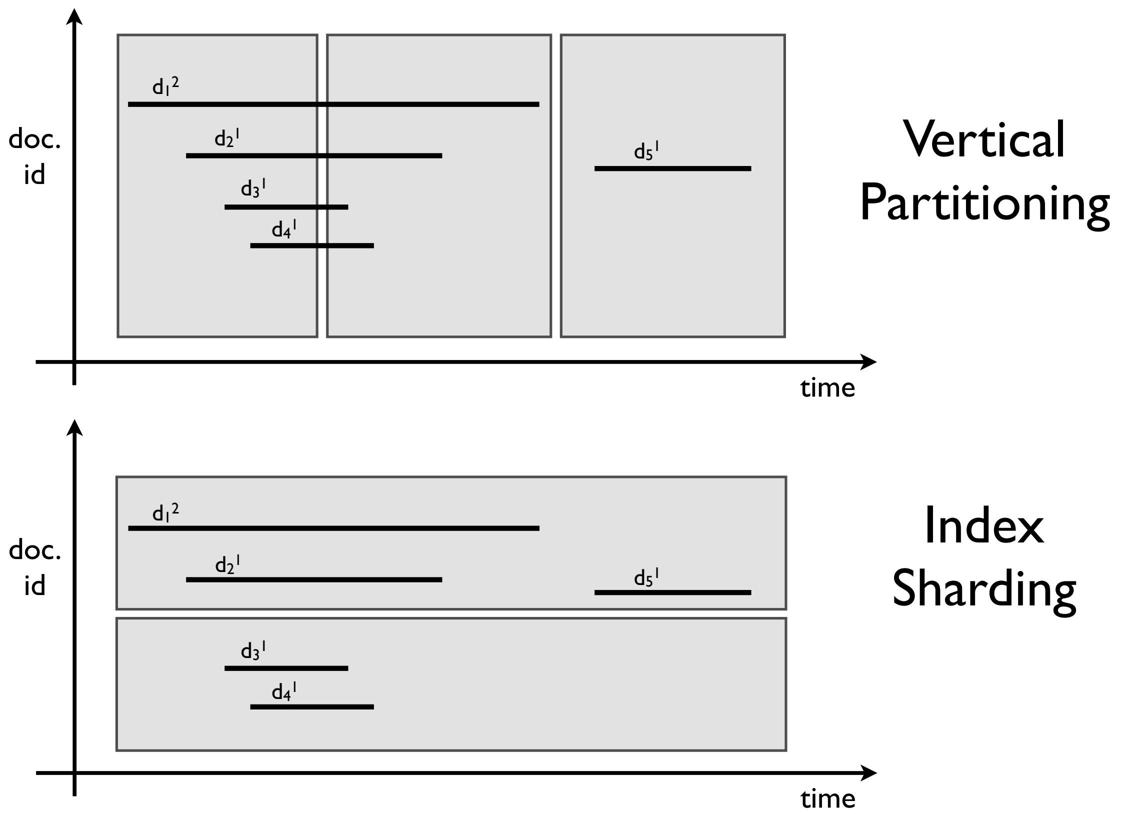


Figure 3.1.: Vertical partitioning vs. sharding of a posting list

The state-of-art approaches to process time-travel queries, as discussed in Chapter 2, transparently extend the inverted index with time-enriched postings. To optimize for time-travel query processing each posting list is partitioned into a number of smaller posting lists along the time dimension. We refer to these methods as *vertical partitioning*. Partitioning in this way reduces access to postings which are irrelevant to the query. Specifically, those postings which do not overlap with the query time-interval. Increasing the degree of vertical partitioning improves the query-processing performance. However, such an index organization suffers from an index-size blowup, incurred due to the replication of postings, due to partitioning. A careful choice of the partitioning boundaries can help to reduce the index-size blowup [BBNW07], but in order to achieve acceptable levels of efficiency 2 to 3 times index size increase is necessary.

We look at a novel way of partitioning posting lists called *index sharding* in which we propose to *shard* – or horizontally partition – *each posting list* along document identifiers, instead of time (see Figure 3.1). An immediate benefit of this alternative partitioning is almost *no increase* in the overall index-size (as we show later the only overhead is to maintain a small set of location pointers in each partition). We develop a single-pass greedy algorithm that optimally shards the posting list, minimizing the number of postings read during query processing.

The idea is to achieve superior query performance by organizing the postings into shards, exploiting the geometry of the associated intervals, which helps in avoiding access to postings which are irrelevant to the query time-interval. Query processing over a sharded index proceeds by accessing all the shards in parallel but reading only a small portion from each of them. As a random access is at least as expensive as a sequential read (as in disk-based and network-based index storage), breaking the posting list into too many shards actually degrades performance, even if we optimize access to only relevant postings. Thus, the practical efficiency of the index organization is achieved only if it is sensitive to the cost ratio of random accesses to sequential accesses. We formulate optimization problems for tuning the parameter for query performance that takes into account the I/O cost ratio of the storage infrastructure and propose a heuristic to combine shards to gain practical runtime efficiency.

Based on the index sharding principle, we propose a framework which efficiently handles additions of new document versions to the archive without sacrificing query efficiency. We propose an algorithm which is incremental in nature thus avoiding the expensive recomputation of shards. Further, it reconciles the relative cost of random and sequential access for better query performance.

3.1.2. Contributions

In summary, the key contributions made in this chapter are the following.

1. A novel sharded index organization for a time-enriched inverted index that overcomes the issue of index-size blowup.

2. An optimal greedy algorithm to shard the posting list, so that no time-travel query reads more than the required postings, thus achieving ideal query-processing performance.

3. A framework that achieves practical runtime efficiency by tuning the number of shards that each posting list is split into, taking into account the I/O cost ratio of the storage infrastructure. Note that the cost of such a partitioning is *independent*

of the dynamics in the query workload, as it depends only on the storage system parameters.

4. A framework to support updates and perform efficient index maintenance based on an incremental sharding algorithm.

5. An extensive empirical evaluation on large-scale versioned document collections using real-world keyword queries with temporal constraints at varying granularities.

3.1.3. Organization

The remainder of this chapter is organized as follows: In Section 3.2, we present the data model and index organization that we use in this chapter. Next, in Section 3.3, we describe in detail the idea of sharding the inverted index and how queries can be processed over the sharded index. In Section 3.4 and in Section 3.5 we present the details of our idealized sharding and a shard-merging strategy. Section 3.7 describes our incremental sharding method for index maintenance. Section 3.8 presents the overall architecture of our update-aware retrieval system. Details of our experimental setup and its results are presented in Section 3.10. Finally, we discuss previous related work in Section 3.11, before summarizing in Section 3.12.

3.2. Model and Index Organization

We adopt the document, collection, and query models from Chapter 2 and briefly recap the notation in Table 3.1.

We call a document version d_i^k an *active version* if it is the most current version of document d_i and consequently has $end(d_i^k) = \infty$ where the current time or "now" is represented as ∞.

Our proposed index for time-travel queries is based on the established *inverted index* as described in Chapter 2. We extend the inverted index to support time-travel queries by modifications to the postings structure, posting-list organization, and the lexicon.

We extend the contents of each posting by additionally storing the valid-time interval, $valid(d_i^k) = [begin(d_i^k), end(d_i^k))$, of the document version d_i^k along with the identifier d_i, and a payload s, i.e.,

$$\langle d_i^k, [begin(d_i^k), end(d_i^k)], s\rangle.$$

The index organization supports different retrieval models by allowing the payloads to be empty (for Boolean retrieval), containing a scalar value (tf in the document version) or containing positional information (phrase-based retrieval). When no confusion

Notation	*Description*
$\mathcal{D}$	collection of documents
$\mathcal{V}$	*vocabulary*, a set of words
$d_i^j \in \mathcal{D}$	a document version, j^{th} version of document d_i
$valid(d_i^j)$	valid-time interval
$begin(d_i^j)$	begin time of d_i^j
$end(d_i^j)$	end time of d_i^j
Q	time-travel query
$keywords(Q)$	set of keywords in Q
$interval(Q)$	query time-interval of Q
$I_1 \sqcup\!\sqcap I_2$	Overlapping intervals I_1 and I_2
$I_1 \cancel{\sqcup\!\sqcap} I_2$	Non-overlapping intervals I_1 and I_2

Table 3.1.: Notation

arises, we simply use $begin(p)$ and $end(p)$ of a posting p to refer to the valid-time interval boundaries of the corresponding document version.

We partition each posting list into disjoint partitions referred to as *shards*. The postings in a shard are ordered according to their begin times. We assign the identifiers to documents and versions in the order of their begin times. This ensures that the postings are also ordered by their identifiers in a shard. Subsequently we can use standard compression techniques, commonly used for posting-list compression, for compressing shards.

As a result of index sharding, each term in the vocabulary may be associated with multiple lists. These mappings have to be appropriately reflected in the lexicon. To this end, we extend the lexicon to store pointers to all the shards for each term. Note that since our partitions are not local to a given time interval we do not need to maintain time intervals corresponding to each partition, unlike the lexicon of the vertically-partitioned index.

Impact Lists For each shard we maintain an additional access structure for efficiently determining the postings whose time intervals overlap with a query time-interval. These are called *impact lists*. An impact list is an associative data structure which maintains, for every possible begin time of a query time-interval, the position in the shard of the earliest posting whose valid time overlaps with the query begin time. In other words, the impact list stores pairs of query begin times (key) and offsets (values) from the shard beginning. The overall size of each impact list can be reduced by storing only the distinct offset values rather than offsets for all possible query begin times. An example of

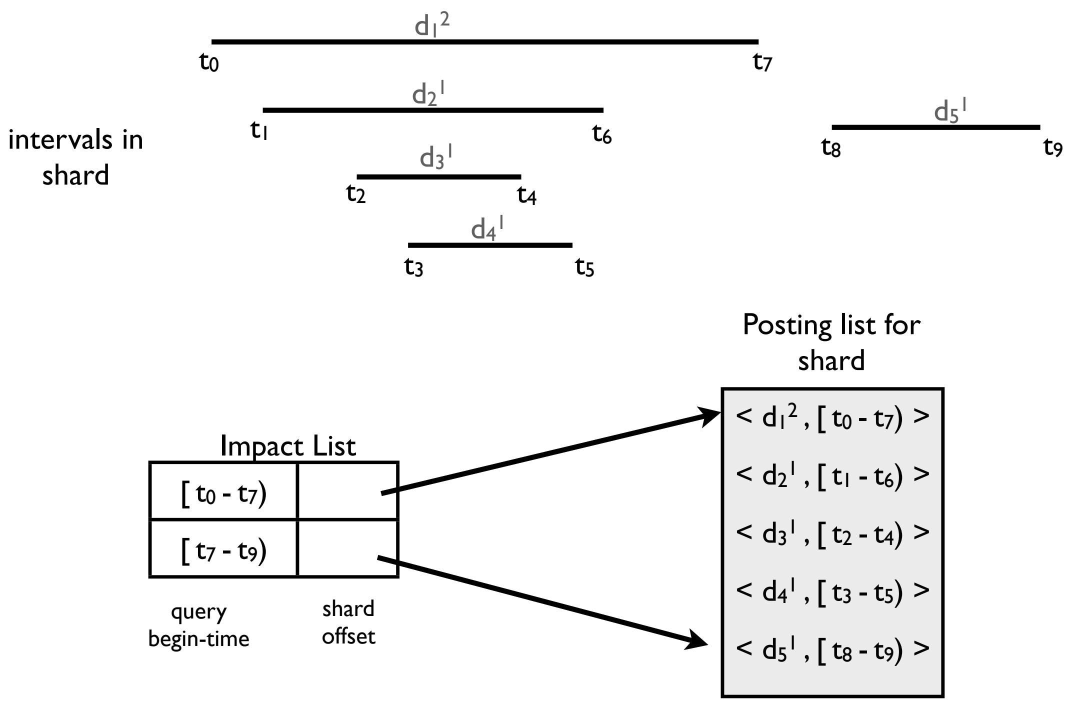


Figure 3.2.: Impact list

an impact list of a shard of five postings is shown in Figure 3.2. The possible begin times are represented as intervals and they map to the offset in the posting list for the shard from where the postings are read sequentially. Thus, for a query interval in $[t_8 - t_{11}]$ we start accessing the list sequentially from the fifth posting. Although represented as intervals, in practice it is sufficient to store the begin or end of the interval. Thus keys admit a non-decreasing integer sequence and a straightforward binary search over the keys efficiently gives the correct offset location. For practical granularities of query begin times such as days, the impact lists for the complete index (i.e., for all shards of all terms) can be easily kept in memory. We discuss how query processing is performed using impact lists in the next section.

3.3. Sharding Posting Lists

Index sharding refers to partitioning or sharding a posting list L_v into disjoint partitions or *shards* such that no two shards share common postings. We now formally define the notion of a shard and posting-list sharding.

Definition 3.1 (Shard) *A shard σ is a sequence of postings, $\sigma = \langle p_i \rangle$, ordered by their begin* *times, i.e.,*

$$\mathtt{begin}(p_i) \leq \mathtt{begin}(p_{i+1}).$$

Definition 3.2 (Posting-List Sharding) *Given a posting list* L_v *for a term* v*, posting-list sharding partitions* L_v *into a set of shards* $\mathcal{S}_v = \{\sigma_1, \ldots, \sigma_m\}$, $\sigma_i \subseteq L_v$ *where*

$$(\forall i, \forall j \; i \neq j \implies \sigma_i \cap \sigma_j = \emptyset) \quad \wedge \quad \left(\bigcup_i \sigma_i = L_v \right).$$

In what follows a sharded index refers to an index which employs posting-list sharding. A sharded index thus maintains multiple shards per term and the disjoint partitioning avoids replication completely (see Figure 3.1).

3.3.1. Query Processing over Sharded Index

For query processing over a sharded index, we employ the established term-at-a-time query-processing model (described in more detail in Chapter 2). In short term-at-a-time processing posting lists are read one after the other and scores of a document version from different lists are merged in memory. For our sharded index, each shard of every query term is processed in a sequence of the following *open-skip-scan* operations.

1. **Open** – Open a shard for a query term. This involves a lookup from the lexicon for the location of beginning of the shard.

2. **Skip** – Given the query begin time, lookup offset position from the impact list of the shard and perform a seek to that position. In practice we can combine the open and skip steps into an *open-skip* operation to avoid an extra I/O operation.

3. **Scan** – Perform sequential reads from this position and terminate when it is certain that the rest of the postings do not overlap with the query time-interval. As we read the postings sequentially in begin-time order, we can safely terminate when the begin time of the next posting exceeds the query end-time.

An illustration of the query-processing operation is shown in Figure 3.3. We are given a sharding $\langle d_1^2, d_2^1, d_5^1 \rangle$ and $\langle d_3^1, d_4^1 \rangle$. Each of the shards is associated with an impact list, as illustrated in the figure. For simplicity we denote the offsets from the shard beginning by the posting which is accessed first.

Consider a time-travel query with time interval $[t_b, t_e]$, shaded in gray, where we have $t_3 < t_b < t_4$. Firstly, by the use of impact lists we try to quickly *skip* to the first overlapping posting in the shard. For instance, in the second shard, by looking up the impact list, we start accessing the shard from the second posting onwards. This avoids

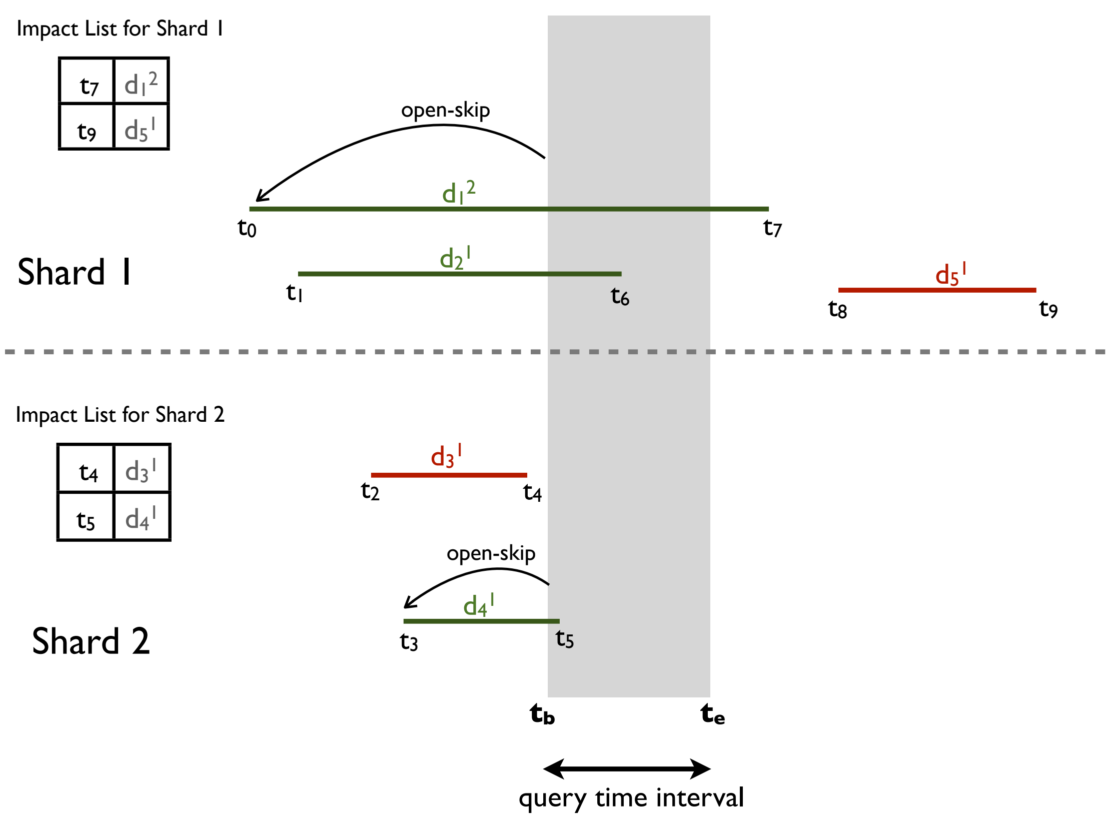


Figure 3.3.: Open-skip and scan operations for query processing

processing of postings at the head of the shard which do not overlap with the query time-interval. Secondly, the postings are accessed sequentially, from the seek position, until a posting p is encountered such that $\mathtt{begin}(p) > t_e$. In the example, we terminate reading Shard 1 after accessing the first two postings and thus avoid reading d_5^1.

In essence, we only access a relatively smaller number of postings of a shard.

For each query, all the participating shards can be processed in parallel. Since the contents postings per shard are disjoint, merging lists is relatively inexpensive. However, if the number of shards per term are large in number, query-processing performance might degrade due to a large number of random seeks (open for each open-seek operation). We now at look at the different posting-list sharding strategies for optimizing query performance.

3.4. Idealized Index Sharding

Although the begin-time order in shards avoids wasteful reading of postings, it does not guarantee the elimination of all wasteful reads. A posting is said to be a *wasted*

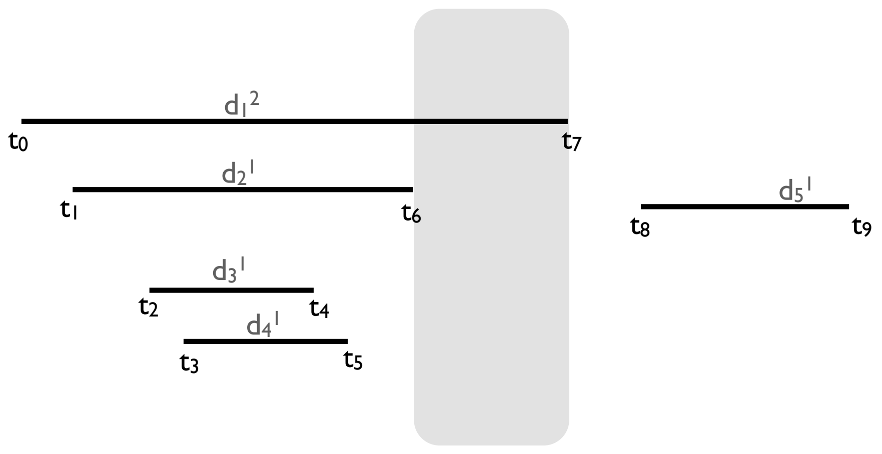


Figure 3.4.: Subsumption of postings

read, if it is accessed during query processing although its valid-time interval does not overlap with the query time-interval. We introduce the concept of *subsumption* in shards to explain why this occurs.

Definition 3.3 (Subsumption of Postings) *For a pair of postings* p *and* q *(from the same posting list),* p *subsumes* q *(for short* $\mathrm{p} \sqsupset \mathrm{q}$*) if*

$$p \sqsupset q \Leftrightarrow (\mathtt{begin}(p) \le \mathtt{begin}(q)) \ \wedge \ (\mathtt{end}(p) > \mathtt{end}(q))\,.$$

A shard is said to exhibit subsumption if it has a pair of postings where one subsumes another. Consider a shard in Figure 3.4 with five postings where $d_1^2 \sqsupset d_2^1$. Now, the queries with $\mathtt{end}(d_2^1) \le \mathtt{begin}(Q) \le \mathtt{end}(d_1^2)$ (highlighted region in the figure) will wastefully read d_2^1, i.e., d_2^1 is processed but does not contribute to the result. Such a scenario, where there are postings like d_1^2 which span long intervals and subsume many postings, can arbitrarily degrade performance. Building on this example, if we further introduce n postings which are subsumed by d_2^1, they are in turn automatically subsumed by d_1^2. Consequently, the number of wasteful reads when the query has a begin time $\mathtt{end}(d_2^1) \le \mathtt{begin}(Q) \le \mathtt{end}(d_1^2)$ will be $n + 1$.

We can avoid any wasteful reads of postings if we can avoid shards with subsumptions of postings. In other words, we require that postings in a shard satisfy the *staircase property*, defined as follows:

Definition 3.4 (Staircase Property) *A shard* σ *is said to have the staircase property if* σ *has exactly one posting or*

$$\forall p, q \in \sigma, \ \ \mathtt{begin}(p) \le \mathtt{begin}(q) \ \Rightarrow \ \mathtt{end}(p) \le \mathtt{end}(q)\ .$$

We let the Boolean predicate $\mathtt{staircase}(\sigma)$ *denote whether* σ *has the staircase property.*

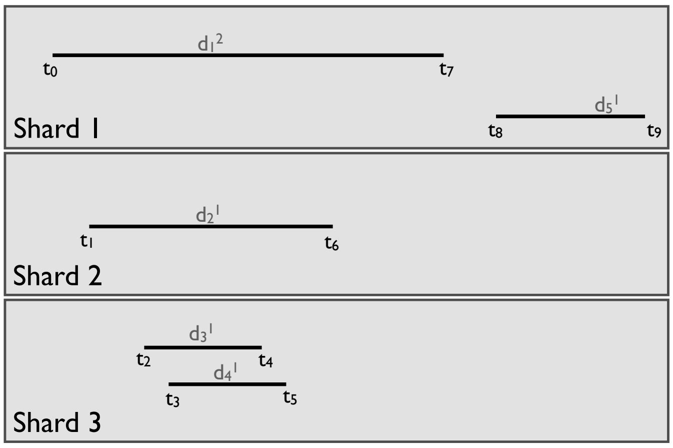


Figure 3.5.: Idealized sharding with staircase property

Clearly, it may be possible to shard a given posting list in many different ways so that the staircase property is satisfied. Since query processing proceeds by open-skip operations for all shards of a term, it is desirable to *minimize* the number of idealized shards. This can be cast into an optimization problem with the input being a list of postings L_v. The objective is to partition L_v into a feasible index sharding $\mathcal{S}$ so as to minimize the overall number of shards where each shard exhibits the staircase property. Formally, the *idealized-sharding problem* is defined as follows:

Definition 3.5 (Idealized-Sharding Problem)

$$\underset{\mathcal{S}}{\mathit{argmin}}\ |\mathcal{S}| \quad \textit{s.t.} \quad \forall \sigma \in \mathcal{S} : \mathit{staircase}(\sigma)\ .$$

where $\mathcal{S}$ is a feasible index sharding of L_v.

3.4.1. Optimal Algorithm for Idealized Sharding

We solve the idealized-sharding problem optimally using the greedy algorithm described in Algorithm 1. We let the last posting added to σ be denoted by $\mathit{last}(\sigma)$. We additionally let each shard σ be associated with an end time $\sigma.\mathit{end}$ representing the end time of $\mathit{last}(\sigma)$, i.e., $\sigma.\mathit{end} = \mathit{end}(\mathit{last}(\sigma))$.

We process all postings of a list L_v in increasing begin-time order. In each iteration, we try to add a posting $L_v[i]$ to an existing shard if the end time of $L_v[i]$ is greater than the end time of the shard, i.e., $\mathit{end}(L_v[i]) \geq \sigma.\mathit{end}$. If there are multiple shards to which $L_v[i]$ can be assigned, we add it to the shard with the minimum gap, i.e., $\mathit{end}(L_v[i]) - \sigma.\mathit{end}$. If there are currently no shards to which $L_v[i]$ can be added, we start a new shard with $L_v[i]$ in it.

Algorithm 1: Idealized sharding algorithm

```
1:  Input: $L_v$ sorted in increasing order of begin times
2:  $\mathcal{I}_v = \emptyset$    // Idealized sharding
3:
4:  for $i = 1 .. |L_v|$ do
5:      // Iterate over all postings in the posting list for $v$
6:      if $\neg\exists\sigma \in \mathcal{I}_v : \sigma.\mathrm{end} \leq \mathrm{end}(L_v[i])$ then
7:          create new shard $\sigma_{new}$
8:          $\sigma_{new}.\mathrm{end} = 0$
9:          $\mathcal{I}_v = \mathcal{I}_v \cup \{\sigma_{new}\}$
10:     end if
11:     $\sigma_t = \underset{\sigma \in \mathcal{I}_v}{\mathrm{argmin}}\ (\mathrm{end}(L_v[i]) - \sigma.\mathrm{end})$
12:     $\sigma_t.\mathrm{end} = \mathrm{end}(L_v[i])$ // Update the end time of the shard
13:     $\sigma_t = \sigma_t \cup \{L_v[i]\}$ // Include the current posting into the shard
14:
15: end for
16:
17: Output: $\mathcal{I}_v$ is the idealized sharding.
```

3.4.2. Proof of Optimality

We develop the proof of optimality of Algorithm 1 by first proving three lemmas about key properties of the generated shards. Let the shards created by Algorithm 1 for a list L_v be numbered by their order of creation starting with σ_1, i.e., σ_1 was created before σ_2 and so on.

The first lemma states that the algorithm produces only shards that have the staircase property.

Lemma 3.1 (Staircase Property) *When Algorithm 1 terminates, every shard created by the algorithm has the staircase property.*

Proof: *We show this by contradiction. Assume that there is a shard σ that does not have the staircase property. This means that there is a pair of postings p and q in this shard such that $\mathrm{begin}(p) \leq \mathrm{begin}(q)$, and $\mathrm{end}(p) > \mathrm{end}(q)$ or $q \sqsubset p$. Since $\mathrm{begin}(p) \leq \mathrm{begin}(q)$, p was added to the shard before q. But when p was added, the end time of the shard was set to $\mathrm{end}(p)$ or $\sigma.\mathrm{end} = \mathrm{end}(p)$. Thus q could not have been be added to the same shard, which contradicts our assumption.* □

Lemma 3.2 (Descending-End Times) *If Algorithm 1 created a shard σ_i before σ_j, i.e. $i > j$, then $\sigma_i.\mathrm{end} > \sigma_j.\mathrm{end}$.*

Proof: *We prove this property by induction over increasing number of postings added.*

$i = 1$: *For the first posting e_1 the property holds since there are no earlier shards.*

$i \rightarrow i + 1$: *Let there be n existing shards $G = \{\sigma_1 \cdots \sigma_n\}$. Depending on the end time of the $i+1$th posting, i.e., $\mathtt{end}(e_{i+1})$ we consider two cases:*

If $\mathtt{end}(e_{i+1})$ is less than all the existing shard ends ($\mathtt{end}(e_{i+1}) < \sigma_k.\mathtt{end}\ , \forall \sigma_k \in G$), then e_{i+1} forms a new shard and the end of the shard is less than all the existing ends. This proves the claim.

Due to the induction hypothesis, the end times of shards are sorted in the descending order i.e., $\sigma_1.\mathtt{end} > \sigma_2.\mathtt{end} > \cdots > \sigma_n.\mathtt{end}$. If $\mathtt{end}(e_{i+1})$ is greater than any of the shards, then by definition (line 6), it will have to be added to the shard which minimizes the difference of their end times. It is easy to see that this shard is the earliest shard which can accommodate e_{i+1} since any other shard will either violate the staircase property or have a greater difference. This proves the claim. □

Lemma 3.3 (Temporal Subsumption of Postings) *For every posting in a shard σ_i ($i > 1$) there exists a posting in σ_{i-1} which completely subsumes it.*

Proof: *When a posting p is added to shard σ_i it is the last added posting, i.e., $\mathtt{last}(\sigma_i) = p$. Since we process all the postings in begin-time order, all postings which have been placed into shards before have a begin time less than $\mathtt{begin}(p)$. And, from the property of descending-end times $\mathtt{end}(\mathtt{last}(\sigma_{i-1})) > \mathtt{end}(\mathtt{last}(\sigma_i))$. Thus, at this current execution state of the algorithm the posting $\mathtt{last}(\sigma_{i-1})$ completely subsumes p, i.e., $\mathtt{last}(\sigma_{i-1}) \sqsupset \mathtt{last}(\sigma_i)$.* □

We now introduce the notion of a *stalactite set* of time intervals which is essential for the rest of the proof.

Definition 3.6 (Stalactite Set) *A stalactite set Υ consists of time intervals such that,*

$$\forall p, q \in \Upsilon, \mathtt{begin}(p) \leq \mathtt{begin}(q) \Rightarrow \mathtt{end}(p) > \mathtt{end}(q).$$

There may be many such stalactite sets that can be formed using postings from a given posting list, L_v. Let us denote the stalactite set of maximum cardinality as $\Upsilon_{max}(L_v)$.

Lemma 3.4 (Stalactite property) *The number of shards created by Algorithm 1 for a list L_v is equal to $|\Upsilon_{max}(L_v)|$.*

Proof: *We prove the lemma by contradiction. Assume that the new posting to be added e_{new} is not a part of $|\Upsilon_{max}(L_v)|$. We also assume that its addition creates a new shard $\sigma_{|\Upsilon_{max}(L_v)|+1}$ which is more than the claimed $|\Upsilon_{max}(L_v)|$ shards. Since the postings arrive in begin-time order, $\mathtt{begin}(e_{new})$ is greater than any of the previously processed postings. This means that $\mathtt{end}(e_{new}) < \mathtt{end}(\sigma_{|\Upsilon_{max}|})$ for it to start a new shard $\sigma_{|\Upsilon_{max}|+1}$. Now Lemma 3.3 says that there exists a posting in $\sigma_{|\Upsilon_{max}|}$ which subsumes e, making e a part of a larger stalactite set of* *cardinality* $|\Upsilon_{max}|+1$ *which is contrary to initial assumption that* Υ_{max} *is the maximal stalactite set.* □

We can now prove the optimality of our algorithm for idealized sharding.

Theorem 3.1 *Algorithm 1 creates an optimal sharding.*

Proof: *Lemma 3.1 establishes that there are no subsumptions in a shard. Since* $\Upsilon_{max}(L_v)$ *is the maximum size stalactite set, from Lemma 3.1, the overall number of shards are lower bounded by* $|\Upsilon_{max}(L_v)|$*. However, Lemma 3.4 proves that we exactly obtain* $|\Upsilon_{max}(L_v)|$ *shards by idealized sharding. This proves that the optimal solution to the idealized sharding of* L_v *has* $|\Upsilon_{max}(L_v)|$ *shards thus proving the optimality of Algorithm 1.* □

Further, we show that the algorithm can be implemented efficiently, by making use of the descending-end times property of the sharding at any stage during the algorithm. Due to this ordering of shard ends the addition step of postings to shards can be efficiently implemented via a binary search over the shard ends.

3.5. Cost-Aware Merging of Shards

Depending on the distribution of valid-time intervals, the idealized sharding introduced in the previous section might generate a large number of shards. Each shard requiring one *open-seek* operation, involving a random seek. If the cost of such a random seek is high and if the distribution of time intervals gives rise to many idealized shards, query-processing performance can degrade. In such cases, it might actually be beneficial to reduce the number of shards arising from idealized sharding at the cost of allowing some wasted reads.

In this section, we present an I/O cost-aware technique to selectively merge idealized shards allowing for a controlled amount of wasted reads while reducing the number of random seeks. We introduce a model for merging idealized shards which limits sequential wasted reads due to merging of a set of idealized shards by taking into consideration costs of random seeks and sequential accesses of the underlying index.

3.5.1. Model for Shard Merging

Multiple shards can be merged into a *merged shard* which contains all postings of the input shards and have a begin-time order on the intervals associated with the postings (see Figure 3.6). We denote a shard created from merging an input set of shards $\mathcal{S}$ as $\mu(\mathcal{S})$. In our model we take as input an idealized sharding $\mathcal{I}_v$ of a posting list L_v. Our intention is to reduce the overall number of shards per posting list by identifying idealized shards which can be merged according to our cost model. To this end we propose

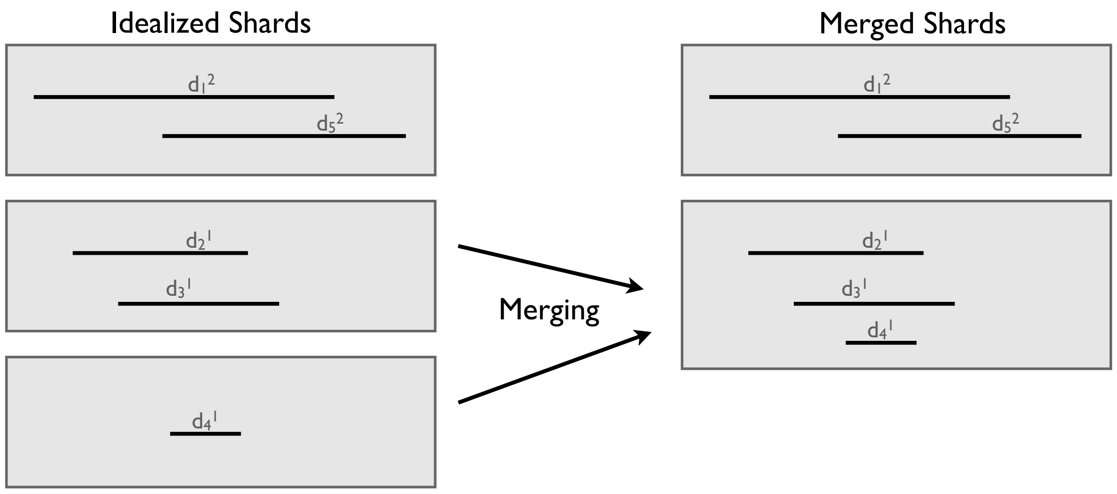


Figure 3.6.: Cost-aware shard merging

to disjointly partition the input set $\mathcal{I}_v$ into sets or partitions of shards M_v. Merged shards $\mu(\mathcal{S})$ are then created using shards from each partition $\mathcal{S} \in M_v$. The partitioning thereby ensures that no two merged shards have postings from the same idealized shard. The resulting merged shards, $\mu(\mathcal{S})$, $\mathcal{S} \in M_v$, thus formed also are a feasible index sharding of L_v, i.e.,

$$\bigcup_{\mathcal{S} \in M_v} \mu(\mathcal{S}) = L_v.$$

Merging of shards might result in subsumptions of postings which further leads to wasted reads during query processing. However, not all query time-points lead to wasted reads. For example, in Figure 3.4 only queries which have a begin time interval [t6, t7] result in wasted reads. Secondly, an open-seek operation to a merged shard accompanied by a few sequential wasted reads may be cheaper than two open-seek operations to idealized shards without any wasted reads. These are two factors which should be taken into considerations in choosing which shards to merge. An accurate estimate of the performance of a shard can be modelled by considering the performance over all query time-points.

Cost Model We let the cost of a random seek be C_r, and that of a sequential read be C_s. We allow for a penalty function $\Psi(\mathcal{S})$, over a set of shards $\mathcal{S}$ and require it to be bounded by the parameter η. To reconcile the costs of random and sequential accesses the parameter η is set to $C_r/C_s - 1$. We refer to this bounding of the penalty function as the *threshold criterion*.

$$\Psi(\mathcal{S}) \leq \eta$$

An example of such a penalty function is *expected wasted reads*, which is defined as the number of wasted reads incurred during query processing, averaged over all possible query time points. We define $w(\sigma, t)$ as the set of wasted postings read for a query with a time-point t over a shard σ. We consider a discrete notion of time and denote the time granularity as δ. If all possible query times lie in the interval $[t_0, t_n]$ the number of valid query time-points is $\frac{t_n - t_0}{\delta}$. Formally,

Definition 3.7 (Expected Wasted Reads) *Given a merged shard $\mu(\mathcal{S})$ and its wasted read distribution $w(\mu(\mathcal{S}), t)$, the expected wasted reads incurred per query time-point is given by*

$$\Psi(\mathcal{S}) = \frac{\sum_{t \in [t_0, t_n]} |w(\mu(\mathcal{S}), t)|}{(t_n - t_0)/\delta}.$$

Under this penalty function, a set of shards can be merged when the expected wasted sequential reads in the *merged shard* is less than the overhead incurred in an open-seek operation. If costs of sequential and random accesses are the same, $C_r = C_s$, we obtain $\eta = 0$ which means we resort to idealized sharding. Understandably, with $C_r = C_s$ the cost of accessing a new shard is the same as reading a wasted posting. Thus it is preferable to avoid wasted reads completely in this situation and employing idealized sharding. However, in more practical scenarios $C_r > C_s$. As an example, if $\eta = 100$, then the wasted reading of less than 100 postings, that do not qualify by the temporal constraint, on an average would be more beneficial than performing a random seek to an additional shard. One can, in principle, use other notions of aggregation measures for $w(\sigma, t)$ and then define the penalty function accordingly. In this work we use expected wasted reads as the penalty function.

The partitioning of idealized shards can now be formulated as an optimization problem where we have as input a set of idealized shards, the parameter η, and the penalty function. Like the idealized-sharding problem we would want to minimize the overall number of shards for minimizing the expensive open-skip operations. The *cost-aware shard merging problem* intends to find a partitioning M_v of idealized shards $\mathcal{I}_v$ so as to have minimum number of partitions subject to the threshold criterion on each partition. Formally,

Definition 3.8 (Cost-aware Shard Merging Problem)

$$\operatorname{argmin} |M_v| \quad \textit{s.t.} \quad \Psi(\mathcal{S}) \leq \eta \ : \ \forall \mathcal{S} \in M_v.$$

To solve this problem we first look at how we can efficiently determine the penalty of merging a partition or set of idealized shards $\mathcal{I}_v$. A partition containing at least two idealized shards will have a non-zero penalty. To compute the overall penalty value of a partition we need to aggregate the wasted read distribution for the combination of shards in the partition over the entire time-period. The combinatorial nature of the problem makes it prohibtive to pre-compute penalty values for all combinations. What we do instead is compute penalty values of pairs of shards and show that the penalty of any partition (≥ 2) can be computed efficiently from these pairwise penalties.

