



Universität des Saarlandes, FR Informatik
Max-Planck-Institut für Informatik, AG 5



Index Partitioning Strategies for Peer-to-Peer Web Archival

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science

von / by

Avishek Anand

angefertigt unter der Leitung von / supervised by

Prof. Dr. Gerhard Weikum

betreut von / advised by

Srikanta Bedathur, Christos Tryfonopoulos

begutachtet von / reviewers

Prof. Dr. Gerhard Weikum

Dr. Srikanta Bedathur

Saarbrücken, 20. January 2009

Non-plagiarism Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

(Avishek Anand)
Saarbrücken, 20. January 2009

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

(Avishek Anand)
Saarbrücken, 20. January 2009

Acknowledgements

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

A special note of thanks to my advisor Srikanta Bedathur. Pursuing this thesis under his supervision gave me many insights into both theoretical and practical research. I also thank him for his persistent support and patience through many discussions we had through the course of the thesis. He has been an excellent advisor.

I also thank Christos Tryfonopoulos, Klaus Berberich and Ralf Schenkel who were also involved closely with the project. Their inputs and comments were always insightful and helpful.

I am very thankful to my parents and friends for the continued emotional support which in the past two years has often proven to be the deciding factor for my successes.

Abstract

The World Wide Web has become a key source of knowledge pertaining to almost every walk of life. The goal is to build a scalable peer-to-peer framework for web archival and to further support time-travel search over it. We provide an initial design with crawling, persistent storage and indexing and also analyze the partitioning strategies for historical analysis of data. Peer-to-peer (p2p) systems are a nice fit here but they suffer from churn and communication overhead and hence require controlled replication for availability and load balancing. The core of the contribution is of index organization by temporally partitioning the time-travel index lists for supporting efficient time-travel search. We also analyze the partitioning strategies in terms of improving replication to improve availability while still keeping the overall blowup of the index in check. We present various heuristic approaches with detailed experimental analysis exploring the nature of partitioning algorithms in a distributed setting.

Contents

Acknowledgements	v
Abstract	vii
1 Introduction	1
1.1 Motivation	1
1.2 Need for a Peer-to-peer architecture	2
1.3 Contributions	3
1.4 Outline of the Thesis	4
2 Related Work	5
2.1 Introduction	5
2.2 Web Archiving Systems	5
2.2.1 Archive Crawling	6
2.3 Peer-to-Peer systems	6
2.4 Peer-to-peer Storage	7
2.5 Replication in p2p systems	10
2.6 Distributed IR and P2P IR	11
2.7 Time-Travel querying and temporal structures	12
3 Framework	15
3.1 Introduction	15
3.2 Crawling, Storage and Indexing: A three-tier architecture	15
3.3 Crawling Layer	16
3.3.1 Human Assisted Crawling	17
3.4 Storage Layer	18
3.5 Indexing Layer	20
3.5.1 Term - Time Partitioning (TeTi)	21
3.5.2 Time - Term Partitioning (TiTe)	23
3.6 Protocol Specifications	24
3.6.1 Node Join	24
3.6.2 Node Departure	25

3.6.3	Querying Storage Layer	25
3.7	Note on Everlast	26
3.7.1	Crawlers	26
3.7.2	Version Directory	27
3.7.3	Time-Travel Index	27
3.7.4	Persistence	29
4	Partitioning Strategies	31
4.1	Definitions	31
4.1.1	Live Entries in a Partition	31
4.1.2	Index Replication	33
4.1.3	Blowup	34
4.2	Maximum-Replication Problem	34
4.3	Maximum-Replication : Proof of NP-hardness	35
4.4	Optimal Approach and Need for Heuristic approaches	37
4.5	Bounded Greedy Partitioning for Max-Replication	38
4.6	Incremental Stitch	40
4.7	Simulated Annealing based Approach	40
4.8	The Maximum-Minimum Reconstructibility Problem	40
4.8.1	Unique Reconstructibility Ratio (URR)	41
5	Implementation and Experimental Analysis	43
5.1	Experimental Setup	43
5.2	Analysis of the dataset	44
5.2.1	Fixed and Greedy Partitioning (unbounded)- Nature of Blowup, Reconstructibility and Replication	44
5.3	Results and Comparison	45
5.4	Analysis	47
6	Conclusion and Future Directions	53
6.1	Discussion	53
6.2	Future Work	55

List of Figures

2.1	Conceptual Architecture for a Global data storage system	8
2.2	TTIX Structure	13
3.1	Three-tier architecture	16
3.2	Term-Time partitioning	21
3.3	Time-Term Partitioning	22
3.4	Architecture of Everlast	26
4.1	Lifetime distribution example.	32
5.1	Lifetimes Analysis	44
5.2	Unbounded Greedy Partitioning - Response of Blowup and Replication	46
5.3	Unbounded Greedy Partitioning vs Performance Guarantee: Replication	46
5.4	Bootstrapped Incremental Stitch and Bootstrapped Bounded Greedy	47
5.5	Effect of Partitioning: Lists for 5 years	48
5.6	Effect of Partitioning: Lists for 2 years	49
5.7	Effect of Partitioning: Lists for 1 years	50

Chapter 1

Introduction

1.1 Motivation

The world wide web contains information from all walks of life. From search engines, news sites, blogs, social networks to scientific, financial, political resources it has become a pervasive medium of communication and is the “information source of the first resort for many” [4]. The democratic nature of content also makes it an unbiased source of information hence capturing this would translate to historical and cultural preservation.

The Web also is constantly evolving. Recent surveys show that search engines like Google index nearly 30 billion pages and Yahoo has reported 16 Bio. also. Apart from the new pages being added content has also been disappearing at a rapid rate. Ntoulas et al.[44] show that 80% of the webpages existing today would either be replaced or disappear in a year. The weekly rate of addition of new pages is said to be 8%. Hence it is all the more important to counter this decay by tracking, collecting and archiving these changes.

Wikipedia defines web archiving as “the process of collecting portions of the World Wide Web and ensuring the collection is preserved in an archive, for future researchers, historians, and the public.” The earliest efforts to build a large scale archive was by Internet Archive [3] and they continue to maintain one of the largest archive collections. Although they are reported to have been increasing their potential and services to archive the Web, the rapid growth of content is overpowering.

The technical challenges of web archives arise from the two major features: acquisition of content and retention or preservation of data for a long time. Web archiving generally relies on web crawling for collection of content and they in turn influence the crawl quality for the archive. Archives require that data must be captured *completely* so that most of the versions, however short-lived, should be faithfully captured and done so *consistently* over a period of time. Secondly, the captures should be done *correctly* and there should be little or no disparity between the version time reported by the crawlers and the actual time when the version came into existence. Effective and efficient solutions for crawling are thus required to

address the high update rates on the Web.

Availability and *persistence* of data are prerequisites for any proposed storage solution for such a large scale archive. The storage solution should be *scalable* to account for the large volumes of data, available for most of the times which requires it to be durable to counter data loss and it should be able to store the data for a long period of time. Further it should be self sustainable and not be under a controlling authority which could govern the "rewriting of history".

Apart from these technical issues [4] enumerates *cultural*, *economic* and *legal* problems. The cultural problem deals with questions like how much to save, what to save and what would be important in the future. Economic problems relates to finding a concrete business model for organizations to adopt such a resource intensive activity where the return investment is slow to emerge. Finally, intellectual property laws about digital documents might change over the long term accounting for the legal problems which might ensue. We however concentrate on the technical aspect of web archiving.

Supporting search functionality over web archives can be considered as a "killer-app" as an effective tool to search the vast amount of information in the archives and learn from the past. Current search engines provide *snapshot-search* over the current view of the web enabling users to search the most recent versions of the web pages. However, search over web-archives should not be limited to a snapshot of the Web, rather it should provide enough capability to search over the time axis as well with richer queries with a temporal context like *time-travel queries*. This would allow temporal analysis of numerous kinds of data like scientific, financial, historic, economic, geological and cultural.

Unfortunately, most of the existing attempts have a limited search capability and their scalability is also questionable. Efforts like Internet archive and the European Archive have limited coverage and lack responsiveness to changes. The only worthwhile searching capability is provided by the *Wayback Engine* where one can search for the version history of a user specified URL.

1.2 Need for a Peer-to-peer architecture

An approach which requires centralized administration leads to limited scalability and high cost infrastructure for storage. Such a system would also require frequent human intervention and monitoring. A centralized approach also means that the archive is under central authority and susceptible to *single-point-of-failure* caused by attacks, hardware degradation or byzantine faults.

A Peer to Peer (or P2P) system connects independent and autonomous entities (called peers) which function both as "clients" and "servers" in a network rather than conventional centralized resources where a relatively low number of servers provide the core value to a service or application. In p2p systems every peer contributes to the overall storage capacity of the system making it scalable and loosely coupled. The content stored by the system is decentralized making it resistant to

any localized failure affecting the entire system. Autonomy of the peers make the storage decisions democratic and hence there is no single authority responsible for the content. P2P systems are also self-organized allowing to them deal with failures and scalability without frequent manual administration. With these natural advantages they seem to be natural fit to a large scale web archive.

The design decisions for a p2p archival storage system pertaining to ensuring availability is based on taking into account resource location and arbitrary peer failures due to network disconnections or autonomous choices. The resource location problem is a well understood space and there are numerous structured and unstructured overlay designs which are commonly used. Churn refers to the joining and leaving of peers in a p2p systems due to the autonomy granted to the peers resulting in data loss. Replication of content is often used to address churn and affects design decisions for data placement, management and peer re-organization. In this thesis we present a peer-to-peer web archiving system and discuss design implications for storage and supporting *time-travel search*.

Time-travel queries: Time-travel queries are keyword queries augmented with a temporal component. For example “barrack obama @ 9/11/2001” or “nicholson @ [9/11/2001 - 9/11/2002]” are examples of time-travel queries. These time-travel queries return documents which existed in the mentioned time point or range which are relevant to the keywords in the query.

1.3 Contributions

We make the following contributions during the course of our work:

- We propose an initial architecture for the searchable p2p web archival system concentrating specifically on index organization and design rationale governing replication and reconstruction of the index. Our goal is to analyze the aspects which govern the design of such a system and propose a indexing framework for efficient distributed time-travel search.
- We introduce the notions of replication and reconstruction in the given context and define optimization problem - *Maximum-Replication problem (max-rep)* corresponding to the partitioning problem. We also analyze the complexity of the max-rep problem and establish its \mathcal{NP} -hardness. We additionally introduce the Maximum Minimum Reconstructibility (MMR) problem.
- We propose heuristic algorithms to as approximate solutions to the max-rep problem namely: *Bounded Greedy*, *Incremental Stitch* and an approach based *simulated annealing*. We have run experiments on the actual wikipedia dataset and we present results showing behavior and comparisons between these algorithms.

1.4 Outline of the Thesis

The remainder of the thesis is structured as follows. In Chapter 2, we discuss the related work done and state-of-art for the ideas presented in the thesis. In chapter 3, we present the framework of the peer-to-peer web archival system and discuss about the design decisions and their implications. In Chapter 4, we introduce the formal model of the partitioning problem, define and prove its hardness. We also present the heuristic approaches and discussion on the reconstructibility problem. In chapter 5, we present the experimental setup and results with a brief look at some of the efficient data structure used. Finally conclusions and future work are presented in Chapter 6.

Chapter 2

Related Work

2.1 Introduction

The design of the proposed archiving system builds on research from different domains. We briefly look at the existing archival solutions and the tools in operation for content acquisition and their maintenance. We then shift our attention to organization of the archive in a distributed and peer-to-peer setup. Firstly, we discuss about the peer-to-peer (p2p) infrastructure for addressing and storage of data objects. Since archives are required to store data durably and in a tamper-free manner we look at existing distributed storage efforts which ensure persistence by redundancy or replication. However, these systems do not exhibit higher-level functionality for queries and analytics. Secondly, we look at existing distributed Information retrieval systems which support querying by storing parts or entire index structures and we analyze the challenges of query processing over a distributed setup. We also look at certain p2p systems which support richer querying like range queries which might be applicable to our setting. To the best of our knowledge, none of them have considered temporal querying and ranking models on evolving data collections. Finally, we look at building index structures for text retrieval over time-evolving data to support time-travel queries.

2.2 Web Archiving Systems

Internet Archive [3] is one the earliest, largest and popular efforts as of today for Web archiving. It has an estimated accumulated size of 500 Terabytes and is steadily growing at a rate of 100 Terabytes per year. Other efforts are mostly focused to archive documents on national culture, history and political importance. The European Archive, Library of Congress and UK Government Web Archive are a few large scale archives which are worth mention. PANDORA, Australia's Web Archive, is another national web archive for the preservation of Australia's online publications which is famous for its scale and coverage.

2.2.1 Archive Crawling

Crawling is a mature technology and there has been a sea of research devoted to specific kinds of crawlings. In our setup we look at distributed crawling and crawlers specializing in harvesting archive style document collections. Distributed crawling [54, 55] aims to overcome the bandwidth limit and coverage issues of centralized crawlers, but suffers from overheads of maintaining the URL frontier. Since these are not archival crawlers, they aim to minimize repeated visits to a page. In contrast, an archive crawler always needs the full contents of a page. Further, traditional crawlers do not take into account the dynamics of the Web to improve their coverage.

There have not been a lot of Archive crawlers, but among the few the ones which warrant mention are Heritrix and NutchWax [5]. Heritrix [2] is an open-source, extensible and large-scale archive crawler used by Internet Archive. It is capable of broad, focused and continuous crawling but it suffers from a key limitation that its a single instance crawler and it cannot coordinate crawling amongst multiple instances. Nutch is open source software based on Lucene and implemented in Java which has a crawler. It also has improved parsers and can produce link-graph databases. NutchWAX is an open source api based on Nutch with special extensions for Web Archives.

Some of the other commercial services which also support archiving and text search are Hanzo Archives [1], ArchiveIT (from Internet Archive) and WebCite.

2.3 Peer-to-Peer systems

There has been a gradual proliferation of applications which use peer-to-peer(p2p) computing and it has received ever increasing attention from the scientific community as it is becoming a hype paradigm for communication on the Web today. It became popular with the file sharing applications (Napster, Bittorrent, Gnutella, Morpheus and Kazaa) but now it has its foray into a host of other applications like streaming (sopcast, coolstreaming, OOX), IP telephony(Skype) and even data management (Amazon's S3 and Dynamo [25]).

A Peer-to-peer system is defined as “a self-organizing system of equal, autonomous entities which aims for the shared usage of distributed resources in a networked environment avoiding central services” [62]. Traditional client-server systems are centralized and susceptible to so called *single – point – failure* and are cumbersome and expensive to administrate. Especially with the explosion of data in the internet a centralized solution for storage is both difficult to scale and manage. On the contrary p2p systems are decentralized, self-organized and composed of autonomous entities and are scalable enough to account for the rapid growth of the web content.

However the major tenet of p2p systems, *decentralization*, is an important feature in system design and gives rise to the oft referred *lookup problem*. The lookup problem deals with resource location in a large p2p system in a

scalable and efficient manner without any centralized service since the p2p system store data in multiple, distant, transient and unreliable locations. P2P systems are organized as a logical overlay over the physical network thus the algorithms and protocols run oblivious to the physical entities underneath. Thus the research community classifies the existing approaches to tackle the lookup problem based on the overlay configuration as *structured* and *unstructured* approaches.

Unstructured p2p architectures do not impose a restriction on the peer organization and use message *flooding* to resource location. They further assign a *Time – to – live(TTL)* to every message to control network flooding and it is supposed to be effective given real-world networks have shown to exhibit the *small – world – phenomena*. Popular implementations are Gnutella[6] and Freenet[19].

Structured p2p architectures are based on overlay networks based on the principle of *resource virtualization* wherein they map resource identifiers like node addresses or data onto a virtual address space and then allocate the virtual addresses to peers. For this purpose each node manages a small number of pointers to carefully selected peers to ensure balanced storage and retrieval. One of the preferred approaches for such a distribution and storage is to use Distributed Hash tables (DHTs) and there are various topologies which implement this. These include rings (Chord [56] , Pastry [51], tapestry[65]), hypercubes (CAN [48]), trees (P-Grid [7], BATON [33]) and butterfly networks (Viceroy [26]).

