

ABSTRACT

Title of Document:

LONG-TERM INFORMATION PRESERVATION AND ACCESS

Sang Chul Song, Doctor of Philosophy, 2010

Directed By:

Professor Joseph F. JaJa, Department of
Electrical and Computer Engineering

An unprecedented amount of information encompassing almost every facet of human activities across the world is generated daily in the form of zeros and ones, and that is often the only form in which such information is recorded. A good fraction of this information needs to be preserved for periods of time ranging from a few years to centuries. Consequently, the problem of preserving digital information over a long-term has attracted the attention of many organizations, including libraries, government agencies, scientific communities, and individual researchers. In this dissertation, we address three issues that are critical to ensure long-term information preservation and access.

The first concerns the core requirement of how to guarantee the integrity of preserved contents. Digital information is in general very fragile because of the many ways errors can be introduced, such as errors introduced because of hardware and media degradation, hardware and software malfunction, operational errors, security breaches, and malicious alterations. To address this problem, we develop a new

approach based on efficient and rigorous cryptographic techniques, which will guarantee the integrity of preserved contents with extremely high probability even in the presence of malicious attacks. Our prototype implementation of this approach has been deployed and actively used in the past years in several organizations, including the San Diego Super Computer Center, the Chronopolis Consortium, North Carolina State University, and more recently the Government Printing Office.

Second, we consider another crucial component in any preservation system – searching and locating information. The ever-growing size of a long-term archive and the temporality of each preserved item introduce a new set of challenges to providing a fast retrieval of content based on a temporal query. The widely-used cataloguing scheme has serious scalability problems. The standard full-text search approach has serious limitations since it does not deal appropriately with the temporal dimension, and, in particular, is incapable of performing relevancy scoring according to the temporal context. To address these problems, we introduce two types of indexing schemes – a location indexing scheme, and a full-text search indexing scheme. Our location indexing scheme provides optimal operations for inserting and locating a specific version of a preserved item given an item ID and a time point, and our full-text search indexing scheme efficiently handles the scalability problem, supporting relevancy scoring within the temporal context at the same time.

Finally, we address the problem of organizing inter-related data, so that future accesses and data exploration can be quickly performed. We, in particular, consider web contents, where we combine a link-analysis scheme with a graph partitioning scheme to put together more closely related contents in the same standard web

archive container. We conduct experiments that simulate random browsing of preserved contents, and show that our data organization scheme greatly minimizes the number of containers needed to be accessed for a random browsing session.

Our schemes have been tested against real-world data of significant scale, and validated through extensive empirical evaluations.

LONG-TERM INFORMATION PRESERVATION AND ACCESS

By

Sang Chul Song

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:
Professor Joseph F. JaJa, Chair/Advisor
Professor Jimmy Lin
Professor Gang Qu
Professor Amitabh Varshney
Professor Donald Yeung

© Copyright by
Sang Chul Song
2010

Dedication

*To Hyesook, Minjoo, Emily,
and my parents...*

Acknowledgements

Although it is only my name that appears on the cover of this dissertation, I am indebted to so many people around me. First and foremost, I would not be able to express my gratitude eloquently enough toward my advisor, Dr. Joseph F. JaJa. I must acknowledge that I have been incredibly fortunate to have him as my mentor. His encouragement, supervision, support and patience throughout the last five years have enabled me to develop an understanding of the subject.

I would like to thank the members of my dissertation examining committee, Dr. Jimmy Lin, Dr. Gang Qu, Dr. Amitabh Varshney, and Dr. Donald Yeung. They all happily agreed to serve on the committee, and kindly spent their time examining the dissertation and providing valuable advices and suggestions to improve it.

I have been incredibly lucky to have such wonderful group members. Mike Smorul has been a great office mate whose vast knowledge about the Linux system administration and Java programming has always been a dependable resource. I especially thank Jusub Kim who introduced me to my advisor five years ago. I enjoyed working with Muluwork Geremew on digital file formats. I have also had the pleasure to associate with Jing Wu, Zheng Wei, and Chi Cui.

I am grateful that I have met so many wonderful fellow Korean graduate students at Maryland: Younggu Kim, Seung-Jong Baek, Dong Hun Park, and Soo Bum Lee have provided me with valuable life advices and guidance in hard times. I have had a pleasure to know Joon-Hyuk Yoo, Sunghyun Chun, Dongwoon Hahn, Jaehwan Lee, Woochul Jeon, Seokjin Kim, Youngho Cho, Zaeill Kim, Sang-Kyo Han, Eunmi Kim,

Kwangsik Choi, Taek-II Oh, Jin Seock Ma, Young Wook Kim, Keunmin Ryu, Hojin Kee, Inseok Choi, Kyunjin Yoo, Woomyung Park, Sukhyun Song, Jookyung Lee, Sungwoo Park, Hyung Tae Lee, Sung Jun Yoon, In Keun Cho, Jonghyun Choi, Jeongho Jeon, Kyowon Kim and Kangmook Lim, Eunyoung Seo, Ginnah Lee and Hyunsoo Kim. I especially thank Il-Chul Yoon and Minkyung Cho - We all hated brainteasers and tricky algorithm questions, but I still thank them for bringing us together during the hardest time of my life.

Above all, I am deeply indebted to my family. My parents have always been the essential component of my life. Their unconditional love, support, encouragement, and sacrifice have made who I am now. I also thank my elder sister, Mi-Yeon Song, and her family who have always stood by me and helped me in countless ways. My parents-in-law have been truly patient and supportive even though they had not been able to see their loving daughter for several years.

My deepest thanks must go to my beautiful wife and our precious kids. I know Hyesook would have become a greater scholar than I am now, but she selflessly and successfully turned herself into an even more respectable person – a caring wife and wise mother. Her love, endurance and support have undeniably been the vital source of the happiness in our family. I thank my two daughters, Minjoo and Emily, for being cute, smart and healthy. They have been, and will be, the greatest reason and motivation for every important step I take in my life. I know that they will soon rock the world, or better yet, they will become as beautiful and clever as their mom.

Table of Contents

Dedication	ii
Acknowledgements	iii
Table of Contents	v
List of Tables	ix
List of Figures.....	x
Chapter 1 Introduction.....	1
1.1 Background.....	2
1.2 Fragility of Digital Information	4
1.3 Information Discovery and Delivery	5
1.4 Data Layout.....	7
1.5 Contributions	8
1.6 Dissertation Outline	11
Chapter 2 Ensuring Long Term Integrity of Digital Information	12
2.1 Overview.....	12
2.2 Related Work	13
2.2.1 Basic Techniques	13
2.2.2 Techniques for Digital Archives	16
2.3 Our Approach	20
2.3.1 Constructing Integrity Tokens and Witness Values.....	22
2.3.2 Updating Integrity Information.....	27
2.4 Putting the Ideas Together – the ACE Tool.....	29

2.4.1 ACE Components	29
2.4.2 ACE Workflow	30
2.4.3 ACE Preliminary Performance Evaluation	33
2.5 Conclusion	35
Chapter 3 Full-text Search Approach for Time-evolving Data	36
3.1 Overview.....	36
3.2 Related Work	39
3.3 Model	41
3.4 Our Approach	43
3.4.1 Analytical Model	46
3.4.2 Temporal Relevance and Scoring	52
3.5 Empirical Evaluation	55
3.5.1 Datasets Used.....	56
3.5.2 Evaluation Methodology.....	57
3.5.3 Empirical Results on Total Number of Postings.....	59
3.5.4 Empirical Results on Average Number of Postings Examined for a Typical Query Load.....	62
3.5.5 Empirical Evaluation of Ranked Search Results	64
3.6 Conclusion	67
Chapter 4 Physical Location Index for Time Evolving Data.....	68
4.1 Overview.....	68
4.2 Related Work	70
4.2.1 Location Index for Web Archives.....	70

4.2.2 Persistent Data Structures	72
4.3 Our Indexing Goal	74
4.4 Our Strategy – Persistent Indexing Structure for Archives (PISA)	75
4.4.1 Persistent Indexing Structure for Archives (PISA).....	76
4.4.2 Operations in PISA	78
4.5 Performance Analysis	88
4.5.1 Query Time	89
4.5.2 Insert Time	90
4.5.3 Space	92
4.5.4 Performance Comparison.....	94
4.6 Summary	94
Chapter 5 Optimizing Data Layout for Web Contents for Fast Access	96
5.1 Overview.....	96
5.2 Related Work	97
5.2.1 Archival Storage	98
5.2.2 Graph Partitioning Techniques	99
5.2.3 Link Analysis Technique – PageRank.....	101
5.3 Our Method.....	104
5.3.1 Edge Weight.....	105
5.3.2 Graph Partitioning.....	106
5.4 Experimental Evaluation of Our Scheme	108
5.4.1 Edge-Cut	112
5.4.2 Simulation	113

5.5 Conclusion	118
Chapter 6 Concluding Remarks and Future Work.....	120
Appendix A PISA: Parameters and Invariants	124
A.1 Parameter Conditions.....	124
A.2 Determining Parameter Values	127
Appendix B Proofs of $E[X]$ and $E[Y]$.....	128

List of Tables

Table 2.1: ACE Performance	34
Table 3.1: Datasets	57
Table 3.2: Time Window Sizes	60
Table 4.1: PISA Parameters	82
Table 4.2: PISA Variables	82
Table 4.3: Performance Comparison (Cells with the best performance are shaded) ..	94
Table 5.1: The Two Datasets Used for Evaluating Our Method	109
Table 5.2: Edge-Cut Results	112
Table 5.3: Simulation Parameters	113

List of Figures

Figure 2.1: Merkle Tree	25
Figure 2.2: ACE Architecture	30
Figure 3.1: Document Versions with Validity Time Intervals	42
Figure 3.2: Structure of Our Proposed Temporally-augmented Inverted Index.....	46
Figure 3.3: Illustration of Parameters X and Y	48
Figure 3.4: Illustration of Values of λ_i , μ_i and δ_i	49
Figure 3.5: Wikipedia: $E[X]$, $E[Y]$ and $E[X]+E[Y]$	51
Figure 3.6: Library of Congress: $E[X]$, $E[Y]$, and $E[X]+E[Y]$	52
Figure 3.7: Temporally-anchored Queries	53
Figure 3.8: Virtual Time Windows.....	58
Figure 3.9: Wikipedia: Total Number of Postings.....	60
Figure 3.10: Library of Congress: Total Number of Postings	61
Figure 3.11: Wikipedia: Average Number of Postings Examined	63
Figure 3.12: Library of Congress: Average Number of Postings Examined.....	63
Figure 3.13: Wikipedia: Relative Recall and Kendall's τ	65
Figure 3.14: Library of Congress: Relative Recall and Kendall's τ	66
Figure 4.1: Blocks and Entries in PISA	77
Figure 4.2: PISA Example	78
Figure 4.3: URLTIMEQUERY Operation	79
Figure 4.4: URLTIMESPANQUERY Operation.....	80
Figure 4.5: TIMESLICEQUERY Operation	81
Figure 4.6: PISA Block (Live entries clustered together for illustrative purpose)	82

Figure 4.7: INSERT Procedure	84
Figure 4.8: BLOCKINSERT Procedure	84
Figure 4.9: VERSIONSPLIT Procedure.....	85
Figure 4.10: KEYSPLIT Procedure.....	85
Figure 4.11: INSERT Z, X, and Z.....	86
Figure 4.12: KEYSPLIT after Inserting T.....	87
Figure 4.13: INSERT S, S, and S	87
Figure 4.14: VERSIONSPLIT after INSERT(S, 11/1/2007, ∞).....	88
Figure 5.1: Conventional Allocation of Pages to Containers	110
Figure 5.2: Container Construction Based on Graph Partitioning with or without EdgeRank (Line 3).....	111
Figure 5.3: Histogram of Number of Inter-Container Hops for UMIACS Web Graph	115
Figure 5.4: Histogram of Number of Distinct Containers Accessed for UMIACS Web Graph	115
Figure 5.5: Histogram of Number of Inter-Container Hops for Stanford Web Graph	116
Figure 5.6: Histogram of Number of Distinct Containers Accessed for Stanford Web Graph	116
Figure 5.7: Average Number of Inter-Container Hops.....	117
Figure 5.8: Average Number of Distinct Containers	118
Figure B.1: Illustration of Values of λ_i , μ_i and δ_i	129

Chapter 1

Introduction

Preservation of digital information is widely recognized as a critical emerging issue that will soon confront most organizations, including government agencies, libraries and museums, and scientific communities. As a consequence, these organizations along with individual researchers have recently started to give considerable attention to the problems that need to be resolved to address long-term preservation and access of digital information. Their studies have identified major challenges regarding institutional and business models, technology infrastructure, and social and legal frameworks. Focusing on the technology component, good summaries of the main technology challenges are presented in [26,27,70]. As a result, a significant number of initiatives have been set up to develop prototypes to address some of the challenges. These initiatives include the Internet Archive [34], the National Library of Australia's Pandora project [62], the Stanford University Libraries' Lots of Copies Keep Stuff Safe (LOCKSS) project [47], the Transcontinental Persistent Archive Prototype (TPAP) [56], the Universal Virtual Computer [45], the Electronic Records Archives program at the National Archives [76], the Library of Congress' National Digital Information Infrastructure and Preservation Program (NDIIP) [76], and the International Internet Preservation Consortium (IIPC) [32].

In this chapter, we provide an overall background and present a summary of some of the major technical challenges involved in digital preservation. We end the chapter with an overview of the main contributions of this dissertation.

1.1 Background

In the era of digital information, the efforts to preserve the human knowledge have broadened to also include documents, images, audio, video, social networking, and their inter-relationships in their digital form. An unprecedented amount of information encompassing almost every facet of human activities across the world exists in digital form, and is also growing at an extremely fast pace. Moreover, the digital representation is often the only form in which such information is recorded. It has become evident that the traditional archiving process of physical artifacts is extremely lacking to manage the preservation of digital information. Clearly, novel methodologies are needed to curate, store, preserve, and access this new type of information on a long-term basis.

There is a generally common agreement that the long term preservation of digital information requires systematic methodologies to address the following requirements [33].

- Encapsulation of information regarding content, structure, context, provenance, and access within each digital object to enable the long-term maintenance and lifecycle management of the digital object.

- Efficient management of technology evolution, both hardware and software, and the appropriate handling of technology obsolescence (for example, format obsolescence).
- Efficient risk management and disaster recovery mechanisms either from technology degradation and failure, or natural disasters such as fires, floods, and hurricanes, or human-induced operational errors, or security failures and breaches.
- Efficient proactive mechanisms to ensure the authenticity and integrity of content, context, and structure of archived information throughout the preservation period.
- Ability for information discovery and content access and presentation, with an automatic enforcement of authorization and IP rights, throughout the lifecycle of each object.
- Scalability in terms of ingestion rate, capacity and processing power to manage and preserve large scale heterogeneous collections of complex objects, and the speed at which users can discover and retrieve information.
- Ability to accommodate possible changes over time in organizational structures and stewardships, relocation, and repurposing.

Meeting the above requirements, however, inevitably involves a number of technical challenges. They are due in part to the large amount of important digital information generated on a daily basis, the fast pace of technology evolution (and the corresponding format changes), the relative fragility of digital information and computing infrastructure (centralized vs. distributed, or federated), the temporality of

each preserved item, and the complex relationship among the preserved items. Some of the issues are explained in more detail in the following sections.

1.2 Fragility of Digital Information

Digital information is extremely fragile and susceptible to various threats – much more so than paper records or physical artifacts [70]. Even in a perfect world with no malicious attacks, there exist no digital media that allow for “permanent” recording. Various media technologies degrade over time, potentially causing random bit errors, yet no precise timeline is usually given for data deterioration, either. For example, Bairavasundaram, et al [2] reports that disk drive failures contributed 400,000 instances of data corruption over a period of 41 months in 1.53 millions of disk drives.

Moreover, technology changes involving systems and software may render old contents inaccessible. There can also be malicious security attacks, altering or destroying digital contents without being immediately detected. Accidental operational errors cannot be overlooked either. Permanent loss of data can also occur due to natural hazards and disasters such as fires, floods, and hurricanes.

We note that most of the archive’s holdings may be accessed infrequently, and hence several cycles of technology evolution may occur, thereby causing corrupted files to go undetected until it is too late to retrieve the initial content. Two additional factors complicate this problem further. First, an object will typically be subjected to a number of transformations during its lifetime, including those migrative transformations due to format obsolescence. These transformations may alter the object in unintended ways. Second, all current integrity checking mechanisms are

based on some type of cryptographic techniques, most of which are likely to become less immune to potential attacks over time, and hence they will need to be replaced with stronger techniques when this occurs. As a consequence, a thorough approach to ensure the integrity of a long-term archive has to also be able to address these two factors.

Efforts to preserve the integrity of the objects have actively been made since the introduction of magnetic tapes in 1950s. As a result, many techniques have been introduced, ranging from simple binary checksums to more complicated public-key-based digital signatures. However, none of these techniques seems to offer a general approach that is scalable, cost-effective, and can efficiently manage the integrity of digital objects over the lifetime of the archive. In particular, none of the existing techniques is capable of proactively monitoring and detecting any alterations, including malicious ones, to the data in a cost effective way.

1.3 Information Discovery and Delivery

Another critical component of long-term preservation is to allow users to find and explore information contained in an archive within a temporal context. In essence, the key purpose of long-term preservation is to pass the current knowledge to future generations. It is, thus, vital for any preservation system to provide an effective way of finding and accessing the relevant contents as needed by future users.

Unfortunately, this is a highly non-trivial problem, due to the large, ever-growing size, and evolving archived data, the temporality of the preserved items, and more importantly, the hard-to-predict access modes that future users would expect.

Preservation systems that solely rely on a relational database with well defined schemas may allow their users to find information easily using well-structured queries. However, fitting every type of digital objects into a fixed set of schemas is simply impossible, especially that future usage may require completely new ways to search and access the archived data. We clearly need a more general and flexible framework to enable effective information discovery and delivery of the preserved data.

A popular access scheme for digital libraries is through a pre-defined metadata catalog format such as MARC [48]. In this scheme, searches are performed on the predefined set of metadata fields. However, it has at least three serious deficiencies. First, it has a scalability problem. For example, as of today, the Minerva project of the Library of Congress [54] was able to catalog only about 2,300 sites among 30,000 archived sites in the September 11 collection. Next, the classification hierarchy currently in use for the catalog-based scheme is likely to evolve over time. Updating existing records according to a new hierarchy will be extremely expensive for large archives. Finally, and most importantly, the catalog scheme significantly restricts users' accessibility, greatly limiting the effective information discovery process. Users who want to find documents that contain a rare term will have a hard time locating the right documents unless the rare term was successfully captured in a metadata field, which is unlikely in most cases.

A much more flexible scheme is full-text search. In order to more effectively support full-text search over a conventional "static" collection, a substantial amount of work has been developed through the traditional information retrieval field,

resulting in particular in extremely effective web search engine technologies. However, the direct application of the search engine techniques is not suitable for long-term archives, due to the temporality of preserved items. Queries with temporal constraints will need to look up the entire index, which will grow in size as the archive grows over time. Furthermore, relevancy scoring is performed against the entire holdings. This renders the conventional full-text search scheme inefficient for handling queries within a temporal context.

We are not aware of any prior work that incorporates the temporal dimension in an integral way for full-text search including temporal scoring. In fact, most of the published papers seem to take the ‘search-all-then-filter’ approach, which is inherently inefficient, and focus on improving the search performance by reducing the search space using a number of augmented data structures.

1.4 Data Layout

Organizing items to be preserved in storage should receive careful attention so as to support quick access and retrieval. This is analogous to arranging physical items in a grocery shop – if items that are likely to be purchased together are placed closely, customers will be able to more efficiently find them. Similarly in a digital archive, in order to support quick access, we must take into account users’ expected access patterns when organizing information in storage; yet, it is almost impossible to predict such patterns during data preparation and ingestion.

A possible approach is through data mining. That is, the access log can be analyzed to discover the group of items that are more likely to be accessed together.

However, not only does this approach require the information about past accesses, the newly ingested data may also be completely of different types, making data mining techniques inappropriate.

A better approach can be based on inter-related items. Indeed, many types of digital information are inter-related. Web pages are hyperlinked to one another, scholarly papers cite one another, and computer program files include/link one another. In many cases, these relationships are explicitly expressed and easy to grasp too. This scheme relies only on the inter-relationships among the newly ingested data, and thus is independent of the current size of the archive. Under the assumption that these inter-related items are more likely to be accessed closely in time, organizing the more closely related items together can help improve the performance of handling the future access requests.

Although the second approach may be more appealing for inter-related data, there are two issues that need to be addressed. First, we need to validate the assumption that more related items will be accessed more closely in time. Second, we need to come up with a good way to find groups of related items, and develop good layout schemes to place related items as close to each other as possible.

1.5 Contributions

In this dissertation, we consider the topic of long-term preservation and access of digital information. We particularly focus on three critical issues: ensuring the integrity of the preserved data over the long term, retrieving relevant information within a temporal context, and optimizing the layout of archived contents.

Briefly, our main contributions include the following.

- A methodology that proactively ensures the integrity of the preserved contents on a long-term basis, in an efficient and rigorous fashion.
- A methodology that enables fast and effective discovery and delivery of preserved information through the full-text search with temporal constraints.
- A methodology that organizes inter-related data such that future navigation through the preserved data can be efficiently performed

Our integrity checking scheme detects any alterations in an archive including malicious ones. More specifically, we introduce efficient and rigorous cryptographic techniques and related procedures to periodically audit the integrity of the various objects held in the preservation system, which, with high probability, will be able to discover any changes made to any preserved item in the system, including changes introduced by a malicious user. Our methodology also allows a party independent of the archive to audit any object in the archive and certify its integrity with extremely high probability, as long as a small size of cryptographic information is kept intact. The current implementation of our scheme, Audit Control Environment (ACE), has been deployed in multiple sites across the country, including the University of Maryland, San Diego Super Computer Center, National Center of Atmospheric Research, and North Carolina State University. For the past two years, it has been actively monitoring 20TB of data managed by the Chronopolis Consortium. Collections include the California Digital Library's Web-at-Risk collection, the collection from InterUniversity Consortium for Political and Social Research, and

also the collections from the Scripps Institution of Oceanography–Geological Data Center and North Carolina Geospatial Data Archiving Project.

Our second major contribution concerns the development of a new methodology for temporal information retrieval. More specifically, we are interested in a full-text search index that returns a list of (document ID, time) pairs given a search phrase and a time constraint. We introduce an approach based on time windows, where a separate inverted index is built for each time window. We empirically prove how our index organization substantially limits the search space while allowing the efficient and scalable computation of the relevance scores relative to the state of the archive as it existed during the time constraint

The final major contribution concerns the problem of how to place inter-related data in storage to optimize access time. We particularly consider a web archive since web objects serve as an excellent example of complex digital information, possessing both spatial and temporal inter-relationship. Note that the techniques and methodologies developed for web objects may be extended to other forms of digital objects without much difficulty. More specifically, we address the problem of how to organize the web objects so that we will be able to navigate through the linking structure of the web objects as effectively as possible. In our approach, we utilize a link analysis scheme and a graph partitioning heuristics to determine which web objects should be placed together in the same container. Our methodology is very general and can be used to optimize different browsing patterns. We perform simulations on multiple real-world data to illustrate the performance of our scheme

and compare it to the common scheme currently used to organize web objects into web containers.

1.6 Dissertation Outline

This rest of the dissertation is organized as follows. We start with the long-term integrity issue in Chapter 2, where we introduce our approach based on simple, yet rigorous cryptographic techniques. In Chapters 3 and 4, we examine the information discovery and delivery for a long-term archive. In particular, we introduce a novel full-text indexing scheme for time-evolving data in Chapter 3, followed by an optimal persistent data structure that can return the location information of a corresponding data item for an arbitrary temporal query in Chapter 4. In Chapter 5, we present a methodology that analyzes inter-relationships among objects to be preserved to better layout them on storage in such a way as to optimize future access performance. We conclude in Chapter 6.

Chapter 2

Ensuring Long Term Integrity of Digital Information

In this chapter, we address the core requirement of ensuring integrity of information in a long-term preservation system. Our approach, based on rigorous cryptographic techniques, involves the generation of a small-size integrity token for each object, some cryptographic summary information, and a framework that enables cost-effective regular and periodic auditing of the archive's holdings depending on the policy set by the archive.

2.1 Overview

One of the most challenging problems in digital preservation is how to ensure the integrity of each object of the archive's holdings throughout the lifetime of the object. Digital information is, in general, very fragile due to many potential risks ranging from hardware and software failures to major technology changes rendering current software and hardware unusable, to the ever-growing number of computer and networking security breaches.

A number of bit-level integrity checking techniques tailored for storage systems have been described in the literature [65,66,72]. However, these techniques fall short of the requirements of a long-term digital archive. Other techniques have been developed specifically for digital archives, including those that appeared in

[14,46,85,47], but none seems to offer a general approach that is applicable to the different emerging architectures for digital archives (including centralized, peer to peer, and distributed archives) and that is capable of proactively monitoring and detecting any alterations to the data in a cost effective way.

The main focus of our study, therefore, is the development of a rigorous methodology to certify the integrity of any object in the archive's holdings, and detect any alterations, including malicious alterations. More specifically, we introduce efficient cryptographic techniques and related procedures to periodically audit the integrity of the various objects held in the archive, which will be able, with high probability, to discover any changes made to any object in the archive, including changes introduced by a malicious user. In fact, our methodology allows a party independent of the archive to audit any object in the archive and certify its integrity with extremely high probability, as long as around 100 KB/year of cryptographic information is kept intact.

2.2 Related Work

In this section, we describe some of the most common strategies used to ensure data integrity starting with the basic techniques for bit streams stored on various types of media or transmitted over a network.

2.2.1 Basic Techniques

Data residing on storage systems or being transmitted across a network can get corrupted due to media, hardware, or software failures. Disk errors, for example, are

not uncommon, and data on disk can get corrupted silently without being detected because a faulty disk controller causes misdirected writes [72]. This type of errors remains undetected because most storage software expects the media to function properly or fail explicitly rather than mis-operate at any point during its life time. The integrity of data can also get compromised because of software bugs. For example, data read from a storage device can get corrupted due to a faulty device driver or a buggy file system which can cause data to become inaccessible [72]. Moreover, data integrity can be violated because of accidental use or operational errors. Unintended user's activity might cause the integrity to be broken. For instance, deletion of a file might lead to a malfunction of specific application/system software that depends on the accidentally deleted file. As a result of this action, integrity violations may occur.

The simplest technique for implementing integrity checks is to use some form of *replication* such as mirroring. The integrity verification can then be made by comparing the replicas against each other. This method can easily detect a change in the stored data only if the modification is not carried out in all the replicas and no errors are introduced during data movement. While maintaining at least one copy of a replica is inevitably necessary to recover from all types of potential data corruptions, performing constant bit-by-bit replica comparisons to detect integrity violation for every object in an archive is an expensive operation that is prone to errors and that cannot counter malicious alterations.

A well-known approach used in RAID storage is based on coding techniques, the simplest of which is *parity checking* [65]. The parity across the RAID array is computed using the XOR logical function. The parity value is stored together with the

data on the same disk array or on a different array dedicated to the parity itself. When the disk containing the data or the parity fails, the data or parity can *sometimes* be recovered using the remaining disk and performing the XOR operation. The XOR parity is a very special type of *erasure codes*, which can be much more powerful ([66] explains erasure codes well). They all involve expanding the data using some types of algebraic operations in such a way that *some* errors may be detected and corrected. While these techniques are critical in maintaining some level of bit-level integrity on storage systems, they are not designed to support high-level data integrity since decoding will be required every time the data accessed, and they entail a significant expansion of the data. Moreover, since only certain errors can be corrected, they still require that a “master copy” be stored in some kind of a back-up system or a “dark archive”.

