Space-Efficient Support for Temporal Text Indexing in a Document Archive Context

Kjetil Nørvåg

Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
Kjetil.Norvag@idi.ntnu.no

Abstract. Support for temporal text-containment queries (query for all versions of documents that contained one or more particular words at a particular time t) is of interest in a number of contexts, including web archives, in a smaller scale temporal XML/web warehouses, and temporal document database systems in general. In the V2 temporal document database system we employed a combination of full-text indexes and variants of time indexes to perform efficient text-containment queries. That approach was optimized for moderately large temporal document databases. However, for "extremely large databases" the index space usage of the approach could be too large. In this paper, we present a more space-efficient solution to the problem: the interval-based temporal text index (ITTX). We also present appropriate algorithms for update and retrieval, and we discuss advantages and disadvantages of the V2 and ITTX approaches.

1 Introduction

The amount of information made available on the web is increasing very fast, and an increasing amount of this information is made available *only* on the web. While this makes the information readily available to the community, it also results in a low persistence of the information, compared to when it is stored in traditional paper-based media. This is clearly a serious problem, and during the last years many projects have been initiated with the purpose of archiving this information for the future. This essentially means crawling the web and storing snapshots of the pages, or making it possible for users to "deposit" their pages. In contrasts to most search engines that only store the most recent version of the retrieved pages, in these archiving projects all (or at least many) versions are kept, so that it should also be possible to retrieve the contents of certain pages as they were at a certain time in the past. The Internet Archive Wayback Machine [4] is arguably the most famous project in this context, and aims at providing snapshots of pages from the whole Web. Similar projects at the national level also exist in many countries, often initiated through the national libraries.

Most current web archives still only support retrieval of a web site as they were at a particular time t (i.e., returning a snapshot), and several (including the Internet Archive Wayback Machine [4]) also do not support text-containment queries yet (a text-containment query is a query for all versions containing a particular set of words, similar to the service offered by a search engine). The restrictions are understandable based on the large amount of text stored in these archives. However, in order to increase the usefulness of these archives as research tools, *temporal text-containment queries* (query for all versions of pages that contained one or more particular words at a particular time t) should also be supported. Support for temporal text-containment queries is also useful in a more general context, in document databases as well as in *temporal XML/web warehouses*, which are archives similar to the one discussed above, but in a smaller scale. We denote a generic system supporting these features a *temporal document database system*.[1] In our project we have studied this issue in the V2 temporal document database system [9]. In the V2 prototype we employed a combination of full-text indexes and variants of time indexes for performing efficient text-containment queries [8]. That approach was optimized for moderately large temporal document databases. However, for "extremely large databases" as in the contexts discussed above, the index space usage of the approach used in V2 could be too large. In this paper we present a more space-efficient solution to the problem, which we call the interval-based temporal text index (ITTX), we present appropriate algorithms for update and retrieval, and we discuss advantages and disadvantages of the V2 and ITTX approaches to temporal text-containment querying.

The organization of the rest of this paper is as follows. In Section 2 we give an overview of related work. In Section 3 we give an overview of the V2 temporal document database system. In Section 4 we describe the ITTX approach. Finally, in Section 5, we conclude the paper.

2 Related work

We have previously described the use of variants of time indexes to support temporal text-containment queries [8] (one of these variants will be presented in Section 3.4).

Another approach to solve some of the problems discussed in this paper, is the proposal from Anick and Flynn [1] on how to support versioning in a full-text information retrieval system. In their proposal, the current version of documents are stored as complete versions, and backward deltas are used for historical versions. This gives efficient access to the current (and recent) versions, but costly access to older versions. They also use the timestamp as version identifier. This is not applicable for transaction-based document processing where all versions created by one transaction should have same timestamp. In order to support temporal text-containment queries, they based the full-text index on bitmaps for words in current versions, and delta change records to track incremental changes to the index backwards over time. This approach has the same advantage and problem as the delta-based version storage: efficient access to current version, but costly recreation of previous states is needed. It is also difficult to make temporal zig-zag joins (needed for multi-word temporal text-containment queries) efficient.

