Supporting Temporal Text-Containment Queries in Temporal Document Databases

Kjetil Nørvåg [1]

Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway

Abstract

In temporal document databases and temporal XML databases, temporal text-containment queries are a potential performance bottleneck. In this paper we describe how to manage documents and index structures in such databases in a way that makes temporal text-containment querying feasible. We describe and discuss different index structures that can improve such queries. Three of the alternatives have been implemented in the V2 temporal document database system, and the performance of the index structures is studied using temporal web data. The results show that even a very simple time-indexing approach can reduce query cost by up to three orders of magnitude.

Key words: Temporal databases, document databases, text indexing, temporal indexes, web archiving

1 Introduction

In this paper we describe techniques for improving queries on text in dynamic temporal document databases, where "dynamic" in this context means databases with regular updates of data, and where access structures should always be consistent with the contents stored in the database. This contrasts to many traditional document database systems, where the text indexes are only updated at regular intervals, because of the associated cost of maintaining a consistent text index. In addition, querying temporal versions should be supported, so that we can query for documents that contained particular words at a particular time in the past.

[1] E-mail: Kjetil.Norvag@idi.ntnu.no

Motivation. In order to motivate the subsequent description, we first describe the example application that motivated our initial work on temporal text-containment queries: *a temporal XML/web warehouse* (Web-DW). In our work, we wanted to be able to maintain a temporal Web-DW, storing the history of a set of selected web pages or web sites. By periodically retrieving these pages and storing them in the warehouse, and at the same time keeping the old versions, we should be able to:

- Retrieve the individual pages or web sites as they were at a particular time t.
- Retrieve all versions of pages that contained one or more particular words at a particular time t.
- Ask for changes, for example retrieve all pages that did not contain "Bin Laden" before 11 September 2001, but contained these words afterwards.

In addition to temporal document databases/warehouses as described above, support for temporal text indexing can also be useful in a number of other areas. For example:

- Temporal relational or object databases with text attributes. For relational databases, the approach is most applicable where row identifiers exist for tuples. The approach is also applicable for *non-temporal databases*, with tuples containing both text and time/date attributes.
- Temporal pattern-tree queries. For example, in Xyleme [27], a text index is used for executing the *PatternScan* [1] operator. This operator takes as input a forest of trees (XML documents or elements), and a pattern tree. The result is based on matching the input trees against the pattern tree. In Xyleme, each entry in the text index contains a text word (the key), and an occurrence tuple for each occurrence containing document identifier, preorder and postorder positions, and level (in the actual implementation the identifier is only stored once, followed by the occurrences in that document). Temporal pattern-tree queries can be supported by an extension of the text index that also supports temporal text-containment queries [20].

  The index can also be used in other approaches to pattern tree queries, including a trie-based algorithm that provides a set of candidate documents. The candidate documents have to be matched with the pattern in order to determine if they are actual matches. The resulting candidate documents are those that match both on path and words.

Our approach. In this paper, we will describe several approaches to improve the performance of temporal text-containment queries.

History tells us that even though a large amount of index structures have been proposed for various purposes, database companies are very reluctant to make their systems more complex by incorporating them into their systems, and still mostly support the "traditional" structures, like B-trees and hash files. As a result, we believe that the techniques used to support temporal features should be compatible

with existing architectures. Thus we put emphasis on techniques that can easily be integrated into existing architectures, preferably using traditional index structures as well as a query processing philosophy compatible with existing architectures. In this paper, we discuss alternative temporal text-indexing techniques, implement three of the them, and compare the performance of the different approaches.

It should be noted that even though our work is done in the context of document databases, the results are also applicable for temporal text-containment queries in temporal relational and object databases.

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 provide the context and general assumptions we make, and also give an overview of the V2 temporal document database system in which we have integrated the ideas described in this paper. In Section 4 we describe approaches to temporal text indexing. Three of these approaches have been implemented, and in Section 5 we compare the performance of the different approaches. Finally, in Section 6 we conclude the paper.

2 Related work

A number of systems have been made for storing versioned documents. Of the most well-known are probably RCS [24] and CVS (which is based on RCS). These are mostly used for storing source code versions. It is possible to retrieve versions valid at a particular time, but they do not support more general document query features.