Apart from these there are other composite or *hybrid* architectures where a select few peers take precedence over others and responsibility of the overlay called *super – peer* architecture like JXTA, Brocade, SHARK and Omicron. There are also systems like *Napster* which are referred to as *centralized* approaches because of the resource information is local to a server. For a comprehensive survey of the p2p systems the reader is referred to [62].

2.4 Peer-to-peer Storage

We shift our focus at some of the existing large scale distributed storage systems which are based on the p2p paradigm. [43] presents a conceptual architecture for a global data storage system in Fig. 2.1. The first layer is the type of overlay structure and it provides mechanisms to join and leave the network like [56],[51], [65]. The second layer provides the basic retrieval and store functionalities for resources in the form of *get* and *put*. The third layer provides the essential services like Load balancing, availability and data coherence and most of the existing system like [23], [27], [7], [36], [30] support a combination of these services. For archival storage the most important requirement is long term availability of resources and for the time being we consider load balancing to be an orthogonal issue.

The autonomy of the peers in a P2P systems means that peers could be unreliable and transient. The dynamics of peer participation is called *churn* and is critical in the design of the system. Providing better *availability* of content in such a dynamic scenario refers to improving the likelihood of resource location (*fault*

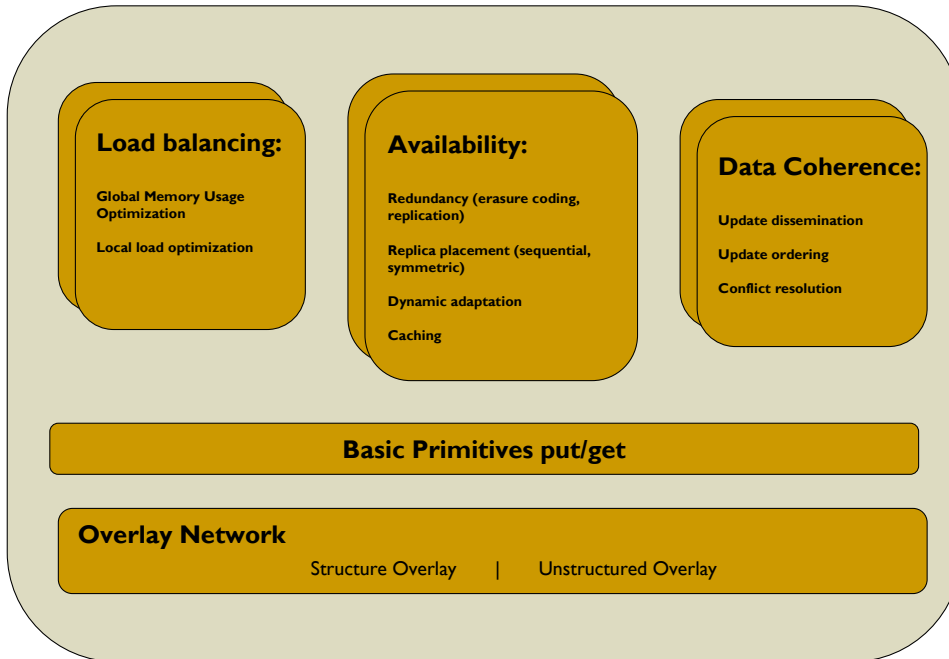


Figure 2.1: Conceptual Architecture for a Global data storage system

tolerance) and doing so in a reasonable amount of time (*responsiveness*). Replication is a key to ensure durable storage, both short and long term, despite failures and is an essential design consideration as a component of availability management. We look into strategies and state-of-art of replication in p2p networks later in this chapter.

To improve responsiveness resources are stored in a manner to make search and retrieval easy. The factors which affect responsiveness are physical position of an object in the WAN, Data object granularity in case if the object is broken into blocks and finally caching mechanisms. Oceanstore uses *floating replicas* to exploit the physical location of replicas in which replicas are moved around the network based on a replica management technique *introspection*. Both PAST and CFS uses unused disk space to cache objects, which can be evicted for replica management. CFS uses the CHORD routing property to cache the replicas on the path leading to resource discovery. Eviction strategies for CFS and PAST are different, while CFS uses *least-recently-used* policy, PAST employs a policy called *Greedy – Dual – Size*.

Approaches like [17],[60],[27] present systems for long term storage of large scale data. We briefly study each of these in the context of applicability to archive-like storage and availability. There are other systems like Freenet[19], Ivy, Kelips and Farsite which also provide distributed file-systems for storage and they are

surveyed in [63]. Finally, most of the durable p2p systems are discussed at length in [64].

Silverback: The purpose of this system[60] is to build a version-based archival storage to *durably* store digitally born information from ubiquitous sources. This identify the challenges of such a system is ensuring *durability, verifiability, availability, maintainability* of data while ensuring *atomicity* of updates with consistent *performance*. They provide an archival model which gives an interface to create, read, write and append versions to the archive atomically. The nodes of the archival system use *Tapestry* as the p2p substrate and the objects are free to reside on servers which are located using hashed-suffix routing structure. The key property of *Tapestry* which the system exploits is that of locality, using which an object can be efficiently retrieved from its blocks due to the natural design of *tapestry* and nomenclature of the block GUIDs.

Silverback makes a case for using erasure coding for replication of content by showing higher availability in times of churn and higher durability over a long period of time. It argues that other replication techniques like complete replication and parity schemes either have higher overheads or are not resistant to high failure rates like network partitioning in WANs.

Verifiability is ensured by creating a *verification tree* over the hashed fragments of the versions. Silverback also takes care about correlated failures by using *data B-Tree* which is a conventional B-Tree with the data stored at its leaves. This structure ensures that all the segments of a version are not geographically collocated hence improving the availability and durability.

Maintenance in Silverback addresses issues of efficient integration and removal of network resources along with localization of errors or loss of data with faster repair. Repairs are carried out mechanisms like *local fragment maintenance, passive detection* and *active sweep* which check and repair the integrity of data locally, globally and periodically.

TotalRecall: TotalRecall is p2p based storage system which automatically manages availability of resources. It attempts to better the current approaches by understanding availability in a distributed setting better. System designers implement a static set of redundant entities and repair mechanisms parametrized by resource consumption. The authors argue that the configuration of mechanisms and availability is a complex task and ill suited to humans. Availability prediction is done by modeling past behavior to predict future outcomes. Short term availability distribution and non-transient failure analysis is used to assess of the duration of departure of a node in the system.

The short and long term availability distribution of peers is used to trigger off a *eager repair* or *lazy repair*. Eager repair promptly replicates contents after it detects a failure oblivious to the type of departure (transient or long term). This is inefficient in terms of network resource usage, overhead maintenance and high

communication cost in case of frequent joins/leaves. Lazy repair uses availability prediction to use the degree of redundancy to mask short-term leaves or joins. Hence replication is not carried out immediately in case of a departure and if there is relatively higher network consumption at that instant. Replication is however carried out sometimes even if there are requisite number of replicas heeding to the global availability distribution which allows for lazy repair during failures. Thus prediction, redundancy management and repair are now built-in system strategies for ensuring overall availability.

TotalRecall is laid out into a three tier architecture of a block store over the DHT, a storage manager and the Recall File system. Chord is used as a p2p substrate and the block store stores objects according to the redundancy management policy (whole files or erasure coding) which are then located using the chord lookup function. The storage manager contains an availability monitor which periodically checks the availability characteristics to enable policy decisions, carried out by the policy module, about the necessary redundancy mechanism (lazy or eager) to be employed by the redundancy engine. Finally the file system provides the user access to the persistently stored data.

LOCKSS: This work analyses the threats to maintaining a long-term and large scale digital information. They identify the fallacies in the assumptions about faults and their characteristics. The fault visibility assumption explains the anomaly about the perception about the time of the fault. It argues that faults could have occurred much before they were diagnosed or manifested and labels it a *latent faults*. The independence assumption attacks the assumption about the independence of replicas, hence calling out the requirement of a replica management scheme or data loss model which takes into considers correlated failures. Finally, the unlimited budget assumption addresses the economic aspect of maintenance of resources.

LOCKSS presents presents a simple abstract model based on the anomalies of the usual assumptions which would be useful while building a large scale digital preservation system. It argues about end-to-end issues affecting designing such a system from technical, economic and human. It also evaluates trade-offs between conflicting factors, for example choice between cheap and higher number of replicas to expensive and higher quality hardware with lower number of replicas.

2.5 Replication in p2p systems

Replication in p2p systems is obtained by creation of extra copies of objects and placing them on different peers. The decision of creation and placement can be done *a priori*[27] or dynamically [17]. Data objects to be replicated can be also broken down into blocks and then replicated as in *erasure coding*[61]. In this approach a file is divided into m fragments and recoded into n fragments with $m \leq n$. The reassembly requires search of any of the m parts or fragments. The degree

of replication is shown to be lesser using erasure codes than replicating entire files. However, erasure coding incurs additional overheads due to the fragmentation and re-assembly procedure. There are improvements over the conventional erasure coding [22] which do not require to track the replicas which have been created and placed. Since the probability of generation of the same fragment twice is minimal, fragments are arbitrarily generated.

Replica placement depends on the block placement policy to achieve object storage system. The goal here is to minimize the likelihood of losing all replicas in case of a concurrent failures in a system. The most widely used policies are *sequential replica placement* and *symmetric replica placement*. CFS and PAST use *sequential replica placement* in which k replicas are placed in its immediate neighborhood. Successor-lists, in CFS, and leaf-sets, in Pastry, implement sequential placement of replicas. One known disadvantage of sequential replica placement is its susceptibility to concurrent failures in case that the peers are co-located. *symmetric replica placement* requires that each identifier in the system should be associated with a constant number of other identifiers and additionally, each node contains all the elements for the keys they are responsible for.

The approaches relating to erasure coding and replica placement could well be applied to any overlay structure. There however have been approaches only for structured p2p networks specifically like Beehive[47]and EpiChord[38].

2.6 Distributed IR and P2P IR

Information retrieval over a distributed setup has been actively studied and evolved over the past few years. Based on [57], on a systemic level they can be classified into two categories: DHT dependent indexing and overlay-dependent indexing. Overlay-dependent indexing can further be subdivided into DHT modified indexing and DHT-free indexing.

In DHT dependent indexing data is indexed over the DHT key space, i.e. using the same hash function for peer identifies. the critical design issue here is generation of the key regarding the data locality. Several efforts come to fore in this category. For supporting keyword queries the classical inverted index is ported into a distributed setup by hashing the keywords in the term space [59][50]. They however suffer from issues of load imbalance due to Zipf distribution of keywords. Other efforts which address locality preservation are pSearch [58] use Latent Semantic Indexing to produce topics out of documents and are indexed in the DHT accordingly. Alvis peers [40] indexes Highly Discriminative Keys (HDK) which are terms and term sets to cope with problem of unscalable bandwidth consumption in p2p networks. Range queries serve as a fundamental query type in the contemporary information retrieval systems and to support them efficiently locality preservation using efficient data structures is of prime importance. Some of the efforts in this regards are Prefix Hash Trees [46] which leverages space filling curves, Distributed Segment Trees [66], Range Search Trees [28] and PRISM [52].

DHT modified indexing preserves the original DHT while modifying contents to preserve data locality. Locality Sensitive Hash (LSH) is a preferred paradigm in such systems [9][29][24]. [10] introduce range forrest to improve on the LSH based solutions and improve applicability to p2p systems. Bloom filters are used over conventional uniform hashing by [34]. DHT-free Indexing does not use the underlying overlay for indexing but architects its own overlay structure. PRing[21] and PTree[20] use distributed BTrees, BATON [33]uses a balanced binary tree and SD-RTee uses a distributed balanced binary tree for spatial indexing. Mercury [18]uses a multi-ring structure for indexing multi-attribute data and supporting range queries while suggesting novel methods to modify links for load balancing. Minerva [11] is another effort in which peers are organized into two overlays. Each peer indexes its own document collection and with help of an overlay of directory peers queries are routed to the relevant peers based on peer profile similarity.

2.7 Time-Travel querying and temporal structures

Time-evolving data has been looked at in the context of database research extensively and various access-structures have been proposed. Time Split B-Trees [39], Persistent B-tree, Exodus and Overlapping B+ -Trees and Multiversion B-Tree (MVT) are few examples of the existing temporal access structures. A comprehensive survey can be referred to at [53].

We are interested in text-search over time versioned data and [12] largely influences our work in terms of the index structures and retrieval. It supports time-travel point queries by maintaining an index referred to as the Time-Travel Index (TTIX).

Time-Travel Inverted Index (TTIX)

The time-travel inverted index builds on the standard inverted index– the workhorse of Information Retrieval (IR). TTIX extends index entries (called *postings* in IR jargon) by validity-time intervals. Index entries thus have the form

$$\langle \text{did}, t_b, t_e, \text{tf} \rangle,$$

where *did* is a document identifier, $[t_b, t_e)$ is the validity-time interval, and *tf* is the term frequency.

Query processing over TTIX involves traversing these index lists which are typically large in case of highly-dynamic collections and frequent terms due to the size of the archived collection. The authors use techniques of called *temporal coalescing* and *sublist materialization* to improve search efficiency. Temporal coalescing reduces the number of postings by coalescing temporally adjacent version payloads into one unified virtual entry. Thus many temporally adjacent entries which have similar scores are considered as one unified entry over a longer period engulfing the lifetimes of all the participants.

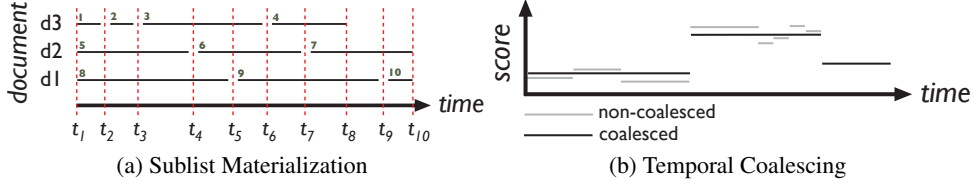


Figure 2.2: TTIx Structure

In addition, TTIx partitions the time axis for each term separately, thus yielding multiple index lists per term, each responsible for an associated time interval. The index list $L_v : [\tau_i, \tau_j)$ thus contains all index entries for term v whose validity-time interval overlaps with $[\tau_i, \tau_j)$. This partitioning of the time axis introduces extra storage-costs, since index entries are replicated across index lists, if their validity-time interval overlaps with more than one of their associated time intervals. [12, 13] proposed techniques, called *sublist materialization*, that determine temporal partitioning of individual inverted lists that trade-off extra storage-costs and query-processing gains. These give rise to two approaches: the Performance Guarantee (PG) approach and Space Bound (SB) approach. They define two extremes P_{opt} , Fig 2.2 (a), which gives the best performance but is not space efficient and S_{opt} , which is requires the minimum space but is not performance optimal.

The performance guarantee(PG) approach allows the performance to be worse by a user defined constant factor while minimizing the space occupied by the partitioned index. This is based on the observation that much of the space is wasted materializing identical sublists and one can significantly save on space if an upper bound on the loss of performance is imposed. Performance of a query is measured by the number of entries read with P_{opt} materialization and hence loss of performance would be the fraction of extra entries that have to be read. They formulate an optimization problem and an optimal linear time solution for materializing the original index list into sublists based on a user defined performance loss parameter γ .

In some cases space might be at a premium and hence the Space Bound(SB) approach is proposed which keeps a check on the overall space occupied by the partitioned index while trying to maximize the performance. The space restriction is modeled by a user-specified parameter $\kappa \geq 1$ that limits the maximum allowed blowup in the index size from the space-optimal solution S_{opt} . It aims at finding a materialization configuration which can minimize the expected processing cost. They correlate the processing cost with the expected performance and propose a dynamic programming based solution which works well for smaller data sets. They go on to provide a scalable approximate solution based on *simulated annealing*. Simulated annealing runs for a fixed number of rounds and they keep track of the best solution so far accepting a better solution and rejecting a worse solution with higher probability.