A widely used method is based on *cryptographic hashing* (also called *checksum*) techniques. In this approach, a checksum of the bit-stream is computed and is stored persistently either with the data or separately. The checksum is calculated using a cryptographic hash algorithm. In general, a cryptographic hash algorithm takes an input of arbitrary length and converts it into a single fixed-size value known as a *digest* or *hash value*. A critical property of cryptographic hash algorithms is that they are based on *one-way* functions, that is, given the hash value of a bit-stream A, it is computationally infeasible to find a different bit-stream B that has the same hash value [37,50]. Assuming that the hash values are correct, data integrity can be verified by comparing the stored hash value with a newly computed hash from the data. Although no known hash function has been proven to be truly one-way, the most

common hash functions in use include MD5, SHA-1, SHA-256, and RIPEMD-160, all of which seem to work well in practice (in spite of the recent attacks that illustrated how to break MD5 [81] and SHA-1 [80]). The major problem with this scheme is that it cannot detect malicious alterations since the hash function used by an archive is usually well-known, and hence an intruder or a malicious user within the archive can change an object and the corresponding hash value so that they still match.

2.2.2 Techniques for Digital Archives

We now describe some of the most notable methods that have been suggested for integrity verification for digital archives.

The most popular and, perhaps, the most important method for addressing integrity checking of digital archives is to compute a hash for each object in the archive and store the hashes in a separate, secure, and reliable registry (the hash could in addition be stored with the object as well). Integrity auditing involves periodic sampling of the content of the archive, computing the hash of each object, and comparing the computed hash with the stored hash value of the object. While such a scheme may be sufficient for small, centralized archives, it has two serious shortcomings relative to our stated goals. The first is that a malicious user within the archive or an external intruder can modify both an object and its corresponding hash value (since the hash function is known), in which case there will be no way to detect such an error. The second shortcoming is the fact that the whole scheme depends on ensuring the integrity of all the hash values, which will grow linearly with the number of objects in the archive. Even in the absence of malicious alterations, this is a non-

trivial problem for large archives over the long term, especially because the hashing schemes themselves will inevitably change over time in which case we have to track the particular hashing scheme used at any specific time. In the method that we will propose, we only need to ensure the integrity of a single hash value per day, independent of the number of objects in the archive, which is a substantially easier problem to manage.

Another approach uses a combination of replication and hashing. In this approach, each digital object is replicated over a number of repositories. Integrity checking can be performed by computing the hash of each copy locally, and sending all the hashes to an auditor. A majority vote enables the auditor to discover the faulty copies, if any. This is the primary integrity scheme used in LOCKSS [47], which is a peer-to-peer replication system for archiving electronic journals in which each participating library collects its own copy of the journals of interest. LOCKSS uses a peer-to-peer inter-cache protocol (LCAP) which is a cache auditing protocol. It runs LCAP continuously among all the caches to detect and correct any damage to cached contents. The process is similar to opinion polls in which all the caches vote. When a storage peer in LOCKSS calls for an audit of a digital object, each peer that owns a replica computes the corresponding hash value and sends back the value to the audit initiator. If the computed digest agrees with the overwhelming majority of the replies, then the object is believed to be intact. If the digest disagrees with the overwhelming majority, the object is believed to be tampered with, and the copy is discarded while a new copy is fetched from the publisher or one of the caches with the right copy. As such, LOCKSS is the only scheme described in this sub-section which handles both

detection and correction simultaneously. However, this approach depends crucially on the assumption that there are many replicas for each object. While this assumption may be reasonable for archiving electronic journals at different libraries, many of the current archives do not use the peer-to-peer infrastructure, or create many replicas of each archival object. A replica voting approach can be expensive, requiring a significant communication overhead. In general, achievement of consensus among distributed nodes that do not trust each other (and some of which may be faulty) is a difficult problem that has been studied extensively in the distributed computing literature. In fact, as reported in [19], about 50 malicious nodes could abuse the LOCKSS protocol to prevent a network of 1000 nodes from auditing their contents. We note that additional set of defenses [19] including admission control, de-synchronization, and redundancy can be used to counter such an attack but clearly this makes the scheme significantly more complicated and costly.

Another possible approach is to make use of *digital signatures* [11] based on *public key cryptography*. In essence, such a scheme involves a private–public key pair for performing signing/verification operations, and a supporting public-key infrastructure. The basic premise is that the private key is only known to the owner, and the public key is widely available. A message signed by a private key can be verified using the corresponding public key. The digital signature technology takes direct advantage of this property. The digital object is signed using the private key (note that the signature depends on the digital object *and* the private key), and anybody can verify the signature using the corresponding public key. If the verification process succeeds, the digital object is considered intact (and the identity

of the author of the signature verified). Hence, a possible approach to preserving the integrity of digital archives would be to sign each digital object using a private key only known to the archive. However, the certificates (public keys signed by a widely trusted certificate authority) have a finite life with a fixed expiration date. Hence, we need to have a trusted and reliable method to track the various public keys used over time. In general, this is a difficult problem that can be solved using sophisticated techniques based on Byzantine agreement protocols [42] and threshold cryptography [10], which shed serious doubts on its practicality in a production environment. Also, should the private key of the archive be compromised, the whole archive becomes at risk. This implies that a malicious user within the archive or an intruder, who gets access to the private key, can easily compromise the contents of the whole archive. Another potential problem with this scheme is its complete dependence on a third party, such as certificate authorities, which may or may not exist over time.

We now introduce the time-stamping technique, which provides an alternative approach to the digital signature scheme outlined above. A time stamp of a digital object D at time T is a record that can be used any time in the future (later than T) to verify that D existed at time T . The record typically contains a time indicator (date and time) and a guarantee (that depends on the time-stamping service) that D existed in exactly this form at time T . One way to implement time stamping is through a Time Stamping Authority (TSA) that attaches a time designation to the object (or its hash) and signs it using the private key of the TSA. The British Library [14] uses this strategy through an independent TSA. With the usage of the public key of the TSA, any alteration to any object, malicious or otherwise, can be detected, which in fact

achieves one of our major objectives. However, the verification procedure depends completely on the trustworthiness of a single entity, namely, the TSA. Should the TSA be compromised or disappear sometime in the future, the whole scheme breaks down completely. Moreover, this scheme is computationally expensive, and we still have to deal with the problem of tracking the various public keys used by the TSA over time.

Another approach to time stamping, which will be used as the basis for our scheme, makes use of *linked* (or *chained*) *hashing* [21], which amounts to cryptographically chaining objects together in a certain way such that a temporal ordering among the objects can be independently verified. In this approach, there is no need for a fully trusted third party or for tracking certificates over time. In an attempt to address the problem of tracking public keys in a digital signature scheme, the linked hashing technique was also suggested to time stamp the public keys [46]. Our scheme directly applies the linked hashing to target objects, thereby eliminating the necessity of maintaining the public-key infrastructure.

In the next section, we will describe the linked hashing technique as used in our approach and demonstrate its ability to achieve our goals in a cost-effective way without depending on a fully trusted archive or a third party.

2.3 Our Approach

As can be seen from the previous section, the previous integrity checking schemes revolve around the following techniques:

- Majority voting using replicated copies of the object or their hashes.

- Computing and saving a digest (“fingerprint”) for each object, using some well-known hash functions. The auditing process consists of computing the digest from the object and comparing it to the saved digest.
- Creating a digital signature of the object and saving it “with the object.” The auditing process makes use of the public key of either the archive or a third party depending on the particular scheme used. Either way, the integrity of the scheme requires a fully trusted third party and the tracking of certificates over time.

We start by introducing the formal notion of a cryptographic hash function. Such a function compresses an arbitrarily long bit-string into a fixed length bit-string, called the hash value, such that the function is easy to compute but it is computationally infeasible to determine an input string for any given hash value. More formally, we would like our hash function H to satisfy the following two properties.

- *Pre-image resistance (one-way property)*: Given any hash value x , it is computationally infeasible to find any bit-string m such that $x = H(m)$.
- *Weak collision resistance*: Given any bit-string m , it is computationally infeasible to determine a different bit-string m' such that $H(m) = H(m')$.

Another property that is sometimes a requirement of cryptographic hash functions is given here.

- *Collision resistance*: It is computationally infeasible to determine any two different strings m and m' such that $H(m) = H(m')$.

These assumptions are the basis for many well-known cryptographic algorithms, including those used in public-key cryptography (see, for example, [50]). Unfortunately, none of the available hash functions can be shown to satisfy these properties. However, several are accepted by the community as reasonably secure and are currently in widespread use. As noted in Section 2.2, recent study has shown how to break the schemes based on MD5 and SHA-1, but the actual threat posed by such study is not clear and, moreover, there are other schemes that remain intact. It is anticipated that stronger algorithms will be developed over time and, hence, any auditing strategy for long-term digital archives has to provide mechanisms to integrate the newer algorithms without compromising the integrity of the objects that used earlier algorithms.

2.3.1 Constructing Integrity Tokens and Witness Values

The starting point of our approach is a scheme that computes a digest for each object and stores the corresponding digests in a separate registry. A digest is typically the result of applying a one-way hash function on the object, but for our purposes, we will not exclude other techniques for generating digests especially for multimedia objects. As mentioned earlier, a major problem with this scheme is how to ensure the integrity of the digest registry over the long term, especially because the registry grows linearly with the number of objects ingested into the archive. Clearly “attaching” the digest to the object does not solve this problem either.

One can address this problem by compressing all the digests into a small number of hash values, which we will call *witness values*, using collision-resistant, one-way

hash functions. For example, we can generate one witness value per day, which cryptographically represents all the objects processed during that day, and hence the total size of all the witness values over a year is quite small (around 100 KB), independent of the number of objects processed during the year.

Given the small size of the witness values, they can be saved on reliable read-only media such as newspapers or archival quality optical media, and hence their integrity can be assured under reasonable assumptions about caring for the media and refreshing the content often as necessary. However it will be extremely time-consuming to conduct regular audits on a large scale archive using the witness values because the auditing of a single object will require the retrieval of the digests of all the objects processed during a day as well as reading the corresponding witness value from a reliable medium. We next show how to counter this problem in a cost effective way.

In order to simplify the presentation, we consider the typical scenario where the generation of the cryptographic information necessary for integrity auditing is placed at the end of the ingestion process, just before an object is archived. We organize the processing of objects into rounds, each of which covers some time interval that is dynamically determined. The length of the time interval depends on the operation of the archive, and may correspond to a fixed duration such as a minute or an hour, a number of objects between a certain minimum and a certain maximum, or may correspond to the time it takes to process a batch of objects according to the archive's schedule. During each round, digests of all the objects being processed can be

compressed using any number of schemes, including, for example, the trivial scheme of hashing a concatenation of all the digests in a certain order.

A particular class of such schemes is based on the so-called hash linking, which was introduced to ensure that the *relative temporal ordering* of the objects processed during a round is preserved and cannot be altered without changing the final value. We will make use of the *Merkle tree* [51], which is one of the most widely used hash linking schemes. More specifically, the digests of all the objects being processed in a round form the leaves of a balanced binary tree such that the value stored at each internal node is the hash value of the concatenated values at the children. A random digest value may also be inserted into the tree at each level to ensure that the number of nodes at each level is even (except for the root). The value computed at the root of the tree is the *round hash value*, which represents the compressed value of all the digests (and objects) processed during the round. That is, a change to any of the objects will result in a different round hash value, and, moreover, it is computationally infeasible to determine another set of objects (including reordering the objects) that will yield the same round hash value.

We now define the proof of the digest of an object, represented in a leaf of the Merkle tree, as the sequence of the hash values of the siblings of all the nodes on the unique path from that leaf to the root.

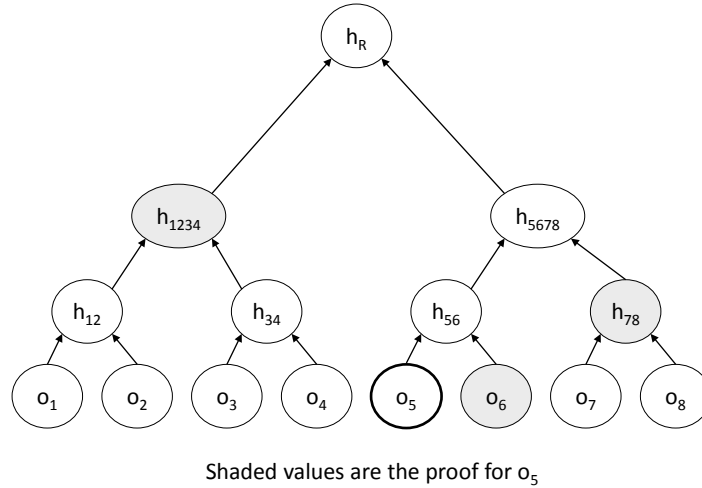


Figure 2.1: Merkle Tree

Consider, for example, a round involving eight objects with the digest values h_1, h_2, \dots, h_8 (See Figure 2.1 for the corresponding tree). The values of the internal nodes are given by:

$$\begin{aligned}
 h_R &= H(h_{1234} \| h_{5678}), \\
 h_{1234} &= H(h_{12} \| h_{34}), & h_{5678} &= H(h_{56} \| h_{78}), \\
 h_{12} &= H(h_1 \| h_2), & h_{34} &= H(h_3 \| h_4), & h_{56} &= H(h_5 \| h_6), & h_{78} &= H(h_7 \| h_8)
 \end{aligned}$$

The proof of the object whose digest value is h_5 will be the following sequence:

$$PR_5 = \{(h_6, r), (h_{78}, r), (h_{1234}, l)\},$$

where r designates right sibling and l left sibling.

In general,

$$PR_i = \{(h_j, r \text{ or } l) \mid h_j \text{ is the sibling of each node}$$

on the unique path from h_i to root}

The proof is an essential part of the *integrity token* that is generated for each object. In essence, the integrity token consists of the digest, the proof, and a time

stamp of the round. It also includes other information that will be needed over the long term, which will be briefly described in the next section.

Given the integrity token of an object, we can quickly compute the round hash value by following the path defined by the proof and performing the concatenation/hash operations as appropriate. For example, with the above PR_5 , the round hash value can be computed from $rh = H(h_{1234} \| ((h_5 \| h_6) \| h_{78}))$. Note that the length of such a path is logarithmic in the number of objects processed during a round and, hence, it is quite small relative to the number of objects.

We reiterate the process by compressing the ordered set of round hash values using one of the hash linking schemes such as Merkle's tree. The resulting value serves as a *witness value*. The granularity of this process can be set dynamically depending on the archive's schedule. Here, we assume that all the round hash values during a day are linked together to generate a witness value. This process can of course be repeated n times, making n -layers of hash linking trees. In our prototype, we stopped at $n = 2$, since the resulting witness values were quite small (less than 100KB a year).

Once determined, the witness values are stored in reliable read-only media. Our approach depends only on the correctness of the witness values, which is a very reasonable assumption given the total size of the witness values. Based on this assumption, we can achieve the following:

- Our scheme can detect an alteration to any digital object in the archive, malicious or otherwise.

- There is a cost-effective procedure that can periodically audit the contents of the archive to discover any alteration on any object within a short time after the alteration was made.
- Any party, independent of the archive, can audit any object in the archive and assert its integrity based on the witness values.
- No fully trusted third party is needed.

We note that a number of schemes can be used to correct errors once they are identified by our method, depending on the architecture of the archive. For a centralized archive with an isolated dark archive, a master copy can be retrieved to correct the corrupted object. For a federated or peer-to-peer distributed archive, a certified (by our scheme) remote replica can be used to replace the corrupted object using a mechanism that will depend on the technical details of the infrastructure.

We will later describe our ACE (Auditing Control Environment) system that accomplishes all the goals stated above. We next show how our approach can be adapted to deal with object transformations and updating the hash schemes used by the archive.

2.3.2 Updating Integrity Information

There are two cases in which the integrity information must be updated. The first case is when the archive decides to substitute a stronger hash function for one of the hash functions currently in use because of some recently discovered potential threats. The second is when the archive decides to apply certain transformations to some of the objects (because of the possibility of a format becoming outdated, for example).

There is an existing solution to deal with renewing the integrity information for the first case by re-registering each related object with the old integrity token attached to it (see, for example, [22]). Such a solution will ensure our ability to verify the integrity of the object since its ingestion into the archive as articulated in this earlier study. This process increases the size of the integrity token, but has no impact on the sizes of the other integrity components.

We now discuss how to renew the integrity information in the case when the object is subjected to a transformation. A possible solution would be to re-register the new object by concatenating the hashes of the old and the new form of the object and an identifier of the transformation, and use the resulting string as if it were the hash of an object to be registered. However, this scheme could be computationally demanding and too complicated to be of practical use. We assume that an archive has to preserve certain (sometimes all) versions of an object which can form an authenticity chain between the original version and the current version of the object. The chain may consist of the current version and the original version of the object. Since a transformation will lead to a new version of the object, and, hence, a new object with its own identifier (which could be the old identifier concatenated with the version number), it will participate in a hashing round to obtain its new cryptographic information using the same method as before. However, in this case, we will include the unique identifier of the previous version in the authenticity chain in the integrity token. Note that the integrity of an object should be verified before it is transformed into a new format to ensure its authenticity at this time of its history. The inclusion of the identifiers of previous versions in the integrity tokens will enable us to go through

the authenticity chain and establish the integrity of each version as well as the validity of the corresponding transformation.

2.4 Putting the Ideas Together – the ACE Tool

Making use of the ideas described in the previous section, we presented in [73], an early implementation of the ACE (Auditing Control Environment) prototype system. More recently, we released Version 1.0 of ACE, which includes some refinements to the earlier prototype. Here, we present a brief overview of the ACE architecture, illustrate its auditing processes, and report on its performance on a large scale production environment.

2.4.1 ACE Components

ACE consists of two major components: the first, called IMS (Integrity Management System), is a third-party service provider that generates the integrity tokens upon request from an archive. A single IMS can simultaneously serve multiple archives, including multiple nodes of a distributed archive. It also maintains the round hash values and generates the witness values. In ACE, the integrity tokens contain several pieces of information in addition to the proof and the time stamp (for example, the ID of the hash algorithm used, the version number of object, and last time the object was audited). Also, ACE links consecutive round hash values sequentially. The second major component is the Audit Manager (AM), which is local to an archive and functions as a bridging component between the IMS and the archive. In particular, the AM sends requests to the IMS to generate the integrity tokens for a number of objects,

and once received, the tokens are stored in a local registry. Figure 2.2 shows the overall ACE architecture of the general case of a distributed archive with dispersed nodes operating asynchronously.

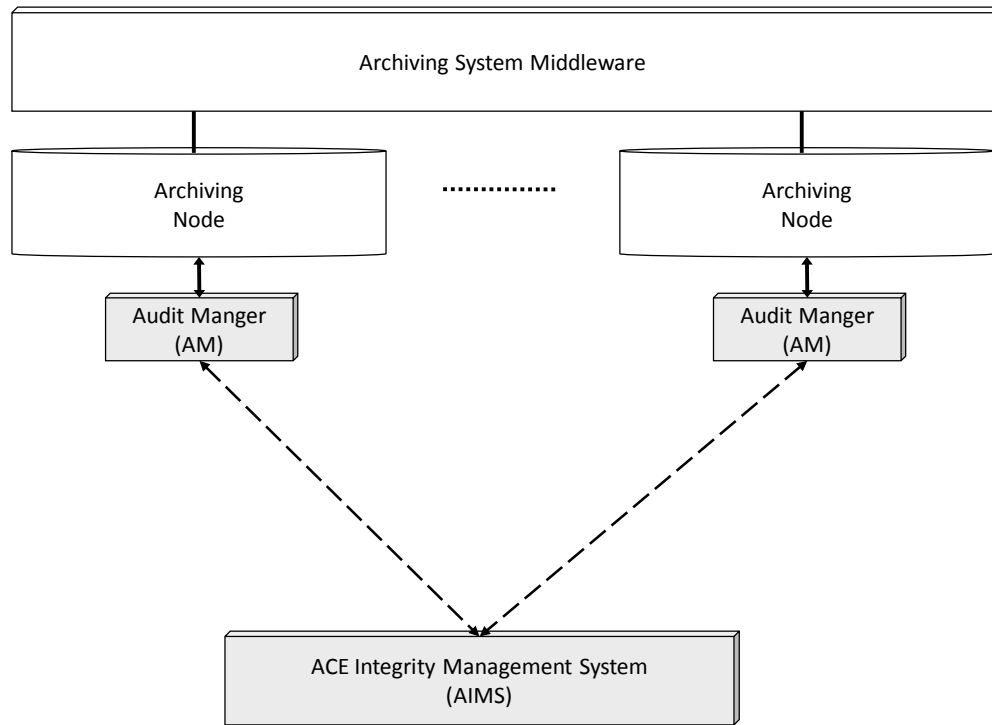


Figure 2.2: ACE Architecture

2.4.2 ACE Workflow

In this subsection, we discuss a typical workflow with ACE, which includes two major operations: registration and auditing.

2.4.2.1 Registration

For an object to be registered into ACE, the audit manager creates a registration request and submits it to the IMS. When the IMS issues an integrity token for the object, the audit manager stores it locally in a special registry for the archive (local node, if it is a distributed archive). In the meantime, the IMS runs a continuous series

of aggregation rounds. Each round closes either when the round receives the maximum number of registration requests, or when the maximum amount of time allocated for a round is reached, whichever comes first. These parameters are assigned by the IMS administrator. This round interval policy can be also overridden through a special object registration request, which forces the current round to immediately close and return an integrity token. Object registration requests received during a round are aggregated together along with a number of random values through the Merkel-tree hash-linking. The random values are added as necessary to ensure a minimum number of hash-linking participants in a round. The resulting round hash value is managed internally within the IMS, and an integrity token is issued to each AM who originally sent a registration request.

At the end of each day, the IMS constructs a witness value using the hash round values of the day, sends it to the participating archives, and stores it in a reliable medium (the current ACE implementation publishes witnesses to a Google group, and stores the value on a CD-ROM). These witnesses are cryptographically dependent on round hash values, which are themselves cryptographically linked to the hashes of the objects registered during that day.

2.4.2.2 Audit Process

ACE currently performs periodic auditing on the archive's objects following the policy set by the manager of the archive. The policy can be set at an object or collection level. For example, the policy for a certain collection could involve auditing all the objects in the collection every three months, while the policy set for another collection could be to audit all the objects in that collection every six months.

A default policy will be set during registration time unless the archive manager sets it differently. Moreover, the auditing process can be invoked by the archive manager at any time on any object.

When applied to a specific object O , the auditing process consists of the following steps:

1. The audit manager computes the hash value of O and retrieves its integrity token.
2. Making use of the computed hash value of O and the proof contained in the integrity token, the audit manager computes the round hash value.
3. Making use of the round time stamp contained in the integrity token, the audit manager requests the corresponding round hash value from the IMS.
4. The audit manager successfully terminates the auditing process if the computed hash value in Step 1 is equal to the hash value stored in the integrity token, and the two round hash values computed in Steps 2 and 3 are equal. Otherwise, it sends an error alert to the archive manager.

It is clear that if the two hash values, as well as the two round hash values, computed through the auditing process are equal, then the object and the integrity token are intact with a very high probability. A more elaborate process, which will happen infrequently, will involve the witness values as follows. The audit manager requests from the IMS the proof for the round hash value. On receiving this request, the IMS aggregates all the round hash values for the day to determine the proof, and returns the proof to the audit manager. Making use of the proof, the AM computes the corresponding witness value of the day and compares it to the value stored on the

read-only media. This elaborate process will ensure the trustworthiness of the IMS. A failure of this process will automatically invalidate the object under the auditing process.

We note that the same process can be applied by a party independent of the archive to verify the integrity of an object. The independent party will request the integrity token from the archive, and then the round hash value and its proof from the IMS. Making use of this information, she/he can quickly compute the witness value of the day on which the object was registered into ACE. She/he can then compare it with the corresponding witness value stored in the read-only media. Any alterations introduced by the archive or the IMS will be detected with very high probability.

2.4.3 ACE Preliminary Performance Evaluation

ACE Version 1.0 has been deployed against a variety of collections managed by the Chronopolis archiving environment. Chronopolis is collaboration between the San Diego Supercomputer Center (SDSC) / the University of California, San Diego (UCSD) Libraries, National Center of Atmospheric Research (NCAR), and the University of Maryland (UMD), which involves a distributed archiving architecture with three geographic nodes at SDSC, NCAR, and UMD. Chronopolis is currently managing substantial collections from NDIIPP partners, including the California Digital Library (CDL) Web-at-Risk collection, the InterUniversity Consortium for Political and Social Research (ICPSR) collection, and collections from the Scripps Institution of Oceanography–Geological Data Center (SIO–GDC), and North Carolina Geospatial Data Archiving Project (NC State). ACE has been operational

during the past year within the Chronopolis environment. The current default ACE auditing policy is to audit files at the University of Maryland every 30 days. Table 2.1 illustrates the performance of a single audit manager on the collections audited at UMD, amounting to approximately 6 million files. A large fraction of the time is spent on accessing the collections across the network.

Table 2.1: ACE Performance

<i>Collection</i>	<i>No. of Files</i>	<i>Size (GB)</i>	<i>Audit Time</i>
CDL	46,762	4,291	20h 32m
SIO-GDC	197,718	815	6h 49m
ICPSR	4,830,625	6,957	122h 48m
NS State	608,424	5,465	32h 14m

During the audit period on the CDL collection, a single audit manager was able to run at the rate of about 60MB per second on average, almost fully utilizing the file transfer bandwidth available. For the other collections, where there were more small files, the audit speed was further limited by the overhead accessing each file. For example, on the ICPSR collection, the audit manager ran at the rate of 13MB per second, having to open up each of about 4.8 million files. These results indicate that the actual time spent by an audit manager to perform the core audit process is negligible. It is small enough to be effectively hidden by the unavoidable overhead for accessing the collections. We note that multiple audit managers can be run concurrently on different collections to increase the performance almost linearly as necessary.

2.5 Conclusion

In this chapter, we presented a new methodology to address the integrity of long-term archives using rigorous cryptographic techniques. Our approach depends only on the use of hash functions and linking schemes, and is independent of an external infrastructure such as PKI. The computational requirements of our approach are minimal and the overall solution can be implemented on any archive architecture. We built ACE as a complete prototype that executes this strategy and showed its effectiveness on large collections in Chronopolis. More details about ACE can be found in [73].

Chapter 3

Full-text Search Approach for Time-evolving Data

A number of emerging large scale applications such as web archiving and time-stamped web objects generated through information feeds involve time-evolving objects that can be most effectively explored through search within a temporal context. We develop in this chapter a new approach to handle the temporal text search of a time evolving collection of documents. Our approach introduces both a new indexing organization that substantially limits the search space and an effective methodology for computing the temporally anchored relevance scores. Moreover, we develop an analytical model that can be used to determine the temporal granularity of the indexing organization which minimizes the total number of postings examined during query evaluation. Our approach is validated through extensive empirical results generated using two very different and significant datasets.

3.1 Overview

The initial driving application behind this work is the temporal text search over an archived collection of time-evolving web contents. Currently, many organizations are building web archives that contain collections of temporal snapshots of web pages that have been captured by a crawler at a frequency that typically depends on the dynamic nature of the pages. For example, the Internet Archive [75] has been capturing significant snapshots of the internet over 15 years. The Internet Archive

currently holds over 4.5 petabytes of data and is growing at the rate of about 100 terabytes per month as of March, 2009 [49]. Other major web archiving efforts include the Minerva project by the Library of Congress [54], UK Web Archiving Consortium [78], the National Library of Australia's Pandora project [62], and the Web-at-Risk led by the California Digital Library [86]. Given the critical role of the internet as the main communication and publication medium in our information-based society, and the ephemeral nature of the web, it is expected that web archiving efforts will dramatically grow in the future. Other similar collections include multi-versioned documents generated through collaborative environments and time-stamped objects generated through various information feeds. It is clear that the exploration of such continuously growing archives can be substantially simplified through text search within a temporal context.

We explore in this chapter a new approach to carry out a temporal text search over a collection of documents that evolve over time. Specifically, given a query that includes a text and a time span, the goal is to return a ranked list of temporally relevant documents. That is, the returned documents must have been valid during the query time span and the relevance scores are computed relative to the state of the collection as it existed during the query time span. The importance of temporal relevance can be illustrated with the example of searching for "September 11" during the month of "May 2001" which, if temporally unconstrained, will return an overwhelming number of irrelevant results.

We present a new methodology to address this problem and outline the necessary core algorithms to support it. More specifically, our main contributions include the following:

- A new indexing organization that substantially limits the search space while allowing the efficient and scalable computation of the relevance scores relative to the state of the collection as it existed during the query time span.
- An analytical model that can be used to determine the temporal granularity of our overall indexing organization which minimizes the total number of postings examined during query evaluation.
- Extensive empirical evaluation of the overall scheme in terms of its storage requirement, query evaluation, and ranking of search results using two rich datasets of sizes 2.8TB (uncompressed) and 5.6TB (gzip-compressed) respectively.