One project related to our V2 project, and which supports retrieving pages that were valid at a particular time is the Internet Archive Wayback Machine [13]. However, they do not yet support text-containment or other temporal queries.

An approach to temporal document databases is the work by Aramburu et al. [3]. 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. The system is implemented on top of Oracle 8.

The only research work we are aware of the directly focuses on access methods for general temporal document querying, is [2] by Anick and Flynn which describes how to support versioning in a full-text information retrieval system. In their proposal, the current versions 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 a version identifier. This is not applicable for transaction-based document processing, where all versions created by one transaction must have the 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. It is also difficult to make temporal zig-zag joins (needed for multi-word temporal text-containment queries) efficient.

Storage of versioned XML documents is studied by Marian et al. [15] and Chien et al. [7–9]. Chien et al. also considered access to previous versions, but only snapshot retrievals. Algorithms for Temporal XML query processing operators are proposed in [20].

In the context of querying semistructured data, Chawathe et al. [5,6] presented a model for representing changes in such data, DOEM, and a language, Chorel, for querying changes.

The inverted file index used as basis in our work is based on traditional text-indexing techniques, see for example [26]. Our implementation into B-trees into the V2 system has also been done in other contexts. For example, Melnik et al. [16] describes a system where the inverted file indexes has been implemented in a way similar to ours.

3 Context and assumptions

In this section, we first give a short introduction to text indexing, and then we give an overview of the V2 temporal document database system.

3.1 Text indexing

The basic lookup operation in text indexing is to retrieve the document identifiers of all documents that contain a particular word w. The most common access method for text indexing is the inverted file, which is also the basis of our approaches. An inverted file index is a mapping from a term (text word) w to the documents $d_1, d_2, \ldots, d_j$ where the term appears. The collection of distinct terms $w_1, \ldots, w_k$ is called the vocabulary.

Posting lists. In the inverted file index, a posting list $PL = (w, d_1, d_2, \ldots, d_m)$ is created for each index term, where w is the text word, and d_i are the document identifiers of the documents the term appears in. The tuple $P = (w, d_i)$, i.e., an index term and a document identifier, is called a *posting*. Traditionally the size of text indexes is reduced by using some kind of compression. Compression techniques

usually exploit the fact that documents can be identified by a document number, making them ordered, and that in this way each document number d_i in the posting list can be replaced by the distance $d = d_i - d_{i-1}$. This distance usually requires a lower number of bits for its representation. Two examples of codings that can be used for this purpose are Elias encoding [11] and variable length encoding [12]. In our implementations, we use a light-weight coding as will be described later in this paper.

Inverted-file index storage. An inverted-file index can be realized in a number of ways, and the two most popular alternatives have been:

(1) Using an index structure for the index terms (vocabulary), containing pointers to posting lists that are stored separately in a heap file (a binary file manipulated as if it were main memory). Nowadays, the vocabulary will normally fit in main memory, making storage of the posting lists the critical issue. This is the traditional approach, for example suited for libraries [25]. If the files are static, 100% storage utilization can be achieved, and all postings related to one term can be stored contiguously on disk, making subsequent retrieval very fast. If incremental updates are allowed, a buddy system or chained overflow blocks can be used. A buddy system with consecutively stored blocks in each bucket has the advantage of fast retrieval because of sequential reading from disk, but gives on average only 75% full buckets [10]. Chained overflow has a higher space utilization for large lists, but retrieval can be inefficient as many random reads will be necessary.

    Many improvements to the basic structure have been proposed, and they often make use of the important characteristic of inverted lists that over 90% of posting lists will be relatively small (they follow a Zipfian distribution), and that vocabulary will continue to grow indefinitely. This means we have a few very large posting lists, and many small lists. One example is the approach by Tomasic et al. [25] where short inverted lists (i.e., lists for infrequently used terms) are stored together in buckets, and long inverted lists are stored separately. A directory that can be resident in main memory is used to index the long lists. When a bucket overflows, the largest short list in that bucket is made a long list and stored separately. Incremental updates are supported by maintaining in-memory lists which, based on certain policies, are periodically merged with the lists on disk.

(2) Storing the posting lists in the leaf nodes in the vocabulary index. This is more suitable in the case of dynamic databases. One example of using this approach is presented by Melnik et al. [16], in the context of a distributed indexing system based on inverted files. Similar to our approach, they also store postings in chunks in a B-tree index based on Berkeley DB.

