EverLast: A Distributed Architecture for Preserving the Web

Avishek Anand
Max-Planck Institute for Informatics,
Saarbrücken, Germany
aanand@mpi-inf.mpg.de

Srikanta Bedathur
Max-Planck Institute for Informatics,
Saarbrücken, Germany
bedathur@mpi-inf.mpg.de

Klaus Berberich
Max-Planck Institute for Informatics,
Saarbrücken, Germany
kberberi@mpi-inf.mpg.de

Ralf Schenkel
Saarland University,
Saarbrücken, Germany
schenkel@mmci.uni-saarland.de

Christos Tryfonopoulos
Max-Planck Institute for Informatics,
Saarbrücken, Germany
trifon@mpi-inf.mpg.de

ABSTRACT

The World Wide Web has become a key source of knowledge pertaining to almost every walk of life. Unfortunately, much of data on the Web is highly ephemeral in nature, with more than 50-80% of content estimated to be changing within a short time. Continuing the pioneering efforts of many national (digital) libraries, organizations such as the International Internet Preservation Consortium (IIPC), the Internet Archive (IA) and the European Archive (EA) have been tirelessly working towards preserving the ever changing Web.

However, while these web archiving efforts have paid significant attention towards long term preservation of Web data, they have paid little attention to developing an global-scale infrastructure for collecting, archiving, and *performing historical analyzes* on the collected data. Based on insights from our recent work on building text analytics for Web Archives, we propose *EverLast* , a scalable *distributed framework* for next generation Web archival and *temporal text analytics over the archive.* Our system is built on a loosely-coupled distributed architecture that can be deployed over large-scale peer-to-peer networks. In this way, we allow the integration of many archival efforts taken mainly at a national level by national digital libraries. Key features of EverLast include support of time-based text search & analysis and the use of human-assisted archive gathering. In this paper, we outline the overall architecture of EverLast, and present some promising preliminary results.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing methods; H.3.1 [**Information Storage and Retrieval**]: Library AutomationLarge text archives

General Terms

Algorithms, Design, Experimentation

Keywords

Web Archives, Crawling, Indexing, Time-travel search

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

JCDL'09, June 15–19, 2009, Austin, Texas, USA.

Copyright 2009 ACM 978-1-60558-322-8/09/06 ...$5.00.

1. INTRODUCTION

The World Wide Web has become a key source of information pertaining to all facets of life, starting from business, entertainment, science to politics, culture etc. It is evolving and growing at a high rate [24], and the loss of born-digital information on the Web is significant. Archiving the Web is an effective way to counter the decay seen on the Web. Web archives can capture the timelines of entities and topics on the Web, and they reflect the evolution and trends of our society, economy, and culture that are of high interest to scientific scholars, market researchers, patent offices, and others.

From a utilitarian point, web archives (WA) serve purposes very different from standard web-search engines. They not only provide the *latest* or near-latest view of the Web, but are also the source of the evolutionary history of the Web. History must be *completely* and *correctly* captured, stored forever in a tamper-resistant manner, must be accessible without fear of censorship or blockage, and most importantly, must be seamlessly searchable in both text and time axes along with support for different forms of time-aware mining. Further, web archiving must also be self-sustaining, and not be placed under the control of any single corporation which could possibly end up with the ability to "rewrite history". This, along with the fact that archiving efforts are currently run at a national level through archival tools developed or sponsored by national digital libraries, introduces the need for a distributed, scalable and durable framework that will manage and make the archived content available.

1.1 Web Archiving - Today

Starting from the pioneering efforts of the Swedish Royal Library and other national libraries in Europe [2], web archiving has been steadily gaining importance. The best known large-scale effort to date has been that of the Internet Archive (IA), which has an accumulated size of more than 500 Terabytes [1] and is growing at 100 Terabytes per year. However, despite many advances made in web archiving, the state-of-the-art is still far away from reaching its ultimate objective of preserving all digital-born information on the Web for posterity. We have recently been involved in Web archiving projects in co-operation with archivists from the European Archive (EA)[1]. The experiences from these projects have helped us to get a better understanding of the current limitations of web-archiving solutions.

Limited coverage and lack of responsiveness. The Internet Archive (IA) performs large-scale crawls to generate snapshots of the Web, using a fixed set of sites nominated by partner organizations. The largest such crawl so far consists of two billion pages and was completed in 2007 after a six month effort. On the other hand, this was only a small fraction of the Web, which is conservatively estimated to consist of about 27 billion pages [36] – discounting the informative pages that are behind forms and rapidly changing sites such as newspapers etc. More importantly, this also is a *single snapshot* of all the pages crawled, not their complete evolutionary history during the time of data collection. Combined with the fact that these large-scale crawls are done very infrequently, the resulting archives offer very limited coverage of the Web, more so of different versions of pages involved. The alternative of recrawling sites at very high rates is not practical, since it does not honor the crawler-politeness requirements, and can therefore easily be mistaken for a denial-of-service attack.

Store but no access. Current web archives, including IA, focus almost exclusively on the preservation of content, but not on providing access to their archives. The best way to access the IA collection is through their WaybackEngine. Given a URL completely specified by the user, this service presents the version history of the URL, but does not provide any content-querying capabilities. Interfaces provided by other archives are similarly limited.