This is close to our setting since we also discuss strategies to partition index lists of TTX while improving replication and reconstructibility keeping the blowup in check.

Chapter 3

Framework

3.1 Introduction

Documents change over time and do so more often than not. Every new update to a document is termed as a new version for the document and the challenge of the collection of information is in effectively and efficiently capturing the new versions. Building a web archive supporting searching capability involves collection of information, storage and organization of collected information into an archive and finally building index structures over this archive. For an archive, it then is essential to store and manage these collected versions in a way which ensures their long term availability. Finally, to support search over the archives we need to build efficient index structures to support keyword and time-travel queries which makes analysis, exploration and mining of these archives easier and better. Peer-to-peer is a scalable, decentralized and self-organized solution for distributed data storage and since there is no central authority a p2p web archiving solution impedes issues pertaining to censorship and rewriting of history. In this chapter we propose a p2p Framework of building such a searchable web archive. We discuss the design issues at length and specifically concentrate on the indexing and index organization in a distributed setup.

3.2 Crawling, Storage and Indexing: A three-tier architecture

We have three layered architecture: Crawling, Storage and Indexing layer and the peers in the system are logically organized into three corresponding roles: Crawling peers, Storage Peers and Indexing peers. These layers have distinct separation of concerns and have exposed interfaces for communication with the other layers. Each of the layer can be thought of an overlay network where peers are organized according to the requirements of the layer. While the Crawling peers can organize themselves in an unstructured way, the indexing and storage peers have to organize

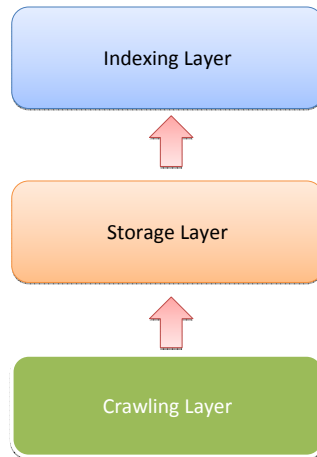


Figure 3.1: Three-tier architecture

themselves in a structured way to adhere to the system constraints and deliverables. We discuss the requirements, challenges and peer organization rationale of each layer with special emphasis on the Indexing Layer. Note that in the entire text we refer to logical entities as peers. Hence a physical peer can act as multiple peers.

3.3 Crawling Layer

Crawlers are an established norm in the present day search engines for information collection. There is a consistent proliferation of content in the web [44] and collection of these ever growing resources is done by deploying crawlers which automatically visit websites periodically to capture them. The challenges of information collection therefore engulfs the technical challenges faced by the state-of-art crawlers and include ones which are which are typical to web archives also.

Firstly, with the growth of web means that it is unrealistic for a single organization to capture the entire web. It is also worth pointing out that the size count is based on “static-page counts freely accessible to search engine users” which overlook the dynamic pages which are generated from databases, unlinked, private and contextual pages[16]. Berg et. al suggests that the so called deep-web might be an

order of magnitude larger than the surface web.

Secondly, not only that there are pages and domains added but the rate at which the existing content change is also rapid. [37] shows that 80% of the webpages existing today will fail to exist in the next year. Our own experiments on the Wikipedia version Lifetimes analysis shows that the average lifetime of a version is 83 days. Hence, crawling a site periodically sometimes is not enough and essential information could still be missed.

Thirdly, there are server side crawler rules which have to be adhered to. The Robot Exclusion Protocol is a convention to prevent cooperating web spiders and other web robots from accessing all or part of a website which is otherwise publicly viewable [49]. The standard implementation is by placing a *robots.txt* file on the website which functions as a request that specified robots ignore specified files or directories in their search.

Finally, there are ancillary issues like “Politeness”, spider traps and spam. Politeness means not to hit or crawl a server often and its often expected from crawlers to maintain a balance between surfers and robot traffic usage of bandwidth and other resources. A Spider trap is a situation when a malicious server generates an infinite sequence on linked pages. Other such attempts provide different views to a crawler and an actual surfer. Although there are existing work on eliminating malicious activities for crawler visits automatic crawling still continues to be a challenge wherein the SEO’s continue to come up with novel ways to mislead crawlers and crawlers becoming efficient to overcome them.

3.3.1 Human Assisted Crawling

Crawler peers continuously collect versions of web documents, either via traditional archival crawlers or via human-assisted crawling. Here, browsers or proxy servers are augmented with plugins that, whenever a document is visited, consult the local history collected so far to determine if the current contents of a document qualify it to be a new version. If so, these are time-stamped with their crawl-time, and are stored in the local history. For simplicity, we assume that crawling peers provide true timestamps, for example by regularly synchronizing their time with time servers in the Internet.

One of the key issues that web archives face is that their captures do not cover the changing Web and are not responsive enough to record the evolution with sufficient accuracy. In this layer we augment the traditional archival crawling with *human-assisted crawling*, which utilizes the collective intelligence of web surfers to obtain high-quality archival dumps of the evolving Web. Here, browsers or proxy servers are augmented with plugins that, whenever a document is visited, consult the local history collected so far to determine if the current contents of a document qualify it to be a new version. If so, these are timestamped with their crawl-time, and are stored in the local history. For simplicity, we assume that crawling peers provide true timestamps, for example by regularly synchronizing their time with time servers in the Internet. At periodic intervals (configurable by the user), these

crawlers offload the history collected so far onto the version directory.

Whenever a user visits a page, contents of the web-page are time-stamped, and stored in a local history database of the user. The local history maintains all unique versions of a page, and at regular intervals (configurable by the user), these are offloaded/migrated to the storage layer. Studies have shown that close to 50% of an individual's browsing activity is page-revisits [32]. Thus, even a single user has a high chance of capturing many versions of a single page.

Retaining such a history of web pages visited by the user for later-time analysis has attracted some attention. For example, Zoetrope [8] implements a variety of browser-side plugins that enable proper capturing and replaying multiple versions of complex web pages – even those which include dynamic content, cookies, and advertisements etc.

Let us now re-visit the limitations and qualitatively analyze it in the light of human assisted crawling. Since it is virtually impossible to crawl the internet for all versions it is essential at least to collect the important and popular pages. Human browsing behavior governing the archived content is especially the case where the “wisdom of crowd” is used to ascribe importance to pages favored by them. It also addresses the other issues where crawlers suffered from low crawl frequencies in the case of pages with high update rates due to technical reasons. For example consider a newspaper site covering an important story which changes frequently might be followed eagerly by users hence capturing most of its versions which might be missed by conventional crawlers. Human Assisted Crawling is not subject any of the limitations of the conventional crawlers like Robot exclusions, politeness, spam farms and spider traps.

Since the Crawling layer is essentially comprised of peers which upload their browsed local history to the storage layers, these peers could well be crawlers themselves which only enriches the archive collection. Further, the human assisted crawling gives us information about popularity of topics, pages and resources which can then be used as seed sets to initialize power crawlers for close monitoring.

3.4 Storage Layer

The storage layer, which forms the archive storage, is responsible to store the data in a *persistent*, *available* and *versioned* manner. Persistence of information in a distributed scenario refers to its preservation for a long time even if the entities storing them are short lived and an object is considered to be *available* if it is stored in the system and the time necessary to retrieve it is reasonable. Availability of an object is evaluated through two orthogonal metrics: fault tolerance or durability and responsiveness. Redundancy is a key factor to ensure durable and persistent storage despite of node failures and it is traditionally obtained through data object replication. There are many distributed storage management system solutions which ensure such properties. We do not propose a novel approach for archive

storage, but bank on building on the already existing systems for the same namely TotalRecall [17], Silverback [60], PAST [27] and LOCKSS [41].

The storage layer contains the following components.

- **Versioned Archive:** This is where the actual documents are stored which belong to the Web archive. Typically storage systems use structured overlay based solutions where one can use *put* and *get* primitives to store and retrieve objects using a object key which is usually a hash of object property such as name of file, string name of the host or a combination of properties such as file name and author. For the storage archive we use the document id and the time-stamp when it was crawled as the lookup key. This component also contains peers which maintain a version directory responsible for collecting the versions from the crawling layer and versioning them after reconciliation.

- **Distributed Time-Travel Index:** This layer also houses the time-travel index [12] over the versioned document collection. The TTX is modeled on the conventional inverted index structures for text retrieval where every entry is augmented with a validity time interval corresponding to the lifetime of the document/version. The TTX in the storage layer is organized such that individual index lists are segmented into non overlapping blocks and assigned to peers. These blocks are called list-segments and they are grow dynamically with addition of entries due to proliferation of the web archive. the end of the index list is represented by a end of index entry in the last segment. Any further additions to the index list is appended to the last segment, and in case of the peer housing this list-segment reaches its space limit, a new list-segment is created and assigned to a new peer. Iterating through the index list in the centralized setting translates to iterating through list-segments in order. The data objects of this component are parts list-segments and are addressed or looked up by combination of the term and the list-segment number.

So in principle this time-travel index can answer time-travel queries but it would be inefficient because of two reasons. Firstly, TTX maintains term lists which are of much larger lengths than snapshot search engines thus query processing is clearly inefficient because of unnecessary read of all the entries. Secondly, query processing in a distributed setup involves query aggregation over multiple peers which individually house parts of the index list(list-segments). Since many peers could be be involved due to the term list lengths this clearly is inefficient in terms of network utilization and latency.

- **Free Pool Manager:** This is another component which keeps track of the available peers and their advertised capacity. This is essential for maintenance and scalability of the indexing layer. We discuss more about this component in the later section about Protocol Specifications.

Interfaces exposed by the Storage Layer: Firstly there is a querying interface to the distributed TTX for time-travel queries. This can be directly used

to process user queries and also for maintenance and repair of the indexing layer. Parts of the index present in the Indexing layer can be recovered by using appropriate time-travel querying. Secondly, it provides an interface to directly retrieving documents from the versioned archive and also parts of the index from the distributed TTIX. Finally it provides access to the free pool manager for partition assignments, retirements and load balancing activities.

3.5 Indexing Layer

Although the storage layer maintains index structures we create another layer on top of the Storage Layer organizing peers to maintain index structures which efficiently answer time-travel queries with better index availability in the face of churn. The actual query processing happens in this layer and we discuss the design rationale and challenges faced in constructing a distributed index over a peer-to-peer overlay.

The distributed TTIX comprises of huge term-lists and query processing is realized by a combination of reading and merge operations in these lists. We already discussed the inefficiency of processing entire index lists in a centralized setup which is compounded more so by a distributed environment. This is overcome by partitioning these lists in the time axis enabling query processing over a part of the entire index list without loss of precision. Let us refer to the figure as an example, in processing the query “ $q @ t1$ ” if we consider the entire list we unnecessarily end up reading entries to which $t1$ does not belong to. Alternately, if we partition the entire index list along the time-axis into smaller components we clearly have to read only a smaller number hence attributing to the efficiency.

The first challenge hence is partitioning individual index-lists for better query processing and so that they could be assigned to individual peers in the p2p system. The second challenge emanates from the nature of peer-to-peer systems, namely churn. In case of peers which store parts of the index leaving the system without notification availability is lost. This results in the need of reclamation or reconstruction of the lost content. Index reconstruction is possible by querying the storage layer in which case the queries during the repair time, corresponding to the content of the peer in question, will not be serviced. The focus thus is to minimize this repair time and improve the availability of the indexing layer. Availability of a p2p system can be improved by controlled replication of content with a given bound of the overall storage limit. We discuss in the following section the introduction of replication due to partitioning of data and how we can use it firstly to improve query service time and secondly to provide guarantees on reconstruction which translates to availability.

Partitioning index lists along the time-axis results in a set of smaller sublists each of which covering relatively smaller and disjoint time durations. The entries corresponding to versions which span across the partition boundaries now belong to all the partitions which have their time-spans overlapping with the version life-

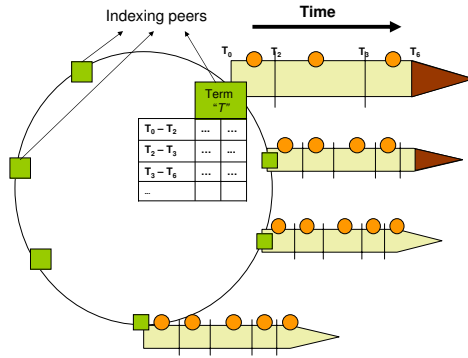


Figure 3.2: Term-Time partitioning

time. This increases the overall size occupied by the partitioned index although it improves query processing efficiency. If we have large number of partitions we improve the reconstruction and efficiency of query processing but we also greatly increase the size of the entire index. On the contrary if we reduce the degree of partitioning we compromise efficiency keeping the index blowup in check. We thus require partitioning strategies to make a trade-off between these two parameters which is discussed elaborately in the next chapter.

We turn now to the p2p overlay design on the indexing layer. There are two dimensions along which we partition the index namely, the term space and along the time axis. The objects in the term space, which in this case are index-lists, are distributed using the hashing function by the well known structured overlay hashing functions [56] [51] and resource location is carried out by well understood DHT's (Distributed Hash tables). Time partitioning on the other hand requires part of an Index list being assigned to a peer. Given these two dimensions we propose two variants of p2p overlay architecture for index organization - TiTe (Time partitioned followed by term partitioning) or TeTi (term partitioning followed by Time partitioning).

Information Retrieval over p2p systems suffer from a vast array of challenges ranging from systems level (network issues, failures characteristics etc) to security (trust issues) and distributed Information retrieval. We concentrate on the retrieval problems and consider problems of trust, load balancing etc as orthogonal as of now which leads to a few assumptions. Firstly, we assume that there are no malicious peers and that all the the peers are thrust worthy. Secondly, we make no assumptions about the term distribution or the lifetimes distribution of versions.

3.5.1 Term - Time Partitioning (TeTi)

This configuration of the Indexing layer consists of two kinds of peers, *directory peers* and *indexing peers*. The directory peers are arranged in a p2p ring and are responsible for terms distributed according to the cryptographic hashing function of the underlying network overlay substrate. In other words terms are distributed

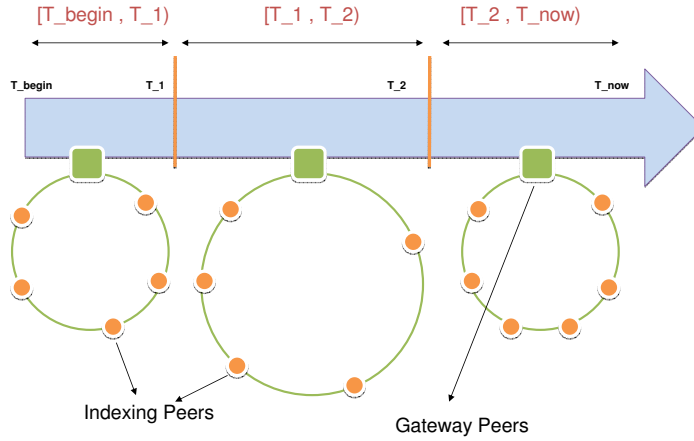


Figure 3.3: Time-Term Partitioning

over the directory peers and each directory peer is responsible for the index list of a particular term. These directory peers then implement partitioning strategies, as discussed above, to divide the entire term list into smaller sublists and assign them to *indexing peers*. Each indexing peer thus contains a part of an index list with entries which have lifetimes overlapping to the time period assigned to the peer. The directory peer additionally maintains mapping of respective indexing peers to the time range for which they are responsible. In TeTi we assume that the directory peers inform their neighbors while they leave the system, hence requiring their contents to be transferred to a substitute and updating the corresponding indexing peers.