The rest of the chapter is organized as follows. The next section provides a summary of related work, while Section 3.3 provides a formal description of our overall model. We introduce our approach and describe our indexing structure, an analytical model for capturing the tradeoff between index space and query evaluation performance, and the computation of the relevance scores in Section 3.4. Section 3.5 describes our two major datasets used for evaluation, and provides a summary of our empirical evaluation results. We conclude in Section 3.6.

3.2 Related Work

For the most part, text retrieval has been concerned with the present state of the document collection. The search problem for multi-version documents involves documents that change over time, and the versions of each document are maintained. In this case, a query will in general include, in addition to a set of terms, a temporal component (*temporally-anchored query*), and hence the search outcome is a ranked list of document versions satisfying the query temporal constraints.

A common approach to handle temporally-anchored queries is to rely on a post-process filtering. In this approach, a regular search is processed first ignoring the temporal component. The search results are then filtered according to the temporal constraints. This approach suffers from two major drawbacks. The first is that the search space is the same regardless of the temporal constraints and hence many documents may need to be filtered out. The second major drawback is more fundamental – the query-document relevancy scores are determined based on the entire collection and not on the state of the collection as it existed during the query time span.

We are not aware of any prior work that incorporates the temporal dimension in an integral way for full-text search including temporal scoring. In fact, most of the published papers seem to revolve around the above common approach and focus on improving the search performance by reducing the search space using a number of data structures. We next summarize the most relevant prior work.

Anick and Flynn [1] describe a “help-desk” system that supports historical queries. In their system, upon a request for a past version, starting with the most

current version of the object, the reverse sequence of delta changes preceding the object are applied back until the view of the request version is reconstructed. Although access costs for the most recent versions are relatively optimized, the cost increases as the versions move farther into the past. The help-desk system reduces the overall space requirement for storing documents, and also minimizes the search space for the most recent version.

Nørvåg introduced a multi-version document database system, V2 [58] and ITTX [57], and also DyST [59] with Nybø. In essence, V2 takes the search-and-then-filter approach discussed earlier, but the filtering can be enhanced by optionally having a supplementary data structure that maps a document version ID to its corresponding time period. ITTX reduces the search space by decreasing the index size of V2 – It replaces term / version mappings of postings in V2 with term / version-range mappings. However, for given query terms, the entire postings lists still need to be examined, regardless of the query time span. To alleviate this problem, DyST [59] improves ITTX by employing an additional temporal index. When a postings list reaches a certain size, a Time Index+ [41], which is a temporal B⁺-tree, is created and the contents of the postings list are migrated to the Time Index+. A major shortcoming in their approach, however, is that neither ITTX nor DyST considers the relevance scoring aspect of the search results.

More recently, Berberich et. al. [6] presented a scheme called Time Machine to handle point queries over temporally versioned document collections. A standard vocabulary is constructed such that, for each term, a postings list of (document ID, score, time-frame) is maintained. Since these lists can grow extremely large, they

introduce two techniques – temporal coalescing and sublist materialization. Temporal coalescing reduces the size of each postings list by merging a sequence of postings that simultaneously have the same document ID and “similar scores”. This is the index space reduction technique similar to the one used in ITTX, but Time Machine differs from ITTX in that it factors in “scores” as one of the merge criteria (In ITTX, a sequence of postings with the same document ID are merged regardless of scores). Sublist materialization divides each postings list into several sublists according to some time intervals depending on each list separately. Although the total index size increases with sublist materialization, the effective search space for a given query can be reduced, since the searches are localized to corresponding sublists. While their scheme allows the relevancy scoring of search results, it has a number of limitations. First, their scheme assumes that scores are comparable to one another regardless of validity time information in the postings. This implies that the scores are computed within the context of the entire history of the collection, regardless of the time constraints of the queries. Also, the index is built based on the pre-computed score information for each posting. This implies that the index is bounded to a specific scoring scheme, making it difficult to adopt another scoring scheme later.

3.3 Model

Following the standard information retrieval terminology, we refer to our objects generically as *documents*, which in our case evolve over time. Each version of a document is identified by the document ID and a *validity time interval* $[t_i, t_j)$, which starts from the time t_i that the version was first seen until the time t_j a different

version is detected or the document ceases to exist. For example, in web archiving, a document version is seen or detected at the time the corresponding page is crawled. A document version in this case refers to a web object together with its validity time interval. We define a document version to be *live* at time t if its validity time interval contains t . In our context, a collection D consists of *document versions over discrete elementary time steps*, that is, all time values defining validity time intervals are non-negative integers and a document version is modified, created, or deleted at only one of these discrete time steps. The *state* of the collection during a time interval $[t_u, t_v]$, denoted by $S[t_u, t_v]$, consists of all the document versions in D whose validity time intervals have a non-empty intersection with $[t_u, t_v]$. Figure 3.1 illustrates an example consisting of seven documents and corresponding document versions over 9 time steps. An arrow head indicates the endpoint of a validity time interval.

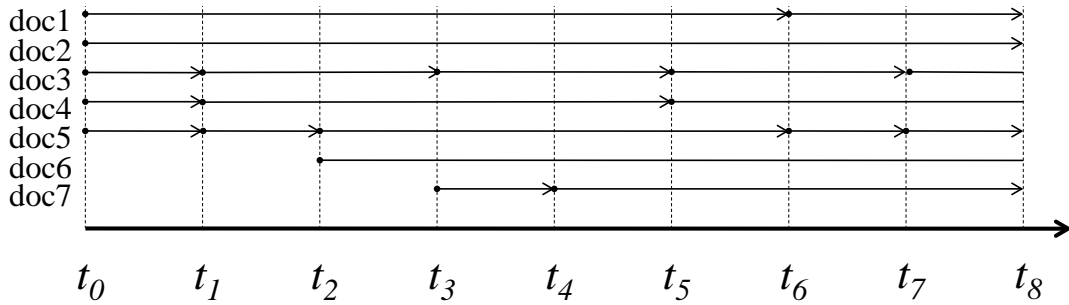


Figure 3.1: Document Versions with Validity Time Intervals

We assume a query model that consists of a set of terms, possibly connected by Boolean operators, and a temporal specification defined by the *query time span* $[q_s, q_f]$. Such queries are called *temporally-anchored* queries. A query reduces to a *point query* when $q_s=q_f$. The result of the search is a ranked set of document versions that

have validity time intervals overlapping with the query time span. Document relevance is determined based on the state of the archive during the query time span. More specifically, relevance is determined by computing similarity scores between the query and the document versions in $S[q_s, q_t]$ using statistics over $S[q_s, q_f]$ as needed. For our experimental evaluation, we use two types of scoring functions, one based on Okapi BM25 [69] and the second based on the KL-divergence smoothed by Dirichlet priors [88]. Hence a number of statistics pertaining to the state $S[q_s, q_f]$ will have to be computed or approximated to determine the similarity scores corresponding to a query whose query time span is $[q_s, q_f]$. In our evaluation, our data model does not take into consideration the linking relationships among documents, and therefore, we do not consider link-based scoring schemes, such as PageRank [7] or HITS [40]. We note that a link-based score of a document version is dependent only on the linking structure of the collection when the document version was live, and hence that score is independent of the query time span. Therefore it is possible to include it in each posting and combine it with the more traditional document scoring techniques.

3.4 Our Approach

In a standard inverted index, each term is associated with a number of postings. Each posting consists of the ID of the document that contains the term and some associated payload necessary for computing query-document scores. In the simplest case, the payload is the term frequency, but may contain additional information such as term positions (e.g., for proximity queries). We denote a posting as $(d_i p)$. To

support temporally-anchored queries, postings must be augmented with temporal information, which will be denoted as $(d_i [t_m, t_n] p)$, where $[t_m, t_n]$ is the validity time interval of the document version. There are two straightforward ways to extend the standard inverted index strategy to handle temporally anchored queries, each of which has a number of significant limitations.

The first consists of building the inverted index of all the document versions in the collection D . There are at least four significant problems with this solution. First, postings lists will grow unbounded and present an efficiency bottleneck since query evaluation algorithms must traverse the postings to score documents. Second, a large fraction of the document versions will have to be filtered out when computing the similarity scores since their validity time intervals may not overlap with the query time span. Such a process is the basis of the prior work mentioned earlier such as V2, ITTX, DyST, and Time Machine. For instance, Time Machine adapted this approach to point queries using postings that contain the scores and introduced a number of heuristics to improve query performance as they relate to point queries. The third significant problem with this approach is the fact that no fast scheme for computing the query-document scores based on the appropriate state of the collection seems to be possible. Finally, as the collection grows, the indexer will face an incremental update problem on postings lists, which complicates document ingestion and processing, which may be interleaved with live querying.

The second straightforward approach consists of building, for each discrete time step, a separate inverted index for all the document versions that are live at that time step. A temporally-anchored query can then immediately target the appropriate set of

inverted indexes to compute the query-document scores, assuming global statistics about the state $S[q_s, q_r]$ can be evaluated quickly. However, such an approach will in general incur substantial index storage overhead since a long-lived document version will appear over many time steps causing the postings of all of its terms to be replicated many times. In addition, this causes many repeated computations of the score of a document version and a query, one for each time step at which the document version is alive.

Our proposed solution allows a more general framework than either of the methods described above. We establish a number of time windows, denoted as $T_1, T_2, \dots T_k$. Each time window will contain postings of document versions whose validity time intervals overlap with or are strictly contained in the time window. That is, for each T_i , we construct an inverted index corresponding to the document versions in $S[T_i]$. Search can thus be localized to one or more appropriate time windows, saving the retrieval algorithm from having to process most postings. Incremental updating as the collection grows is dramatically simplified since only the most recent time window is affected. The indexing of new document versions will affect only the most recent time window, and once a time window is “closed off” corresponding indexing structures become immutable.

Within a particular time window, the postings associated with each term might look something like this: $(d_1 [t_1, t_2) p) (d_1 [t_2, t_3) p) (d_1 [t_3, t_4) p) \dots$. We note that such a representation can be compressed substantially since each document ID may occur multiple times on the same list, and there is no need to store time stamps explicitly

since interval widths can be reconstructed from the beginning of the time window. For the rest of this chapter, we assume this explicit representation for our postings.

The overall structure of our proposed solution is illustrated in Figure 3.2.

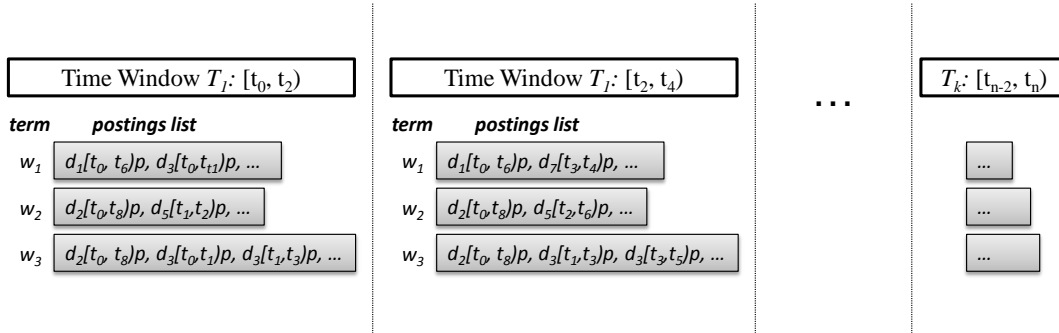


Figure 3.2: Structure of Our Proposed Temporally-augmented Inverted Index.

For our approach to work, we have to establish the existence of appropriate time windows that enable fast query evaluation using compact indexing. To move toward this goal, we first present an analytical model that shows the existence of time windows that achieve an optimal tradeoff between index space and query evaluation time. We then describe an efficient approach to determine the necessary statistics required for computing the temporal query-document version scores, which are evaluated relative to the state of the collection over the time span specified by the query. The claims made in the next two sections will be evaluated through empirical results presented in Section 3.5.

3.4.1 Analytical Model

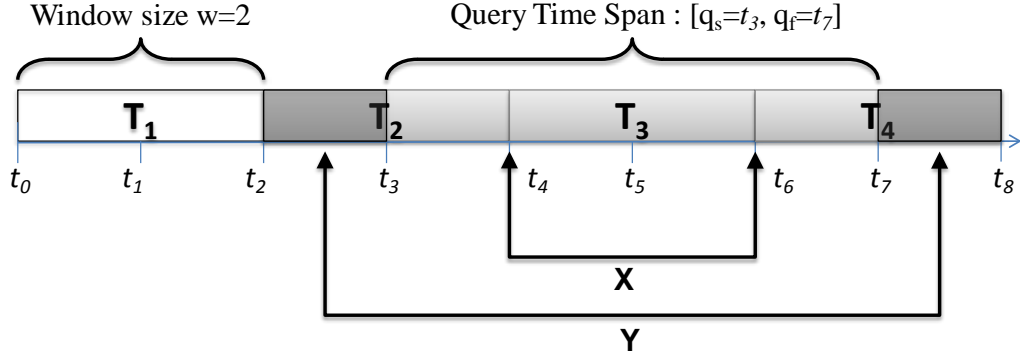
The determination of appropriate time windows involves a tension between two competing goals. Large time windows result in less index space, since fewer document versions will live across multiple windows, but at the cost of longer postings lists and the possibility that many of the postings will have to be filtered out

during query evaluation (because their validity time intervals do not overlap with the query time span). On the other hand, smaller time windows mean that more document versions will span possibly many consecutive time windows, resulting in many duplicate postings across the time windows. Based on this dichotomy, we formulate an optimization problem and derive an analytical solution that achieves an optimal tradeoff between these competing goals.

We start by introducing some notation. Let t_1, t_2, \dots, t_n be the elementary time steps over which documents in our collection evolve. To simplify our analysis, we assume that the time steps are equally spaced and that the time windows all have the same size, say z . We note, however, that our approach is more general, and the analysis can be carried out incrementally as the collection grows. Without loss of generality, we also assume that $k=n/z$ is an integer representing the number of time windows T_1, T_2, \dots, T_k . To handle a query with a time span $[q_s, q_f]$, we need to consider the postings associated with the consecutive time windows, say T_i through T_j , which overlap with $[q_s, q_f]$. These postings include two superfluous types of postings that are not needed for processing the query. The first type pertains to the duplicate document versions that appear in T_{i+1} through T_j . The second pertains to those document versions whose validity time intervals do not overlap with $[q_s, q_f]$. We aim at determining time windows that minimize the total number of these two types of document versions. More formally, we define our optimization problem as follows.

Let X be the total number of duplicate document versions that appear in T_i through T_j , that is, the total number of boundary crossings of validity time intervals between any pair of consecutive time windows. Let Y be the total number of

document versions that appear in T_i through T_j whose validity time intervals do not overlap with $[q_s, q_f]$. Figure 3.3 illustrates X and Y for a simple example. Note that X decreases as the time window size increases while Y decreases as the time window size decreases. Our goal is to come up with a value of z that minimizes the sum $X+Y$.



- X:** The number of document versions crossing boundaries of T_2 — T_3 or T_3 — T_4
- Y:** The number of document versions appearing within $[t_2, t_8]$ but not overlapping with $[t_3, t_7]$

Figure 3.3: Illustration of Parameters X and Y

We develop an analytical solution assuming that the query time span $[q_s, q_f]$ is selected *randomly* from among all possible time spans. In Appendix A, we prove the following results concerning the expected value $E[X]$ of X and the expected value $E[Y]$ of Y .

Let δ_i be the number of document versions whose validity time intervals contain t_i and let δ be the average of all the δ_i 's. Then $E[X]$ can be shown to be given by the following expression:

$$E[X] = \frac{2z^2}{n^2} \sum_{i=1}^{k-1} i \cdot (k-i) \delta_{iz}$$

We can substitute δ for the individual δ_i 's to approximate $E[X]$ as follows.

$$E[X] \approx \frac{(k-1)^2}{3k} \delta \quad (1)$$

For sufficiently large k , $E[X]$ can be further approximated by:

$$E[X] \approx \frac{k}{3} \delta = \frac{n}{3z} \delta$$

This implies that $E[X]$ is linearly proportional to the number of time windows or inversely proportional to the window size. Note that the two straightforward approaches mentioned earlier correspond to $k=1$ (a global index for all the time steps) and $k=n$ (an inverted index for each time step). Clearly, for $k=1$, there are no duplicate document versions, as predicted by the expression of $E[X]$ for small k . For $k=n$, the number of duplicate document versions can be proportional to the number of time windows since long lived documents may cross a fraction of the time steps.

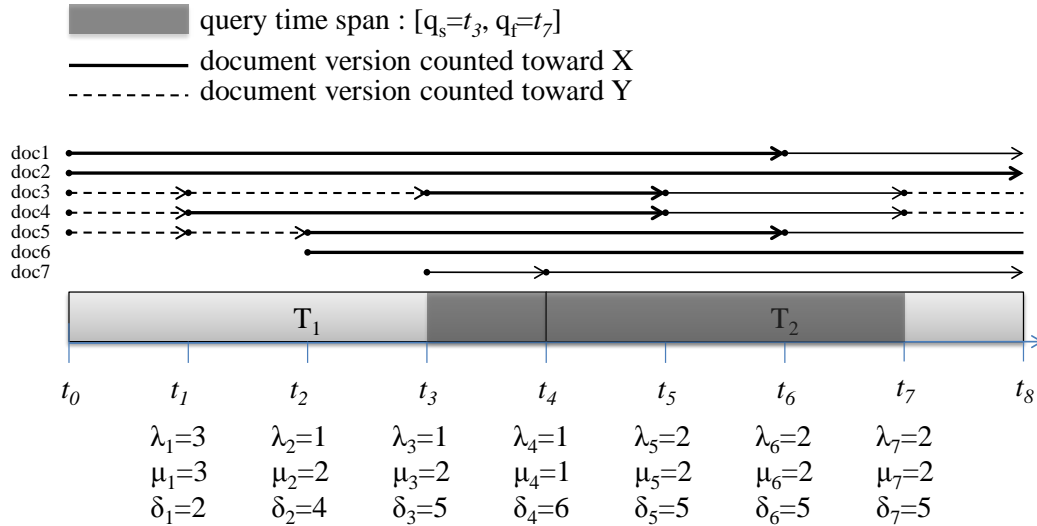


Figure 3.4: Illustration of Values of λ_i , μ_i and δ_i

To estimate $E[Y]$ we let λ_i be the number of document versions whose validity time interval end at t_i (that is, a new version is created or document is deleted at t_i)

and let μ_i be the number of document versions whose validity time interval start at t_i . Figure 3.4 illustrates these parameters for an example consisting of seven documents.

Using the average value λ of all the λ_i 's and the average value μ of all the values μ_i 's, $E[Y]$ can be shown to be given by the following expression:

$$E[Y] \approx \frac{\lambda + \mu}{6} \left(3z - \frac{z}{k} + \frac{1}{k} - 3 \right) \quad (2)$$

For sufficiently large k , $E[Y]$ can be further approximated by:

$$E[Y] \approx \frac{\lambda + \mu}{2} \cdot z$$

That is, $E[Y]$ is linearly proportional to the size of the time window. This expression can be justified intuitively since the larger the time window size, the more document versions are irrelevant to a random query. For a single time window covering all the time steps, $z=n$ ($k=1$) and hence many postings will have to be filtered out for a random query time span. On the other hand, for $z=1$ (or equivalently, $k=n$), no postings have to be filtered out, consistent with the expression derived for $E[Y]$.

Given the expressions of $E[X]$ and $E[Y]$, it is easy to minimize our objective function $f(z)=X+Y$. One can set the derivative of $f(z)$ to 0 to obtain the value of z that minimizes f . For sufficiently large k , this value is given by:

$$z = \sqrt{\frac{2\delta \cdot n}{3(\lambda + \mu)}}$$

For concreteness, we apply these formulas using parameter values derived from two large datasets to be introduced in Section 3.5. In Figure 3.5, based on the approximations given in (1) and (2), we plot the graphs of $E[X]$ and $E[Y]$ and the

graph of $E[X]+E[Y]$ using the statistics ($\mu=200,223$, $\lambda=175,190$ and $\delta=362,299$) derived from the Wikipedia dataset to be introduced in Section 3.5.1. We can see $f(z)$ is minimized when z is around 7. Similarly, Figure 3.6 plots the graphs corresponding to the Library of Congress dataset ($\mu=2,071,661$, $\lambda=1,197,155$ and $\delta=10,828,027$) also to be introduced in Section 3.5.1, where the time window size z that minimizes $f(z)$ is found around 8.

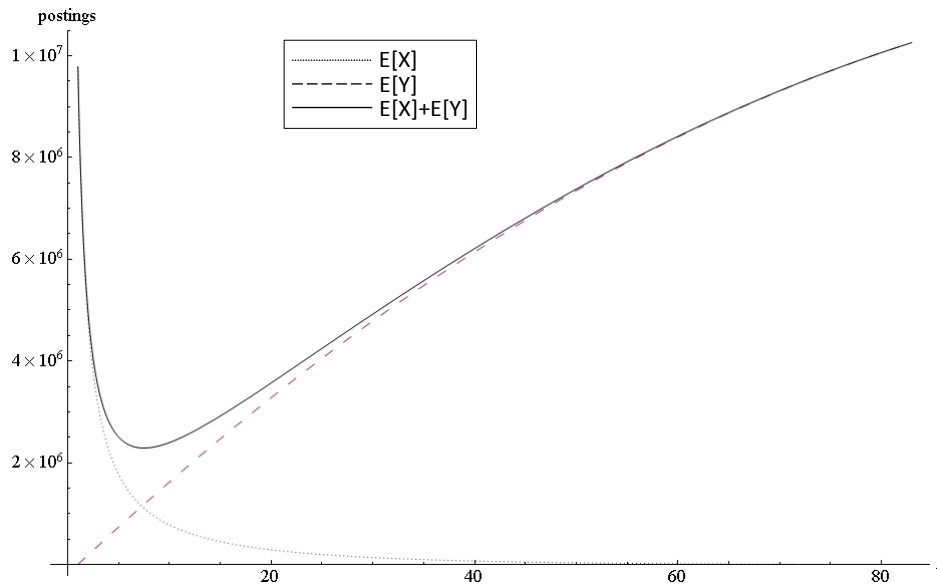


Figure 3.5: Wikipedia: $E[X]$, $E[Y]$ and $E[X]+E[Y]$

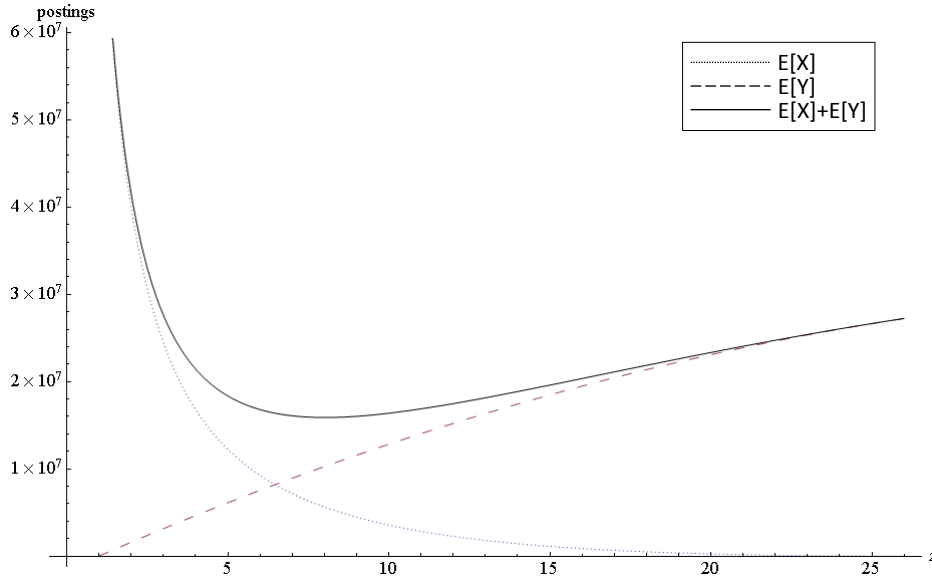


Figure 3.6: Library of Congress: $E[X]$, $E[Y]$, and $E[X]+E[Y]$

We end this section with two comments. The first is that the same type of analysis can be carried out to determine the average number of postings to be examined for handling any fixed query text assuming a randomly selected query time span. The second is that the empirical results to be described in Section 3.5.4 strongly support the analytical results reported here.

3.4.2 Temporal Relevance and Scoring

We describe our extensions of retrieval algorithms for scoring temporally-anchored queries. Figure 3.7 illustrates the interaction between temporally-anchored queries and the indexing structures introduced earlier in this section. Our framework will have multiple structures, each associated with a time window. In addition, each time window will contain statistics such as document frequencies, document lengths, and other metadata, pertaining to the collection state over that time window. The figure shows the two possible query scenarios: a query (Q1) with an associated time

span that falls within a time window completely, and a query (Q2) with an associated time span that covers more than one time window. A point query is obviously a special case of the query time span reduced to a point within a time window.

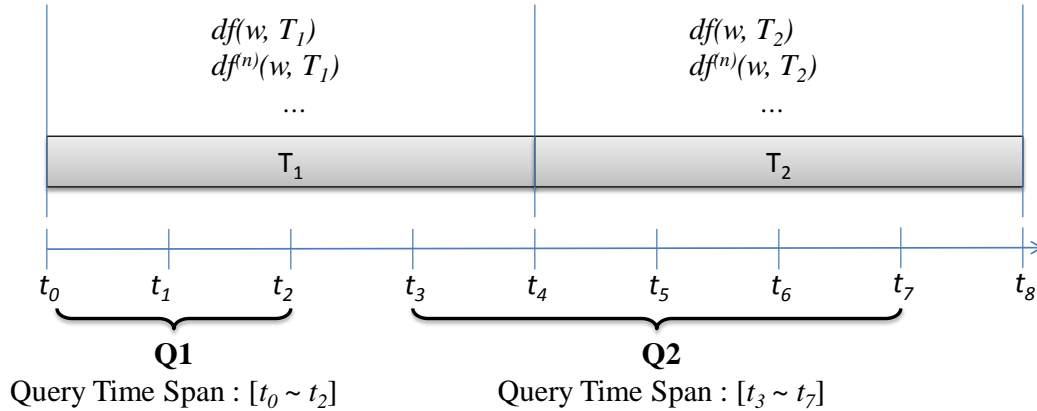


Figure 3.7: Temporally-anchored Queries

Nearly all retrieval algorithms, from simple vector space models to modern language modeling techniques, rely on three types of statistics: local statistics for term incidence in documents (in the simplest case, term frequency), global term statistics (e.g., document frequency, or df), and collection statistics (e.g. the total number of documents as used in Okapi BM25, or the total number of terms as used in some language models). Local statistics are contained directly in the postings, while global term statistics are usually stored in the head nodes of the postings lists. Collection statistics are typically stored separately. In addition to the above statistics, many retrieval models also require information on document lengths to be used for length normalization or smoothing.

In our framework, we will maintain statistics over each time window T_i which will enable us, not only to compute global term and collections statistics over T_i , but also to compute statistics over a consecutive set of time windows that include T_i . This

is needed since a query time span may overlap with several consecutive time windows. To accomplish this, we maintain, for each such statistic, two values – one computed over *all* document versions in T_i , and the other computed over the *newly created* document versions in T_i . *Newly created* document versions refer to those whose validity time intervals have their left endpoints properly inside T_i .

Let us consider for example the document frequency $df(w, T_i)$ of a term w statistic over the time window T_i . We maintain in addition to this value another statistic, namely the document frequency $df^{(n)}(w, T_i)$ of the term w over the *newly created document versions* within the time window T_i . Given a query time span $[q_s, q_e]$, we approximate the term document frequency over that query time span by combining the term document frequencies in time windows i through j overlapping with $[q_s, q_e]$ as follows.

$$df(w) \Big|_{S[q_s, q_e]} \approx df(w, T_i) + \sum_{l=i+1}^j df^{(n)}(w, T_l)$$

Another example is to approximate the average document length L_{ave} over the query time span using the following formula.

$$L_{ave} \Big|_{S[q_s, q_e]} \approx \frac{N(T_i) \cdot L_{ave}(T_i) + \sum_{l=i+1}^j N^{(n)}(T_l) \cdot L_{ave}^{(n)}(T_l)}{N(T_i) + \sum_{l=i+1}^j N^{(n)}(T_l)},$$

where $N(T_i)$ is the number of document versions in time window T_i . Similarly, all the other global term statistics or collection statistics can be approximated in the same way.

