FluxCapacitor* : Efficient Time-Travel Text Search

Klaus Berberich, Srikanta Bedathur, Thomas Neumann, Gerhard Weikum
Max-Planck Institute for Informatics
Saarbrücken, Germany
{kberberi, bedathur, neumann, weikum}@mpi-inf.mpg.de

ABSTRACT

An increasing number of temporally versioned text collections is available today with Web archives being a prime example. Search on such collections, however, is often not satisfactory and ignores their temporal dimension completely. *Time-travel text search* solves this problem by evaluating a keyword query on the state of the text collection as of a user-specified time point. This work demonstrates our approach to efficient time-travel text search and its implementation in the FluxCapacitor prototype.

1. INTRODUCTION

Driven by the need to preserve digital content for future generations, and fueled by rapidly falling storage costs, an increasing number of large-scale versioned text collections are available today. Web archives (e.g., the Internet Archive [1]) and wikis that have built-in version-control mechanisms (e.g., Wikipedia [3]) are prime examples of such prevalent versioned collections.

Access to these collections and especially search on them is still rather limited. The search functionality offered by Wikipedia, for instance, considers only the most recent versions of articles in the encyclopedia. On Web archives, either (i) a search functionality is often completely missing, as in the case of the Internet Archive whose Wayback machine [2] only offers per URL lookups of archived versions, or (ii) different versions are treated as independent documents, thus potentially diluting query results with nearly-identical document versions. Neither approach is appropriate to serve *historical information needs* like the following,

In preparation for a documentary, a journalist needs to research changing political and societal opinions about the war in Iraq that began in 2003.

This information need cannot be effectively satisfied using standard Web search-engines, since many of the relevant Web pages have disappeared in the meanwhile and results are therefore dominated by more recent content. The Internet Archive, on the other hand, is likely to have preserved the relevant contemporary Web pages, but our journalist is unlikely to know their precise URLs. The key for a satisfactory solution is the temporal dimension of the text collection.

In order to serve such historical information needs effectively, a *time-travel text search* functionality is needed that allows the user to enrich a keyword query q by a time point t, or the *temporal context* of the query. This *time-travel query* is then evaluated over the state of the temporally versioned text collection as of time t.

Given that the targeted data sets are at the order of Terabytes (as the English Wikipedia) or even Petabytes (as the Internet Archive), naïve approaches to time-travel text search fail to scale and do not provide quick response times as today's users are accustomed to.

Our approach builds on the highly scalable inverted file index that has become the de facto standard for large-scale text-indexing. The structure of the index is extended to deal with temporally versioned text. The index is kept compact by means of the novel *approximate temporal coalescing* technique. This tunable technique capitalizes on the high redundancy in successive versions of the same document. Experiments have shown that it can reduce index size by more than 60% without noticeably affecting top-10 query results. Two principled *sublist materialization* techniques to trade-off index size and query performance complete our approach. The key idea behind these techniques is that query processing can be speeded up significantly, if shorter index lists that contain information valid for a contiguous time interval are maintained. Experiments have shown that these two techniques can achieve close-to-optimal query performance inducing only a small blowup of the index.

The demonstrated FluxCapacitor prototype provides an efficient implementation of the aforementioned techniques for *time-travel text search*. The system's scalability and the viability of our approach has been proven on two large-scale real-world data sets: (i) a subset of the Internet Archive comprising weekly crawls of 11 .gov.uk sites throughout the years 2004 and 2005, and (ii) the complete revision history of the English Wikipedia (which will be the dataset used for the demonstration). In addition, FluxCapacitor offers an intuitive user interface and provides quick response times to *time-travel queries*.

The remainder of this paper is organized as follows. In Section 2 we give an overview of the novel techniques that FluxCapacitor is based on. The architecture and implementation of the system are described in Section 3. The final Section 4 outlines our demonstration.

*In the "Back to the Future" movie trilogy the flux capacitor is the device that enables time travel.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

2. TIME-TRAVEL TEXT SEARCH

The inverted file is the de facto standard method for large-scale text-indexing, and lies at the heart of many systems including modern-day search engines. In FluxCapacitor, we make significant extensions to this powerful and well-studied text indexing structure to make it amenable for time-travel text search.