1.2 Web Archiving - The EverLast Approach

In this paper, we present the design of EverLast, a scalable and durable framework that we are building as a next generation web archiving (WA) infrastructure. EverLast is a loosely coupled distributed architecture that can be deployed over peer-to-peer (P2P) networks, thus making it naturally decentralized and non-authoritarian. We broadly define a P2P network as any large-scale distributed system with self-organization capabilities to counter problems of localized failures (such as a peer disappearing from the network) and to deal with dynamics (content, query, and behavioral). As a consequence, smaller-scale instances of EverLast operating within intranets of organizations/(digital) libraries and data grids or clouds are possible as well.

[1]http://www.liwa-project.eu/

EverLast pursues two main goals: (i) Efficiently collecting and maintaining an archive of the World Wide Web with good coverage, and (ii) Enabling historical text analytics by efficiently processing *Time-travel Keyword Queries* to return a ranked list of relevant documents from a web snapshot in the past. Towards these goals, advances are made in each aspect of the web-archiving life-cycle, viz., capturing or harvesting, storing, and indexing of web documents in a timely fashion. The data and metadata are distributed and managed using a structured P2P network substrate such as Pastry [27] or Chord [32]. We rely on these networks to provide consistent primary key lookups, and robust handling of churn. We briefly present the key features provided by EverLast:

Human-assisted web capture. The evolution of digital content on the Web is highly bursty and non-localized. Most of the time, only web users can locate such "hot" regions of change. Therefore, EverLast combines the standard methods of crawling with human-assisted crawlers – archival plugins built into browsers or proxies which selectively capture Web pages of archival interest along with their timestamps and publish them into the archive at regular intervals.

Time-travel keyword queries. EverLast is built around the need to provide efficient support for time-travel queries, a feature which can be used to build rich temporal analysis on archives.

Distributed archival storage. The P2P research community has proposed peer-to-peer storage architectures aiming at long-term (nearly immortal) storage [21, 28, 35]. Such persistent storage systems that support efficient primary key lookups, although essential, are not sufficient for Web archiving. We augment these persistence solutions with a storage-metadata management layer required for Web archiving.

1.3 An Application Scenario for Everlast

As an example for an application scenario let us consider Marco, a journalist of a local newspaper, living in Marbella, Spain. Marco is writing an article about the environmental policy of the local authorities over the last 10 years, and he is using a variety of sources to prepare the article, including the web pages of the municipality, as well as those of local environmental groups. Apart from the current versions of these Web pages, stating the recent environmental actions, Marco queries the EverLast engine using the URLs of these Web pages for the time frame 1999 – 2009 to receive a chronologically sorted list of pages. By clicking on a specific date, Marco is now able to see and use the page as it appeared at the time it was archived.

However, after lunch, Marco recalls that in November 2003, there was a big debate in the local community about the construction of a parking lot in the place of a neighborhood grove. Since this is an important issue that he would like to add in the article, Marco specifies the time as November 2004, and issues the query "parking lot grove Marbella" to the EverLast engine, to receive pages of November 2004, containing the specified keywords.

The same evening, Marco has the idea of further strengthening his article by comparing the environmental policy of his municipality, with the ones from other European cities.

He decides that recycling is one of the most important showcases for environmental actions, and wants to use it as a working example for this comparison. He resorts to EverLast, specifies the time frame 1999 – 2009, and issues the query "recycling (Patras OR Lyon OR Aachen)" to receive documents from the given time frame stating the recycling actions of these cities.

Clearly, Marco would benefit from utilizing a system that is able to provide access to archived versions of web pages, either by specifying the page URL, or through free text search. Furthermore, the integration, in a single search interface, of archival efforts conducted at a national level, and typically lead by national digital library organizations, would be a valuable tool, beyond anything supported in current archival efforts. In our application scenario, each local archival site (e.g., digital library or institution) would maintain its own EverLast peer, that would provide the EverLast functionality, and act as an access point the content provider. Users utilizing EverLast may also utilize the archival plug-in to archive web pages of interest that they are visiting. In this way, users may act as crawlers that are able to capture dynamic and interesting regions of the Web, and increase the coverage of the Web archiving sites.

1.4 Organization

The remainder of this paper is organized as follows. Section 2 discusses related research with respect to P2P storage systems, distributed crawling and access structures for managing time-evolving data, while Section 3 presents the underlying data and query model. Sections 4 and 5 present the EverLast architecture and the associated protocols respectively. In Section 6 we provide an experimental evaluation of archival coverage in EverLast. Finally, Section 7 concludes the paper.

2. RELATED WORK

The design of EverLast builds on research from different domains. Peer-to-peer storage with the promise of high availability and durability has been an active area of research [14, 15, 28, 35]. These efforts focus only on building reliable storage services but do not address the necessary higher-level functionality for querying and analytics.

Distributed crawling [30, 31] 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 fact, web crawlers may sometimes be tuned to maximize the number of detected links as anchor texts do already lead to index entries for the search engine. 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.

Various access structures for managing time-evolving data in centralized settings have been proposed. Two examples are the TSB-Tree [23], which recently regained attention in [22], and the MVB-Tree [8]. For a comprehensive survey of these access structures, we refer the reader to [29]. Their applicability in decentralized settings and to manage textual data, however, is not well understood.

P2P Information Retrieval (IR), the ability to perform efficient information retrieval tasks over P2P networks has attracted considerable attention in recent times (see [7, 9, 17, 20, 26, 34] and references given there). To the best of our knowledge, none of them have considered temporal querying and ranking models on evolving data collections.

Finally, within digital library research community there has been interest in building large (centralized) archives of the Web [6] and in developing techniques for smaller-scale personal Web archiving solutions [33]. While both these approaches have text searching capabilities over archives, they do not capture temporal aspect explicitly.