In the next section, we will present empirical evaluations of the ranked search results that come from using the above scheme for approximating global statistics and

collection statistics. These results show that, as long as the time windows are not too large, the ranked search results are very close to those produced by using the exact statistics for each of the scoring schemes used in Okapi BM25 and KL Divergence with Dirichlet priors.

3.5 Empirical Evaluation

In this section we provide empirical evaluation of our approach on two significant datasets to be introduced in the next section. We use the following performance metrics:

Total number of postings in the indexing structures built for all the time windows. This metric captures the overall index space requirement. As observed earlier, the larger the time window, the less the number of duplicate document versions that appear in consecutive time windows, and hence the smaller the overall index space.

Average number of postings examined for a typical query and a random query time span, where a typical query load is defined for each dataset. This metric clearly impacts the query evaluation time. The dependence of this metric on the time window size is subject to conflicting requirements that are similar to those described in Section 3.4.1.

Relative recall and Kendall's τ for the top 100 ranked search results when compared to the list generated using exact global and collection statistics for Okapi BM25 and KL Divergence with Dirichlet priors. These two metrics clearly favor smaller window sizes, with the smallest window size resulting in exact global and collection statistics.

We will look at each metric separately to try to better understand the nature of its dependence on the time windows, and then conclude with empirically best time window sizes that are close to the predicted values of our model developed in Section 3.4.1.

We start by describing the two datasets used, followed by an outline of our methodology for running the empirical evaluation process. The results corresponding to each of the metrics above will be described separately in the following sections.

3.5.1 Datasets Used

We use two large-scale datasets – the English Wikipedia revision history from 2001 to 2007, and a dataset given to us by the Library of Congress involving crawls of selected news and government websites. The English Wikipedia revision history is a publicly available XML dump created on January 3, 2008. It contains about two million articles (documents), each of which has one or more revisions (document versions) during the period. We pre-process the Wikipedia dataset and organize it into 83 monthly snapshots between February 2001 and December 2007. Included in each snapshot is the most recent revision of each article at the end of the month. The Library of Congress collection was compiled by the Internet Archive involving crawls to selected news and government websites during 2003 and 2004. The next table highlights some of the main features of each collection.

Table 3.1: Datasets

	Wikipedia	Library of Congress Collection
Original Data	English Wikipedia XML dump created on Jan. 3 2008	News, Government, and Other Sites
Extracted Data	83 monthly snapshots between Feb. 2001 ~ Dec. 2007	26 weekly snapshots between Jul. 2004 ~ Dec. 2004
Included in Each Snapshot	Most recent revision of each article as of the end of the month.	Most recent version of each crawled text web page as of the end of the week.
Number of Documents	2,077,745	21,455,523
Number of Document Versions	16,618,497	53,863,195
Average Number of Versions per Document	8.00	2.51
Average Lifespan of Document	22.47 months	15.07 weeks
Average Lifespan of Document Version	2.81 months	6.13 weeks

3.5.2 Evaluation Methodology

A straightforward way to conduct the empirical evaluation amounts to building the inverted indexes for all possible time windows for each of the two datasets. Given the sizes of the datasets and the numbers of the time steps, this approach is computationally prohibitive. However, we can generate the same empirical results using the following substantially more efficient strategy. We build an inverted index for each elementary time step separately (i.e., time window size is equal to 1) and collect two separate sets of statistics as required by our approach. For example, for each term w and each elementary time step t_i , we compute $df(w, t_i)$ and $df^{(n)}(w, t_i)$ representing respectively the document frequency of w over all the document versions that are live at t_i and over all the newly created document versions

at t_i . We can then use this information to generate the experimental results for an arbitrary time window size as follows.

3.5.2.1 Total Number of Postings

For a target time window size z , we consider a series of virtual time windows $\{VT_1:[t_0 \sim t_z], VT_2:[t_z, t_{2z}], \dots\}$, each consisting of z time steps as shown in Figure 3.8. For each virtual time window, we compute the required statistics, by carefully combining the statistics of the elementary time steps that fall within the virtual time window. For example, the document frequency $df^{(n)}(w, VT_l)$ of newly created document versions for virtual time window VT_l in Figure 3.8 can be computed using

the formula $df^{(n)}(w, VT_l) = \sum_{i=(l-1)s}^{ls-1} df^{(n)}(w, t_i)$, and the document frequency $df(w, VT_k)$ can

be obtained by using the formula $df(w, VT_l) = df(w, t_{(l-1)s}) + \sum_{i=(l-1)s+1}^{ls-1} df^{(n)}(w, t_i)$.

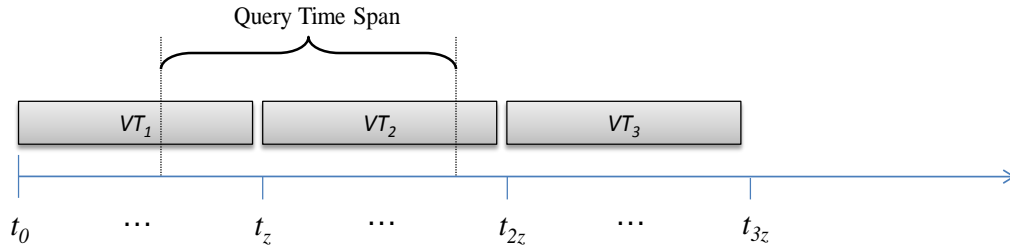


Figure 3.8: Virtual Time Windows

The total number of postings in each virtual time window VT_i is simply the sum

$\sum_w df(w, VT_i)$ and the overall total number is given by $\sum_i \sum_w df(w, VT_i)$.

3.5.2.2 Average Number of Postings for a Typical Query

Given a query $q[t_s, t_f]$ and a target window size z , we can compute the number of postings that have to be examined using the derived set of statistics for virtual windows as follows: $\sum_{w \in q} \sum_{l=i}^j \{df(w, VT_l)\}$, where virtual time windows VT_i through VT_j overlap with $[t_s, t_e]$ and the outer sum is over all the query terms. Note that this sum captures all the postings lists that need to be examined when handling the corresponding query.

3.5.2.3 Ranked Search Results

Given a query $q[t_s, t_e]$ and a target time window size z , our goal is to determine the top 100 ranked search results obtained by Okapi BM25 and KL Divergence using the statistics associated with the virtual time windows that overlap $[t_s, t_e]$. We use the postings lists associated with each query term at each elementary time step in the interval $[t_s, t_e]$ but with the global and collection statistics associated with the virtual time window containing the time step. The corresponding document version IDs and scores are sorted and merged, and the top 100 document version IDs are those that would have been returned had we generated the postings lists for the time window size z (obviously duplicate document versions are eliminated).

3.5.3 Empirical Results on Total Number of Postings

For each dataset, we consider all possible time window sizes that result in different numbers of time windows. For time window sizes that lead to the same number of time windows, we consider the largest such time window. Table 3.2 shows

the time window sizes used in our tests, and the corresponding numbers of time windows for each dataset.

Table 3.2: Time Window Sizes

	Time Window Size (Number of Time Windows)
Wikipedia	1 (83), 2 (42), 3 (28), 4 (21), 5 (17), 6 (14), 7 (12), 8 (11), 9 (10), 10 (9), 11 (8), 12 (7), 14 (6), 17 (5), 21 (4), 28 (3), 42 (2), 83 (1)
Library of Congress	1 (26), 2 (13), 3 (9), 4 (7), 5 (6), 6 (5), 7 (4), 9 (3), 13 (2), 26 (1)

For each time window size, we sum up the numbers of postings in all the corresponding time windows. The results from the two datasets are illustrated in Figures 3.9 and 3.10.

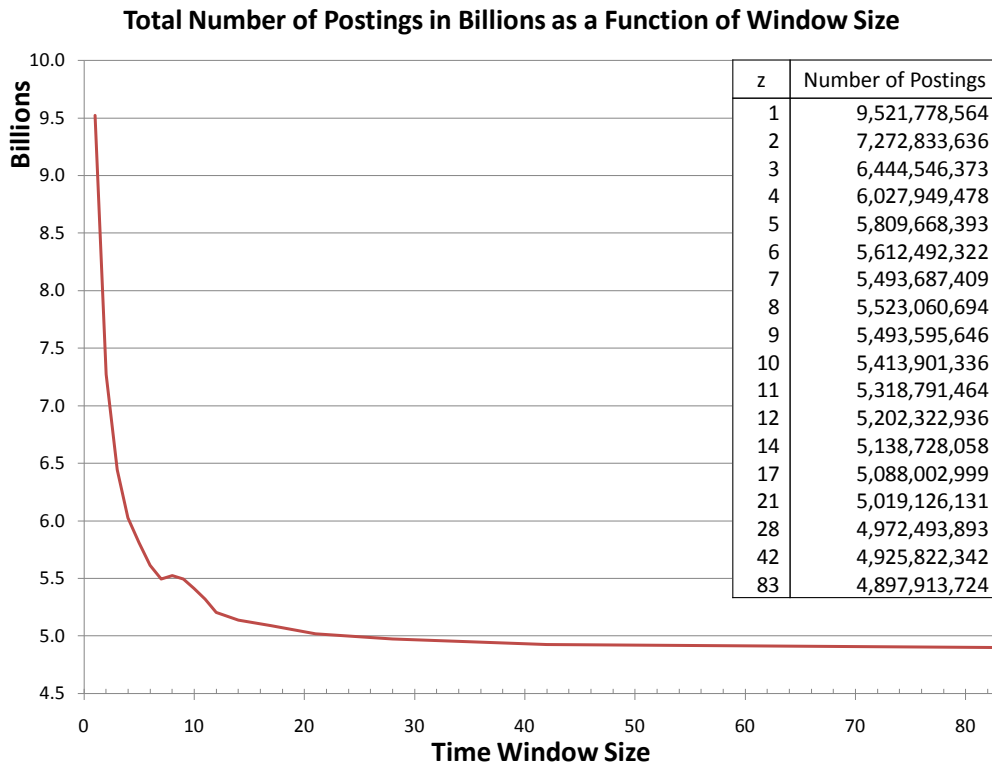


Figure 3.9: Wikipedia: Total Number of Postings

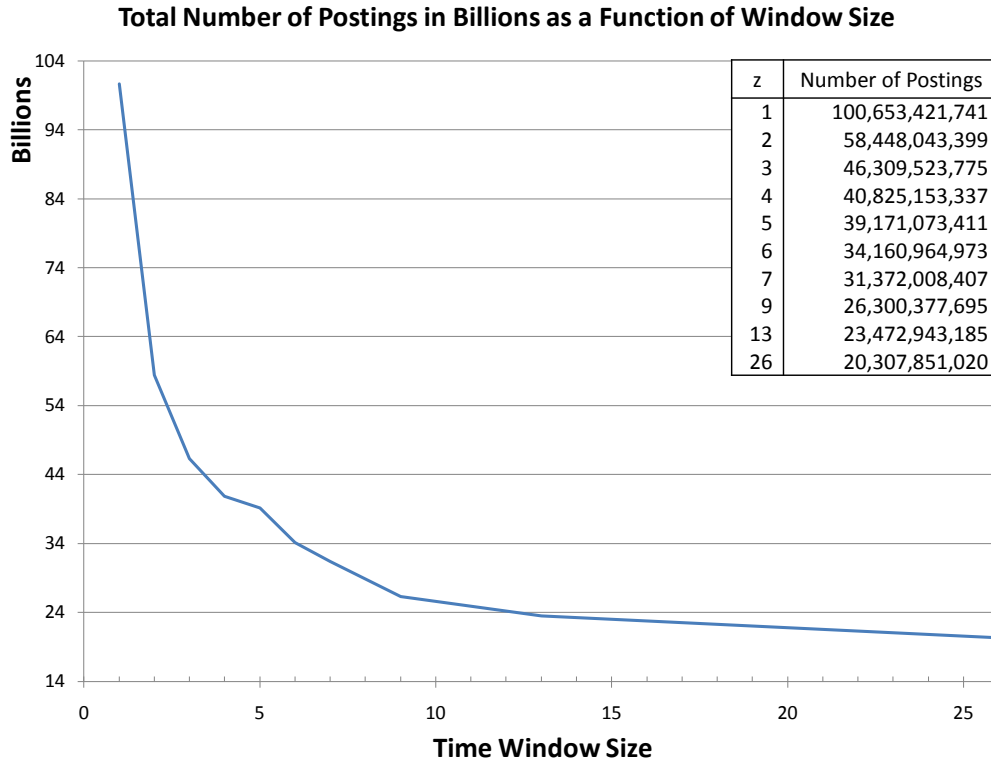


Figure 3.10: Library of Congress: Total Number of Postings

As expected, the larger the time window, the smaller the number of postings since fewer document versions will cross time window boundaries. However we note that for our datasets, the reduction in the number of postings becomes relatively small after $z = 11$ for the Wikipedia dataset and $z = 9$ for the Library of Congress dataset. This fact implies that the storage overhead is small compared to the best possible for relatively small time windows. For instance, the storage overhead is only around 12% and 30% for the previously determined "optimal" time window sizes of 7 and 8 for the Wikipedia dataset and the Library of Congress dataset, respectively, compared to the minimum space required when $z = n$.

3.5.4 Empirical Results on Average Number of Postings Examined for a Typical Query Load

Our temporal query load for the Wikipedia dataset is constructed as follows. Based on the AOL query log made briefly available in 2006 [64], we extract 223 most frequent multi-term query phrases where the user selected an English Wikipedia article among the search results. Each query phrase is combined with 100 random query time spans resulting in a query load of 22,300 temporal queries for each time window size. Similarly for the Library of Congress dataset, we extract 100 most frequent multi-term query phrases where the user selected one of the seed websites. The seed websites are those included in the seed URLs that the Library of Congress used as an input to the crawler. Again, each query phrase is combined with 100 random query time spans resulting in a query load of 10,000 temporal queries for each time window.

For each time window size of the two datasets, we compute the average number of postings examined over all these temporal queries. The results are illustrated in the next figures.

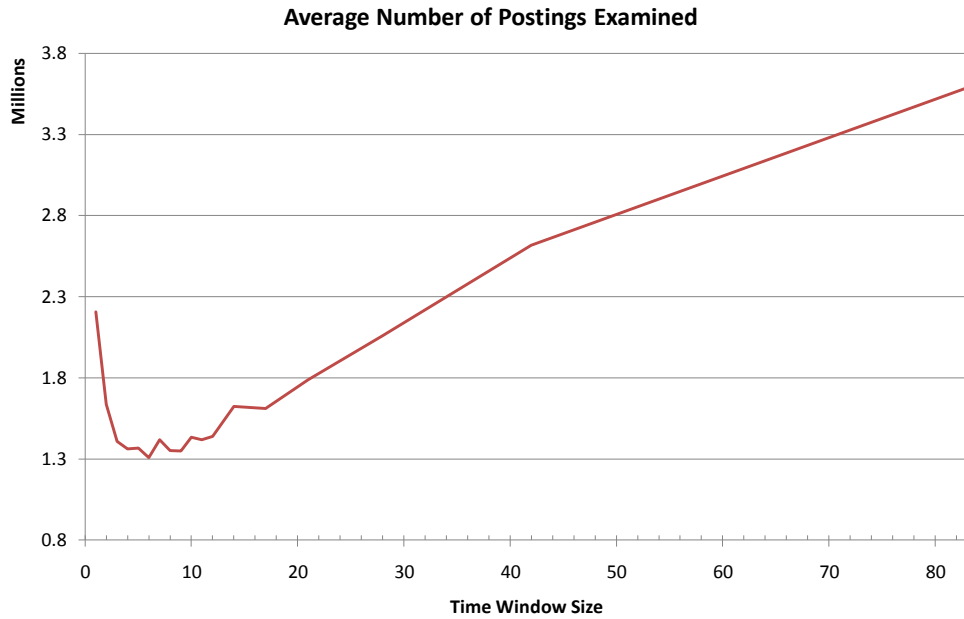


Figure 3.11: Wikipedia: Average Number of Postings Examined

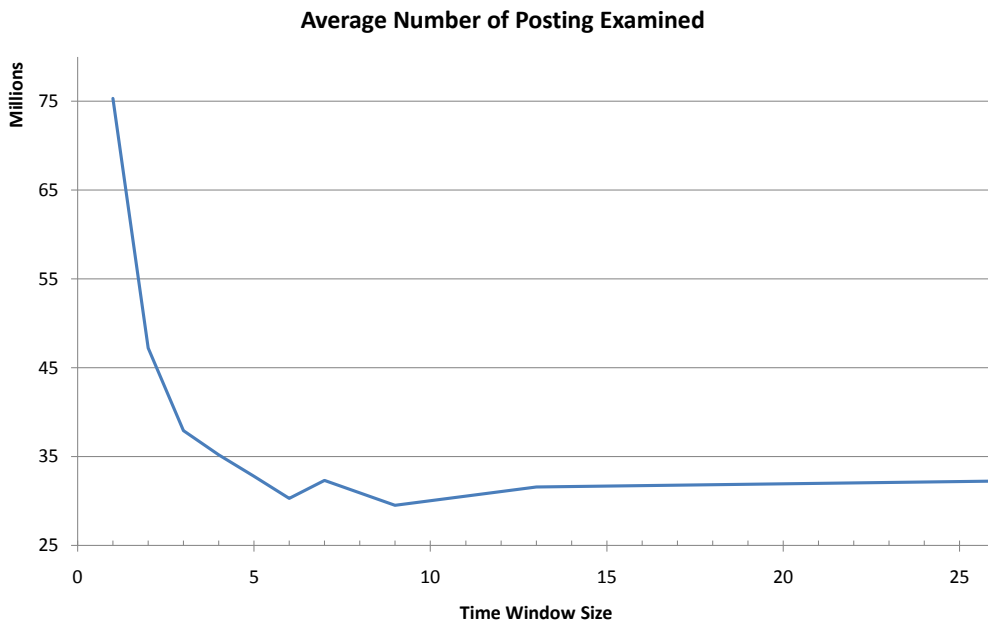


Figure 3.12: Library of Congress: Average Number of Postings Examined

As expected, the relationship between this metric and the time window size follows more or less the same behavior as the sum $X+Y$ introduced in Section 3.4.1. In particular, note that the larger the time window the less the number of duplicate

postings, but the more the number of postings that are irrelevant to the given query time span. Based on the results of these tests, the corresponding best values are $z = 6$ and $z = 9$ for the Wikipedia and the Library of Congress datasets respectively, which are very close to the values ($z = 7$ and $z = 8$, respectively) predicted by our analytical analysis presented in Section 3.4.1.

3.5.5 Empirical Evaluation of Ranked Search Results

Using the same temporal query loads as described in the previous section, we evaluate the top 100 ranked results obtained by using our approach according to two measures – *Relative Recall* and *Kendall’s τ* – assuming a ground truth list of the top 100 ranked document versions generated by using the same scoring functions with exact state statistics. Relative Recall is defined as the fraction $\frac{n_r}{100}$, where n_r is the number of document versions among the 100 returned by our scheme which also appear on the ground truth list. Kendall’s τ is defined as the fraction $\frac{n_{concord} - n_{discord}}{4950}$, where $n_{concord}$ is the number of concordant pairs, and $n_{discord}$ is the number of discordant pairs. A pair (a, b) is concordant if a and b appear in the same order in the list produced by our scheme and the ground truth list, and is discordant otherwise. Note that the number of the distinct pairs of 100 elements is 4950.

We compare the ranked search results as a function of time window size. Clearly the smaller the time window size, the more accurate the statistics are and hence the better the recall and Kendall’s τ are. In fact, the case when the time window size is equal to one reduces to computing the exact statistics for any query time span.

Therefore, this case represents the ground truth to which we compare the performance of any other time window size.

For each dataset, we perform two sets of tests using Okapi BM25 and KL-divergence smoothed by Dirichlet priors, respectively. From the search results of each test, we compute Relative Recall and Kendall's τ for the top 100 ranked search results. The resulting data is illustrated in the new graphs.

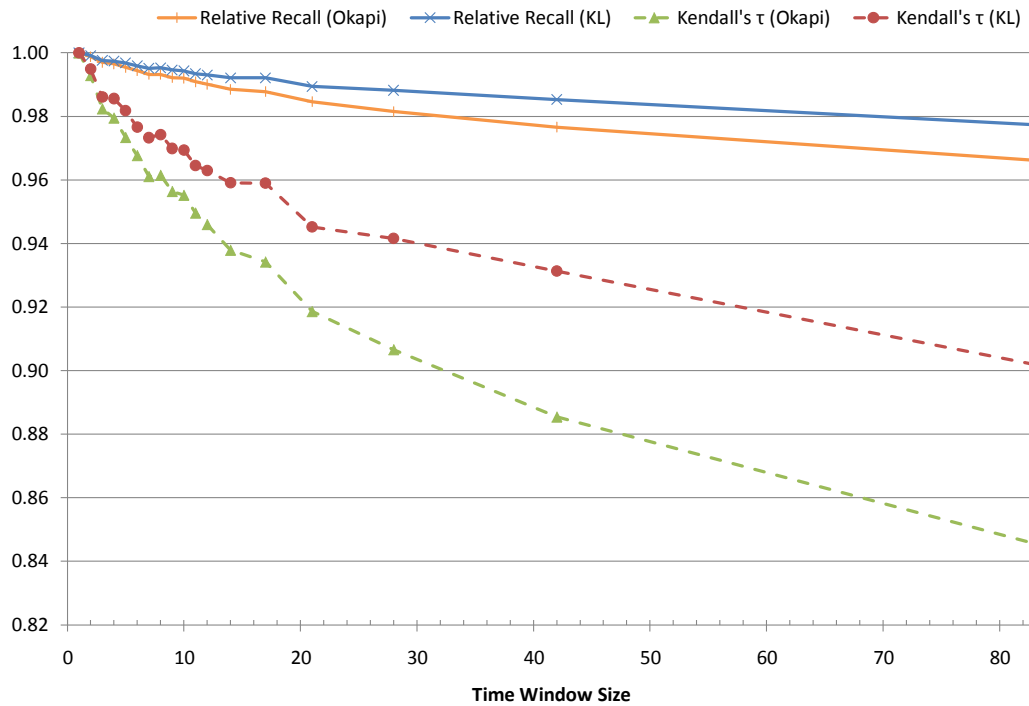


Figure 3.13: Wikipedia: Relative Recall and Kendall's τ

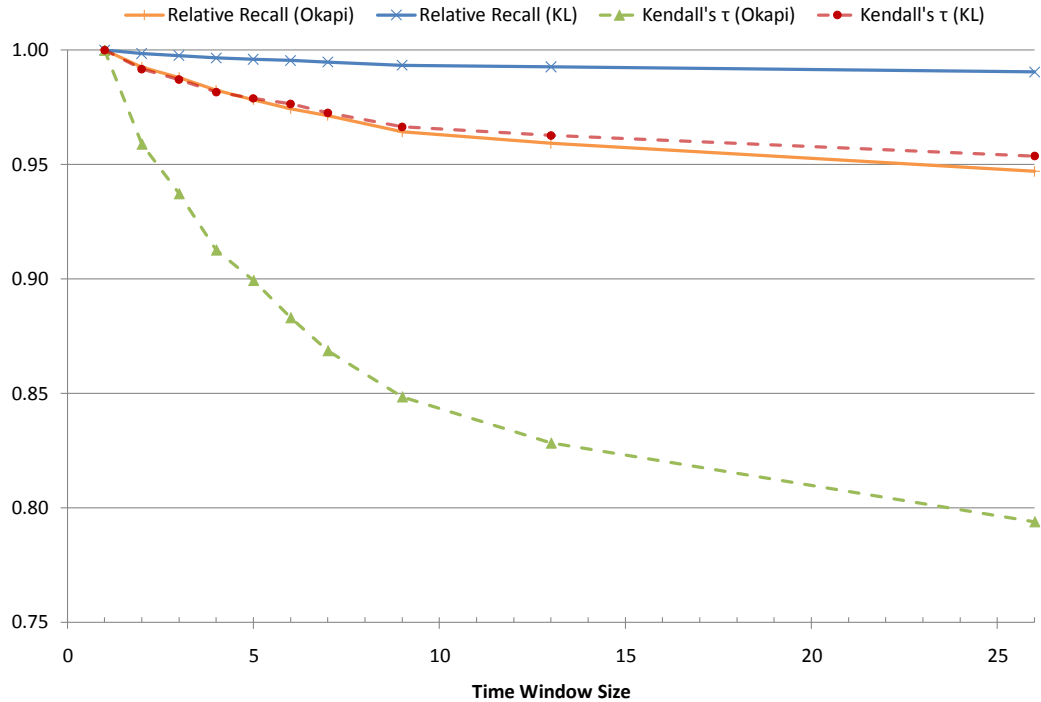


Figure 3.14: Library of Congress: Relative Recall and Kendall's τ

From the graph for the Wikipedia dataset the values of Relative Recall and Kendall's τ are higher than 0.99 and 0.96 respectively for the “optimal” time window size of 7 as determined before. Similarly for the Library of Congress dataset, for the time window size of 8, the values of Relative Recall and Kendall's τ are higher than 0.97 and 0.85, respectively. This implies that the search results for our “optimal” time window size are almost the same as those produced using the exact state statistics.

In our datasets, even the largest time window size does not yield search results that are substantially different than those appearing on the ground truth list (for instance, Kendall's τ for time window size 26 in the Library of Congress is almost 0.80). This implies that the use of the statistics of the entire history of our datasets would give reasonable search results. However, a careful examination of the curves indicates a definite negative trend as the temporal range increases. Extrapolating these

curves implies a substantial degradation of the search results after the temporal range moves beyond a certain point.

3.6 Conclusion

In this chapter, we presented a new approach to index a collection of multi-version documents, which incorporate the temporal dimension in an integral way to enable the handling of temporally anchored queries. In particular, our approach introduces the notion of time windows, each of which is organized using standard structures. We show that the time window size directly affects the search performance, and provide an analytical model that can be used to derive optimal values for window sizes. Empirical evaluations on two large-scale real world datasets provided a strong support for our overall approach. In particular, we show that our approach effectively supports effective temporal search and the computation of relevance scoring based on the state of collection as it existed during the query time span.

Chapter 4

Physical Location Index for Time Evolving Data

In the previous chapter, we examined information retrieval schemes based on temporally anchored queries. Once the pertinent information is returned, the next step is to deliver the actual data to the user. In this chapter, we consider the issue of physically locating the data identified through the information retrieval scheme. In particular, we introduce a persistent data structure to be used to locate the physical location of time-evolving data. Our data structure indexes archived temporal contents such that it can provide fast access to the location of the archived contents for arbitrary temporal queries. Our scheme supports the insert operation and the most query types optimally, in terms of both time and space. It provides more compact and faster operations than the existing location indexing schemes used for web archives, while being simpler than the existing optimal persistent data structures. To simplify the presentation, we focus on web archives for the rest of this chapter.

4.1 Overview

When archiving web contents, one predominant storage method is to manage a smaller number of larger-size containers, each containing a multitude of web objects. The Internet Archive, the world's largest web archive, has also been using the container method, followed by many other web archiving consortiums and libraries.

Their collaborative efforts recently resulted in an international standard of the container's internal format, WARC [82].

Although the container method has become popular for its space-efficiency, easy-manageability, and many existing tools that deal with the container format, a drawback is the requirement of an external indexing scheme to locate web objects in the containers. Unlike traditional indexing schemes, a web archive index faces several additional hurdles due to the fact that archived web contents are temporal objects. As a consequence, an index must also take into consideration acquisition time in addition to the URLs. This is in contrast to the index used for web search engines where only currently available web materials need to be maintained.

In this chapter, we introduce an indexing scheme based on a persistent B^+ -Tree. Since long-term archives do not call the delete operation frequently, if ever, we loosen our requirement such that the delete operation can be sub-optimal. This allows our scheme to be much more compact and simpler than other optimal persistent data structures, while supporting the optimal query and insert operations in terms of both time and space.