As a document relevance model in this work we adopt OKAPI BM25 [9] as the state-of-the-art technique having its roots in probabilistic information retrieval. For a given keyword query a document's relevance score is determined as a sum of $idf \cdot tf$ products, where idf is a keyword-specific measure reflecting the keyword's relative occurrence frequency in the text collection and tf is a keyword-document-specific measure indicating the keyword's occurrence frequency within the document. Further details are omitted for space reasons.

An inverted file index consists of a *vocabulary*, commonly organized as a B^+-Tree, that maps each term to its idf-score and an *inverted list*. The inverted list L_v of a term v contains a list of *postings*, each of which has the form $(\, d,\, p\,)$, where d is a document identifier and p is the so-called payload of the entry. The payload p typically contains information about the tf-score of v in d, but in some cases can contain auxiliary information such as the positional information of the term occurrence in d. For a recent survey about inverted file indexes we refer to [11]. Further, we refer to [7, 10] as two recent approaches that reduce index size by exploiting redundancy in the collection.

In standard text search systems, the inverted index is built over the latest snapshot of the corpus at regular intervals, and the previous index is no longer maintained. Such a standard inverted index built on the latest snapshot of English Wikipedia required $184,825,636$ postings – approximately 2.8 Gigabytes of storage (assuming 16 bytes per posting).

In contrast, supporting time-travel text search functionality demands that all historical indexing information such as, postings in each inverted list, past payload details, and past idf-scores of all terms, be completely retained. However, simply accumulating all the past indexes over time will incur significant storage costs and seriously hampers the query processing. Our innovations address this very issue, and comprise three stages that we detail in the remainder of this section.

2.1 Temporal Extensions

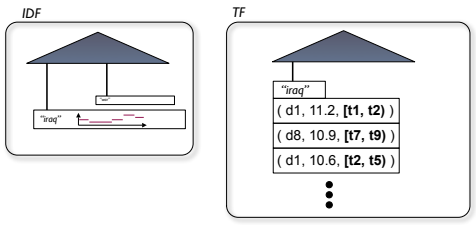


Figure 1: Temporal Extensions

In the first stage, we extend the inverted file for time-travel text search as illustrated in Figure 1. As can be seen from the figure, idf-scores and tf-scores are decoupled in our system. We note that idf-scores are time-varying in our setting, so that we conceptually manage a collection of time series. The data volume that is managed here and the overall impact on query processing performance are low in comparison to tf-scores. Therefore, we do not provide further details about the management of idf-scores here and concentrate on the more challenging tf-scores.

In detail, for inverted list postings we include a validity time-interval $[t_b, t_e)$ to denote the duration during which the the associated payload information was valid. This results in the following structure of postings: $(\, d,\, p,\, [t_b, t_e)\,)$ (as can be seen in Figure 1). Note that now there is one posting for every version of each document in the collection.

As a consequence of this temporal extensions, the index-size blows up significantly. For instance, on the English Wikipedia dataset, the thus temporally extended index for the full revision history for 5 years contains $8,634,711,830$ postings – requiring more than 257 Gigabytes (assuming 32 bytes per extended posting)! The next step of approximate temporal coalescing is designed to dramatically reduce this blow up.

2.2 Approximate Temporal Coalescing

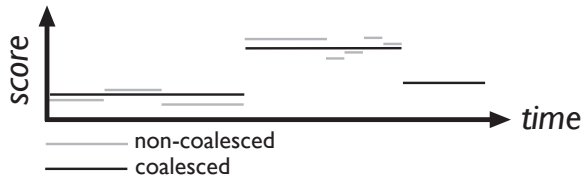


Figure 2: Approximate Temporal Coalescing

The key observation that helps to effectively counter the index-size blow-up is that *most changes in a versioned document collection are minor, leaving large parts of the document untouched*. As a consequence, the payload of many postings belonging to neighboring versions of a document differ only slightly or not at all. Approximate temporal coalescing reduces the number of postings in an index list by merging sequences of temporally adjacent postings (of a document) that have almost equal payloads, while keeping the maximal error made bounded by a tunable parameter ϵ. This idea is illustrated in Figure 2, which plots non-coalesced and coalesced scores of postings belonging to a single document. Approximate temporal coalescing is greatly effective given such fluctuating payloads and reduces the number of postings from 9 to 3 in the example.