[1] Note that such a system is different from traditional systems where versions of documents are stored and retrieved using the document name and a version/revision number. In our system we can take advantage of the temporal aspect, and make it possible to retrieve documents based on predicates involving both *document contents* and *validity time*, and in this way satisfy temporal queries.

Xyleme [12] supported monitoring of changes between a new retrieved version of a page, and the previous version of the page, but did not actually support maintaining and querying temporal documents (only the last version of a document was actually stored).

Storage of versioned documents is studied by Marian et al. [6] and Chien et al. [3]. Algorithms for temporal XML query processing operators are proposed in [7], where queries can also be on structure as well as text content.

Another approach to temporal document databases is the work by Aramburu et al. [2]. Based on their data model TOODOR, they focus on static document with associated time information, but with no versioning of documents. Queries can also be applied to metadata, which is represented by temporal schemas.

3 An overview of the V2 temporal document database system

In order to make this paper self-containing, and provide the context for the rest of this paper, we will in this section give a short overview of V2. We omit many optimizations in this overview, and for a more detailed description, and a discussion of design choices, we refer to [9, 8].

3.1 Document version identifiers and time model

A document version stored in V2 is uniquely identified by a *version identifier* (VID). The VID of a version is persistent and never reused, similar to the object identifier in an object database.

The aspect of time in V2 is *transaction time*, i.e., a document is stored in the database at some point in time, and after it is stored, it is *current* until logically deleted or updated. We call the non-current versions *historical versions*. When a document is deleted, a tombstone version is written to denote the logical delete operation.

3.2 Functionality

V2 provides support for storing, retrieving, and querying temporal documents. For example, it is possible to retrieve a document stored at a particular time t, retrieve the document versions that were valid at a particular time t and that contained one or more particular words. A number of operators is supported, including operators for returning the VIDs of all document versions containing one or more particular words, support for temporal text-containment queries, as well as the Allen operators, i.e., *before, after, meets*, etc.

V2 supports automatic and transparent compression of documents if desired (this typically reduces the size of the document database to only 25% of the original size).

3.3 Design and storage structures

The current prototype is essentially a library, where accesses to a database are performed through a V2 object, using an API supporting the operations and operators described previously. The bottom layers are built upon the Berkeley DB database toolkit [11], which is used to provide persistent storage using B-trees.

The main modules of V2 are the version database, document name index, document version management, text index, API layer, operator layer, and optionally extra structures for improving temporal queries. We will now give an overview of the storage-related modules.

Version database. The document versions are stored in the version database. In order to support retrieval of parts of documents, the documents are stored as a number of chunks (this is done transparently to the user/application) in a tree structure, where the concatenated VID and chunk number is used as the search key. The VID is essentially a counter, and given the fact that each new version to be inserted is given a higher VID than the previous versions, the document version tree index is append-only.

Document name index. A document is identified by a *document name*, which can be a filename in the local case, or an URL in the more general case. Conceptually, the document name index has for each document name some metadata related to all versions of the document, followed by specific information for each particular version.

Text indexing. A text-index module based on variants of inverted lists is used in order to efficiently support text-containment queries, i.e., queries for document versions that contain a particular word (or set of words). In our context, we consider it necessary to support dynamic updates of the full-text index, so that all updates from a transaction are persistent as well as immediately available. This contrasts to many other systems that base the text indexing on bulk updates at regular intervals, in order to keep the average update cost lower. In cases where the additional cost incurred by the dynamic updates is not acceptable, it is possible to disable text indexing and re-enable it at a later time. When re-enabled, all documents stored or updated since text indexing was disabled will be text indexed. The total cost of the bulk updating of the text index will in general be cheaper than sum of the cost of the individual updates.