Given a partition S of m idealized shards the wasted reads at a query time-point t is $w(\mu(S), t)$. We retain the order in which idealized shards were created, i.e., shards created early have a lower index. We further let $\mathtt{first}(\mathcal{S})$ denote the shard which was created first in $\mathcal{S}$. For shards $\sigma_i, \sigma_j, \sigma_k \in S$ and $i < j < k$, it holds

$$w(\mu(i,k), t) = w(\mu(j,k), t).$$

Thus $w(\mu(S), t) = \sum_{\sigma \in \mathcal{S}} \Psi(\{\mathtt{first}(\mathcal{S}), \sigma\})$. The penalty $\Psi(\mathcal{S})$ follows from this observation and Definition 3.8:

$$\Psi(\mathcal{S}) = \sum_{\sigma \in \mathcal{S}} \Psi(\{\mathtt{first}(\mathcal{S}), \sigma\}).$$

As an example, the penalty incurred due to merging idealized shards {7,10,3,12} would be $\Psi(3,7) + \Psi(3,10) + \Psi(3,12)$. Computing wasted reads at each time point can be efficiently implemented by interleaving computation of pairwise wasted reads within Algorithm 2.

3.5.2. Algorithm for Shard Merging

We present a heuristic algorithm which is shown to perform well in practice in our experimental evaluation. As inputs we expect the set of the idealized shards and η. We retain the order in which idealized shards were created, i.e., earlier created shards have a lower index.

The pseudo code for merging the idealized shards is presented in Algorithm 2. Every iteration employs a two stage greedy process. The first stage is an *ascending choice phase* in which it chooses all the unmerged/available idealized shards in ascending order of their index until the threshold constraint is violated (lines 11 to 20).

The second stage is a greedy phase (lines 23 to 27) where the remaining capacity is greedily chosen with smallest unmerged shard first (as in the standard greedy approach to the knapsack problem).

Algorithm 2: Cost-Aware Shard Merging

```
1:  Input: $\mathcal{I}_v$ and $\Psi(\sigma_i, \sigma_j)$
2:  $M_v = \emptyset$   // Merged shards
3:
4:  for $i = 1 .. |\mathcal{I}_v|$ do
5:    Let $\sigma_i \in \mathcal{I}_v \setminus M_v$ be next shard in order
6:    create new shard $r_i$
7:    $r_i = r_i \cup \{\sigma_i\}$
8:    $capacity = \eta$
9:
10:   // Ascending choice phase
11:   for $j = i + 1 .. |\mathcal{I}_v|$ do
12:     if $(\Psi(\sigma_i, \sigma_j) \leq capacity) \wedge (\sigma_j \notin M_v)$ then
13:       $capacity = capacity - \Psi(\sigma_i, \sigma_j)$
14:       $r_i = r_i \cup \{\sigma_j\}$
15:     else
16:       if $(\sigma_j \notin M_v) \wedge (\sigma_j \notin r_i)$ then
17:         break
18:       end if
19:     end if
20:   end for
21:
22:   // Smallest size first
23:   while $capacity > 0$ do
24:     $\sigma_{min} = \underset{g \in \mathcal{I}_v \setminus M_v}{argmin} \{\Psi(\sigma_i, g)\}$
25:     $capacity = capacity - \Psi(\sigma_i, \sigma_{min})$
26:     $r_i = r_i \cup \{\sigma_{min}\}$
27:   end while
28:   $M_v = M_v \cup r_i$
29: end for
30:
31: Output: $M_v$ is the set of merged shards.
```

3.6. Index Maintenance

So far, we have assumed a static document collection. These indexing techniques are fine for collections that change infrequently or never. In such cases, occasionally rebuilding indexes is a viable solution in practice. However, for web archives, which grow frequently and dynamically, the index needs to be updated frequently. New terms are added to the index and shards for existing terms are modified. In this section we describe index-maintenance strategies to deal with dynamic updates. We first discuss our assumptions about the nature of the updates in an archive setting and the factors relevant for indexing.

Updates in web archives are a result of periodic crawls. An update might result in either (i) new *unseen* document, or (ii) report modifications to an already existing document. Recollect that in Section 3.2 we established the document model where each document is a sequence of versions. Modifications to an existing document, either addition or deletion of content, results in the creation of a new version in the document version sequence. If a new document is reported, a new sequence is started for the document with the reported version being the first. Importantly, existing versions are never removed. This means that the index steadily grows over time with the older collection indexed by the archive index being a subset of the current collection. We assume that deletions in the past are rare and in our current system arbitrary deletions in the past are not supported.

Each version is associated with a valid-time interval. Every update results in a change in the valid-time interval of the *current version* of each document. Consider the state of the archive at time t_{now} when it was updated. If the update reports no change in the state of a document version d_i^4 (document d at its fourth version) the end of the valid-time of d_i^4 is updated to t_{now}. On the contrary if a new version for d would have been reported, the end time of d_i^4 would have been *finalized* to t_{now} (never to be modified again). Additionally, the fifth version d_i^5 would have been started with a begin time as t_{now}. An accurate estimate of the valid-time intervals of the versions are determined when they are finalized, and hence they are sent to the indexing pipeline in the order of their end times.

With these assumptions in mind we distinguish between *active versions*, as the most recent and still current document versions, and *archive versions*, as the document versions already superseded by a more recent version of the same document. The active version of a document turns into an archive version when the document is re-crawled and either a new active version is found or the document was removed from the Web. In both cases, the end time of the old version is set to the current time.

During indexing we organize postings for active versions into an *active index* and the archived versions are indexed in an *archive index*. The active index is implemented as an incrementally updatable inverted index [BC08, LMZ08]. Depending on the fraction of time-travel queries among all queries posed to the system, posting lists in the active index can be organized to efficiently support queries on active versions or time-travel queries. For the former, postings may be ordered by their document identifier to allow for more efficient query processing, possibly together with additional structures [BCH+03, DS11, MZ96, SC07]. For the latter, postings may be ordered by the begin boundary of their valid-time interval to support efficient filtering of recent document versions.

The archive index on the other hand is implemented as a sharded index and is our primary focus. Finally, from the indexing point of view it is preferable to have an updating scheme which appends newly created postings at the end of the posting list. The major benefit of an append operation is that the indexes built for the older indexes can be used as partial solutions to build an up-to-date archive index efficiently thus avoiding recomputation. Recomputation of shards with a non-append based technique (say cost-aware shard merging) is expensive because it involves processing the entire input (existing data along with the new updates) thus limiting its applicability to an update aware indexing system.

To this end, we develop *incremental sharding* which

- is competitive in query processing by trading-off, like shard merging, wasted reads for random-accesses,
- takes into account the end-time order of arrival of input, and
- can be maintained easily because it can be incrementally computed and results in append-only operations to shards.

3.7. Incremental Sharding

Incremental sharding takes into account the relative costs of random and sequential accesses by a more restrictive bound on the number of subsumptions for a given shard. It bounds the absolute number of subsumptions for a given shards. This is opposed to shard merging where the number of subsumptions per shard is on an average limited to a system-wide parameter. A bound on the number of subsumptions per shard translates to bounding the number of sequential wasted reads. We term this restriction as the *bounded subsumption* property. A shard is said to exhibit a bounded subsumption property if a posting in that shard does not subsume more than η postings, where η is a system-wide constant (same as in sharding merging). Formally,

Definition 3.9 (Bounded Subsumption) *A shard σ is said to satisfy the bounded subsumption property with threshold η if each posting $p_i \in \sigma$ does not subsume more than η postings:*

$$\forall p_i \in \sigma : |\{p_j \in \sigma \mid p_j \neq p_i \ \wedge \ p_i \sqsupset p_j\}| \leq \eta \ .$$

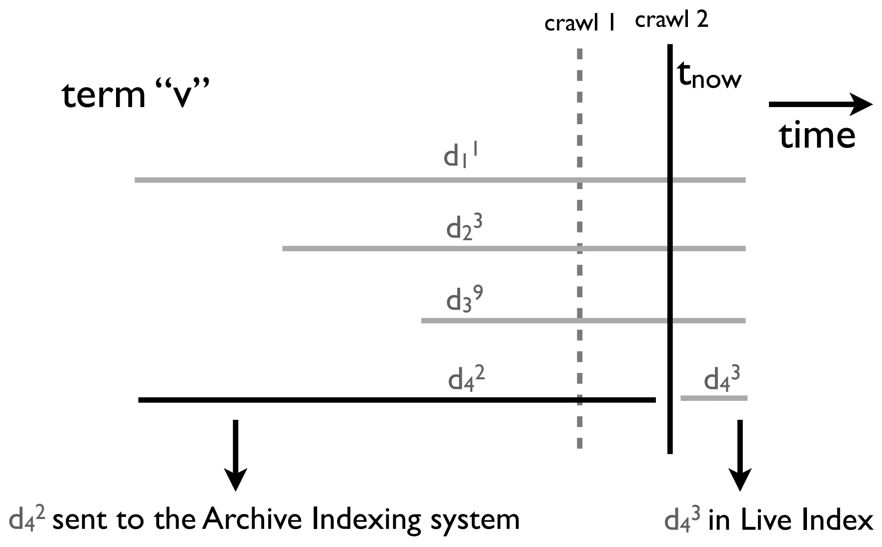


Figure 3.7.: End time order of finalizing versions

We let the Boolean predicate $\mathtt{bounded}(\sigma, \eta)$ *denote whether the shard* σ *has the bounded-subsumption property with a threhold* η.

The problem of minimizing the number of random accesses with a limited number of wasted sequential accesses can be redefined in terms of minimizing the overall number of shards such that each shard exhibits the bounded subsumption. Formally,

Definition 3.10 (Incremental Sharding Problem) *Given a set of postings* L_v *for a term* v, *partition* L_v *into a feasible index sharding* $\mathcal{S} = \{\sigma_1, \dots, \sigma_m\}$,

$$\operatorname*{argmin}_{\mathcal{S}} |\mathcal{S}| \quad \textit{s.t.} \quad \forall \sigma \in \mathcal{S} : \mathtt{bounded}(\sigma, \eta) \ .$$

Before attempting to solve the above problem let us revisit the properties in an archive indexing setting. Firstly, the archive setting is very specific in terms of arrival of the input sequence, i.e., in the order of arrival of new versions to the archive index. Whenever the end time of an existing version is determined, it is sent to the archive indexing system (see Figure 3.7). Since versions are generated in end-time order, the input intervals also follow the same order.

Secondly, we do not deal with deletions of versions since a deletion of a document results in a posting in the archive index for that document, and existing versions are never removed.

Finally, we would want to avoid recomputation of shards by only allowing appends to the materialized shards. Apart from avoiding recomputation ensuring an append-only operation also avoids expensive decompression and compression cycles. While

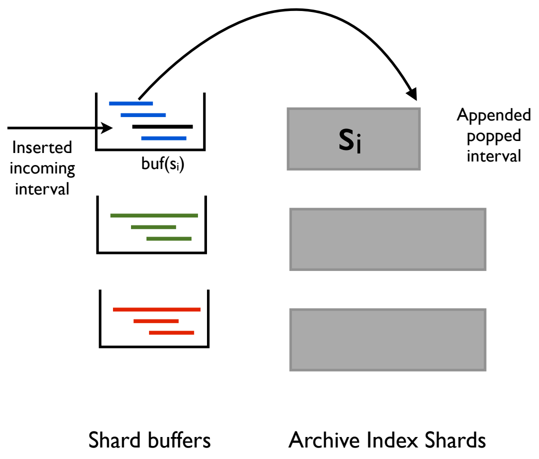


Figure 3.8.: Incremental sharding

merging two posting lists by employing recomputation, entire posting lists are decompressed and re-ordered (according to posting-begin times) to form usable input for the sharding algorithm. After recomputation, the new set of shards are compressed back again for storage. The most popular compression algorithms used in posting list compression are based on gap-encoding schemes which are local schemes and hence append friendly. Hence, rather than optimally solving the problem, we look for an approach which is based on an append-only operation to the existing shards and exploits the end-time order of input arrival. In the following section, we introduce the *incremental sharding* algorithm which apart from being incremental also has an approximation guarantee.

3.7.1. Incremental Sharding Algorithm

We present the *incremental sharding algorithm* which is an update aware incremental algorithm with a factor $(2 - \frac{2}{\eta+2})$ approximation guarantee (see Algorithm 3). Apart from having the natural benefit of being an append-only algorithm, it also exploits the end-time arrival order of the postings by processing the postings in that order thus avoiding expensive sort operations on the input.

The algorithm processes postings in the increasing order of end times (see line 1) and creates or updates shards incrementally. It follows a scheme of immediate assignment but deferred append of a posting to a shard. For each shard, we maintain a *shard buffer* of size $\eta + 1$ and a *shard-begin time*. The assignment of the posting to a shard is based on the begin time of the shard and the shard buffer defers the actual writing or appending of the posting of the shard to satisfy the bounded subsumption property. In other words, the shard buffer maintains the posting until it deems it right to be appended to the end

Algorithm 3: Incremental sharding algorithm

```
1: Input: (i)$\eta$, (ii)$L_v$ sorted in increasing order of end times
2: $\mathcal{S} = \emptyset$    // Incremental sharding
3:
4: for $i = 1 .. |L_v|$ do
5:    //Creates new shard
6:    if $\neg\exists S \in \mathcal{S} : S.\mathtt{begin} \leq \mathtt{begin}(L_v[i])$ then
7:       create new shard $\sigma_{new}$ and $\mathtt{buf}(\sigma_{new})$
8:       $\sigma_{new}.\mathtt{begin} = 0$
9:       add $L_v[i]$ to $\mathtt{buf}(\sigma_{new})$
10:      $S = \mathcal{S} \cup \{\sigma_{new}\}$
11:   end if
12:
13:   //Find the best candidate shard for assignment
14:   $\sigma_{cand} = \{\sigma_i \mid \sigma_i \in \mathcal{S} \wedge \sigma_i.\mathtt{begin} \leq \mathtt{begin}(L_v[i])\}$
15:   $\sigma_t = \underset{g \in \sigma_{cand}}{\mathtt{argmin}} (\mathtt{begin}(L_v[i]) - g.\mathtt{begin})$
16:
17:   //Update buffers and begin times of shards
18:   if $\sigma_t \neq \sigma_{new}$ then
19:      insert $L_v[i]$ into $\mathtt{buf}(\sigma_t)$ in begin-time order
20:   end if
21:   if $|\mathtt{buf}(\sigma_t)| = \eta + 1$ then
22:      //first element in the buffer finalized
23:      $\sigma_t = \sigma_t \cup \mathtt{removefirst}(\mathtt{buf}(\sigma_t))$
24:      $\sigma_t.\mathtt{begin} = \mathtt{begin}(\mathtt{first}(\mathtt{buf}(\sigma_t)))$
25:   end if
26: end for
27:
28: //Finally append the buffer postings to the shards
29: for $S \in \mathcal{S}$ do
30:    $S = S \cup \mathtt{buf}(S)$
31: end for
32:
33: Output: $\mathcal{S}$ is the incremental sharding.
```

of the shard.

When a posting $L_v[i]$ is processed, it is either assigned to an existing shard based on the posting and shard-begin times (see line 15), or it results in the creation of a new shard (lines 7-10). The creation of a new shard involves setting the begin time of the shard to zero and placing the chosen posting in its shard buffer. Until the buffer reaches its capacity of η the begin time of the shard remains zero. The shard-begin time is first updated when a posting is popped out of it after the buffer reaches its capacity $\eta + 1$. When the assignment for the posting is decided, say σ_t, it is placed in the respective shard buffer $\mathtt{buf}(\sigma_t)$ (see line 19). The incremental sharding chooses the shard whose begin time has the least difference with the begin time of the incoming posting (see line 15).

The shard buffers determine the relative position of the postings in the shard where it will be finally stored. The insertions into the buffer are made to preserve the begin-time order (see line 19) which in turn ensures a begin-time order when postings are removed from it. This is shown in Figure 3.8. Only the first posting or the posting with the minimum begin time is removed from the buffer to limit the buffer size to $\eta + 1$ (line 23) and it is appended to the end of its corresponding shard σ_t. The shard buffers also ensure that no posting in a shard subsumes more than η postings. This is done by setting the begin time of a shard $\sigma_t.\mathtt{begin}$ to the first posting $\mathtt{begin}(\mathtt{first}(\mathtt{buf}(\sigma_t)))$(or the posting with the least begin time) of the shard buffer as in line 24. The posting with the minimum begin time in the shard can subsume the maximum number of postings and any posting with a begin time lesser than it is disallowed.

Note that the use $\cup$ in lines 23 and 30 indicates the *append* operation on the shard that is logically organized as a list of postings in their begin-time order.

3.7.2. Approximation Guarantee for Incremental Sharding

Theorem 3.2 *Incremental sharding is a* $(2 - \frac{2}{\eta+2})$ *approximation.*

We use the following lemmas to prove the theorem. Assuming that incremental sharding produces m shards, we first construct a worst case scenario. For notational convenience let us assume that shards are numbered according to their creation times in incremental sharding, i.e., σ_1 was created before σ_2 and so on.

Lemma 3.5 (Descending Begin Times) *If incremental sharding created a shard* σ_{i+1} *after* σ_i, *then* $\sigma_i.\mathtt{begin} > \sigma_{i+1}.\mathtt{begin}$.

Proof: *We prove this property by induction over increasing number of postings* $p_i \in L_v$ *which are added in in end-time order, i.e,* $\mathtt{end}(p_{i+1}) > \mathtt{end}(p_i)$.

$i = 1$: *For the first posting* p_1 *the property holds since there are no earlier shards.*

$i \rightarrow i + 1$: *Let there be* n *existing shards* $\mathcal{S} = \{\sigma_1 \cdots \sigma_n\}$. *Depending on the begin time of the* $(i + 1)$*-th posting* $\mathtt{begin}(p_{i+1})$ *we consider the following two cases:*

Case 1: *If* $\mathtt{begin}(p_{i+1}) < \sigma_k.\mathtt{begin}$, $1 \le k \le n$, *then* p_{i+1} *forms a new shard* σ_{new} *and the begin of the shard* $\sigma_{new}.\mathtt{begin}$ *is less than all the existing shard-begin times. This proves the claim.*

Case 2: *Assuming* $\exists \sigma_k : \mathtt{begin}(p_{i+1}) \ge \sigma_k.\mathtt{begin}$ *and on addition of* p_{i+1} *to* σ_k *there is a violation of the descending begin-time order of shards, i.e.,* $\sigma_k.\mathtt{begin} > \sigma_{k-1}.\mathtt{begin}$. *This means that* $\sigma_{k-1}.\mathtt{begin} < \mathtt{begin}(p_{i+1})$ *and* p_{i+1} *should have been assigned to* σ_{k-1} *due to a smaller difference according to the induction hypothesis of* $\sigma_{k+1}.\mathtt{begin} > \sigma_k.\mathtt{begin}$, $\forall 1 \le k < n$. □

Lemma 3.6 (Incremental Subsumption) *The number of postings subsumed a posting added to* σ_i *is at least* $(i-1)(\eta+1)$.

Proof: *Note that* $\sigma_i.\mathtt{begin}$ *refers to the time of the first posting in the buffer of shard* σ_i *or the earliest begin time in the shard buffer* $\mathtt{buf}(S)$.

By Lemma 3.5 we know that there is an ordering of the shard-begin times. Hence for a posting p *assigned to* σ_i *the following holds –* $\mathtt{begin}(p) < \sigma_{i-1}.\mathtt{begin} < \cdots < \sigma_1.\mathtt{begin}$. *Since we assume end-time arrival order of postings, it subsumes all postings in* $i-1$ *shards buffers, which are* $\sigma_1, ..., \sigma_{i-1}$. *Further the fixed buffer size of* $\eta+1$ *results in making the subsumptions lower bounded by* $(i-1)(\eta+1)$. □

We now introduce the notion of *stalactite groups*, building on the notion of the *Stalactite set* Υ according to the definition 3.6. Formally,

Definition 3.11 (Stalactite Groups) *A shard* σ *is said to be exhibit stalactite grouping if we can partition the shard into sets of postings called groups* G *such that for groups* $s_i, s_j \in \sigma$ *and* $i < j$ *the following holds:*

$$q \sqsupset p, \quad \forall p \in s_i, \forall q \in s_j.$$

In other words, *stalactite groups* (see Figure 3.9) in a shard is the organization of postings into groups such that choosing one posting from each group results in a *stalactite set*.

Lemma 3.7 (Stalactite Grouping) *Stalactite grouping with a staircase property in each shard is the worst case for incremental sharding.*

Proof: *From the previous lemma, any input resulting in* m *shards from incremental sharding will have at least postings with* $\eta+1$, $2(\eta+1)$, $\cdots$, $(m-1)(\eta+1)$ *subsumptions in* $\sigma_2, \sigma_3, \cdots, \sigma_m$ *respectively. Let us suppose that the set of subsumed postings when the first posting added to* σ_i *be represented as* $\mathtt{sub}(\sigma_i)$. *To reduce the overall number of shards we should strive for a configuration with a minimum number of subsumptions. This is possible when* $\mathtt{sub}(\sigma_1) \subseteq \mathtt{sub}(\sigma_2) \subseteq \cdots \subseteq \mathtt{sub}(\sigma_m)$ *and* $|\sigma_i| = \eta+1, \forall i = 1, \ldots, m$. *This arrangement forms a* stalactite group *(see Figure 3.9) with each of the groups having a cardinality of* $\eta+1$.

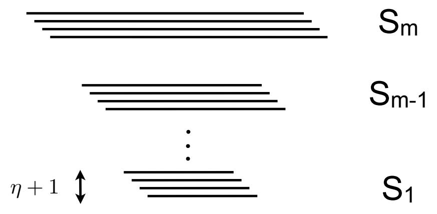


Figure 3.9.: Stalactite groups

Additionally, each of these stalactite groups should have a staircase arrangement to allow for the maximum capacity - η - out-of-place insertions. Any additional posting or misaligned posting either increases subsumption or reduces capacity for out of place insertions. Any removal of postings on the other hand result in contradiction to the original assumption that there are m *shards from incremental sharding.* □

Now we can complete the proof for Theorem 3.2.

Proof: *From the Lemma 3.7 we know that there are* m *stalactite groups with each group residing in the shards formed from incremental sharding. It is easy to see that none of the optimal shards will have more than* $2\eta + 1$ *postings. Thus we choose an assignment where we try to minimize the number of shards, i.e., choose as many shards with* $2\eta + 1$ *postings as possible. One such assignment is when we assign* η *of the* $\eta + 1$ *postings of* σ_i *to* σ_{m+1-i}, $\forall i \leq \frac{m}{2}$*. The remaining* $\frac{m}{2}$ *postings (a posting from each* σ_i*) can then be placed in* $\frac{m}{2(\eta+1)}$ *shards. Hence for* m *shards created by incremental sharding we get a minimum of* $\frac{m}{2} + \frac{m}{2(\eta+1)}$ *shards. Notice that we can have other arrangements which give the same number of minimum shards. The ratio*

$$\frac{|S|}{OPT} = \frac{m}{\frac{m}{2} + \frac{m}{2(\eta+1)}} = 2\left(1 - \frac{1}{\eta + 2}\right)$$

proves that incremental sharding is a factor $(2 - \frac{2}{\eta+2})$ *approximation algorithm.* □

3.8. System Architecture

Figure 3.10 shows a high-level overview of the architecture of a search engine using our incremental sharding method. It consists of

- the *active index* for all active versions of documents, consisting of an in-memory inverted list for each term that keeps the active versions of documents,
- the *archive index* for all archive versions of documents, consisting of an inverted list for each term that is organized in shards. The archive index consists of an in-memory index *IMAI* and an on-disk index *EMAI*, both organized in shards.
- A *crawler* that continuously crawls the target Web sites, for example, a predefined set of domains or the complete Web.

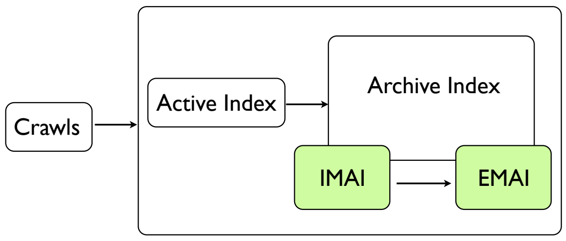


Figure 3.10.: System architecture

When the crawler encounters a new document that has been unknown so far, it adds it to the active index; this is an inexpensive operation since the active index is in main memory. When a document is found again, it is checked for changes (using, for example, a fingerprinting technique such as [BGMZ97, Hen06]). If changes are detected, the active version of that document turns into an archive version (with end time equal to the crawl time) and is sent to the archive index, and postings for the new active version are added to the active index.

The archived version is then added to the in-memory archive index by first creating the corresponding postings for each term, which are then added to the in-memory archive index using the incremental technique from Section 3.7. Figure 3.7 shows an example for this, where in a crawl at time t_{now} a new version d_4^3 for document d_4 is detected. This results in firstly adding a new version d_4^3 with a begin time t_{now} to the affected terms in the active index. Secondly, the end time of d_4^2 is finalized and the posting $\langle d_4^2, [t_1, t_{now}], \mathtt{score} \rangle$ with the complete posting information is sent to the archive indexing system. In the archive indexing system this posting is processed by placing it in the shard buffer of term "v" and updating the IMAI from the popped posting in the buffer as shown in Figure 3.8. As soon as the in-memory index IMAI is full, postings are merged into the disk-based archive index EMAI, merging corresponding shards; this essentially corresponds to incremental maintenance of standard inverted lists.

3.9. Experimental Evaluation

In this section, we describe our experimental evaluation of index sharding. We first present our experimental setup, datasets, and workloads used. We then examine in detail the impact of all indexing methods considered on query processing, index sizes and index maintenance.

Dataset	Coverage	Size (in GB)	$\mathcal{N}$	$\mathcal{V}$	μ/σ
WIKI	2001 to 2005	~700	1,517,524	15,079,829	9.94 / 46.08
UKGOV	2004 to 2005	~400	685,678	17,297,548	25.23 / 28.38

Table 3.2.: Characteristics of datasets used

3.9.1. Setup

All experiments were conducted on Dell PowerEdge M610 servers with 2 Intel Xeon E5530 CPUs, 48 GB of main memory, a large iSCSI-attached disk array, and Debian GNU/Linux (SMP Kernel 2.6.29.3.1) as operating system. Experiments were conducted using the Java Hotspot 64-Bit Server VM (build 11.2-b01).

3.9.2. Datasets Used

For our experiments we use the following two real-world datasets WIKI and UKGOV. The characteristics of these datasets are detailed below and summarized in Table 3.2.

WIKI The *English Wikipedia Revision History* [WIK13], whose uncompressed raw data amounts to 0.7 TBytes, contains the full editing history of the English Wikipedia from January 2001 to December 2005. We indexed all versions of encyclopaedia articles excluding versions that were marked as the result of a minor edit (e.g., the correction of spelling errors etc.). This yielded a total of 1,517,524 documents with 15,079,829 versions having a mean (μ) of 9.94 versions per document at standard deviation (σ) of 46.08.

UKGOV This is a subset of the European Archive [EA13], containing weekly crawls of eleven governmental websites from the U.K. We filtered out documents not belonging to MIME-types `text/plain` and `text/html` to obtain a dataset that totals 0.4 TBytes. This dataset includes 685,678 documents with 17,297,548 versions ($\mu = 25.23$ and $\sigma = 28.38$).

These two datasets represent realistic classes of time-evolving document collections. WIKI is an explicitly versioned document collection, for which all its versions are known. UKGOV is an archive of the evolving Web, for which, due to crawling, we have only incomplete knowledge about its versions. The incomplete knowledge can be attributed to inability to capture some versions in between crawls and inability to determine with certainty the version valid-times. For ease of experimentation, we rounded timestamps (in both datasets) to day granularity.

3.9.3. Index Management

We use the following types of indexes in our experiments:

1. **Sharded Index** We consider idealized sharding (IS) and three variants of cost-aware shard merging (CAS) as introduced in Section 3.5. The parameter η reflects the I/O cost ratio. Since the C_r/C_s values of disks usually vary in the order of 100 and 1000 thus we choose the parameter for shard merging accordingly, i.e., $\eta \in \{10, 100, 1000\}$. The corresponding indexes are denoted as CAS-10, CAS-100 and CAS-1000. The penalty function used for shard merging is based on *expected wasted reads*.

   We also consider indexes created with *incremental sharding* (INC) as described in Section 3.7, with the same parameter values for CAS, i.e., $\eta \in \{10, 100, 1000\}$.

2. **Vertically-Partitioned Index** As the first competitor, we consider the vertically-partitioned index, referred to as VERT from now on, that are partitioned following the *space-bound* approach [BBNW07]. The parameter κ denotes the space restriction that models the maximum blowup in the index size relative to a non-partitioned index. For our experiments we consider four variants of the space-bound approaches i.e., parameter values for $\kappa \in \{1.5, 2.0, 2.5, 3.0\}$. These variants are denoted subsequently in the text as VERT-1.5, VERT-2.0, VERT-2.5 and VERT-3.0.

3. **Naïve Unpartitioned Index** As a second competitor, we build an unpartitioned index with provision for impact lists over ordered begin times referred to as CAS-inf. This serves as a proof that our techniques are effective not only because of the impact list construction and a global begin-time order.

We also evaluate the effect of temporal coalescing [BBNW07] on index size and query processing. To this effect we build sharded and vertically-partitioned indexes with application of temporal coalescing using a parameter $\epsilon = 0.01$. Other than that, we use the same choice of parameters as in the experiments without temporal coalescing.

Both the vertically-partitioned indexes and the sharded indexes are stored on disk using flat files containing both the lexicon as well as the partitioned posting lists. We do not filter out stop words, nor do we apply stemming/lemmatization when indexing the above datasets. We assigned document identifiers in the order of the begin time of the document versions. For compression, we employ 7-bit encoding on d-gaps and apply *temporal coalescing* whenever necessary. Note that variable-byte encoding is complementary to *temporal coalescing*. We use scalar payloads and store the tf-scores as floating point numbers using eight bytes.

At runtime, the lexicon and impact lists are read completely into main memory, and for a given query the appropriate partitions or shards are retrieved from the index flat file on disk.

3.9.4. Query Workloads and Execution

We compiled two dataset-specific query workloads by extracting frequent queries from the AOL query logs, which were temporarily made available during 2006. For the WIKI dataset we extracted 300 most frequent queries which had a result click on the domain `en.wikipedia.org` and similarly for UKGOV we compiled 50 queries which had result click on `.gov.uk` domains. Both the vertically-partitioned and sharded index structures are built for terms specific to the query workload. Using these keyword queries, we generated a time-travel query workload with five instances each for the following four different temporal predicate granularities: day, month, year and queries spanning the full lifetime of the respective document collection. Hence, the workload comprised of 6000 time-travel queries for WIKI and 1000 queries for UKGOV.

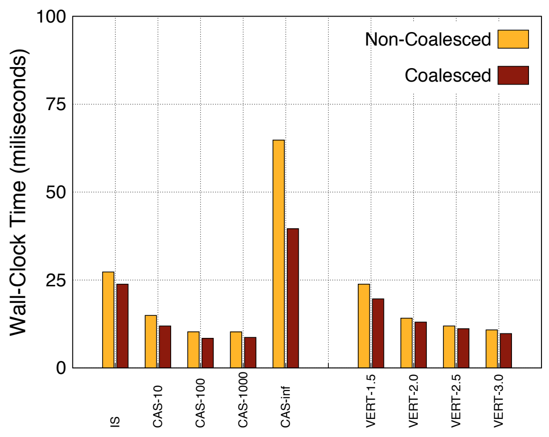
For query processing, we employed conjunctive query semantics, i.e., query results contain documents that include all the query terms. We use wall-clock times (in milliseconds) to measure the query-processing performance on warm caches using only a single core. Specifically, each query was executed five times in succession and the average of the last four runs was taken for a more stable and accurate runtime measurement.

3.10. Experimental Results

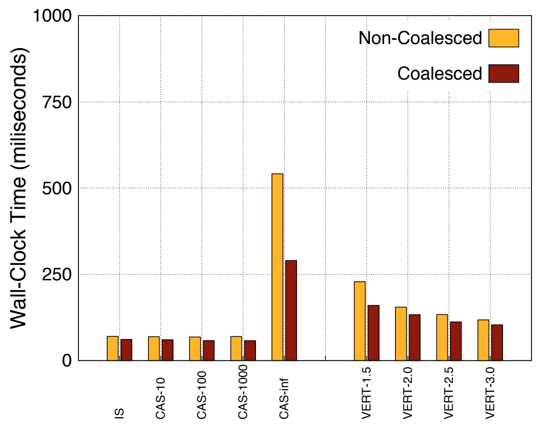
3.10.1. Sharding vs Vertical Partitioning

In the first set of experiments, we compare the performance of CAS and VERT on different query granularities. Query granularity of a time-travel text query refers to the time interval component of the query. Figures 3.11 through to 3.14 presents the query response times, in milliseconds, for the different variants of CAS and VERT. Each chart corresponds to the performance of CAS and VERT on a given query granularity – day (Figure 3.11), month (Figure 3.12), year (Figure 3.13) or the full lifetime of the respective collection (Figure 3.14). While comparing CAS against VERT, we choose CAS-1000 as a reasonable representative for sharding since the wall-clock times for both CAS-100 and CAS-1000 show low variance throughout all experiments.

VERT-3.0 is optimized for query performance, because of a higher degree of partitioning, and as expected it has the best performance among its other counterparts ($\kappa = \{1.5, 2.0, 2.5\}$). In case of WIKI, we see a low difference in query processing times between VERT-3.0 (10.63 ms) and CAS-1000 (11.86 ms) for day-granularity queries (see



(a) Wikipedia

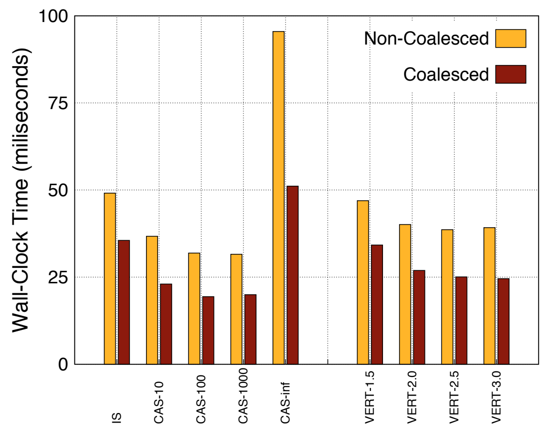


(b) UKGOV

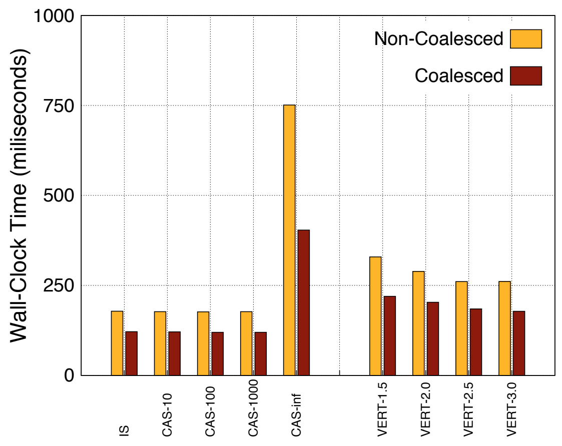
Figure 3.11.: Wall-clock times for day-granularity queries

Figures 3.11(a)). The difference is notable for month-granularity queries with CAS-1000 exhibiting a 19.5% improvement over VERT-3.0 (see Figure 3.12(a)). In UKGOV, CAS-1000 takes 69.88 ms to process day-granularity queries which is almost a 40% improvement over VERT-3.0 which takes 117.9 ms (see Figure 3.11(b)). This shows that although VERT is optimized for short time interval queries, and only one or very few partitions have to be accessed for processing, the number of wasted reads accessed in VERT is substantially more those accessed by CAS.

Next, we compare performances for longer time-granularity queries, i.e., year and full-lifetime queries. In case of WIKI, comparing VERT-1.5, which has the best runtimes for year queries and lifetime queries, with CAS-1000 shows that the latter consistently outperforms the former in both cases. Query-processing times improve by 22.2% (see Figure 3.13(a)) for year-granularity queries and by 19% for full lifetime queries (see Fig-



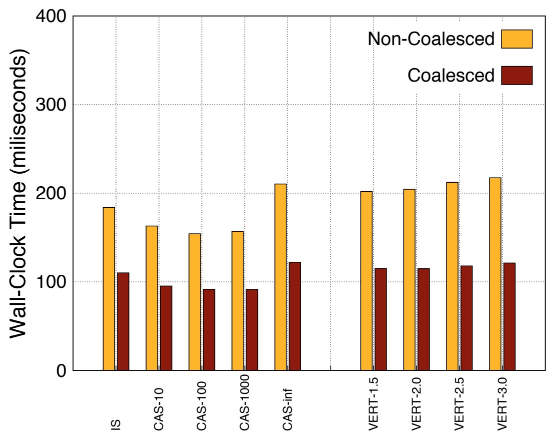
(a) Wikipedia



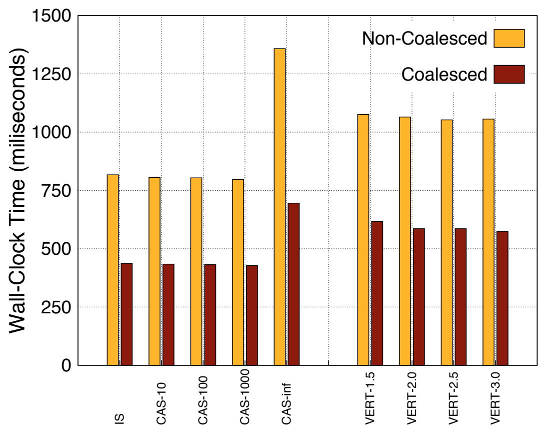
(b) UKGOV

Figure 3.12.: Wall-clock times for month-granularity queries

ure 3.14(a)). Note that although VERT-3.0 is optimized to access fewer postings than VERT-1.5, it performs slightly worse. This is possibly because VERT-3.0 has to access more number of partitions than VERT-1.5 for longer time-granularity queries due to its high degree of partitioning. In UKGOV, there is an improvement of almost 29.9% or 279.26 ms (CAS-1000 vs VERT-1.5, see Figure 3.13(b)) for year-granularity queries. The full-lifetime queries are faster by 931 ms or 21% (CAS-1000 vs VERT-1.5, see Figure 3.14(a) and 3.14(b)) which in absolute terms is a considerable difference. Thus, the first insight which we draw is that sharded posting list avoid wasted reads substantially as compared to VERT resulting in superior performance.