Query processing in TeTi proceeds by first resolving the time-travel query into its components, term component and the time component. The query is then routed to the directory peers responsible for the constituting terms which further routes it to the indexing peer responsible for the time point in question. Lets say the query is “john @ t1”, it is first sent to the directory peer responsible for the keyword “john”. The directory peer then looks up to find the time-period to which “t1” belongs and forwards its to the corresponding indexing peer which then does the query processing. In case of absence of the indexing peer, the query is routed to its neighboring index peers for a partial match. Finally, In case of any transient network failures at any stage of the query processing, the query can be forwarded to the storage layer. The centralized implementation of TTIIX can then process the

queries but the responses can get delayed.

Finally, a note about data migration due to repartitioning or change in partition boundaries. The nature of updates are unpredictable since we cannot predict the nature of data in the future it is possible that we might need to change the partition boundaries for a more desirable configuration. This would involve undesirable migration of data over the network. One can think of a cost-benefit model to account for such a scenario in which the data migration is the cost we have to pay for the benefit of a better index partitioning (overall replication or minimum reconstructibility). This means that we do not re-partition to a certain degree, and are ready to be inefficient, until there is a gain compelling enough.

3.5.2 Time - Term Partitioning (TiTe)

In Time - Term partitioning, we first partition the entire lifetime of the document collection and assign each of the resultant time-ranges to peers which are called gateway peers. Each gateway peer is responsible for a time interval and is the gateway to a p2p ring of indexing peers which maintain term-lists for that time range. The TiTe configuration provides time-partitioning on the lifetime of the entire document collection as opposed to the lifetime of an individual index list. Unlike the TeTi updates are majorly localized and not spread around the network. Although, this accounts for easier management but on the flip side it also means localization of network traffic because number of queries could belong to the present or the recent past rather than further back in time. Similar to TeTi we assume that gateway peers in TiTe are fair and report their need to retire whenever they decide to leave the overlay network facilitating others to find a substitute followed by updation to the corresponding indexing peers.

Query processing in TiTe also is a two way process (query routing and local query processing) after query resolution into its time and keyword components. Firstly, the query is sent to the directory peer which belongs to the time range containing the time reference of query and then it uses the DHT to locate the index peer responsible for the keyword. Again, based on the fact that updates do not happen in the past and we do not frequently repartition the past, we can improve query routing and reduce traffic at the directory peers by judicious use of some data structures. In case of absence of an index peer the query is sent to their respective neighbors for a partial match on entries which overlap the time-range of the missing partition. Finally, in case of any failures in query processing in the Indexing layer the query is sent to the index in the storage layer for a slow but sure result.

TiTe suffers from a few disadvantages because of which TeTi is the preferred configuration. One notable observation is that the number of partitions obtained by partitioning the time-axis are much lesser as compared to the number of peers in the p2p ring for term distribution. So in case of TiTe the gateway peers are generally overloaded since they are much lesser as compared to the Directory peers in the TeTi orientation. Secondly, it is possible that most of the queries could belong

to the recent past rather than further back in past. This results in query hot spots on certain partitions. Finally, TiTe would work fine if the past partitions are not disturbed frequently. Changing partition boundaries would result in a large amount of undesirable data migration. In TeTi changing partition boundaries is total to each term list and migration would be controllable.

3.6 Protocol Specifications

This section details the communication protocols specific to this architecture for joining and retirement of peers, index management, handling updates and query routing in the indexing layer. We consider the TeTi configuration of the Indexing layer. Every new node joining the indexing layer is first assigned to a *Free Pool* of peers. The Free pool is a collection of peers which have content yet to be assigned and are maintained by a *free pool manager* which contains a directory of peer information and space they can contribute viz. $(peer_{id}, capacity)$. The free pool manager is located in the storage layer and interacts with the directory peers in the indexing layer whenever it requires a new peer or hands over a retired peer. We assume that there are always enough available peers in the free pool and that the directory peers do not fault/miss on a request for a new assignment.

Another alternative for maintaining a free pool is by creating a multi-cast group of available peers to which the directory peers can send a request message.

3.6.1 Node Join

Information about at least one of the participant nodes, is essential as in most of the distributed overlay structures. We assume for now that the information can be obtained by out of band means or from a match-making server. A node join message, $NodeJoin(peer_{id}, storage_limit)$, is then sent to the free pool manager which adds it to the directory list. The new node now is a part of the free pool. The directory peers can query the free pool manager in the following scenarios.

- Adding a new Indexing Peer: The directory peer implements temporal partitioning (Partitioning Strategies) and needs to assign part of the index-list to a peer with the increase of Archival content. It sends a $getNewPeer(size)$ request to the free pool manager which returns a candidate after referring to the table/directory of available peers.
- Adding a new Directory Peer: If a directory peer is responsible for many terms or if it is responsible for popular terms it might get face higher query traffic. In such cases it can send a $getNewPeer(size)$ to the free pool manager to add another directory peer to the p2p substrate it belongs to. The join follows the conventional structured overlay p2p mechanism of node join [56].

3.6.2 Node Departure

The node when departing notifies the neighbors of its intentions with a *NodeDepart()* message. This is followed by changing the routing table entries and copying of data into the neighboring nodes. In case of Chord which is responsible for all the identifiers between itself and its immediate successor the index gets copied to its previous peer.

3.6.3 Querying Storage Layer

The distributed Index in the storage layer can be accessed by a lookup using a key which is a combination of the term string and the list-segment number. The distributed TTIX in the storage layer can be accessed either for actual query processing in case the indexing layer fails to service a query or for index maintenance of the indexing layer.

- User query processing: One can directly query the TTIX in the storage layer using *Query (term , time_begin, time_end)*. The primary-key for looking up the peer housing the beginning of the index list for the term is a combination of the term and the list-segment number which in this case would be zero. The list-segment numbers are incremented until the entire index is read. However a user query is relayed onto the distributed TTIX only in case of failures at the indexing layer.
- Index updates: The directory peer periodically queries the distributed TTIX to check for index updates by *updateList(term , last_time_of_update, last_list_segment)*. The directory keeps track of the last list-segment so as not to iterate the entire list again, and all the entries with begin times later than the *last_time_of_update* are sent to the directory peer.
- Index reconstruction: The directory peer sends a request for reconstruction of a part of an index list by sending a *reconstructList (term, begin_time, end_time, type)*. This is essentially a time-range query with an exception of the *type* field. Because of some part of the index would have been already reconstructed from the neighboring peers it is only essential to collect the entries which have not yet been reconstructed. The *type* field can take 3 values namely: *SUBSUMED_ONLY* , *LEFT* , *RIGHT* and *COMPLETE*. *SUBSUMED_ONLY* refers to only the completely subsumed entries in the partitions which are not shared with any other partition. *LEFT* refers to the subsumed entries along with the entries which share the left partition. Similarly refers to collecting subsumed entries along with the entries which overlap the right partition. Finally the *COMPLETE* option collects all the entries which are live in the given time range.

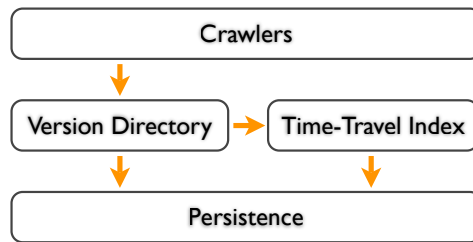


Figure 3.4: Architecture of Everlast

3.7 Note on Everlast

Conceptually, Everlast consists of the following four kinds of peers:

- Crawler peers
- Version-directory peers
- Persistence peers
- Time-travel index peers

Peers playing the role of the version directory, persistence and the index are coordinated through a role-specific overlay network. Each peer in the system can dynamically switch on/off its role by controlling its participation in the corresponding overlay. A structured P2P substrate such as Pastry [51] or Chord [56] can be used to organize these overlays. A high-level data-flow model of Everlast is depicted in Figure 3.4.

The archive is harvested via crawler peers which continuously collect versions of documents (URLs) on the Web, either by explicitly running archival crawlers such as Heritrix [31], or via human-assisted crawling. These versions are locally collected into a history database, which is offloaded periodically to the *version directory*. The version directory is responsible for managing version time lines of all the documents seen so far, keeping document-version-level statistics such as its PageRank score, and managing their placement and location on the persistence layer. The *indexing layer* manages the time-travel index that enables rich temporal text queries over the contents of the archive. The persistence of objects in the system, viz., document versions or inverted (sub)lists, is handled in an uniform manner by the persistence layer. The persistence storage is provided by adapting a distributed object storage service that allows for efficient primary key lookups, and, in addition, provides high availability and durability of content. In the rest of this section, we discuss each of these components in detail.

3.7.1 Crawlers

In Everlast we currently utilize a simple proxy enhanced with archiving abilities to achieve reasonable captures. One of the key technical challenges that requires

further research pertains to preserving the privacy of the user and automatic ways to skip sensitive content from being archived. As a first step, our implementation currently does not capture any content obtained via secure HTTP, but even more powerful schemes are needed. Note also that for simplicity we assume that crawler peers are completely trusted, an assumption that does not hold in general. Foregoing this assumptions opens a range of issues to be tackled at the version directory for constructing accurate version time lines.

3.7.2 Version Directory

Each URL is mapped to a unique version directory peer, which is responsible for maintaining the version history of the URL, i.e., for constructing and updating the sequence of time intervals where in each interval a different version of the URL was active. Since the crawler peers offload their local history without any global synchronization, version arrivals can be highly irregular and conflicting. For example, an older version may arrive after a newer version, or two crawler peers may report different content/size for the same versions of the URL (perhaps due to different language contents being served for the same URL).

In order to deal with these issues, the version directory peer must first reconcile the versions carefully, by taking into account the timestamps and content-signatures. Version reconciliation may result in either append, insert or branch operations on the version time line as well as a simple extension of the current version. If a new version of the URL is created with time stamp t_s after reconciliation, its contents vc are posted into the persistence layer calling $put((URL, t_s), vc)$ of the persistence layer. Additional page-level measures, e.g., its PageRank, can also be computed by the version directory peers using methods such as JXP [45] that are designed for P2P systems.

Once a new version is readied by the reconciliation stage, its contents are *inverted* and term-level statistics of the version are posted into the Time-Travel Indexing layer.

The contents in the version directory form a critical resource for the functioning of Everlast, demanding its high-availability at all times. Since this layer only houses compact meta data structures, we can achieve these availability demands via eager replication mechanisms commonly suggested in distributed storage research.

3.7.3 Time-Travel Index

Efficiently supporting time-travel text querying is an important aspect of Everlast. Existing access structures for time-evolving data, as mentioned earlier, may at first seem adaptable to handle this task. There are, however, several details unique to our setting that make their applicability – at best – questionable: (i) out-of-order insertions of documents versions, since no synchronization between crawler peers is enforced, (ii) the decentralized setting for which none of the existing access

structures was designed, and, finally, (iii) management of text data, which is clearly not a key application area of the existing access structures.

Everlast employs a distributed variant of the time-travel inverted index proposed in [12, 15]. We provide a concise overview of the index structure, before describing how it can be adapted to a decentralized setting.

Two key benefits of TTX are (i) its ease of implementation and (ii) the fact that well-known optimizations to the inverted index (e.g., *compression* and *pruning* techniques) remain applicable.

TTIX in Everlast

There are – at least – two ways of adopting the TTX structure to a distributed setting.

One of the natural ways is to partition the index first by time and then by term, thus yielding a **time-term-partitioned index** (Ti-Te for short). In this case, the whole collection of documents in the archive is partitioned up-front based on the validity-time intervals of documents, and a separate standard inverted index for each of the partitions is built. Index entries corresponding to document versions that span multiple temporal partitions are replicated in each partition. The partitioning of the document collection can be done, for instance, (i) to optimize reconstructability, i.e., to maximize the portion of a partition that can be reproduced from its neighboring partitions in case of a failure of the partition, or, (ii) to balance the query load between partitions.

However, in a truly decentralized setting, such as Everlast, the Ti-Te partitioning scheme has significant drawbacks. Determining a partitioning of the document collection up-front requires, among other things, collecting statistics about the collection as a whole. Regarding the size of the targeted document collections, it is questionable whether keeping such statistics is a task manageable by a single peer having rather limited computational and storage capacity. In settings that include privileged super peers, this judgment would have to be re-evaluated.

Alternatively, the index can be partitioned first by term and then by time, thus yielding a **term-time-partitioned index** (Te-Ti for short). In this partitioning scheme, the entire term space is first partitioned using the cryptographic hash function of the underlying P2P substrate. Each term is thus assigned to a time-travel index peer, which is then responsible for managing index entries for the term. The peer's responsibilities include (i) keeping statistics about the index entries, (ii) determining and continuously adjusting the temporal partitioning, (iii) interacting with persistence peers who keep the index lists, and (iv) interacting with version-directory peers in the presence of additional document versions containing the term. Since the meta data needed to perform partitioning of the inverted list of a single term is small enough to be stored and processed in a single peer, the Te-Ti partitioning scheme seems to be better suited in a decentralized setting. Accordingly, Everlast adopts this model of TTX organization.

Figure 3.4 shows how Everlast implements the Te-Ti indexing. Indexing peers

(denoted by green boxes) are organized into an overlay, with each peer housing the meta data for all the temporal partitions of a term-specific inverted list. In addition to the boundaries of the partition, the meta data could include access-statistics, number of entries, etc. which could be exploited during query processing or subsequent index reorganization. Each entry in this metadata table also uniquely determines the primary-key into the persistence layer (denoted as a orange node), which can be used to retrieve the entries during query processing.

3.7.4 Persistence

Persistence of objects, more precisely document versions and inverted lists, is handled in an uniform manner using existing P2P storage technologies such as OceanStore or PAST that provide high durability and availability of objects. These storage services provide $\text{put}(\text{pk}, \text{obj})$ and $\text{get}(\text{pk})$, where pk is the primary-key of the object obj .

In Everlast, the timestamps associated with objects is incorporated into defining its primary-key. Thus, the document versions and temporal partitions of inverted lists can be quickly located from the storage layer. We define the primary key of a version starting at τ_s of the web page URL as a pair $\langle \text{URL}, \tau_s \rangle$. Similarly, for a index list partition $L_v : [\tau_i, \tau_j)$, we assign the pair $\langle v, \tau_i \rangle$ as the primary key.

Chapter 4

Partitioning Strategies

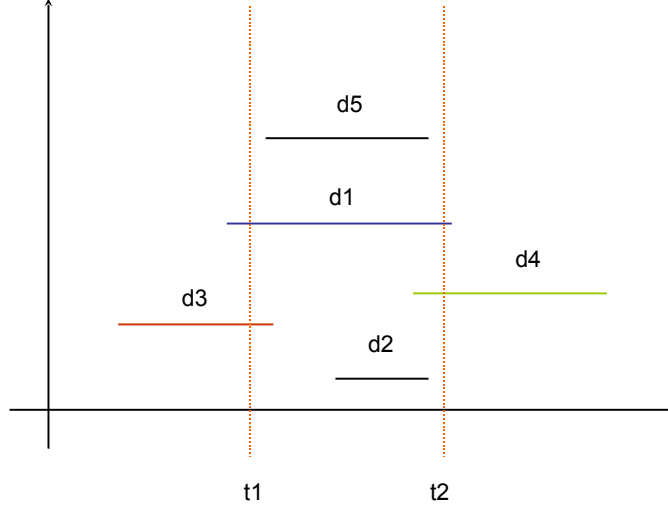
We focus on the problem of partitioning individual term-lists along the temporal axis. Partitioning serves not only to break larger lists into sublists which fit in the individual peers, but also improves the query service time and index reconstruction. We consider replication of index entries across partitions as a means to providing fast reconstruction leading to higher index availability. In this chapter we define the system characteristics like live entries, blowup, replication and reconstruction and formally propose an optimization problem which aims to maximize the replication in the system termed as the Maximum Replication problem. We analyze the complexity of the problem and show that it is \mathcal{NP} -hard. We present an optimal solution to the problem based on branch and bound for limited time-ranges and finally present some heuristic approaches which attempt to provide scalable solutions in polynomial time.