3.4 Temporal text-containment queries in the V2 prototype

Several approaches for text-containment queries are supported by V2 [8]. We will now describe the most general of these, the VP-index-based approach, where the validity period of every document version is stored in a separate index. This index maps from VID to validity period, i.e., each tuple in the VP index contains a VID, a start timestamp, and an end timestamp. In the V2 prototype the VP index is implemented as a number of chunks, where every chunk is an array where we only store the start VID which the period tuples are relative to. In this way we reduce the space while still making delete operations possible.

Even for a large number of versions the VP index is relatively small, but it has a certain maintenance cost: 1) at transaction commit time the VID and timestamp has to be stored in the index, and 2) every document update involves updating the end timestamp in the previous version, from UC (until changed) to a new value which is the start (commit) timestamp of the new version. The first part is an efficient append operation, but the second is a more costly insert. However, considering the initial text indexing cost for every version, the VP-index maintenance cost is not significant in practice.

Temporal text-containment queries using the VP-index-based approach can be performed by 1) a text-index query using the text index that indexes all versions in the database, followed by 2) a time-select operation that selects the actual versions (from stage 1) that were valid at the particular time or time period.

4 Space-efficient support for temporal text-containment queries

Many of the design decisions in V2 are based on the assumption that it shall support typical database features as transaction-based consistency, arbitrarily large transactions, and (relatively) fast commit. However, in a document warehouse context, these can be relaxed. For example, it is possible to assume only one write/update process.

Before we continue the discussion of the text index, we emphasize that as a minimum, the text index should:

– Support the mapping operation from a word W to the identifiers of all the document versions that contain the word.
– Ensure that *if* a identifier is returned by a lookup, it should be *guaranteed* that the document version with this identifier actually exists and contains the word W.

4.1 A new storage architecture

One problem with the original text index in V2 is that each occurrence of a word in a version require a separate posting in the text index. This makes the size of the text index proportional with the size of the document database. For a traditional document databases where only current versions are stored it is difficult to improve on this property (although compression of index entries normally is used to reduce the total index size). However, in a document database with several versions of each document, the size of the text index can be reduced by noting the fact that the difference between consecutive versions of a document is usually small: frequently, a word in one document version will in also occur in the next (as well as the previous) version. Thus, we can reduce the size of the text index by storing word/version-range mappings, instead of storing information about individual versions.

In V2 every document version is assigned a VID=v, where the new VID is one higher than the previous assigned VID, i.e., VID=v+1. There is no correlation between VID and document, i.e., VID=v and VID=v+1 would in general identify versions of different documents. The advantage of this strategy of assigning VIDs is that it gives the version database an efficient append-only property when the VIDs are used as the keys of the versions. In a range-based index a number of versions with VID=$\{$5,6,7,8$\}$ containing the word W could be encoded as $(W, 5, 8)$. However, there is not necessarily much similarity between document versions VID=v and VID=v+1 when the versions are versions of different documents. Except for the most frequently occurring words, there will not be many chances to cover large intervals using this approach. In order to

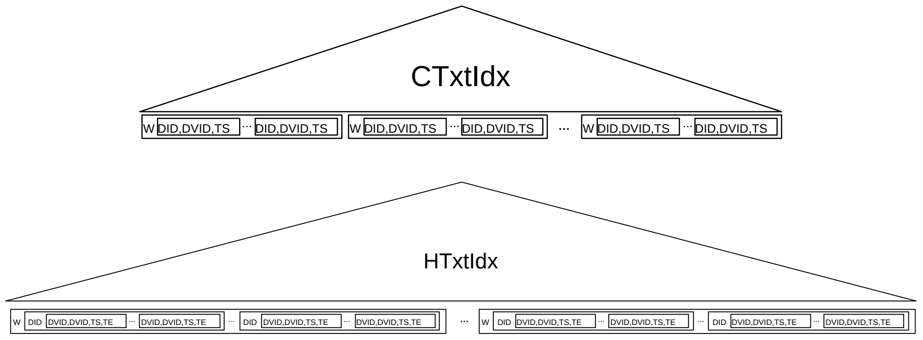

Fig. 1. The ITTX temporal text-index architecture.

benefit from the use of intervals, we instead use *document version identifiers* (DVIDs). Given a version of a document with DVID=v, then the next version of the same document has DVID=v+1. In contrast to the VIDs that uniquely identify a document version stored in the system, different versions of different documents can have the same DVID, i.e., the DVIDs are not unique between different versions of different documents. In order to uniquely identify (and to retrieve) a particular document version, a *document identifier* (DID) is needed together with the DVID, i.e., a particular document version in the system is identified by (DID$||$DVID). In this way, consecutive versions of the same document that contain the same word can form a range with no holes.