(a) Wikipedia

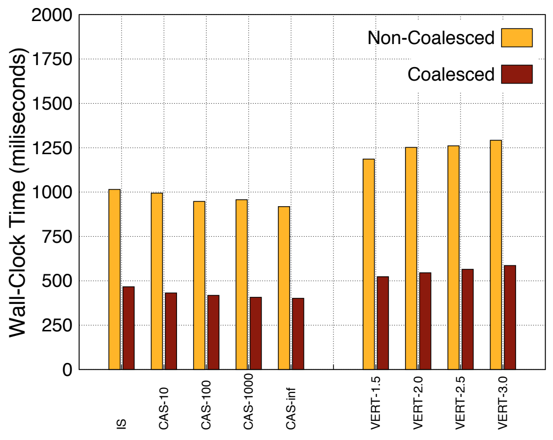


(b) UKGOV

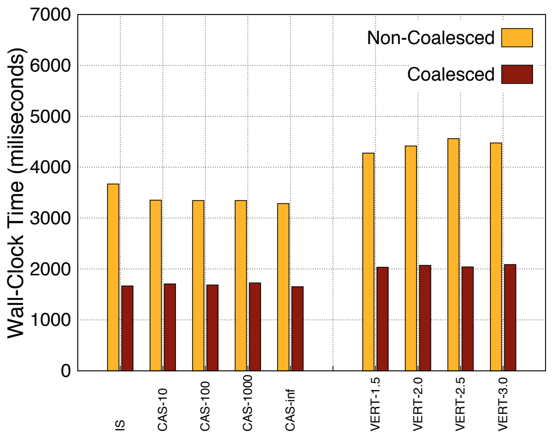
Figure 3.13.: Wall-clock times for year-granularity queries

3.10.2. Effect of Coalescing

Our experimental results with temporal coalescing of postings lead to similar results as our experiments on the original uncoalesced indexes. Independent of the partitioning/sharding used, the index sizes thus obtained are much smaller than with uncoalesced indexes—up to an order of magnitude for UKGOV and up to a factor of 2 for Wikipedia. Also the indexes created with sharding are always smaller than VERT (see Figure 3.15). The wall-clock times of CAS and VERT methods with temporal coalescing are depicted in Figures 3.11(a) through 3.14(b). It is evident that query performance improves with temporal coalescing, with longer time interval queries gaining more than those with smaller time ranges as more postings need to be read.



(a) Wikipedia



(b) UKGOV

Figure 3.14.: Wall-clock times for full-lifetime queries

3.10.3. Comparing Sharding Approaches

In the next set of results, we first present the improvements, in query-processing wall-clock times, due to reductions in the number of shards by carefully allowing for wasted reads regulated by the parameter η. Secondly, we compare the query performance of both sharding approaches CAS and INC.

To explain the query performance of the sharding methods, we first compare the average number of shards generated by both the algorithms for the terms present in the query workload. These results are summarized in Table 3.3. Interestingly, INC has lesser number of shards than CAS in UKGOV in-spite of being more restrictive. This means that the INC, with an approximation guarantee, seems to make better choices than the heuristic approach employed by CAS. On the contrary for WIKI the number of shards

		Sharding	
Dataset	η	INC	CAS
	IS	79.75	79.75
WIKI	10	33.37	32.59
	100	14.42	11.78
	1000	6.83	4.71
	IS	16.75	16.75
UKGOV	10	11.95	14.12
	100	8.05	11.67
	1000	5.49	5.84

Table 3.3.: Average number of shards per term

per term created by INC is more than that of CAS. However, in both these cases the difference between CAS and INC in the number of shards is not significant.

Effect of the Parameter η Depending on the value of the parameter η there are two conceptual extremes in sharding. The scenario $\eta = 0$ represents idealized sharding, denoted by IS, when wasted reads are completely avoided. Note that both CAS and INC would give rise to IS when $\eta = 0$. The other extreme is CAS-inf which results in no sharding of the posting list. cop

Idealized sharding is a restrictive form of sharding and for certain distributions produces a fairly high number of shards (see Table 3.3). Although query processing on IS results in reading only the postings intersecting with the query time-interval, they suffer from inefficiencies due to a large number of random seeks in accessing each shard individually. Especially for disk with $C_r >> C_s$, the open-seek operation on idealized shards might result in considerable overheads.

To put this into perspective with the actual query performance, we present the wall-clock times in Table 3.4. In WIKI, we see a consistent improvement from IS to CAS-1000. This is because idealized sharding admits a fairly large number of shards in this case and thus the I/O costs are dominated by initial random seeks to access these idealized shards. Improvements result from the reduction in the number of shards due to careful merging of idealized shards, as presented before in Section 3.5. Although these reductions might not be significant for queries with longer time intervals, but they reduce query processing time by a sizable fraction for small-time granularity queries – day-granularity queries improve by 62% (see Figure 3.11(a)) and month-granularity queries

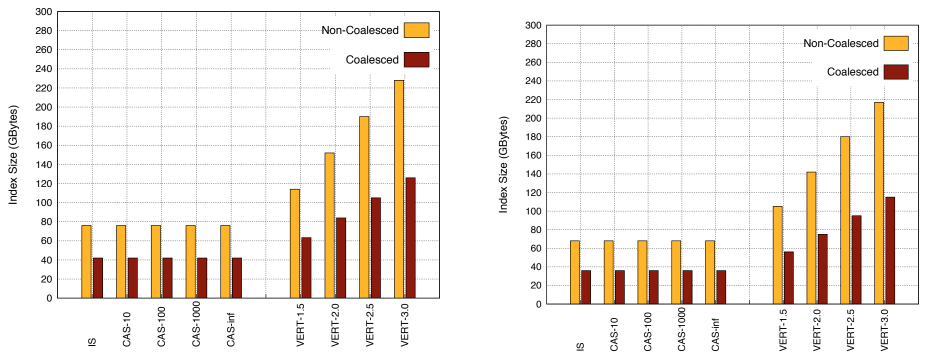
	$\eta = 10$		$\eta = 100$		$\eta = 1000$	
	CAS	INC	CAS	INC	CAS	INC
WIKI						
Day	17.64	15.81	11.57	10.65	11.86	8.70
Month	29.01	28.88	25.03	24.22	24.75	22.70
Year	131.06	132.08	127.96	125.32	127.57	123.76
Full	851.43	817.76	834.74	806.87	815.20	809.90
UKGOV						
Day	55.78	50.22	53.34	44.97	49.25	43.27
Month	158.89	155.72	156.28	145.77	153.89	143.60
Year	711.63	724.56	707.84	702.69	699.81	696.87
Full	2,875.35	2,839.80	2,794.46	2,845.31	2,940.5	2,999.32

Table 3.4.: Comparison of wall-clock times between CAS and INC – in milliseconds

by 35% (see Figure 3.12(a)). A similar trend is seen in the case of INC (see Table 3.4) where there is a 47% improvement in day-granularity queries in INC-1000 from INC-10 for WIKI. Unlike WIKI, the difference in performance between CAS and INC in UKGOV is not considerable, which is due to the already low number of initial idealized shards. This indicates that sharding can be applied as a self-organizing approach depending on the distribution of initial shards.

CAS-1000, and eventually INC-1000, outperforms CAS-inf by a fairly large margin in all query granularities except one scenario. This shows that the improvement in query performance is not only due to begin-time order of postings in the shards but a result of careful sharding of posting lists to avoid wasted reads. The only scenario when CAS-inf outperforms others is when we consider queries spanning the full-lifetime of the collection. This behavior is to be expected because all postings in CAS-inf become relevant for such kind of queries and have to be subsequently read.

Comparing CAS and INC As one can observe from the in Table 3.4, the sharded index generated using INC compares quite favourably with the CAS. This behaviour is consistent across all the granularities of temporal predicates for all values of η. In a small number of cases, the performance of INC is slightly better than that of CAS, and is never worse. Although we see a less shards in some scenarios, it is counteracted more number of wasted reads per shard for a given query. Hence, the small differences in the number of shards between CAS and INC do not to make a considerable difference in



(a) Wikipedia (b) UKGOV

Figure 3.15.: Index sizes

	$\eta = 10$		$\eta = 100$		$\eta = 1000$	
	CAS	INC	CAS	INC	CAS	INC
WIKI	76.31	78.50	76.29	77.60	76.18	77.20
UKGOV	68.31	68.50	68.28	68.47	68.18	68.46

Table 3.5.: Comparison between index sizes of CAS and INC - in gigabytes

the wall-clock times. Hence the arguments presented above comparing CAS and VERT also apply when comparing INC and VERT.

3.10.4. Index Sizes

As expected, the size of the index files of the sharded indexes is the same as that of the unpartitioned index. This is due to the fact that sharding partitions the postings of the unpartitioned lists in a disjoint manner. On the contrary the postings in VERT are subject to replication across the vertical partitions. The index sizes of the different variants of VERT show a direct correlation with the input parameter κ as shown in Figure 3.15(a). As discussed before, κ regulates the upper bound to the index-size blowup. The higher the κ, the more efficient is the performance of time-travel queries at the expense of a larger index size. Thus the vertically-partitioned index has to be carefully tuned trading off index size and query efficiency. This is not the case with the sharded index where the tradeoff is between number of random seeks and sequential reads, which are local tunable parameters depending on physical characteristics of disks (where the index is stored) irrespective of the query workload.

The difference in overall index sizes is due to the impact lists. Since each shard is associated with an impact list, the number of shards in an index is directly correlated with size of the impact-list file. Thus, the file which store the impact lists decrease with the increasing η. Each impact list is stored as list of pairs of integers without compression. From our experiments we see that the impact-list file is typically 1%-7% of the entire index. There is, however, scope for compactly representing the impact lists using integer compression over d-gaps when the key and values are stored in integer array separately.

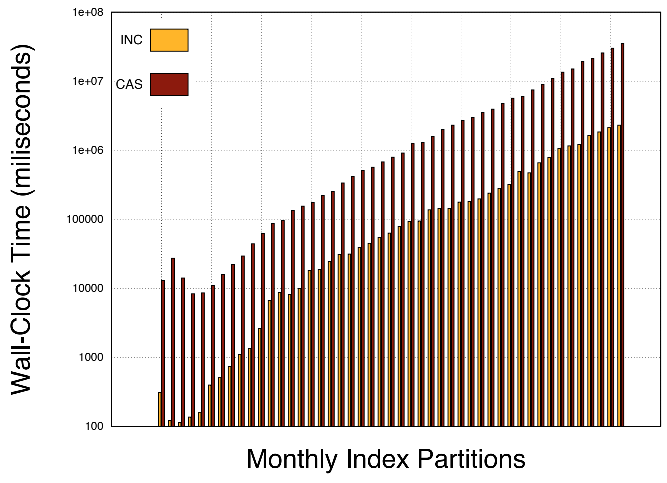
From our experiments, we observe that the time taken to build a sharded index is roughly twice the time taken for the standard unpartitioned inverted index. Since the sharded index building process can be easily parallelized, one can efficiently build sharded indexes using a distributed processing platform (e.g., Hadoop).

3.10.5. Index Maintenance Performance

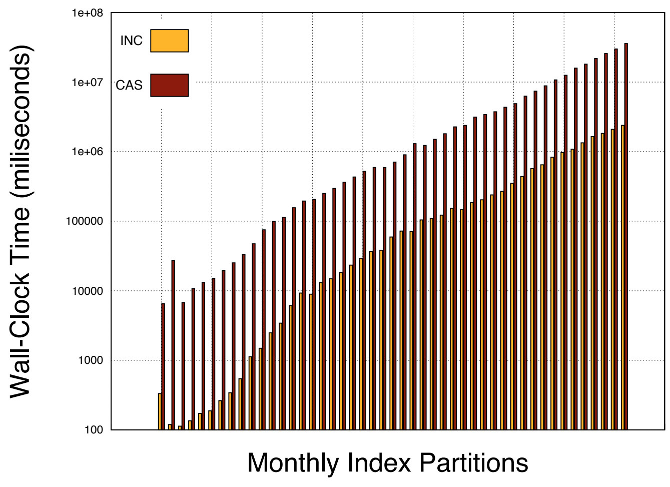
As we introduced in Section 3.8, the archive index is maintained by first collecting the updates to a partial-archive index which is then periodically merged into the primary archive index. A partial index is responsible for all versions which end in the time interval between two consecutive merges. To simulate index management for archive indexes as follows: we first created partial indexes for each month containing postings of only those versions that have end time within that month. The index is incrementally maintained, starting from an empty index, by merging partial indexes of each month in sequence. We employed *immediate-merging* [BCC10] to create one consolidated index at the end of every monthly merge operation, and each sharded posting list in the index is incrementally maintained using our incremental sharding algorithm. We compared this with CAS, which recomputes the entire sharding for the combined index every time from scratch. In other words, all shards of a posting list in the currently merged index are read and decompressed, the corresponding list from the partial index for the next month is also read and decompressed, these two are combined, and finally, a new sharding is generated using the CAS algorithm for the merged index, which is finally written in compressed form to disk.

Figures 3.16 and 3.17 show results of our experiments on index maintenance with WIKI and UKGOV datasets respectively. Note the log-scale used on the y-axis, which represents the time-taken for the consolidated sharded index to be built in milliseconds.

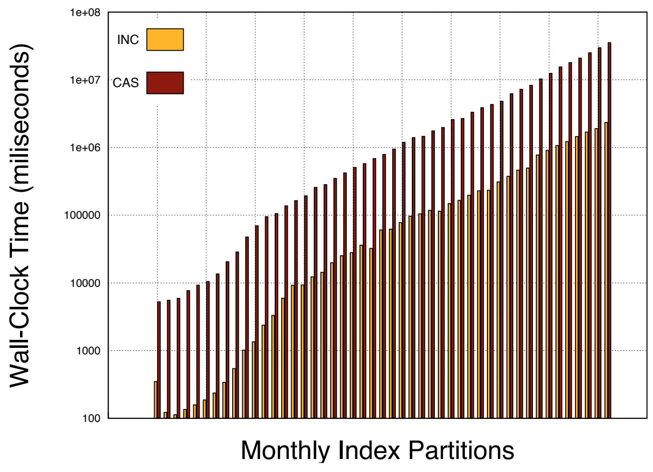
It is evident from these charts that INC outperforms CAS by a large margin. In UKGOV, the improvements are nearly a factor of 4 (see Figure 3.17) while the improvements in WIKI are around a factor of 10 (see Figure 3.16). This efficiency comes from two advantages that INC enjoys over CAS: first, recomputing the sharding by CA on the merged index takes much of the time and grows as the index size grows. Since INC does not recompute the sharding, its performance improves significantly. Second,



(a) η = 10

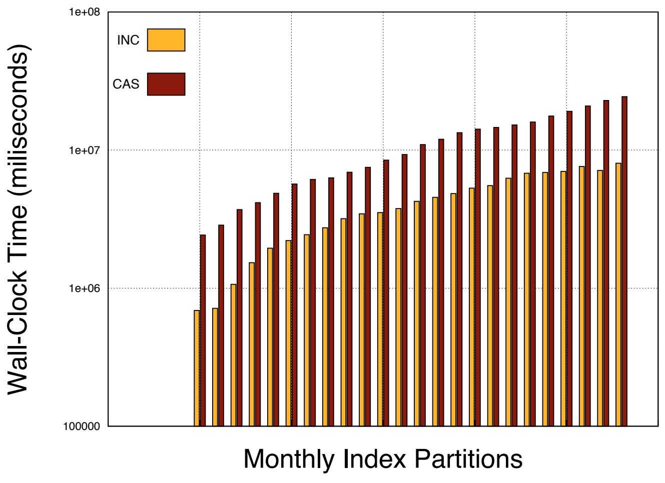


(b) η = 100

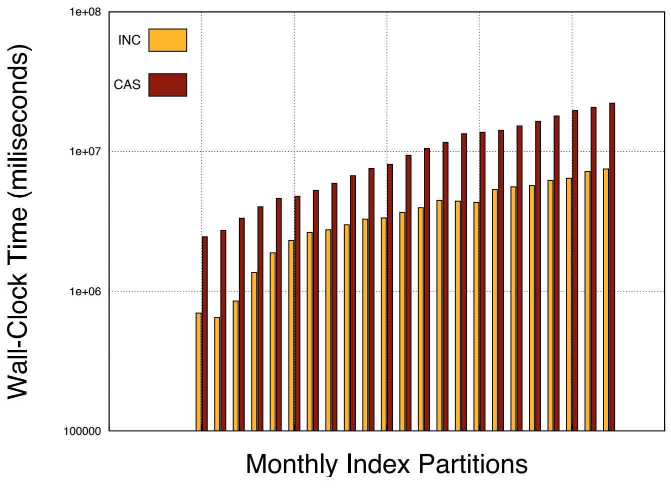


(c) η = 1000

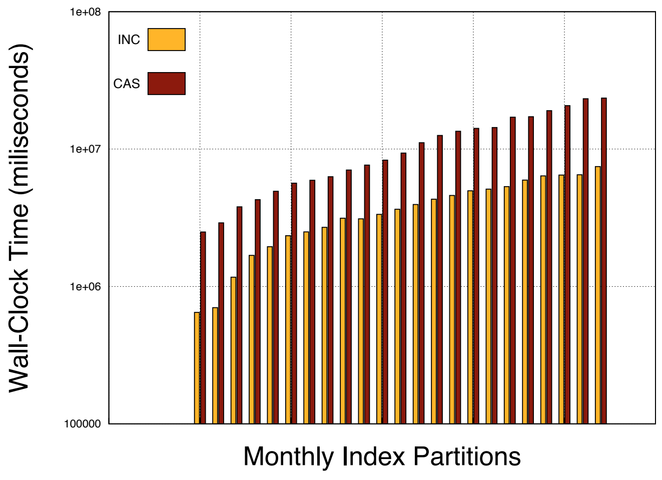
Figure 3.16.: Performance of index maintenance - WIKI



(a) η = 10



(b) η = 100



(c) η = 1000

Figure 3.17.: Performance of index maintenance – UKGOV

it does not have to decompress, merge and shard the entire list before writing do the disk. Instead, it has to just read two posting lists in parallel and append corresponding shards (without decompressing), and write to the disk. From our experiments, we observe that recomputation of shards accounts for an average of 55%-60% of the entire maintenance time. Compression and decompression take upto 15% of the overall time but since we use 7-bit encoding for compression we expect that a more involved compression technique would only increase the maintenance time. It should be noted that in our simulation we do not perform an append using in-place merge techniques. Instead we resort to creating a new index file in each step incurring additional overheads. Thus, the performance of INC can be further improved by carefully implementing advanced index merging methods.

3.11. Related Work

Temporal information associated with documents has recently seen increasing attention in information retrieval. One of the earliest known efforts in this direction is by Anick and Flynn [AF92] who developed a framework for versioning the complete index for historical queries. Recently, Alonso et al. [AGBY07] give an overview of relevant research directions. The work by Herscovici et al. [HLY07] focuses on exploiting the redundancy commonly seen in versioned documents to compress the inverted index. Similarly, He et al. [HYS09, HZS10] consider the problem of efficiently storing inverted indexes on disk using compression; these are orthogonal to our work and could be combined with our sharding techniques.

The closest to our work and the most relevant related work is the work on vertical partitioning of posting lists by Berberich et. al. [BBNW07]. They consider posting-list partitioning strategies which trade-off index size and query-processing performance. Two approaches employed by them either bound the index size, called the *space-bound approach*, or bound the worst performance called the *performance-guarantee approach*. Further, they also introduced index compression techniques called temporal coalescing aimed at supporting different query types while keeping the index compact. These compression techniques are also relevant in our setting as shown in our experiments in Section 3.10.2.

Research in temporal databases has taken a broader perspective beyond text documents and targeted general class of time-annotated data. Index structures tailored to such data like the Multi-Version B-Tree [BGO$^+$96] or LHAM [MOPW00] are related to our work, since they also, implicitly or explicitly, rely on a temporal partitioning and replication of data. It is therefore conceivable to apply our proposed techniques in conjunction with one of these index structures.

Work on index maintenance can be categorized into (a) work on maintaining inverted indexes when faced with changes in the document collection and (b) approaches that make search aware of temporal information associated with documents. No work, to the best of our knowledge, has been done at the intersection of the two categories.

Given that the construction of inverted indexes is well understood and can easily be parallelized, one existing maintenance strategy has been to rebuild the index periodically. Returning stale query results, most of the time, is an obvious disadvantage of this approach. For a long time, though, this has been the approach taken by major search engines on the Web. Only lately, Peng et al. [PD10] have addressed the issue of handling updates in web-scale indexes. Instead of rebuilding the index, Lester et al. [LZW06] suggest to collect updates in an in-memory index that is then merged, from time to time to amortize costs, with a disk-resident inverted index. This merge can either be performed in-situ, thus modifying posting lists at their current location, or by storing entirely new posting lists. A hybrid approach that chooses between these alternatives is described by Büttcher et al. [BC08]. Guarajada and Kumar [GK09] can be seen as another extension that leverages query logs to determine terms whose posting lists mandate eager maintenance. In a spirit similar to log-structured methods [OCGO96], Lester et al. [LMZ08] propose to keep multiple indexes of geometrically increasing size and merge them, when they overflow, in a rolling manner. Query results reflecting the current state of the document collection can be obtained in these approaches by executing queries both on in-memory and disk-resident indexes. For more detailed discussions of inverted index maintenance we refer the reader to Chapter 2.

3.12. Summary

This chapter presents a novel method of index organization based on sharding for processing time-travel queries efficiently. Previous approaches traded-off index size and query performance resulting in an index-size blowup. We take an alternative approach of index partitioning by exploiting the valid-intervals and taking into account index-access costs. The resulting index has a small space overhead and we show by experiments that the sharded indexes consistently outperform the vertically-partitioned index in query performance. We also introduce index-maintenance strategies, based on incremental sharding, which avoid expensive shard recomputations when dealing with dynamic collections. We show through experiments that, by employing incremental sharding, we outperform sharding based on recomputation by at least four times.

4 Query Optimization for Approximate Processing of Time-Travel Queries

4.1. Introduction

4.1.1. Motivation and Problem Statement

Text search is an expensive operation on web archives due to their large scale. However, many of the documents in the archive contain redundant information. In certain scenarios, users are often satisfied with a subset of the true results that are determined *quickly*. That is, missing a few results during search might not lead to substantial information loss. Particularly in search tasks which involve multiple interactive steps of query reformulation, expansion, and refinement. Consider the following two use cases in the context of time-travel text search.

- A sports journalist is in interested the game between the popular cricket teams, from Mumbai and Rajasthan, and issues a query "indians vs royals" @ `[6/2008]`. However content from the popular baseball teams from Cleveland and Kansas, also having the same titles, which also were in the news in the same period might dilute the results. For an interactive and constructive search experience the analyst should be able to quickly identify the ambiguity of the keywords used and alter the query to "mumbai indians vs rajasthan royals" @ `[6/2008]`. Here a subset of the exact results is good enough for the user to adapt "indians vs royals" to "mumbai indians vs rajasthan royals".

- Partial results can also help reformulating for the correct temporal predicates of the query. Consider the utility of time-travel text search in patent retrieval. The time-interval of a query "retina display patent" @ `[1/2010 – 12/2012]` can quickly be reformulated to "retina display patent" @ `[6/2012 – 6/2013]` when the user observes that the initial results cluster around the second half of the year 2012.

In such scenarios, a subset of the results is often representative of the information contained in the entire result set. Increasing the number of elements in this subset, or the recall, improves the information contained therein. Also, it is imperative that the user needs to have the ability to control the performance of the time-travel retrieval task. To this extent, we develop query processing techniques to maximize recall, for a time-travel query, given a user-specified bound on the performance.

4.1.2. Approach

In the previous chapters we discussed different index-organization strategies for web archives with the focus on *exact* solutions to time-travel queries. We showed that horizontal partitioning is an efficient index-organization strategy for exact time-travel queries. In this chapter, we argue that, for approximate processing of time-travel queries vertical partitioning is more suitable. This is due to the fact that vertical partitions are clustered in the time dimension.

By identifying the time periods which have high result contribution, and in turn the partitions which temporally overlap with it, the recall can be increased by processing such partitions early on. In contrast, a lack of such clustering in horizontal partitioning or sharding prevents easy identification of shards which have high result contribution. Consequently, we operate on an index with vertically-partitioned posting lists and propose query-optimization methods to determine partial results efficiently.

A natural side effect of vertical partitioning is that postings with long valid-time intervals are replicated across several temporally-adjacent partitions. This is not an issue for time-point queries where only one partition per term is accessed during query processing. For the more practical and general class of time-travel (text) queries where the temporal predicate is a *time interval*, the straightforward query processing of [BBNW07] quickly becomes inefficient as a consequence of this replication, as repeatedly reading replicated postings from several partitions within the query time-interval wastes I/O operations.

We introduce an approach called *partition selection* which exploits the following observation: if we can determine that a partition largely consists of postings that are replicated in already processed partition(s), we can avoid processing this partition without significantly compromising the final result quality. We aim at *selecting a set of partitions* which can be processed incurring no more than a given maximal processing cost and that yield high recall. We consider abstract cost measures for processing a query, namely the number of partitions or the number of postings accessed during execution. A user would specify bounds on the execution time that can be transformed into bounds on the abstract cost by the system. Alternatively, the user can also stop the processing at

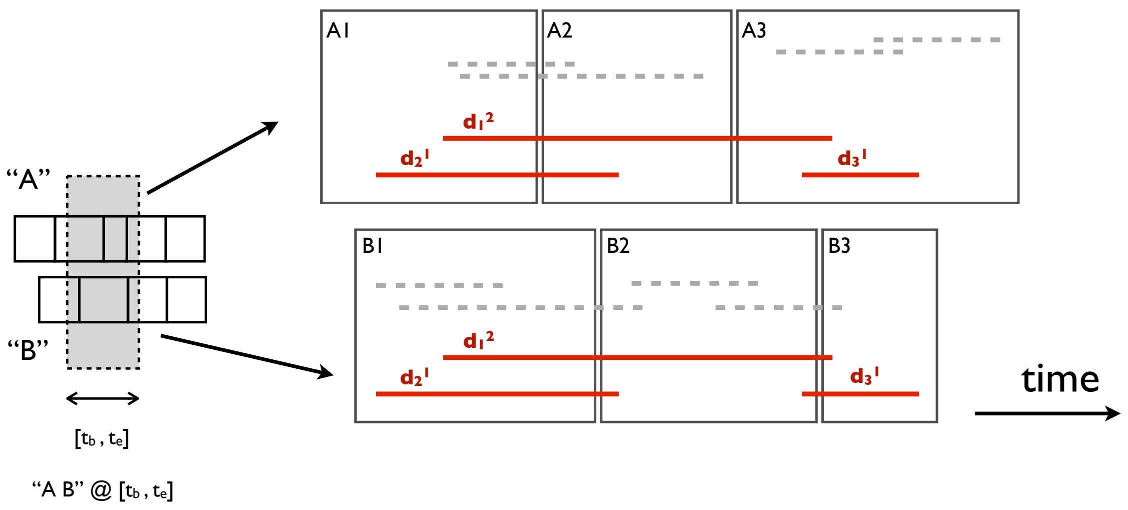


Figure 4.1.: Processing a time-travel query "A B" @ $[t_b\ , t_e]$ using partition selection

any time when she determines that the results are already satisfying (or the query needs to be refined); our methods support this by selecting partitions first that are likely to contain many unseen answers.

We use the example in Figure 4.1 to illustrate the general idea of our approach. This figure shows posting lists for two terms A and B built over documents with valid-time intervals. On the left side of the figure, posting lists are shown, each spanning the entire time interval. The region, shaded in gray, represents a temporal predicate that spans a small time range over these lists. In the absence of temporal partitioning, query processing needs to entirely scan both lists and filter out postings that do not satisfy the temporal predicate. When the index is partitioned, however, the processing can be sped up by reading only the relevant partitions that overlap with the temporal predicate, represented on the right. Thus, in our example, a total of 6 partitions – A_1, A_2, A_3 and B_1, B_2, B_3 – have to be processed to determine the result set of $\{d_1, d_2, d_3\}$ – marked as red line-segments.

However, a closer inspection of Figure 4.1 reveals that the same result set can be obtained by processing only *2 partitions*, A_2 and B_2, since the replicas of postings for documents in the result set are fully available within these two partitions.

How can we make use of this observation in practice ? In order to do so, we need to answer the following questions: (i) does a partition contribute non-redundantly towards the final result set when it is processed ?, (ii) how much does a partition contribute to the final result set ? (iii) is there an alternative set of partitions which can contribute these answers at a lower access cost ? Based on answers to these questions, we can generate a partition-access plan so that for a specified cost budget only those partitions are chosen for processing which maximize the number of results.

4.1.3. Contributions

We formally model these *partition selection* problems as optimization problems. Making use of KMV synopses for cardinality estimation under set operations [BHR+07], we develop algorithms for efficiently solving such partition-selection problems. In particular, the contributions made in this chapter are:

1. An optimal dynamic programming based partition selection algorithm for single-keyword queries;
2. An efficient greedy alternative for partition selection that can be applied for both single keyword as well as multi-keyword queries;
3. A detailed experimental evaluation on three large-scale real-world text archives: the revision history of the English Wikipedia, a Web archive, and the Annotated New York Times archive spanning 20 years.

4.1.4. Organization

The remainder of this chapter is organized as follows: in Section 4.2, we present index organization and query processing in our setup. In Section 4.3, we detail an optimal algorithm and its greedy approximation for selecting the set of partitions to process for the case of single-keyword queries. Extensions for multi-term query setting are described in Section 4.4. Our experimental setup and results are detailed in Section 4.7 before summarizing in Section 4.9.

4.2. Index Organization and Query Processing

We adopt the document, collection and query models from the previous chapters (c.f. Section 3.2 and Section 2.3.3). We briefly recap the notation in Table 4.1.

Index Organization We use a temporally-partitioned index as described in Chapter 2 with the following posting structure

$$\langle d_i^k, [\mathtt{begin}(d_i^k), \mathtt{end}(d_i^k))\rangle.$$

d_i^k refers to the version identifier d_i^k, $[\mathtt{begin}(d_i^k), \mathtt{end}(d_i^k))$ is its valid-time interval.

The temporally-partitioned index consists of posting lists vertically partitioned into a set of partitions. We let $\mathtt{partitions}(v)$ denote the set of partitions of the posting list

Notation	*Description*
$\mathcal{D}$	collection of documents
$\mathcal{V}$	*vocabulary* as a set of words
$d_i^j \in \mathcal{D}$	the j^{th} version of document d_i
$\mathtt{valid}(d_i^j)$	valid-time interval
$\mathtt{begin}(d_i^j)$	begin-time of d_i^j
$\mathtt{end}(d_i^j)$	end-time of d_i^j
Q	time-travel query
$\mathtt{keywords}(Q)$	set of keywords in Q
$\mathtt{interval}(Q)$	query time-interval of Q
$I_1 \sqcap I_2$	Overlapping intervals I_1 and I_2
$I_1 \not\sqcap I_2$	Non-overlapping intervals I_1 and I_2

Table 4.1.: Notation.

L_v for term $v \in \mathcal{V}$. Each partition $\phi_{v,j} \in \mathtt{partitions}(v)$ has an associated time-interval $\mathtt{span}(\phi_{v,j}) = [\mathtt{begin}(\phi_{v,j}), \mathtt{end}(\phi_{v,j}))$ and stores postings representing document versions whose valid-time intervals overlap with $\mathtt{span}(\phi_{v,j})$, i.e.,

$$\phi_{v,j} = \left\{ d_i^k \in \mathcal{D} \mid v \in d_i^k \wedge \mathtt{valid}(d_i^k) \sqcap \mathtt{span}(\phi_{v,j}) \right\} .$$

Further, we assume that partition spans for a given term v are disjoint, i.e.,

$$\forall i \ \forall j \ : \ \mathtt{span}(\phi_{v,i}) \ \not\sqcap \ \mathtt{span}(\phi_{v,j}) \ .$$

A lexicon $\mathcal{L}$ stores this partitioning information as a mapping from term to the partition statistics (partition span, partition size and location).

For a pair of partitions $\phi_{v,i}$ and $\phi_{v,i+k}$, or simply ϕ_i and ϕ_{i+k}, $\phi_i \cap \phi_{i+k}$ represents the set of postings common to both the partitions. The overlap of valid-time intervals and partition spans gives rise to the time-continuity property as described below.

Lemma 4.1 (Continuity Property in Vertical Partitions) *For a set of partitions belonging to a term v, the overlaps of contents of partition ϕ_i with ϕ_{i+k}, $\forall k \geq 0$ and $0 \leq j \leq k$ have the following property:*

$$\phi_i \cap \phi_{i+k} \subseteq \phi_i \cap \phi_{i+j}$$

Proof: *Every posting $p \in \phi_i \cap \phi_{i+k}$ represents a version d_k^t for which $\mathtt{begin}(d_k^t) < \mathtt{end}(\phi_i)$ and $\mathtt{end}(d_k^t) > \mathtt{begin}(\phi_{i+k})$. This implies that*

$$\mathtt{valid}(d_k^t) \sqcap \mathtt{span}(\phi_{i+k})$$

and hence the claim holds. □

Query Processing In this work we use conjunctive query semantics, i.e., we identify as results documents versions that contain all query terms. We employ the Term-at-a-Time (TAAT) approach of posting-list intersection during query processing. First, the partitions overlapping with the query time-interval are determined by consulting the lexicon. We refer to these partitions as *affected partitions*.

Definition 4.1 (Affected Partitions) *For a time-travel query Q, the set of partitions of the term* $q_i \in \mathtt{keywords}(Q)$ *which overlap with the query-time interval* $\mathtt{interval}(Q)$ *are*

$$a(q_i, Q) = \{ \phi_{i,j} \mid \mathtt{interval}(Q) \sqcap \mathtt{span}(\phi_{i,j}) \} .$$

When it is clear from the context we use a_i for $a(q_i, Q)$. The overall set of affected partitions is $A(Q) = \bigcup_{q_i \in \mathtt{keywords}(Q)} a_i$. The determined a_is are merged to determine a candidate set of postings which overlap with $\mathtt{interval}(Q)$. Following that, these candidate sets are intersected employing TAAT posting-list intersection, for the final set of results.

For approximate processing of time-travel queries we additionally go through a query-optimization phase prior to query processing - referred to as *partition selection*. Partition selection determines a subset $\mathcal{S} \subseteq A(Q)$ of all the affected partitions for query processing. Query processing over $\mathcal{S}$ yields partial results. We formalize the notion of a result set $R(\mathcal{S}, Q)$ given a time-travel query Q over a subset of affected partitions $\mathcal{S} \subseteq A(Q)$.

$$R(\mathcal{S}, Q) = \begin{cases} \left\{ \bigcup_{\phi_j \in \mathcal{S}} \phi_j \right\} & : |Q| = 1 \\ \left\{ \bigcap_{1 \le i \le |Q|} \bigcup_{\phi_{i,j} \in \mathcal{S}} \phi_{i,j} \right\} & : |Q| > 1. \end{cases} \tag{4.1}$$

Using the notation above, the exact set of results is captured hence by $R(A(Q), Q)$. The case when $|Q| = 1$ refers to the scenario when the query contains a single keyword. In such a case all postings accessed are relevant and are captured by the union operation over all affected partitions. However, when $|Q| > 1$, only those postings which are common to all the terms are relevant. This is captured by the intersection over unions. As an example, we refer to Figure 4.1. For a query "A B" @ `[`t_b`, `t_e`]` the document versions, colored in red, are retrieved as results. However, for a single-term query, say "A" @ `[`t_b`, `t_e`]`, all document versions index in "A" get qualify as results.

To quantify how much of the exact results are retrieved using $\mathcal{S}$ we use relative recall. The relative recall for $\mathcal{S} \subseteq A(Q)$, $RR(\mathcal{S}, Q)$, is defined as the ratio of the number of retrieved results to that of the exact results, i.e., fraction of results retrieved. Formally,

$$RR(\mathcal{S}, Q) = |R(\mathcal{S}, Q)| / |R(A(Q), Q)|. \tag{4.2}$$

The intention of partition selection is to choose $\mathcal{S}$ in order to maximize $RR(\mathcal{S}, Q)$.

Use of Partition Synopsis Partition-selection algorithms, as we discuss in detail later, use cardinality values of set operations (unions and intersections) on partitions as primitive operations. However, to determine the exact cardinalities the partitions have to be accessed which is exactly what we want to avoid. Instead, we depend on high quality cardinality estimates under union and intersection of large sets of document identifiers. For this purpose, we utilize the recently proposed KMV synopses [BHR+07]. In a pre-computation step, we build and store the synopsis for each partition on disk of the temporally-partitioned index, which we use during our partition-selection process. Pointers to each partition-synopsis can be either stored in the existing lexicon or an altogether separate lexicon can be constructed explicitly for this task.

During partition selection, the lexicon storing pointers to partition synopses is consulted followed by retrieval of the synopsis of each affected partition. These highly compact partition synopses are used by our selection algorithms to determine cardinalities of set operations over partitions. In the next section, we first detail partition-selection methods for queries with only single terms. We then generalize these approaches to multi-term queries.

4.3. Single-Term Partition Selection

Let us consider the special case where the time-travel keyword query Q consists of only a single query term, i.e., $\mathsf{keywords}(Q) = \{ q \}$. Since we deal with single-term queries, we denote $\phi_{q,j}$ as ϕ_j for ease of notation from now on.

Our objective, when selecting partitions to process the time-travel keyword query, is to retrieve as many of the original query results as possible, while not violating a user-specified I/O bound. Our optimization objective, to put it differently, is to maximize the *relative recall* as the fraction of original query results retrieved. The user-specified I/O bound, which constrains the space of valid solutions, can be of two types – *size-based partition selection* or *equi-cost partition selection*.

In both selection problems we model the performance to be proportional to the index accesses. Hence a bound on the accesses simulates a bound on the response time for a time-travel query. In case of *size-based partition selection* we bound the number of postings accessed. A similar analysis is undertaken for most query-processing methods over posting lists.

However, we also take a more coarse-grained approach in modelling the constraint on accesses in *equi-cost partition selection*. Accesses to each partition results in a random access and random I/O are substantially more expensive than sequential I/O. Also, partitions tend to be smaller than entire posting lists. Hence the bottleneck in answering a time-travel query in such scenarios is the number of distinct partitions that are accessed. To this extent, in *equi-cost partition selection*, we model the allowable budget to be accessed as a fixed number of affected partitions that can be accessed rather than number of postings.