3. TIME-TRAVEL: MODEL AND ISSUES

When it comes to search and exploration of web archives, the time axis plays an important role. This is in contrast to today's standard web search that typically ignores the time axis. In our earlier work [10, 11, 12, 13], we introduced the concept of *time-travel text search* and proposed techniques for its efficient realization in a centralized setting. In this section we recap the underlying data and query model.

3.1 Data Model

We consider timestamped document versions $\mathtt{d}^{\mathtt{t}}$ where $\mathtt{d}$ is the actual document content (e.g., a bag of words) and $\mathtt{t}$ is the associated timestamp. We employ a discrete notion of time allowing us to use non-negative integers for timestamping. In addition, as commonly done in temporal databases [16], we introduce a special value *now* that always points to the current time. The validity time-interval of a document version $\mathtt{d}^{\mathtt{t}}$ is defined as $[\,\mathtt{t}, \mathtt{t}'\,)$ where $\mathtt{t}'$ is the timestamp of the currently known subsequent document version and $[\,\mathtt{t}, \mathtt{now}\,)$ if no such subsequent version exists.

On the Web, accurate timestamping of retrieved documents is difficult, since most web servers do not report truthful last-modification timestamps. In EverLast the timestamping authority is assigned to the crawler. Crawler peers thus timestamp document versions with their time of observation – in other words, the time when the user visited the page via the browser. To avoid identical versions with different timestamps, version-reconciliation mechanisms, as detailed later, can be invoked.

The design choice of letting crawler peers timestamp document versions has important implications. Since we cannot synchronize the activities of crawler peers, they may feed document versions into our distributed archive in an order that is different from their temporal order. As a consequence, unlike most temporal index structures, our system must support out-of-order insertions.

3.2 Query Model

We support *time-travel queries* $\mathtt{q}@[\mathtt{t_b}, \mathtt{t_e}]$ consisting of a *content part* $\mathtt{q}$ and a *time interval of interest* $[\,\mathtt{t_b}, \mathtt{t_e}\,]$. The content part q has the same structure as document versions in the above data model (e.g., a bag of query keywords). When evaluating the *time-travel query*, only document versions that existed at any point in $[\,\mathtt{t_b}, \mathtt{t_e}\,]$ are taken into consideration and are thus potential query results. An interesting special case are *time-point queries* $\mathtt{q}@\,\mathtt{t}$ for which the time interval of interest consists of a single time instant $\mathtt{t}$, thus evaluating the query $\mathtt{q}$ "as of" time $\mathtt{t}$.

The following two examples demonstrate the usefulness of our query model and the value that it adds to web archives in general.

- For an article about Al Gore, a journalist wants to compare Gore's recent statements regarding climate

change and those made during his run for presidency. Many web pages that existed in 2000 have disappeared from the "live" Web in the meantime. However, time-travel queries such as

{al gore climate change}@[2000/01, 2000/12]

help our journalist to retrieve relevant archived documents from our distributed web archive.

- Eddie Electric has lost the manual of his beloved 7-year-old cellphone. Given the age of the phone and the fact that the cellphone producer DunQ has been acquired by another company two years ago, there is little hope of finding the manual on today's Web. When Eddie issues the query

  {DunQ CP825X manual}@[2001/01, *now*]

  against the peer-to-peer web archive, he quickly locates the sought cellphone manual.

In addition to their usefulness as a tool for accessing web archives, time-travel queries provide a powerful platform to support higher-level mining and exploration tasks, as the following example demonstrates.

- FourWheels Inc., a car company, wants to study how customer sentiments towards their company changed during the past ten years. By issuing time-travel queries for the company name and each year in the period of interest, document versions from the respective period can be identified, retrieved, and analyzed further.

4. ARCHITECTURE

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

(a) Crawler peers,

(b) Version-directory peers,

(c) Persistence peers, and

(d) 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 [27] or Chord [32] can be used to organize these overlays. A high-level data-flow model of EverLast is depicted in Figure 1.

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 [18], 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 timelines of all the documents seen so far, keeping document-version-level statistics such as PageRank scores, 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

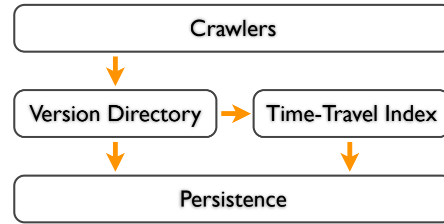

Figure 1: Architecture of EverLast

objects in the system, viz., document versions or inverted (sub)lists, is handled in a 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.

4.1 Crawlers

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 EverLast, 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 crawler 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.

Studies have shown that close to 50% of an individual's browsing activity is page-revisits [19]. 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 [3] 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. In EverLast we currently utilize a proxy server that has been enhanced with archiving abilities to achieve reasonable captures. One of the key technical challenges that requires further research is to preserve the privacy of the user and automatic ways to avoid sensitive content from being archived. Even though a good fraction of such content is sent through secure HTTP which cannot be intercepted by the proxy, more powerful schemes are needed to identify content that must not be archived. 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 timelines.

4.2 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. The lifetime of a version, consisting of a begin and end time, is constructed based on these reported timestamps. To accurately construct a version history or a timeline, we should firstly be able to identify if the content in a page (change detection) has changed. We assume that a version has not undergone a change until another version with different content is served with a later timestamp. This works well in principle in case of versions arriving in sequence (which in reality is rarely the case). Version updates are typically reported out-of-order in case of human-assisted crawling and this makes the problem challenging. For example lets say we have two existing versions for `doc1` at time-points $\mathtt{t}_1$ and $\mathtt{t}_5$.