4.1 Definitions

Entries in the term-list are of the form $(did, score, t_b, t_e)$ where *did* is the document identifier corresponding to a document version in the archive. Every document has an associated *validity lifetime* or simply lifetime, $[t_b, t_e)$ denoting the time in which the document existed. In the rest of the chapter we use the term entry and document interchangeably. We follow the open-closed semantics for lifetimes of documents and partition boundaries which requires the begin time of the partition to belong to the partition (closed) and end time of the partition being open. Hence documents which begin at the begin time of the partition do not belong to the partition before it. Similarly, documents which exist at the end-time of the partition belong to both the adjoining next partition and to the current partition.

4.1.1 Live Entries in a Partition

Live entries in a partition or interval are the set of all the entries which have their lifetimes overlapping with the partition. The live entries, for a term v , for the



$$L_v : t_1 = \{d1, d3\}, L_v : t_2 = \{d1, d4\}, L_v : [t_1, t_2) = \{d1, d2, d3, d4, d5\}$$

Figure 4.1: Lifetime distribution example.

interval $[t_i, t_j)$ are defined by $L_v : [t_i, t_j)$. Formally:

$$L_v : [t_i, t_j) = \{(d, [t_b, t_e)) \in L_v \mid t_b < t_j \wedge t_e \geq t_i\}.$$

$|L_v|$ is the total number of entries in the index list. Assuming that the begin and end times of the entire list are t_{begin} and t_{end} , L_v corresponds to $L_v : [t_{begin}, t_{end})$.

Having introduced live entries in a partition, it is easy to observe that live entries at a time point are all the entries which are cut by that time point. These are the set of entries which either start at or before the time-point and end after it. Hence following the open-closed semantics the documents which end at a given time-point are not live at that instant. This helps us to remove any discrepancy relating to multiple existence or non-existence at a time point.

We present an example which we will reiterate throughout this chapter to illustrate the definitions. Lets consider the time range $[t_1, t_2)$ as the partition in question, hence the live entries in this partition is the set $\{d_1, d_2, d_3, d_4, d_5\}$. The live entries at t_1 are $\{d_1, d_3, d_5\}$ and at t_2 are $\{d_1, d_4\}$. Notice that d_5 is live at its begin time and not at its end time because of the open-closed semantics.

Classification of Live Entries: We can further classify the live entries in a partition into subsumed, strike-through, left and right entries. **Left entries** are those live-entries in the partition which begin before the start of the partition and end before the end and after the begin of the partition.

$$Left : [t_i, t_j) = \{(d, score, [t_b, t_e)) \in L_v : [t_b, t_e) \mid t_b < t_i \wedge t_e \geq t_i \wedge t_e < t_j\}.$$

Right entries are the set of entries which start inside the partition and end after the end of the partition.

$$Right : [t_i, t_j) = \{(d, score, [t_b, t_e)) \in L_v : [t_b, t_e) \mid t_b \geq t_i \wedge t_b < t_j \wedge t_e \geq t_j\}.$$

Subsumed entries are those live-entries which are completely contained inside the partition.

$$Sub : [t_i, t_j) = \{(d, score, [t_b, t_e)) \in L_v : [t_b, t_e) \mid t_b \geq t_i \wedge t_e < t_j\}.$$

Finally, **Strike-through entries** are those live entries which start before the partition and end after the partition.

$$Str : [t_i, t_j) = \{(d, score, [t_b, t_e)) \in L_v : [t_b, t_e) \mid t_b < t_i \wedge t_e \geq t_j\}.$$

Hence following the example, we have following classification:

- $Right : [t_1, t_2) = \{d_4\}$
- $Left : [t_1, t_2) = \{d_3\}$
- $Sub : [t_1, t_2) = \{d_2, d_5\}$
- $Str : [t_1, t_2) = \{d_1\}$

4.1.2 Index Replication

Any entry which spans across a partition time boundary is said to be replicated. Index Replication or Replication is a global measure, which gives us the fraction of the total entries ever replicated or replicated at least once. Before formally defining replication we introduce the notion of a *partitioning*. Time is a continuous attribute but its measured in quantum or basic time intervals and this interval length accounts for the granularity, δ , of the index. This means that all the time intervals in the system have a length of at least one basic interval. For an index which is n time-points long with granularity δ we define \mathcal{B} , as a set of basic time-points as:

$$\mathcal{B} = \{t_i \mid 1 \leq i < n \wedge t_{i+1} - t_i = \delta\}$$

We assume that all the document and partition boundaries lie exactly on these points and not within. We refer to the set of time intervals for the time-partitions or simply referred to as partitions as

$$\mathcal{M} \subseteq \{[t_i, t_j) \mid 1 \leq i < j \leq n\},$$

and demand

$$\forall t \in [t_1, t_n) \exists m \in \mathcal{M} : t \in m,$$

i.e., the time intervals in \mathcal{M} must completely cover the time interval $[t_1, t_n)$, so that time-travel queries q^t for all $t \in [t_1, t_n)$ can be processed. We also assume

that intervals in \mathcal{M} are disjoint. Thus, a *partitioning*, $P(\mathcal{M})$ is defined as set of time-points that determine partition boundaries. Thus for a partitioning $P(\mathcal{M})$, replication $\mathcal{R}(\mathcal{M})$

$$\mathcal{R}(\mathcal{M}) = \frac{|\bigcup_{t_i \in P(\mathcal{M})} L_v : t_i|}{|L_v|}.$$

In the discussed example the partitioning is $\{t_0, t_1, t_2, t_{end}\}$ and the replication is 3/5. The replication introduced by partitioning increases the overall size of the index since the resultant index lists house redundant replicated entries. This leads to defining the notion of blowup.

4.1.3 Blowup

Blowup refers to the overall increase in size of the partitioned term-list. It is defined by the final size occupied by all the sublists or partitions to the original size of the term-list. The size of a partition is the number of the live entries in that partition, hence the blowup given a certain partitioning $P(\mathcal{M})$ is:

$$blowup = \frac{|L_v| + \sum_{t_i \in P(\mathcal{M})} |L_v : t_i|}{|L_v|}.$$

alternatively,

$$blowup = \frac{\sum_{m \in \mathcal{M}} |L_v : m|}{|L_v|}.$$

Both definitions are equivalent formulations of blowup. The first equation defines blowup based on the partitioning of the index and the second defines it based on the live entries in a partition. Every partition boundary introduces two new partitions from an initially existing partition thus the total number of index partitions goes up by unity. The first equation is based on the fact that the number of entries replicated by the new partition boundary and hence accounting for the overall size of the index is the number of live entries at the partition boundary. The second equation derives itself from the first using the fact that the sum of the entries of the resultant two partitions is more than the existing parent partition by the number of live entries at the new partition boundary.

In the optimization problems presented later we allow the user to specify a limit to the index blowup. This user specified parameter, referred to as γ , is a hard bound on the space blowup incurred due to partitioning.

4.2 Maximum-Replication Problem

To obtain the maximum reconstruction keeping the entire blowup in check we need to maximize the number of documents which are replicated at least once. Let us,

further for the ease of usage, define the allowable number of entries in the interval $[0, t_i]$ as A_{t_i} ,

$$A_{t_i} = \gamma |L_v : [0, t_i]_{l>1}| + |L_v : [0, t_i]_{l=1}|.$$

where $|L_v|_{l>1}$ are the documents which have a lifetime longer than the elementary time interval. In other words these documents can be partitioned (since partitions can only take place at the elementary time points). $|L_v|_{l=1}$ are the documents which have a lifetime which are a subset of the elementary time intervals \mathcal{E} . The reason why they are not multiplied by γ is because the elementary lifetimes cannot be partitioned (they do not contribute to the live entries of any of the time points) and hence they do not contribute to the relaxation factor γ .

In the rest of the thesis whenever we refer to new documents we refer to newly added documents with a lifetime greater than the elementary time interval. Hence the problem statement now for a given time period $[0, t_N]$ is the following:

$$\arg \max_{\mathcal{M}} \mathcal{R}(\mathcal{M}) \quad \text{s.t.}$$

$$\sum_{m \in \mathcal{M}} |L_v : m| \leq \gamma |L_v|.$$

or alternatively from the definition of allowable entries above:

$$\arg \max_{\mathcal{M}} \mathcal{R}(\mathcal{M}) \quad \text{s.t.}$$

$$\sum_{m \in \mathcal{M}} |L_v : m| \leq A_{t_N}.$$

In both of these formulations above, the objective function aims at maximizing the overall replication of the system subject to the restriction that the overall size of the resulting index is γ -bounded where γ is the user specified blowup parameter. The number of allowable entries, given by A_{t_N} , is in a way the budget allowed determined by the value of γ and our goal is to increase the degree of partitioning in a way that best increases the overall replication.

4.3 Maximum-Replication : Proof of NP-hardness

We show that *Max – Replication* is \mathcal{NP} -hard by constructing a reduction which maps all instances of the well known *subsetsum* problem (known to be \mathcal{NP} -complete) into an instance of *Max – Replication* in polynomial time. Given n positive integers w_1, \dots, w_n and a positive integer W , the *Subset – Sum Problem (SSP)* [42] is to find a subset-sum of $N = \{1, \dots, n\}$ so as to

$$\max(Z = \sum_{i=1 \dots n} w_i)$$

s.t.

$$\sum_{i=1 \dots n} w_i \leq W.$$

Proof. For the reduction we construct an instance of the *Max – Replication* problem with $2n$ time-points $1, \dots, 2n$. We then define n sets of entries, S_1, \dots, S_n , such that each set S_i consists of w_i entries respectively, each of length 2 time intervals:

$$S_i = \{(d, [t_b, t_e]) \mid t_b = 2i - 1 \wedge t_e = 2i + 1\}$$

and

$$|S_i| = w_i$$

We further define a set of n unpartitionable entries, U as :

$$U = \{(ud_i, [t_b, t_e]) \mid t_b = 2i \wedge t_e = 2i + 1\}$$

Hence the set of all documents L in the collection is :

$$L = \left\{ \bigcup_{i=1..n} S_i \right\} \cup U$$

and in case of absence of partitions the space occupied by the entries for the documents is the number of documents in the collection i.e, $|L|$.

$$|L| = \left| \left\{ \bigcup_{i=1..n} S_i \right\} \right| + |U|$$

or

$$|L| = \sum_{i=1..n} w_i + n$$

Given the blowup factor gamma, γ , the number of allowable entries or available space is given by:

$$A = \gamma \left| \left\{ \bigcup_{i=1..n} S_i \right\} \right| + |U| = \gamma \sum_{i=1..n} w_i + n$$

which gives us a remaining space of

$$A - |L| = (\gamma - 1) \sum_{i=1..n} w_i$$

When a partition is introduced at a time-point t_p , the total number of entries of the system increase. This increase as a result of introduction of a partition at t_p is:

$$\delta(t_p) = |L_{t_p}| - |\{(d, [t_b, t_e]) \mid t_b = t_p\}|$$

From this instance of *Max – Replication* we can deduce that we either have a delta, δ , or increase of w_i (if there is a partition at $2i$) or zero. Now by setting the remaining available space to W , we have a situation where we have to choose a subset of points from the n points which have a non-zero delta (for all t_i equaling $2i$) which introduce a size increment equal to w_i , hence we have a direct correspondence to the *subset – sum* problem.

$$W = (\gamma - 1) \sum_{i=1..n} w_i$$

and hence

$$\gamma = \frac{W}{\sum_{i=1..n} w_i} + 1$$

Hence, every instance of a *subset – sum* problem can be reduced to an instance of the *maximum – replication* problem. This is done by constructing sets of partitionable entries S_i , unpartitionable entries U and a gamma as defined above proving that *Maximum – Replication* is \mathcal{NP} - Hard. \square

4.4 Optimal Approach and Need for Heuristic approaches

We use a branch and bound approach to come up with an optimal solution for limited time ranges. The bounds are determined by the blowup constraint. We initially start with a configuration without any partitions and then progressively partition it to form subproblems of smaller time ranges and updated available space. In every iteration we calculate the increment in replication of the system and update the remaining available space. We keep track of the best solution and continue partitioning until it exceeds the space constraint.

Efficient data structures: The lifetimes of documents are represented by a lifetimes matrix which one dimension represents the begin time and the other the end time. So the (i,j) th cell contains the number of documents which begin at time i and end at j. We maintain a cumulative begin time matrix which contains the cumulative number of documents beginning at a given time and ending at the time in question. The (i,j)th cell in this matrix would contain all the documents which begin at time i, and which end either before or at time j. The aggregation operator used that of addition hence the value at a cell is the sum of row cells to its left. We similarly have a cumulative end time matrix where the (i,j) th cell in this matrix would contain all the documents which begin at or after time i, and which end exactly at time j. Again the aggregator operator is addition and the value at each cell is sum of all the column cells above it. These matrices help to efficiently calculate the new replicated entries after a partition has been added into the partitioning.

Although we improve the performance by reducing the number of operations to calculate the increment in replication at every iteration, this approach doesn't scale to larger time-ranges because of the exponential number of options. This motivates the need for heuristic approaches which have a polynomial running time and still are approximate solutions to the maximum-replication problem.

Algorithm 4.4.1: BOUNDEDGREEDY($gamma, live[1..n], numEntries$)

comment: price[i] : number of unpartitionable entries

comment: size[i] : increase in overall size by selecting i

$price[1..n] \leftarrow live[1..n]$

$size[1..n] \leftarrow live[1..n]$

$candidateTimePoint = 0$

$partition = \langle \rangle$

$currentSize \leftarrow numEntries$

$availableSize \leftarrow gamma \cdot numEntries$

while $currentSize \leq availableSize$

do	{	<p>comment: choose a feasible time-point i with maximum price</p> <p>$i \leftarrow argmax(price)$</p> <p>if $noFeasibleCandidates$</p> <p style="padding-left: 20px;">then break</p> <p style="padding-left: 20px;">else {</p> <p style="padding-left: 40px;">$candidateTimePoint \leftarrow i$</p> <p style="padding-left: 40px;">$partition \leftarrow partition \cup i$</p> <p style="padding-left: 40px;">$updatePrice(partition)$</p> <p style="padding-left: 40px;">$currentSize \leftarrow currentSize + size[i]$</p> <p style="padding-left: 20px;">}</p>
-----------	---	---

return ($partition$)

procedure UPDATEPRICE($price, partition$)

comment: left(i) : set of entries which are live at partition left of i

comment: right(i) : set of entries which are live at partition right of i

$price[i] \leftarrow |left(i) \cup right(i)| - |left(i) \cap right(i)|$

4.5 Bounded Greedy Partitioning for Max-Replication

An obvious way to partition the timeline is to place partitions at fixed intervals or *fixed partitioning*. One of the disadvantages of fixed partitioning is that it does not heed to the nature of the data. Consider the case when the lifetimes are not uniformly distributed and there are long periods of inactivity (low density of live entries) interspersed with shorter ones of higher activity. Fixed partitions might tend to over-partition the inactive periods while the higher activity regions are under-partitioned. This might result in partitions having large differences in quantities and lifetime distribution of entries which is undesirable.

The bounded greedy strategy is one of the heuristic approaches employed to

greedily partition regions of with higher degree of replication. We attribute two properties to every time-point: *price* and *size*. The number of live entries remains constant with the choice of different partitioning and this accounts for the *size* of a time-point. Choosing this time-point would result in increasing the overall size of the partitioned list by exactly the *size* of the time-point. On the other hand *price* is the gain or increase in replication the system would benefit from the addition of the time-point to the existing set of partition boundaries. This unlike weight varies with the partitioning and has to be re-calculated with every new addition. With the introduction of a partitioning boundary at a time point, the gain due to replication for the other time-points which share the same live entries as the partition boundary decreases. The choice of a time-point as a partition boundary depends on the gain in replication it can provide while adhering to the space requirement. At each iteration step the gain due to replication for the non-partition points are calculated and a candidate time-point is chosen. The candidate time-point has the maximum gain due to replication without overflowing the space constraint. This is continued until we keep adhering to the available space limit.