Conceptually, the text index that use ranges can be viewed as a collection of (W,DID,DVID$_i$,DVID$_j$)-tuples, i.e., a word, a document identifier, and a DVID range. Note that for each document, there can be several tuples for each word W, because words can appear in one version, disappear in a later version, and then again reappear later. A good example is a page containing news headlines, where some topics are reoccurring.

When a new document version with DVID=DVID$_i$ is inserted, and it contains a word that did not occur in the previous version, a (W,DID,DVID$_i$,DVID$_j$) tuple is inserted into the index. DVID$_i$ is the DVID of the inserted version, but DVID$_j$ is set to a special value UC (until changed). In this way, if this word is also included in the next version of this document, the tuple does not have to be modified. This is an important feature (a similar technique for avoiding text index updates is also described in [1]). Only when a new version of the document that does not contain the word is inserted, the tuple has to be updated. It is important to note that using this organization, it is impossible to determine the DVIDs of the most recent versions from the index. For the [DVID$_i$,UC] intervals only the start DVID is available, and we do not know the end DVID. As will be described later, this makes query processing more complicated.

Under the assumption that queries for current documents will be frequent enough to compensate for the additional cost of having separate indexes for currently valid entries, a separate index is used for the entries that are still valid (an assumption and solution also used in some traditional temporal database systems), i.e., where the end of the interval is UC. In this index the end value UC is implicit, so that only the start DVID needs to be stored. We denote the index for historical entries *HTxtIdx* and the index for valid entries *CTxtIdx*. Note that the space possibly saved by this two-index approach is not an issue, the point is the smaller amount of index entries that have to be processed during a query for current versions (and prefix queries can in particular benefit from this architecture).

So far, we have not discussed how to make temporal text-containment queries efficient. The first issue is whether timestamps can be used *instead* of DVIDs in the index. This would be very useful in order to have document version validity time in the index. However, this is difficult for several reasons. First of all, more than one document version could have the same timestamp, because they were created at the same time/by the same transaction. However, the most important problem is that the intervals would not be "dense": in general, given a document version with timestamp $T = t_i$, the next version of the document will not have timestamp $T = t_i$+1. The result is that it is impossible to determine the actual timestamps or document identifiers of the document versions from the index.

In V2, the VP index is used to make temporal text-containment queries more efficient (see Section 3.4) by providing a map from VID to version validity period (i.e., conceptually each tuple in the VP index contains a VID, a start timestamp, and an end timestamp). With sequentially allocated VIDs as in V2, the VP index is append-only. The result is a low update cost, and it is also space efficient because the VIDs do not have to be explicitly stored.

One of the main reasons why the VP index is very attractive in the context of V2, is that storing the time information in the VP index is much more space efficient than storing the timestamps replicated many places in the text index (once for each word). However, when intervals are used, one timestamp for each start- and end-point of the intervals is sufficient, and the increase in total space usage, compared with using a VP index, is less than what is the case in V2. It is also more scalable, because the V2 approach is most efficient when the VP index can always be resident in main memory. To summarize, our final solution for the ITTX is to store (W,DID,DVID$_i$,DVID$_j$,T_S,T_E) in the HTxtIdx (where T_S and T_E are the start- and end-timestamps of the interval [DVID$_i$,DVID$_j$$>$, and to store ($W$,DID,DVID,$T_S$) in the CTxtIdx.