In case of arrival of an out-of-order update $(\mathtt{doc1}, \mathtt{t}_3)$ where $t_1 \leq t_3 \leq t_5$ whose content is same as $(\mathtt{doc1}, \mathtt{t}_1)$ it is ignored assuming the document has not changed. But now consider the arrival of an update $(\mathtt{doc1}, \mathtt{t}_2)$ where $t_1 \leq t_2 \leq t_3$ whose content is different from $(\mathtt{doc1}, \mathtt{t}_1)$. In such a case we cannot ascertain the end time of the version which begins at t_2 if we do not keep track of the updates after t_2 leading to inaccuracy in timeline construction (in this case the end time is reported to be t_5 whereas it actually should be less than t_3).

Version reconciliation may result in either append, insert or branch operations on the version timeline as well as a simple extension of the current version. If a new version of the `url` is created with timestamp $\mathtt{t}_\mathtt{s}$ after reconciliation, its contents vc are posted into the persistence layer calling $\mathtt{put}(\langle \mathtt{url}, \mathtt{t}_\mathtt{s} \rangle, \mathtt{vc})$ 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 [25] 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 metadata structures, we can achieve these availability demands via eager replication mechanisms commonly suggested in distributed storage research.

4.3 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.

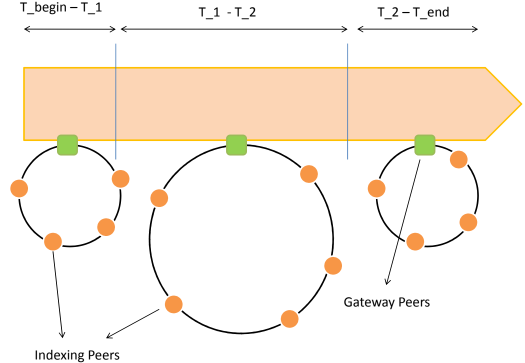

Figure 2: Time-term partitioning of Time-travel Index

EverLast employs a distributed variant of the time-travel inverted index proposed in [10, 13]. We provide a concise overview of the index structure, before describing how it can be adapted to a decentralized setting.

Time-Travel Inverted Index (TTIX)

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

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

where `did` is a document identifier, $[\mathtt{t_b}, \mathtt{t_e})$ is the validity-time interval, and `tf` is the term frequency.

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 $\mathtt{L_v} : [\, \mathtt{t_i},\, \mathtt{t_j}\,)$ thus contains all index entries for term v whose validity-time interval overlaps with $[\, \mathtt{t_i},\, \mathtt{t_j}\,)$. 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. In [10, 11], we proposed techniques that determine temporal partitionings of individual inverted lists that trade-off extra storage-costs and query-processing gains.

Two key benefits of TTIX 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.

Distributed TTIX in EverLast

There are – at least – two ways of adopting the TTIX structure to a distributed setting. One of the natural ways is to partition the index first by time and then by term, thus

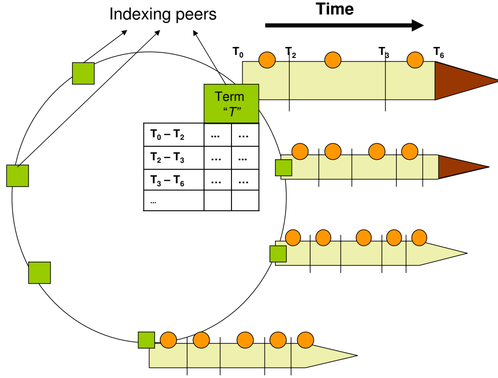

Figure 3: Term-time Index Partitioning in EverLast

yielding a **time-term-partitioned index** (Ti-Te for short), illustrated in Figure 2. 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.

However, in a 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. Even more importantly, this partitioning also suffers from skewed distribution of queries – it is likely that most queries will be about recent past. Thus, peers responsible for the recent past will tend to be overloaded, and as time progresses they are under utilized.

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 metadata 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 TTIX organization.

Figure 3 shows how EverLast implements the Te-Ti indexing. Indexing peers (denoted by green boxes) are organized into an overlay, with each peer housing the metadata for all the temporal partitions of a term-specific inverted list. In addition to the boundaries of the partition, the metadata 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 (orange shaded nodes), which can be used to retrieve the entries during query processing.

TTIX Partitioning Strategies in EverLast

Since peers participating in EverLast are completely autonomous, it is possible that one of the partitions is temporarily unavailable during query processing. A straightforward solution is to simply keep multiple copies of each partition placed in different peers. However, due to replication of entries spanning across partitions, we already introduce some amount of redundancy into the index. How can we exploit this inherent redundancy in TTIX to avoid complete copying of partitions?

To answer this question, we again observe that the index entries whose validity-time interval spans the partition boundary are available in *at least two* partitions. Even when one of the partitions is lost, corresponding index entry is still *reconstructible* by consulting the neighboring partition.

One way to optimize reconstructibility is to improve the overall replication of the index entries, while keeping the overall index size under check. This can be formulated as an optimization problem, similar to those explored in [10]. The resulting formulation, which we call maximum-replication problem [4], turns out to be NP-Hard (by reducing from subset-sum problem), and even approximation algorithms are not known. Our solution for maximum-replication problem in EverLast is based on greedy heuristics, and can be solved in time linear in the number of entries in the index. It partitions the time-axis greedily at time-points where the index can has maximum number of newly added replicated entries.