In the implementation we maintain two arrays storing the *price* and *size* of the time-points and another dynamic list for the partition boundaries. In every iteration we find the time-points which if selected would not exceed the *availableSize* and we call them as *feasible* candidates. The greedy step is to select the feasible time-point which has the maximum price or largest number unpartitioned live entries. The following steps are then to update the state variables for the next iteration.

Algorithm 4.5.1: INCREMENTALSTITCH(*gamma*, *endTime*, *sampleSize*)

```

partition  $\leftarrow \langle \rangle$ 
beginTime  $\leftarrow 0$ 
endTime  $\leftarrow$  NEWSAMPLESIZE(endTime, sampleSize, beginTime)

while nextSamplesSize  $\geq 0$ 
    {
    comment: Extract the next sample
    allowableSpace  $\leftarrow$  gamma  $\cdot$  numNewEntries
    partition  $\leftarrow$  partition  $\cup$  OptimalStrategy(sample, allowableSpace)
    do {
    comment: calculate the next sampleSize
    nextSampleSize  $\leftarrow$  minimum{nextSampleSize, systemEndTime  $-$  endTime};
    beginTime  $\leftarrow$  endTime
    endTime  $\leftarrow$  endTime  $+$  nextSampleSize
    }
return (partition)

```

4.6 Incremental Stitch

In this approach we use the fact that optimal approach can run over smaller time-ranges to break the problem space into smaller components and run the optimal algorithm on them. The partition boundaries thus formed are merged in a *stitch* operation to the already existing partition boundary set. The selection of the sub-problems is done by selecting a

The optimal algorithm runs over fixed lengths of samples which are taken incrementally from the start of the begin time of the list. Each subproblem or sample calculates the allowable space limit by the product of gamma and the number of documents introduced after the begin time of the current sample. The currently evaluated partitioning is then added to the existing ones before. We continue this incremental calculation of optimal partitioning for samples and stitching them with the past until we reach the end of time.

For every sample we first calculate the number of new entries, *numNewEntries*, which originate after the begin time of the sample and before its end time. The optimal partitioning is then established for the sample and merged/stitched to the list of partitions, the *partition* list here. Since the Max-rep problem is not proven to have optimal substructure we cannot use dynamic programming approaches hence the *stitch* operation might result in sub-optimality.

4.7 Simulated Annealing based Approach

Incremental Stitch can be slow in case of large data sets or systems where the blowup factor, γ , is higher. We look at an approximate solution to the problem using *simulated annealing* [35]. Simulated annealing explores the solution space in a fixed number of rounds. In each round a random successor of the current solution is looked at and checked if it gives a better configuration than the current best. It is accepted if it improves the current best and is accepted with a lower probability, $e^{-\Delta/r}$ where Δ is the decrease in replication from the current best and $R \geq r \geq 1$ denotes the number of remaining rounds, if it does not. We continue to maintain a list of partitions as before and we check this list for containment for every random choice.

Bootstrapping with Incremental Stitch and Bounded Greedy is also possible since we maintain a current best solution. This evidently gives us a better start point thus improving our solution from the bounded greedy approach, but it also has a bigger running time complexity.

4.8 The Maximum-Minimum Reconstructibility Problem

The maximum-replication problem might not be representative of the actual configuration which we desire at times. A higher replication does not always translate

to higher reconstruction in cases of failures. There could be instances when high replication is only due to a few partitions while the other majority of partitions with relatively lesser density of document distribution suffer and are not globally represented. This calls for reconstruction problem which we define as a maximum-minimum problem called as the Maximum-Minimum Reconstructibility problem to ensure that each partition is ensured of a minimum level of high reconstruction. We firstly introduce the notion of Reconstructibility or the Unique Reconstructibility Ratio before defining the optimization problem.

4.8.1 Unique Reconstructibility Ratio (URR)

Reconstructibility of the partition is the degree to which a partition can be reconstructed from its neighboring partitions in case of a loss of the index list corresponding to the current victim partition. Reconstruction is possible through the replicated entries which are live in the victim partition in question. These replicated entries span across the partition boundary and also are live in the adjacent neighboring partitions and are therefore used to reconstruct a part of the victim. Note that these entries are all the live entries except the subsumed entries which are anyways lost and have to be reconstructed independently.

The unique reconstructibility ratio of a partition is defined as the ratio of the sum of the number of right and left entries to the total number of live entries in the partition.

$$\mathcal{URR} : [t_i, t_j) = \frac{|Right : [t_i, t_j) \cup Left : [t_i, t_j)|}{|L_v : [t_i, t_j)|}$$

or alternately:

$$\mathcal{URR} : [t_i, t_j) = \frac{|L_v : t_i \cup L_v : t_j| - |L_v : t_i \cap L_v : t_j|}{|L_v : [t_i, t_j)|}$$

We define the Maximum-Minimum Reconstructibility(MMR) Problem which where we intend to maximize this minimum reconstruction value for every allowable partitioning scheme possible. The objective function here is URR and we intend to find a partitioning which has the maximum of minimum URR value of all the other configurations given a user regulated γ -bounded allowable blowup. Formally stated:

$$\arg \max_{\mathcal{M}} \min_{\mathbf{m} \in \mathcal{M}} \mathcal{URR}(\mathbf{m}) \quad \text{s.t.}$$

$$\sum_{m \in \mathcal{M}} |L_v : m| \leq A_{I_N} .$$

The complexity of this problem still remains to be established and is a contender for the future work to be undertaken as an extension of this thesis. In the implementations we have and results we show, we also mention the minimum URR values.

Chapter 5

Implementation and Experimental Analysis

5.1 Experimental Setup

We used the Wikipedia version history for our experiments. The Wikipedia version history is downloadable and we consider the first five years of version history i.e. from its inception in 2002 to 2007. A TTX style index-list is constructed from the triples $(d, p, [t_b, t_e])$, but since the payload p does not affect the partitioning of the index it is safe to consider document-lifetime pairs as the constituent index entries. The level of granularity of data representation is day level or in other words the smallest step size of measurement if time of documents is one day.

The document-lifetime pair list does not necessarily correspond to any particular term list, but serves as an example for simulation of index lists over a sizable period of time. The simulation of index lists involves extraction of entries representing documents belonging to three types of time periods. We classify the term-lists into three types depending on the time-range they span. The lists which cover five years are classified as *large* signifying longer time-ranges, the ones which span 2 years belong to the *medium* range and finally *small* time-ranges span for one year. Further the length of the lists are regulated by selecting or dropping parts of the *large*, *medium* and *small* lists. Very long lists are realized by no document removals, long lists comprise 10% of the original lists, and then we consider 1% and 0.1% of the entire lists are relatively shorter and very short lists.

For every combination of index-list length and entry lifetime a sample list is obtained. On each of these selected sample lists we run the three algorithms for gamma values ranging from 1.1 to 2.7 with an increment 0.1 and observe their responses in terms of overall replication achieved. For each of these combinations a fixed number of representative lists are randomly chosen from the original data set. The average over the replication values obtained from these samples from each scenario is then calculated for each given gamma. The graphs presented later in this section refer to these average replication values

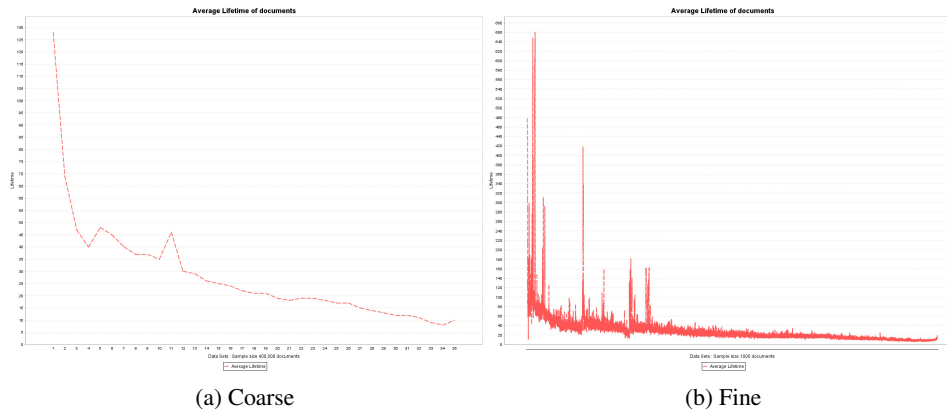


Figure 5.1: Lifetimes Analysis

The experiments were conducted on a 2.66GHz Intel Core2duo processor with 2 GB of memory.

5.2 Analysis of the dataset

We analyzed the Wikipedia dataset about the nature of the lifetimes distribution, response of varying system variables due to partitioning. The total number of document versions in the entire collection is nearly 14 million and the entire time-span of the collection is nearly 5 years. We take sets of fixed number of documents over the entire time-span and calculate the average document or version lifetimes for each of these sets. Figure 5.1 plots two kinds of samples: coarse and fine grained samples of the Wikipedia document lifetimes. The coarse-grained samples refer to sets of 400,000 documents and fine-grained samples refer to sets of 1000 documents. The documents are first sorted according to their respective begin times and then iteratively sampled. This leads to the observation that the documents which existed before have a longer life-time than the ones which came into existence later. There however are some initial versions which are not changed over a long time attributing to peaks at the beginning of the plot.

5.2.1 Fixed and Greedy Partitioning (unbounded)- Nature of Blowup, Reconstructibility and Replication

Although we look at the algorithms which limit user-specified blowup of the index while maximizing the replication, we also looked at a few heuristic methods liked fixed partitioning, unbounded greedy and performance guarantee method [14]. Fixed partitioning considers partitions made at fixed time intervals. The idea behind unbounded greedy partition is that for a given start time of a partition, URR values are calculated for all allowable end time points for the partition. The choice

of the end time point is based on the highest URR value of the partition. The end point of this partition becomes the start point for the next partition which again chooses its end time point greedily based on the highest URR values of the allowable end time points. Figure 5.2 shows the response of replication and URR of the unbounded greedy method with increasing blowup. The thing to note here is that we plot blowup on the x-axis as opposed to the gamma value in the heuristic methods which have bounded solutions. We observe that as expected with increase in the degree of partitioning we have higher blowup along with higher replication. The maximum-minimum URR on the other hand decreases owing to the fact that closely placed partition boundaries result in lesser number unique entries (left and right) since most of the entries which have a longer span would become strike-through entries.

The performance guarantee (PG) approach is worth mention here since the replication values reported by PG is nearly the same as that of the unbounded greedy approach. The plot in Figure 5.3 comparing PG and unbounded greedy shows that increased replication results in increased blowup due higher degree of partitioning in the system. Secondly it also shows the close correspondence between both the methods although both these methods optimize different objective functions. The performance guarantee method minimizes entire index space while the unbounded greedy improves URR locally.

However all the aforementioned methods work unregulated and cannot re-adjust their partitioning according to a higher or lower allowable blowup. We however use lessons learnt from these approaches to design the other heuristic algorithms which are bounded, regulated and can alter system state according to changing user input.

5.3 Results and Comparison

We simulated varying lengths of index lists containing entries spanning short, medium and long lifetimes. We considered duration of one year for entries relevant to shorter time-spans, three years for medium spans and five years (the entire lifetime of the collection) for the long time-spans. The representation of the lifetimes of the entries was made by using lifetime matrices. The lifetime matrix of a collection is a $n \times n$ matrix where n is the end time of the collection in question. Thus the (i,j) th cell of the matrix contains the number of documents which begin at time i and end at j .

Simulating an index list for short lifetimes was done by randomly selecting sub-matrices of size 365×365 from the parent matrix (which contains entire lifetime information). We do the same for medium spans and consider the entire matrix for the long lifespans. To simulate the different lengths of list we randomly drop entries until we reach the size of the list desired. We classify the lists into *very long lists* where we do not drop any of the entries, *long lists* which are 10% of the entries, *medium lists* which are 1% of the entries and *small lists* which are

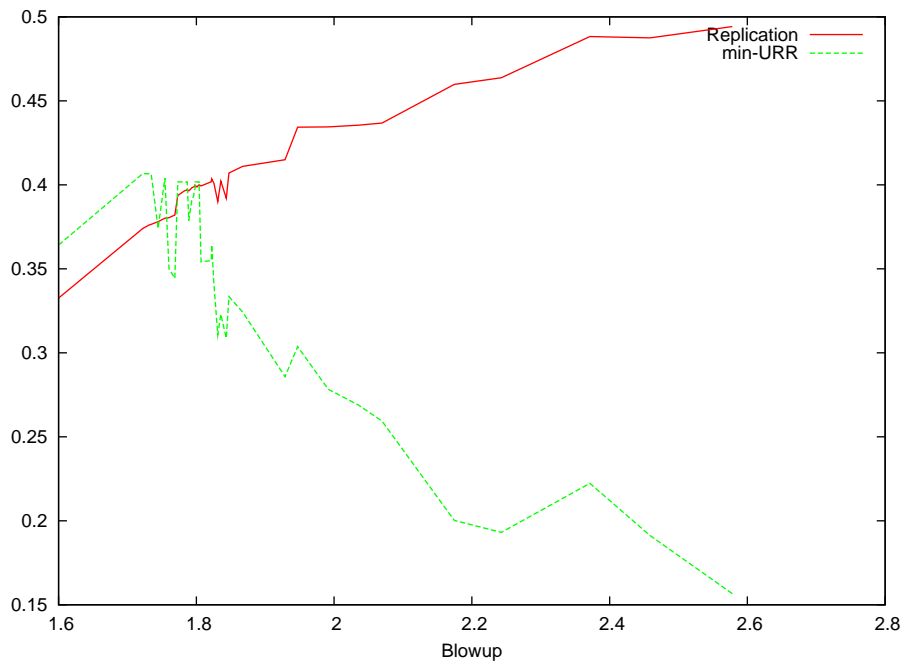


Figure 5.2: Unbounded Greedy Partitioning - Response of Blowup and Replication

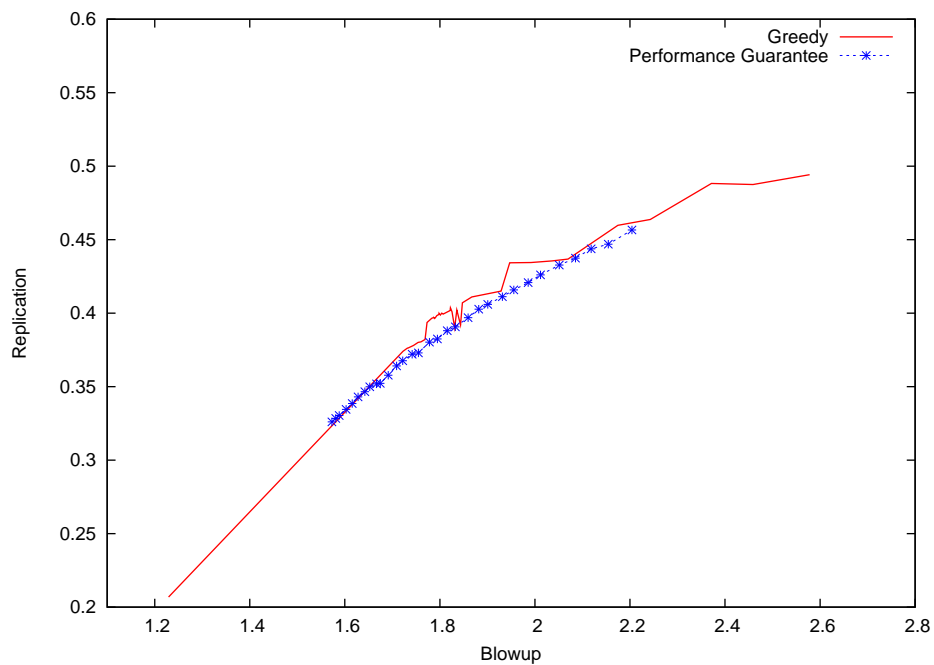


Figure 5.3: Unbounded Greedy Partitioning vs Performance Guarantee: Replication

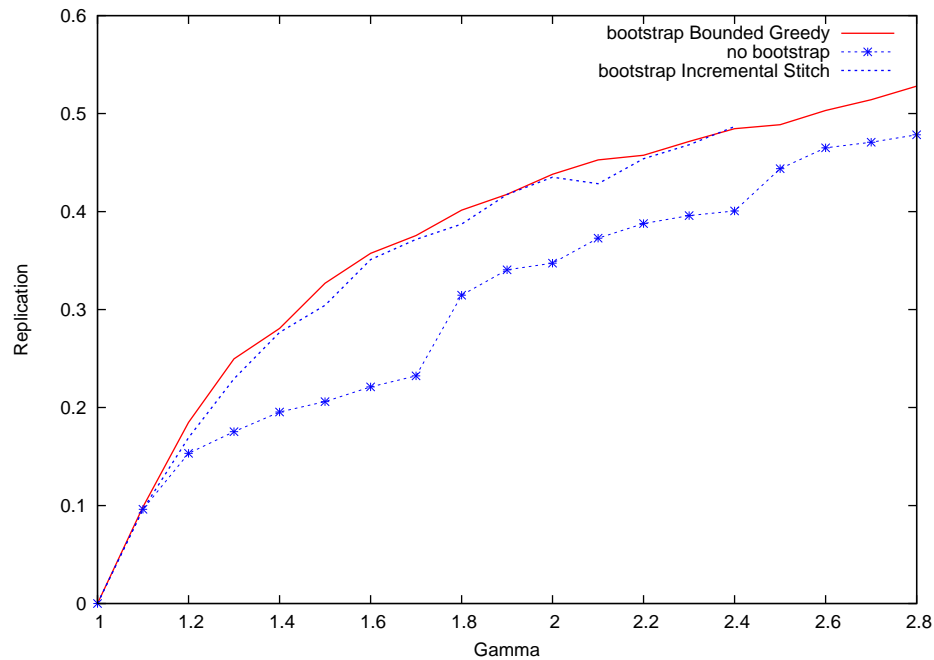


Figure 5.4: Bootstrapped Incremental Stitch and Bootstrapped Bounded Greedy

0.1% of the total number of entries.