4.2 Physical organization

The physical organization of the ITTX is illustrated in Figure 1. In the HTxtIdx there are for each word a number of document identifiers (DIDs), and for each DID there is a number of [DVID$_i$,DVID$_j$$>$/[$T_S$,$T_E$$>$ ranges. The CTxtIdx is similar, except that the end DVIDs and end timestamps implicitly have the value UC and are therefore omitted. In an actual implementation based on B-trees, this data is stored in chunks in the leaf. The key of a chunk is (W,DID,DVID), where DID is the smallest DID in the chunk, and the DVID is the start of the first interval for this DID. As will be explained below, this information is needed in order to efficiently implement multi-word queries where we only need to retrieve postings for a subset of the DIDs. Note that the size of a chunk must be smaller than a page, and as a result there will in general be more than one chunk for each word, and there can also be more than one chunk for each DID.

The *document name index* is basically the same as the one used in V2, except that for each document name there is also a document identifier. We emphasize that the document name index is a map from document name to DID *and* the DVID/timestamp of all the document versions.

The *document version database* is also identical to the one used in V2, except that (DID$||$DVID) is used as the key, instead of only a VID. The append-only property it had in V2 when using VID as the key is lost, but the cost of updating the version database is much smaller than the text indexing cost, so this cost is compensated by a more efficient text indexing. By including some time information in the ITTX we also avoid having a separate VP index.

4.3 Operations

Given the description of the ITTX, the algorithms for the most important operations are as follows.

Insert document: When inserting a new document (i.e., a document with a document name not already stored in the database) at time t:

1. A new DID is allocated, and the document is inserted into the version database.
2. For all distinct words W in the document, a (W,DID,DVID=0, T_S=t) tuple is inserted into the CTxtIdx.

Update document: If the document to be inserted at time t is a new version of a document that is already stored in the database:

1. The previous version of the document with DVID=j has to be read in order to determine the difference (or delta) between the two versions. The difference is needed for several purposes: 1) often we do not want to store identical versions, 2) we might want to reduce the storage costs by storing delta versions instead of complete versions, and 3) we need to determine which *new words* appear in the new version, and which words that existed in the previous version *do not* occur in the new version.
2. A new DVID=j+1 is allocated and the document is inserted into the version database. In addition, if we want to store delta versions, the previous document version is replaced with the delta version.
3. For all *new* distinct words W in the document, a (W,DID,DVID=j+1,T_S=t) tuple is inserted into the CTxtIdx.
4. For all words that disappeared between the versions, (W,DID,DVID=i,T_S) is removed from the CTxtIdx and (W,DID,DVID=i,DVID=j,T_S,$T_E = t$) is inserted into the HTxtIdx.

Delete document: A document delete in a temporal database is a *logical* operation, in contrast to non-temporal databases where it is a *physical* operation. The delete operation in a temporal database can be realized as an update that creates a (logical) *tombstone* *version*. This tombstone can be considered as an empty document, and using the procedure for documents as described above, all the words of the previous version is moved to the HTxtIdx, but none are inserted into the CTxtIdx.

Retrieve document version: A document version can be retrieved from the version database using (DID,DVID) as the key. If a document with a particular name *DocName* is to be retrieved, a lookup in the document name index is performed first. The (DID,DVID) for all versions are stored here, sorted on DVID/time so that if a document version valid at a particular time t is to be retrieved, the actual (DID,DVID) can be found by a simple lookup in the document name index. The current version of a particular document DID is simply the version with the largest DVID.

Non-temporal single-word text-containment query: All *documents* currently containing a particular word W_S can be found by a lookup in the CTxtIdx for all (W,DID,DVID$_i$,T_S) where W=W_S. However, it is impossible from the CTxtIdx alone to know the most recent DVID for the actual DIDs. This can be found from the version database, where the most recent DVID for a document is found by searching for the largest DVID for the actual DID. Note that this is a simple lookup operation, the actual document(s) do not have to be retrieved.