The problem of finding a compact way to perform approximate temporal coalescing for a given threshold ϵ on the maximal error made, is equivalent to finding the compact piecewise constant representation of a time-series [8]. as well as to generating a compact relative-error bounded histogram [6]. We adapt the techniques proposed for these two problems to develop a linear-time algorithm that generates nearly-optimal temporal coalescing for the given error threshold.

Clearly, with increasing value of ϵ we can achieve greater index size reduction. Our empirical observations on a range of datasets has revealed the following: Even with very small (non-zero) values of ϵ, we noticed a sharp drop in the index size. We scrutinized the extent to which the final result ranking is affected as a consequence of approximate temporal coalescing. Our experiments on a variety of datasets including Wikipedia revealed that coalescing hardly disturbs the obtained top-10 and top-100 ranked results. We computed

the result overlap and Kendall's τ between the top-100 results obtained on the original and our temporally coalesced indexes for various values of ϵ. On the Wikipedia dataset, for $\epsilon = 0.05$, as an example, we observed a result overlap of more than 90% and a Kendall's τ of 0.85 indicating strong agreement in the order of results. Note that the corresponding index size was only 6.43% of the original index size.

2.3 Sublist Materialization

During query processing, postings are read sequentially from each of the inverted index lists and the postings that do not satisfy the given temporal context are discarded. As a consequence, query processing efficiency on our time-travel inverted index is adversely affected by the wasted I/O due to postings that are read and then discarded. Although temporal coalescing implicitly addresses this problem by reducing the overall index list size, still a significant overhead remains. The sublist materialization stage is specifically aimed at addressing this performance concern.

The key idea is to identify contiguous subintervals from the total time-span of the index list such that by separately materializing valid postings from corresponding subintervals any time-travel query can be answered efficiently. Note that coalesced postings that span different subintervals chosen for materialization are replicated in each of these materialization.

At a first glance, it may seem counterintuitive to reduce index size in the first step (using temporal coalescing), and then, to increase it again using the sublist materialization techniques presented in this section. However, we reiterate that our main objective is to improve the efficiency of processing queries, not to reduce the index size alone. The use of temporal coalescing improves the performance by reducing the index size, while the sublist materialization improves performance by judiciously replicating entries.

Note that, for a given temporal context, t, the optimal (in performance) sublist would consist of only those entries which are valid during t, nothing else. Clearly, it is impractical to materialize such optimal sublists for all time-points spanned by the collection.

We formulate two variants of the sublist materialization problem: in the first variant, a performance guarantee is retained such that the achieved performance is worse than optimal at most by a user-defined threshold. In the second variant, the index size blow-up due to sublist materializations is upper bounded by a user-specified threshold factor and a configuration is determined that satisfies this constraint while providing optimal expected performance.

We experimented with different values of the user-defined thresholds on a variety of datasets. For example, when guaranteeing that performance be at most twice the optimal performance for any time-point, we obtained a configuration that required close to 1% the space of the configuration giving optimal performance! Similarly, for the second variant, allowing a blow-up factor of 2 we achieve expected performance that is within 30% of the optimal performance.

For a more detailed description of the techniques outlined in this section we refer to [4].

3. ARCHITECTURE & IMPLEMENTATION

In this section we describe FluxCapacitor's system architecture and its main components illustrated in Figure 3.