We observe that randomized approach involving simulated annealing fares the worst as compared to the others in most of the scenarios. Additionally, we also analyze how simulated annealing can be used to better the results of bounded greedy and incremental stitch by using the partitioning given by them to bootstrap the simulated annealing. Figure 5.4 shows these gains of bootstrapping the simulated annealing approach with initial configurations obtained by bounded greedy or incremental stitch. We used the entire Wikipedia time-range including all the document versions without any drops until gamma values of 2.8.

For rest of the lists we run the partitioning algorithms for gamma values of 1 through 2.6 and plot the replication attained. Each of the plots show four curves for Simulated annealing, Incremental stitch, Simulated annealing bootstrapped by incremental stitch and finally Bounded greedy respectively.

5.4 Analysis

We observe from the results that Bounded Greedy and Incremental Stitch consistently outperform the Simulated Annealing based approach. Figure 5.4 shows the effect of bootstrapping Simulated annealing with partitioning obtained from Bounded Greedy and Incremental Stitch for the entire collection lifetime.

Figure 5.5 is the plot on the dataset of entries which have lifetimes until five

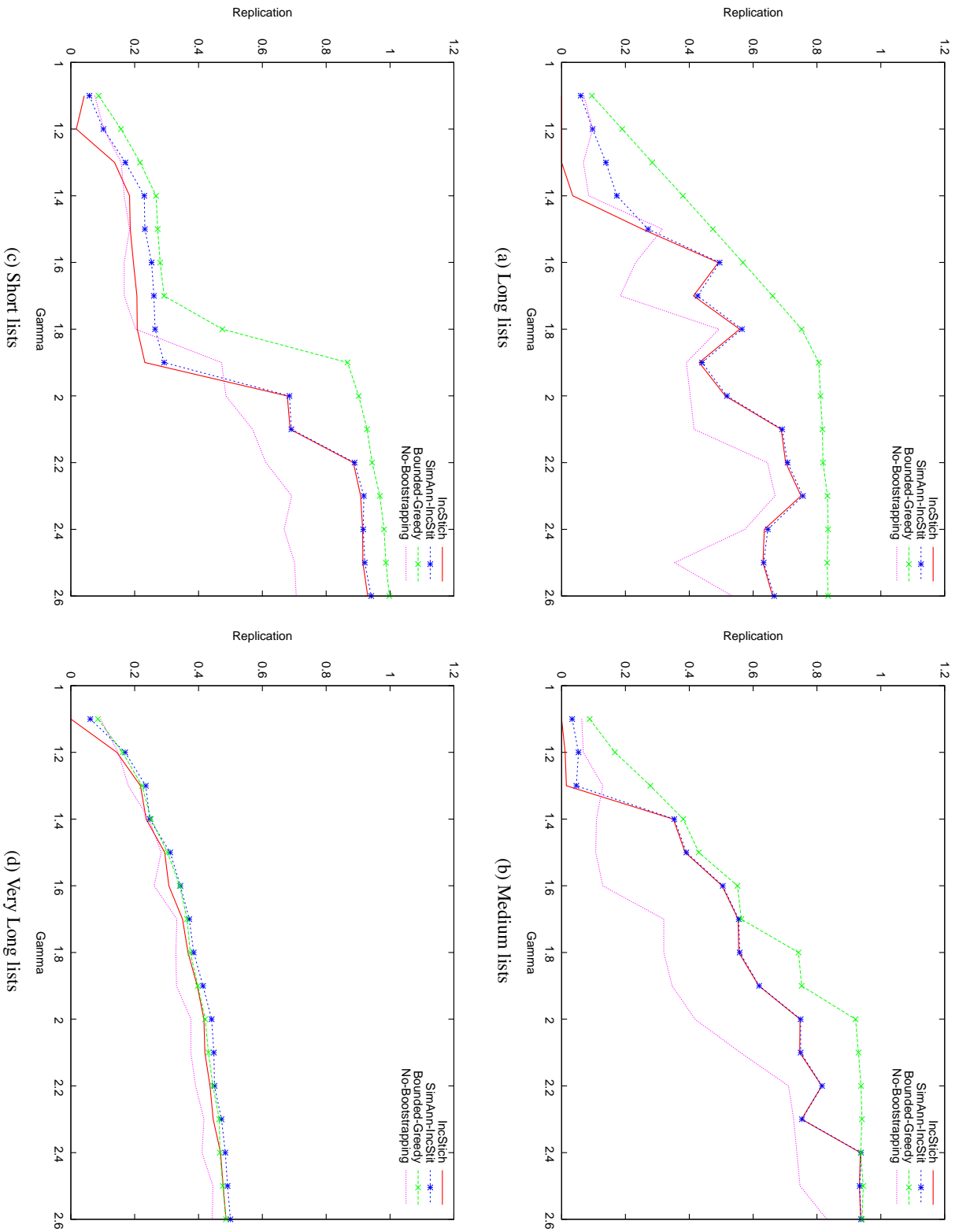
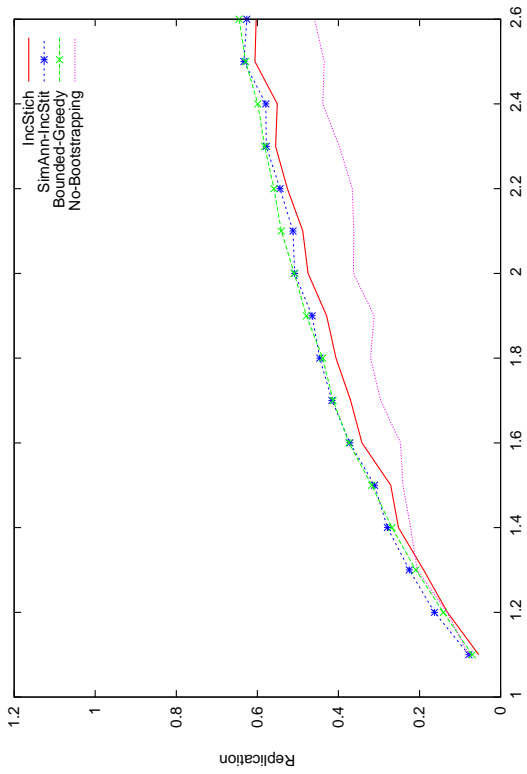
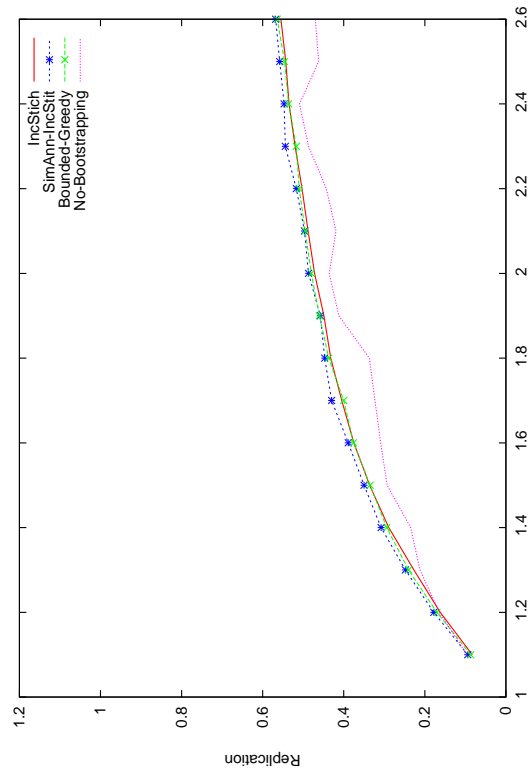


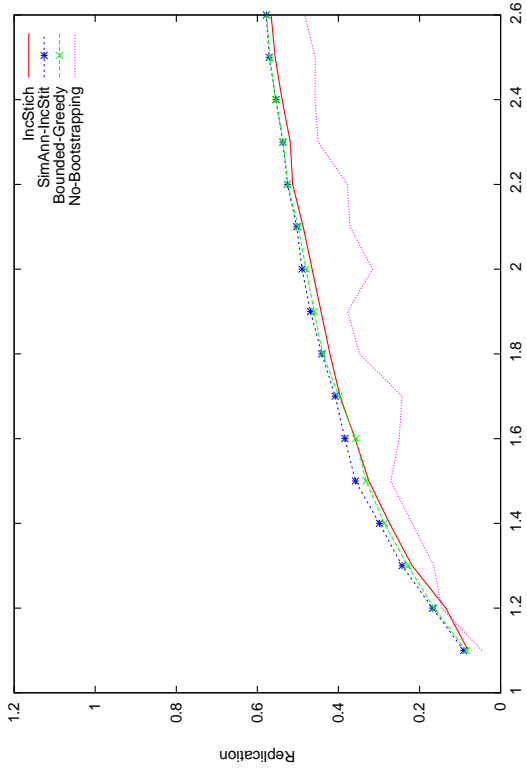
Figure 5.5: Effect of Partitioning: Lists for 5 years



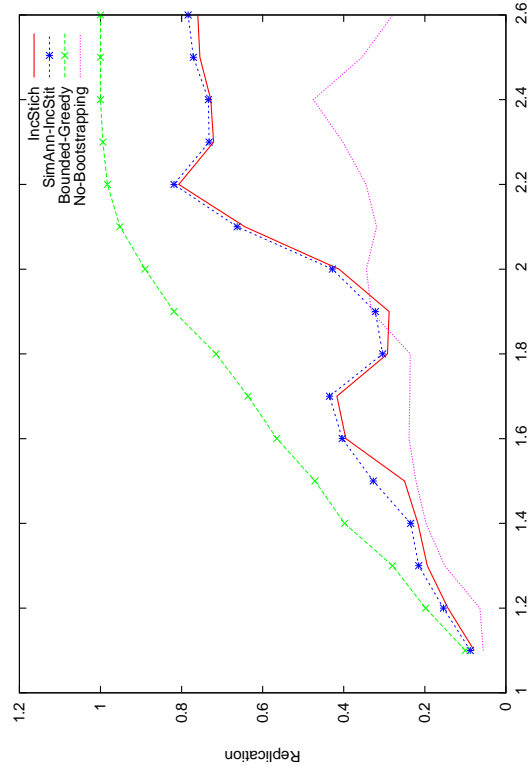
(a) Long lists



(b) Medium lists



(c) Short lists



(d) Very Long lists

Figure 5.6: Effect of Partitioning: Lists for 2 years

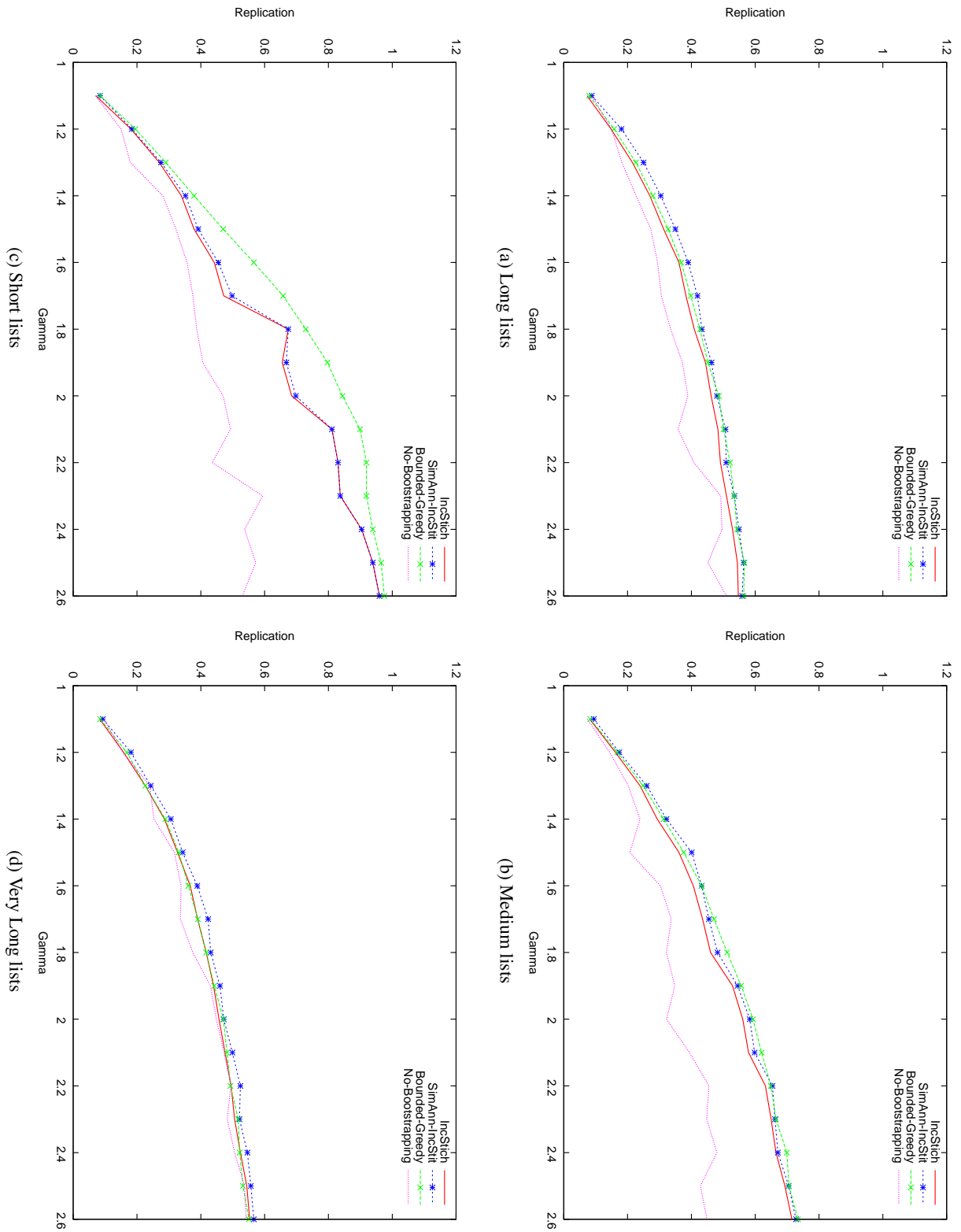


Figure 5.7: Effect of Partitioning: Lists for 1 years

years. It shows that lists which contain entries having long lifespans when partitioned temporally react best to the bounded greedy approach. We see in Figure 5.5a, for long lists, and Figure 5.5c, for short lists, bounded greedy clearly dominates the others while in Figure 5.5d, very long lists, all the methods perform equally well. Short lists in Figure 5.5c contain 14000 have a sharp growth at gamma value 2.0 and report almost 100% replication at gamma greater than 2. This growth essentially shows the improvement of replication with a slight increase in gamma values. This can serve as an important guiding point when specifying gamma for the index partitioning. Bounded greedy also works well for long lists with almost a million entries with incremental stitch showing bumpy behavior. This bumpy behavior might be attributed to two reasons; firstly the stitch operation which leads to sub-optimality and secondly its the averaging over the samples obtained by random dropping of entries.