Size-Based Partition Selection The input to this optimization problem is the set of affected partitions $A(Q)$, and a user-specified I/O bound β where $0 < \beta \leq 1$. Here, β denotes the fraction of postings of all affected partitions that we are allowed to read. The cardinality of each partition $|\phi_j|$ represents the number of postings accessed on selecting the $|\phi_j|$ for processing. Note that maximizing result size $\left|\bigcup_{\phi_j \in \mathcal{S}} \phi_j\right|$ is equivalent to maximizing relative recall since the original result size $\left|\bigcup_{\phi_j \in A(Q)} \phi_j\right|$ is constant. Thus, we intend to determine $\mathcal{S} \subseteq A(Q)$ so as to maximize the result size while retaining our I/O budget. Formally,

Definition 4.2 (Size-Based Selection for Single-Term Queries)

$$\underset{\mathcal{S} \subseteq \phi}{\operatorname{argmax}} \left| \bigcup_{\phi_j \in \mathcal{S}} \phi_j \right| \quad \textit{s.t.}$$

$$\sum_{\phi_j \in \mathcal{S}} |\phi_j| \;\leq\; \beta \cdot \left(\sum_{\phi_i \in A(Q)} |\phi_i| \right).$$

Equi-Cost Partition Selection The inputs to this problem is the same as before – ϕ_j, and bound β. However, we assume that all partitions are equally expensive to process irrespective of their sizes. The constraint is now bound to a fixed number of partitions that can be accessed $\beta \cdot |\phi|$. The objective function remains the same as before. Formally,

Definition 4.3 (Equi-Cost Selection for Single-Term Queries)

$$\underset{\mathcal{S} \subseteq \phi}{\operatorname{argmax}} \left| \bigcup_{\phi_j \in \mathcal{S}} \phi_j \right| \quad \textit{s.t.}$$

$$|\mathcal{S}| \;\leq\; \beta \cdot |A(Q)|.$$

4.3.1. Optimal Algorithm for Single-Term Partition Selection

The above problems can be solved using dynamic programming over an increasing number of affected partitions. We first present a solution to the more general size-based partition selection. Our solution to equi-cost partition selection follows from this, as we show later.

We process the affected partitions $A(Q)$ in the order of their begin times, i.e.,

$$\mathtt{begin}(\phi_{v,j}) < \mathtt{begin}(\phi_{v,j+1}).$$

Consider a prefix sub problem which considers affected partitions $\{\phi_1, \cdots, \phi_k\}$, and capacity $c = \beta \cdot (\sum_j |\phi_j|)$. Let $\textsc{Opt}(c,k)$ denote the optimal set of partitions for the prefix sub-problem with capacity c, the optimal result-set size hence is $R(\textsc{Opt}(c,k))$. The optimal solution is computed by the following recurrence on the constituent sub-problems.

$$R\ (\ \textsc{Opt}(c,k)\ ) = \ max \begin{cases} R\ (\ \textsc{Opt}(c, k-1)\ ) \\ \max\limits_{0<k'<k} R\ (\ \textsc{Opt}\ (c - |\phi_k|\ ,\ k'\ )\ \bigcup \{\phi_k\}\ ) \end{cases}$$

We now state and prove the optimality of the recurrence in Theorem 4.1.

Theorem 4.1 *Given a query* Q *and access budget of capacity* $c = \beta \cdot (\sum_{\phi_j \in A(Q)} |\phi_j|)$ *the optimal solution to the size-based selection problem is given by* $\textsc{Opt}(c, |A(Q)|)$.

Proof: *Assume that we have optimal solutions for all sub-problems by the given recurrence,* $\textsc{Opt}(c', k')$, *with capacities* c' *such that* $0 < c' < c$, *for the set of partitions* $\{\phi_i\}$ *where*

$$0 < k' < k.$$

Now we consider computing the optimal selection set by the recurrence $\textsc{Opt}(c,k)$ *using* $\textsc{Opt}(c',k')$*'s. Let us assume that there is a better solution* $\overline{\textsc{Opt}}(c,k)$ *such that*

$$R(\overline{\textsc{Opt}}(c,k)) > R(\textsc{Opt}(c,k)).$$

Case 1 – $\phi_k \notin \textbf{Opt}(c,k)$ **:** *From the recurrence this means*

$$R(\overline{\textsc{Opt}}(c,k)) > R(\textsc{Opt}(c,k-1))$$

since $R(\textsc{Opt}(c,k)) = R(\textsc{Opt}(c,k-1))$ *when* $\phi_k \notin \textsc{Opt}(c,k)$. *As a consequence, we have a new optimal solution for the sub-problem for capacity* c *and the set of partitions* $\{\phi_i\}$ *where* $0 < i < k$, *i.e,* $R(\overline{\textsc{Opt}}(c,k-1)) > R(\textsc{Opt}(c,k-1))$. *But, this is contrary to our initial assumption since we assumed* $R(\textsc{Opt}(c,k-1))$ *is optimal for* $\{\phi_i\}$ *where* $0 < i < k$. *Thus, by contradiction our claim holds.*

Case 2 – $\phi_k \in \textbf{Opt}(c,k)$ **:** *The second relation in the recurrence is applicable now. Our assumption leads to the condition* $R(\overline{\textsc{Opt}}(c,k)) > \max\limits_{0<k'<k} R\ (\ \textsc{Opt}\ (c - |\phi_k|\ ,\ k'\ )\ \bigcup \{\phi_k\}\ )$.

Let us denote the index of the partition selected just before ϕ_k *for* $\overline{\text{OPT}}(c,k)$ *to be* k'*, i.e., for* $k' = \operatorname{argmax} \overline{\text{OPT}}(c,k) \backslash \{\phi_k\}$. *According to our assumption we have*

$$\begin{aligned} R(\overline{\text{OPT}}(c,k)) - |\phi_k \cap \phi_{k'}| &> \left(\max_{0<k'<k} R(\text{OPT}(c,k))\right) - |\phi_k \cap \phi_{k'}| \\ \implies R(\overline{\text{OPT}}(c - |\phi_k|, k')) &> R(\text{OPT}(c,k')) - |\phi_k \cap \phi_{k'}| \\ \implies R(\overline{\text{OPT}}(c - |\phi_k|, k')) &> R(\text{OPT}(c - |\phi_k|, k')) \end{aligned}$$

This is a contradiction since $\text{OPT}(c - |\phi_k|, k')$ *is optimal for all* $0 < c' < c$. *Hence our claim in the theorem holds true.* $\square$

Algorithm 4: Partition Selection - dynamic programming solution

```
1:  $c_{max} = \lfloor \beta \cdot (\sum_j |\phi_j|) \rfloor$
2:  // Dynamic programming table, $n$ is number of affected partitions
3:  $DP[0..c_{max}][0..n]$
4:
5:  for $i = 0..c_{max}$ do
6:      $DP[i][0] = \emptyset$
7:  end for
8:
9:  for $k = 1..n$ do
10:     for $i = 0..|\phi_k| - 1$ do
11:         $DP[i][k] = \emptyset$    // No partitioning possible
12:     end for
13:     for $i = |\phi_k|..c_{max}$ do
14:         for $k' = 0..k-1$ do
15:             // Update if recall is better than current value
16:             $r_{k'} = DP_r[i - |\phi_k|][k'] + (|\phi_k| - (\phi_k \cap DP_{lp}[i - |\phi_k|][k']))$
17:         end for
18:         $k' = \operatorname{argmax} r_{k'}$
19:         // Update the DP table with the best partitioning
20:         $DP_r[i][t] = \max\{DP_r[c_j][k_i - 1], r_{k'}\}$
21:         $DP_{lp}[i][t] = \operatorname{argmax} DP_r[i][t]$
22:     end for
23: end for
24:
25: return $DP[c_{max}][n]$
```

Algorithm 4 efficiently implements the recurrence relation presented above. Each of the DP table cell contains a pair of values – (i) the last partition selected, DP_{lp}, for the corresponding sub-problem (i.e., the selected partition with the maximum begin-time), and (ii) the optimal recall value DP_r. Since the choice of partitions cannot be made independently, the computation of recall for a newly selected partition takes into account only the postings that are not already included in previously selected partitions. Using Lemma 4.1, we can efficiently compute the optimal recall for each sub-problem since all overlaps with the preceding partitions to lp are already covered in lp.

The DP-based algorithm, outlined in Algorithm 4, has a time complexity $\mathcal{O}(n^2 \cdot (\sum_j |\phi_j|))$, where n is the number of affected partitions, and a space complexity of $\mathcal{O}(n \cdot (\sum_j |\phi_j|))$. The optimal partitioning can be easily computed by backtracking from the best solution seen at $DP_{lp}[c_{max}][n]$. Observe that the complexities depends on the cardinalities of the partitions, i.e.,$\sum_j |\phi_j|$). Thus, Algorithm 4 is a pseudo-polynomial algorithm, for size-based partition selection, which is polynomial in the value of the size bound.

Equi-cost partition selection is a special case of size-based selection and employs a uniform cost per partition. The recurrence relation for equi-cost selection is

$$R\,(\,\text{OPT}(c,k)\,) = \max \begin{cases} R\,(\,\text{OPT}(c,k-1)\,) \\ \max\limits_{0<k'<k} R\,(\,\text{OPT}\,(c-1\,,\,k'\,)\,\bigcup\,\{\phi_k\}\,) \end{cases}$$

This results in improvements in both the space and time complexity of the algorithm, as the optimal value is independent of the sum of the sizes of the partitions read. The time complexity of the algorithm reduces to be $\mathcal{O}(n^3)$ and a space complexity of $\mathcal{O}(n)$. Note that unlike the algorithm for size-based selection, the algorithm for equi-cost selection is a polynomial algorithm which is polynomial in the number of partitions affected.

4.3.2. Approximation Algorithm

While the selection algorithms outlined above allow for polynomial run times, they might not be efficient enough to be applied during query processing, e.g., when the partitions contain a large number of postings. Alternatively, we propose the use of $(1-\frac{1}{e})$-approximation algorithm called *GreedySelect*, developed in [KMN99] for solving *budgeted maximum coverage* (BMC) problem. We first show the equivalence of our partition selection problem and the BMC problem.

Definition 4.4 (Budgeted Maximum Coverage) *A collection of sets $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m\}$ with associated costs $\{c_i\}$ is defined over a domain of elements $X = \{x_1, x_2, \ldots, x_n\}$ with associated weights $\{w_i\}$. The goal is to find a collection of sets $\mathcal{S}' \subseteq \mathcal{S}$, such that the total cost of the elements in $\mathcal{S}'$ does not exceed a given budget L, and the total weight of every element covered by $\mathcal{S}'$ is maximized.*

Lemma 4.2 *Partition selection is an instance of budgeted maximum coverage (BMC).*

Proof: *The selection problem for single terms can be cast into an instance of the BMC problem [KMN99] in the following way: The affected partitions, ϕ_js, are the analogous to the* sets *in the BMC problem with the postings in the partitions being the* elements *of the respective* set. *For size-based partition selection, the cost for each set is its cardinality; for equi-cost selection, the cost for each set is unity. The cost budget is exactly the I/O bound c_{max}. With this reduction we can use the approximation algorithm proposed by Khuller et al. [KMN99] which has a constant factor approximation guarantee of* $(1 - \frac{1}{e})$. □

Algorithm 5: GREEDYSELECT for single-term partition selection

```
 1: input: $c_{max}$ , $\phi$
 2: $\mathcal{S} = \emptyset$
 3: $\mathcal{A} = \phi$
 4: $C = 0$
 5:
 6: repeat
 7:    Select $\phi_i \in \mathcal{A}$ that maximizes $\frac{B_i}{c_i}$
 8:    if $C + c_i \leq c_{max}$ then
 9:       $\mathcal{S} = \mathcal{S} \cup \phi_i$
10:       $C = C + c_i$
11:    end if
12:    $\mathcal{A} = \mathcal{A} \backslash \phi_{q,i}$
13: until $\mathcal{A} = \emptyset$
14:
15: Select a partition $\phi_t$ that maximizes $B_t$ over $S$
16: if $B(\mathcal{S}) \geq B_t$ then
17:    output $\mathcal{S}$
18: else
19:    output $\{\phi_t\}$
20: end if
```

Algorithm The greedy approximate algorithm, GREEDYSELECT is shown in Algorithm 5. The input to GREEDYSELECT is the bound on the allowable accesses denoted as c_{max} and the set of partitions ϕ. The solution to the selection problem is denoted as $\mathcal{S}$ and we refer to it as the *selection set*. We associate every partition ϕ_i with a cost c_i and a benefit B_i. Here, c_i is the cost of processing an unselected partition $\phi_j \notin \mathcal{S}$. For size-based selection c_i is the number of postings in the partition, and for equi-cost selection c_i is one. Its benefit, B_i, is the number of unprocessed postings in ϕ_j, i.e., $|\phi_j \setminus \cup_{s \in \mathcal{S}} s|$. $\mathcal{A}$ maintains the available partitions for selection and is updated after each iteration.

Each iteration of GREEDYSELECT consists of a **selection step** and an **update step**. In the selection step, the most promising partition based on the benefit-cost ratio $\frac{B_i}{c_i}$ is chosen from $\mathcal{A}$ (cf. line 7). We do not select a partition if it leads to exceeding the cost budget (cf. line 8). The update step updates $\mathcal{A}$ and the benefits of all the partitions in $\mathcal{A}$ which are future candidates for selection (cf. line 12).

Finally, to determine the solution set we compare the overall benefit of the selection set S, i.e. $B(\mathcal{S}) = |\bigcup_{\phi_j \in \mathcal{S}} \phi_j|$, with the partition with the best overall benefit (c.f. line 16-20).

4.4. Multi-Term Partition Selection

In the case of partition selection for single-term queries, every posting read from a partition qualifies as an answer, given that the time span of the partition overlaps with the query time-interval. Unlike this simpler setting, for multi-term queries there is an additional constraint imposed by the *conjunctive semantics* of query evaluation which requires that every result document also contain *all* the query keywords. Mimicking the conventional query processing (over standard posting lists), multi-term queries can be evaluated by intersecting partitions of individual query terms. Now, the partition selection aims to increase the coverage of postings that belong to this *intersection space* of partitions. Formally, both the selection variants, size-based and equi-cost, for a query Q where $m = |keywords(Q)|$, are defined as follows.

Definition 4.5 (Size-Based Selection for Multi-Term Queries)

$$\underset{\mathcal{S} \subseteq \phi}{\operatorname{argmax}} \left| \bigcap_{1 \le i \le m} \bigcup_{\phi_{i,j} \in \mathcal{S}} \phi_{i,j} \right| \quad \textit{s.t.}$$

$$\sum_{\phi_{i,j} \in \mathcal{S}} |\phi_{i,j}| \;\le\; \beta \cdot \left(\sum_{\phi_{i,j} \in A(Q)} |\phi_{i,j}| \right).$$

Definition 4.6 (Equi-Cost Selection for Multi-Term Queries)

$$\underset{\mathcal{S} \subseteq \phi}{\operatorname{argmax}} \left| \bigcap_{1 \le i \le m} \bigcup_{\phi_{i,j} \in \mathcal{S}} \phi_{i,j} \right| \quad \textit{s.t.}$$

$$|\mathcal{S}| \;\le\; \beta \cdot |A(Q)|.$$

Similar to the argumentation for single-term selection, the result size is the objective function. Observe that the constraints for multi-term selection are preserved from the previous problem formulations. Let us consider the objective function of both the multi-term selection problems. The intersection space, in the objective function above, is the intersection of the unions of the affected partitions. Using the distributive property of the set intersection operator we can also represent the above into unions of intersections of partitions $\phi_{i,j}$'s.

$$\bigcap_{1 \leq i \leq m} \bigcup_{\phi_{i,j} \in \mathcal{S}} \phi_{i,j} = \bigcup_{\phi_{i,j} \in \mathcal{S}} \bigcap_{1 \leq i \leq m} \phi_{i,j}$$

4.4.1. GreedySelect for Multi-term Selection

Let each of these resulting smaller intersections, consisting of one partition each from every term, be represented as a tuple **x**. These tuples **x** come from the Cartesian product among affected partitions for each term $a(q_i, Q)$, i.e., for a m-term query the Cartesian-product set $\mathcal{X} = a(q_1, Q) \times \ldots \times a(q_M, Q)$. We formally define **x**, an element of this Cartesian-product set $\mathcal{X}$, as :

$$\mathbf{x} = \{ (x_1, \ldots, x_m) \mid x_i \in a(q_i, Q)\}$$

Although this is a m-ary tuple, we treat this as a set whenever necessary. Having turned the objective function into a disjunctive formulation, analogous to the single-term setting, the problem formulation now intends to maximize the coverage of the results in the intersection space. We can now use GreedySelect over $\mathcal{X}$, where each element **x** is equivalent to a partition in single-term selection scenario.

The *benefit* of **x**, $B(x_i)$, is defined as the cardinality of the documents in the intersection of the partitions in **x** which are not in the selection set $\mathcal{S}$.

$$B_{\mathbf{x}} = |\ R(\ \mathbf{x} \setminus \mathcal{S}\ )\ |$$

In other words, the benefit or contribution of **x** represents the number of *new documents* which are present in *every* element partition of **x**. The *cost* definition of **x** depends on the sizes of the element partitions in **x**. As earlier, the cost of **x**, $c_{\mathbf{x}}$, in size-based selection is the sum of the sizes of the partitions in $\mathbf{x} \setminus \mathcal{S}$, i.e.,

$$c_{\mathbf{x}} = \sum_{\phi_{i,j} \in \mathbf{x} \setminus \mathcal{S}} |\phi_{i,j}|.$$

Equi-cost selection on the other hand defines the cost of **x** as the number of participating partitions not in $\mathcal{S}$.

$$c_{\mathbf{x}} = |\mathbf{x} \setminus \mathcal{S}|\,.$$

Algorithm 6: GreedySelect for multi-term partition selection

```
1:  input: $c_{max}$ , $\mathcal{X}$
2:  $\mathcal{S} = \emptyset$
3:  $\mathcal{A} = \mathcal{X}$
4:  $C = 0$
5:
6:  repeat
7:     Select $\mathbf{x} \in \mathcal{A}$ that maximizes $\frac{B_x}{c_x}$
8:     if $C + c_{\mathbf{x}} \leq c_{max}$ then
9:        $\mathcal{S} = \mathcal{S} \cup \{\phi_{i,j} | \phi_{i,j} \in \mathbf{x}\}$
10:       $C = C + c_{\mathbf{x}}$
11:    end if
12:    $\mathcal{A} = \mathcal{A} \backslash \mathcal{S}$
13:    Update $B_{\mathbf{x}'}$ and $c_{\mathbf{x}'}$ for $\mathbf{x}' \in \mathcal{A}$
14: until $\mathcal{A} = \emptyset$
15:
16: Select $\mathbf{y} \in \mathcal{X}$ that maximizes $R(\{\mathbf{y}\})$
17: if $|R(\mathcal{S})| \geq R(\{\mathbf{y}\})$ then
18:    output $\mathcal{S}$
19: else
20:    output $\{\phi_t | \phi_t \in \mathbf{y}\}$
21: end if
```

The modified inputs to GreedySelect is the set $\mathcal{X}$, with benefit $B_{\mathbf{x}}$ and cost $c_{\mathbf{x}}$ for each of its elements **x**. GreedySelect now proceeds conventionally by greedily choosing the **x** with the best benefit by cost ratio $\frac{B_x}{c_x}$. Observe that the choices of elements from $\mathcal{X}$ are not independent. A pair of tuples can share the same partition. Thus, selection of a tuple **x** might result in reducing the cost (which is not the case in single-term selection) of others which have at least one of the constituent partitions common with **x**. Hence in the update step apart from updating the benefit of **x**, we also update its cost $c_{\mathbf{x}}$. Owing to such a cost dependence among the tuples, the approximation guarantee of GreedySelect does not apply to multi-term selection.

Exploiting temporal overlap among partitions For a time-travel query with m terms with p affected partitions per term, the input size is exponential in the number of terms, i.e., $|\mathcal{X}| = p^m$. In case of queries with large m or p, computations in GreedySelect can become prohibitive. This is because of the update steps in each iteration where the

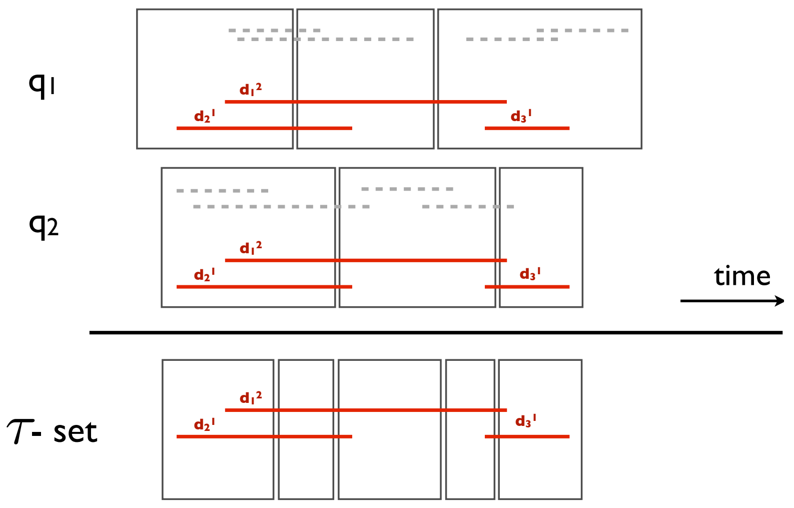


Figure 4.2.: τ for the affected partitions time-travel query "q_1 q_2"

benefits and costs of the remaining partitions $\mathcal{A}$ are recomputed.

To alleviate this, we operate on a constrained set, $\tau \subseteq \mathcal{X}$, which has a cardinality linear in the number of participating partitions as opposed to high number of combinations in $\mathcal{X}$. This constrained set is obtained by defining a τ-join operation over the term-partition sets ϕ_i's. Each element $\mathbf{t}$ of the resulting tuple $\mathbf{c} \in \mathcal{X}$ has the property that there is a non-zero time-overlap between all of the constituent partitions.

$$\mathbf{c} = \{(c_1, \ldots, c_m) \mid \forall c_i \in a(q_i, Q), c_j \in a(q_j, Q), \;\; span(c_i) \sqcap span(c_j)\}$$

For example, in Figure 4.2, the queries q_1 and q_2 have 3 partitions each. The $|\mathcal{X}| = 9$ and the resulting τ has a cardinality 5 after the τ-join operation.

For a time-travel query with m terms and p partitions per term, the number of elements in τ is linear in the number of affected partitions, i.e., $p \leq |\tau| \leq m.p$. The partitions in τ exploits the temporal overlap across partitions of different query terms and has a reduced input size as compared to $\mathcal{X}$. The temporal partitioning induces a temporal clustering of the postings. Hence, the results contained in the intersection of partitions in $\mathcal{X}$ are already captured in the $\tau \subset \mathcal{X}$. We further show for equi-cost based partition selection, GreedySelect chooses elements only from τ. In other words, GreedySelect over the Cartesian set is equivalent to GreedySelect over τ.

Theorem 4.2 GreedySelect *for equi-cost based selection on $\mathcal{X}$ always chooses elements which belong to τ.*

We prove this theorem by contradiction, by first choosing an element from $\mathcal{X} \backslash \tau$ and showing that we can replace this element with a better candidate from τ. For the formal proof, we introduce the notion of *selected-partition space*. Let the selection set $\mathcal{S}$ be the set of already selected partitions and thus the result set $R(\mathcal{S}, Q)$ denote the actual set of result documents covered by the partitions in $\mathcal{S}$. A time interval $[t_b, t_e]$ is said to be *selected* if there is a partition from each term q_i in $\mathcal{S}$ which covers it, i.e.,

$$\exists \phi_{i,j} \in a_i, \mathtt{begin}(\phi_{i,j}) < t_b \ \wedge \ t_e < \mathtt{end}(\phi_{i,j}).$$

In other words, an unselected-space refers to a range where all of the partitions have been selected at the current state of the algorithm.

We first prove that for any $\mathbf{x} \in \mathcal{X}$ there exists a $\mathbf{t} \in \tau$ which has at least the same result size.

Lemma 4.3 *For any $\mathbf{x} \in \mathcal{X}$ there exists a $\mathbf{t} \in \tau$ such that the following holds*

$$\left| \bigcap_{\phi_x \in \mathbf{x}} \phi_x \right| \leq \left| \bigcap_{\phi_t \in \mathbf{t}} \phi_t \right|.$$

Proof: *Let us consider the candidate $\mathbf{x} \in \mathcal{X} \setminus \tau$. Let* $\textsc{MaxBP} = \underset{\phi \in \mathbf{x}}{\mathtt{argmax}}\, \mathtt{begin}(\phi)$ *denote the partition in $\mathbf{x}$ that has the maximum begin-time. Also consider a $\mathbf{t} \in \tau$ such that*

$$\forall \phi_{i,t} \in \mathbf{t}, \mathtt{begin}(\phi_{i,t}) \leq \mathtt{begin}(\textsc{MaxBP}) < \mathtt{end}(\phi_{i,t}).$$

Since $\mathbf{x} \in \mathcal{X} \backslash \tau$ there exists a $\phi_{i,x} \in \mathbf{x}$ such that $\mathtt{span}(\phi_{i,x}) \not\sqsubset \mathtt{span}(\textsc{MaxBP})$. Consider a version d_i^k which belongs to the result set $R(\mathbf{x}, Q)$ (assuming $R(\mathbf{x}, Q) \neq \emptyset$). This means that the following holds

$$\mathtt{begin}(d_i^k) < \mathtt{end}(\phi_{i,x}) \ \wedge \ \mathtt{end}(d_i^k) > \mathtt{begin}(\textsc{MaxBP}).$$

We further note that $(\phi_{i,x} \ \cap \ \textsc{MaxBP}) \ \subseteq \ (\phi_{i,t} \ \cap \ \textsc{MaxBP})$ since for all versions d_i^k in $\phi_{i,x} \ \cap \ \textsc{MaxBP}$,

$$\mathtt{valid}(d_i^k) \sqsubset \mathtt{span}(\phi_{i,t}).$$

We can thus replace $\phi_{i,x}$ with $\phi_{i,t}$ for a larger result set. We can carry the same replacement for all terms $q_i \in \mathtt{keywords}(Q)$ where $\mathtt{span}(\phi_{i,x}) \not\sqsubset \textsc{MaxBP}$ for a better overall result set $\left|\cap_{\phi \in \mathbf{t}} \phi\right|$. This proves our claim.

□

We now proceed with the proof of Theorem 4.2.

Proof: *We prove this by induction on the number of iterations i of the* GreedySelect *algorithm.*

$i = 1$: *For the first iteration $\mathcal{S} = \emptyset$. We argue that, given that there is enough budget for selection the candidate selected is always from τ. We prove this by contradiction. We assume* *that the candidate* $\mathbf{x} \in \mathcal{X} \backslash \tau$ *has the best* $\frac{B_x}{c_x}$. *According to Lemma 4.3 there exists a* $\mathbf{t} \in \tau$ *with* $B_t \geq \mathbf{x}$. $\mathcal{S} = \emptyset$ *means that the costs are the same for all candidates* $c_t = c_x$, *hence,* $\frac{B_x}{c_x} \geq \frac{B_t}{c_t}$.

$i \to i+1$: *Choosing from* τ *for the first* i *iterations induces multiple selected regions in the intersection space. Because of the nature of the* τ*-join certain time intervals are completely covered.*

Now choosing a candidate $\mathbf{x}' \in \mathcal{X} \setminus \tau$ *could have a cost (where* $0 \leq c_{x'} \leq m$*) depending on the number of constituent partitions already in the selection set. To prove that the choice of the candidate is still made from* τ *we argue as in the proof of Lemma 4.3. Assume that there is a better candidate* $\mathbf{x} \in \mathcal{X} \setminus \tau$ *(best benefit/cost ratio), and a non-zero cost* $c_{\mathbf{x}}$. *We can always replace the partitions* $x_i \in \mathbf{x} \setminus \mathcal{S}$ *by another partition of the same term in the following ways:*

Case 1 – $x_i \notin \mathcal{S}\ \ \forall x_i \in \mathbf{x}$ **:** *In the case of* $\mathbf{x}$ *having no partitions from the selection set* $\mathcal{S}$, *i.e.,*

$$\forall x_i \in \mathbf{x}\ :\ x_i \notin \mathcal{S}$$

we use Lemma 4.3 to choose a better candidate from τ *since there are only non-selected regions from where a choice can be made.*

Since the selected regions provide no benefit we operate only within unselected regions. We denote the minimum time boundary in the region as left region boundary and the maximum time boundary as the right region boundary. For cases 2 and 3, we consider candidates $\mathbf{x}$ *with non-zero benefit, and non-zero cost less than* m, *i.e.,*

$$\exists x_i,\ \ x_i \in \mathbf{x} \cap \mathcal{S}.$$

Case 2 – *Suppose that the partition* $x'_i \in \mathbf{x}' \mid x'_i \in \mathcal{S}$, *only belong to the right region boundary. We can always choose a replacement partition* r_j *for* $x'_j \in \mathbf{x}' \mid x'_j \notin \mathcal{S}$, *where* r_j *and* x'_j *belong to the same term, such that the new replacement candidate* $\mathbf{r} \in \tau$ *has a better benefit than* $\mathbf{x}'$. *More specifically, the replacement candidate* $\mathbf{r} \in \tau$ *has the following selected and unselected partitions*

- **selected partitions:** *Selected partitions* x'_i *such that* $x'_i \in \mathbf{x}' \mid x'_i \in \mathcal{S}$.
- **unselected partitions:** *Unselected replacement partitions* r_j *which contain the minimum begin time of the selected partitions,* $t_{mbt} = \min\{b_{x'_i} | x'_i \in \mathbf{x}'\ \wedge\ x'_i \in \mathcal{S}\}$ *, i.e.,*

$$r_j \mid b_{r_j} \leq t_{mbt} < e_{r_j}$$

The replacement candidate $\mathbf{r}$ *has the same cost as its counterpart* $\mathbf{x}'$, $\mathrm{cost}(\mathbf{r}) = \mathrm{cost}(\mathbf{x}')$, *and a benefit-cost ratio* $\frac{B_r}{c_r} \geq \frac{B_{x'}}{c_{x'}}$. *Since such a replaced candidate belongs to* τ, *this is contrary to our assumption and our claim holds.*

Case 3 – *Similar to Case 2, if* $\mathbf{x}'$ *has partitions belonging to the left boundary of the region, we can replace the unselected partitions of each term by a replacement partition which contains/overlaps with the* maximum end time *among the partitions which belong to the selection set in* x, *i.e.,* $t_{met} = \max\{e_{x'_i} \mid x'_i \in \mathbf{x}'\ \wedge\ x'_i \in \mathcal{S}\}$. *The new replacement candidate* $\mathbf{r}$ *belongs to* τ *and has a better or equal benefit than* $\mathbf{x}$ *contrary to our assumption.* $\square$

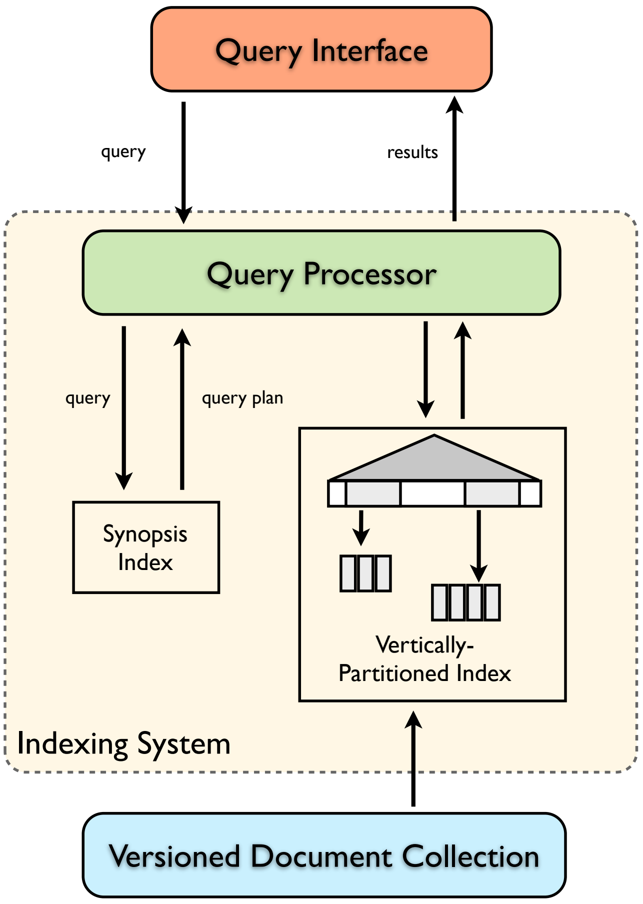


Figure 4.3.: System architecture

4.5. System Architecture

Figure 4.3 shows a hihg-level overview of the indexing system where our selection methods can be employed. The indexing system consists of *vertically-partitioned index* and the *synopsis index*.

- **Vertically-partitioned index :** The entire document collection is indexed employing the partitioning schemes described in [BBNW07].
- **Synopsis Index :** During index building we materialize a KMV synopsis for every partition into a *synopsis index*. When a query is issued, synopses (i.e., multiple synopsis) corresponding to the affected partitions are retrieved. In the partition selection algorithm that follows, GREEDYSELECT for single and multi-terms, cardinality estimates are determined for benefit values B_i and B_x.

The query interface can be a user of a system which generates time-travel queries. During processing queries, query optimization is performed on the synopsis index by invoking the selection algorithms and a plan is generated. Based on the query plan, accesses are scheduled on the vertically-partitioned index and the results are reported. Because of the anytime nature of the selection algorithm, the user can terminate the search, when satisfied, giving her the maximum recall computed thus far.

4.6. Practical Issues

While the previous two sections presented the theoretical underpinnings for the partition-selection problem, in this section, we discuss a few issues relevant to their implementation that we faced in practice and present our solutions.

4.6.1. Dealing with Partition and Query Boundary Alignment

In our descriptions of the algorithms, we assumed that if a partition overlaps with the query time-interval, then its contribution to the final answer set is from *all the postings* in the partition. In other words, we ignored the fact that even within a partition, possibly a large number of postings may not satisfy the temporal predicate if the temporal boundaries of the partition are not completely contained within the range specified by the temporal predicate. Note that this affects the estimates of the *benefit* values of the partitions in the boundaries of the query time – thus the benefits of at most 2 partitions per term are in error.

This error can be significantly improved if we adjust the value of benefit of a partition to account for incomplete overlap along the time axis. A straightforward approach for this, which we employ in our implementation, is to *scale* the benefits by the fraction of temporal overlap between the query and the partition. In practice, we observed that this simple scaling (which is similar to making an uniformity assumption during cardinality estimates) works very well.

4.6.2. I/O Budget Underflow

Another issue that comes up when we are using only *estimates* of benefit provided by partition(s) towards the final answer set is that during partition selection, we may encounter a situation where *none of the partitions* show any non-zero benefit, although in reality they may contain some results. When faced with such a situation, the partition selection algorithms described in Sections 4.4 and 4.3 simply terminate – even if the specified I/O budget allows for more partitions to be read.

To avoid this undesirable behaviour, the partition-selection algorithm can be modified to ignore the estimates of benefits when *all* the unselected partitions have zero estimated benefits. At this stage, partitions are selected in *decreasing* order of their size as long as the I/O budget is not violated.

4.7. Experimental Evaluation

4.7.1. Evaluation Framework

In this section, we present and discuss the results of a detailed experimental evaluation of our algorithms in terms of their effectiveness in achieving high relative recall with a specified budget of index accesses.

4.7.2. Setup

All our algorithms, including the underlying time-travel inverted index framework, were implemented using Java 1.6. All experiments were conducted on Dell PowerEdge M610 servers with 2 Intel Xeon E5530 CPUs, 48 GB of main memory, a large iSCSI-attached disk array, and Debian GNU/Linux (SMP Kernel 2.6.29.3.1) as operating system. Experiments were conducted using the Java Hotspot 64-Bit Server VM (build 11.2-b01).

4.7.3. Datasets Used

For our experiments we used three different datasets, all derived from real-world data sources.

WIKI The *English Wikipedia revision history* [WIK13], whose uncompressed raw data amounts to 0.7 TBytes, contains the full editing history of the English Wikipedia from January 2001 to December 2005. We indexed all versions of encyclopedia articles excluding versions that were marked as the result of a minor edit (e.g., the correction of spelling errors etc.). This yielded a total of 1,517,524 documents with 15,079,829 versions having a mean (μ) of 9.94 versions per document at standard deviation (σ) of 46.08.

UKGOV This is a subset of the European Archive [EA13], containing weekly crawls of eleven governmental websites from the U.K. We filtered out documents not belonging to MIME-types `text/plain` and `text/html` to obtain a dataset that totals 0.4 TBytes. This dataset includes 685,678 documents with 17,297,548 versions ($\mu = 25.23$ and $\sigma = 28.38$).

NYT The *New York Times Annotated corpus* [NYT13] comprises more than 1.8 million articles from the New York Times published between 1987 and 2007. Every article has an associated time-stamp which was taken as the begin time for that article. The end time for each article was chosen to be 90 days after the begin time, giving every document a validity time of 90 days. This is done to reflect the real world setting where the news articles are publicly available only for a limited period from their publication.

Note that each of these datasets represents a realistic class of time-varying text collection typically used in temporal text analytics. Specifically, WIKI corresponds to an explicitly version controlled text collection, UKGOV is an archive of the evolving Web, and NYT is an instance of archive of continually generated newspaper content. For the ease of experimentation, we rounded the time-stamps of versions to the nearest day for all datasets.

4.7.4. Query Workload

We compiled three dataset-specific query workloads by extracting frequent queries from the AOL query logs, which were temporarily made available during 2006. For the WIKI dataset we extracted 300 most frequent queries which had a result click on the domain `en.wikipedia.org` and similarly for NYT and UKGOV we compiled 300 queries which had a result hit on `nytimes.com` and 50 queries which had result hit on `.gov.uk` domains (cf. Appendix). Using these keyword queries, we generated a time-travel query workload with 3 instances each for the following 2 different temporal predicate granularities: 30 days and 1 year.

4.7.5. Index Management

Index	**UKGOV**	**NYT**	**WIKI**
Fixed-7	11GB	13GB	13GB
Synopsis Index - 5% sample	146MB	134MB	146MB
Synopsis Index - 10% sample	291MB	258MB	290MB
Fixed-30	4.4GB	3.5GB	6.3GB
Synopsis Index - 5% sample	61MB	39MB	75MB
Synopsis Index - 10% sample	122MB	74MB	149MB

Table 4.2.: Synopsis index

Since our selection techniques operate on temporally-partitioned posting lists, we chose the following partitioning schemes :

- **Fixed-time partitioning** A simple partitioning scheme in which a partition boundary is placed after a fixed time window. We present results for two time window sizes: (i) 1 week (referred to as **Fixed-7** partitioning), and (ii) 1 month (referred to as **Fixed-30** partitioning). Unless otherwise mentioned, all the results presented in this chapter are from **Fixed-7** partitioning.

- **Vertical Partitioning** These are index structures built using partitioning strategies discussed in [BBNW07]). More specifically, we build index structures using the *space-bound approach* with the parameters $\kappa = 1.5, 3.0$ as two representatives of lower and higher degree of partitioning. These are represented as **VERT-1.5** and **VERT-3.0** respectively.