In Section 4.2, we discuss current popular methods to index web archives, and also examine several existing persistent data structures closely related to our scheme, followed by Section 4.3 where we state our indexing goal. We explain our strategy in Section 4.4, and the performance analysis is presented in Section 4.5. The performance analysis is based on a number of assumptions that are justified in Appendix A.

4.2 Related Work

In this section, we describe existing location indexing schemes for web archives, followed by examining some of the most notable persistent data structures.

4.2.1 Location Index for Web Archives

Indexing schemes can be divided into two categories: sequential and structured indexes. The former kind is simple and straightforward in terms of its organization. A plain file directly on a native file system can hold a sequential index without much effort. The sequential index usually relies on external sorting and other data arrangement algorithms before records are written on a final index file. This implies a batch-mode indexing, where during each batch all the data to be indexed need to be available. On the other hand, structured schemes are generally more sophisticated and flexible at the expense of added complexity. Data records are more efficiently managed, typically using a B-Tree or one of its variants such as B⁺-Tree. A B-Tree is a tree data structure where each internal node stores n records of the $\{key, data\}$ pair in non-decreasing order, such that $key_1 \leq key_2 \dots \leq key_n$. Each internal node also contains $n+1$ pointers to its children, and the adjacent keys separate the ranges of keys in the subtree each pointer points to. That is, all the keys in the first subtree are smaller than the first key of the parent node, and all the keys in the second subtree are larger than the first key and smaller than the second key of the parent node, and so on. In a B⁺-Tree, in contrast to a B-Tree, all *data* are stored at the lowest level of the tree; only keys are stored in interior nodes. For more details on these data structures, we refer to Bayer and McCreight [4]. The B-Tree and B⁺-Tree support fast dynamic

insertion and deletion of index entries as well as fast querying. In practice, databases are often involved as middleware to accommodate structured schemes. As of this writing, the Wayback Machine [75] from the Internet Archive supports both the file-based sequential and B⁺-Tree-based structured indexes. We compare the current indexing schemes to our scheme when we perform the performance analysis in Section 4.4.

4.2.1.1 Sequential: File-Based Scheme

The file-based scheme is essentially a sequential indexing scheme. One URL entry occupies each line containing information such as when the URL was accessed, the message digest of the content, and the location in the storage (such as a filename and an offset). To facilitate the query responses, the entries are sorted alphabetically by the URLs before actually serving as an index. Once sorted, the index can be queried in $O(\log_2 N)$ where N is the total number of entries in the index, using a binary search algorithm, otherwise, it would take $O(N)$. However, the necessity of sorting makes this type of indexing structures especially unattractive in cases where a batch-mode indexing cannot be afforded.

4.2.1.2 Structured: Database-Based Scheme

A popular indexing scheme such as B⁺-Tree is also widely used in web archives. For example, the Wayback Machine also optionally provides a database-assisted indexing scheme. Specifically, it makes use of the B⁺-Tree data structure implemented within Berkeley DB. However, to accommodate both the URL and time indexes, rather than using either the URL or time as a key, the Wayback Machine

concatenates the URL and time, and uses the result as a key. Although this scheme is very easy to implement, and works well to handle queries with a specific URL and time, it has significant limitations when it comes to handling time slice or time span queries. For example, given a specific date or time span, a query to retrieve all the web pages of interest cannot be handled efficiently using this scheme. Moreover, in this scheme, the cost for insert operations depends on the total number of entries in the structure, which keeps ever incrementing as time goes by.

A possible alternative is to set up two indexing structures, one using the URLs as keys and the second using the crawl times as keys. Two separate searches are performed on the indexing structures, and the results are then matched to find the final result. As the archive grows, the time it takes to combine the results of the separate searchers grows rapidly, and response time will suffer substantially. That is, if there are V versions and as many as M data in a version, a simple matching algorithm requires $O(V \times M)$ processing time, making the response time unacceptably slow in many searches.

4.2.2 Persistent Data Structures

Conventional database captures a single state of a collection of data. Transactional database evolves from one state to the next, but the previous state is discarded once a transaction commits. “Persistent” (or multi-version) data structures, on the other hand, concurrently handle multiple versions of data, allowing efficient access to the previous versions. In this section, we discuss some of the notable persistent data structures below.

Easton [13] proposed Write-Once B-Tree (WOBT) that has been the basis of several subsequent persistent data structures [5,12,43,44,79]. WOBT mainly focuses on implementing an access structure on write-once media such as optical disks. WOBT allows insert operations only on the current version, but access operations on any previous versions.

Lomet and Salzberg [43,44] improved WOBT by allowing a time split according to an arbitrary point of time in the past, rather than the current time only. Although they manage to maintain good space and time efficiency for certain queries, some query types such as key history queries are not optimally supported.

Multiversion B⁺-Tree (MVBT) is developed by Becker, et al [5]. Most notably, MVBT differentiates itself from others by allowing delete operations too. Even with the introduction of the delete operations, it maintained time and space complexity at their optimal level. Further enhancing MVBT, Varman and Verma [79] developed Multiversion Access Structure (MVAS), using more storage conservative overflow and underflow policies. For example, they merge sparse siblings after deletes, and they also avoid creating two blocks in a version split in some cases by copying only as many live entries as needed from the sibling, instead of all entries. Furthermore, they achieve the optimality in key-history queries using the access-lists.

Although WOBT, MVBT and MVAS support the insert operation optimally, the involved overhead is not negligible in terms of both time and space, since they frequently require splitting an internal node.

4.3 Our Indexing Goal

When a web crawler gathers web objects, they are typically aggregated in containers, such that each container holds a number of web objects, ranging from several to hundreds of thousands. With the container method, the number of archived objects to manage is significantly decreased, and can overcome any practical limits of the underlying storage system (file system or database). As explained earlier, a notable standard format of such a container is WARC, which resulted from an international collaborative effort among many national libraries and archives. Although an individual container may be self-contained, almost all web archives set up and manage external indexes that map a URL and crawl time to the ID of the container and the offset where the corresponding information is archived. Without an external index, a sequential scan through all ARC files to search for the web information will take a prohibitive amount of time for any significant size web archive.

The goal of this chapter is to design an URL indexing scheme that will be able to execute the following operations as fast as possible, using minimal amount of storage space at the same time.

- $\text{INSERT}(e)$: inserts an entry e into the index. The entry e contains a URL as a key, the acquisition time, and the location of the archived content in the archive.
- $\text{URLTIMEQUERY}(url, t)$: returns the location of the archived content ($\text{key}=url$) that was acquired most recently but no later than t .
- $\text{URLTIMESPANQUERY}(url, t_1, t_2)$: returns all the locations of the contents ($\text{key}=url$) that were acquired between time $t_1 \sim t_2$.

- $\text{TIMESLICEQUERY}(t_1, t_2)$: returns all the locations of the contents that were acquired between time $t_1 \sim t_2$.

4.4 Our Strategy – Persistent Indexing Structure for Archives (PISA)

A web archive deals with a massive set of temporal data where each data is updated frequently. Therefore, we incorporate a persistent data structure that allows us to access and insert indexing entries very efficiently. We call our data structure PISA (Persistent Indexing Structure for Archives).

As with many existing persistent (or multi-version) data structures, such as [5,12,43,44,79], our indexing structure is also rooted in the Write-Only B-Tree [13]. More recently, researchers [5,79] found asymptotically optimal structures that support multiple versions of data. They showed that their operations such as insert, update, delete and query can be performed with optimal time and space.

Compared to previous data that these existing schemes handle, in a digital preservation system, once archived, data are seldom deleted, if ever. Therefore, in most cases, it is acceptable not to achieve optimal delete operations in an archiving environment, which allows us to greatly reduce the complexity in the indexing structure, while maintaining the same or better performance than the existing persistent data structures. We now discuss our indexing structure in detail.

4.4.1 Persistent Indexing Structure for Archives (PISA)

In this section, we explain how PISA is constructed. Like other persistent data structures, PISA is also based on a B⁺-Tree. We start by introducing the terms we use in the rest of this chapter. In our tree structure, we call internal nodes in the tree including the root *index blocks*, while we call leaf nodes *data blocks*. We call data items in an index block as *index entries*, and those in a data block *data entries*. Throughout this chapter, we use the term *block* to mean both index block and data block, and *entry* to mean both index entry and data entry, whenever the distinction is not necessary. Included in each entry is a lifespan, $(t_1 \sim t_2)$, where we call t_1 *birth time*, and t_2 *death time*. We call an entry *live* if its death time is ∞ (i.e. it has not currently been either updated or deleted), and *dead* otherwise. Similarly, we call a block *live* if at least one of the entries in the block is *live*, and *dead* otherwise.

Each index block is composed of a header entry and a series of index entries. Inside the header entry are the number of all entries in the block, the number of live entries in the block, the pointer to the previous version, and the key with which the upper level block indexes the current block. Each index entry contains the index key, birth time, death time, and a pointer to a block in the child level. We use $\{key, t_birth, t_death, ptr\}$ to indicate these four fields.

Similarly, a data block includes a header entry and a series of data entries. The header entry in a data block contains the same types of information as in the header entry in an index block. Each data entry is composed of four fields, $\{key, t_birth, t_death, loc\}$, that correspond to the data key (URL), birth time, death time, and the

location where the actual content with the data key resides in the archive. Note that the location information typically has a container ID (or filename) and an offset.

Figure 4.1 illustrates an example of PISA that has one index block and two data blocks.

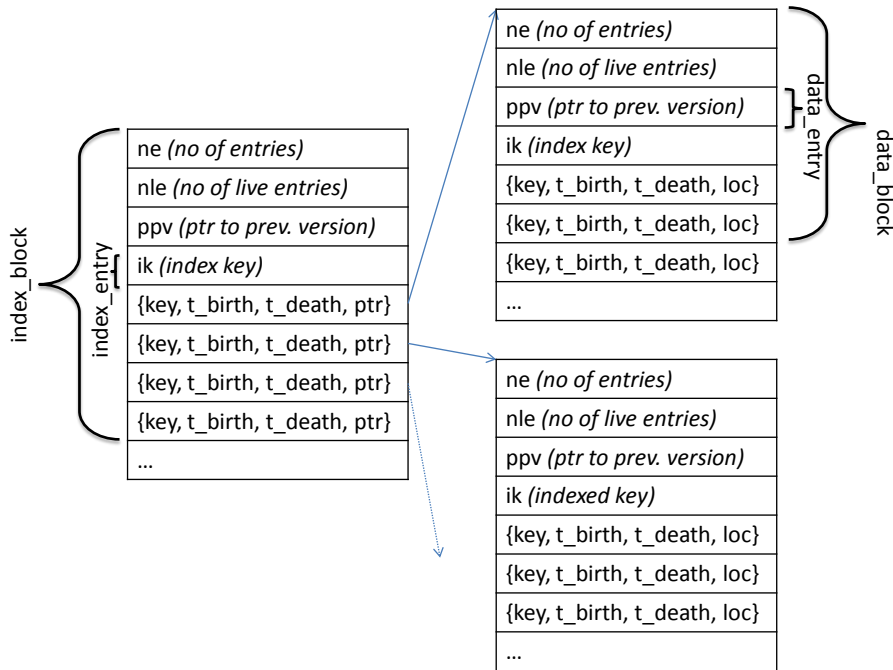


Figure 4.1: Blocks and Entries in PISA

As in a B^+ -Tree, the two index keys in two consecutive index entries determine the range of key values in the entries of the child block that the first entry's pointer points to. An example is shown in Figure 4.2 where the data entries (keys: A, B, C, E, F, L, N, P, S, and V) with the same birth time (7/1/2007) are inserted in an empty structure. Here, BLOCK 1 is an index block, and BLOCKS 2~4 are data blocks. For simplicity, an entry has an alphabet letter as a key in Figure 4.1. However, in practice, either a URL or a hash of a URL serves as a key.

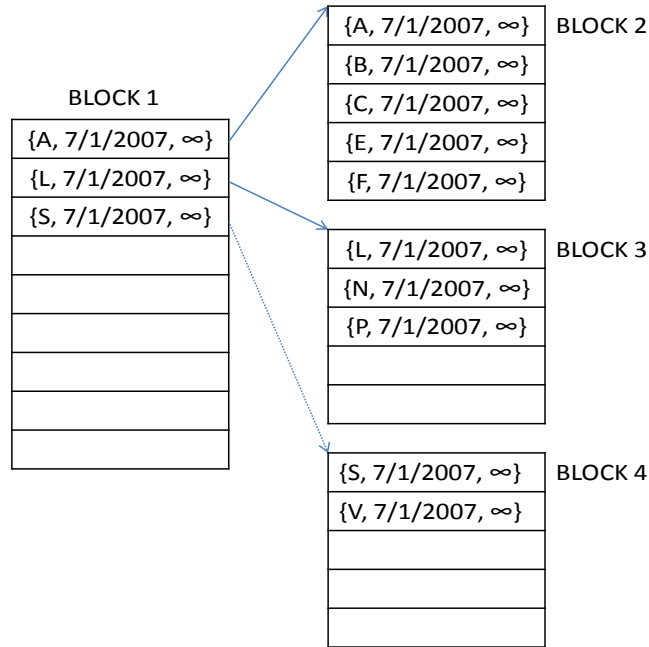


Figure 4.2: PISA Example

4.4.2 Operations in PISA

PISA supports `URLTIMEQUERY`, `URLTIMESPANQUERY`, `TIMESLICEQUERY` and `INSERT` optimally. Each of the operations is explained below.

- `URLTIMEQUERY`

Upon called with a URL and time as input parameters, `URLTIMEQUERY` returns the location of the object whose key matches the URL, and lifespan includes the input time (there is only one such entry or none). For URL k and time t , `URLTIMEQUERY` begins the search from the root block of PISA. It ignores the entries whose birth time is greater than t . Also ignored are entries whose death time precedes t . Among the surviving entries, it chooses the one with the largest key that is no larger than k . It follows the pointer in the chosen entry to the child block. It continues performing the

same selection process until it arrives at a data entry that has the location information for the URL at time t . The pseudo code is shown in Figure 4.3.

```

Input
URL key: URL of a web object
Timestamp ts: time stamp of the web object

Return
data_entry: data entry that contains key at ts
data_block: data block that contains the above entry

Procedure URLTIMEQUERY(key, ts)
1: index_entry ie ← null
2: index_block ib ← ROOT
3: data_block db ← null
4: while ( true )
5:   for ( each index_entry ce ∈ ib )
6:     if ((ce.t_birth ≤ ts < ce.t_death) and (ce.key < key))
7:       if ( (ie = null) or (ie.key < ce.key) )
8:         ie ← ce
9:   if (ie.pointer references leaf)
10:    db ← ie.pointer
11:    break while-loop
12:   else
13:    ib ← ie.pointer
14:   for (each data_entry de ∈ db)
15:     if ( (de.t_death ≥ ts) && (de.key = key) )
16:       return {de, db}
17:   return {null, db}

```

Figure 4.3: URLTIMEQUERY Operation

- URLTIMESPANQUERY

URLTIMESPANQUERY has three input parameters, a URL, start time, and end time. It returns all the locations of the objects whose key is the input URL, that were live during the time span of $(t_1 \sim t_2)$. It first performs the same steps as URLTIMEQUERY for time t_2 . When we find a data block that contains an entry for t_2 , it backtracks to the block that may contain the previous version by following the backtracking pointer in the current block. The backtracking continues until it encounters a block that died before t_1 . If any previous versions of the entry at t_2 existed since t_1 , all of these

previous entries can be found in the blocks it encountered during the backtracking.

Figure 4.4 has the pseudo code.

```
Input
URI key : URI of a web object
Timestamp ts_start: start time stamp of the web object
Timestamp ts_end: end time stamp of the web object

Return
A set of data_entry de's whose de.key=key and (de.t_birth~de.t_death) overlaps
(ts_start~ts_end)

Procedure URLTIMESPANQUERY(key, ts_start, ts_end)
1: List E<data_entry> ← null
2: data_entry de ← null
3: data_block db ← null
4: (de, db) ← URLTIMEQUERY(key, ts_end)
5: while ( (de ≠ null) && (de.t_death > ts_start) )
6:   E ← E ∪ {de}
7:   db ← db.ppv
8:   de ← FINDENTRYINBLOCK(de, db)
9: return E
```

Figure 4.4: URLTIMESPANQUERY Operation

- **TIMESLICEQUERY**

TIMESLICEQUERY returns the locations of all the objects that were alive during the supplied time span. Starting from the root, the search travels through all the blocks that were alive at a specific time period, $t_1 \sim t_2$. It follows the pointer in an entry only if one or more of the following three conditions are met. 1) The birth time of the entry is between t_1 and t_2 . 2) The death time of the entry is between t_1 and t_2 . 3) The birth time of the entry is before t_1 and the death time of the entry is after t_2 . Upon arriving at the leaf nodes, the entries that were alive during $t_1 \sim t_2$ are returned, using the same eligibility check above. The pseudo code is shown in Figure 4.5.

```

Input
Timestamp ts_start: start time stamp of the web object
Timestamp ts_end: end time stamp of the web object

Return
A set of data_entry de's whose (de.t_birth~de.t_death) overlaps (ts_start~ts_end)

Procedure TIMESLICEQUERY(ts_start, ts_end)
1: List E<data_entry> ← null;
2: List B<index_block> ← { root }
3: data_entry de ← null
4: data_block db ← null
5: index_block ib ← null
6: block cb ← null
7: for ( each index_block ib ∈ B )
8:   for ( each index_entry ce ∈ ib )
9:     if ((ts_start ≤ ce.t_birth < ts_end) or (ts_start ≤ ce.t_death < ts_end)
        or ( ce.t_birth ≤ ts_start) and (ce.t_death > ts_end))
10:      cb = ce->pointer
11:      if (cb is data block)
12:        for (each data_entry de ∈ cb )
13:          if ((ts_start ≤ de.t_birth < ts_end) or (ts_start ≤ de.t_death < ts_end)
              or ( de.t_birth ≤ ts_start) and (de.t_death > ts_end))
14:            E ← E U {de}
15:          else /* cb is index block */
16:            B = B U {cb}
17: return E

```

Figure 4.5: TIMESLICEQUERY Operation

- INSERT

INSERT adds a new data entry into PISA. In case that there already exists a previous entry with the same key, INSERT marks the previous entry dead and adds the new entry. INSERT is a more involved operation than queries. Before we explain the INSERT operation, we first define two parameters and three variables in Tables 4.1 and 4.2.

B_{max} and B_{min} are configurable parameters depending on the performance and space needs. Since B_{max} is often set to a value equal to the block size of the storage to achieve the maximum disk-seek performance, one usually has to decide only B_{min} .

The value of B_{min} has a direct impact on the time and space performance as can be seen in Section 4.5. However, we cannot use any arbitrary value. We formally discuss how to determine B_{min} and B_{max} , considering their impacts on the time and space performance in Appendix A. In this chapter, we only make one simple assumption: all parameters are positive integers.

Table 4.1: PISA Parameters

Parameter	Description
B_{max}	The maximum number of entries that a block can contain.
B_{min}	The minimum number of <i>live</i> entries that a block <i>must</i> contain.

Table 4.2: PISA Variables

Variable	Description
n_a	The number of <i>all</i> entries that a block currently contains.
n_l	The number of <i>live</i> entries that a block currently contains.

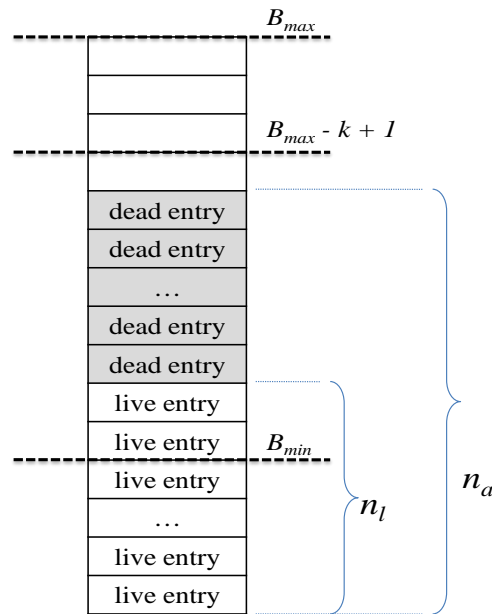


Figure 4.6: PISA Block (Live entries clustered together for illustrative purpose)

The two variables, n_a and n_l , on the other hand, are defined to simplify the presentation of the algorithms. Although not strictly necessary, it is assumed that n_a and n_l are stored in the header of a block to conveniently describe the algorithms. Figure 4.6 illustrates these parameters and variables.

When a new data entry is inserted, the appropriate data block that the data entry belongs to is first identified. Once the block is identified, we mark any existing entry with the same key as *dead*. If there is an empty slot in the block for the new entry, the entry is added to the block. If the block is full ($n_a = B_{max}$), a new block is created, according to the following rules:

Case 1: $n_l + 1 \geq 4 \times B_{min}$

KEYSPLIT: Among the n_l live keys, we select a median entry, m_e , such that 1) no other live entries are older than m_e , 2) there are at least B_{min} live entries whose key is smaller than m_e 's, 3) and there are at least B_{min} live entries whose key is no smaller than m_e 's. If there is no such median key, we perform VERSIONSPLIT, and restart KEYSPLIT over the new block spawned from VERSIONSPLIT. Otherwise, we create a new block, move live entries whose key is no smaller than the median key into the new block. We also move dead entries that were alive at the birth time of the median entry into the new block. We INSERT an index entry that points to the new block in the parent. In case the current block is the root block, we also create the new root block which points down to the previous root block and the new block.

Case 2: $n_l + 1 < 4 \times B_{min}$

VERSIONSPLIT: We create a new block. We copy all the n_l live keys from the overflowing block into the new block. We mark the copied entries in the overflowing

block as *dead*. We set the birth time of the copied entries of the new block the same as the birth time of the new entry. In case the overflowing block is the root block, we also create the new root block which points down to the previous root block and the new block.

The pseudo codes for INSERT and its supplementary functions (KEYSPLIT and VERSIONSPLIT) are shown in Figures 4.7 ~ 4.10.

<p>Input data_entry de : data entry to insert</p> <p>Procedure INSERT(de) 1: data_block db \leftarrow null 2: db \leftarrow FINDBLOCK(de.key, de.t_birth) 3: BLOCKINSERT(de, db)</p>
--

Figure 4.7: INSERT Procedure

<p>Input entry e: (data_ or index_)entry to insert into db block b: (data_ or index_)block that de is inserted into</p> <p>Procedure BLOCKINSERT(e, b) 1: block newb \leftarrow null 2: entry newe \leftarrow ALLOCATEMEMORY() 3: entry olde \leftarrow FINDENTRYINBLOCK(e, b) 4: <i>if</i> (olde \neq null) 5: olde.t_death \leftarrow e.t_birth 6: <i>if</i> (b.na = B_{max}) 7: <i>if</i> ((b.nl + 1) \geq 4*B_{min}) 8: newb \leftarrow KEYSPLIT(e, b) 9: newe.key \leftarrow newb.ik /* ik: index key of the block */ 10: newe.t_birth \leftarrow e.t_birth 11: <i>else</i> 12: newb \leftarrow VERSIONSPLIT(e, b) 13: newe.key \leftarrow b.ik 14: newe.t_birth \leftarrow b.t_birth 15: newe.t_death \leftarrow ∞ 16: newe.ptr \leftarrow newb 17: BLOCKINSERT(newe, PARENT(b)) 18: <i>else</i> 19: ADDENTRYTOBLOCK(e, b)</p>

Figure 4.8: BLOCKINSERT Procedure

Input
 entry e : new entry to insert
 block b : block to version-split

Return
 block : new block resulting from the split

Procedure VERSIONSPLIT(e, b)

```

1: entry olde ← null
2: entry newe ← null
3: block newb ← ALLOCATEMEMORY()
4: newb.t_birth ← e.t_birth
5: newb.t_death ← ∞
6: newb.ppv ← b
7: newb.ik ← b.ik
8: for ( each live entry olde ∈ b ) /* olde.t_death > e.t_birth */
9:   if (olde.t_death > e.t_birth)
10:    newe ← DUPLICATEENTRY(olde)
11:    olde.t_death ← e.t_birth
12:    newe.t_birth ← e.t_birth
13:    ADDENTRYTOBLOCK(newe, newb)
14:  ADDENTRYTOBLOCK(e, newb)
15:  return newb

```

Figure 4.9: VERSIONSPLIT Procedure

Input
 entry e : new entry to insert
 block b : block to key-split

Return
 block: new block resulting from the split

Procedure KEYSPLIT(e, b)

```

1: entry olde ← null
2: block oldb ← b ∪ {e}
3: entry me ← FINDMEDIANKEY(oldb)
4: if (me = null) KEYSPLIT(e, VERSIONSPLIT(null, b))
5: block newb ← ALLOCATEMEMORY()
6: newb.t_birth ← oldb.t_birth
7: newb.t_death ← ∞
8: newb.ppv ← b.ppv
9: newb.ik ← me.key
10: for ( each entry olde ∈ oldb )
11:   if ( olde.key ≥ me.key )
12:     if (olde.t_death = ∞)
13:       MOVEENTRY(olde, oldb, newb)
14:     else if (olde.t_death > me.t_birth)
15:       COPYENTRY(olde, oldb, newb)
16: return newb

```

Figure 4.10: KEYSPLIT Procedure

For clarity, we illustrate an example of the INSERT operation in Figures 4.11 ~ 4.13. In Figure 4.11, no overflowing occurred after inserting three entries (keys: Z, X and Z) into the previous index we saw earlier in Figure 4.2. Note that the first inserted Z is marked dead.

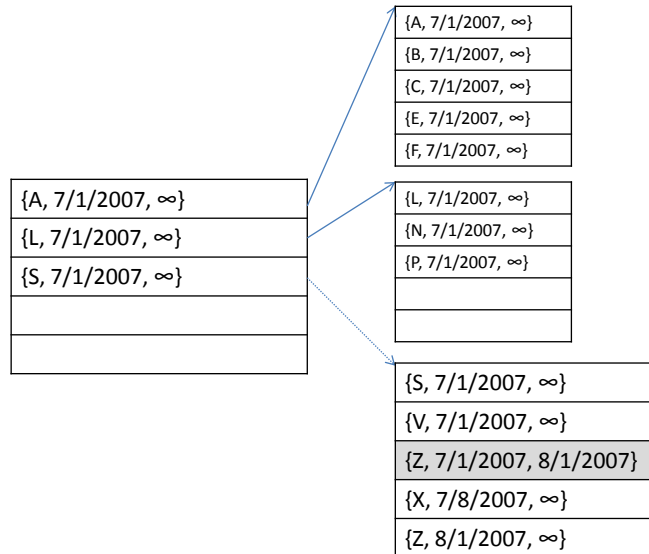


Figure 4.11: INSERT Z, X, and Z

Figure 4.12 shows a case where another entry with key ‘T’ is inserted into this index, making the block overflow. Since we have five live entries including the new entry, if we assume that B_{min} is one (thus $4 \times B_{min} = 4$), we fall into Case 1 where a key split is performed with {V, 7/1/2007, ∞} selected as a median entry.

Before explaining Case 2, we now go back to the original index (Figure 4.2), and insert three entries with key ‘S’. Again, since there are available slots, no overflowing occurs. We only mark all the previous entries with the same key ‘S’ as dead (Figure 4.13).