4.4 Persistence

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

In EverLast, the timestamps associated with objects is incorporated into defining their primary keys. 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 t_s of the web page `url` as a pair $\langle \mathtt{url}, t_s \rangle$. Similarly, for a index list partition $L_v : [t_i, t_j)$, we assign the pair $\langle v, t_i \rangle$ as the primary key.

4.5 Query Processing

To evaluate a query $q@[t_b, t_e]$, a client needs to first contact the Time-Travel Indexing layer to get the keys to the corresponding inverted list partitions that are stored in the persistence service. If $t_b = t_e$ in the time-travel query, only one persistence peer per term needs to be contacted. On the other hand, if the query is specified over a time-range, multiple persistence peers that hold the inverted lists for overlapping time partitions need to be contacted. In order to speed up the query processing, we can utilize caching techniques [37] at the indexing peer.

5. PROTOCOL SPECIFICATIONS

In EverLast there are four kinds of peers: version directory peers, indexing peers, storage peers and crawler peers. The

version directory peers are responsible for version reconciliation, managing document versions and index lists. The indexing peers are responsible for partitioning of the index and maintaining the index partitions to peer mapping. They, hence, act as a lookup service for routing of queries to the appropriate partitions. The storage peers are responsible for storing the actual archived documents and the time-travel index, and implement the `get` and `put` functions to provide a primary key lookup functionality. Finally, the crawler peers are responsible for offloading the local versions of crawled pages periodically to the version directory peers.

In this setting, we can also distinguish between four kinds of data objects, namely document versions, partitioned index blocks, parts of the version table maintained by the directory peers and the Index Partition Table (IPT), containing partition-to-peer mappings maintained by the index peers. These data objects are resident in the persistence layer and can be accessed by primary key lookups issued to the storage peers. The primary keys for these data objects utilize a hash function, H, on one or more of their properties like URL, term, begin and end times, and are constructed as follows:

- Document versions: $\texttt{H}(\texttt{url}, \texttt{ts})$
- Index partition blocks: $\texttt{H}(\texttt{term}, \texttt{tb}, \texttt{te})$
- IPT entries maintained by Indexing peer for each term: $\texttt{H}(\texttt{term})$

In the following sections we discuss the protocols regulating peer interactions, such as version reconciliation, document version and index updates, query processing, and peer joining and departure, by utilizing the aforementioned primitives.

5.1 Version Reconciliation and Document Version Updates

In EverLast, the crawler peers maintain a local repository of content crawled along with their time-of-crawl time-stamps. When a crawler peer wants to add the crawled content to the version directory peers, it creates a $\texttt{addVersion}(\texttt{url}, \texttt{document}, \texttt{timestamp})$ message, and sends it to the version directory peer responsible for the `url`. This peer is found through the mapping function $\texttt{M}(\texttt{url})$. Each version directory peer typically receives numerous $\texttt{addVersion}()$ messages from different crawler peers. It reconciles these versions by referring to the versions table and keeps the version which is consistent with the previous ones.

The directory peer updates the document version by calling the $\texttt{updateVersion}(\texttt{url}, \texttt{timestamp}, \texttt{document})$ function. The version table is then updated according to the version selected and the version is finally written into the persistent storage by using $\texttt{put}(\texttt{H}(\texttt{url}, \texttt{timestamp}), \texttt{document})$.

5.2 Updating the Index

When a directory peers wants to update the index, it issues the $\texttt{lookupTerm}(\texttt{term}, \langle \texttt{did}, \texttt{t}_\texttt{b}, \texttt{t}_\texttt{e} \rangle)$ message, to receive the partition information from the indexing peer that responsible for this. The partition boundaries $\texttt{t}_\texttt{begin}$ and $\texttt{t}_\texttt{end}$ which enclose `timestamp` are looked up from IPT, and are subsequently sent to the directory peer. The data blocks for these partitions are then fetched via the $\texttt{get}(\texttt{H}(\texttt{term}, \texttt{t}_\texttt{begin}, \texttt{t}_\texttt{end}))$ and updated with the new entry $\langle \texttt{did}, \texttt{t}_\texttt{b}, \texttt{t}_\texttt{e}, \texttt{tf} \rangle$. This is then written into the storage layer using the $\texttt{put}(\texttt{H}(\texttt{term}, \texttt{t}_\texttt{begin}, \texttt{t}_\texttt{end}), \texttt{obj})$ where `obj` is the modified index list. Finally, a $\texttt{updateIndex}(\texttt{term}, \langle \texttt{did}, \texttt{t}_\texttt{b}, \texttt{t}_\texttt{e} \rangle)$ message is sent to the indexing peer, which updates the statistics in the IPT.

5.3 Updating the Indexing Layer

In order to keep the IPT in the storage and index layers synchronized, index peers issue a $\texttt{get}(\texttt{H}(\texttt{term}))$ message to the storage layer at regular intervals. The fetched IPT for the corresponding term is then synchronized, if needed, with the current version stored at the indexing peer. Finally, the modified IPT object (`obj`) is written back into the persistence layer by using $\texttt{put}(\texttt{H}(\texttt{term}), \texttt{obj})$.