Each of the above time-travel inverted indexes is stored on disk using flat files containing both the lexicon as well as posting lists. At run time, the lexicon is read completely into memory, and for a given query the appropriate partition is retrieved from the index flat file on disk. These posting lists are stored using variable-byte compression.

Synopsis structures The estimates from the KMV synopses [BHR+07] that we chose to implement are naturally dependent on their size in relation to the raw data size. We experimented with two sizes of synopses: 5% and 10% of the partition size (with minimum size set to 100). Unless otherwise mentioned, we report results for 10% size of the KMV synopsis. A synopsis index was generated during index construction time and stored as flat files on disk. Instead of storing the list of hashed double values of the KMV synopsis, the corresponding document identifiers (32-bit integers) were stored for better compression (Table 4.2). The document identifiers were translated to their respective doubles during query time for the necessary KMV intersection estimation. An additional entry in the lexicon was stored the offset in the synopsis index file corresponding to the synopsis for each partition.

Finally, we employed a practically infeasible **oracle** for partition selection, which computes the *accurate values* of set operations (intersection and union) between partitions. Oracle computes these values by simply evaluating the query completely, without any partition selection, and then uses them in partition selection to overcome the errors due to estimates from the KMV synopses. In our experiments, we consider the oracle as a competitor, where exact cardinalities are known, to compare against our synopsis-based-selection approaches.

4.7.6. Evaluation Methodology

We evaluate the impact of partition selection by measuring the recall obtained at different values of the parameter β. In each experiment we measure the average recall value obtained per query, for a certain selection method, over increasing values of β from 0 to 1 with a step size of 0.1. During averaging we exclude the measurements for two types of queries. First, we exclude queries with term/s not in the lexicon as they contribute to false-positives for partition selection. Since we employ conjunctive query semantics we need to select at least one partition per-term. Thus, secondly we also ignore queries which result in exactly one affected partition for each term.

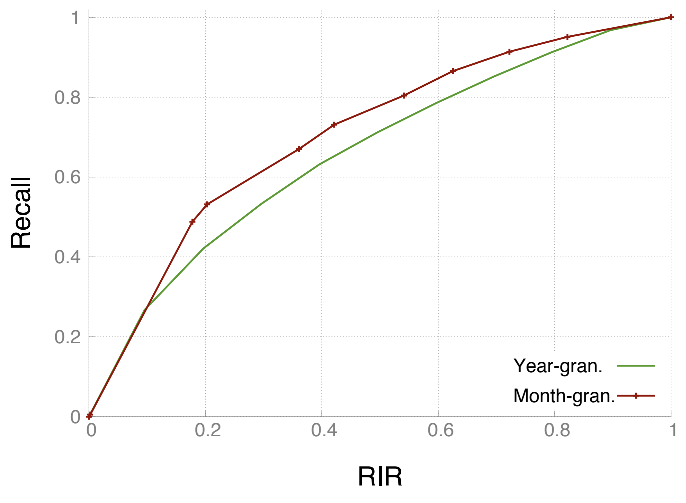
To compare the I/O performance of different techniques, we measure the number of postings read after applying partition selection – denoted as **RWS**, and the number of postings read without applying partition selection – denoted as **RWOS**. The ratio $\frac{RWS}{RWOS}$, called *Ratio-of-index read*, is denoted as **RIR**. The ratio-of-index read captures the amount of index accessed relative to the overall index-access cost for processing the entire query. We also measure the wall-clock times during query processing, reported in milliseconds.

4.7.7. Performance of Partition Selection

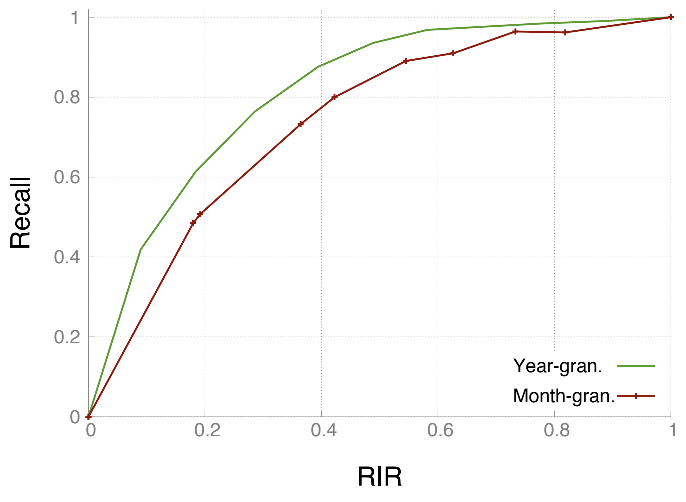
In the first set of experiments, we examine the impact of both the selection methods described in this chapter - size-based selection and equi-cost selection. To this end, we execute the different query granularities on the **Fixed-7** index for the given datasets, and measure the recall values at various stages of query execution. The cost incurred at a given stage of query execution is measured, as introduced above, by RIR. The results presenting the recall levels achieved at different RIR values are shown in Figure 4.4 (size-based selection) and Figure 4.5 (equi-cost partition selection).

We observe that both selection algorithms achieve perfect recall already when accessing about 50% of the index. Since both the algorithms are incremental in nature, recall always increases with an increase in allowable I/O budget β. Both the selection methods are able to achieve a recall of 80% by accessing less than 30% of the affected postings for NYT. In UKGOV, a 80% recall is achieved by accessing 40% of the index. In WIKI, the results are not as temporally clustered as in NYT and UKGOV but we still reach 80% of recall by accessing around 60% of the affected postings.

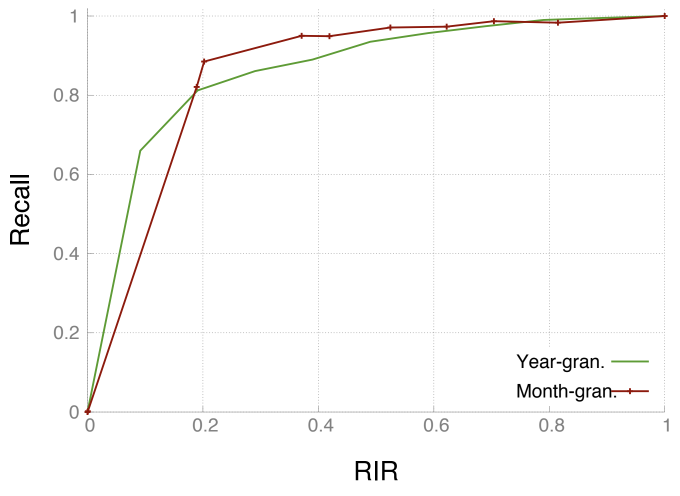
Next, we observe that, with the exception of UKGOV, the partition-selection methods respond better to month-granularity queries than the year-granularity queries. Month-granularity queries have fewer affected partitions than year-granularity queries. This suggests that there is (i) a high degree of temporal clustering of results and (ii) high replication of postings, which the selection methods exploit. Selecting the partitions with a high concentration of results gives the observed boost to the recall levels. The



(a) Wikipedia

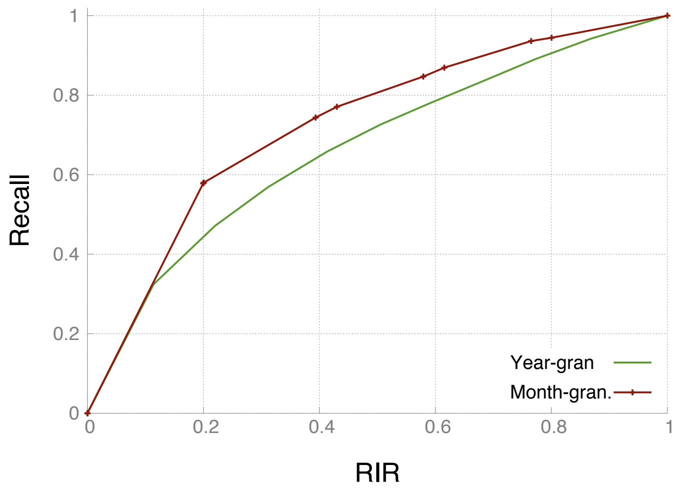


(b) UKGOV

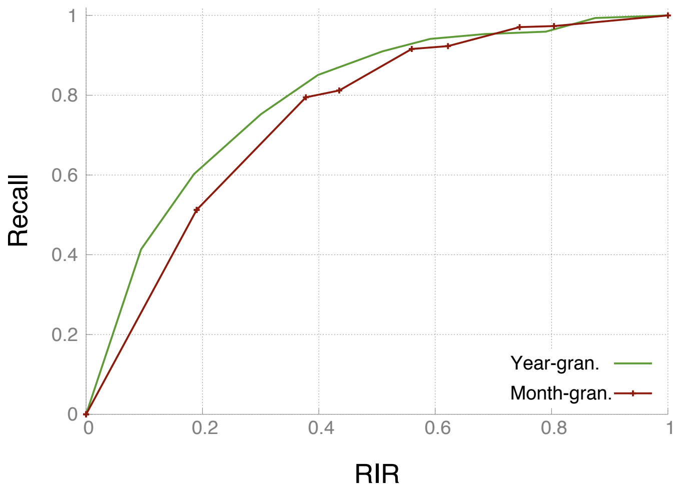


(c) New York Times

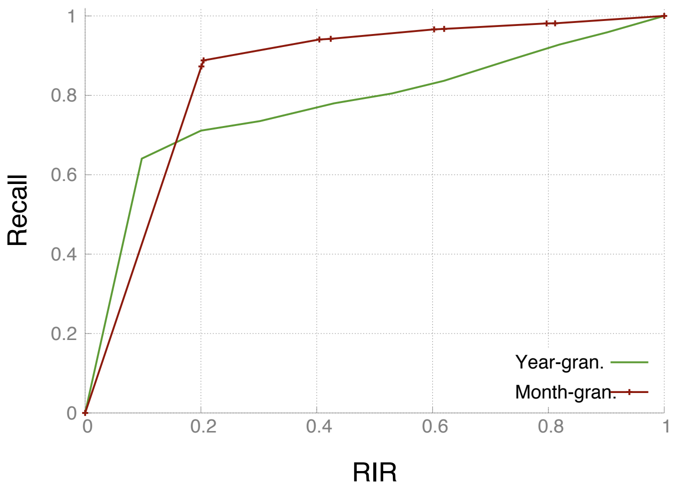
Figure 4.4.: Performance of size-based partition selection on Fixed-7 index



(a) Wikipedia

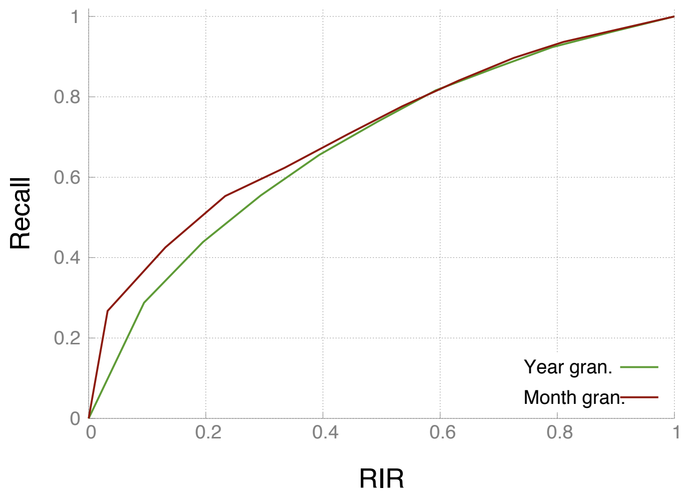


(b) UKGOV

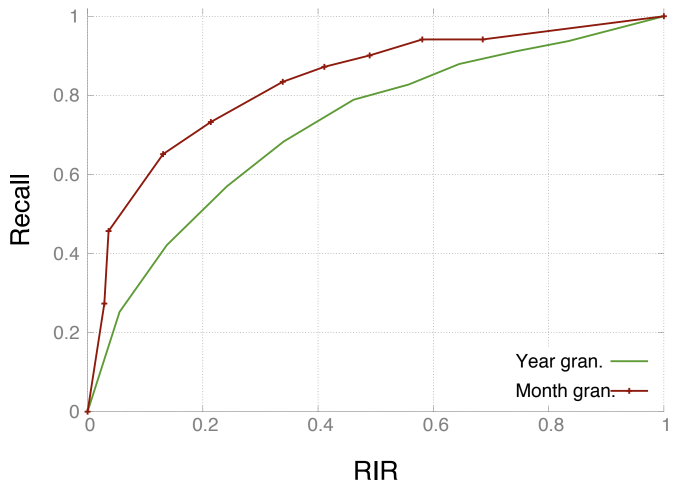


(c) New York Times

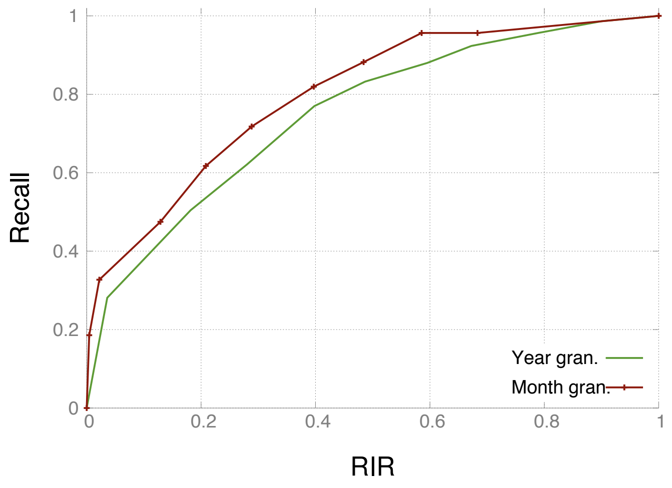
Figure 4.5.: Performance of equi-cost partition selection on Fixed-7 index



(a) Wikipedia



(b) UKGOV



(c) New York Times

Figure 4.6.: Performance of size-based partition selection on VERT-3.0 index

subsequent choice of partitions understandably adds lower improvement than the initial choices exhibiting the property of diminishing returns. When we conducted the same experiment on the vertically-partitioned index VERT-3.0, we did not observe gains as significant as in **Fixed-7**. This is because the replication of the postings is controlled and bounded.

The final observation which we make is that both the selection methods perform almost at par. There is no considerable difference in performance and the choices of partitions selected, in our experiments, is almost the same. This prompts the use equi-cost partition selection which is easier to implement and has a proven approximation guarantee. Thus from now on, the charts and tables contain results which employed equi-cost partition selection.

4.7.8. Query-Processing Performance

In the next set of experiments,s we examine the performance of query processing, guided by partitioning selection, by measuring wall-clock times during query execution. Since partition selection is typically useful when there are no caching effects, the focus was on measuring run-times in a cold-cache setup. We start with cold caches and flush them after each query execution step. Each time-travel query from the workload was evaluated for different β values (0.1 through to 1.0) and the average time taken (in milliseconds) for each of these bounds are presented in Tables 4.3 and 4.4. The first column represents the tunable parameter β, followed by the recall attained, and finally the average wall-clock per query. We compare the results of selection based retrieval by introducing two competitors :

- the standard unpartitioned posting list, NOPARTITION, and
- partitioned lists not supporting partition selection, NOSELECTION.

Notice that the wall-clock times reported for $\beta = 1.0$ are the times taken for NOSELECTION. The reported wall-clock times include the time taken by the synopsis-based partition selection along with the time taken for the actual query processing. The time taken for partition selection, however, is negligible and the major fraction of the overall reported time is spent on query processing. The wall-clock times further corroborate the observations presented before. We observe that in case of executing time-travel queries, NOPARTITION takes almost 3 secs for WIKI, 1 sec for NYT and as long as 12 secs for UKGOV (see Table 4.3) irrespective of the query-time granularity. Firstly, employing a partitioned index results in superior performance as indicated by the NOSELECTION values in the tables. Secondly, a partitioned index allows for partition selection further reducing wall-clock times to give recall values of almost 0.8 in only 50%-60% of

		Year Granularity		Month Granularity	
	BOUND	Recall	Wall-clock times (ms)	Recall	Wall-clock times (ms)
WIKI					
	0.1	0.27	207.6	0.01	5.7
	0.2	0.42	367.7	0.49	88.5
	0.3	0.53	436.8	0.53	94.3
	0.4	0.63	521.0	0.67	134.3
	0.5	0.71	594.8	0.73	135.9
	0.6	0.78	646.7	0.80	165.2
	0.7	0.85	736.3	0.87	165.9
	0.8	0.91	798.5	0.91	186.4
	0.9	0.97	877.4	0.95	192.0
NOSELECTION	1	1.00	1,020.8	1.00	212.0
NOPARTITION	1	1.00	3,217.0	1.00	3,217.0
UKGOV					
	0.1	0.42	1,615.9	0.00	0.0
	0.2	0.61	2,788.8	0.48	352.6
	0.3	0.76	3,705.2	0.51	370.4
	0.4	0.88	4,592.1	0.73	590.3
	0.5	0.94	5,183.2	0.80	644.3
	0.6	0.97	5,772.6	0.89	751.2
	0.7	0.98	6,427.9	0.91	852.7
	0.8	0.98	7,025.2	0.96	927.4
	0.9	0.99	7,635.8	0.96	1,026.6
NOSELECTION	1	1.00	8,490.1	1.00	1,225.9
NOPARTITION	1	1.00	12,598.0	1.00	12,598.0
NYT					
	0.1	0.66	200.9	0.00	0.0
	0.2	0.81	254.5	0.82	103.5
	0.3	0.86	301.4	0.89	106.0
	0.4	0.89	308.2	0.95	117.5
	0.5	0.94	328.1	0.95	122.0
	0.6	0.96	358.7	0.97	125.0
	0.7	0.97	384.8	0.97	126.0
	0.8	0.99	428.6	0.99	128.5
	0.9	0.99	468.2	0.98	141.5
NOSELECTION	1	1.00	525.7	1.00	146.0
NOPARTITION	1	1.00	1,014.0	1.00	1,014.0

Table 4.3.: Wall-clock times for selection over Fixed-30

		Year Granularity		Month Granularity	
	Bound	Recall	Wall-clock times (ms)	Recall	Wall-clock times (ms)
WIKI					
	0.1	0.28	105.8	0.26	12.8
	0.2	0.43	154.4	0.42	33.4
	0.3	0.55	212.2	0.55	44.0
	0.4	0.65	268.1	0.62	55.5
	0.5	0.73	320.5	0.70	64.6
	0.6	0.81	375.6	0.77	75.6
	0.7	0.87	429.4	0.84	87.4
	0.8	0.92	477.6	0.89	97.2
	0.9	0.96	539.0	0.93	104.3
NoSelection	1.0	1.00	596.4	1.00	129.0
NoPartition	1.0	1.00	3,217	1.00	3,217.0
UKGOV					
	0.1	0.25	295.1	0.27	2.3
	0.2	0.42	601.6	0.45	41.8
	0.3	0.56	915.6	0.65	124.9
	0.4	0.68	1260.7	0.73	172.3
	0.5	0.78	1544.4	0.83	239.7
	0.6	0.82	1803.1	0.87	271.1
	0.7	0.87	2098.3	0.90	325.0
	0.8	0.91	2265.2	0.94	358.7
	0.9	0.94	2436.2	0.95	395.7
NoSelection	1	1.00	2720.2	1.00	572.7
NoPartition	1	1.00	12,598	1.00	12,598.0
NYT					
	0.1	0.28	26.7	0.00	0.0
	0.2	0.50	53.0	0.18	1.0
	0.3	0.62	50.87	0.32	3.0
	0.4	0.76	54.9	0.47	11.8
	0.5	0.83	60.6	0.61	21.4
	0.6	0.87	65.3	0.71	19.9
	0.7	0.92	72.4	0.81	24.1
	0.8	0.95	81.9	0.88	26.1
	0.9	0.98	83.6	0.95	27.2
NoSelection	1.0	1.00	90.1	1.00	44.7
NoPartition	1.0	1.00	1,014	1.00	1,014.0

Table 4.4.: Wall-clock times for selection over VERT-($\kappa = 3.0$)

time taken by NoSelection. Although, in smaller collections, like NYT, the absolute improvements in wall-clock times might not be large but in larger corpora, which is typically what web archives represent, the gains are substantial.

Additionally, the *anytime* nature of the selection algorithm also means that the user can terminate the query processing at any instant she wishes and can still get the maximum recall at that stage of the computation. A quick preview at the results after 3/4 of a second can prove beneficial with almost 90% recall (UKGOV monthly-granularity queries) or 85% recall (WIKI year-granularity queries). The results for space-bound indexing follow a similar trend (as reported in Table 4.4).

4.7.9. Impact of using Synopses

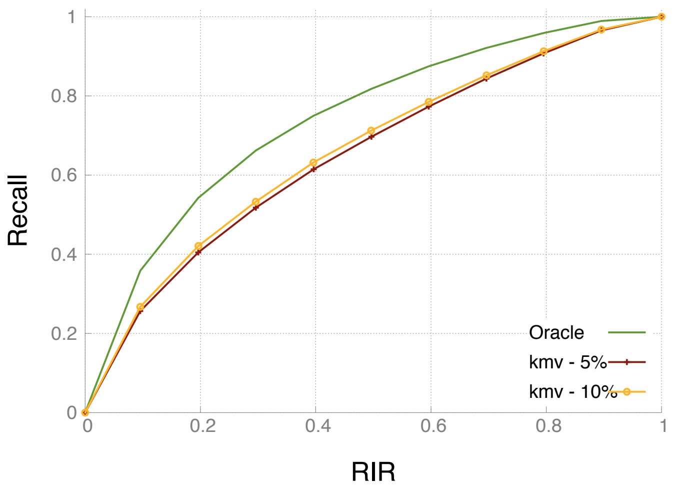
The next set of experiments is aimed at quantifying the impact of using KMV synopses for the estimation of benefits and the effect of different synopses size. For each dataset, we measure the average recall obtained for each granularity of time-travel queries, using 5% and 10% synopses, and compare them with those of *oracle* outlined earlier. The results of this experiments over indexes with **Fixed-7** partitioning, are shown in Figure 4.7 for query-granularity of one year.

We can make the following observations from these plots: (i) The gap between a 5% KMV synopsis and 10% synopsis is negligible, prompting our choice of using 5% KMV synopsis. (ii) Although oracle-based estimates are, as expected, better overall, improvements over using KMV synopsis estimates are not significantly large.

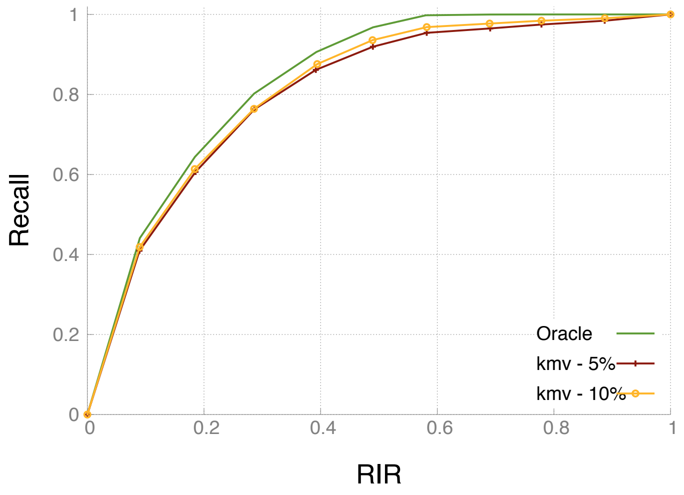
KMV synopsis are stored as arrays of doubles and much smaller than individual postings and can also be compressed and kept in memory. We also see from our experiments that query optimization using partition selection is a small fraction of the overall query-processing time. Thus, with a small memory footprint, and a quick estimation capability, we perceive that the use of KMV synopsis is a reasonable choice for partition selection. We used KMV-5% in all our experiments unless explicitly mentioned.

4.7.10. Impact of Partition Granularity

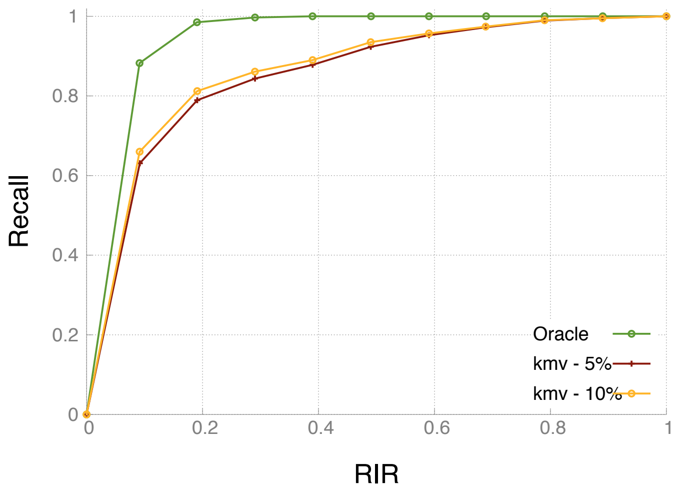
In our final experiment, we wanted to examine the effect of partition selection over varying partitioning granularities. We experimented with two different granularities of fixed partitioning – 7-day and 30-day time-intervals, resulting in **Fixed-7** and **Fixed-30** index configurations. **Fixed-7** has a higher number of partitions, thus can be seen as having smaller partition sizes in comparison to **Fixed-30**. Clearly, this allows efficient processing of time-point or short duration queries. However, it deteriorates for larger time-interval queries if partition selection is not employed. On the other hand, performance with partition selection shown in Figure 4.8 for year-granularity queries, shows



(a) Wikipedia

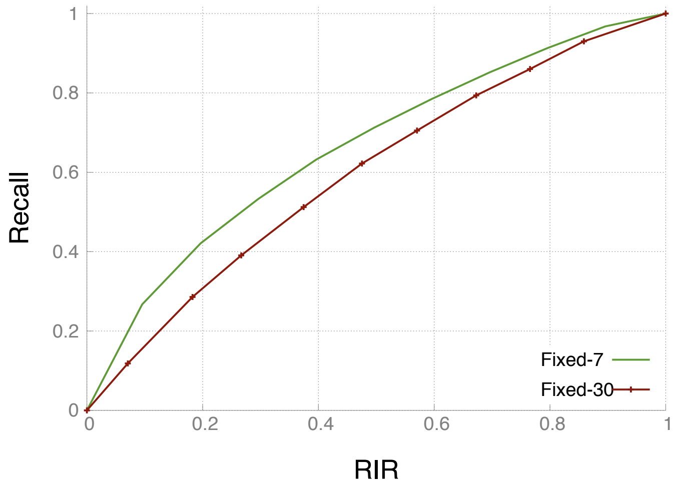


(b) UKGOV

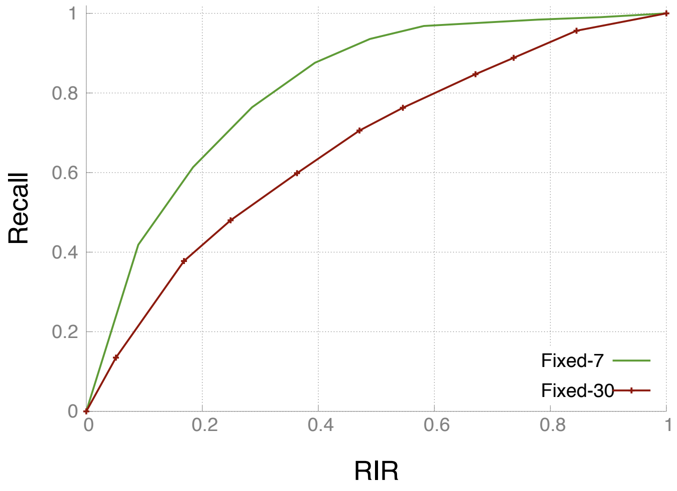


(c) New York Times

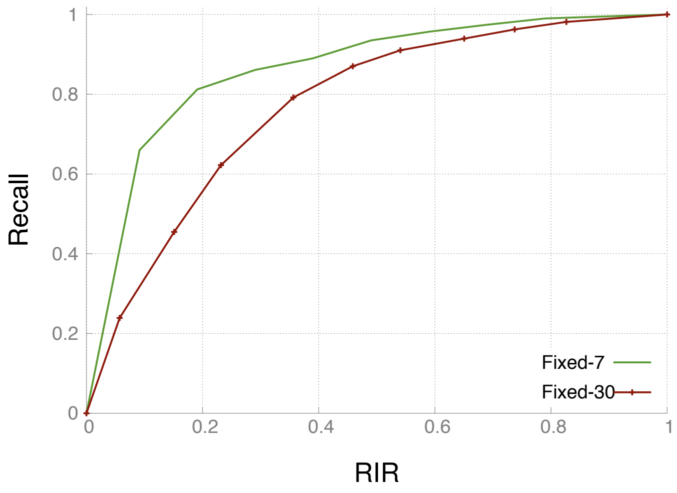
Figure 4.7.: Impact of using synopses



(a) Wikipedia

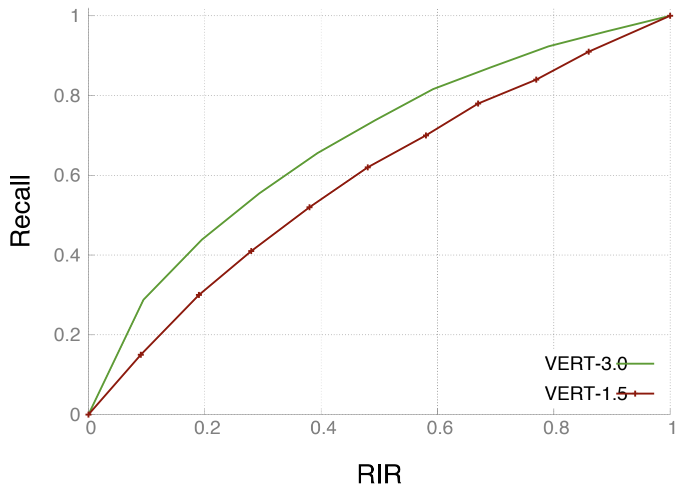


(b) UKGOV

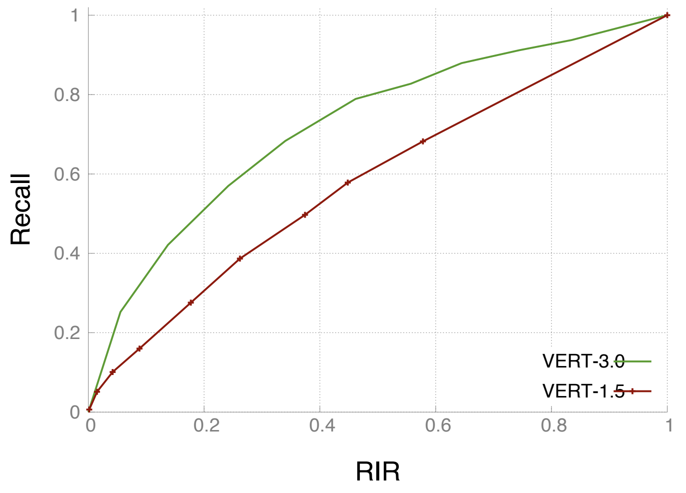


(c) New York Times

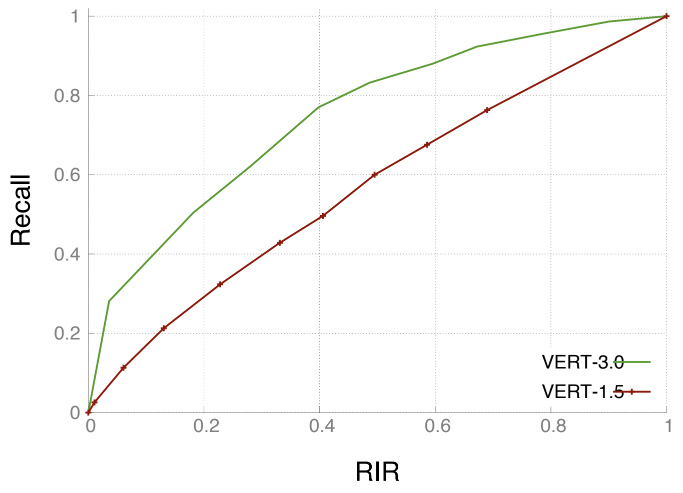
Figure 4.8.: Effect of varying partition granularity on FIXED indexes



(a) Wikipedia



(b) UKGOV



(c) New York Times

Figure 4.9.: Effect of varying partitioning granularities on VERT indexes

that even for smaller partitions sizes this issue can be effectively alleviated.

We also considered the partitions created by vertical partitioning of granularities — VERT-1.5 and VERT-3.0. We see a similar effect as in the case of fixed partitioning, i.e., selection is much more effective for partitioning schemes with high replication of postings. In this case, selection over VERT-3.0 provides higher benefit consistently over VERT-1.5 (see Figure 4.9).

4.8. Related Work

Closest to the ideas presented here is the work on time-travel text search [BBNW07] that allows users to search only the part of a document collection that existed at a given time point. To support this functionality efficiently, posting lists from an inverted index are temporally partitioned either according to a given space bound or required performance guarantee. Postings whose valid-time interval overlaps with multiple of the determined temporal partitions are judiciously replicated and put into multiple posting lists, thus increasing the overall size of the index.

Join-processing techniques for temporal databases [GJSS05] are a second class of related work whose focus, to the best of our knowledge, has been on producing accurate query results opposed to the approximate results that our techniques deliver.

As data volumes grow, many queries are increasingly expensive to evaluate accurately. However, an approximate but almost accurate answer that is delivered quickly is often good enough. Approximate query processing techniques [AGP99, AGPR99] developed by the database community aim at quickly determining an approximate answer and, to this end, typically leverage data statistics (often approximated using histograms), sampling, and other data synopses. In contrast to our scenario, approximate query processing techniques target scenarios with a well-designed relational schema that implies certain reasonable queries (e.g., based on foreign keys). When cast into a relational schema, our scenario gives rise to millions of relations (corresponding to terms and their corresponding partitions).

4.9. Summary

In this chapter, we present a framework for efficient approximate processing of keyword queries over a temporally-partitioned inverted index. By using a small synopsis for each partition we identify partitions that maximize the number of final non-redundant results and schedule them for processing early on. Our approach aims to maximize the recall at each stage of query execution given budget on the index-access cost.

Our experimental evaluation shows that our proposed methods can compute more than 80% of final results even when the I/O budget is set as low as 50% of the total size of the partitions that satisfy the temporal predicate. We derive the following insights from our experimental results. Firstly, the choice of the selection methods does not seem to affect the results. Both selection methods are able to deliver a relatively high recall by accessing a small fraction of the index. Secondly, our model for expected query processing time depending on the access costs is vindicated as the wall-clock times seem to be correlated with the fraction of the index accessed. Finally, selection methods are more effective for fine-granular indexes.

5 Phrase Indexing and Querying

5.1. Motivation and Problem Statement

Phrases are sequences of multiple words. Phrase queries are such multi-word sequences, typically expressed with quotation marks (e.g., "the republic of india"). In this chapter we deal with the problem of document retrieval for phrase queries– Given a document collection D and a phrase query p, we intend to find all documents $d \in \mathcal{D}$ that literally contain p. Our focus in this work is on supporting phrase queries more efficiently.

Phrase queries are supported by all modern search engines and are one of their most popular among their *advanced* features. Phrases tend to be unambiguous concept markers [Sal89, CTL91, LC89] and are known to increase precision in search [DLP99]. Treating a query as a phrase yields documents which are closer to the intended concept like *"six pack"*, *"times square"*, *"hurt locker"*. Even when unknown to the user, phrase queries can still be implicitly invoked, for instance, by means of query-segmentation methods [HPBS12, LHZW11]. Query segmentation refers to pre-retrieval algorithms that automatically introduce phrase queries in the user's input.

Beyond their usage in search engines, phrase queries increasingly serve as a building block for other applications such as (a) *entity-oriented search and analytics* [ACCG08] (e.g., to identify documents that refer to a specific entity using one of its known labels), (b) *plagiarism detection* [Sta11] (e.g., to identify documents that contain a highly discriminative fragment from the suspicious document), (c) *culturomics* [MSA+10] (e.g., to identify documents that contain a specific n-gram and compute a frequency time-series from their timestamps).

These applications are also relevant in the context of web archives making phrase queries an important workload type. Search over archives can be extended to use phrase queries by replacing the keyword component in time-travel queries with phrases. Interesting applications which capture entity evolution could potentially use phrase queries to aid entity extraction and tracking over periods of time.

5.1.1. Approach

Traditional approaches to phrase-query processing, as outlined in Chapter 2, uses inverted indexes with postings containing positional information. This posting payload captures where words occur in documents has to be maintained to support phrase queries, which leads to indexes that are larger. Büttcher et. al. [BCC10] report a factor of about 4$\times$ for the inverted index than those required for keyword queries. Phrase query processing, unlike keyword queries where stop words can be ignored, considers all words in the query. It enforces conjunctive query semantics and also devotes extra processing cycles and memory to ensure that the terms occur in the same order as in the query. Consequently, phrase queries are substantially more expensive to process.

The problem of substring matching, which is at the core of phrase queries, has been studied extensively by the string-processing community. However, the solutions developed (e.g., suffix arrays [MM93] and permuterm indexes [FV10]) are designed for main memory and cannot cope with large-scale document collections such as web archives. Solutions developed by the information-retrieval community [TS09, WZB04] build on the inverted index, extending it to index selected multi-word sequences, so-called *phrases*, in addition to single words. The intuition behind indexing phrases into such an augmented index exploit the fact that word sequences are more selective than words resulting in improved response times. However the selection of which phrases to index has been addressed in a limited manner. Typically phrases having a fixed length are selected based on heuristics (e.g., whether they contain a stopword [CP08, WZB04]) or taking into account characteristics of either the document collection [TS09] or the workload [WZB04]. Indexing additional phrases leads to an increase in index size which brings us to the topic of the natural trade-off between inde size and query performance. All the existing approaches barring [TS09] are agnostic to the index size blowup due to indexing extra phrases and hence do not provide size-based index tuning.

Once we construct an augmented index as described before we now turn to how queries are processed using such an enriched vocabulary. With the addition of more terms to the lexicon there are multiple choices as to how a phrase query can be processed. As an example, for a query *"we are the champions"* can be processed by {*"we are"*, *"we are the"*, *"champions"*} or {*"we are"*, *"the champions"*}. A non-optimal choice can sometimes lead to a considerable performance degradation compared to the best choice. In the literature this problem of *phrase-query optimization*, that is, selecting a set of terms to process a given phrase query has been addressed only using greedy heuristics.

We follow the general approach of augmenting the inverted index by selected phrases, but our approach differs in several important aspects. Firstly, it allows for *variable-length phrases* to be indexed while keeping the total index size under a user-specified size budget. We tackle the problem of phrase selection, that is, deciding which phrases should be indexed, by taking into account both the document collection and the workload. The workload indicates how frequent a particular phrase appears in queries. The document collection on the other hand establishes how expensive it is to index a given phrase. We balance both the benefit of usage and storage cost of a phrase to chose a set of phrases which maximize the expected query performance. Secondly, we take a more principled approach to solving the query optimization problem by proposing algorithms which produce an optimal or close-to-optimal set of terms. Note that the "time-travel" aspect is orthogonal to the indexing methods proposed in this chapter. In principle one could partition each posting list corresponding to a multi-word sequence for efficiently processing time-travel queries where the keyword component is a phrase query.