Bounded greedy works significantly well with short lists of medium timespan with longevity of two years as shown in Figure 5.6c. On the other hand for the medium sized(Figure 5.6b) and long lists(Figure 5.6a) both Simulated Annealing bootstrapped by Incremental Stitch and bounded greedy give similar results. Finally, we see a similar trend in list with entries having a short lifespan in Figure 5.7 as well where all the methods perform as well as each other with bounded greedy working slightly better for short length lists (Figure 5.7c).

We hence observe that the bounded greedy approach performs well in most of the scenarios of different list-lengths and entry lifetimes. The bounded greedy algorithm is also the fastest among the other heuristic algorithms. It clearly is the best choice for short length lists with any type of entries with a possibility of even more improvement if its partitioning is used as a seed set to bootstrap simulated annealing. Incremental works well when the size of the lists are medium to long with minor fluctuations due to the stitch operation.

Chapter 6

Conclusion and Future Directions

In this thesis we made a case for large scale web archiving and the need for a scalable, distributed and peer-to-peer solution for storing, maintaining and indexing data. Web archiving is essential for variety of reasons, primarily for preservation of content reflecting socio-cultural heritage for future generations. Recent studies report massive rate of content evolution in the web. Not only is content being created at a high rate, its rate of loss of content is something which makes the web extremely ephemeral. The challenges of realizing such a large scale web archiving hence includes efficient collection of highly evolving data and available and persistent storage. We argued that making the archive searchable is an essential step towards analysis and usability of the archives. We developed essential methods to support time-travel search in a distributed setup and discussed both architectural and algorithmic aspects pertaining to it.

The contributions of this thesis can be classified into three major parts. Firstly, we presented a framework which is a peer-to-peer solution for building a scalable, persistent and available web archive. We concentrated especially on index organization in the distributed setup which makes retrieval over a p2p setting possible. Secondly, we looked at index partitioning along the time-axis to improve query efficiency and to enable faster index-reconstruction in case of peer loss. We introduced replication of index entries specific to our setup and proposed an optimization problem which ensured maximum replication given a certain user controllable blowup. We also prove the hardness of the proposed problem and motivate the need for approximate solutions. And finally, we propose heuristic algorithms for the Maximum-Replication problem and present experimental evaluations and simulations.

6.1 Discussion

We proposed a three layer decoupled architecture for a searchable peer-to-peer Web Archival System with the layers being responsible for collection or crawling, storage and indexing respectively. The decoupling of responsibilities means that

each layer can focus on its responsibilities while adhering to a fixed set of contractual interfaces with the other layer or layers. This allows additional freedom of design for each of these layers, considering the rest of the system as a black box, and exposing only the functionality of interest to the other layers. There has been some active research on distributed and peer-to-peer storage (refer Chapter 2 for more details) and most of them stress on availability of data and ease of access and hence find similar application in our setting. The separation of concerns by layering would then make it possible to try out these storage stories without any serious disruptions to the other functionalities.

The realization of such a system involves with dealing with various conflicting design issues and addressing every scenario befittingly is difficult or perhaps impossible. We focused on a specific set of requirements and make judicious assumptions about others in constructing a peer-to-peer framework for storage and search organization. We specifically concentrated on the layout of the indexing layer and peer organization to support time-travel queries. In doing so we assumed issues such as load balancing, query rates and update rates as orthogonal issues. These however basis for challenging research questions and we hope to answer them in future work.

We dealt with different kinds of peer organization for the indexing layer like mutli-ring architectures (term-time partitioning organization, time-term partitioning organization), CAN [48] style peer layout and other algorithmic approaches like space filling curves. We describe Te-Ti and Ti-Te organizations in detail in Chapter 3. We adopted the Te-Ti as an organization of choice because of its natural advantages in index partitioning and also because of the fact that it had a natural structure for query load balancing with easier management.

In formulating partitioning strategies for the entire distributed TTX we first defined system properties like blowup incurred due to partitioning, replication and reconstructibility of an index list in case of peer loss. We considered, for the time being, issues pertaining to local peer capacity, load balancing, adjacent peer failures as orthogonal to our problem statement. The Maximum-Replication problem was defined as an optimization problem which intends to maximize the replication of the entire index keeping the blowup under a respectable user defined limit. We showed that the problem is NP-hard and thus warranted use of approximation solutions. We considered three kinds of heuristic algorithms in this regard. Firstly, we considered a greedy algorithm, which we termed as the bounded greedy approach, which partitions at time-points with highest replication gains bounded by a γ bounded space limit. Secondly, we looked at the incremental stitch algorithm which progressively computes optimal solutions for subproblems and combines them via the *stitch* operation. Finally, we considered a randomized approach based on simulated annealing for partitioning.

The experimental evaluations for the heuristic algorithms considered various lengths of lists with different entry types. The bounded greedy approach has the best running time and it also fares better than the others for short and medium sized lists. Incremental stitch works well for very long time duration lists, with the

algorithm returning optimal solutions for time-periods less than the step window size. However, the inefficiencies draw from the fact that the stitch operation result in sub-optimality which might be arbitrarily bad hence requiring us to consider a neighborhood of window sizes and choosing the best among them. Simulated annealing in isolation does not fare as well as the other two methods, but when it is bootstrapped with an initial good configuration it can improve the baseline considerably. The price one has to pay though is of the higher time-complexity. The results show that when incremental stitch is used as a baseline to bootstrap simulated annealing it outperforms bounded greedy for long and very long lists but only by a small margin. Thus bounded greedy seems to be the better algorithm in most cases and if time is not at premium bounded greedy approach can still be used to bootstrap simulated annealing for better results.

6.2 Future Work

There are various directions for continuing the current work. On the architectural level we have considered a number of issues as “black-box” issues such as load balancing, query rates, update rates, data migration, peer availability and individual peer capacity. These form a basis for further research into some of the other design issues affecting a web-archival system. We introduced crawling as an essential component for information collection and harvesting and touched upon the possibility of human-assisted crawling. One issue with human-assisted crawling is that of lack of central management which introduces asynchronous crawling and redundant information capturing. This requires reconciliation of various versions at the storage layer which includes identification of duplicates and accurate time based versioning of the content.

We also proposed the Maximum Minimum Reconstructibility (MMR) problem which aims to improve the overall reconstructibility of a partitioning. This aims at overcoming cases when limited partitions in a given partitioning account for most of the system replication and most of the other partitions are left with a small amount of replicated entries. Establishing the complexity of the MMR problem and finding possible solutions would be the next steps in this regard. We can also consider reconstructibility and replication together as a system property and aim at maximizing their product as an objective function. This might give us an alternative estimate of how much one partition can contribute to a given query. This becomes important for scenarios where only a few peers are queried for partial results rather than a full fledged query routing operation over many peers.

Finally, implementation of the proposed architecture might open up numerous challenges ranging from system level issues like synchronization, network latency, query routing to algorithm issues like local query processing and other algorithms for index partitioning. But its difficult to predict any fundamental factors affecting the fate of web-archival systems due to external reasons like trust, security, copyright issues at this stage. Only time will tell.

Bibliography

- [1] Hanzo archives. <http://www.hanzoarchives.com/>.
- [2] Heritrix. <http://crawler.archive.org/>.
- [3] Internet archive. <http://archive.org>.
- [4] lyman. <http://www.clir.org/pubs/reports/pub106/web.html>.
- [5] nutchwax. <http://archive-access.sourceforge.net/projects/nutch/index.html>.
- [6] Gnutella website. <http://www.gnutella.com/>, January 2007.
- [7] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Puceva, and Roman Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD - Special Interest Group on Management of Data Rec.*, 32(3):29–33, 2003.
- [8] E. Adar, M. Dontcheva, J. Fogarty, and D. Weld. Zoetrope: Interacting with the Ephemeral Web. In *Proc. of ACM UIST*.
- [9] Xu Andrzejak. Scalable, efficient range queries for grid information services, 2002.
- [10] Mayank Bawa. Lsh forest: self-tuning indexes for similarity search. In *WWW (International World Wide Web Conference)*, pages 651–660. ACM Press, 2005.
- [11] Matthias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Minerva: collaborative p2p search. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1263–1266. VLDB Endowment, 2005.
- [12] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. A Time Machine for Text Search. In *Proc. of ACM SIGIR*.
- [13] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. FluxCapacitor: Efficient Time-Travel Text Search. In *Proc. of*.

- [14] K. Berberich, S. Bedathur, and G. Weikum. Efficient Time-travel on Versioned Text Collections. In *Proc. of GI-Fachtagung fr Datenbanksysteme in Business, Technologie und Web (BTW)*, 2007.
- [15] K. Berberich, S. Bedathur, and G. Weikum. Tunable Word-Level Index Compression for Versioned Corpora. In *Proc. of Workshop EIIR*, 2008.
- [16] Michael K. Bergman. The deep web: Surfacing hidden value. *Journal of Electronic Publishing*, 7(1), August 2001.
- [17] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. Totalrecall: System support for automated availability management. In *Proc. of ACM/USENIX NSDI*, 2004.
- [18] Ashwin R. Bharambe. Mercury: Supporting scalable multi-attribute range queries. In *In SIGCOMM*, pages 353–366, 2004.
- [19] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–??, 2001.
- [20] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. P-tree: a p2p index for resource discovery applications. In *In WWW Alt. 04: Proc. of the 13th Int. World Wide Web conference on Alternate track papers and posters*, pages 390–391. ACM Press, 2004.
- [21] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: an efficient and robust p2p range index structure. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 223–234, New York, NY, USA, 2007. ACM Press.
- [22] Francisco Matias Cuenca-acuna, Richard P. Martin, and Thu D. Nguyen. Autonomous replication for high availability in unstructured p2p systems. In *In Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, 2003.
- [23] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM.
- [24] Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range queries in trie-structured overlays. In *In P2P05: Proceedings of the 5th International Conference on Peer-to-Peer Computing*, page 05, 2005.
- [25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter

- Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.
- [26] Oren Dobzinski, Anat Talmy, Dahlia Malkhi, Moni Naor, and David Ratajczak In [viceroy. Viceroy - on the implementation of a peer to peer network. Technical report, 2003.
- [27] Peter Druschel. Past: A large-scale, persistent peer-to-peer storage utility. In *In HotOS VIII*, pages 75–80, 2001.
- [28] Jun Gao and Peter Steenkiste. An adaptive protocol for efficient support of range queries in dht-based systems. In *In ICNP 04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP04*, pages 239–250. IEEE Computer Society, 2004.
- [29] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Abbadi. approximate range selection queries in peer-to-peer systems. In *In Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.
- [30] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *In Proc. of NSDI*, 2005.
- [31] Heritrix Archival Crawler. <http://crawler.archive.org/>.
- [32] E. Herder. Characterizations of User Web Revisit Behavior. In *Proc. of Workshop on Adaptivity and User Modeling in Interactive Systems*, 2005.
- [33] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *In VLDB*, pages 661–672, 2005.
- [34] Yuh-Jzer Joung, Chien-Tse Fang, and Li-Wei Yang. Keyword search in dht-based peer-to-peer networks. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 339–348, Washington, DC, USA, 2005. IEEE Computer Society.
- [35] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
- [36] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage . In *Proc. of ASPLOS*, 2000.
- [37] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. Irlbot: scaling to 6 billion pages and beyond. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 427–436, New York, NY, USA, 2008. ACM.

- [38] Ben Leong, Barbara Liskov, and Eric D. Demaine. Epichord: Parallelizing the chord lookup algorithm with reactive routing state management. In *In Proceedings of the 12th International Conference on Networks*, pages 1243–1259, 2004.
- [39] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proc. of ACM SIGMOD*, 1989.
- [40] Toan Luu, Fabius Klemm, Ivana Podnar, Martin Rajman, and Karl Aberer. Alvis peers: a scalable full-text peer-to-peer retrieval engine. In *P2PIR '06: Proceedings of the international workshop on Information retrieval in peer-to-peer networks*, pages 41–48, New York, NY, USA, 2006. ACM.
- [41] Petros Maniatis and David S. H. Rosenthal. The lockss peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23:2005, 2005.
- [42] Silvano Martello and Paolo Toth. Approximation schemes for the subset-sum problem: Survey and experimental analysis.
- [43] H. Miranda, S. Leggio, L. Rodrigues, and K. Raatikainen. Epidemic dissemination for probabilistic data storage. In R. Baldoni, G. Cortese, F. Davide, and A. Melpignano, editors, *Emerging Communication: Studies on New Technologies and Practices in Communication*, volume 8 of *Global Data Management*. IOS Press, 2006.
- [44] A. Ntoulas, J. Cho, and C. Olston. What's New on the Web?: The Evolution of the Web from a Search Engine Perspective. In *Proc. of WWW*, 2004.
- [45] J. X. Parreira, C. Castillo, D. Donato, S. Michel, and G. Weikum. The JXP Method for Robust PageRank Approximation in a Peer-to-Peer Web Search Network. *VLDB Journal*, 17(2), 2008.
- [46] Sriram Ramabhadran and Joseph M. Hellerstein. Prefix hash tree: An indexing data structure over distributed hash tables. Technical report, 2004.
- [47] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.
- [48] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.
- [49] Robots Exclusion Standard. <http://en.wikipedia.org/>.

- [50] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching.
- [51] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.
- [52] O. D. Sahin, A. Gulbeden, F. Emekci, D. Agrawal, and A. El Abbadi. Prism: indexing multi-dimensional data in p2p networks using reference vectors. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 946–955, New York, NY, USA, 2005. ACM.
- [53] B. Salzberg and V. Tsotras. Comparison of Access methods for Time-evolving Data. *ACM Computing Surveys*, 31(2):158–221, 1999.
- [54] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. of ICDE*, 2001.
- [55] A. Singh, M. Srivatsa, L. Liu, and T. Miller. Apoidea: A decentralized peer-to-peer architecture for crawling the world wide web. In *Proc. of ACM SIGIR*, 2003.
- [56] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, 2001.
- [57] Tang. A short survey on p2p data indexing, 2008.
- [58] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks, 2002.
- [59] Christos Tryfonopoulos, Stratos Idreos, and Manolis Koubarakis. Libraring: An architecture for distributed digital libraries based on dhds. In *In ECDL*, pages 25–36. Springer-Verlag, 2005.
- [60] H. Weatherspoon, C. Wells, P. R. Eaton, B. Y. Zhao, and J. D. Kubiatowicz. Silverback: A Global-Scale Archival System. Technical Report UCB//CSD-01-1139, U.C. Berkeley, 2000.
- [61] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *In Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, 2002.
- [62] Klaus Wehrle and Ralf Steinmetz. Introduction. In *Peer-to-Peer Systems and Applications*, pages 1–5, 2005.
- [63] Ragib H. Zahid. A survey of peer-to-peer storage techniques for distributed file systems.

- [64] Alexander Zangerl. Tamper-resistant replicated peer-to-peer storage using hierarchical signatures. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*, pages 50–57, Washington, DC, USA, 2006. IEEE Computer Society.
- [65] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.
- [66] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over dht. In *In Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS06, 2006)*.