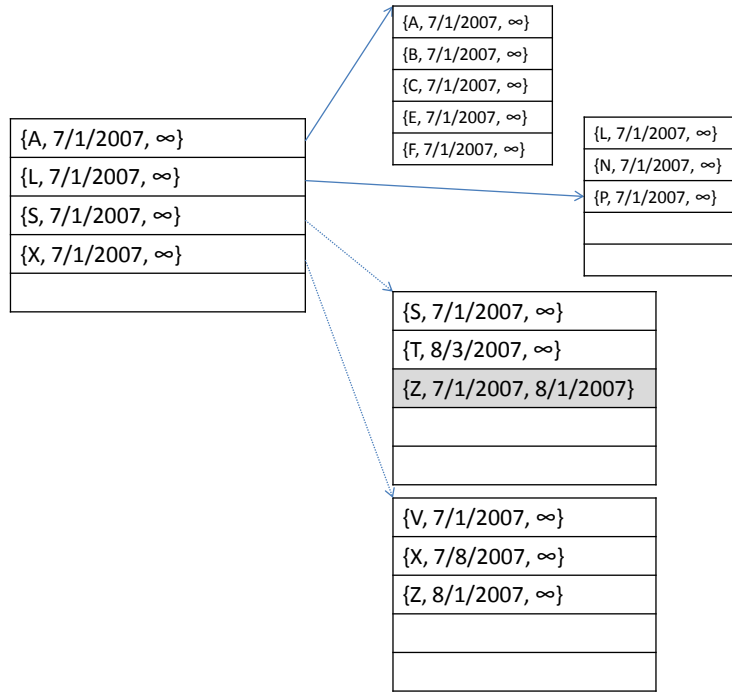


Figure 4.12: KEYSPLIT after Inserting T

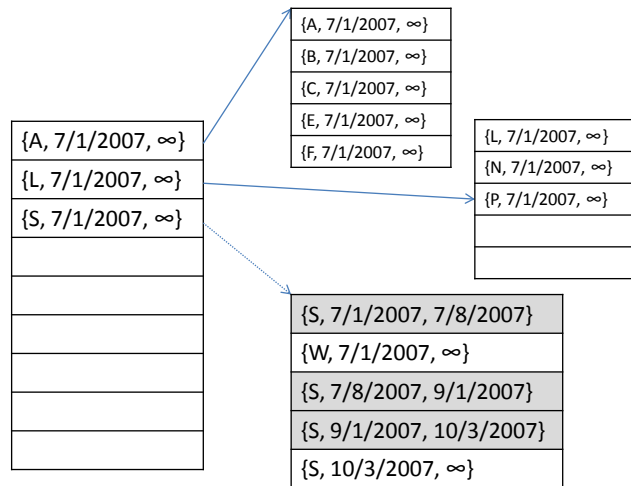


Figure 4.13: INSERT S, S, and S

Now we insert another entry with key ‘S’ again, and encounter an overflow. Since there are not enough live entries to key-split, we now fall into Case 2 where a version split is performed. Figure 4.14 shows the result of the version split.

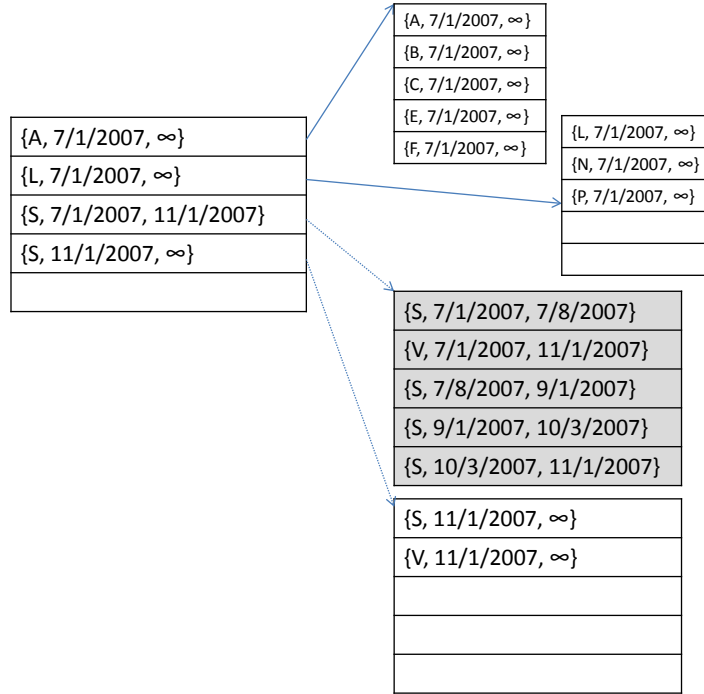


Figure 4.14: VERSIONSPLIT after INSERT(S, 11/1/2007, ∞)

4.5 Performance Analysis

In this section, we analyze the performance of PISA in terms of the time and storage space. We first list two invariants we want to maintain all the time throughout any operations.

Invariant 1: For any *live* block in PISA, $3 \leq B_{min} \leq n_l \leq n_a \leq B_{max}$

Invariant 2: For any *new* block in PISA, $3 \leq B_{min} \leq n_l \leq n_a \leq B_{max} - B_{min} + 1$

As defined earlier in Tables 4.1 and 4.2, B_{min} represents the minimum number of live entries in a block, and B_{max} represents the maximum number of entries in a block. Variables n_l and n_a represent the number of live entries and of all entries in a block, respectively.

By ‘any new block’ in Invariant 2, we mean any block that is newly created by a version split or a key split *as well as* the existing block that a key split has moved its entries out from. We note that the only case where the number of entries in an existing block decreases is after a key split, but since we put the existing block for the key split into the new block category, we need to maintain Invariant 2 for this block, too.

In this section, we assume that the invariants hold all the time, and analyze the PISA’s performance under this assumption. This assumption is justified in Appendix A. From each invariant, the following observations can be made.

Observation 1: From Invariant 1, there are at least B_{min} *live* entries at any point of time.

Observation 2: From Invariant 2, at least B_{min} operations need to be performed on a new block before an overflow can occur.

4.5.1 Query Time

For URLTIMEQUERY(Key k , Time t), since at least B_{min} entries are valid at t in every block (from Observation 1), we have the same performance boundary as B⁺-Tree where each block has at least B_{min} entries. Thus, the number of the accessed blocks is $O(\log_{B_{min}} N)$, where N is the total number of entries (or equivalently, the total number of the INSERT operations) in PISA. This is the best known lower bound for any multi-version data structure [71].

For URLTIMESPANQUERY(Key k , StartTime t_1 , EndTime t_2), after we identify the block and entry with key k that was live at t_2 , we backtrack to older blocks until we

reach to a block whose death time precedes t_1 . Supposing that there are R such entries, we need to backtrack to as many as $R-1$ older blocks. Therefore, we need to access a total of $O((\log_{B_{\min}} N) + R)$ blocks. We note that our complexity is sub-optimal for this particular query type. The optimal bound of $O((\log_{B_{\min}} N) + \frac{R}{B_{\max}})$ is not achieved because all qualifying entries are not clustered together. We note, however, that Varman and Verma [79] describe a way to achieve the optimal bound with the access list integration at the expense of the increase in time and space for other operations such as inserts. The same technique can be adopted in PISA to achieve the optimal performance.

`TIMESLICEQUERY(StartTime t_1 , EndTime t_2)` begins the search from the root block. All the child blocks that are alive during $t_1 \sim t_2$ are visited. Again, since at least B_{\min} entries are valid at any point of time in every block (from Observation 1), this type of query is processed optimally.

4.5.2 Insert Time

For `INSERT(DataEntry e)`, we search from the root to locate the appropriate data block whose key ranges include the key of e . Since there are at least B_{\min} live entries in every block, the search takes $O(\log_{B_{\min}} M)$ time where M is the number of current live blocks.

After we identify the block, we check whether or not there is a free slot for the new entry. If there is an available slot, then we add the entry and complete the job. However, if the block is full, we split the block by creating one or two new blocks depending on the split type and other conditions explained previously. We then insert

an index entry that points to the new block into the parent block, which may also overflow. In the worst case, every ancestor block on the path to the root may also overflow. Since we need to access as many as two blocks each level, the worst case requires us to visit as many as $(2 \times \log_{B_{\min}} N) + 1$ blocks (one added in the end represents the new root). Therefore, the total time for INSERT is bounded by $O(\log_{B_{\min}} N)$, which is larger than the optimal INSERT complexity of $O(\log_{B_{\min}} M)$.

However, we can show that the amortized complexity is still optimal. We consider Observation 2: at least B_{\min} operations are needed before an overflow occurs. Since an overflow adds at most two index entries into the parent block, an overflow can occur at the parent after at least $\frac{B_{\min}}{2}$ INSERT operations in the child level.

Similarly, the grandparent block can overflow only after at least $\frac{B_{\min}}{2}$ INSERT operations in the parent block, and so on. That is, at level L ($L=1$ for data blocks), at least $(\frac{B_{\min}}{2})^L$ leaf-level INSERT operations need to be performed before an overflow can occur. Since an overflow requires us to access as many as three blocks in the same level, one operation contributes to $3 / (\frac{B_{\min}}{2})^L$ block accesses at level L .

Therefore, the amortized number of block accesses caused by one INSERT is at most

$$\left(3 \cdot \sum_{i=1}^h \frac{1}{(\frac{B_{\min}}{2})^i} \right) + \frac{1}{(\frac{B_{\min}}{2})^{h+1}} = \frac{3(1 - (\frac{1}{\frac{B_{\min}}{2}})^h)}{\frac{B_{\min}}{2} - 1} + \frac{1}{(\frac{B_{\min}}{2})^{h+1}} < \frac{3}{\frac{B_{\min}}{2} - 1} = \frac{6}{B_{\min} - 2},$$

where h is the height of the tree, or $\log_{B_{\min}} N$. This shows that not only does INSERT cost a constant complexity after an appropriate block is located, the constant can be very small (less than $\frac{6}{B_{\min} - 2}$) depending on B_{\min} . Consequently, the amortized INSERT cost is no more than the time it takes to locate the target block, i.e., $O(\log_{B_{\min}} M)$. This is asymptotically optimal since this complexity is the same as that of B⁺-Tree that only maintains the current version of data.

4.5.3 Space

In order to analyze the space required for PISA after N INSERT operations, we first examine data blocks first. From Observation 2, we know that there need at least B_{\min} INSERT operations before an overflow can occur. Therefore, one INSERT operation contributes to at most $\frac{B_{\min}}{2}$ blocks (in most cases only one block is created as a result of a split, however, in some case as many as two blocks can be created). If we consider the initial block (whose need is explained in Section Appendix A), the space required for data blocks after N INSERT operations is upper-bounded by $\frac{2N}{B_{\min}} + 1$.

The space required by index blocks can be computed as follows. In the first index level (the parent level of data blocks), at least $\frac{B_{\min}}{2}$ new data blocks need to be created before an index block overflows and creates a new index block. We saw in the previous paragraph that at least $\frac{B_{\min}}{2}$ INSERT operations are required for a new

data block to be created. Therefore, at least a total of $(\frac{B_{\min}}{2})^2$ operations are required

for a new index block to be created. In other words, one INSERT operation contributes

to at most $\frac{1}{(\frac{B_{\min}}{2})^2}$ index blocks in the first index level. If we generalize this for all

the other index levels in the tree, we obtain that one INSERT operation contributes to at

most $\frac{1}{(\frac{B_{\min}}{2})^{i+1}}$ for the i^{th} index level. Therefore, the maximum number of new index

blocks after N INSERT operations is

$$\sum_{i=1}^{\log N} \frac{N}{(\frac{B_{\min}}{2})^{i+1}} = \frac{2N}{B_{\min}} \sum_{i=1}^{\log N} \frac{1}{(\frac{B_{\min}}{2})^i} = \frac{2N}{B_{\min}} \times \frac{\frac{2}{B_{\min}}(1 - (\frac{2}{B_{\min}})^{\log N})}{1 - \frac{2}{B_{\min}}} < \frac{4N}{B_{\min}(B_{\min} - 2)}.$$

If we also consider an initial index block, there are at most $\frac{4N}{B_{\min}(B_{\min} - 2)} + 1$

index blocks after N INSERT operations.

To summarize, the maximum number of both data blocks and index blocks

required after N INSERT operations is $\frac{2N}{B_{\min}} + 1 + \frac{4N}{B_{\min}(B_{\min} - 2)} + 1 = \frac{2N}{B_{\min} - 2} + 2$.

Therefore, PISA has an amortized space bound of $O(N/B_{\min}) = O(N/B_{\max})$, which is optimal (we show that B_{\min} is selected linearly to B_{\max} in Appendix A).

4.5.4 Performance Comparison

Based on the numbers that we have so far come up with, we compare the asymptotic performance complexities of PISA to the existing indexing structures for web archives. Table 4.3 summarizes the results.

Table 4.3: Performance Comparison (Cells with the best performance are shaded)

	File-based	B ⁺ -Tree -based		PISA
		One Merged Tree	Two Separated Trees	
URLTIMEQUERY	$O(\log_2 N)$	$O(\log_B N)$	$O(V \times M)$	$O(\log_B N)$
URLTIMESPANQUERY	$O(\log_2 (N + R))$	$O((\log_B N) + R/B)$	$O(R \times N \times M)$	$O((\log_B N) + R/B)$
TIMESLICEQUERY	$O(N)$	$O(N/B)$	$O(\log_B V)$	$O((\log_B N) + R/B)$
INSERT	$O(N)$	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B M)$
space	$O(N)$	$O(N/B)$	$O(N/B)$	$O(N/B)$
incremental	No	Yes	Yes	Yes

N: # of entries. M: # of entries in a version. V: # of versions B: Block size R: # of matching entries

We also note that, compared to other persistent data structures, PISA runs more efficiently, even for the operations that have the same asymptotic runtime. For example, PISA has the same asymptotic runtime as WOBT, MVBT and MVAS for the operations in Table 4.3. However, they all require two new blocks created for every key split, where as PISA only spawns one block in many cases and two new blocks in some cases, allowing PISA to run faster in practice, requiring compacter storage at the same time.

4.6 Summary

In this chapter, we introduced a persistent data structure, called PISA, to be used as a location index for temporal data. We showed how insert and queries are

performed in PISA. Our performance analysis showed that PISA provides insert and queries optimally in terms of both time and space, and it also has a lower overhead than other optimal schemes.

Chapter 5

Optimizing Data Layout for Web Contents for Fast Access

In Chapters 3 and 4, we discussed information discovery and delivery in a long-term preservation system, respectively. Once the relevant information is discovered and delivered, the user begins to navigate around the returned results, following the hyperlinks in an html, for example. In this chapter, we consider the problem of storing a linked set of inter-related data into containers in such a way as to minimize the number of containers accessed during an information browsing session. Our method makes use of link analysis and optimized graph partitioning to enable faster browsing of archived web contents in the future.

Our overall methodology is very general and can be used to optimize different browsing patterns. We include simulation results that illustrate the performance of our scheme and compare it to the common scheme currently used to organize web objects into web containers.

5.1 Overview

In web archiving, most web pages tend to be small, and are typically aggregated into relatively large containers as the objects are accessed during the crawling process. An emerging standard for such containers is the WARC format [82], which evolved from the ARC container format developed by the Internet Archive, currently the

world's largest internet archive. Moreover, many web crawlers and access tools, Heritrix [55], NutchWAX [60], Wayback [84], WAXToolbar [83] and WERA [87], assume this format.

Given a set of WARC containers that hold an archived linked set of web objects, a future browsing process of the archived objects starts with a web object defined by a seed link, followed by navigation through the linked structure until the desired web object is found. Our goal is to organize the web objects into containers so as to minimize the number of containers needed to complete a typical browsing process. We develop an algorithm that assigns web objects to containers by performing an initial link analysis on the given linked structure, followed by a partitioning process that leads to an efficient solution to this problem. We show that our method enables effective navigation through the archived linked structure and compare its performance to the dominant scheme in use today.

We start in Section 5.2 by describing the previous work related to our problem, followed by developing and justifying our method in Section 5.3. We apply our method to two web site examples and examine the performance gains achieved by our method in Section 5.4. We conclude in Section 5.5.

5.2 Related Work

We review in this section the possible storage formats for archiving web contents and a couple of techniques in link analysis and graph partitioning which will form the core of our method.

5.2.1 Archival Storage

In order to organize and store web objects in an archive, several methods have been proposed and are currently in use. A straightforward method (such as the one implemented in [31]) is based on using the local file system where the target web content is copied object by object to the local file system, maintaining the relative structure among the objects. For future access, the html tag ‘file’ can replace the ‘http’ tag in the original object. We can then use the local file system for navigation through the archived web objects. For example, ‘<http://www.example.org/index.html>’ can be rewritten as ‘<file:///archive/2007.08.01/www.example.org/index.html>’. It is relatively easy to set up and run this type of web archiving and the retrieval process is carried out using local file access mechanisms. However, there are several problems in using this method for web archiving including its limited scalability to what the local file system can handle, and the difficulty to preserve the contents over time as they are tightly coupled to the specific file system. Moreover, this strategy requires modifications to the original contents, and thus the strict faithfulness to the original contents cannot be maintained in most cases [27,26,77].

The second approach extracts documents from the hypertext context and reorganizes them in a different format while setting up different access mechanisms. For example, a small set of web pages can be converted into a single PDF document. However, this strategy makes sense mainly for specific objects that were originally created independently of the web. Although it is possible to maintain the hypertext structure within the converted documents, for the broader range archiving, this approach loses the hypertext structure between multiple such documents.

The most popular method currently in use by most web archives, including the Internet Archive, stores web objects in WARC [82] container files. A WARC file holds a set of harvested web files, each with its own auxiliary metadata. The size of a WARC file can vary up to hundreds of megabytes (usually 100~500MB). Typically, an external indexing server is maintained to provide the mapping between hyperlinks inside a WARC file and the location of the archived object that the hyperlinks point to. For example, if, inside a WARC file, there is a web page archived on September 24, 2007 which has an outgoing hyper link with a tag ``, the indexing server could return in response to the tag and date something like '20070924082031-00007.warc' and '1463539' which are the WARC file name and the offset in the WARC file, respectively. In this chapter, we will also assume that web files are placed in such containers such that a certain upper bound on the size of the container is assumed.

5.2.2 Graph Partitioning Techniques

Web material can be considered as a graph (web graph) where each constituent web page is represented by a vertex, and each incoming/outgoing link corresponds to a directed edge. Once represented as a graph, the web graph can be partitioned into multiple sub-graphs using one of existing graph partitioning techniques. The basic goal of a minimum edge-cut partitioning is to minimize some defined cost on the edges connecting the partitions. There are many ways to define the external cost of graph partitioning but the two notions most widely used are the maximum weight of the edges between vertices which lie on different partitions, and the total weight of all

the edges connecting distinct partitions. Although the graph partitioning problem is known to be NP-complete, many heuristic algorithms have been developed which find very good partitions in practice [8,9,23-25,28,35,39,53,52,67]. However, for our application, we will require additional constraints, which cannot necessarily be handled by many of the well-known graph partitioning algorithms. We review here some of the algorithms that can be used to solve our graph partitioning problem that will be defined formally in Section 5.3.

Perhaps the best known graph partitioning algorithm is the Kernighan-Lin algorithm [39], where the partitioning process starts with an arbitrary partition, and then proceeds to decrease the external cost by a series of interchanges of subsets of the partitions repeatedly until no further improvement is possible. To avoid local optimality, the algorithm is applied repeatedly to obtain a number of locally optimum partitions among which the best partition is chosen. Although Fiduccia and Mattheyses [15] later improved the performance of the Kernighan-Lin algorithm, their algorithm is considered computationally expensive especially if the graph is large, which is clearly the case for our application.

In order to cope with large graphs, researchers devised multilevel graph partitioning schemes [8,9,24,35,67] where the algorithms reduce the size of the graph (or “coarsen” the graph) by collapsing vertices and edges, partition the resulting smaller graph, and then “uncoarsen” it to construct a partition for the original graph. While the multilevel scheme was mainly developed and used to improve the partitioning performance of a large graph at the expense of worse partition quality [67], more recent multilevel algorithms, such as in [8,9,24,35], further refine the

partition during the uncoarsening phase, thus obtaining a partition quality that is comparable or even better than other existing techniques [28]. The Kerningham-Lin algorithm is often used as the refinement algorithm.

5.2.3 Link Analysis Technique – PageRank

PageRank [61] is a link analysis algorithm that assigns a numerical weight to each element of a hyperlinked set of documents, such as web material. Intuitively, a web page with a higher PageRank should have a higher probability of being visited. The intuition behind PageRank is that if page u has a link to page v , then page u is implicitly conferring some importance to page v . In other words, page u can be thought as voting for page v . The more votes a page receives, the more important it is considered. However, not every vote counts equally: votes cast by pages that are themselves “important” weigh more heavily and help other pages become more “important”.

In the ideal model, the PageRank value for page u , $PR(u)$, can be expressed as:

$$PR(u) = \sum_{v \in I_u} p_{vu} PR(v),$$

where I_u is the set of pages with links to page u , and P_{vu} is the probability that a random surfer visiting page v jumps to page u . Since it is not possible to know the exact value of P_{vu} , P_{vu} is usually set to $1/out_degree(v)$, that is, all outgoing links from v are assumed to be equally likely.

However, the ideal model has two problems. The first problem is the presence of dangling pages that shut the surfer when visited. A solution to the problem is to patch dangling pages by artificially placing outgoing links from each dangling page to all

the other pages. Each artificial link can be given either equal probability of $1/N$ (N : total number of pages), or personalized probability which records a generic surfer's preference for each page. The second problem with the ideal model is that the surfer can get trapped by a cyclic path in the web graph. Brin and Page [7] suggest enforcing irreducibility by adding a new set of artificial transitions that, with low probability, jump to all nodes. Mathematically, this corresponds to the following equation:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in I_u} p_{vu} PR(v),$$

where N is the total number of pages, and d is the probability the random surfer jumps to a random page without a link.

We note that this equation is slightly different from the original PageRank equation as proposed by Brin and Page [7]. The original equation, $PR(u) = 1 - d + d \sum_{v \in I_u} p_{vu} PR(v)$, has brought up some confusion since, unlike the inventors' claim, the sum of all PageRanks is not one, but N . The above scaled version, however, leads to $\sum PR(v) = 1$, and each PageRank can be thought as a probability. In the above equation, the parameter d is called the damping factor which can be set somewhere between 0 and 1. As suggested in [61] and [7], we use $d = 0.85$ in our work which will be further described in the next section.

If we let $G = (V, E)$ be a web graph, and \mathbf{A} the modified adjacency matrix of G defined by:

$$A_{ij} = \begin{cases} \frac{1-d}{N} + \frac{d}{O_j}, & \text{if } (j,i) \in E \\ \frac{1-d}{N}, & \text{otherwise} \end{cases},$$

where O_j is the number of out-links from page j .

If we also let \mathbf{P} be an N -dimensional column vector of PageRank values, then \mathbf{P} can be expressed by the following matrix equation: $\mathbf{P} = \mathbf{A}\mathbf{P}$

This is the characteristic equation of the eigensystem whose solution is the eigenvector corresponding to the eigenvalue of one. Furthermore, \mathbf{A} can be considered as a stochastic matrix that is also irreducible and aperiodic, due to the modifications we performed earlier to avoid dangling nodes and cyclic paths. Therefore, by the Ergodic theorem of Markov chains [63], a finite Markov chain defined by the stochastic transition matrix \mathbf{A} has a unique stationary probability distribution. This implies that, starting with any initial value of \mathbf{P} , we can iterate the application of the matrix \mathbf{A} to \mathbf{P} , and \mathbf{P} will converge to a steady-state probability vector, which in turn is the eigenvector of \mathbf{A} corresponding to the eigenvalue of one. In practice, a well known mathematical technique called power iteration [20] can be used to efficiently determine \mathbf{P} .

As will be discussed further in the next section, our link analysis technique is based on the PageRank algorithm. However, unlike the PageRank algorithm that assigns a weight to each page, we assign a weight to each link, which will then be used to partition the graph.

5.3 Our Method

As discussed earlier, the most popular storage method for web archiving is to use containers where each container holds a number of web pages. Typically, web material is archived using many containers. The primary goal of our work is to develop techniques to allocate web pages to containers such that each container has as closely related web pages as possible, thereby minimizing the chances of accessing many different containers when a user browses through the archived web material. When web contents are archived in the form of multiple containers, we can view these containers as a coarsened web graph (or container graph) where the original nodes within the same container are collapsed together to form a super node, and only edges between different containers survive with assigned weights as will be explained next.

In the container graph, $G_c=(V_c, E_c)$, we define the cost of the edge-cut, EC , as follows:

$$EC = \sum_{e \in E_c} w_e,$$

where w_e is the weight of edge e .

In order to accomplish our goal, we analyze the link structure within the web material to be archived to find, for each edge, a good estimate of the probability that the edge will be taken. Using this estimate as the edge weight, we partition the web graph in such a way as to minimize EC . The following two subsections discuss our link analysis and the partitioning technique used to minimize EC .

5.3.1 Edge Weight

Edge weights should represent the relative likelihood of an edge being taken during a browsing session. In the live web, edges are hyperlinks embedded in web pages, but in an archive, edges in the web graph can also exist between two consecutive versions of a web page. In order to assign edge weights, a link analysis is performed. Before proceeding we note that our scheme will be based on a browsing pattern similar to what is expected in today's live web. However it is easy to accommodate other access patterns within our methodology using a different weight function on the edges. For example, should browsing of successive versions of a web page dominate, we will assign heavy weights to the corresponding edges relative to the remaining edges. Similarly, should the access pattern to sub-domains of sites dominate, the corresponding edge weights will be assigned high values. For the rest of this chapter, we are assuming an access pattern similar to the one currently encountered on the live web.

We start with some simple observations. If a vertex has only one outgoing edge, this edge will be more likely taken than an edge from another vertex with many out-links, and thus should be weighed more heavily. A possible simple solution is to assign edge weights depending on the number of out-links of the source vertex. For instance, if the source vertex of edge e has k outgoing edges, the weight of $1/k$ is given to edge e .

When a personalized vector is not in use, the PageRank algorithm also uses the same method in assigning edge weights. In this case, the only deciding factor to the edge weight is the number of the outgoing edges from the source vertex, and thus the

edge weight only represents the local probability of the edge being taken, once the source vertex is visited. In other words, the edge weight is only locally meaningful, and thus it is not possible to say that an edge is more likely to be taken than the other if they belong to different vertices.

For our method, the probability of each vertex being visited is computed first using the PageRank algorithm. The PageRank value (or steady-state probability) of each vertex is then divided by the number of outgoing edges from the vertex. We call this quotient EdgeRank (ER) and assign the same EdgeRank value as the weight to every edge coming out from the same vertex.

$$ER(e) = \frac{PR(v)}{OD(v)},$$

where vertex v is the source vertex of edge e , and $OD(v)$ is the out-degree of vertex v .

Note that, because $\sum PR(v) = 1$, $\sum ER(e) = 1$ too.

Now that we have an edge-weighted graph representing our web contents, the allocation of web pages to containers is performed using a graph partitioning algorithm.

5.3.2 Graph Partitioning

As discussed in Section 5.2, there are a number of existing min-cut graph partitioning heuristics that seem to work well in practice. Although their primary partitioning criterion is to minimize the cost of the edge-cut, they differ from one another in input, output, and partitioning parameters. For example, some algorithms

support size-constrained partitioning while others do not. Also, not all algorithms support weighted vertices and edges. Before proceeding let's define our graph partitioning problem more formally.

Web Graph Partitioning Problem: *Given a directed web graph $G:(V, E)$ with weighted nodes (weight of a node is the size of the corresponding page) and weighted edges, determine a partition $V = P_1 \cup P_2 \cup P_3 \cup \dots \cup P_n$ such that,*

1. The sum of the weights of the edges that connect any two different partitions is minimized.

2. For all i 's, $|P_i| \leq K$ for some fixed K , where $|P_i|$ is the sum of the weights of the vertices in the partition and K is an upper bound on the size of a container.

The first condition is shared by almost all partitioning algorithms (some require non-weighted edges), while the second condition, which is the size constraint imposed to every partition, is supported only by a few partitioning algorithms (such as [28,36,35]), sometimes with a slight modification.

In this work, we adopt the multilevel graph partitioning algorithm to solve our problem. The primary reason is that it supports the constraints on the partition size; moreover, the method is fast, which is important in our case considering the typically large sizes of web graphs. In particular, we adapt a partitioning technique suggested by Karypis and Kumar [35] as follows.

Their scheme first computes a maximal matching using a randomized algorithm, and coarsens the graph by collapsing the matched vertices together. This coarsening step is repeated until a desired size of the coarsened graph is achieved. Once the

graph is coarsened, the minimum edge-cut bisection is computed using some of existing algorithms such as spectral bisection [3,68], geometric bisection [53] or combinatorial methods [16,17,39]. The partitioned graph is then refined and uncoarsened. The improved Kernighan-Lin algorithm that was developed by Karypis and Kumar is applied to this uncoarsening-with-refinement phase.

In particular, we use Metis [79], a partitioning tool that implements the Karypis-Kumar scheme. Although Metis does not explicitly support the partition size constraints – our second condition, it does support vertex-weight-based size balancing among partitions, making the size of all partitions similar. Therefore, based on the sum of all the vertex weights of the web graph, we pre-compute the necessary number of partitions before running the partitioning tool, so that the resulting partitions will meet the second condition.