5.1.2. Contributions

We make the following contributions in this chapter.

1. We introduce the *augmented inverted index* as a generalization of existing approaches.
2. We study the problem of *phrase-query optimization*, establish its $\mathcal{NP}$-hardness, and describe an exact exponential algorithm as well as an $O(\log n)$-approximation algorithm to its solution.
3. We propose two *novel phrase-selection methods* tunable by a user-specified space budget that consider characteristics of both the document collection and the workload.
4. We carry out an *extensive experimental evaluation* on ClueWeb09 and a corpus from The New York Times, as two real-world document collections, and entity labels from the YAGO2 knowledge base, as a workload, comparing our approach against state-of-the-art competitors and establishing its efficiency and effectiveness.

With as little as 5% additional space, our approach improves phrase-query processing performance by a factor of more than $3\times$ over a standard positional inverted index, thereby considerably outperforming its competitors.

5.1.3. Organization

The rest of this chapter unfolds as follows. Section 5.2 introduces our formal model. The augmented inverted index is described in Section 5.3. Section 5.4 deals with optimizing phrase queries. Selecting phrases to be indexed is the subject of Section 5.5. Section 5.7 describes our experimental evaluation. We relate our work to existing prior work in Section 5.8 and conclude in Section 5.9.

5.2. Model and Index Organization

We introduce our formal model and the notation used throughout the rest of the chapter. For easy reference we also recap the notation in Table 5.1.

We let $\mathcal{V}$ denote the *vocabulary* of all words. The *set of all non-empty sequences of words* from this vocabulary is denoted $\mathcal{V}^+$. Given a word sequence $\mathbf{s} = \langle s_1, \ldots, s_n \rangle \in \mathcal{V}^+$, we let $|\mathbf{s}| = n$ denote its length. We use $\mathbf{s}[i]$ to refer to the word s_i at the i-th position of $\mathbf{s}$, and $\mathbf{s}[i..j]$ $(i \leq j)$ to refer to the word subsequence $\langle s_i, \ldots, s_j \rangle$.

Definition 5.1 (Position within a sequence) *Given two word sequences* $\mathbf{r}$ *and* $\mathbf{s}$, *we let* $\mathtt{pos}(\mathbf{r}, \mathbf{s})$ *denote the set of positions at which* $\mathbf{r}$ *occurs in* $\mathbf{s}$, *formally*

$$\mathtt{pos}(\mathbf{r}, \mathbf{s}) = \{\, 1 \leq i \leq |\mathbf{s}| \mid \forall\, 1 \leq j \leq |\mathbf{r}| : \mathbf{s}[i+j-1] = \mathbf{r}[j] \,\} \ .$$

For $\mathbf{r} = \langle\, ab \,\rangle$ and $\mathbf{s} = \langle\, cabcab \,\rangle$, as a concrete example, we have $\mathtt{pos}(\mathbf{r}, \mathbf{s}) = \{2, 5\}$. We say that $\mathbf{s}$ *contains* $\mathbf{r}$ if $\mathtt{pos}(\mathbf{r}, \mathbf{s}) \neq \emptyset$. To ease notation, we treat single words from $\mathcal{V}$ also as word sequences when convenient. This allows us, for instance, to write $\mathtt{pos}(w, \mathbf{s})$ to refer to the positions at which w occurs in $\mathbf{s}$.

Until now we operated on versions, however the approaches discussed in this chapter are general enough to be applied to all text collections. Thus for ease of notation we consider a document collection $\mathcal{D}$ of documents $\mathbf{d} \in \mathcal{D}$. Each document $\mathbf{d} \in \mathcal{D}$ is a word sequence from $\mathcal{V}^+$. Since we allow for *duplicate documents*, $\mathcal{D}$ is a bag of word sequences.

We let $\mathcal{W}$ denote our *workload*. Each query $\mathbf{q} \in \mathcal{W}$ is a word sequence from $\mathcal{V}^+$. Since we allow for *repeated queries*, $\mathcal{W}$ is also a bag of word sequences.

Using our notation, we now define the standard notions of *document frequency* and *collection frequency*, as common in Information Retrieval. Let $\mathcal{S}$ be a bag of word sequences (e.g., the document collection or the workload), we define the document frequency of the word sequence $\mathbf{r}$, as the total number of word sequences from $\mathcal{S}$ that contain it, as

$$\mathtt{df}(\mathbf{r}, \mathcal{S}) = |\{\, \mathbf{s} \in \mathcal{S} \mid \mathtt{pos}(\mathbf{r}, \mathbf{s}) \neq \emptyset \,\}| \ .$$

Analogously, its collection frequency, as the total number of occurrences, is defined as

$$\mathtt{cf}(\mathbf{r}, \mathcal{S}) = \sum_{\mathbf{s} \in \mathcal{S}} |\, \mathtt{pos}(\mathbf{r}, \mathbf{s}) \,| \ .$$

Notation	*Description*
$\mathcal{W}$	*workload*, a bag of queries
$\mathbf{q} \in \mathcal{W}$	a query
$\mathcal{D}$	collection of documents
$\mathbf{d} \in \mathcal{D}$	a document
$\mathcal{V}$	*vocabulary*, a set of words
$\mathbf{w} \in \mathcal{V}$	a word
$\mathcal{V}^+$	set of non-empty word sequences
$\mathbf{s} \in \mathcal{V}^+$	a word sequence
$\mathcal{L} \subseteq \mathcal{V}^+$	lexicon, the set of indexed word sequences
$\mathbf{t} \in \mathcal{D}$	an indexed word sequence
$\mathbf{S}(\mathcal{D})$	size of the index for the lexicon $\mathcal{D}$

Table 5.1.: Notation

5.3. Indexing Framework

Having introduced our formal model, we now describe the indexing framework within which we operate.

We build on the *inverted index* as the most widely-used index structure in Information Retrieval that forms the backbone of many real-world systems. The inverted index consists of two components, namely, a *lexicon* $\mathcal{L}$ of *terms* and the corresponding *posting lists* that record for each term information about its occurrences in the document collection. For a detailed discussion of the inverted index and its efficient implementation we refer to Chapter 2.

To support arbitrary phrase queries, an inverted index has to contain all words from the vocabulary in its lexicon (i.e., $\mathcal{V} \subseteq \mathcal{L}$) and record positional information in its posting lists. Thus, the posting $(\,\mathbf{d}_{13},\ \langle 3, 7 \rangle\,)$ found in the posting list for word w conveys that the word occurs at positions 3 and 7 in document $\mathbf{d}_{13}$. More formally, using our notation, a posting $(\,\mathtt{id}(\mathbf{d}),\ \mathtt{pos}(w, \mathbf{d})\,)$ for word w and document $\mathbf{d}$ contains the document's unique identifier $\mathtt{id}(\mathbf{d})$ and the positions $\mathtt{pos}(w, \mathbf{d})$ at which the word occurs.

Query-processing performance for phrase queries on such a *positional inverted index* tends to be limited, in particular for phrase queries that contain frequent words (e.g., stopwords). Posting lists for frequent terms are long, containing many postings each of which with many positions therein, rendering them expensive to read, decompress, and process.

Several authors [CP08, TS09, WZB04] have proposed, as a remedy, to augment the inverted index by adding multi-word sequences, so-called *phrases*, to the set of terms.

The lexicon $\mathcal{L}$ of such an *augmented inverted index* thus consists of *individual words* along-side *phrases* (i.e., $\mathcal{L} \subseteq \mathcal{V}^+$) as terms. Phrase selection can be done, for example, taking into account their selectivity [TS09], whether they contain a stopword [CP08, WZB04], or based on part-of-speech tags [MRS08]. Our approaches to select phrases, which take into account both the document collection and the workload and keep index size within a user-specified space budget, are detailed in Section 5.5.

To process a given phrase query $\mathbf{q}$, a set of terms is selected from the lexicon $\mathcal{L}$, and the corresponding posting lists are intersected to identify documents that contain the phrase. Intersecting of posting lists can be done using *term-at-a-time* (TAAT) or *document-at-a-time query processing* (DAAT). For the former, posting lists are read one after each other, and bookkeeping is done to keep track of positions at which the phrase can still occur in candidate documents. For the latter, posting lists are read in parallel and a document, when seen in all posting lists at once, is examined for whether it contains the phrase sought. In both cases, the cost of processing a phrase query depends on the sizes of posting lists read and thus the set of terms selected to process the query.

5.4. Query Optimization

In this section, we describe how a phrase query $\mathbf{q}$ can be processed using a given augmented inverted index with a concrete lexicon $\mathcal{L}$. Our objective is thus to determine, *at query-processing time*, a subset $\mathcal{P} \subseteq \mathcal{L}$ of terms, further referred to as *query plan*, that can be used to process $\mathbf{q}$.

To formulate the problem, we first need to capture when a query plan $\mathcal{P}$ can be used to process a phrase query $\mathbf{q}$. Intuitively, each word must be covered by at least one term from $\mathcal{P}$.

Definition 5.2 (Cover of a query)

$$\begin{aligned} \mathsf{covers}(\mathcal{P}, \mathbf{q}) \;=\; & \forall\, 1 \leq i \leq |\,\mathbf{q}\,| : \exists\, \mathbf{t} \in \mathcal{P} : \exists\, j \in \mathsf{pos}(\mathbf{q}[i], \mathbf{t}) : \\ & \forall\, 1 \leq k \leq |\,\mathbf{t}\,| : \mathbf{q}[i - j + k] = \mathbf{t}[k] \end{aligned}$$

Consider a lexicon

$$\mathcal{L} = \{\,\langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, c \,\rangle, \langle\, d \,\rangle, \langle\, ab \,\rangle, \langle\, bc \,\rangle, \langle\, cd \,\rangle\,\}.$$

The phrase query $\mathbf{q} = \langle\, abc \,\rangle$, for instance, can be processed using $\{\,\langle\, ab \,\rangle, \langle\, bc \,\rangle\,\}$ but not $\{\,\langle\, ab \,\rangle, \langle\, cd \,\rangle\,\}$.

Without the augmented index, each query term was covered by exactly one term. With the augmented index there are more choices to process each query term. The choices for covering b are $\langle\,\mathsf{b}\,\rangle, \langle\,\mathsf{ab}\,\rangle$ and $\langle\,\mathsf{bc}\,\rangle$. Multiple choices for processing each position in the query leads to combinatorial choices for covering the query. In order to chose the best query plan we would need to quantify the cost of each query cover.

As detailed above, in Section 5.3, different ways of processing a phrase query (i.e., TAAT vs. DAAT) exist. In the worst case, regardless of which query-processing method is employed, all posting lists have to be read in their entirety. We model the cost of a query plan $\mathcal{P}$ as the total number of postings that has to be read

$$c(\mathcal{P}) = \sum_{t \in \mathcal{P}} df(t, C)\ .$$

While posting sizes are not uniform (e.g., due to compression and varying numbers of contained positions), which may suggest collection frequency as a more accurate cost measure, we found little difference in practice and thus, for simplicity, stick to document frequency in this work. This is in line with [MTO12], who found that aggregate posting-list lengths is the single feature most correlated with response time for full query evaluation, as required for phrase queries, which also do not permit dynamic pruning.

Prior work in [TS09, WZB04] use cost-based heuristics to determine a valid query plan. A candidate set of sequences ρ is first determined which are both present in the query as well as the lexicon, i.e.,

$$\rho \;=\; \{\, r \mid r \in \mathcal{L} \wedge pos(r, q) \neq \phi \,\}.$$

Candidates are then greedily chosen from R in the order of ascending df values until the entire query is covered. In every round the best candidate which overlaps with the uncovered regions of the query is chosen. However, such a heuristic algorithm might not yield an optimal plan. We illustrate this with an example. Consider a lexicon $\mathcal{L}$ with the following df values

$$\mathcal{L} = \{\,\langle\,a\,\rangle, \langle\,b\,\rangle, \langle\,c\,\rangle, \langle\,d\,\rangle, \langle\,ab\,\rangle, \langle\,bc\,\rangle, \langle\,cd\,\rangle\,\} \quad \left| \quad \begin{array}{c} df(\langle\,ab\,\rangle) = 10 \\ df(\langle\,ab\,\rangle) = 10 \\ df(\langle\,cd\,\rangle) = 60 \end{array} \right.$$

A phrase query $\mathbf{q} = \langle\,abcd\,\rangle$ evaluated according to the greedy heuristic above yield the following plan – $GRD = \{\,\langle\,ab\,\rangle, \langle\,bc\,\rangle, \langle\,cd\,\rangle\,\}$ – with an cost $c(GRD) = 120$. It is easy to see that the optimal cover OPT of this query is $OPT = \{\,\langle\,ab\,\rangle, \langle\,cd\,\rangle\,\}$ with an cost $c(OPT) = 70$.

Theoretically, one can construct a worst case scenario which can lead to arbitrary degradation of performance. As an example let there be a query $q = \langle\,q_1 \cdots q_m\,\rangle$. Assuming that only bi-grams are indexed and the distribution of df values are such that

$\mathrm{df}(\langle q_i q_{i+1} \rangle) < \mathrm{df}(\langle q_{i+1} q_{i+2} \rangle)$. The cost of the query plan using the greedy heuristic is $\sum_{1 \le i < m} \mathrm{df}(\langle q_i q_{i+1} \rangle)$.

We take a more principled approach and model the query-optimization as an optimization problem. Assembling the above definitions of coverage and cost, we now formally define the problem of finding a cost-minimal query plan $\mathcal{P}$ for a phrase query $\mathbf{q}$ and lexicon $\mathcal{L}$ as the following $\mathcal{NP}$-hard optimization problem

Definition 5.3 PHRASE-QUERY OPTIMIZATION

$$\operatorname*{argmin}_{\mathcal{P} \subseteq \mathcal{L}} c(\mathcal{P}) \quad \textit{s.t.} \quad \mathrm{covers}(\mathcal{P}, \mathbf{q}) \; .$$

Theorem 5.1 PHRASE-QUERY OPTIMIZATION *is $\mathcal{NP}$-hard.*

Proof: *Our proof closely follows Neraud [Nér90], who establishes that deciding whether a given set of strings is elementary is $\mathcal{NP}$-complete. We show through a reduction from* VERTEX COVER *that the decision variant of* PHRASE-QUERY OPTIMIZATION *(i.e., whether a $\mathcal{P}$ with $c(\mathcal{P}) \le \tau$ exists) is $\mathcal{NP}$-complete, from which our claim follows.*

Let $G(V, E)$ be an undirected graph with vertices V and edges E. We assume that there are no isolated vertices, i.e., each vertex has at least one incident edge. VERTEX COVER *asks whether there exists a subset of vertices $VC \subseteq V$ having cardinality $|\,VC\,| \le k$, so that*

$$\forall\, (u, v) \in E \; : \; u \in VC \vee v \in VC,$$

that is, for each edge one of its connected vertices is in the vertex cover.

We obtain an instance of PHRASE-QUERY OPTIMIZATION *from $G(V, E)$ as follows:*

- *$\mathcal{V} = V \cup E$ – we introduce a word to the vocabulary for each vertex (v) and edge ($\overline{uv}$) in the graph;*
- *$\mathbf{q} = \biguplus_{(u,v) \in E} u \; \overline{uv} \; v$ – we obtain the query $\mathbf{q}$ as a concatenation of all edge words $\overline{uv}$ bracketed by the words of their source (u) and target (v);*
- *$\mathcal{D} = \{\, \mathbf{q} \,\}$ – the document collection contains only a single document that equals our query;*
- *$\mathcal{L} = \mathcal{V} \quad \cup \bigcup_{(u,v) \in E} \{\, \langle\, u \; \overline{uv} \,\rangle \,\} \quad \cup \bigcup_{(u,v) \in E} \{\, \langle\, \overline{uv} \; v \,\rangle \,\}$ – each vertex word (v) and edge word ($\overline{uv}$) is indexed as well as each combination of edge and source ($u\,\overline{uv}$) and edge and target ($\overline{uv}\,v$).*

This can be done in polynomial time and space. Note that, by construction, $\forall\, \mathbf{t} \in \mathcal{L} \; : \; \mathrm{df}(\mathbf{t}, \mathcal{D}) = 1$ holds. We now show that $G(V, E)$ has a vertex cover with cardinality $|\,VC\,| \le k$ iff there is a query plan $\mathcal{P}$ with $c(\mathcal{P}) \le k + |\,E\,|$.

($\Rightarrow$) Observe that VC *contains at least one of* u *or* v *for each* $u\ \overline{uv}\ v$ *from the query, incurring a cost of* $|\mathrm{VC}| \leq k$. *We can complement this to obtain a query plan* $\mathcal{P}$ *by adding exactly one term* ($\langle\,\overline{uv}\,\rangle$, $\langle\, u\,\overline{uv}\,\rangle$, *or* $\langle\,\overline{uv}\,v\,\rangle$) *for each* $u\ \overline{uv}\ v$ *from the query, incurring a cost of* $|\,E\,|$. *Thus,* $c(\mathcal{P}) \leq k + |\,E\,|$ *holds.*

($\Leftarrow$) Observe that $\mathcal{P}$ *must cover each* $u\ \overline{uv}\ v$ *from the query in one of four ways: (i)* $\langle\,u\,\rangle\langle\,\overline{uv}\,\rangle\langle\,v\,\rangle$, *(ii)* $\langle\,u\,\rangle\langle\,\overline{uv}\,v\,\rangle$, *(iii)* $\langle\,u\,\overline{uv}\,\rangle\langle\,v\,\rangle$ *(iv)* $\langle\,u\,\overline{uv}\,\rangle\langle\,\overline{uv}\,v\,\rangle$. *Whenever a* $u\ \overline{uv}\ v$ *from the query is covered as* $\langle\,u\,\overline{uv}\,\rangle\langle\,\overline{uv}\,v\,\rangle$, *we replace* $\langle\,u\,\overline{uv}\,\rangle$ *by* $\langle\,u\,\rangle$, *thus reducing case (iv) to case (ii). We refer to the query plan thus obtained as* $\mathcal{P}'$. *Note that* $c(\mathcal{P}') \leq c(\mathcal{P})$, *since all terms have the same cost.* $\mathcal{P}'$ *contains exactly one term* ($\langle\,\overline{uv}\,\rangle$, $\langle\,u\,\overline{uv}\,\rangle$, *or* $\langle\,\overline{uv}\,v\,\rangle$) *for each* $u\ \overline{uv}\ v$ *from the query, incurring a cost of* $|\,E\,|$. *Let* VC *be the set of vertices whose words are contained in* $\mathcal{P}'$. *We can thus write* $c(\mathcal{P}') = |\mathrm{VC}| + |\,E\,|$. VC *is a vertex cover, since after eliminating case (iv), each* $u\ \overline{uv}\ v$ *from the query is covered using either* $\langle\,u\,\rangle$ *or* $\langle\,v\,\rangle$. *Thus,* $c(\mathcal{P}') = |\mathrm{VC}| + |\,E\,| \leq c(\mathcal{P}) \leq |\,E\,| + k \Rightarrow |\mathrm{VC}| \leq k$. □

5.4.1. Optimal Solution

If an optimal query plan $\mathcal{P}^*$ exists, so that every term therein occurs exactly once in the query, we can determine an optimal query plan using dynamic programming based on the recurrence

$$\textsc{Opt}(i) = \begin{cases} df(\mathbf{q}[1..i], \mathcal{D}) & : \mathbf{q}[1..i] \in \mathcal{L} \\ \min\limits_{j<i} \left(\textsc{Opt}(j) + \min\limits_{\substack{k \leq j+1 \,\wedge \\ \mathbf{q}[k..i] \in \mathcal{L}}} df(\mathbf{q}[k..i], \mathcal{D}) \right) & : \text{otherwise} \end{cases}$$

in time $\mathcal{O}(n^2)$ and space $\mathcal{O}(n)$ where $|\,\mathbf{q}\,| = n$. $\textsc{Opt}(i)$ denotes the cost of an optimal solution to the prefix subproblem $\mathbf{q}[1..i]$ – once the dynamic-programming table has been populated, an optimal query plan can be constructed by means of backtracking. In the first case, the prefix subproblem can be covered using a single term. In the second case, the optimal solution combines an optimal solution to a smaller prefix subproblem, which is the optimal substructure inherent to dynamic programming, with a single term that covers the remaining suffix.

Theorem 5.2 *If there is an optimal query plan* $\mathcal{P}^*$ *for a phrase query* $\mathbf{q}$ *such that*

$$\forall\, \mathbf{t} \in \mathcal{P}^* \ : \ |\,pos(\mathbf{t}, \mathbf{q})\,| = 1 \ ,$$

then $c(\mathcal{P}^*) = \textsc{Opt}(|\,\mathbf{q}\,|)$, *that is, an optimal solution can be determined using the recurrence* $\textsc{Opt}$.

Proof: *Observe that Theorem 5.2 is a special case of Theorem 5.4 for* $\mathcal{F} = \emptyset$. *We therefore only prove the more general Theorem 5.4.* □

It entails that we can efficiently determine optimal query plans for phrase queries that do not contain any repeated words.

Corollary 5.3 *We can compute an optimal query plan for a phrase query* $\mathbf{q}$ *in polynomial time and space, if*

$$\forall 1 \leq i \leq |\mathbf{q}| \; : \; |\mathrm{pos}(\mathbf{q}[i], \mathbf{q})| = 1 \; .$$

In practice this special case is important: We found that about 99% of phrase queries in our workload do not contain any repeated words, as detailed in Section 5.7.

Algorithm We present the algorithm which efficiently implements the recurrence relation. The input to the algorithm are the query $\mathbf{q}$ and the lexicon $\mathcal{L}$. A set of candidate word sequences $\mathcal{R}$ is constructed such that $\mathrm{Cand} = \{ \mathbf{s} \in \mathcal{L} \; : \; |\mathrm{pos}(\mathbf{s}, \mathbf{q})| > 1 \}$. Let us denote the begin and of a sequence $\mathbf{s} \in \mathcal{R}$ as $\mathrm{begin}(\mathbf{s})$ and end of the sequence as $\mathrm{end}(\mathbf{s})$. For example, $\mathrm{begin}(\mathbf{q}) = 0$ and $\mathrm{end}(\mathbf{q}) = |\mathbf{s}| - 1$. The cost of each sequence $\mathbf{s}$ is denoted by $c(\mathbf{s}) = \mathrm{df}(\mathbf{s})$.

We note that the sequences have a anti-monotonocity property on their costs as stated below which we utilize to prune the candidate set of sequences.

Lemma 5.1 (Cost-anti monotonicity of sequences) *For a pair of sequences* r *and* s*, it holds*

$$|\mathrm{pos}(r, s)| > 1 \quad \Longrightarrow \quad \mathrm{df}(r) \geq \mathrm{df}(s).$$

Proof: *Every occurence of a sequence of* s *in the collection is also an occurence of* r. □

We exploit the cost-anti monotonicity property of sequences to eliminate sequences in $\mathcal{R}$ which are substrings of other candidate sequences. If the entire query $\mathbf{q}$ is indexed, i.e, $\mathbf{q} \in \mathcal{L}$, then all the remaining candidates are pruned out and we can terminate immediately.

After the pruning step it is easy to see that each there can be only one sequence which begins from or ends at any given position $0 \leq i < |\mathbf{q}|$ in the query. We organize the remaining sequences into a list $\mathcal{I}$ ordered by their increasing end positions $\mathrm{end}(\mathbf{s})$, i.e,

$$\mathcal{I} = \langle \mathbf{s}_i \rangle \; : \; \mathbf{s}_i \in \mathrm{Cand} \; \wedge \; \nexists \mathbf{s}_j \in \mathcal{I}, \; \mathrm{pos}(\mathbf{s}_i, \mathbf{s}_j) > 0 \; \wedge \; \mathrm{end}(s_i) \leq \mathrm{end}(s_{i+1}).$$

Dynamic programming proceeds by computing optimal solutions $\textsc{Opt}(i)$ to the prefix subproblems $\mathbf{q}[1...i]$, i.e, optimal cost of covering the first i positions of (q). Since $\mathcal{I}$ is ordered ends of sequences, in every iteration we consider a new sequence $\mathbf{s}$ which increases the problem size to $\mathbf{q}(1...\mathrm{end}(\mathbf{s}))$. We compute $\textsc{Opt}(\mathrm{end}(\mathbf{s}))$ by trying to minimize $\textsc{Opt}(j) + c(\mathbf{s})$ for $1 < j < \mathrm{end}(\mathbf{s})$. We also ensure that the new solution $\textsc{Opt}(\mathrm{end}(\mathbf{s}))$ also a cover for $\mathbf{q}(1...\mathrm{end}(\mathbf{s}))$. Thus $\mathbf{s})$ is the last sequence in the optimal solution for the $\mathbf{q}(1...\mathrm{end}(\mathbf{s}))$. We store these last sequences for optimal subproblems in a queue Q. Thus each sequence $\mathbf{t}$ in a queue encodes the solution to the subproblem $\mathbf{q}(1...\mathtt{end}(\mathbf{t}))$, i.e., $\textsc{Opt}(\mathtt{end}(\mathbf{t}))$. Since each element of Q is the last sequence in their respective subproblems, the future sequences considered only have to check overlaps with the sequences in the queue to ensure that the subproblems that they represent are covered. We let $\mathtt{last}(Q)$ denote the sequence at the end of the queue or the last sequence added.

We establish an invariance in plan cost, i.e., $\textsc{Opt}(i+1) > \textsc{Opt}(i)$ over the sequences in Q. Thus if an incoming sequence computes with end i has $\textsc{Opt}(i)$ greater than earlier values of $\textsc{Opt}$ in the queue, then those sequences are removed until the invariance is established. After determining the optimal solution for which $\mathbf{s}$ is the last sequence, and maintaining the invariance by pruning out sequences in Q, $\mathbf{s}$ is appended to the end of Q.

After we have exhausted all input values from $\mathcal{I}$, as a final step, we backtrack from the last sequence representing $\textsc{Opt}(|\mathbf{q}|)$ to construct the plan with the minimum cost (c.f. lines 18-21). Then invariance improves the efficiency of the algorithm by decreasing the number of comparisons (i) while determining a feasible cover (line 7) and (ii) while backtracking for the determination of the final plan.

Queries with Repetition Otherwise, when there is no optimal query plan $\mathcal{P}^*$ according to Theorem 5.2, dynamic programming can not be directly applied, since there is no optimal substructure. Consider, as a concrete problem instance, the phrase query $\mathbf{q} = \langle\, abxayb \,\rangle$ with lexicon $\mathcal{L} = \{\langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, x \,\rangle, \langle\, y \,\rangle, \langle\, ab \,\rangle\}$ and assume $\mathtt{df}(\mathbf{t}, \mathcal{D}) > 1$ for $\mathbf{t} \in \{\langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, x \,\rangle, \langle\, y \,\rangle\}$ and $\mathtt{df}(\langle\, ab \,\rangle, \mathcal{D}) = 1$. For this problem instance, the optimal solution $\mathcal{P}^* = \{\langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, x \,\rangle, \langle\, y \,\rangle\}$ does not contain an optimal solution to any prefix subproblem $\mathbf{q}[1..i]$ $(1 < i < |\,\mathbf{q}\,|)$, which all contain the term $\langle\, ab \,\rangle$.

However, as we describe next, an optimal query plan can be computed, in the general case, using a combination of exhaustive search over sets of repeated terms and a variant of our above recurrence.

For a phrase query $\mathbf{q}$ let the set of repeated terms be formally defined as

$$\mathcal{R} = \{\, \mathbf{t} \in \mathcal{L} \;|\; |\,\mathtt{pos}(\mathbf{t}, \mathbf{q})\,| > 1 \,\} \;.$$

Let further $\mathcal{F} \subseteq \mathcal{R}$ denote a subset of repeated terms. We now define a modified document frequency that is zero for terms from $\mathcal{F}$, formally

$$\mathtt{df}'(\mathbf{t}, \mathcal{D}) = \begin{cases} 0 & : \quad \mathbf{t} \in \mathcal{F} \\ \mathtt{df}(\mathbf{t}, \mathcal{D}) & : \quad \text{otherwise} \end{cases}$$

and denote by $\textsc{Opt}'$ the variant of our above recurrence that uses this modified document frequency.

Algorithm 7: Dynamic programming solution for optimal plan for queries with non-repeated tokens

```
Input: Phrase query $\mathbf{q}$, lexicon $\mathcal{L}$
Output: Cost $\mathtt{optCost}$ of optimal query plan
begin
    $\mathtt{Cand} = \{\mathbf{s} \in \mathcal{L} \,:\, |\mathtt{pos}(\mathbf{s}, \mathbf{q})| > 0\}$;
    $\mathcal{I} = \langle \mathbf{s}_i \rangle \,:\, \mathbf{s}_i \in \mathtt{Cand} \;\wedge\; \nexists \mathbf{s}_j \in \mathcal{I},\; \mathtt{pos}(\mathbf{s}_i, \mathbf{s}_j) > 0 \;\wedge\; \mathtt{end}(s_i) \leq \mathtt{end}(s_{i+1})$;
    $Q \longleftarrow \langle i_1 \rangle$;
    //Q is the queue of processed sequences.

    for $\mathbf{s} \in \mathcal{I}$ do
        $i_{min} = \underset{i_k \in Q \,\wedge\, \mathtt{begin}(i_k) \leq \mathtt{begin}(\mathbf{s}) \leq \mathtt{begin}(i_k)}{\mathtt{argmin}} (\mathtt{end}(i_k) - \mathtt{begin}(\mathbf{s}))$;

        // $i_{min}$ is the sequence with the minimum intersection with $i_j$. if $i_{min} = \emptyset$ then
            $\textsc{Opt}(i_{min}) = 0$ ;
        $\textsc{Opt}(\mathtt{end}(\mathbf{s})) = \textsc{Opt}(i_{min}) + c(\mathbf{s})$;

        // Preserving increasing score order in Q
        while $\textsc{Opt}(\mathtt{end}(Q.\mathtt{last})) > \textsc{Opt}(\mathtt{end}(\mathbf{s}))$ do
            $\mathtt{removeLastSequence}(Q)$;
        $\mathtt{AppendInterval}(\mathbf{s}, Q)$ ;
    $\mathtt{optCost} = \textsc{Opt}(\mathtt{end}(Q.\mathtt{last}))$;

    // Backtracking for plan construction
    $i_{next} \leftarrow Q.\mathtt{last}$ ;
    while $i_{next} \neq Q.\mathtt{first}$ do
        $\mathcal{P} \leftarrow \mathcal{P} \cup i_{next}$ ;
        $i_{next} = \underset{i_k \in V \wedge (i_{next} \cap i_k \neq \emptyset)}{\mathtt{argmin}} (\mathtt{end}(i_k) - \mathtt{begin}(i_{next}))$
end
```

Algorithm 8 considers all subsets of repeated terms and, for each of them, extends it into a query plan for $\mathbf{q}$ by means of the recurrence $\textsc{Opt}'$. The algorithm keeps track of the best solution seen and eventually returns it. Its correctness directly follows from the following theorem.

Algorithm 8: Phrase-query optimization

```
Input: Phrase query $\mathbf{q}$, lexicon $\mathcal{L}$
Output: Cost optCost of optimal query plan
1 $\mathcal{R} = \{ \mathbf{t} \in \mathcal{L} : |\mathrm{pos}(\mathbf{t}, \mathbf{q})| > 1 \}$
2 optCost = $\infty$
3 for $\mathcal{F} \in 2^{\mathcal{R}}$ do
4     cost = $c(\mathcal{F}) + \textsc{Opt}'(|\mathbf{q}|)$
5     if cost < optCost then
6         optCost = cost
7 return optCost
```

Theorem 5.4 *Let $\mathcal{P}^*$ denote an optimal query plan for the phrase query $\mathbf{q}$ and let*

$$\mathcal{F} = \{ \mathbf{t} \in \mathcal{P}^* \mid |\mathrm{pos}(\mathbf{t}, \mathbf{q})| > 1 \}$$

be the set of repeated terms therein, then

$$c(\mathcal{F}) + \textsc{Opt}'(|\,\mathbf{q}\,|) \leq c(\mathcal{P}^*) \ .$$

Proof: L*et $\mathcal{P}^*$ denote an optimal query plan for the phrase query $\mathbf{q}$ and*

$$\mathcal{F} = \{ \mathbf{t} \in \mathcal{P}^* \mid |\mathrm{pos}(\mathbf{t}, \mathbf{q})| > 1 \}$$

be the set of repeated terms and $\bar{\mathcal{F}} = \mathcal{P}^* \setminus \mathcal{F}$ be the set of non-repeated terms therein. Without loss of generality, we assume that $\mathbf{q}$ ends in a non-repeated terminal term # having $\mathrm{df}(\#, \mathcal{D}) = 0$ – this can always be achieved by "patching" the query. We order non-repeated terms $\mathbf{t} \in \bar{\mathcal{F}}$ by their single item in $\mathrm{pos}(\mathbf{t}, \mathbf{q})$ to obtain the sequence $\langle \mathbf{t}_1, \ldots, \mathbf{t}_m \rangle$ with $m = |\bar{\mathcal{F}}|$. We refer to the first position covered by $\mathbf{t}_i$, corresponding to the single item in $\mathrm{pos}(\mathbf{t}, \mathbf{q})$, as b_i and to the last position as $e_i = (b_i + |\,\mathbf{t}_i\,| - 1)$.

We now show by induction *that*

$$\textsc{Opt}'(e_i) \leq \sum_{j=1}^{i} \mathrm{df}(\mathbf{t}_j, \mathcal{D}) = c(\mathcal{P}^*) - c(\mathcal{F}) \ .$$

$(i = 1)$ *We have to distinguish two cases: (i) $b_1 = 1$, that is, $\mathbf{q}[1..e_1]$ is covered using a single non-repeated term –* Opt′ *selects this term according to its first case. (ii) $b_1 > 1$, that is, there is a set of repeated terms from $\mathcal{F}$ that covers $\mathbf{q}[1..k]$ for some $b_1 - 1 \leq k < e_1$ –* Opt′ *can select the same repeated terms at zero cost and combine it with $\mathbf{t}_1$ that covers $\mathbf{q}[b_1..e_1]$. Thus, in both cases, $\textsc{Opt}'(e_1) \leq \mathrm{df}(\mathbf{t}_1, \mathcal{D})$.*

$(i \rightarrow i+1)$ *We assume* $\text{OPT}'(e_i) \leq \sum_{j=1}^{i} \mathrm{df}(\mathbf{t}_j, \mathcal{D})$. *Again, we have to distinguish two cases: (i)* $e_i \geq b_{i+1} - 1$, *that is, the term before* $\mathbf{t}_{i+1}$ *is also a non-repeated term. Thus, our recurrence considers* $\text{OPT}'(e_i) + \mathrm{df}(\mathbf{t}_{i+1}, \mathcal{D})$ *as one possible solution. (ii)* $e_1 < b_{i+1} - 1$, *that is, there is a gap covered by repeated terms between* $\mathbf{t}_i$ *and* $\mathbf{t}_{i+1}$ *–* OPT' *can select the same repeated terms at zero cost and thus considers* $\text{OPT}'(e_i) + 0 + \mathrm{df}(\mathbf{t}_{i+1}, \mathcal{D})$ *as one solution. Thus, in both cases,* $\text{OPT}'(e_{i+1}) \leq \sum_{j=1}^{i+1} \mathrm{df}(\mathbf{t}_j, \mathcal{D})$. □

The cost of Algorithm 8 depends on the number of repeated terms $|\mathcal{R}|$, which is small in practice and can be bounded in terms of the number of positions in $\mathbf{q}$ occupied by a repeated word

$$r = |\{ 0 \leq i \leq |\mathbf{q}| \mid |\mathrm{pos}(\mathbf{q}[i], \mathbf{q})| > 1 \}| \ .$$

For our above example phrase query $\mathbf{q} = \langle\, abxayb \,\rangle$ we obtain $r = 4$. Note that the following holds $|R| \leq \frac{r\cdot(r+1)}{2}$. Algorithm 8 thus has time complexity $\mathcal{O}(2^{\frac{r\cdot(r+1)}{2}} n^2)$ and space complexity $\mathcal{O}(n^2)$ where $|\mathbf{q}| = n$.

5.4.2. Approximation Guarantee

Computing an optimal query plan can be computationally expensive in the worst case, as just shown. We observe that our query-optimization problem can be seen as an instance of SET COVER [Vaz01]. This means that we can re-use well known approximation algorithms for SET COVER which are known to be efficient. We state the SET COVER problem and show how our query-optimization problem is its instance.

Definition 5.4 (SET COVER problem) *Given a universe* $\mathcal{U}$ *of* n *elements, a collection of subsets* $\mathcal{U}$, $\mathcal{S} = \{S_1, \ldots, S_n\}$, *and a cost function* $c : \mathcal{S} \implies \mathbf{Q}^+$, *finds a minimum cost subcollection of* $\mathcal{S}$ *that covers all elements of* $\mathcal{U}$.

To this end, we convert an instance of our problem, consisting of a phrase query $\mathbf{q}$ and a lexicon $\mathcal{L}$ with associated costs, into a SET COVER instance as follows: Let the universe of items $\mathcal{U} = \{1, \ldots, |\mathbf{q}|\}$ correspond to positions in the phrase query. For each term $\mathbf{t} \in \mathcal{L}$, we define a subset $S_\mathbf{t} \subseteq \mathcal{U}$ of covered positions as

$$S_\mathbf{t} = \{ 1 \leq i \leq |\mathbf{q}| \mid \exists j \in \mathrm{pos}(\mathbf{t}, \mathbf{q}) \ : \ j \leq i < j + |\mathbf{t}| \} \ .$$

The collection of subsets of $\mathcal{U}$ is then defined as

$$\mathcal{S} = \{ S_\mathbf{t} \mid \mathbf{t} \in \mathcal{L} \}$$

and we define $\mathrm{cost}(S_\mathbf{t}) = \mathrm{df}(\mathbf{t}, \mathcal{D})$ as a cost function.

For our concrete problem instance $\mathbf{q} = \langle\, abxayb \,\rangle$ and $\mathcal{L} = \{ \langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, x \,\rangle, \langle\, y \,\rangle, \langle\, ab \,\rangle \}$ from above, we obtain $\mathcal{U} = \{1, \ldots, 6\}$ and $\mathcal{S} = \{\{1,4\},\{2,6\},\{3\},\{5\},\{1\}\}$ as the corresponding SET COVER instance.

We can now use the greedy algorithm which is known to be a $\mathcal{O}(\log n)$-approximation algorithm [Vaz01]. The greedy algorithm iteratively the most cost effective set from $\mathcal{S}$ until it covers all elements. The cost-effectiveness is defined as a benefit-cost ratio where benefit is the number of yet-uncovered elements and the cost is $\mathtt{cost}(S_t)$. This can be implemented in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space where $|\,q\,| = n$.