Combinations of these alternatives are also possible. In general, the vocabulary index is a multi-tree variant (usually B-tree), because this supports prefix searches at no additional cost.

3.2 An overview of the V2 temporal document database system

The techniques described in this paper have been implemented in the V2 temporal document database system. In order to make this paper self-contained, and provide the context for the rest of this paper, we give in this section an overview of V2. For a more detailed description, and a discussion of design choices, see [21].

3.2.1 Document version identifiers

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 an object identifier in an object database.

3.2.2 Time model and timestamps

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.

The time model in V2 is a linear time model (time advances from the past to the future in an ordered, step-by-step fashion). However, in contrast to most other transaction-time database systems, V2 does support "reincarnation", i.e., a (logically) deleted version can be updated, thus creating a non-contiguous lifespan, with possibility of more than one tombstone for each document. Support for reincarnation is particularly interesting in a document database system because even though a document is deleted, a new document with the same name can be created at a later time (in the case of a Web-DW this could also be the result of a server or service being temporarily unavailable, but then reappearing later). In a document database system, it is also possible that a document was deleted by a mistake, and with temporal support the document can be brought to life again by retrieving a historical version and rewriting this as a new current version.

3.2.3 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 or during a particular period p. V2 also supports some basic operators. In contrast to many existing systems that support versioning of documents, time is an integrated concept of V2, and is efficiently supported by the query operators.

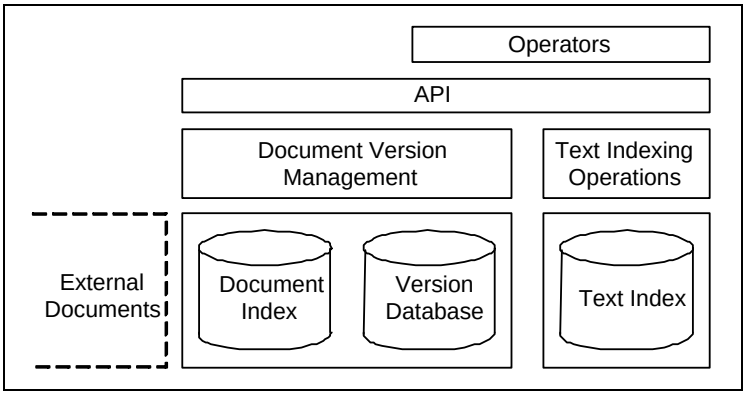


Fig. 1. The V2 prototype architecture.

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

3.2.4 Design and implementation

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 [22], which we employ to provide persistent storage using B-trees. However, we will later consider using an XML-optimized/native XML storage engine instead. Using native XML storage should result in much better performance for many kinds of queries on XML documents, in particular those only accessing subelements of documents, and also facilitate our extension for temporal XML operators.

The architecture of V2 is illustrated in Figure 1, and the main modules 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 describe these modules/structures in more detail.

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. This is interesting, because it makes it easy to retrieve all versions inserted during a certain VID interval (which can be mapped from a time interval). One application of this feature is reconnect/synchronization of mobile databases, which can retrieve all versions inserted into the database after a certain VID (last time the mobile unit was connected). In order to support reverse mapping from VID to document name and time, this information, together with other low-level metadata, is stored in a separate meta-chunk together with the document.

Document name index: A document is identified by a *document name*, which can be a filename in the local case, or 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. For each document the document name is stored and whether the document is temporal (i.e., whether previous versions should be kept when a new version of the document is inserted into the database). For each document *version*, some metadata is stored in structures called *version descriptors*: 1) timestamp and 2) whether the actual version is stored compressed. For some documents, the number of versions can be very high. In a query we often want to query only versions valid at a particular time. In order to avoid having to first retrieve the document metadata, and then read a very large number of version descriptors spread over possibly a large number of leaf nodes until we find the descriptor for the particular version, document information is partitioned into chunks. Each chunk contains a number of descriptors, valid in a particular time range, and each chunk can be retrieved separately. In this way, it is possible to retrieve only the descriptors that are necessary to satisfy the query. The chunks can be of variable size, and because transaction time is monotonously increasing they will be append-only, and only the last chunk for a document will be added to. When a chunk reaches a certain size, a new chunk is created, and new entries will be added to this new chunk.