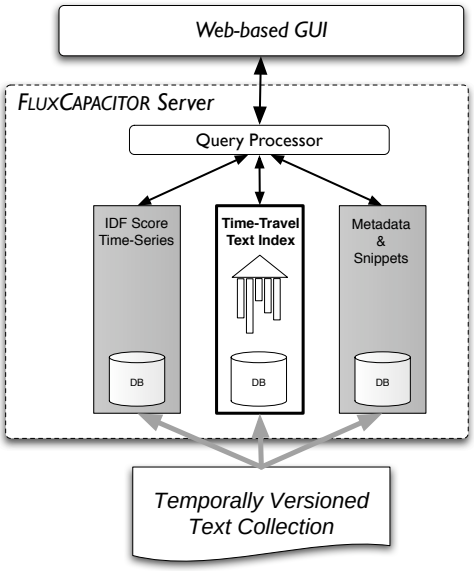


Figure 3: FluxCapacitor's System Architecture

The **Web-based GUI** is implemented using Google's AJAX Toolkit. Figure 4 shows a screenshot of the user interface that has the following key components: (1) *search box* for entering keyword queries, (2) dynamically updated *result size estimate over time* for the entered keyword query giving the user a hint on interesting temporal contexts, (3) *timeline* by clicking on which the temporal context is determined and the time-travel query is evaluated, and (4) *result presentation* including title, snippet, relevance score, as well as creation timestamp of the result.

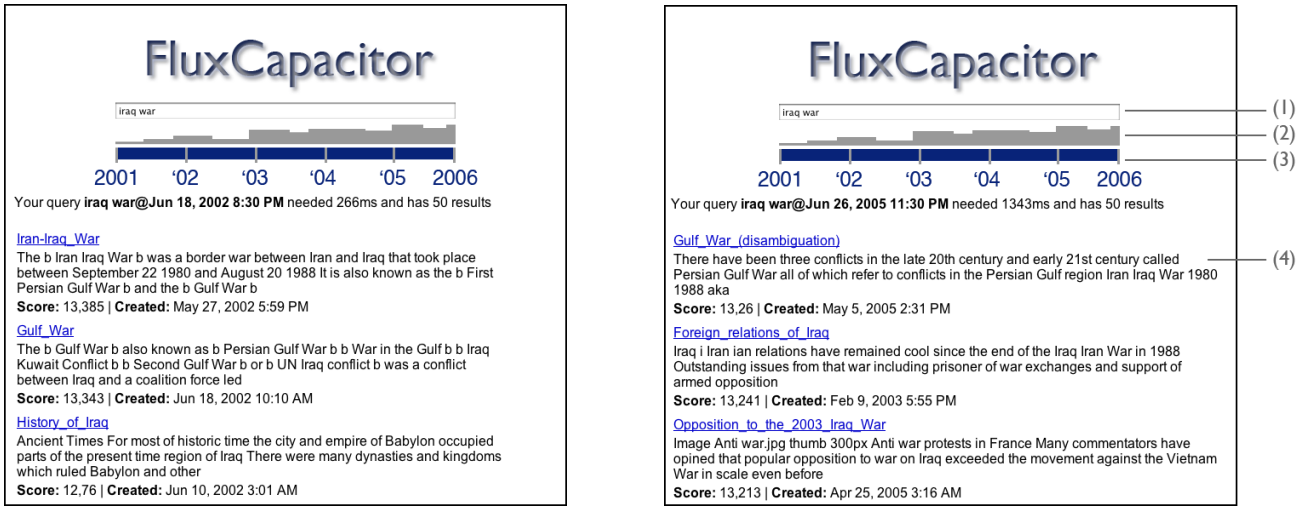
When a time-travel query (e.g., *"Iraq war"* as of June 26th, 2005) is submitted in the GUI, it is sent to the **FluxCapacitor Server**. There, the **Query Processor** component takes the following steps:

i) Retrieval of *idf* scores for the query keywords.
ii) Selection of appropriate index lists from the time-travel text index (the key technique of this work as described in Section 2), followed by the query evaluation on the selected lists.
iii) Enrichment of query results with metadata (e.g., the corresponding version timestamps) and snippets.

The server application runs inside an Apache Tomcat 5.5 servlet engine and is implemented in Java 1.5. All data (i.e., *idf*-score time-series, the time-travel text index, metadata, and snippets) is currently kept in an Oracle 10g database. The techniques for loading a **temporally versioned text collection** into our system (as depicted in the bottom of Figure 3), including approximate temporal coalescing and the sublist materialization techniques delineated in Section 2, are implemented in Java 1.5 as well.