Note that, as a key difference to the greedy algorithm described in [WZB04], which to the best of our knowledge does not give an approximation guarantee, our greedy algorithm (**APX**) selects subsets (corresponding to terms from the lexicon) taking into account the number of additional items covered and the coverage already achieved by selected subsets. This is unlike the approach in [WZB04], referred to as **GRD**, where items are also chosen greedily based solely on their costs. The number of uncovered items is not factored in and hence they do not make a distinction if one element is uncovered or many. As an example consider a $\mathbf{q} = \langle\, abcd \,\rangle$ and a lexicon $\mathcal{L}$ with the following costs:

$$\mathcal{L} = \{\, \langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, c \,\rangle, \langle\, d \,\rangle, \langle\, ab \,\rangle, \langle\, bc \,\rangle, \langle\, cd \,\rangle \,\} \;\left|\; \begin{array}{c} \mathtt{cost}(\langle\, ab \,\rangle) = 10 \\ \mathtt{cost}(\langle\, bc \,\rangle) = 20 \\ \mathtt{cost}(\langle\, cd \,\rangle) = 30 \end{array}\right.$$

GRD would in the first choose $\langle\, ab \,\rangle$, followed by $\langle\, bc \,\rangle$ and finally $\langle\, cd \,\rangle$. However, **APX** after choosing $\langle\, ab \,\rangle$ in the first round would choose $\langle\, cd \,\rangle$ over $\langle\, bc \,\rangle$ which has a better benefit-cost ratio because of larger number of uncovered positions.

5.5. Phrase Selection

Having described how phrase queries can be efficiently processed on a given augmented inverted index, we now turn to the complementary problem of phrase-selection. We identify two key ingredients on which we build upon. Firstly, we analyze the workload to determine the most frequently used word sequences. The frequency of usage of a sequence is an indicator that it is an important sequence and a potential component in many queries in the workload. But a word sequence might be frequent, in the workload or collection or both, but might be expensive in terms of query processing and storage. Consider the phrase "of the", which is a frequent sequence in the workload but also frequent in the document collection. A high $\mathtt{df}$ value of the phrase means that the corresponding posting list would be big leading high storage overhead. Since we model our query processing cost as the sum of $\mathtt{df}$ values of the participating terms, it also means that using "of the" has performance overheads as well. However, note that the performance using "of the" is still better than using the single terms "of" and "the". This brings us to the second ingredient which is the cost of using a certain word sequence. We determine the cost of usage of the word sequence from the occurrence in the document collection. In our methods we balance these two ingredients to select a set of sequences to be indexed which are perceived to improve query performance.

In what follows, we introduce two phrase-selection methods that determine, at index-build time, which phrases should be included in the lexicon, taking into account both characteristics of the document collection and the workload.

5.5.1. Query-Optimizer-Based Phrase Selection

Our first method, coined query-optimizer-based phrase-selection (QOBS), builds on Section 5.4 and considers how phrase queries from the workload would actually be processed if a specific lexicon was available.

Let $c(\mathbf{q}, \mathcal{L})$ denote the cost of processing the phrase query $\mathbf{q}$ with lexicon $\mathcal{L}$, in terms of the number of postings that have to be read, determined using one of the query-optimization methods from Section 5.4. We define the expected cost of processing a phrase query from the workload $\mathcal{W}$ with lexicon $\mathcal{L}$ as

$$c(\mathcal{W}, \mathcal{L}) = \frac{1}{|\mathcal{W}|} \sum_{\mathbf{q} \in \mathcal{W}} c(\mathbf{q}, \mathcal{L}) \; .$$

Recall that $\mathcal{W}$ is a bag of word sequences, so that repeated phrase queries are taken into account. The benefit of having the lexicon $\mathcal{L}$ instead of only single words from $\mathcal{V}$, which serves as our baseline as described in Section 5.3, as the improvement in expected processing cost is defined as

$$b(\mathcal{L}) = c(\mathcal{W}, \mathcal{V}) - c(\mathcal{W}, \mathcal{L}) \; .$$

The space consumed by the augmented inverted index with lexicon $\mathcal{L}$ is captured as

$$s(\mathcal{L}) = \sum_{\mathbf{t} \in \mathcal{L}} df(\mathbf{t}, \mathcal{D}) \; ,$$

corresponding to the total number of postings in the index.

Our objective is to compile a lexicon that minimizes the expected processing cost for phrase queries from the workload and results in an index whose size is within a user-specified space budget. We model the user-specified space budget as a percentage overhead α $(0 \leq \alpha)$ over the size of our baseline augmented inverted index having lexicon $\mathcal{L} = \mathcal{V}$. We obtain the optimization problem

Definition 5.5 QUERY-OPTIMIZER-BASED PHRASE SELECTION

$$\underset{\mathcal{V} \subseteq \mathcal{L} \subseteq \mathcal{V}^{+}}{\operatorname{argmax}} \; b(\mathcal{W}, \mathcal{L}) \quad \textit{s.t.} \quad s(\mathcal{L}) \leq (1 + \alpha) \cdot s(\mathcal{V}) \; .$$

Observe that our formulation implicitly encodes the trade-off between how frequent a phrase is and how expensive it is to process. We obtain the first information from the workload and the second from the document collection. A phrase $\mathbf{s}$ having low $\mathtt{df}(\mathbf{s}, \mathcal{L})$ and high $\mathtt{df}(\mathbf{s}, \mathcal{W})$ is more likely to be selected. In addition, our formulation takes into account the cost of the terms that a phrase substitutes. For two candidate phrases $\mathbf{s}$ and $\mathbf{r}$ with $\mathtt{df}(\mathbf{s}, \mathcal{L}) = \mathtt{df}(\mathbf{r}, \mathcal{L})$, the one that substitutes more expensive terms is preferred.

Unfortunately, our optimization problem is $\mathcal{NP}$-hard as we prove below.

Theorem 5.5 QUERY-OPTIMIZER-BASED PHRASE SELECTION *is $\mathcal{NP}$-hard.*

Proof: *We show this through reduction from the 0-1 or binary* KNAPSACK PROBLEM *which is proven to be NP complete [LSS88]. In the* binary knapsack problem*, we have a set of* n *items* $\{e_i\}$ *associated with integral weights* $\{w_i\}$ *and profits* $\{p_i\}$*. We want to select items to put in the knapsack such that the sum of the profits is maximized and the sum of weights is less than* L *- which is the size of the knapsack.*

We can map every instance of the binary problem into our problem by the following construction: (i) We construct a query workload with n *two word queries, each query* $\langle\, a_i b_i \,\rangle$ *corresponding to an item* e_i*, where both words are different and also requiring that no two queries share a word. (ii) We create a document collection of* n *documents, where each document contains the phrase* $\langle\, a_i b_i \,\rangle$ *exactly* w_i *times. This ensures that* $\mathtt{df}(\langle\, a_i b_i \,\rangle, \mathcal{D}) = w_i$*.*

We additionally populate the rest of the document with single words a_i *and* b_i *such that* $\mathtt{df}(\langle\, a_i \,\rangle, \mathcal{D}) + \mathtt{df}(\langle\, b_i \,\rangle, \mathcal{D}) = p_i + w_i$*. This ensures that the benefit of materializing a lexicon* $\mathcal{L} = \{\,\langle\, a_i b_i \,\rangle\,\}$ *is* p_i*. We want to maximize the benefit of materializing a lexicon* $\mathcal{L}$ *subject to the constraint that the sum of costs is not greater than a given space budget, which maps to the same objective function as the binary knapsack problem.* □

Our objective function is *non-decreasing*, so that $\mathcal{L} \subseteq \mathcal{L}' \Rightarrow b(\mathcal{L}) \leq b(\mathcal{L}')$ – the overall benefit can only improve when a phrase is added to the lexicon. However, somewhat surprisingly, it is *not submodular* and thus does not have the property of diminishing returns, which has unfortunate ramifications detailed below.

To show this, we present a counter example. Let the marginal benefit of adding the phrase $\mathbf{s}$ to lexicon $\mathcal{L}$ be defined as

$$\begin{aligned} m(\mathbf{s}, \mathcal{W}, \mathcal{L}) &= b(\mathcal{W}, \mathcal{L} \cup \{\,\mathbf{s}\,\}) - b(\mathcal{W}, \mathcal{L}) \\ &= c(\mathcal{W}, \mathcal{L}) - c(\mathcal{W}, \mathcal{L} \cup \{\,\mathbf{s}\,\}) \;. \end{aligned}$$

Now consider $\mathbf{q} = \langle\, abcd \,\rangle$ as the only phrase query in $\mathcal{W}$, two dictionaries $\mathcal{L}_1$ and $\mathcal{L}_2$

$$\mathcal{L}_1 = \{\,\langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, c \,\rangle, \langle\, d \,\rangle, \langle\, bc \,\rangle\,\},$$
$$\mathcal{L}_2 = \{\,\langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, c \,\rangle, \langle\, d \,\rangle, \langle\, ab \,\rangle, \langle\, bc \,\rangle\,\}.$$

Further, assume that $\mathtt{df}(\mathbf{t}, \mathcal{D}) = 2$ for single words $\mathbf{t} \in \{\langle\, a \,\rangle, \langle\, b \,\rangle, \langle\, c \,\rangle, \langle\, d \,\rangle\}$ and $\mathtt{df}(\mathbf{t}, \mathcal{D}) = 1$ for phrases $\mathbf{t} \in \{\langle\, ab \,\rangle, \langle\, bc \,\rangle, \langle\, cd \,\rangle\}$. When using OPT from Section 5.4, we obtain $c(\mathbf{q}, \mathcal{L}_1) = 5$ and $c(\mathbf{q}, \mathcal{L}_2) = 4$. When we add $\langle\, cd \,\rangle$ to $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively, we obtain $c(\mathbf{q}, \mathcal{L}_1 \cup \{\langle\, cd \,\rangle\}) = 4$ (from the query plan $\{\langle\, a \,\rangle, \langle\, bc \,\rangle, \langle\, cd \,\rangle\}$) and $c(\mathbf{q}, \mathcal{L}_2 \cup \{\langle\, cd \,\rangle\}) = 2$ (from the query plan $\{\langle\, ab \,\rangle, \langle\, cd \,\rangle\}$) – expressed in terms of marginal benefits $m(\langle\, cd \,\rangle, \mathcal{W}, \mathcal{L}_1) = 1$ and $m(\langle\, cd \,\rangle, \mathcal{W}, \mathcal{L}_2) = 2$. Although $\mathcal{L}_1 \subset \mathcal{L}_2$, the marginal benefit of adding the phrase $\langle\, cd \,\rangle$ to $\mathcal{L}_2$ is larger.

Since our objective function is not submodular, we can not leverage the result from Nemhauser et al. [NWF78], which shows that the greedy algorithm that selects items in descending order of their benefit-cost ratio has a $(1 - \frac{1}{e})$-approximation guarantee.

Nevertheless, even without an approximation guarantee, we rely on this greedy algorithm to compile the lexicon $\mathcal{L}$. The algorithm considers as candidates all phrases that are contained in both the workload and the document collection – no other phrase can impact our objective function. It iteratively extends the lexicon $\mathcal{L}$ until the space budget is exhausted. In every iteration, the algorithm selects the phrase that has the largest benefit-cost ratio $m(\mathbf{s}, \mathcal{W}, \mathcal{L})/\mathtt{df}(\mathbf{s}, \mathcal{D})$. After adding a phrase to the lexicon $\mathcal{L}$, the benefit-cost ratios of the remaining candidates have to be updated. To this end, for each of the candidates, the algorithm performs a what-if analysis, computing query plans for each phrase query from the workload under the assumption that the candidate has been added to the lexicon. This is clearly prohibitive when implemented naïvely. It can be made feasible in practice by keeping track of which candidates can potentially be used to process which phrase query. One can then selectively update the benefit-cost ratios of only those candidates that can potentially be used to process any of the phrase query whose query plan has changed after the latest addition to the lexicon.

5.5.2. Coverage-Based Phrase Selection

While QOBS presented above is aware of the query-optimization method used and actively invokes it, our second phrase-selection method, coined coverage-based phrase-selection (CBS), is agnostic to phrase-query optimization. It is based on a simpler problem formulation that considers how many distinct positions in phrase queries from the workload can be covered using phrases from the lexicon and also keeps index size within a user-specified space budget.

We first formally define the notion of coverage of a position. A word sequence $\mathbf{s} = \langle\, w_1 w_2 \dots w_n \,\rangle$ is a sequence of n words where the index n indicates the nth position or nth word of the sequence. A position k is said to be covered by a phrase p, $|p| > 1$, if the following holds $\exists i \in \mathtt{pos}(\mathbf{s}, p) | i \leq k \leq i + |p|$.

Using this notion of a covered position we extend it to capture the concept of positions covered by a word sequence p in $\mathbf{s}$ denoted by $\mathsf{poscov}(\mathsf{p}, \mathbf{s})$.

$$\mathsf{poscov}(\mathsf{p}, \mathbf{s}) \;=\; \{\mathsf{k} \in \mathbf{Z}^{+} \;\mid\; \mathsf{i} \in \mathsf{pos}(\mathbf{s}, \mathsf{p}),\;\; \mathsf{i} \leq \mathsf{k} \leq \mathsf{i} + |\mathsf{p}|\;\}.$$

As an example positions covered by $\langle\, ab \,\rangle$ in phrase $\langle\, abxyab \,\rangle$ is

$$\mathsf{poscov}(\langle\, abxyab \,\rangle, \mathcal{V} \;\cup\; \{\langle\, ab \,\rangle\}) = \{1, 2, 4, 5\}.$$

To measure how much of a phrase query $\mathbf{q}$ can be covered using phrases from the lexicon $\mathcal{L}$ we introduce the measure $coverage(\mathbf{q}, \mathcal{L})$. The value of $coverage(\mathbf{q}, \mathcal{L})$ value ranges in $[0, |\,\mathbf{q}\,|]$ and conveys how many distinct positions from the phrase query can be covered. Formally,

Definition 5.6 (Coverage)

$$coverage(\mathbf{q}, \mathcal{L}) \;=\; \left| \left\{ \bigcup_{\mathsf{p} \in \mathcal{L}} \mathsf{poscov}(\mathsf{p}, \mathbf{q}) \mid \forall \mathsf{pos}(\mathsf{p}, \mathbf{q}) > 0 \right\} \right| .$$

In other words, coverage encodes the distinct positions covered by word sequences (not single words) from the lexicon in a query. Some examples of coverage are given below.

$$\begin{aligned} coverage(\langle\, abxy \,\rangle, \mathcal{V} \cup \{\langle\, abc \,\rangle, \langle\, xy \,\rangle\}) &= 2 \\ coverage(\langle\, abaxaba \,\rangle, \mathcal{V} \cup \{\langle\, aba \,\rangle, \langle\, ab \,\rangle\}) &= 6 \\ coverage(\langle\, ababac \,\rangle, \mathcal{V} \cup \{\langle\, aba \,\rangle, \langle\, xy \,\rangle\}) &= 5\,. \end{aligned}$$

We extend our definition of coverage to the workload as

$$coverage(\mathcal{W}, \mathcal{L}) = \sum_{\mathbf{q} \in \mathcal{W}} coverage(\mathbf{q}, \mathcal{L})\;.$$

Again, $\mathcal{W}$ is a bag of word sequences, so that the coverage of repeated phrase queries is reflected.

Our objective is to compile a lexicon $\mathcal{L}$ that maximizes the $coverage(\mathcal{W}, \mathcal{L})$ and results in an index whose size is within a user-specified space budget. To measure index size and model the user-specified space budget, we use the same formalism as in QOBS. We obtain the optimization problem

Definition 5.7 COVERAGE-BASED PHRASE SELECTION

$$\underset{\mathcal{V} \subseteq \mathcal{L} \subseteq \mathcal{V}^{+}}{argmax}\; coverage(\mathcal{W}, \mathcal{L}) \quad s.t. \quad s(\mathcal{L}) \leq (1 + \alpha) \cdot s(\mathcal{V})\;.$$

This problem formulation considers the same ingredients as the problem formulation behind QOBS. It implicitly takes into account the frequency $\mathtt{df}(\mathbf{s}, \mathcal{W})$ of a phrase **s** in the workload, using the coverage measure, and also $\mathtt{df}(\mathbf{s}, \mathcal{D})$ reflecting how expensive the phrase is to select. Unlike QOBS, this formulation does not take into account the costs of the terms which are replaced by a phrase. Consider a case where two phrases $\langle\, xy \,\rangle$ and $\langle\, ab \,\rangle$ have $\mathtt{df}(\langle\, xy \,\rangle, \mathcal{W}) = \mathtt{df}(\langle\, ab \,\rangle, \mathcal{W})$ but

$$\mathtt{df}(\langle\, a \,\rangle, \mathcal{D}) + \mathtt{df}(\langle\, b \,\rangle, \mathcal{D}) > \mathtt{df}(\langle\, x \,\rangle, \mathcal{D}) + \mathtt{df}(\langle\, y \,\rangle, \mathcal{D}).$$

CBS is agnostic of the fact that $\langle\, ab \,\rangle$ is more beneficial to select. Every instance of coverage-based selection can be mapped to an instance of the *budgeted-maximum coverage* (BMC) problem [KMN99]. BMC takes as input a collection of sets $\mathcal{S} = \{\, s_1, \ldots, s_m \,\}$ with associated costs $\{\, c_1, \ldots, c_m \,\}$ defined over a domain of items $\{\, x_1, \ldots, x_n \,\}$ that have associated weights $\{\, w_1, \ldots, w_n \,\}$. The goal is to find a collection of sets $\mathcal{S}' \subseteq \mathcal{S}$ that maximizes the total weight of items covered and whose total cost does not exceed a given budget L. The transformation is straightforward: (i) candidate phrases are sets $\mathbf{s_i}$ with costs $\mathtt{df}(\mathbf{s}_i, \mathcal{D})$, (ii) items are distinct positions in phrase queries from the workload each with unit weight, (iii) the budget is set as $L = (1 + \alpha) \cdot s(\mathcal{V})$.

The greedy algorithm that selects items in descending order of their benefit-cost ratio gives a $(1 - \frac{1}{e})$-approximation guarantee for BMC. The cost of a phrase **s** is $\mathtt{df}(\mathbf{s}, \mathcal{D})$; its benefit is defined as the number of yet-uncovered distinct positions that it covers

$$coverage(\mathcal{W}, \mathcal{L} \cup \{\, \mathbf{s} \,\}) - coverage(\mathcal{W}, \mathcal{L}) \ .$$

Since the objective function captures coverage of items in a set, it is submodular which implies that the benefits of candidates are non-increasing. This offers opportunities for optimization in practice which we discuss in the next section.

5.5.3. Optimizations for Practical Indexing

Both the phrase selection methods outlined before use a greedy algorithm to select sequences which are to be indexed. The greedy algorithm, in every iteration, chooses the most promising candidate and adds it to the lexicon. Candidates are maintained in a priority queue and the best candidate is chosen based on the benefit-cost ratio. After a choice has been made, the benefit values of all the remaining candidate sequences are updated. The bottleneck of the greedy algorithm is usually the update of candidate benefits after a phrase has been added to the lexicon. We discuss optimizations for both selection algorithms to make them feasible in practice.

Maintaining Dependent Queries

For QOBS, the benefit of each candidate $\mathbf{s}$, given the current state of the lexicon $\mathcal{L}$, is given by $c(\mathcal{W}, \mathcal{L}) - c(\mathcal{W}, \mathcal{L} \cup \{\mathbf{s}\})$. Implementing this in a naïve manner involves re-computing query plans for all queries in the workload with any addition of $\mathbf{s}$ to the lexicon ($\mathcal{L} \cup \{\mathbf{s}\}$). This is clearly an expensive operation. However, a candidate is not present in all queries. Only a subset of queries $\mathcal{W}' \subseteq \mathcal{W}$ where $\mathbf{s}$ is present in, $\{\mathbf{q} \in \mathcal{W}' \mid \mathrm{pos}(\mathbf{s}, \mathbf{q}) \neq \phi\}$, are affected. Based on this we pre-compute and store such dependencies between candidates and queries. This has a couple of benefits. Firstly, as discussed above, they avoid unnecessary recomputation of query plans. Secondly, once the query plans for $\mathcal{W}'$ have to be recomputed, we use the dependencies to update the benefits of only those candidates affected by $\mathcal{W}'$, i.e,

$$\left\{ \mathbf{s} \in \mathcal{L} \setminus \mathcal{V}^{+} \mid \mathrm{pos}(\mathbf{s}, \mathbf{q}) \neq \phi, \ \ \mathbf{q} \in \mathcal{W}' \right\}.$$

This optimization is also applicable for CBS where instead of computing query plans we compute coverage $coverage(\mathcal{W}, \mathcal{L})$ at every update step. As an additional optimization one can dynamically prune out queries which have been completely covered.

Lazy Updates

We discussed in the previous section that CBS admits sub-modularity due to which the benefits of candidates are non-increasing over the increasing execution states of the algorithm. We say that a pair of candidates $\mathbf{s}$ and $\mathbf{s}'$ are *independent* if they do not have a query in common, or,

$$\left\{ q \mid \mathrm{pos}(\mathbf{s}, q) \neq \emptyset \wedge \mathrm{pos}(\mathbf{s}', q) \neq \emptyset \right\} = \emptyset.$$

Hence the choice of $\mathbf{s}$ does not affect the benefit value of $\mathbf{s}'$ for the subsequent iteration. Utilizing this observation we can defer updating the benefit values of candidates until it is deemed required. Specifically, we can avoid the update step if the next best candidate is independent to all the previously selected candidates $C \subseteq \mathcal{L}$ from the last update step. Once this condition is violated all the dependent queries of candidates in C are updated. This allows us to lazily update them, resulting in runtime improvements by almost 50%.

Candidate Pruning

Finally for CBS, when the query plans are indeed updated, the partial contributions for all candidate phrases associated with each query have to be determined. This is implemented by maintaining associatively the query to candidate phrase mapping $M(q)$.

Whenever a query q is updated with the new lexicon $\mathcal{L}$, the modified benefit of each candidate **s** dependent on q needs to be computed. However, if sequences from $\mathcal{L}$ already cover the regions covered by **s**, choosing **s** would not provide no further benefit to q. This allows to prune away candidate phrases from $M(q)$ when

$$\mathrm{poscov}(\mathbf{s}, q) \subseteq \{\mathrm{poscov}(\mathbf{s}', q) | \mathbf{s}' \in \mathcal{L}\}.$$

This allows to update fewer number of candidates in the subsequent stages of the algorithm potentially leading to further improvements in efficiency.

5.6. System Architecture

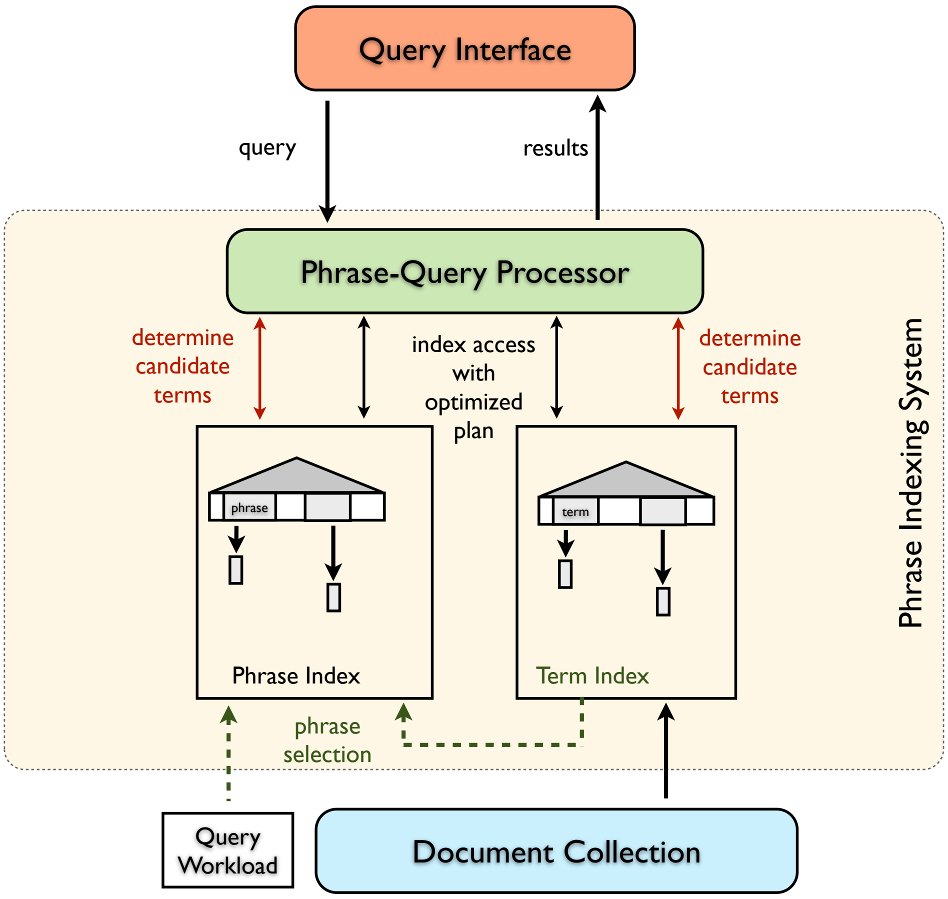


Figure 5.1.: System architecture

Figure 5.1 shows a high-level overview of the architecture of our phrase-indexing system. It consists of two indexes – the *term index* and the *multi-word index*.

- **Term Index** It is the standard word-level inverted index consisting of the lexicon and the inverted files built over the document collection. It indexes all single words with their positional information.

- **Phrase Index** It is constructed over the document collection for a set of phrases selected employing the phrase-selection algorithms detailed in Section 5.5.

In practice, we build the term index first and its associated lexicon. The statistics required for the phrase-selection algorithm are computed from the lexicon and the query workload. The selection algorithms are then executed and a *selection set* of word-sequences are determined which have to be indexed in the phrase index. Typically, the selected set is small in size and fits in memory. Hence, indexing infrastructure for indexing words can be reused employing the selection set of word sequences to filter out sequences which do not need to be indexed.

While processing queries, both the lexicons, for term and the phrase index, are consulted to determine the candidate words or word sequences presented in the query. Query optimization is performed over the candidate terms to determine the best plan. Finally, the respective indexes are accessed, for the terms in the optimized plan, for fetching and intersecting the posting lists to compute results.

5.7. Experimental Evaluation

In this section, we describe our experimental evaluation. We begin with details about our experimental setup including employed datasets, before describing our comparison of the query-optimization methods from Section 5.4, followed by an evaluation of our phrase-selection methods from Section 5.5 against state-of-the-art competitors.

5.7.1. Setup

All indexes were built on a local Hadoop cluster consisting of ten Dell R410 server-class computers, each equipped with 64 GB of main memory, two Intel Xeon X5650 6-core CPUs, and four internal 2 TB SAS 7,200 rpm hard disks configured as a bunch-of-disks. The machines are connected by 10 Gbit Ethernet, run Cloudera CDH3u0 as a distribution of Hadoop 0.20.2, and use Oracle Java 1.6.0_26. Query optimization and phrase-selection experiments were performed on a Dell PowerEdge M610 server with 2 Intel Xeon E5530 CPUs, 48 GB of main memory, a large iSCSI-attached disk array, Debian GNU/Linux (SMP Kernel 2.6.29.3.1) and running Oracle Java 1.6.0_34. Wall-clock time measurements were performed with the Java Hotspot 64-Bit Server VM using the CMS garbage collector.

5.7.2. Datasets Used

We use two real-world document collections for our experiments:

- *ClueWeb09-B* [CWC13] (CW) – ClueWeb09-B is a subset of the ClueWeb09 corpus consisting of more than 50 million web documents in English language crawled in 2009;
- *The New York Times Annotated Corpus* [NYT13] (NYT) – The New York Times Annotated Corpus, as introduced in the previous chapter, contains more than 1.8 million newspaper articles published by The New York Times between 1987 and 2007.

Both document collections were processed using Stanford CoreNLP [COR13] for tokenization. To make CW more handleable, we use boilerplate detection as described in [KFN10] and available in the `DefaultExtractor` of boilerpipe [BOI] .

5.7.3. Query Workload

As a workload we use entity labels from the YAGO2 knowledge base [HSBW13]. In its `rdfs:label` (formerly `means`) relation, YAGO2 collects strings that may refer to a specific entity, which are mined from anchor texts in Wikipedia. For the entity `Bob_Dylan`, as a concrete example, it includes among others the entity labels "bob dylan", "bob allen zimmerman", and "robert allen zimmerman". In total, the workload that we obtain contains 13.4 million entity labels having an average length of 2.41 words. Interestingly, almost 99% of them do not contain any repeated word; we observe at most eight repeated words for the phrase queries in our workload. We only consider those entity labels for which all constituent words occur in the document collection at hand, leaving us with 10.7 million and 8.0 million phrase queries for CW and NYT, respectively.

For our experiments on the query-optimizers effectiveness, we additionally consider a subset of our workload which refer to artist names, albums and song titles. Some examples of phrases in this workload are "american national anthem", "and the green grass grew all around" etc. The workload that we obtain has $107,245$ entity labels having an average length of 3.4 words.

5.7.4. Index Management and Competitors

We implemented our indexing framework using Hadoop. The lexicon, containing for each term its term identifier, document frequency, and collection frequency, is stored in a flat file and loaded into main memory at runtime. Posting lists are kept in an indexed file (implemented using Hadoop's `MapFile`) and are stored using variable-byte encoding in combination with d-gaps for document identifiers of consecutive postings and offsets within each posting. We use TAAT to process phrase queries.

We compare against the following methods in our experiments – some from the literature and others conceivable baselines:

- **Uni-Gram Index** (UNI) indexes unigrams and does not select any phrases;
- **Oracle Index** (ORA) indexes unigrams and selects all phrase queries from the workload as phrases;
- **Next-Word Index** [WZB04] (NEXT) indexes unigrams and selects all bigrams from the workload that contain a stopword as phrases;
- **Combined Index** [WZB04] (COMB) combines NEXT and ORA. In the original paper, the authors considered, instead of ORA, a phrase index that contains pre-computed results of popular phrase queries. ORA thus selects a superset of the phrases that the original approach considered. We further strengthen this competitor, in comparison to its original description, by using phrases from ORA also to process other longer phrase queries. Thus, if "united states" has been selected by ORA, our query optimizer may use it to processing the phrase query "president of the united states".
- **Bi-Gram Index** (BI) indexes unigrams and selects all bigrams from the workload as phrases;
- **Tri-Gram Index** (TRI) indexes unigrams and selects all bigrams and trigrams from the workload as phrases;
- **Out-of-Box Index** [TS09] (OOBI) indexes unigrams and selects all bigrams whose cost is above a user-specified threshold. Unlike the other competitors, OOBI is thus also *tunable*. To make it comparable to our phrase-selection methods, we adapt it, so that it ranks bigrams in descending order of their document frequency and selects phrases from the obtained list until the user-specified space budget has been exhausted.

We compare these approaches against our query-optimizer-based selection (QOBS) and coverage-based selection (CBS). We use document frequency as a cost measure for all our experiments. As mentioned earlier, one could use collection frequency instead. In practice, though, the two measures are highly correlated and we did not observe big differences. Also, as a one-time pre-processing performed using Hadoop and made available to all methods, we computed document frequencies in the workload and the document collection for all n-grams from the entire workload.

		GRD			APX		
		$l = 2$	$l = 4$	$l = 6$	$l = 2$	$l = 4$	$l = 6$
	%						
NYT	[0]	7,586,656	7,839,328	7,877,367	7,662,566	7,868,885	7,900,573
	$(0 - 20)$	403,399	200,017	174,744	382,428	192,417	166,658
	$[20 - 40)$	70,852	31,399	22,281	35,157	18, 750	12,712
	$[40 - 60)$	19,256	9,368	5,706	12	111	220
	$[60 - 80)$	-	51	65	-	-	-
	%						
CW	[0]	10,080,400	10,488,737	10,589,685	10,176,979	10,523,862	10,607,792
	$(0 - 20)$	547,289	204,716	135,579	525,781	27,894	132,417
	$[20 - 40)$	98,391	45,313	21,526	49,995	27,894	12,496
	$[40 - 60)$	26,700	13,740	5,764	25	136	75
	$[60 - 80)$	-	274	226	-	-	-

Table 5.2.: Percentage improvement in query-processing cost by OPT over GRD and APX

		GRD			APX		
		$l = 2$	$l = 4$	$l = 6$	$l = 2$	$l = 4$	$l = 6$
	%						
NYT	[0]	77,130	82,058	83,708	79,459	82,703	84,031
	$(0 - 20)$	7,705	3,686	2,747	6,656	3,654	2,667
	$[20 - 40)$	1,735	950	379	839	594	254
	$[40 - 60)$	387	261	117	3	6	5
	$[60 - 80)$	-	2	6	-	-	-
	%						
CW	[0]	84,108	91,039	94,839	86,281	92,034	95,218
	$(0 - 20)$	9,888	4,129	1,849	9,772	4,259	1,832
	$[20 - 40)$	2,710	1,753	505	1,342	1,105	347
	$[40 - 60)$	704	480	197	15	12	13
	$[60 - 80)$	-	9	20	-	-	-

Table 5.3.: Percentage improvement in query-processing cost by OPT over GRD and APX – on song titles

5.7.5. Performance of Query Optimization

Our first experiment examines the effect that the choice of query-optimization method can have on query-processing performance. We consider three query-optimization methods for this experiment: the greedy algorithm (GRD) from [WZB04], our greedy algorithm (APX) that gives an approximation guarantee, and our exponential exact algorithm (OPT). GRD considers terms in increasing order of their document frequency, thus based on their selectivity, and chooses a term if it covers any yet-uncovered portion of the phrase query. Originally designed to deal with bigrams only, we extend GRD to break ties based on term length, and thus favor the longer term, if two terms have the same document frequency.

To compare the three query-optimization methods, we built augmented inverted indexes whose dictionaries include all phrases up to a specific maximum length $l \in \{2, 4, 6\}$. Thus, for $l = 4$, all phrases of length four or less are indexed. This allows us to study the behavior of the methods as more terms to choose from become available.

First, we examine the different methods in terms of their runtime in practice. We observe an average runtime of 0.01 ms for each of them, showing that there is no difference in practice. The maximum runtime observed for OPT for any of the phrase queries from our workload is 2.00 ms, indicating that its exponential nature rarely affects its runtime in practice. Thus, for all further experiments we use OPT as a query-optimization method.

Second, we compare the different methods in term of the costs of their generated query plans. To this end, we determine for each phrase query from the workload the percentage improvement over GRD and APX, respectively that one can achieve by using OPT. Let C_{OPT}, C_{GRD}, and C_{APX} denote the cost of the query plan (in terms of total number of postings read) determined by the respective query-optimization method. The percentage improvement is given by the values $(C_{\mathrm{GRD}} - C_{\mathrm{OPT}})/(C_{\mathrm{GRD}})$ and $(C_{\mathrm{APX}} - C_{\mathrm{OPT}})/(C_{\mathrm{APX}})$ that range in $[0, 1)$. Table 5.2 gives bucketed percentage improvements for our two datasets and three augmented inverted indexes. Each cell reports the number of phrase queries from the workload for which a percentage improvement in the given range was observed.

From Table 5.2, we observe that, for the majority of phrase queries, there is no substantial improvement (i.e., less than 20%) when using OPT instead one of the non-optimal methods. Further, we see that the number of phrase queries for which an improvement is achieved is generally lower for APX than for GRD, which is expected given the former's approximation guarantee. As can also be seen from the table, there is a non-negligible number of phrase queries for which OPT improves by 40% or more over GRD. For the query "we are the champions" with $l = 2$, as a concrete example, GRD picks {we are, are the, the champions} as a query plan, which is more than twice as expensive as the query plan {we are, the champions} determined by OPT. When comparing percentage improvements across different augmented inverted indexes, we see less improvement for larger values of l, which makes sense since those include longer, more selective phrases favored by the non-optimal methods.

To examine the effect of the query optimizer of longer query lengths, we now look at the results on experiments on song titles summarized in Table 5.3. We observe that, like in the previous table, for majority of the queries, OPT shows no improvement over GRD and APX. However, OPT shows non-zero improvement for around 11% of the queries in Table 5.3, as compared to the entire workload where the improvements are for less than 6% of the queries. This means that OPT improves over the other optimizers for longer query lengths. Consistent with the previous observation, we see that APX performs better than GRD in majority of the scenarios. It is also interesting to note that, although the improvements for larger values of l is lesser, the magnitude of improvement is larger as compared to $l = 2, 4$. As an example, for $l = 6$ in CW, we see an 60%-80% improvements over GRD when using OPT in some queries.

In summary, neither GRD nor APX falls far behind OPT in terms of the cost of its generated query plans. APX is robust, comes with an approximation guarantee, and is easy to implement. However, as stated above, we did not see any phrase query for which OPT was too expensive to run, making it a viable choice in practice.

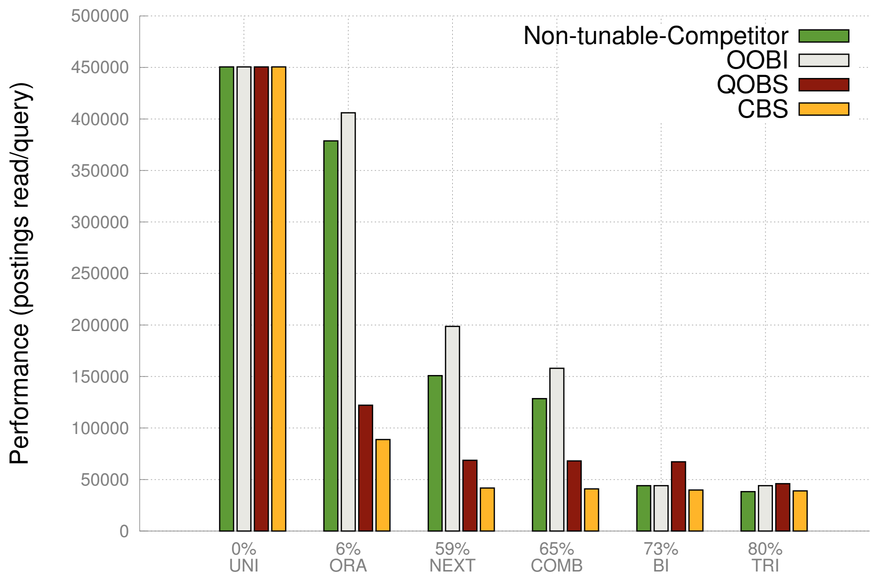
5.7.6. Effect of Phrase Selection

Our second experiment compares our phrase-selection methods against their tunable and non-tunable competitors in terms of query-processing performance.