5.4 Query Processing

In our setting any peer may issue a query of the form $\texttt{Q} = \texttt{q}_1 \ldots \texttt{q}_\texttt{n}@\texttt{t}$, where n is the number of keywords contained in the content part of Q. To resolve this query, the query initiator forwards it to an indexing peer, by sending a $\texttt{processQuery}(\texttt{Q})$ message. When an indexing peer receives a $\texttt{processQuery}()$ message, it extracts the keywords $q_i, 1 \leq i \leq n$, and routes the query to the responsible indexing peers. If a peer responsible for a term $q_i, 1 \leq i \leq n$ is not online at the time of the lookup, the IPT can be fetched by using the $\texttt{get}(\texttt{H}(\texttt{q}_\texttt{i}))$ and the index partition for the time component `t` is looked up. The storage peers are then contacted for the corresponding index lists for the query terms either directly from the IPT or by using the lookup function $\texttt{get}(\texttt{H}(\texttt{q}_\texttt{i}, \texttt{t}_\texttt{begin}, \texttt{t}_\texttt{end}))$ where $\texttt{t}_\texttt{begin}$ and $\texttt{t}_\texttt{end}$ are the partition boundaries which enclose `t`. The index lists are then sent to the query initiator which locally merges the index lists, and ranks the results.

5.5 Node Join and Node Departure

Each overlay in EverLast demands a specific join and departure protocol. The *NodeJoin* operation follows the conventional DHT-based joining protocols, in which the node sends a $\texttt{NodeJoin}(\texttt{nodeID}, \texttt{available_space}, \texttt{role_type})$ message, where `role_type` is either a storage, version directory or an indexing peer. The corresponding overlay proceeds with its own protocol for adding a new peer to the overlay and provides the node with an `overlay_id` as a system-specific identifier. Node departure is similarly invoked before a node intends to leave the overlay by issuing a $\texttt{NodeDepart}(\texttt{overlay_id}, \texttt{role_type})$ message.

6. EXPERIMENTS

In this section we present a series of experiments demonstrating the benefits of augmenting standard archive crawling with human-assisted crawling, and show how index partitioning improves the reconstructability of TTIX through replication.

6.1 Human-assisted Crawling

While we are experimenting with the archive harvesting – the process of capturing versions of a set of URLs – via crawler peers, we provide some evidence to show the effectiveness of human-assisted crawling. We focus on the achieved coverage of different versions of web pages over time during the archive harvesting.

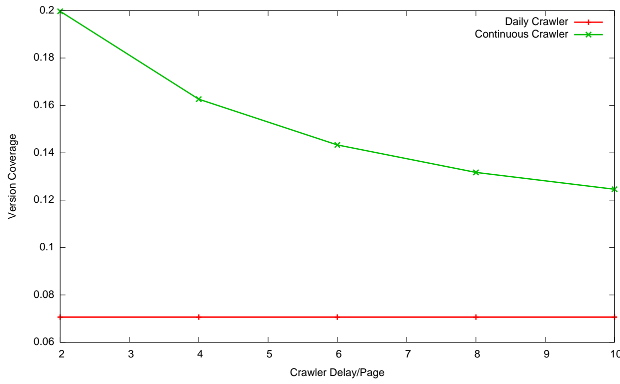

Figure 4: Version Coverage of Traditional Archival Crawlers

We are aiming at getting insight into the dynamics of different types of web sites to draw some conclusions on the potential coverage of our web archive. To this end, the most important statistics are (i) how frequently do pages within the site change, and (ii) how frequently do users access these pages. This allows us to compute estimates of the fraction of versions that would be covered by our system, given that a certain fraction of users contribute to the system.

Unfortunately, no web site provides both an archive of its different versions and a reasonably sized access log. Therefore, we resort to reverse engineering the version trail of a web site from an access log, namely the publicly available log of the 1998 FIFA Soccer WorldCup [5] with more than 1.3 billion accesses in 88 days. Abstractly, such an access log is a sequence of tuples (d, u, p, s, c), where, d is the date and time of access, u is a unique identifier of the accessing user (e.g., the IP address), p is the URL of the page, s is the size of the returned information, and c is the return code. Tuples in this sequence are ordered by date. We consider only successful accesses, i.e., accesses with a return code of 200, and failed accesses that indicate that a page was permanently dropped (with a return code of 404). Accesses with a return code of 304 ("unchanged") are mapped to the most recent previous successful access, accesses with other return codes are ignored.

We make the simplifying assumption that a new version of a page p was uploaded to the site whenever the size in a successful access to p is different from the previous successful access to p in the log. This allows us to extract a sequence of page versions from the access log. A failed access to a page that has been successfully read before, ends the lifetime of the current version of that page, without starting a new version. We also account for partial accesses, i.e., accesses where the client loaded only a prefix of the page, which are marked as successful in the log.

After consolidation, we identified 1,343,563,889 accesses to 560,569 distinct versions of 19,210 documents, made by 2,764,625 distinct users.

Archival Crawler. An archival crawler is designed to access a web site completely while respecting the exclusion directives and crawler-politeness requirements set by the site administrator. Our experience with the Heritrix crawler shows that this induces an average delay of 2-10 seconds between two successive page requests. These crawlers can be set up to harvest the archive of a web site, so as to run (i) at regular intervals, or (ii) continuously – as soon as the current crawl is completed, start another crawl. The duration of a crawl clearly depends on the number of pages accessed, but we make a simplifying assumption that it depends on the number of pages active on the site *at the beginning of the crawl*. Further, we also assume that within a single crawl the same URL is not revisited. Results of our experiments over the WorldCup web site of a daily and a continuous crawl, using per-page delays in the range of 2-10 seconds, are shown in Figure 4. As these results show, even in the "best" possible configuration of running a crawler continuously, we can capture only about 12%-20% of the versions!