5.4 Experimental Evaluation of Our Scheme

In order to examine the performance of our algorithm in terms of the number of containers accessed during a typical browsing session, we consider two datasets. The first is the web graph of the University of Maryland Institute for Advanced Computer Studies (UMIACS) web site, located at <http://umiacs.umd.edu> domain, which we call the UMIACS web graph. We crawled every web page within five-hop distance (or depth) under this domain, and constructed the web graph corresponding to this crawling. The second dataset is the Stanford web graph which was generated from a crawl of the stanford.edu domain created in September 2002 by the Stanford WebBase project [29], and is widely used by the web graph analysis community.

Unlike the first dataset, the Stanford web graph has neither the size information of vertices, nor the actual URLs with which we might have been able to obtain estimates of the web pages (which undoubtedly have changed since then). Consequently, we randomly assign vertex sizes using two Gaussian distributions – one for html files, the other for non-html files. Their parameters are based on the findings from a web statistics study [30]. In particular, we assumed there are about 18% html objects by total file size, and the average html file size is 605 KB. This size modeling is not intended to mimic the actual web object sizes in the Stanford web page. Rather, we intend to assign some reasonable sizes to run our experiments. Note that the quality of our method does not depend on the accuracy of the vertex sizes. Table 5.1 describes these two datasets.

Table 5.1: The Two Datasets Used for Evaluating Our Method

Datasets	# Vertices	# Edges	Total Vertex Weight
UMIACS Web Graph	4579	9732	2.49GB
Stanford Web Graph	281903	2312497	215.82GB

In our experiments, we allocate pages to containers (or WARC files) in three different ways.

- CONV: Pages are allocated to containers as they are fetched during the crawling process. Once a container is full, we use a new container (Figure 5.1).
- GP: The graph partitioning technique is applied so as to minimize the number of edges connecting any two partitions. All the pages belonging

to a partition are allocated to a single container (Line 3 in Figure 5.2 is omitted).

- ER+GP: The EdgeRank technique is used to assign weights to edges (Line 3 in Figure 5.2), and the graph is partitioned using a minimum-weight partitioning algorithm. Again, containers are constructed based on the resulting partitions. In each case, the damping factor, $d = 0.85$, is used in EdgeRank.

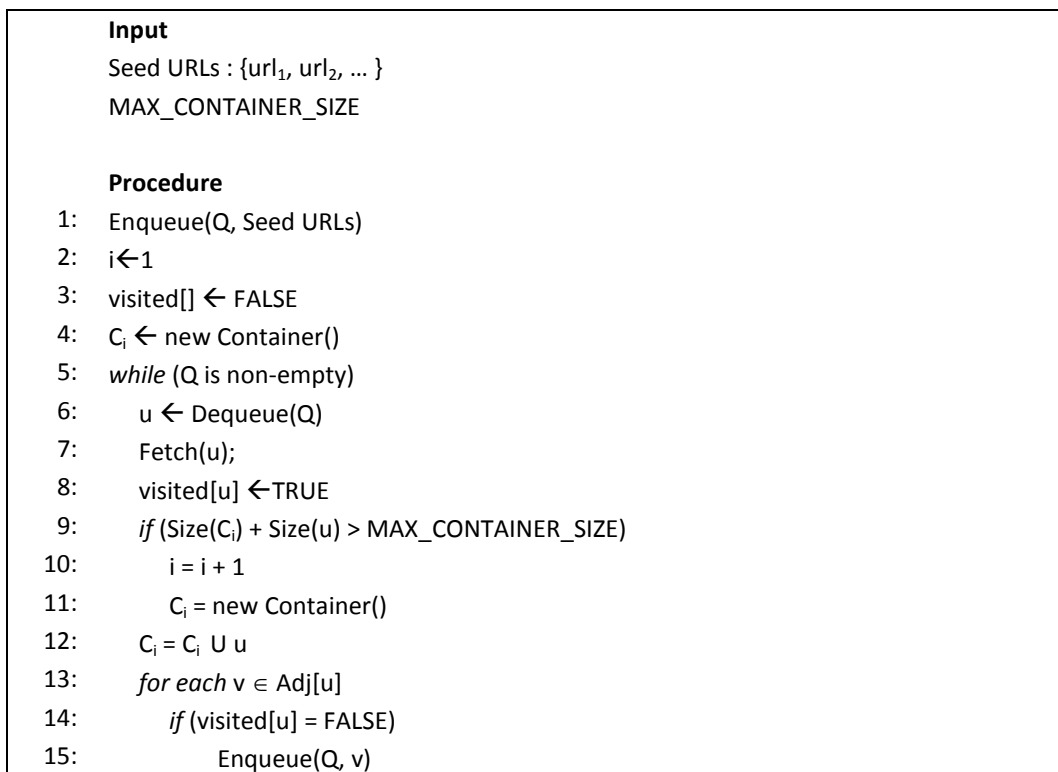


Figure 5.1: Conventional Allocation of Pages to Containers

Figure 5.1 shows a typical BFS algorithm where a visited node is stored in the current container as long as the size of the resulting container does not exceed the predefined value (MAX_CONTAINER_SIZE) (Lines 9~12). A new container is created if necessary.

In the algorithm shown in Figure 5.2, a web graph is first built (Line 1) using a BFS-based crawling algorithm similar to the one in Figure 5.1, followed optionally by computing EdgeRank (Line 3) in order to obtain edge weights in the graph. This graph is then partitioned into the pre-calculated (Line 2) number (n) of partitions. This number depends on the total sum of vertex weights (page sizes) in the graph, as well as the predefined maximum container size (MAX_CONTAINER_SIZE). Once partitioned, the URLs in each partition are re-visited and packaged in the n containers (Lines 5~9). In practice, depending on the resource availability, the web objects downloaded from the previous crawl (Line 1) can be stored and reused in the packaging process.

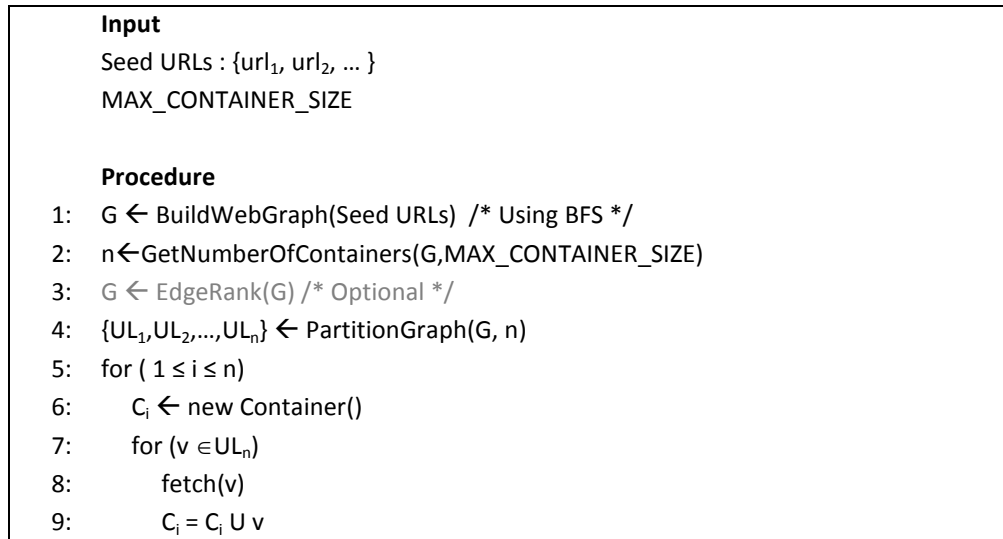


Figure 5.2: Container Construction Based on Graph Partitioning with or without EdgeRank (Line 3)

In our simulation, to be discussed in Section 5.4.2, the UMIACS dataset was partitioned into 25 partitions and the Stanford dataset was partitioned into 2200 partitions, resulting in the size of each partition being between 100MB and 200MB.

5.4.1 Edge-Cut

In order to evaluate the graph partitioning performance, we measure the edge-cut obtained from the graph partitioning scheme, and compare it to the conventional breadth-first-search (BFS) partitioning. We defined the cost of an edge-cut earlier as the sum of the weights of all the external edges between partitions. However, as we performed the experiments on the two separate datasets with different numbers of nodes, edges and partitions, we scaled down the cost of the edge-cut to a web graph with the total edge weight of 100, as follows:

$$EC_{scaled} = \frac{EC \times 100}{|E|}, \text{ where } |E| \text{ is the total edge weights in the web graph.}$$

We begin by considering the case where the web graph has no edge weight (or equal edge weight). We observe that the edge-cuts generated by the conventional method were about 70~80 for both datasets while those generated by the graph partitioning scheme are 12 and 47 for the UMIACS and Stanford datasets respectively. Using edge weights based on the PageRank technique, the graph partitioning approach similarly reduces the costs of the edge-cuts relative to the conventional approach as illustrated in Table 5.2.

Table 5.2: Edge-Cut Results

Edge-Cut	Unweighted Edges		Weighted Edges	
	CONV	GP	ER+CONV	ER+GP
UMIACS Web Graph	73.87	12.38	62.36	36.03
Stanford Web Graph	80.50	47.33	63.56	32.20

5.4.2 Simulation

Although the edge-cut figures show favorable results when the partitioning technique is employed, we additionally ran simulations to further see how much the partitioning and the EdgeRank will in fact reduce the number of containers necessary for a random user to browse through the archived web material. In these simulations, we set a virtual user who randomly walks through links, and counted the number of containers that the user had to access.

Table 5.3: Simulation Parameters

Parameter	Value
Number of Hops	10
Probability of Going Back	30%
Outdegree of Starting Vertex	> 5
Policy At Dangling Vertex	Go back

Each random walk consists of ten random hops, and at each random hop, each outgoing link is given an equal probability of being taken. Also, we assume that the BACK button on a browser is pressed with 30% probability. We base this choice on a recent browser usage research [38] which shows that hyperlinks are taken 41.7% of time, followed by other navigation (23.6%) and the back button (18.9%). Since, in our simulation, we only consider hyperlinks and the back button, we assume that the back button is pressed about 30% ($\approx 18.9 / (41.7 + 18.9)$) of the time. Once the random walk reaches a vertex with no outgoing link (or a dangling), the random walk goes back to the previous vertex, if any, as if the user presses the BACK button. In order to avoid the situation where there are no more vertices left to visit soon after the start of the simulation, we insist that the randomly selected starting vertex has an out-

degree of five or larger. This is achieved through repeating the random selection process until we find a starting vertex that meets this criterion. Table 5.3 shows our parameter-value pairs used in the simulations.

In the simulations, we ran the random walk 1000 times over each dataset, where we monitored both the number of inter-container hops and the number of distinct containers needed for each random walk. Inter-container hops occur whenever a different container needs to be accessed. For example, if a random walk switches back and forth between two containers, A and B, ten times, the number of inter-container hops will be ten, while the total number of distinct containers is only two. In a system with no caching policy or a limited memory, the inter-container hops will serve as a more useful metric because, even if a user requests a previously retrieved container, the system will always need to retrieve it from storage. However, if a system can cache enough containers, the total number of distinct containers will make more sense in assessing the system's performance. Figures 5.3 and 5.4 show the histograms of the number of inter-container hops and distinct containers accessed for the UMIACS web graph, respectively, while Figures 5.5 and 5.6 show the corresponding histograms for the Stanford web graph. In these histograms, we categorized 1000 random walks by the number of containers that each random walk was required to access. The X-axis represents eleven categories (0, 1, ..., 10 ; Note that the total number of hops in each random walk is ten, so there can be at most ten inter-container hops in worst case), whereas the Y-axis represents the number of random walks that fall into each category.

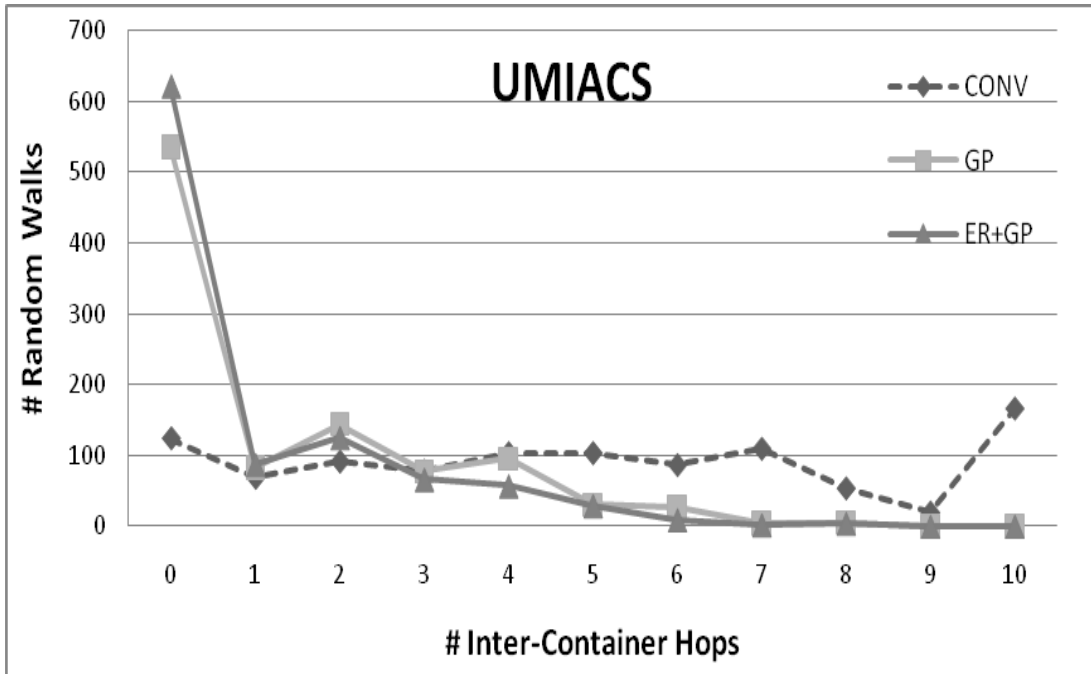


Figure 5.3: Histogram of Number of Inter-Container Hops for UMIACS Web Graph

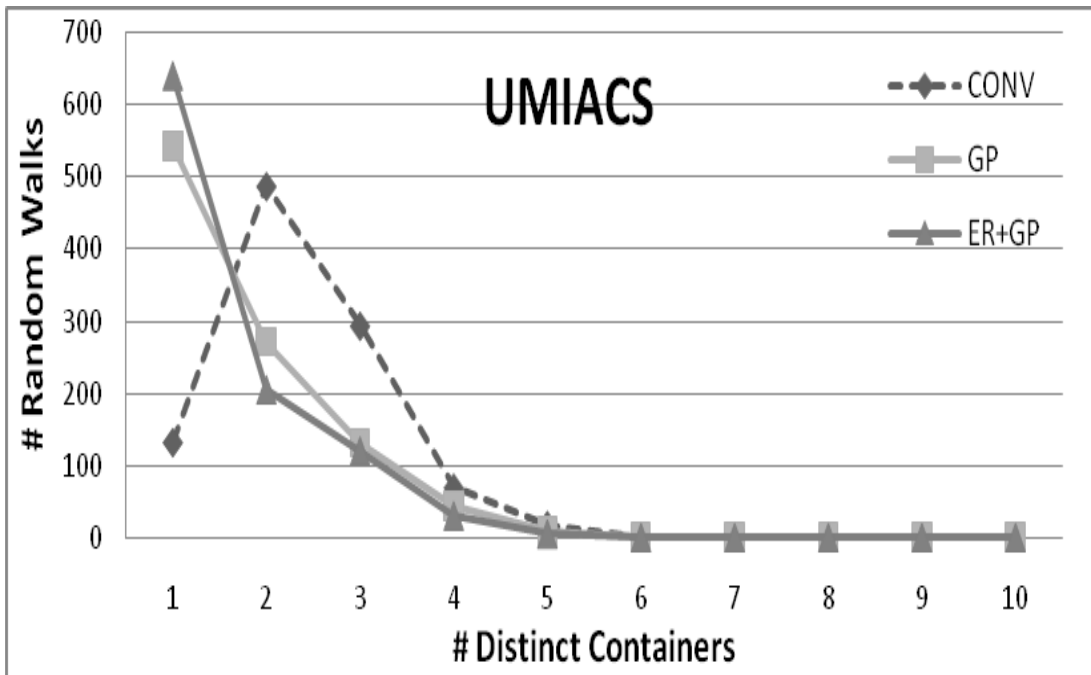


Figure 5.4: Histogram of Number of Distinct Containers Accessed for UMIACS Web Graph

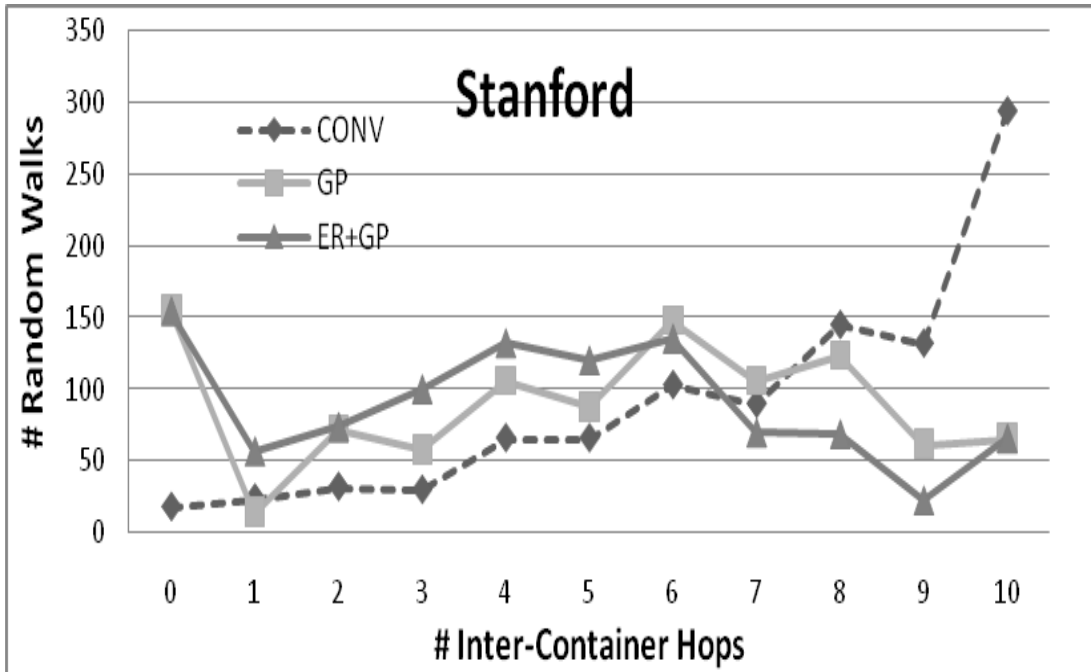


Figure 5.5: Histogram of Number of Inter-Container Hops for Stanford Web Graph

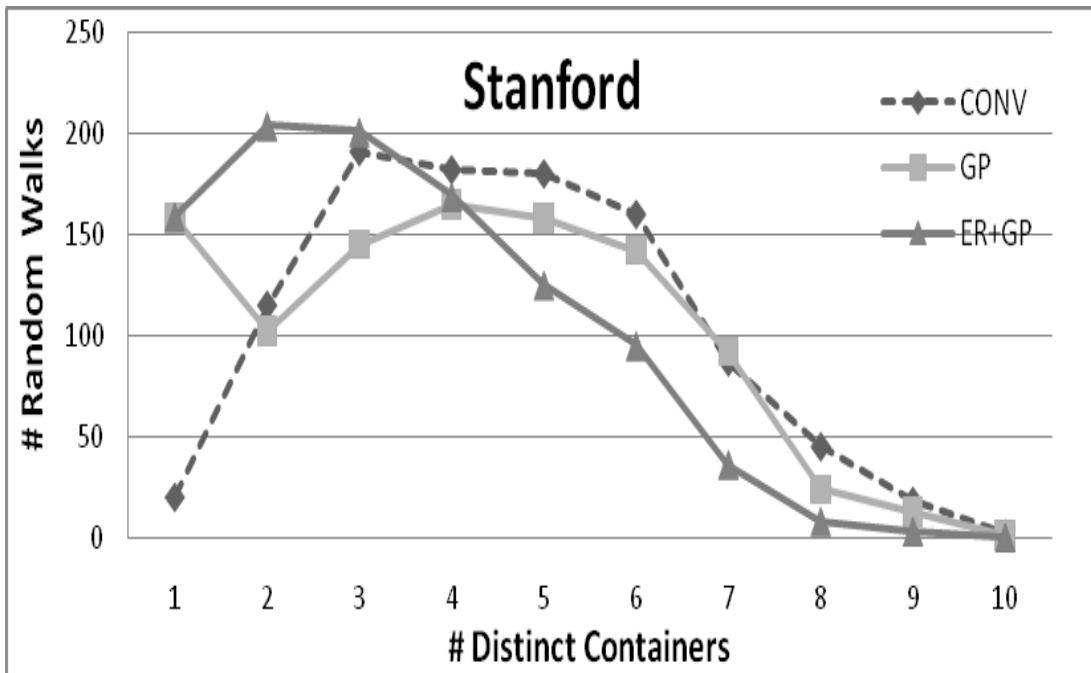


Figure 5.6: Histogram of Number of Distinct Containers Accessed for Stanford Web Graph

It can be observed that when the graph partitioning scheme is used, many random walks only need a single container (thus, zero inter-container hops). Figure 5.7 depicts the average number of inter-container hops during the random walks over the two web graphs. From the figure, it can be seen that the GP and ER+GP schemes reduced the average number of inter-container hops from five to one for the UMIACS web graph. For the Stanford web graph, the GP scheme reduced the number from seven to five, while ER+GP further reduced the number down to four. The average number of containers needed is shown in Figure 5.8. Although the improvements are not as dramatic as the number of inter-container hops, compared to the CONV scheme, the GP scheme required about 28% and 11% less number of distinct containers for the UMIACS and Stanford web graph, respectively. The ER+GP scheme further reduced the numbers 9% and 17% less than those from the GP scheme.

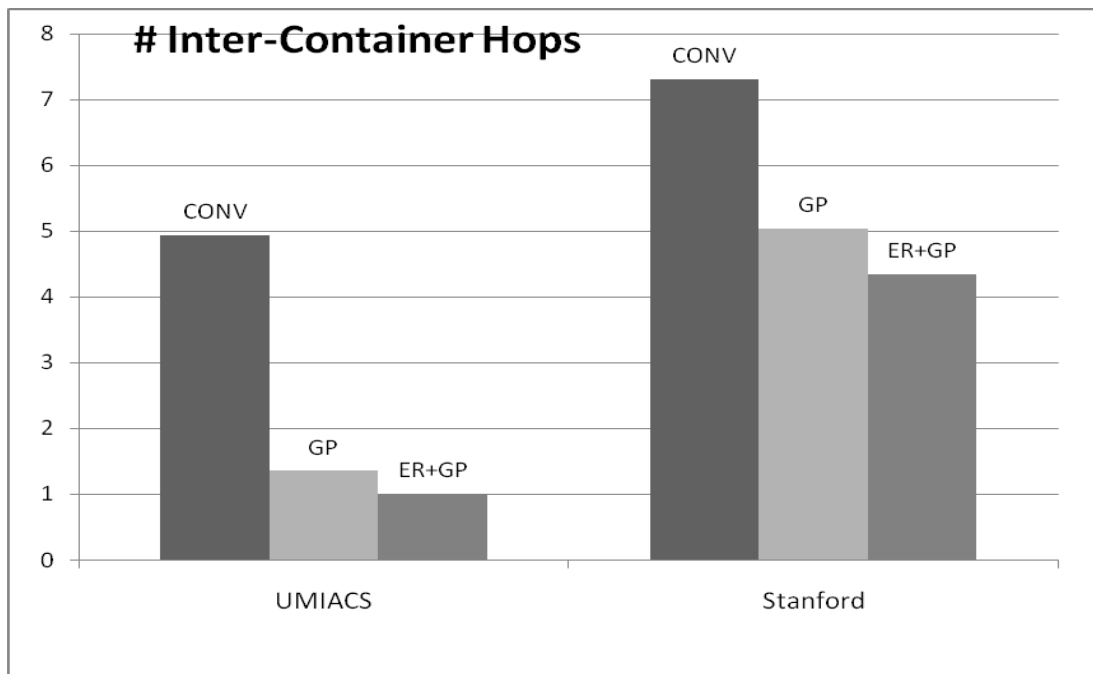


Figure 5.7: Average Number of Inter-Container Hops

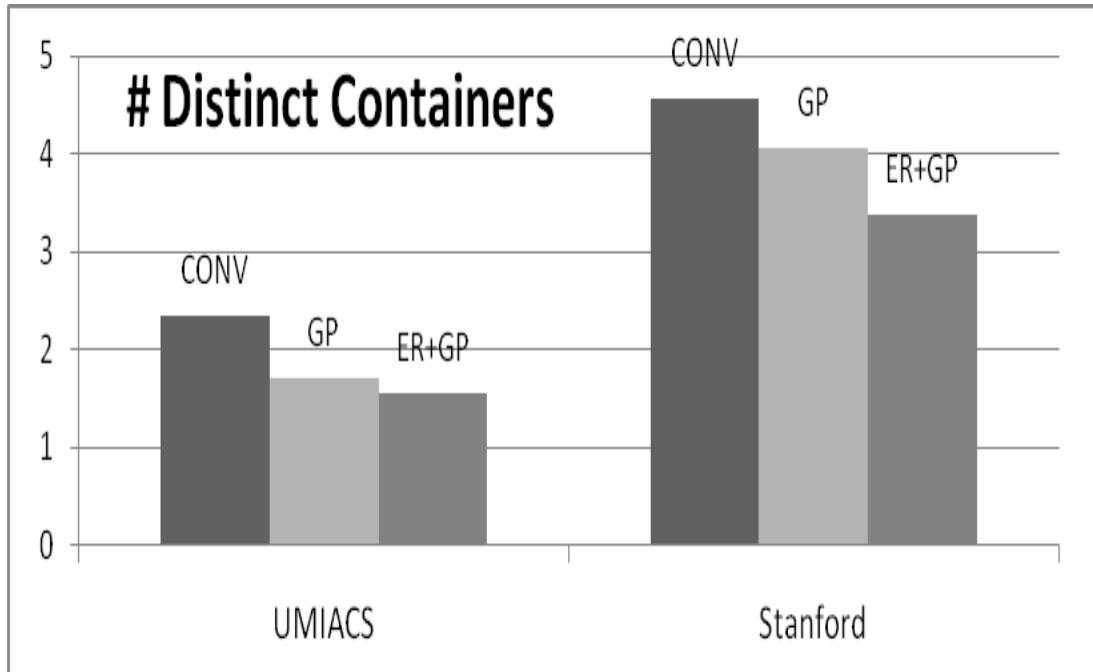


Figure 5.8: Average Number of Distinct Containers

In these experiments, based on our assumption about the access pattern discussed in Section 5.3.1, we only considered the cases where users follow hyperlinks on pages. We note that the link analysis scheme can be tailored to capture a number of access patterns by adjusting the weight function appropriately. The corresponding partitioning technique will optimize the allocation of web pages to containers so that the average number of containers accessed is minimized.

5.5 Conclusion

In this chapter, we have shown that a graph partitioning scheme for organizing archive containers significantly reduces the number of containers that need to be accessed when a user browses through the archived web material. Also shown was a PageRank-derived technique, called EdgeRank, which can improve this number even further. The overhead required by this technique is relatively small. For instance, on

our 2 Ghz Intel Core 2 Duo processor, we could fully partition and compute EdgeRank of a large graph (the Stanford web graph that contains about 300,000 vertices, and 2.3 million edges) within minutes.

Chapter 6

Concluding Remarks and Future Work

In this dissertation, we examined the topic of long-term information preservation and access from three different perspectives.

First, we introduced a methodology to ensure the authenticity of the preserved contents on a long-term basis. Based on efficient and rigorous techniques, our integrity monitoring scheme detects with high probability any errors in an archive including those introduced by malicious alterations. Our scheme allows an independent third-party to audit any object in the archive and certify its integrity, as long as a small size of cryptographic information is kept intact. The current implementation of our scheme, ACE, has been deployed in a number of institutions that are using it to monitor the integrity of their digital holdings.