Non-temporal multi-word text-containment query: A multiword query, i.e., a query for all documents that contain a particular set of words $W_1, ..., W_n$, can be executed by:

1. A lookup for word W_1 in the CTxtIdx, resulting in a set of DIDs which initializes the set R_1.
2. For each of the words $W_2, ..., W_n$, lookup the word W_i in the CTxtIdx, let the set S_i be the resulting DIDs, and let $R_i = R_{i-1} \cap S_i$. When finished, R_n will contain the DIDs of the documents that contain the words. In practice an optimized version of this algorithm will be used. For example, if the number of entries for each word is known in advance, it is possible to start the query with retrieving the entries for the most infrequently occurring word(s), and only do lookups where a possible match can occur, e.g., given R_1, only do lookups for all (W_2,DID$_x$) where DID$_x$ is a member of R_1. In order to be able to do this, statistics for word occurrences is needed. In static document databases this information is often stored in the index. However, maintaining this statistics considerably increases the update cost, so in a dynamic document database we rather advocate maintaining a *word-frequency cache* for the most frequently occurring query words. This approximation should be enough to reduce the query cost. Even without the help of a word-frequency cache or statistics, the size of R_i, where $i \geq 2$, will in general be small enough to make it possible to reduce the cost by doing selective lookups.

Temporal single-word text-containment query (snapshot): When querying for all document versions that contained a particular word W_S at a particular time t, both the CTxtIdx and the HTxtIdx have to be searched:

- Search HTxtIdx: All (W,DID,DVID$_i$,DVID$_j$,T_S,T_E) where W=W_S and $T_S \leq t \leq T_E$ are retrieved.
- Search CTxtIdx: All (W,DID,DVID$_i$, T_S) where W=W_S and $t \geq T_S$ are retrieved.
- At most one version of each document can be valid at a particular time, so the interesting part of the result is essentially a set of (DID, DVID$_i$,DVID$_j$) tuples. In general we do not know the actual DVIDs of the matching versions (assuming a fine-granularity timestamp, a match on t=T_S or t=T_E has a low probability). Given a (DID, DVID$_i$,DVID$_j$) tuple and a time t, the actual DVID can in most cases efficiently be determined by first doing a lookup in the document version database in order to find the name of the document with the actual DID, followed by a lookup in the document name index to find the DVID of the version of the document that were valid at time t.

Temporal multi-word text-containment query (snapshot): The single-word temporal text-containment query can be extended to multi-word similar to the case of a non-temporal query. However, determining the actual DVIDs should be postponed until the final step.

Temporal text-containment query (time range): Querying for all document versions that contained a particular word at some point during a specified time *interval* is performed similar to the temporal snapshot query as described above, the only difference is that there can be more than one matching version of each document, resulting in more than one (DID, DVID$_i$,DVID$_j$) tuple for each DID.

4.4 Comparison

The most important advantages of the ITTX, compared to the original V2 indexing approach, are:

- Smaller index size.
- More efficient non-temporal (current) text-containment query (i.e., on the documents that are currently valid). In the V2 approach, it was impossible to tell from the text index whether a version with a certain VID was a current version or not. In addition, it was impossible to determine whether two VIDs in the text index were VIDs of the same document or not, resulting in a larger number of VIDs to inspect than desired.
- The time intervals in the index drastically reduce the number of DVIDs that have to be checked. For example, for a snapshot query only one lookup is needed for each matching document in order to determine the DVID.
- The average cost of updating a document is much lower. Assuming most words in version DVID=i also occur in version DVID=i+1, only a few postings need to be updated.

However, as with most indexing problems, there is not a single index structure that is best in all situations: the context decides which indexing method is most beneficial. This is also the case in our case, and in some contexts the V2 indexing approach can be better than the ITTX:

- Although the insert of new documents will have the same *indexing cost* (for new documents, all words have to be indexed), the actual insert into the version database will be slightly more expensive because it is not an append operation as before. Another result of insert instead of append is that the storage utilization in the version database is likely to be lower.
- In the V2 system, two important operations that can be used to reduce the amount of data that is stored in the system (or identify candidates for tertiary-storage migration) are *vacuuming* and *granularity reduction* [10]. Vacuuming is the traditional temporal database method for data reduction, and removes the oldest versions. However, in a document database context, granularity reduction is often more appropriate. Two simple granularity reduction approaches that illustrate the concept, are deleting versions of an document that are closer in time than t_t, or simply removing every second version of a document. This can be useful when we have a large number of document versions created during a relatively short time period, and after a while we do not really need all these versions. Although a (logical) delete in a temporal database does not result in removal of any historical versions, physical deletion can be performed as a result of vacuuming and granularity reduction operations [10]. When versions are physically removed from the database, fragmentation of time ranges can occur. For example, the use of a granularity reduction strategy that removes every second version will have as a result that all intervals only cover one DVID. Vacuuming on the other hand removes all versions older than a certain age, and does not affect the efficiency of the interval representation of DVIDs. Thus, the ITTX approach is most suitable for application areas where vacuuming, and not granularity reduction, is employed. In the context of this paper, we can assume that all versions will be kept (and that eventual granularity reduction has already been applied to the data), and there is less need for granularity reduction.
- Bulk-updating can be performed relatively efficiently using the original V2 approach. VIDs are assigned sequentially, so that new VIDs are appended to the posting lists. Even of a large number of VIDs are inserted for a particular word, they are stored clustered in the leaf nodes of the index, so that only a much smaller number of writes are actually needed. In the ITTX, the DVIDs are clustered on DIDs, so that in worst case one write is needed for each word in a document (however, we emphasize that when updating documents only a few entries actually have to be updated in the index).

5 Conclusions and further work

In our V2 temporal document database system we employed a combination of full-text indexes and variants of time indexes for performing efficient text-containment queries. That approach was optimized for moderately large temporal document databases. However, for "extremely large databases" the index space usage of the V2 approach could be too large. In this paper we have presented a more space-efficient solution to the problem, the interval-based temporal text index (ITTX). We have presented appropriate algorithms for update and retrieval, and we discussed advantages and disadvantages of these two approaches.

Future work includes integrating the ITTX into V2 and use our temporal document database benchmarks to compare the performance of ITTX with the VP index approach currently used in V2. We also plan to investigate approaches that can achieve better clustering in the temporal dimension, for example by using an extension of indexing structures like the TSB-tree [5].

Acknowledgments This work was done when the author visited Athens University of Economics and Business in 2002, and Aalborg University in 2003, supported by grant #145196/432 from the Norwegian Research Council.

References

1. P. G. Anick and R. A. Flynn. Versioning a full-text information retrieval system. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1992.
2. M. J. Aramburu-Cabo and R. B. Llavori. A temporal object-oriented model for digital libraries of documents. *Concurrency and Computation: Practice and Experience*, 13(11), 2001.
3. S.-Y. Chien, V. Tsotras, and C. Zaniolo. Efficient schemes for managing multiversion XML documents. *VLDB Journal*, 11(4), 2002.
4. Internet archive. http://archive.org/.
5. D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.
6. A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *Proceedings of VLDB 2001*, 2001.
7. K. Nørvåg. Algorithms for temporal query operators in XML databases. In *Proceedings of Workshop on XML-Based Data Management (XMLDM) (in conjunction with EDBT'2002)*, 2002.
8. K. Nørvåg. Supporting temporal text-containment queries. Technical Report IDI 11/2002, Norwegian University of Science and Technology, 2002. Available from http://www.idi.ntnu.no/grupper/DB-grp/.
9. K. Nørvåg. V2: A database approach to temporal document management. In *Proceedings of the 7th International Database Engineering and Applications Symposium (IDEAS)*, 2003.
10. K. Nørvåg. Algorithms for granularity reduction in temporal document databases. Technical Report IDI 1/2003, Norwegian University of Science and Technology, 2003. Available from http://www.idi.ntnu.no/grupper/DB-grp/.
11. M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, 1999.
12. L. Xyleme. A dynamic warehouse for XML data of the web. *IEEE Data Engineering Bulletin*, 24(2), 2001.