We believe that support for temporal data should not significantly affect efficiency of queries for current versions, and therefore either a one-index approach with sub-indexes [19] or a two-index approach where current and historical information is stored in different indexes should be employed. One of our goals is to a largest possible extent using structures that can easily be integrated into existing systems, and based on this we have a two-index approach as the preferred solution. An important advantage of using two indexes is that the current version index can be assumed to be small enough to always fit in main memory, making accesses to this index very cheap. The disadvantage of using separate current-version and historical-version indexes, is potentially high update costs: when a temporal document is updated, both indexes have to be updated. This could be a bottleneck. To avoid this, we actually use a more flexible approach, using one index that indexes the most recent n document versions, and one index that indexes the older historical versions. Every time a document is updated and the threshold of n version descriptors in the current-version index is reached, all but the most recent version descriptors are moved to the historical-version index. This is an efficient operation, effectively removing one chunk from the current-version index, and rewriting it to the historical-version index.

Text indexing: A text-indexing 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 the sum of the cost of the individual updates.

As mentioned previously, one of our goals is that it should be possible to use existing structures and indexes in systems, in order to realize the V2 structures. This also applies to the text-indexing module. The text-indexing module actually provides three different text-index variants suitable for being implemented inside ordinary B-trees. All the variants have different update cost, query costs, and disk size characteristics. The variant that is most likely to be used, uses one or more chunks for each index word. Each chunk contains the index word and a number of VIDs. For each VID inserted into the chunk, the size increases until it reaches its maximum (we use a value of 400 bytes). At that time, a new chunk is created for new entries (i.e., we will in general have a number of chunks for each indexed word).

As mentioned in Section 3.1, the size of text-indexes is usually reduced by using some kind of compression. Given the moderate size of our chunks and the desire to keep complexity and CPU cost down, we use a simple approach. We use a constant-size small integer in order to represent the distance between two VIDs. Each chunk contains the ordinary-sized VID for the first version in the chunk, but the rest of the VIDs are represented as distances, using short 16-bit integers. In the case when the distance is larger than what can be represented using 16 bit, a new chunk is started.

4 Temporal full-text indexing

In this section we first discuss query categories in temporal text queries in order to show what kind of queries the access structures should support, and then we describe the design of appropriate data structures.

4.1 Accesses in temporal text-containment queries

In non-temporal/current-version full-text indexes, we can divide typical accesses into the following categories:

- **OneWord/AllPost:** Lookup for all postings for a word resulting from a one-word query. An example of a query generating this type of lookups is “find all

documents that contain the word w".

- **RangeWords/AllPost:** Lookup for all postings for a range of words with the same prefix. An example of a query generating this type of lookups is "find all documents that contain the words $w*$", where $w*$ denotes all words with prefix w.
- **OneWord/SubsetPost:** Lookup for a (non-contiguous) subset of postings for a word, using a document identifier as search key when searching in the postings for the word w. This can for example be the case during a multi-word query when a zig-zag join algorithm is used. In this case, not all postings for all words need to be retrieved. An example of a query that generates this type of lookup is "find all documents that contain the words w_1 and w_2".
- **RangeWords/SubsetPost:** Lookup for a subset of postings, for a range of words with the same prefix. An example of a query that generates this type of lookup is "find all documents that contain the words w_1* and w_2*".

In temporal text-indexing we also want to support lookups in the temporal dimension. In our context, this means extending the access categories above to include *query on document versions valid at a particular time*, and query on *document versions valid in a particular time range.* An example of such a query is "find all documents that contained the word w at time t".

We assume that queries for complete words will occur more frequently than word prefix queries. In order to reduce the complexity of the following discussion and the process of analyzing the index alternatives, we will only consider the OneWord categories as described above. In all the techniques described here, supporting RangeWords is straightforward.

When temporal support is added, we extend the categorization with a new parameter, which is **TSnapshot** or **TRange**, giving the following categories:

(1) **OneWord/AllPost/TSnapshot:** Lookup for all postings valid at time t for a word resulting from a one-word query. An example query is "find all document versions valid at time t that contained the word w".