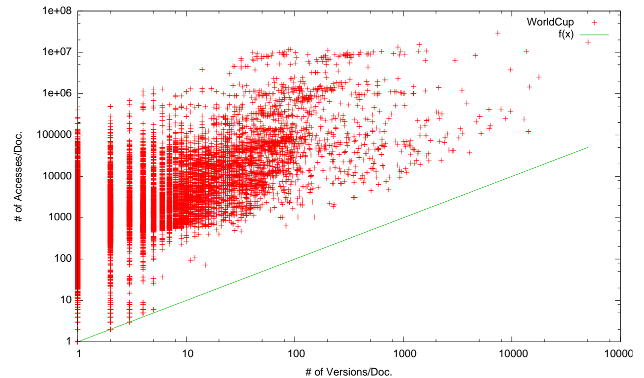

Figure 5: Correlation between Page Dynamics and Visits

Human-assisted Crawler. In contrast to the above scenario, a human-assisted crawler simply piggy-backs the archive harvesting on the regular browsing behavior at crawler peers. Effectiveness of such an approach depends on answers to the following two questions:

- Are highly dynamic pages, which have more versions, also visited by more surfers?
- What fraction of surfers of a site need to be subscribed as crawler peers in EverLast to ensure better version coverage than an archival crawler?

The scatter-plot shown in Figure 5 addresses the first question. The x-axis of the plot represents the dynamics of URLs in the WorldCup web site, as number of versions per document observed from the logs. The y-axis shows the absolute number of accesses made to the same URL during the same period. It is evident from the figure that higher dynamics of a URL typically entails a higher access rate. We use the *Pearson's rank coefficient* to show that higher the number of versions per document, higher are the relative number of accesses to it. It is preferred over the *Pearson's product-moment correlation* coefficient, which represents the linear dependence between two variables, because we compare the relative orders between number of versions and number of accesses. The *Pearson's rank coefficient* has a value of 0.70 hence representing a high degree of correlation between the number of versions per page and accesses per page. Although there is a large-spread in the number of accesses (y-axis), it is important to observe that the number of accesses is significantly above the baseline – represented by the line with gradient 1 in the graph.

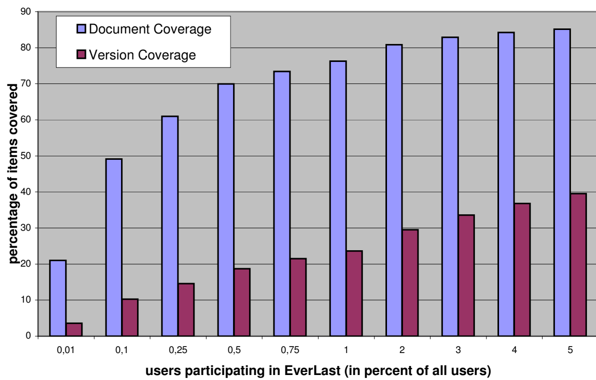

Figure 6: Version Coverage from Human-assisted Archiving in EverLast

To answer the second question, we randomly selected a fraction of surfers and tagged them as human-assisted crawlers subscribed with EverLast. All the documents and versions visited by this subset of surfers are considered as the harvested archive. We measured the fraction of documents and versions that were thus harvested, for different fractions of users participating in EverLast. Results are shown in Figure 6. As shown in the graph, we need less than 1% of surfers to participate with EverLast in order to capture at least as many versions as the best archival crawler assumed in our setting – i.e., continuous crawling with mean delay of 2 seconds per-page (leftmost point in x-axis of Figure 4). If 5% of surfers participate, we are able to capture more than 40% of the versions.

6.2 Partitioning Strategies

We used the Wikipedia version history to show the effect of index partitioning. The time-travel index list consists of version entries along with their validity time intervals. The index list which we consider has all the document versions, which are nearly 14 million, ranging over a period of five years. Index partitioning is done along the time axis using our greedy algorithm, boosted by a randomized simulated annealing algorithm. The results are presented in Figure 7. As expected, we observe that with increasing index-size (x-axis), the amount of replication in the index increases steadily, with up to 50% of index already replicated due to partitioning. These results are encouraging as they suggest a churn-resilient query processing strategy that scans the neighboring partitions to provide most of the relevant results.

Finally, notice that for a blowup of 2 in the index we get values of around 45% replication, contrary to the expected replication of around 100%. This happens because our replication mechanism is trying to also support efficient query processing by materializing the large index lists into sublists with smaller time-intervals. Additionally, there exist documents that cannot be replicated because their lifetimes are shorter from the time granularity assumed.

6.3 Distributed TTIX

We have experimented extensively with the time-travel inverted indexing under Te-Ti partitioning in a centralized setting. Note that we do not consider a top-k model of

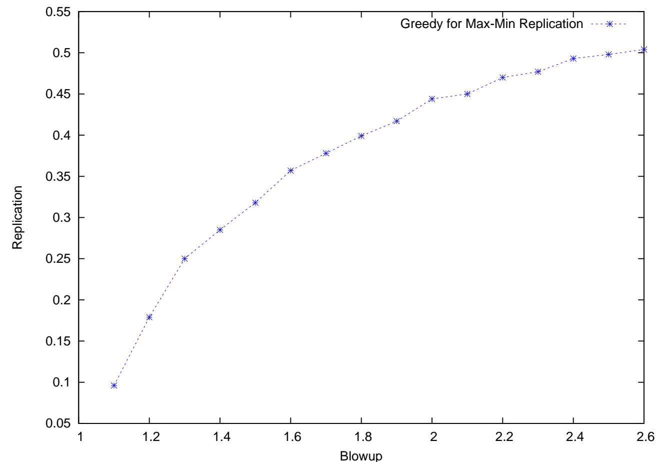

Figure 7: Greedy algorithm for Max-Min replication problem

processing time-travel queries. Therefore, during processing a query we scan all the contents of the relevant partition of the inverted list. This model is equivalent to the processing model in EverLast, where partitions are accessed by looking them up via the `get(pk)` interface of the persistence layer. Thus, results from our earlier work are carried over in the current setting as well. The interested reader is referred to [10, 11, 13] for more details.