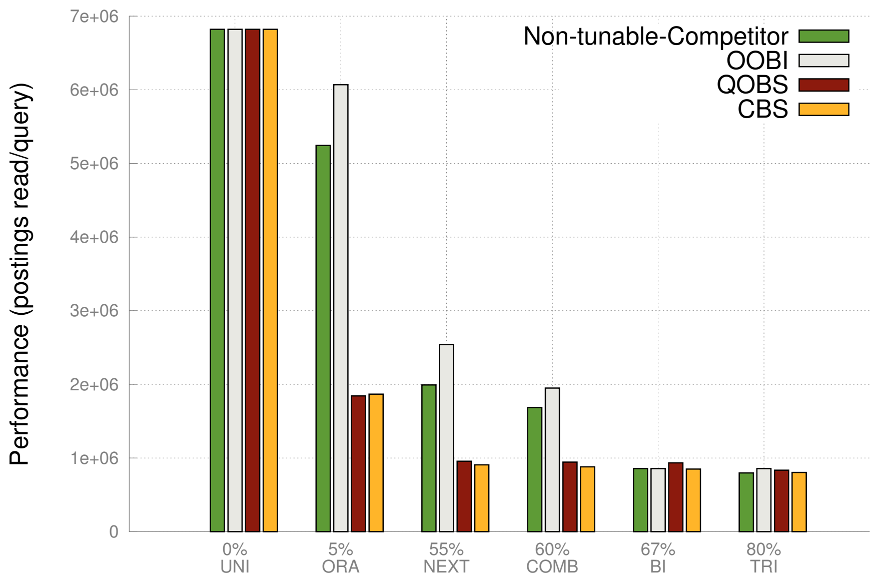
We use five-fold cross validation throughout this experiment. Our workload is split into five folds, yielding five training-test configurations. Phrase selection is then performed using the four training folds; query-performance measurements are performed using the test fold. We report averages over the five training-test configurations.

For all methods, we assume that single words are present in the lexicon – UNI thus serves as a baseline that all methods under comparison build upon. On CW the standard positional inverted index obtained by UNI amounts to 71 GB and contains a total of 90.17 billion postings; on NYT the corresponding index amounts to 3 GB and contains a total of 4.91 billion postings. Index sizes, in the following, are indicated in terms of their percentage overhead over UNI – an index size of 10% thus means that the corresponding is $1.1\times$ larger than the baseline index.

Query-processing performance is measured both in abstract and concrete terms. As an abstract cost measure, we use the average number of postings that is read to process a



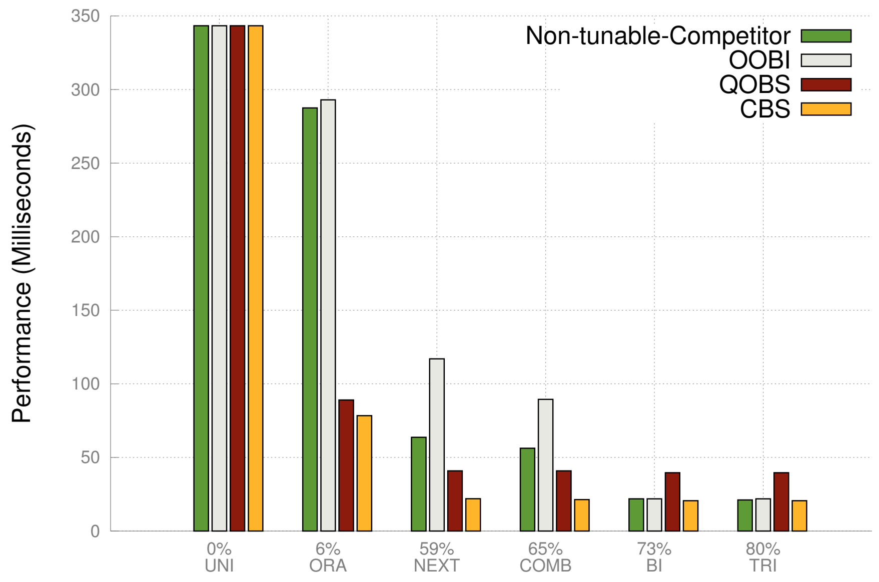
(a) NYT



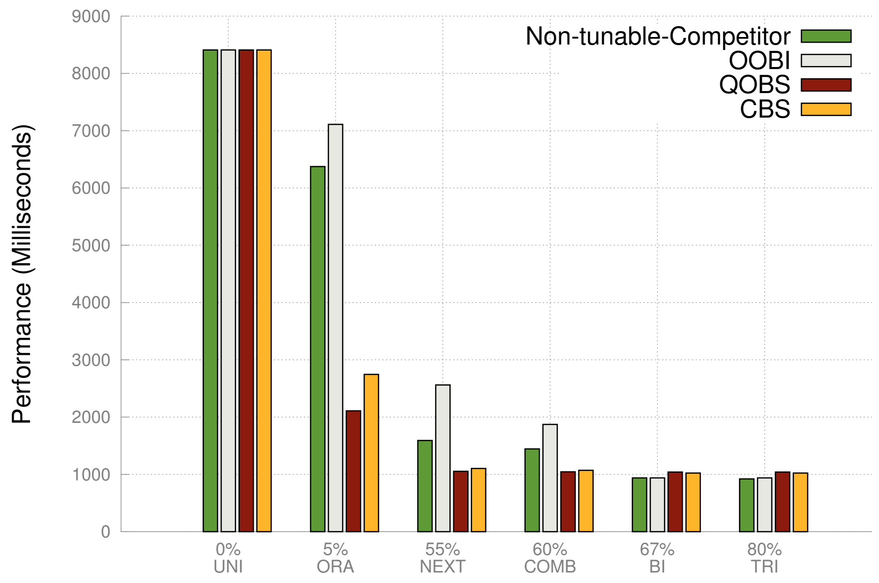
(b) CW

Figure 5.2.: Performance of tunable phrase-selection methods relative to non-tunable competitors in terms of abstract cost measures

phrase query from the test folds. We use wall-clock times (in milliseconds) as a concrete cost measure. These were obtained based on a sample of 25,000 phrase queries (5,000 per test fold), using a single core, and pre-fetching all required posting lists into main



(a) NYT

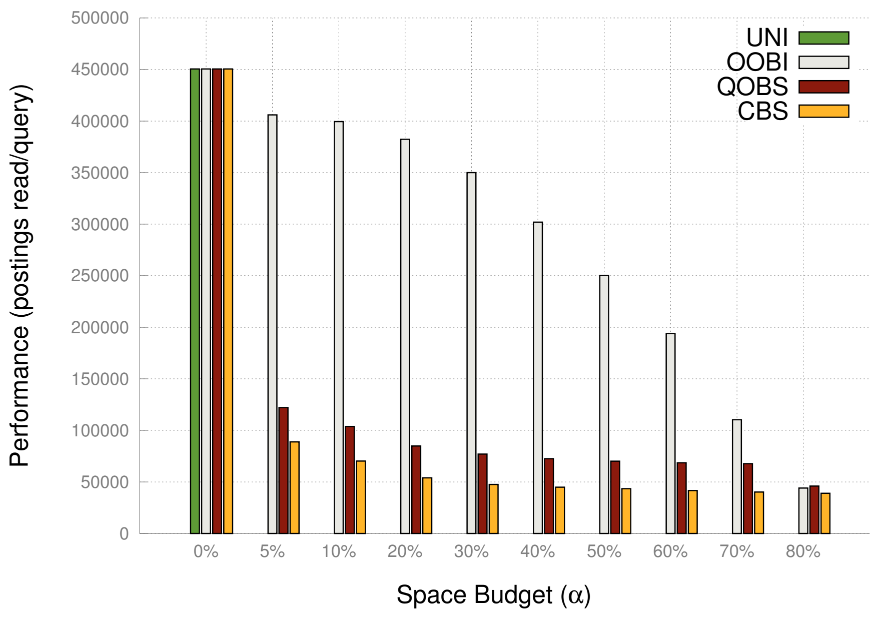


(b) CW

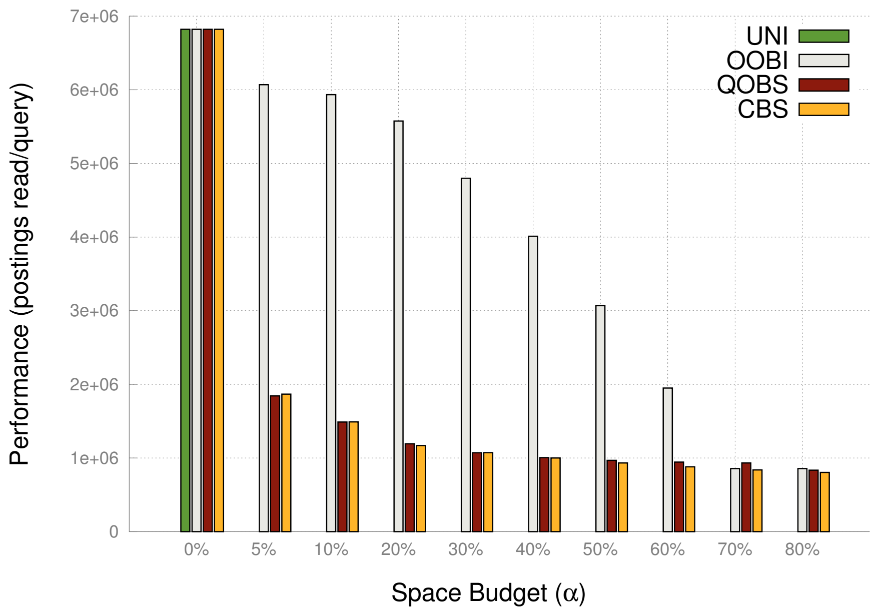
Figure 5.3.: Performance of tunable phrase-selection methods relative to non-tunable competitors in terms of wall-clock times

memory.

Figure 5.3 compares the tunable phrase-selection methods (QOBS, CBS, OOBI) against their non-tunable competitors. As a first step, we built an index for each of the non-



(a) NYT



(b) CW

Figure 5.4.: Performance of tunable phrase-selection methods

tunable competitors. The sizes of these indexes determine the values of α, which we feed into the tunable phrase-selection methods to obtain indexes of correspond sizes, in a second step. For the sake of comparison, we include UNI in Figure 5.3, corresponding to $\alpha = 0$, that is, no phrases are selected.

We observe that ORA results in the smallest index, since it materializes only phrases that occur in exactly that form as phrase queries in the workload. However, doing so, it overfits to the training folds, resulting in query-processing performance that is only slightly better than our baseline UNI. When given the same amount of additional space (6% for NYT and 5% for CW), our tunable phrase-selection methods QOBS and CBS achieve considerably better query-processing performance – an improvement by at least a factor 3× over ORA and the baseline UNI in terms of both abstract and concrete measures. NEXT and COMB, materializing all bigrams from the training folds that contain a stopword, result in indexes that consume $55\% - 60\%$ additional space. While they achieve good improvements in query-processing performance over the baseline, they are consistently outperformed by QOBS and CBS. BI and TRI, which result in indexes that require $67\% - 80\%$ of additional space, improve query-processing performance by at least a factor 7× over the baseline. From the tunable phrase-selection methods OOBI and CBS perform at par, when given this much additional space. While this also holds for QOBS on CW, it performs slightly worse than its competition on NYT. Moreover, we see that the trends observed in abstract and concrete measures of query-processing performance are consistent. In the rest of this section, we thus only consider abstract cost measures.

Figure 5.4 compares our tunable phrase-selection methods QOBS and CBS against OOBI as the only tunable competitor. For all three methods, we constructed indexes considering values of α ranging from 5% to 80%. UNI is again included at $\alpha = 0\%$ for the sake of comparison. With as little as 5% additional space, QOBS and CBS improve query-processing performance by at least a factor 3×. When comparing our two methods, we observe that CBS performs slightly better than QOBS on NYT, whereas their performance is comparable on CW. QOBS and CBS perform consistently better than their sole competitor OOBI. When given 5% additional space, as a concrete figure, OOBI improves over UNI by at most a factor 1.2×. What is also apparent from the figure are the diminishing returns of additional space, which are more pronounced for our methods that already make highly effective use of the initial 5% of additional space. Finally, when given ample additional space, all tunable phrase-selection methods perform at par.

5.8. Related Work

We now discuss the connection between our work and existing prior work, which we categorize as follows:

Phrase Queries. Williams et al. [WZB04] put forward the *combined index* to support phrase queries efficiently. It assembles three levels of indexing: (i) a *first-word index* as a positional inverted index, (ii) a *next-word index* that indexes all bigrams containing a stopword, and (iii) a *phrase index* with popular phrases from a query log. Its in-memory lexicon is kept compact by exploiting common first words between bigrams. Query processing escalates through these indexes – first it consults the phrase index and, if the phrase query is not found therein, processes it using bigrams and unigrams from the other indexes. Transier and Sanders [TS09] select bigrams to index based only on characteristics of the document collection. Selecting bigrams makes sense in settings where phrase queries are issued by human users and tend to be short – as observed for web search by Spink et al. [SWJS01]. We also target application-generated queries (e.g., quotations and titles of movies or songs) and thus select variable-length phrases. Those have previously been considered by Chang and Poon [CP08] in their *common phrase index*, which builds on [WZB04], but indexes variable-length phrases common in the workload. Our methods, in contrast, consider both the document collection and the workload.

Proximity scoring, such as the model by Büttcher et al. [BCL06b], is similar in spirit to phrase queries but targets ranked retrieval. Proximity of query words is an important signal in modern web search engines. Several authors have looked into making the computation of proximity scores more efficient. Yan et al. [YSZ$^+$10] propose a word-pair index and develop query-processing methods that support early termination. Broschart and Schenkel [BS12] describe a tunable word-pair index that relies on index pruning to keep its size manageable. Fontoura et al. [FGJV11] describe an alternative method of indexing word pairs, which maintains them as bitmaps along with posting lists for single words.

Caching is often used to speed up query processing and reduce the overall system load. It can be applied at different granularities including query results, posting lists of single words, and posting-list intersections. The first two are considered by Saraiva et al. [SSdMZ$^+$01] as well as Baeza Yates et al. [BYGJ$^+$08]. Long and Suel [LS06] propose a three-level cache that also includes posting-list intersections. Going beyond that, Ozcan et al. [OAC$^+$12] describe a five-level cache that additionally includes result snippets and documents. Policies for admitting/evicting items to/from the cache have been described, among others, by Baeza-Yates et al. [BYJPW07] as well as Gan and Suel [GS09]. Fagni et al. [FPSO06] distinguish between a static and a dynamic cache, where the former is periodically bootstrapped from query logs, and the latter is managed using a replacement policy such as LRU. While none of the aforementioned works has specifically addressed phrase queries, it is conceivable to add a layer that caches phrases as intersections of multiple posting lists. Our phrase-selection methods could then be used to bootstrap a (static) cache.

5.9. Summary

In this chapter we developed efficient solution to processing phrase queries. We studied how arbitrary phrase queries can be efficiently processed over an augmented-inverted index of word sequences. We developed methods to select multi-word sequences to be indexed to optimize query-processing cost while keeping the index size within a user-specified budget. We also proposed novel query-optimization techniques to efficiently process phrase queries over such an augmented index.

With regard to phrase-query optimization, a first insight from our experiments is that the non-optimal methods perform close to the optimum for a majority of phrase queries. As a second insight, we observed that our tunable phrase-selection methods make highly effective use of additional space, in particular when there is only little of it, and considerably improve the processing performance of phrase queries.

6 Conclusions

6.1. Summary of Results

Supporting workloads which combine text (keywords and phrases) and time are useful in many interesting search, mining, and exploration tasks over web archives. In this work, we have addressed three problems in indexing text to support such workloads in web archives.

We presented a novel index-organization scheme called *index sharding* to process *time-travel* text queries that partitions each posting list with almost zero increase in index size. Our approach is based on avoiding access to irrelevant postings by exploiting the geometry of the valid-time intervals associated with the document versions. We proposed an optimal algorithm to completely avoid access to irrelevant postings. We further fine tuned the index, taking into account the index-access costs, by allowing for a few wasted sequential accesses while gaining significantly by reducing the number of random accesses. Finally, we proposed an incremental index sharding approach that supports efficient index maintenance for dynamic updates to the index without compromising the query performance. We empirically established the effectiveness of our sharding scheme with experiments over the revision history of the English Wikipedia, and an archive of U.K. governmental web sites. Our results demonstrate the feasibility of faster time-travel query processing with no space overhead. Moreover, we showed that maintaining our index structure incrementally has large benefits over indexes which are recomputed periodically.

Next, we looked at the problem of query optimization in time-travel text search. We presented approaches for *efficient approximate* processing of time-travel queries over a vertically-partitioned inverted index. By using a small synopsis for each partition we identified partitions that maximize the result size, and schedule them for processing early on. Our approach aims to balance the estimated gains in the result recall against required index-access cost. Our experiments with three diverse, large-scale text archives – the English Wikipedia revision history, the New York Times collection and, the UK-GOV dataset – show that our proposed approach can provide close to 80% result recall even when only about half the index is allowed to be read.

Finally, we proposed indexing and query-optimization approaches to efficiently answer *phrase queries*. We considered an augmented inverted index that indexes selected variable-length multi-word sequences in addition to single words. We studied how arbitrary phrase queries can be processed efficiently on such an augmented inverted index. Moreover, we developed methods to select multi-word sequences to be indexed so as to optimize query-processing cost while keeping index size within a user-specified space budget, taking into account characteristics of both the workload and the document collection. We demonstrated experimentally the efficiency and effectiveness of our methods on two real-world corpora, i.e, the New York Times collection and the ClueWeb09 dataset.

6.2. Outlook on Future Directions

The problems addressed in this work are some of the many efficiency issues which arise in the context of text search and mining for web archives. Hence, there are many opportunities for future research.

Exploiting Redundancy for Better Retrieval Web archives are characterized by a lot of redundant content. Content is continuously added to such collections, but the addition of new content does not necessarily contribute novel content. Much of the content is either copied, enriched or recompiled from existing documents. Redundancy in text collections, apart from wasting storage space, degrades search results. Initial attempts have been made to remove redundancy in web archives by giving user the flexibility to define her notion of redundancy [PAB13]. However, there are challenges in identifying and removing redundancy from search results. Many versions of the same document are likely to match the query because (i) either they are near duplicates or (ii) they share the context of the query terms. Designing retrieval models and indexing methods to counter the effect of such redundancy in search results, specifically for web archives, is an interesting direction for future research.

Phrase Indexing for Batched Query Processing In our work, we indexed commonly occurring word sequences to improve query processing efficiency of a given phrase query. However, there are scenarios when multiple phrase queries are issued in a batch. Consider a retrieval task of finding all documents which contain mentions of the entity Barack_Obama. Assume that we already know the different textual representations or labels used to denote this entity, i.e., “president of U.S.A”, “president of the united states”, “leader of the united states” and so forth. Treating each label as a phrase, each entity is associated with a batch of phrases as above. The research challenge is to come up with novel query processing methods, based on our current work on phrase indexing, to efficiently process such batches of phrase queries.

Mining and Exploration of Web Archives Exploratory tasks over web archives require multiple rounds of searching and aggregation over both the text and time axes. A promising research direction would be to identify how users interact with such collections and what features are they interested in. As an example, consider a user interested in all entities present in the documents which are results of the time-travel query “google io” @ **[03/2013 - 06/2013]**. The challenges are in improving retrieval effectiveness and evaluation of such systems.

Bibliography

[ABBS10] Avishek Anand, Srikanta Bedathur, Klaus Berberich, and Ralf Schenkel. Efficient temporal keyword search over versioned text. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management*, pages 699–708, 2010.

[ABBS11] Avishek Anand, Srikanta Bedathur, Klaus Berberich, and Ralf Schenkel. 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*, pages 545–554, 2011.

[ABBS12] Avishek Anand, Srikanta Bedathur, Klaus Berberich, and Ralf Schenkel. Index maintenance for time-travel text search. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 235–244, 2012.

[ACCG08] Sanjay Agrawal, Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. Scalable ad-hoc entity extraction from text collections. *Proceedings of the International Conference on Very Large Data Bases*, pages 945–957, 2008.

[ADGK07] Benjamin Arai, Gautam Das, Dimitrios Gunopulos, and Nick Koudas. Anytime measures for top-k algorithms. In *Proceedings of the International Conference on Very Large Data Bases*, pages 914–925, 2007.

[AF92] Peter Anick and Rex Flynn. Versioning a full-text information retrieval system. In *Proceedings of the 15th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 98–111, 1992.

[AGBY07] Omar Alonso, Michael Gertz, and Ricardo Baeza-Yates. On the value of temporal information in information retrieval. *SIGIR Forum*, pages 35–41, 2007.

[AGP99] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Aqua: A Fast Decision Support Systems Using Approximate Query Answers. In *Proceedings of the International Conference on Very Large Data Bases*, pages 754–757, 1999.

[AGPR99] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 275–286, 1999.

[AMC07] Rodrigo Almeida, Barzan Mozafari, and Junghoo Cho. On the Evolution of Wikipedia. In *International Conference on Weblogs and Social Media*, 2007.

[ARK13] http://www.netarkivet.dk, (accessed on 28-06-2013).

[ATDE09] Eytan Adar, Jaime Teevan, Susan T. Dumais, and Jonathan L. Elsas. The web changes everything: understanding the dynamics of web content. In *Proceedings of the 2nd ACM International Conference on Web Search and Data Mining*, pages 282–291, 2009.

[BBNW07] Klaus Berberich, Srikanta Bedathur, Thomas Neumann, and Gerhard Weikum. A Time Machine for Text Search. In *Proceedings of the 30th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 519–526, 2007.

[BC08] Stefan Büttcher and Charles Clarke. Hybrid index maintenance for contiguous inverted lists. *Information Retrieval*, pages 175–207, 2008.

[BCC10] Stefan Büttcher, Charles Clarke, and Gordon Cormack. *Information Retrieval - Implementing and Evaluating Search Engines*. The MIT Press, 2010.

[BCH$^+$03] Andrei Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 13th ACM Conference on Information and Knowledge Management*, pages 426–434, 2003.

[BCL06a] Stefan Büttcher, Charles Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 356–363, 2006.

[BCL06b] Stefan Büttcher, Charles Clarke, and Brad Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 621–622, 2006.

[BGMZ97] Andrei Broder, Steven Glassman, Mark Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks*, pages 1157–1166, 1997.

[BGO$^+$96] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB JournalThe International Journal on Very Large Data Bases*, pages 264–275, 1996.

[BHR$^+$07] Kevin Beyer, Peter J Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 199–210, 2007.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, pages 422–426, 1970.

[BMS$^+$06] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of the International Conference on Very Large Data Bases*, pages 475–486, 2006.

[BNF13] National library of france. http://www.bnf.fr, (accessed on 28-06-2013).

[BOI] Boilerpipe. http://code.google.com/p/boilerpipe/.

[BS12] Andreas Broschart and Ralf Schenkel. High-performance processing of text queries with tunable pruned term and term pair indexes. *ACM Transactions for Information Systems*, page 5, 2012.

[BYGJ$^+$08] Ricardo Baeza-Yates, Aristides Gionis, Flavio P Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. Design trade-offs for search engine caching. *ACM Transactions on the Web (TWEB)*, page 20, 2008.

[BYJK$^+$02] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science*, pages 1–10. 2002.

[BYJPW07] Ricardo Baeza-Yate, Flavio Junqueira, Vassilis Plachouras, and Hans-Friedrich Witschel. Admission policies for caches of search engine results. In *String Processing and Information Retrieval*, pages 74–85. 2007.

[Cho10] Gobinda Chowdhury. *Introduction to modern information retrieval*. 2010.

[CHT11] Sarah Cohen, James Hamilton, and Fred Turner. Computational journalism. *Communations of ACM*, pages 66–71, 2011.

[CLYY11] Sarah Cohen, Chengkai Li, Jun Yang, and Cong Yu. Computational journalism: A call to arms to database researchers. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pages 148–151, 2011.

[CMS10] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*. 2010.

[Con10] Paul Conway. Preservation in the age of google: Digitization, digital preservation, and dilemmas. 2010.

[COR13] Stanford corenlp. http://nlp.stanford.edu/software/corenlp.shtml, (accessed on 21-04-2013).

[Coy06] Karen Coyle. Mass digitization of books. *The Journal of Academic Librarianship*, pages 641 – 645, 2006.

[CP08] Matthew Chang and Chung Keung Poon. Efficient phrase querying with common phrase index. *Information Processing and Management*, pages 756–769, 2008.

[CTL91] Bruce Croft, Howard Turtle, and David Lewis. The use of phrases and structured queries in information retrieval. In *Proceedings of the 14th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 32–45, 1991.

[CWC13] ClueWeb09. http://lemurproject.org/clueweb09/, (accessed on 28-06-2013).

[Den12] Dimitar Denev. *Models and methods for web archive crawling*. PhD thesis, Saarland University, 2012.

[DLP99] Erika De Lima and Jan Pedersen. Phrase recognition and expansion for short, precision-biased queries based on a query log. In *Proceedings of the 22nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 145–152, 1999.

[DMSW09] Dimitar Denev, Arturas Mazeika, Marc Spaniol, and Gerhard Weikum. Sharc: framework for quality-conscious web archiving. *Proceedings of the International Conference on Very Large Data Bases*, pages 586–597, 2009.

[DS11] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 993–1002, 2011.

[EA13] European archive. http://www.europarchive.org, (accessed on 28-06-2013).

[FGJV11] Marcus Fontoura, Maxim Gurevich, Vanja Josifovski, and Sergei Vassilvitskii. Efficiently encoding term co-occurrences in inverted indexes. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management*, pages 307–316, 2011.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 102–113, 2001.

[FMNW03] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. A large-scale study of the evolution of web pages. In *Proceedings of the 12th International Conference on World Wide Web*, pages 669–678, 2003.

[FNM85] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, pages 182–209, 1985.

[FPSO06] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems*, pages 51–78, 2006.

[FV10] Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, page 10, 2010.

[GJSS05] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join operations in temporal databases. *The VLDB JournalThe International Journal on Very Large Data Bases*, pages 2–29, 2005.

[GK09] Sairam Gurajada and P. Sreenivasa Kumar. On-line index maintenance using horizontal partitioning. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 435–444, 2009.

[GS09] Qingqing Gan and Torsten Suel. Improved techniques for result caching in web search engines. In *Proceedings of the 18th International Conference on World Wide Web*, pages 431–440, 2009.

[HAN13] Hanzo archives. http://www.hanzoarchives.com/, (accessed on 28-06-2013).

[Hen06] Monika Rauch Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 284–291, 2006.

[HER13] Heritrix Archival Crawler. http://www.digitalhistory.uh.edu/, (accessed on 28-06-2013).

[HLY07] Michael Herscovici, Ronny Lempel, and Sivan Yogev. Efficient indexing of versioned document sequences. In *Advances in Information Retrieval, 29th European Conference on IR Research*, pages 76–87, 2007.

[HPBS12] Matthias Hagen, Martin Potthast, Anna Beyer, and Benno Stein. Towards optimum query segmentation: in doubt without. In *Proceedings of the 21st ACM Conference on Information and Knowledge Management*, pages 1015–1024, 2012.

[HSBW13] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, pages 28–61, 2013.

[HYS09] Jinru He, Hao Yan, and Torsten Suel. Compact full-text indexing of versioned document collections. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 415–424, 2009.

[HZS10] Jinru He, Junyuan Zeng, and Torsten Suel. Improved index compression techniques for versioned document collections. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management*, pages 1239–1248, 2010.

[IA13] Internet archive. http://archive.org, (accessed on 28-06-2013).

[IIP13] International internet preservation consortium. http://netpreserve.org, (accessed on 28-06-2013).

[IM13] Internet memory. http://www.internetmemory.org, (accessed on 28-06-2013).

[KFN10] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. Boilerplate detection using shallow text features. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*, pages 441–450, 2010.

[KMN99] Samir Khuller, Anna Moss, and Joseph (Seffi) Naor. The budgeted maximum coverage problem. *Information Processing Letters*, pages 39–45, 1999.

[KN10] Nattiya Kanhabua and Kjetil Nørvåg. Quest: query expansion using synonyms over time. In *Machine Learning and Knowledge Discovery in Databases*, pages 595–598, 2010.

[KUL] Swedish royal library: Kulturarw^3 – long-term preservation of electronic documents. http://www.kb.se/kw3/ENG/.

[Lal12] Mounia Lalmas. Xml information retrieval. *Understanding Information Retrieval Systems: Management, Types, and Standards*, page 345, 2012.

[LC89] David Lewis and Bruce Croft. Term clustering of syntactic phrases. In *Proceedings of the 13th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 385–404, 1989.

[LHZW11] Yanen Li, Bo-June Paul Hsu, ChengXiang Zhai, and Kuansan Wang. Unsupervised query segmentation using clickthrough for information retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 285–294, 2011.

[LMZ08] Nicholas Lester, Alistair Moffat, and Justin Zobel. Efficient online index construction for text databases. *ACM Transactions on Database Systems*, 2008.

[LS93] David Lomet and Betty Salzberg. Exploiting a history database for backup. In *Proceedings of the International Conference on Very Large Data Bases*, pages 380–380, 1993.

[LS06] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. *Proceedings of the 15th International Conference on World Wide Web*, 9(4):369–395, 2006.

[LSS88] Jong Lee, Eugene Shragowitz, and Sartaj Sahni. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing*, pages 438 – 456, 1988.

[LZW06] Nicholas Lester, Justin Zobel, and Hugh Williams. Efficient online index maintenance for contiguous inverted lists. *Information Processing & Management*, pages 916–933, 2006.

[Mas06] Julien Masannes. *Web archiving*. Springer, 2006.

[MBP13] The million books project. http://archive.org/details/millionbooks, (accessed on 28-06-2013).

[MM93] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.

[MOPW00] Peter Muth, Patrick E. O'Neil, Achim Pick, and Gerhard Weikum. The LHAM Log-Structured History Data Access Method. *The VLDB JournalThe International Journal on Very Large Data Bases*, pages 199–221, 2000.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[MSA$^+$10] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Joseph P Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, and Jon Orwant. Quantitative Analysis of Culture Using Millions of Digitized Books. *science*, 2010.

[MTB$^+$10] Michael Matthews, Pancho Tolchinsky, Roi Blanco, Jordi Atserias, Peter Mika, and Hugo Zaragoza. HCIR Challenge 2010. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[MTO12] Craig Macdonald, Nicola Tonellotto, and Iadh Ounis. Learning to predict response times for online query scheduling. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 621–630, 2012.

[MZ96] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, pages 349–379, 1996.

[NCO04] Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. What's new on the web?: the evolution of the web from a search engine perspective. In *Proceedings of the 13th International Conference on World Wide Web*, pages 1–12, 2004.

[Nér90] Jean Néraud. Elementariness of a finite set of words is co-np-complete. *Informatique théorique et applications*, pages 459–470, 1990.

[NUT13] Nutchwax. http://archive-access.sourceforge.net/projects/nutch/index.html, (accessed on 28-06-2013).

[NWF78] George Nemhauser, Laurence Wolsey, and Marshall Fisher. An analysis of approximations for maximizing submodular set functions. *Mathematical Programming*, pages 265–294, 1978.

[NYT13] New york times annotated corpus. http://corpus.nytimes.com, (accessed on 28-06-2013).

[OAC$^+$12] Rifat Ozcan, Sengor Altingovde, Barla Cambazoglu, Flavio Junqueira, and Oezguer Ulusoy. A five-level static cache architecture for web search engines. *Information Processing and Management*, pages 828 – 840, 2012.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, pages 351–385, 1996.

[PAB13] Bibek Paudel, Avishek Anand, and Klaus Berberich. User-Defined Redundancy in Web Archives. In *Proceeding of the 2013 ACM Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2013.

[PAN13] Pandora archive - national library of australia. `http://pandora.nla.gov.au`, (accessed on 28-06-2013).

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–15, 2010.

[Sal89] Gerard Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[SBBW10] Vinay Setty, Srikanta Bedathur, Klaus Berberich, and Gerhard Weikum. Inzeit: efficiently identifying insightful time points. *Proceedings of the 34rd International Conference on Very Large Data Bases*, pages 1605–1608, 2010.

[SC07] Trevor Strohman and W Bruce Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 175–182, 2007.

[SOL13] Solr. http://lucene.apache.org/solr/, (accessed on 28-06-2013).

[SSdMZ$^+$01] Paricia Correia Saraiva, Edleno Silva de Moura, Novio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 51–58, 2001.

[SSU08] Susan Schreibman, Ray Siemens, and John Unsworth. *A Companion to Digital Humanities*. 2008.

[ST99] Betty Salzberg and Vassillis Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, pages 158–221, 1999.

[Sta11] Efstathios Stamatatos. Plagiarism detection based on structural information. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management*, pages 1221–1230, 2011.

[SWJS01] Amanda Spink, Dietmar Wolfram, Major B. J. Jansen, and Tefko Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science and Technology*, pages 226–234, 2001.

[SWY75] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, pages 613–620, 1975.

[TGMS94] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. *Incremental updates of inverted lists for text document retrieval*. ACM, 1994.

[TIM13] The times. http://www.thetimes.co.uk, (accessed on 28-06-2013).

[TS09] Frederik Transier and Peter Sanders. Out of the box phrase indexing. In *String Processing and Information Retrieval*, pages 200–211, 2009.

[TWS04] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the International Conference on Very Large Data Bases*, pages 648–659, 2004.

[Vaz01] Vijay Vazirani. *Approximation algorithms*. Springer, 2001.

[WAR13] Uk web archive. http://www.webarchive.org.uk, (accessed on 28-06-2013).

[WIK13] Wikipedia. http://en.wikipedia.org/, (accessed on 28-06-2013).

[WMB99] Ian Witten, Alistair Moffat, and Timothy Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

[WSJ13] The Wall Street Journal. http://www.wsj.com, (accessed on 28-06-2013).

[WZB04] Hugh Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Transactions on Information Systems*, pages 573–594, 2004.

[YSL$^+$12] Jing Yuan, Guangzhong Sun, Tao Luo, Defu Lian, and Guoliang Chen. Efficient processing of top-k queries: selective nra algorithms. *Journal of Intelligent Information Systems*, pages 687–710, 2012.

[YSZ$^+$10] Hao Yan, Shuming Shi, Fan Zhang, Torsten Suel, and Ji-Rong Wen. Efficient term proximity search with term-pair indexes. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management*, pages 1229–1238, 2010.

A Appendix

10 commandments, hurricane season, abortion, abraham lincoln, acre, adenocarcinoma, adolf hitler, africa, agnostic, alexander great, allegory, american idol, anal, anderson cooper, andrea lowell, andy milonakis, anus, aorta, appendix, argentina, ash wednesday, ask jeeves, audie murphy, beastiality, bees, beethoven, bill gates, blood tests, brazil, buddha, buddhism, cameltoe, candy samples, chamber horrors match, characters yu gi oh gx, charles darwin, charlie rose, charmed, cher, chris daughtry, chris penn, christianity, chuck norris jokes, cinco de mayo, cleveland steamer, clitoris, cocaine, cold war, columbine, communism, concentration camps, crystal meth, cuba, da vinci code, dana reeve, danzig, darfur, david blaine, deal or no deal, deaths 2006, debra lafave, deposition, dixie chicks, dna, dominican republic, domino harvey, donnie mcclurkin, dopamine, doxycycline, drudge report, dubai, easter, ebay, eleanor roosevelt, elmo's world, emancipation proclamation, emily rose, emo, encarta, england, erection, estrogen, existentialism, facebook, fascism, del castro, ngerprints, av , orence nightingale, french revolution, genocide, georg fuerst, george rr martin, george washington, germany, gloria vanderbilt, goggle, good friday, gospel judas

Figure A.1.: Queries used for WIKI dataset

guns n roses, gwen stefani, hades, haiku, hanso, hanso foundation, harlem renaissance, hawaii, he-man toys, helga sven, henry ford, hermaphrodite, hesiod, high school musical 2, hitler, hotmail, hurricane katrina, hurricane rita, football season , hurricane wilma, hustler, hydrocodone, imperialism, incest, industrial revolution, israel, italy, jack dunphy, japan, joan arc, john adams, john lennon, johnny cash, julian beever, julius caesar, june carter, june carter cash, kama sutra, karma, kelly clarkson, kkk, knights templar, korean war, ku klux klan, lafave, lecithin, led zeppelin, lenin, lent, liger, limewire, liver, lost, louisiana, purchase, lymph nodes, manifest destiny, marcheline bertrand, marijuana, martin luther, marvel scream, maslow's hierarchy needs, matisyahu, may day, maya angelou, mayo clinic, mccarthyism, memorial day, metaphor, mexico, mime, mississippi river, missouri compromise, monroe doctrine, moors, morphine, mortal kombat characters, moses, mozart, mrsa, msnbc, mussolini, naruto, neil armstrong, neuropathy, newgrounds, n, niacin, norepinephrine, nudity, opium, opus dei, oxycodone, palm sunday, pamela rogers, panama canal, pancreas, pandemic, penis, penthouse, peru, pete wentz, peter tomarken, ph, phentermine, plato, playboy, playgirl, poland, polygamy, potassium prednisone, priory sion, prohibition, protein, pus, randy jackson, randy orton, rape, renaissance, roe v. wade, roman numerals, romanticism, rome, ronald reagan, rosa parks, satire, schizophrenia, scientific method, scientologist, scientology, segregation, serotonin, sesame street, shakira, shane macgowan, silent hill, simon cowell, skull island, snopes, sociopath, sodomy, sonny moore, spain, spanish civil war, spanish inquisition, spanking, spiderman 3, spleen, sportsnet new york, stadium arcadium, stalin, statue liberty, stephanie mcmahon, sudoku, superman, symbols, syntax, tachycardia, ten commandments, tet offensive, american dream, beatles, cold war, da vinci code, great depression, kennedy family, last supper, neocons, ten commandments, thomas edison, thong, tiffany fallon, timothy treadwell, trees, truman capote, truman, doctrine, tsunami, tuberculosis, united, vagina, vatican city, venezuela, vicodin, vietnam war, vivian, liberto, vulva, watergate, whitney houston, wiccan, winmx, winston churchill, world war, world war ii, wwe, x-men, xiaolin showdown, yalta conference

Figure A.2.: Queries used for WIKI dataset - cont.

action plan template, animals boarding, boys names, breast implants, british crown jewels, british government summary, british royal family, british royalty, citizenship ceremony dates, criminal record, diana princess wales, different types writing, disability, edinburgh castle, food poisioning, free emergencey signs, international contributions longbenton, kensington palace, kevin ramsay, king george, london metro, mergers acquisitions business, mi5, migrant workers statistics, millenium development goals, minimum wage, national archives, parents rights, pensions, power ranger clip art, renewal wedding vows, renne de chateau, residential home, residential home extensions, retirement speeches, risk assessment, road surfaces, robert edwards fortune, royal family, stuart kings, summary judgement, tax tables, uk immigration, uk passports, vehicle recalls, william morris wallpaper, winchester museum, windsor castle, witness intimidation, woodchurch, work-life balance surveys, world war two names, yin yang symbol

Figure A.3.: Queries used for UKGOV dataset

List of Figures

List of Tables

List of Algorithms