Second, we developed an information retrieval and content delivery scheme. Based on the notion of time windows, our information retrieval scheme allows an efficient handling of full-text search with temporal constraints by minimizing the search space according to the query time span. It also allows document relevancy scoring to be based on the temporal context of the query time span. An analytical model was introduced to determine the best time window size, and validated against two real-world datasets of significant size. The empirical study strongly supported our analytical model. For content delivery, we proposed a persistent data structure, called PISA, to be used as a location index that identifies the location of a document version.

The asymptotic analysis shows that PISA is as good or better at insert and query operations, compared to existing schemes.

Third, we developed a data layout scheme to organize inter-related data such that future navigation through the preserved data can be efficiently performed. In particular, we addressed the problem of how to organize inter-related objects such that future navigations through the objects can be more efficiently performed. Based on a graph partitioning technique combined with a graph analysis technique, our scheme showed significant performance improvement over conventional schemes.

The issues examined in this dissertation represent a small sample of the challenging problems facing long term preservation. We discuss a few of these below.

In Chapter 2, we examined integrity preservation where the focus was given to ensuring the bit-level integrity of the preserved data. A promising future direction is to extend this to categorical integrity preservation, where the actual readability of the preserved data is also ensured. The categorical integrity preservation will need to also consider file format obsolescence. We note that we conducted a preliminary study on the issue of file format obsolescence [18]. While this is a study that we performed independently of the integrity preservation work, a careful adaptation of such work may provide a useful component toward the goal. Meanwhile, it will also be interesting to extend ACE auditing to the cloud computing environment. A possible deployment scenario of ACE in the cloud will involve storing integrity tokens in the cloud, extending cloud tools to support retrieval of and validation using the integrity tokens, and also developing a set of audit services.

In Chapter 3, we assumed that the time windows have the same size in our analytical model. An interesting future work will be to extend the approach to dynamic time window sizes. A possible starting point involves a periodic evaluation for the best time window size over some period of time, say a year or two. We also assumed uniformly random query time spans when building our analytical model, as well as when we conducted empirical studies. However, depending on the nature of the collection, different access patterns may be more reasonable. For instance, one might expect time point queries to be prevalent for a certain type of collections. Should we have better knowledge about such access patterns available, we will be able to obtain a better estimate for the best time window size. Yet another research area related to information retrieval will be to include the spatial aspect of preserved contents. Spatial information retrieval or spatial and temporal information retrieval should generate various interesting and challenging issues that will have a large number of applications.

We introduced a location index for time-evolving data in Chapter 4, where we showed asymptotic analysis for the insert and query performance. Although an initial real-world performance comparison between PISA and B⁺-Tree has been already performed in our paper [74], another set of real-world performance comparison against other multi-version access structures will also be interesting future work.

Experiments on the link-analysis and graph partitioning techniques introduced in Chapter 5 may be more generally extended to other types of digital objects, not just web data. More importantly, the scalability of the scheme will need to be given considerable attention. We expect the collection size to steadily increase in the long-

term, and therefore, the scalability aspect of the graph analysis and partitioning techniques is a critical problem that needs to be addressed.

Appendix A

PISA: Parameters and Invariants

In Chapter 4, we assumed that Invariant 1 and Invariant 2 always hold under only one parameter condition: all parameters are positive integers. Since these conditions are obviously not enough to maintain the invariants, we now look for stronger conditions that indeed satisfy both invariants, and based on the new conditions, we show how we can determine parameter values and what implications they have on the performance.

A.1 Parameter Conditions

We begin by making the first self-evident parameter condition.

Condition 1: $B_{min} \geq 3$

Now, we show how to maintain Invariant 1: For *any live* block in PISA, $3 \leq B_{min} \leq n_l \leq n_a \leq B_{max}$.

First of all, $3 \leq B_{min}$ holds with Condition 1. Also, since the inequalities $n_l \leq n_a \leq B_{max}$ are automatically maintained by the definitions of n_l and n_a , we only need to maintain $B_{min} \leq n_l$.

Note first that there are only two cases for an entry in any existing block to become dead: 1) an updated entry with the same key is inserted into the block, or 2) an overflow occurs and a version-split is performed in the block. In the former case, the number of live entries does not change, since a new live entry always replaces the

dead entry. In the latter case, the overflowing block becomes dead, so we do not have to deal with it anymore (Invariant 1 concerns only *live* blocks).

We now examine whether updated blocks (either from key splits or version splits) satisfy $B_{min} \leq n_l$. By updated blocks, we include both the newly created block and the original block that a key-split moves entries out from. By the result of a key split, at least B_{min} live blocks remain in the original block, and also at least B_{min} live blocks are moved to the newly created block. Thus, key splits do not violate $B_{min} \leq n_l$. In case of a version split, the new block after the version split contains the same number of live entries (plus one new entry that caused the overflow), as the original block. Thus, as long as the original block maintained $B_{min} \leq n_l$ while it was alive, the inequality still holds for the new block.

Consequently, we only need to make sure that the very initial block satisfies $B_{min} \leq n_l$. One easy way of ensuring this is to have an initial block which we artificially insert B_{min} dummy (but live) entries into in the beginning. Recall that we also took this initial block into consideration when we counted the number of blocks in the previous section.

We now look for a condition to maintain Invariant 2: For *any new* block in PISA, $3 \leq B_{min} \leq n_l \leq n_a \leq (B_{max} - B_{min} + 1)$.

Again, $3 \leq B_{min}$ holds by Condition 1, and $n_l \leq n_a$ always holds by the definitions of the variables. Moreover, since a new block is also a live block, and we already saw that any live block can maintain $B_{min} \leq n_l$ with initial dummy entries, we only need to investigate the last inequality $n_a \leq (B_{max} - B_{min} + 1)$, for an updated block.

In order to evaluate the upper bound of the number of entries ($n_a \leq B_{max} - B_{min} + 1$), we first look at a key split. Sometimes, a key split is preceded by a version split when a qualifying median entry cannot be found. However, it is sufficient here to only consider the key split procedures after a median entry is found. The case with the version split will be separately discussed later. Just before a key split occurs on block A, there are $B_{max} + 1$ entries (or less if a version split had to be preceded) in block A of which at least $4B_{min}$ are alive. After the key split, somewhere between B_{min} and $3B_{min}$ live entries are moved out to a new block B. Some qualifying dead entries in block A are also copied into block B. As a result, block B can have as many as $3B_{min}$ live entries and $B_{max} + 1 - 4B_{min}$ dead entries, making a total of $B_{max} - B_{min} + 1$ at most. Therefore, block B clearly maintains the upper bound of $n_a (\leq B_{max} - B_{min} + 1)$. As for block A, the maximum number of entries it can contain after a key split occurs when only B_{min} live entries were moved out and it has the greatest possible number of dead entries (i.e. $B_{max} + 1 - 4B_{min}$ dead entries). Even in this worst case, block A may only contain $3B_{min} + B_{max} + 1 - 4B_{min} = B_{max} - B_{min} + 1$ entries, clearly maintaining the upper bound of n_a .

We now move onto the case of a version split. After a version split on an overflowing block A, a new block B contains all the previously live entries in the overflowing block A and the newly inserted entry. Thus, there are $n_l + 1$ entries in the new block B, which we require to be less than or equal to $B_{max} - B_{min} + 1$. Since $n_l + 1 < 4 \times B_{min}$ (the version-split condition), $n_l + 1 \leq B_{max} - B_{min} + 1$ holds if $4 \times B_{min} \leq B_{max} - B_{min} + 1$, which can be rewritten as our second condition below.

Condition 2: $B_{min} \leq B_{max} / 5$

To summarize, if the two parameters values (B_{min} and B_{max}) meet the two conditions above (Condition 1 and Condition 2), the two invariants (Invariant 1 and Invariant 2) indeed hold. In the following subsection, we further examine how to determine the parameter values.

A.2 Determining Parameter Values

In Chapter 4, we analyzed the operation and space performance, where we saw the time complexity for QUERY and INSERT operations increases logarithmically with the logarithm base of B_{min} . Hence, the larger B_{min} we set, the faster these operations become. Furthermore, we also saw that $(2N / (B_{min} - 2)) + 2$ block spaces are required for N INSERT operations. Thus, the larger B_{min} we set, the less space we will need. As far as B_{max} is concerned, in most cases, we do not have many options but to set it as a fixed value depending on the physical block size of the underlying storage. Therefore, it is always better to set B_{min} to the greatest value allowed. Clearly, setting $B_{min} = B_{max} / 5$ gives us the best performance both in time and in space.

Appendix B

Proofs of $E[X]$ and $E[Y]$

In this appendix, we will provide proofs of the claims made in Section 3.4.1. regarding the expected values of X and Y , where X is the number of duplicate versions falling within a set of consecutive time windows that overlap with the query time span and Y is the number of document versions that have to be filtered out relative to the same query time span. Our basic assumption is that the query time span $[q_s, q_f]$ is selected randomly – that is, each end point is selected randomly from the n time steps t_1, t_2, \dots, t_n , the smaller of which will become q_s and the other will become q_f . Hence, for two fixed values $t_s \neq t_f$, $[t_s, t_f]$ will be selected with probability $2/n^2$ and a point query at t_s , for any fixed t_f , will be selected with probability $1/n^2$. Also, $[t_s, t_f]$ spans across more than one time window with probability of $2z^2/n^2$, and lies within a single time window with probability of z^2/n^2 . We start by estimating the expected value $E[X]$ of X , and then derive the expected value $E[Y]$ of Y .

Let δ_i be the number of document versions whose validity time intervals contain t_i and let δ be the average of all the δ_i 's. $E[X]$ can be expressed as follows.

$$\begin{aligned} E[X] &= \sum_{i,j} \text{P}[\text{query spans } T_i \text{ through } T_j](\delta_{iz} + \delta_{(i+1)z} + \dots + \delta_{(j-1)z}) \\ &= \sum_{i=1}^{k-1} \sum_{j=i+1}^k \frac{2z^2}{n^2} (\delta_{iz} + \delta_{(i+1)z} + \dots + \delta_{(j-1)z}) \quad (\text{note that no duplicates exist for } i = j) \\ &= \frac{2z^2}{n^2} \sum_{i=1}^{k-1} i(k-i)\delta_{iz} \end{aligned}$$

We can substitute δ instead of the individual δ_i 's to approximate $E[X]$ as follows.

$$E[X] \approx \frac{(k-1)^2}{3k} \delta$$

For sufficiently large k , $E[X]$ can be further approximated by:

$$E[X] \approx \frac{k}{3} \delta = \frac{n}{3z} \delta$$

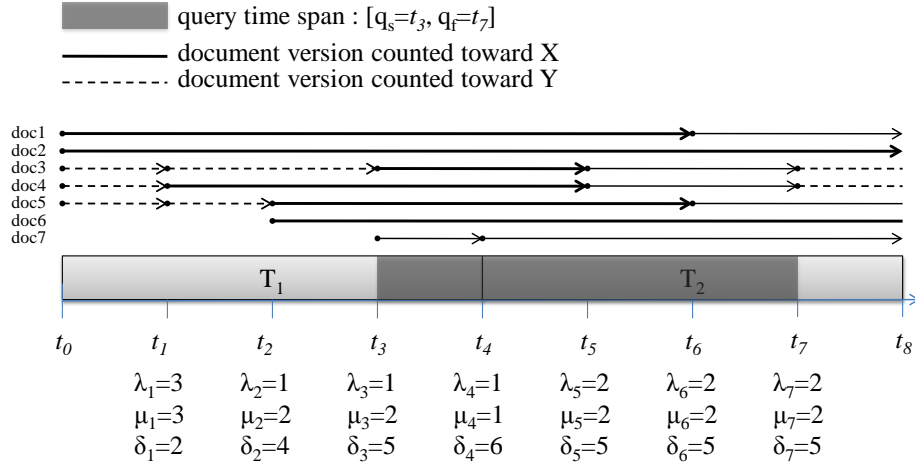


Figure B.1: Illustration of Values of λ_i , μ_i and δ_i

Estimating $E[Y]$ is a bit harder. Let λ_i be the number of document versions whose validity time intervals ends at t_i , that is, a new version is created or document is deleted at t_i , and let μ_i be the number of document versions whose validity time intervals start at t_i . For a randomly selected query time span $[t_s, t_f]$, the number $Y_{s,f}$ of document versions whose time intervals do not overlap with $[t_s, t_f]$ is given by

$$Y_{s,f} = \lambda_{p_s z + 1} + \lambda_{p_s z + 2} + \dots + \lambda_{p_s z + r_s} + \mu_{p_f z + r_f} + \mu_{p_f z + r_f + 1} + \dots + \mu_{(p_f + 1)z - 1},$$

where p_s (or p_f) and r_s (r_f) are defined respectively as the quotient and remainder when s (f) is divided by z .

Therefore the expected value of Y is given by:

$$E[Y] = \sum_{i,j} \text{P}[\text{query spans } T_i \text{ through } T_j \mid i \neq j] \times Y_{s \in T_i, f \in T_j} + \sum_i \text{P}[\text{query within } T_i] \times Y_{s, f \in T_i}$$

$$E[Y] = \frac{2z^2}{n^2} \sum_{i=1}^{k-1} \sum_{j=i+1}^k Y_{s \in T_i, f \in T_j} + \frac{z^2}{n^2} \sum_{i=1}^k Y_{s, f \in T_i}$$

The term $Y_{s \in T_i, f \in T_j}$ and $Y_{s, f \in T_i}$ are also random variables that depend on query time

span $[t_s, t_f]$, and whose expected values can be shown to be equal to:

$$E[Y_{s \in T_i, f \in T_j}] = \frac{1}{z} \sum_{l=1}^{z-1} \sum_{m=1}^l (\lambda_{p_s w+z} + \mu_{(p_f+1)w-z})$$

$$E[Y_{s, f \in T_i}] = \sum_{l=1}^{z-1} \sum_{m=1}^l \left((\lambda_{p_s z+m} + \mu_{(p_f+1)z-m}) \cdot \frac{2 \cdot (z-l)}{z \cdot (z+1)} \right)$$

Therefore,

$$E[Y] = \frac{2z}{n^2} \sum_{i=1}^{k-1} \sum_{j=i+1}^k \sum_{l=1}^{z-1} \sum_{m=1}^l (\lambda_{p_s z+m} + \mu_{(p_f+1)z-m}) + \frac{2z}{n^2} \sum_{i=1}^k \sum_{l=1}^{z-1} \sum_{m=1}^l \left((\lambda_{p_s z+m} + \mu_{(p_f+1)z-m}) \cdot \frac{(z-l)}{(z+1)} \right)$$

Substituting the average value λ of all the λ_i 's and the average value μ of all the values μ_i 's, $E[Y]$ can be approximated by the following expression:

$$E[Y] \approx \frac{\lambda + \mu}{6} \left(3z - \frac{z}{k} + \frac{1}{k} - 3 \right)$$

For sufficiently large k , $E[Y]$ can be further approximated by

$$E[Y] \approx \frac{\lambda + \mu}{2} \cdot z$$

and the proof for $E[Y]$ is complete.

Bibliography

- [1] Anick, P.G. and Flynn, R.A. 1992. Versioning a full-text information retrieval system. *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval* (Copenhagen, Denmark, 1992), 98-111.
- [2] Bairavasundaram, L.N., Arpaci-Dusseau, A.C. et al. 2008. An analysis of data corruption in the storage stack. *ACM Trans. Storage*. 4, 3 (2008), 1–28.
- [3] Barnard, S.T. and Simon, H.D. 1993. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing* (Norfolk, Virginia, USA, 1993), 711-718.
- [4] Bayer, R. and McCreight, E.M. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta informatica*. 1, 3 (1972), 173-189.
- [5] Becker, B., Gschwind, S. et al. 1996. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal*. 5, 4 (1996), 264-275.
- [6] Berberich, K., Bedathur, S. et al. 2007. A time machine for text search. *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval* (Amsterdam, The Netherlands, 2007).
- [7] Brin, S. and Page, L. 1998. The anatomy of a large-scale hypertextual Web search engine. *Proceedings of the seventh International Conference on World Wide Web* (Brisbane, Australia, 1998), 107-117.
- [8] Bui, T.N. and Jones, C. 1993. A heuristic for reducing fill in sparse matrix

- factorization. *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing* (Norfolk, Virginia, USA, 1993), 445–452.
- [9] Cheng, C. and Wei, Y.A. 1991. An improved two-way partitioning algorithm with stable performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 10, 12 (1991), 1502-1511.
- [10] Desmedt, Y.G. and Frankel, Y. 1989. Threshold cryptosystems. *CRYPTO '89: Proceedings on Advances in cryptology* (New York, NY, USA, 1989), 307–315.
- [11] Diffie, W. and Hellman, M.E. 1976. New Directions in Cryptography. *IEEE Transactions on Information Theory*. IT-22, 6 (1976), 644–654.
- [12] Driscoll, J.R., Sarnak, N. et al. 1989. Making Data-Structures Persistent. *J. Comput. Syst. Sci.* 38, 1 (1989), 86-124.
- [13] Easton, M.C. and C 1986. Key-Sequence Data Sets on Indelible Storage. *IBM J. Res. Dev.* 30, 3 (1986), 230-241.
- [14] Farquhar, A., Martin, S. et al. 2005. Design for the Long Term: Authenticity and Object Representation. *Proceedings of Archiving 2005* (2005), 104–108.
- [15] Fiduccia, C.M. and Mattheyses, R.M. 1982. A linear-time heuristic for improving network partitions. *Proceedings of the 19th Conference on Design Automation* (1982), 175-181.
- [16] George, A. 1973. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*. 10, 2 (1973), 345-363.
- [17] George, A. and Liu, J.W. 1981. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference.
- [18] Geremew, M., Song, S. et al. 2006. Using Scalable and Secure Web

- Technologies to Design a Global Format Registry Prototype: Architecture, Implementation, and Testing. *Archiving 2006* (Ottawa, Canada, 2006), 92-95.
- [19] Giuli, T.J., Maniatis, P. et al. 2005. Attrition defenses for a peer-to-peer digital preservation system. *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005* (Berkeley, CA, USA, 2005), 163–178.
- [20] Golub, G.H. and Van Loan, C.F. 1996. *Matrix computations*. Johns Hopkins University Press.
- [21] Haber, S. and Stornetta, W.S. 1991. How to Time-Stamp a Digital Document. *Journal of Cryptology*. 3, 2 (1991), 99-111.
- [22] Haber, S. and Kamat, P. 2006. Content Integrity Service for Long-Term Digital Archives. *Proceedings of Archiving 2006* (2006), 159–164.
- [23] Hagen, L. and Kahng, A. 1991. Fast spectral methods for rapid cut partitioning and clustering. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (Santa Clara, CA, USA, 1991), 10-13.
- [24] Hagen, L. and Kahng, A.B. 1992. A new approach to effective circuit clustering. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (Santa Clara, CA, USA, 1992), 422-427.
- [25] Heath, M.T. and Raghavan, P. 1995. A Cartesian parallel nested dissection algorithm. *SIAM Journal on Matrix Analysis and Applications*. 16, 1 (1995), 235-253.
- [26] Hedstrom, M. 2002. It's About Time: Research Challenges in Digital Archiving and Long-term Preservation. The National Science Foundation & The Library of Congress.

- [27] Hedstrom, M., Ross, S. et al. 2003. Invest to Save: Report and Recommendations of the NSF-DELOS Working Group on Digital Archiving and Preservation.
- [28] Hendrickson, B. and Leland, R. 1995. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*. 16, 2 (1995), 452-469.
- [29] Hirai, J., Raghavan, S. et al. 2000. WebBase: A Repository of Web Pages. *The 9th International World Wide Web Conference (WWW9)* (Amsterdam, 2000).
- [30] How Much Information. <http://www.webcitation.org/5SCSQh9n9>.
- [31] HTTrack. <http://www.httrack.com/>. Accessed: 06-14-2010.
- [32] IIPC: International Internet Preservation Consortium. <http://netpreserve.org>. Accessed: 06-14-2010.
- [33] JaJa, J., Smorul, M. et al. 2009. Tools and Services for Long-Term Preservation of Digital Archives. *Indo-US Workshop on International Trends in Digital Preservation* (Pune, India, 2009).
- [34] Kahle, B. 1997. Preserving the Internet. *Scientific American*.
- [35] Karypis, G. and Kumar, V. 1998. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*. 48, 1 (1998), 96-129.
- [36] Karypis, G. and Kumar, V. 2007. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Version 5.0pre2.
- [37] Kaufman, C., Perlman, R. et al. 2002. *Network security: private communication*

in a public world. Prentice-Hall, Inc.

- [38] Kellar, M., Watters, C. et al. 2006. The impact of task on the usage of web browser navigation mechanisms. *GI '06: Proceedings of Graphics Interface 2006* (Quebec, Canada, 2006), 235-242.
- [39] Kernighan, B.W. and Lin, S. 1970. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical journal*. 49, 2 (1970), 291-307.
- [40] Kleinberg, J.M. 1999. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*. 46, 5 (1999), 604–632.
- [41] Kouramajian, V., Kamel, I. et al. 1994. The time index+: an incremental access structure for temporal databases. *Proceedings of the third international conference on Information and knowledge management* (Gaithersburg, Maryland, 1994).
- [42] Lamport, L., Shostak, R. et al. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401.
- [43] Lomet, D. and Salzberg, B. 1989. Access Methods for Multiversion Data. (1989).
- [44] Lomet, D. and Salzberg, B. 1990. The Performance of a Multiversion Access Method. (1990).
- [45] Lorie, R. 2002. The UVC: a Method for Preserving Digital Documents - Proof of Concept.
- [46] Maniatis, P. and Baker, M. 2002. Enabling the Archival Storage of Signed Documents. *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2002), 3.

- [47] Maniatis, P., Roussopoulos, M. et al. 2005. The LOCKSS peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.* 23, 1 (2005), 2-50.
- [48] MARC Standards. [tp://www.loc.gov/marc/](http://www.loc.gov/marc/). Accessed: 06-14-2010.
- [49] Mearian, L. 2009. The Internet Archive's Wayback Machine gets a new data center. *Computerworld.com*.
- [50] Menezes, A.J., Vanstone, S.A. et al. 1996. *Handbook of Applied Cryptography*. CRC Press, Inc.
- [51] Merkle, R.C. 1980. Protocols for Public Key Cryptosystems. *IEEE Symposium on Security and Privacy* (1980), 122-134.
- [52] Miller, G.L., Teng, S. et al. 1993. Automatic Mesh Partitioning. *IMA Volumes in Mathematics and its Applications*: Springer-Verlag. 57-84.
- [53] Miller, G.L., Teng, S. et al. 1991. A unified geometric approach to graph separators. *the 32nd Annual Symposium on Foundations of Computer Science* (San Juan, Puerto Rico, 1991), 538-547.
- [54] Minerva: Library of Congress Web Archives.
<http://lcweb2.loc.gov/diglib/lcwa/html/lcwa-home.html>. Accessed: 06-14-2010.
- [55] Mohr, G., Kimpton, M. et al. 2004. Introduction to Heritrix, an archival quality web crawler. *4th International Web Archiving Workshop* (Bath, UK, 2004).
- [56] Moore, R., Marciano, R. et al. 2003. NARA Persistent Archives: NPACI Collaboration Project. San Diego Supercomputer Center.
- [57] Nørnvåg, K. 2003. Space-Efficient Support for Temporal Text Indexing in a Document Archive Context. *Proceedings of the seventh European Conference on Digital Libraries* (Trondheim, Norway, 2003), 511-522.

- [58] Nørnvåg, K. 2003. V2: a database approach to temporal document management. *Proceedings of the seventh Database Engineering and Applications Symposium (IDEAS 2003)* (Hong Kong, China, 2003), 212-221.
- [59] Nørnvåg, K. and Nybø, A.O. 2006. DyST: Dynamic and Scalable Temporal Text Indexing. *Proceedings of the Thirteenth International Symposium on Temporal Representation and Reasoning* (2006), 204-211.
- [60] NutchWAX. <http://archive-access.sourceforge.net/projects/nutchwax/>. Accessed: 06-14-2010.
- [61] Page, L., Brin, S. et al. 1999. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report #SIDL-WP-1999-0120. Stanford University.
- [62] Pandora: Australia's Web Archive. <http://pandora.nla.gov.au/>. Accessed: 06-14-2010.
- [63] Papoulis, A. and Pillai, S.U. 2002. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill.
- [64] Pass, G., Chowdhury, A. et al. 2006. A picture of search. *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems* (New York, NY, USA, 2006), 1.
- [65] Patterson, D.A., Gibson, G. et al. 1988. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1988), 109–116.
- [66] Plank, J.S. 1997. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software – Practice & Experience*. 27, 9 (Sep. 1997), 995-

1012.

- [67] Ponnusamy, R., Mansour, N. et al. 1993. Graph contraction and physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. *International Conference of Supercomputing* (Tokyo, Japan, 1993).
- [68] Pothen, A., Simon, H.D. et al. 1990. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*. 11, 3 (1990), 430-452.
- [69] Robertson, S., Walker, S. et al. 1995. Okapi at TREC-4. *Proceedings of the Fourth Text REtrieval Conference (TREC-4)* (Gaithersburg, Maryland, 1995), 73–96.
- [70] Rothenberg, Jeff 1995. Ensuring the Longevity of Digital Documents. *Scientific American*. 272, 1 (Jan. 1995), 42--47.
- [71] Salzberg, B. and Tsotras, V.J. 1999. Comparison of Access Methods for Time-Evolving Data. *ACM Comput. Surv.* 31, 2 (1999), 158-221.
- [72] Sivathanu, G., Wright, C.P. et al. 2005. Ensuring data integrity in storage: techniques and applications. *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability* (New York, NY, USA, 2005), 26–36.
- [73] Song, S. and JaJa, J. 2007. ACE: A Novel Software Platform to Ensure the Integrity of Long Term Archives. *Proceedings of Archiving 2007* (2007), 90–93.
- [74] Song, S. and JaJa, J. 2008. Archiving Temporal Web Information: Organization of Web Contents for Fast Access and Compact Storage. University of Maryland Institute for Advanced Computer Studies.

- [75] The Internet Archive: The Wayback Machine. <http://www.archive.org>.
Accessed: 06-14-2010.
- [76] The National Digital Information Infrastructure and Preservation Program, the
Library of Congress. <http://www.digitalpreservation.gov/>.
- [77] Thibodeau, K. 2002. Overview of Technological Approaches to Digital
Preservation and Challenges in Coming Years. *The State of Digital
Preservation: An International Perspective* (Washington, D.C., 2002).
- [78] UK Web Archiving Consortium. <http://www.webarchive.org.uk/>. Accessed: 06-
14-2010.
- [79] Varman, P.J. and Verma, R.M. 1997. An Efficient Multiversion Access
Structure. *IEEE Trans. Knowl. Data Eng.* 9, 3 (1997), 391-409.
- [80] Wang, X., Yin, Y.L. et al. 2005. Finding Collisions in the Full SHA-1. *CRYPTO*
(2005), 17-36.
- [81] Wang, X. and Yu, H. 2005. How to Break MD5 and Other Hash Functions.
EUROCRYPT (2005), 19-35.
- [82] WARC, Web ARChive file format.
http://www.iso.org/iso/catalogue_detail.htm?csnumber=44717. Accessed: 06-
14-2010.
- [83] WAXToolBar. <http://archive-access.sourceforge.net/projects/waxtoolbar/>.
Accessed: 06-14-2010.
- [84] Wayback. <http://archive-access.sourceforge.net/projects/wayback/>. Accessed:
06-14-2010.
- [85] Weatherspoon, H., Wells, C. et al. 2003. Naming and Integrity: Self-verifying

Data in Peer-to-Peer Systems. *Future Directions in Distributed Computing* (2003), 142-147.

[86] Web-at-Risk. <https://wiki.cdlib.org/WebAtRisk/tiki-index.php>. Accessed: 06-14-2010.

[87] WERA. <http://archive-access.sourceforge.net/projects/wera/>. Accessed: 06-14-2010.

[88] Zhai, C. and Lafferty, J. 2001. A Study of Smoothing Methods for Language Models Applied to Ad Hoc Information Retrieval. *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2001)* (New Orleans, Louisiana, 2001), 334–342.