In this experiment, we focus on a single term-specific inverted list that is partitioned along its time-axis. One drawback of such a partitioning is that entries spanning across partition boundaries are replicated, resulting in an overall index-size blowup. As a result, even fine granularity partitioning helps in reducing per-query index access cost, but incurs high storage expenses. Extreme solutions, such as not partitioning or partitioning at a very fine granularity, are not practical since query processing becomes very expensive. The partitioning strategies we developed in [10] trade-off processing performance with storage overheads, and provide interesting results; with as low as 10% additional data sent from persistence peers to indexing peer, the index-size blowup can be reduced by an order of magnitude.

7. CONCLUSIONS

In this paper, we outlined the overall architecture of EverLast aimed at addressing the challenges posed by the complete life cycle of capturing, storing and querying of web archives at large-scale. EverLast is designed over a loosely coupled distributed framework, applicable for deployment over P2P networks as well as data grids and small-scale server networks. Current estimates show that the proposed techniques of augmenting the standard archive crawler with human-assisted crawlers improves the quality of archives in terms of number of versions captured. The time-travel indexing scheme has been shown to be efficient and versatile for advanced queries over web archives.

8. REFERENCES

[1] Internet archive. http://archive.org.
[2] Swedish royal library: Kulturarw^3 – long-term preservation of electronic documents. http://www.kb.se/kw3/ENG/.

[3] E. Adar, M. Dontcheva, J. Fogarty, and D. Weld. Zoetrope: Interacting with the Ephemeral Web. In *Proc. of ACM UIST*, 2008.

[4] Avishek Anand. Indexing partitioning techniques for peer-to-peer web archival. Master's thesis, Universität des Saarlandes, FR Informatik, 2009.

[5] M. Arlitt and T. Jin. 1998 World Cup Site Access Logs. http://www.acm.org/sigcomm/ITA/, 1998.

[6] William Y. Arms, Selcuk Aya, Pavel Dmitriev, Blazej J. Kot, Ruth Mitchell, and Lucia Walle. Building a research library for the history of the web. In *JCDL*, 2006.

[7] R. A. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *Proc. of ICDE*, 2007.

[8] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4), 1996.

[9] M. Bender, S. Michel, J. X. Parreira, and T. Crecelius. P2p web search: Make it light, make it fly (demo). In *Proc. of CIDR*, 2007.

[10] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. A Time Machine for Text Search. In *Proc. of ACM SIGIR*, 2007.

[11] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. FluxCapacitor: Efficient Time-Travel Text Search. In *Proc. of VLDB*, 2007.

[12] K. Berberich, S. Bedathur, and G. Weikum. Efficient Time-travel on Versioned Text Collections. In *Proc. of GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW)*, 2007.

[13] K. Berberich, S. Bedathur, and G. Weikum. Tunable Word-Level Index Compression for Versioned Corpora. In *Proc. of Workshop EIIR*, 2008.

[14] 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.

[15] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proc. of ACM/USENIX NSDI*, 2006.

[16] James Clifford, Curtis E. Dyreson, Tomás Isakowitz, Christian S. Jensen, and Richard T. Snodgrass. On the semantics of "now" in databases. *ACM Trans. Database Syst.*, 22(2):171–214, 1997.

[17] P. Cudré-Mauroux, S. Agarwal, and K. Aberer. Gridvine: An infrastructure for peer information management. *IEEE Internet Computing*, 11(5), 2007.

[18] Heritrix Archival Crawler. http://crawler.archive.org/.

[19] E. Herder. Characterizations of User Web Revisit Behavior. In *Proc. of Workshop on Adaptivity and User Modeling in Interactive Systems*, 2005.

[20] P. Kalnis, W. S. Ng, B. C. Ooi, and K.-L. Tan. Answering similarity queries in peer-to-peer networks. *Inf. Syst.*, 31(1), 2006.

[21] R. Kotla, M. Dahlin, and L. Alvisi. Safestore: A durable and practical storage system. In *USENIX Annual Technical Conference*, June 2007.

[22] D. Lomet, M. Hong, R. Nehme, and R. Zhang. Transaction Time Indexing with Version Compression. In *Proc. of VLDB*, 2008.

[23] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proc. of ACM SIGMOD*, 1989.

[24] 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.

[25] 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.

[26] I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable peer-to-peer web retrieval with highly discriminative keys. In *Proc. of ICDE*, 2007.

[27] 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.

[28] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP*, 2001.

[29] B. Salzberg and V. Tsotras. Comparison of Access methods for Time-evolving Data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[30] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. of ICDE*, 2001.

[31] 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.

[32] I. Stoica, R.t 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.

[33] Stephan Strodl, Florian Motlik, Kevin Stadler, and Andreas Rauber. Personal & soho archiving. In *JCDL*, 2008.

[34] C. Tryfonopoulos, C. Zimmer, G. Weikum, and M. Koubarakis. Architectural alternatives for information filtering in structured overlays. *IEEE Internet Computing*, 11(4), 2007.

[35] 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.

[36] The Size of the World Wide Web. http://www.worldwidewebsize.com/, March 2008.

[37] C. Zimmer, S. Bedathur, and G. Weikum. Flood Little, Cache More: Effective Result-reuse in P2P IR Systems. In *Proc. of DASFAA*, 2008.