4. DEMONSTRATION

Our demonstration will showcase the power and utility of FluxCapacitor using the *full revision history of the English Wikipedia* during 2001-2005. This dataset is publicly available and comprises a total of 892,255 documents in 13,976,915 versions. This large dataset will assist in highlighting the following key points.



(a) June 18th, 2002 (b) June 26th, 2005

Figure 4: FLUXCAPACITOR's Web-based GUI showing results for "Iraq war" and two temporal contexts

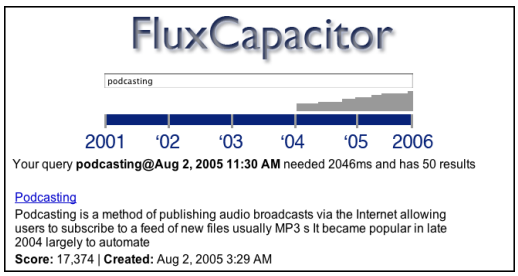


Figure 5: Result size distribution for "podcasting"

Temporal Exploration

Using a collection of example keyword queries such as "Iraq war", "Enron", "intelligent design", "david beckham", "podcasting" etc., under various temporal contexts, we illustrate the use of time-travel querying as a means of temporal exploration of the underlying corpus.

For example, using the query "Iraq war", one can observe how the top-results change to reflect the current opinions of Wikipedians. Specifically, for temporal contexts before the invasion of Iraq, shown in Figure 4(a), top results relate to the country Iraq and its war-torn history. Results later on, in late 2002 and early 2003, reflect the disarmament crisis, anti-war movements and the following invasion of the country. Finally, for temporal contexts in 2004 and 2005, as shown in Figure 4(b), query results relate to the analysis and criticism of the war, for example, the article titled Opposition_to_the_2003_Iraq_War can be seen to be one of the top-ranked results.

User Interface

We demonstrate the utility of our simple, yet powerful graphical user interface in formulating time-travel queries with almost the same ease as in prevalent Web search-engines. Knowledge of the distribution of expected results across the timeline of the corpus can provide an intuitive hint as to the choice of the temporal context. In our user interface, we include a histogram of estimated result size distribution, which is dynamically updated as the user enters new keywords into the search box. Note that we align this histogram with the timeline used for choosing the temporal context, thus making the choice intuitive and effortless. Consider the screenshot for the query "podcasting" shown in Figure 5. Clearly, there was no "podcasting" until as recent as 2004, which is reflected in the estimated result size distribution displayed. Thus, it gives an immediate visual feedback to the user that choosing a temporal context, $t < 2004$, will be of no use.

Efficiency

As our main innovations in FLUXCAPACITOR are geared towards achieving increased query processing efficiency, it forms a key aspect of our demonstration. We pose a variety of "hard" queries involving keywords with very low selectivity, and those which exhibit significant bursts of activity, so as to showcase the performance of FLUXCAPACITOR. For most of these queries, we have been able to achieve response time of below 1 sec.

Finally, we invite the participation of all visitors to a hands-on session trying out their favorite time-travel queries on Wikipedia with FLUXCAPACITOR.

5. REFERENCES

[1] http://www.archive.org.
[2] http://www.archive.org/web/web.php.
[3] http://www.wikipedia.org.
[4] K. Berberich, S. Bedathur, T. Neumann, G. Weikum. A Time Machine for Text Search. *SIGIR*, 2007.
[5] K. Berberich, S. Bedathur, G. Weikum. A Pocket Guide to Web History. *SPIRE*, 2007.
[6] S. Guha, K. Shim, J. Woo. REHIST: Relative Error Histogram Construction Algorithms. *VLDB*, 2004.
[7] M. Hersovici, R. Lempel, S. Yogev. Efficient Indexing of Versioned Document Sequences. *ECIR*, 2007.
[8] E. J. Keogh, S. Chu, D. Hart, M. J. Pazzani. An Online Algorithm for Segmenting Time Series. *ICDM*, 2001.
[9] S. E. Robertson and S. Walker. OKAPI/Keenbow at TREC-8. *TREC*, 1999.
[10] J. Zhang and T. Suel. Efficient Search in Large Textual Collections with Redundancy. *WWW*, 2007.
[11] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Comput. Surv.*, 38(2):6, 2006.