(2) **OneWord/SubsetPost/TSnapshot:** Lookup for a (non-contiguous) subset of postings for a word. An example of a query that generates this type of lookup is "find all document versions valid at time t that contained the words w_1 and w_2". Another situation where only a subset of postings is accessed, is in a query on the versions of *a particular document* (or a subset of the indexed documents). An example of such a query is "find all document versions of *particular document* that contained a word w at a particular time t.".

(3) **OneWord/AllPost/TRange:** Lookup for all postings valid in the time range $[t_1, t_2>$ for a word resulting from a one-word query ($[t_1, t_2>$ denotes a closed-open interval). An example query is "find all document versions valid in the time range $[t_1, t_2>$ that contained the word w."

(4) **OneWord/SubsetPost/TRange:** Lookup for (non-contiguous) subset of postings for a word. An example of a query that generates this type of lookup is "find all document versions valid in the time range $[t_1, t_2>$ that contained the words w_1 and w_2".

4.2 Access structures for supporting temporal text-containment queries

In this section we discuss possible techniques that can be employed in order to improve performance in temporal text-containment queries. In general, the techniques can be classified into:

- Techniques that 1) first perform a text-index query using an 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:
  - **Basic:** No additional structures, but uses time information stored in the meta-chunk (document version header) that is stored together with the document version itself.
  - **TIVID index:** Summary time index, map from time to VID.
  - **VIDPI index:** Indexes validity time for versions, mapping from VID to time period for every version.
- Techniques that use a temporal text index, so that a lookup $L(w, t)$ in the index only returns the VIDs of document versions that contained the actual word at the particular time or time period, i.e., it is not necessary with any particular post-processing of the VIDs or searching other access structures after the lookup in the temporal text index:
  - **Temporal text index:** A temporal text index realized by including the validity time for each document identifier in the text index.
  - **Current-version text index:** Use a separate text index for postings of current version documents (i.e., non-deleted documents). Avoids postprocessing for queries on current versions only.

The techniques will now be described in more detail.

4.2.1 Basic approach

The basic and straightforward implementation for solving temporal text queries is to:

(1) Retrieve the VIDs of all versions that contain a particular word (or set of words) by a lookup operation in the text index.
(2) Perform a time-select operation on the VIDs. The versions are indexed using the VID, and the timestamp and document name is stored in a header together with the document version itself, so that the time-select operation for a VID

can be implemented by a lookup in the version database in order to find the document name, followed by a lookup in the document name index in order to find the end timestamp t_E. With this information, it is possible to determine whether the actual document version was valid at the time/period we query for.

The text index lookup is relatively cheap, as many postings for a particular word are stored clustered. However, the second phase is expensive: even if only the header of the document version has to be read, a random read operation is needed for each VID returned from the text index, and a random read operation is also needed for a lookup in the document name index (although it can be assumed that many VIDs actually represent versions of the same document, and this will in most cases reduce the average number of random reads for document name index pages). In any case, temporal text-containment queries for frequently occurring words can be very time consuming if this technique is used.

It should be emphasized that even in applications where only a limited set of results should be returned, for example maximum 10 matches, a much larger number of versions might have to be checked before we reach the maximum number that match on both word containment and validity time.

In practice many searches are for documents containing a set of words. In this case, each additional word in the query in general reduces the number of possible document versions. In many cases, the number of document versions matching all words, and that have to be checked for validity time, can be low enough to also make this basic technique feasible.

4.2.2 TIVID index

A simple and inexpensive technique that can reduce the number of document versions that have to be retrieved and checked for validity time is to use a time-VID (TIVID) index. The TIVID index is created by writing a (time,VID) tuple for each new VID. This list is append-only and the size of the entries is small, and as such contributes very little to the total document update time.

When performing the time selection for a time t or a time period ending with t_E, we know that no version created after t (or t_E) can be a match. Thus, by a lookup on t in the TIVID index, we can determine the maximum VID that can possibly match. On average, the use of the TIVID index reduces the number of versions that have to be checked to 50% of the original number of versions. It can also reduce the amount of postings that have to be retrieved from the text index. The postings are stored in sequence according to the VID, so that during retrieval from the text index we can stop when we come to a VID that is larger than the one determined to be an upper bound.

VIDs are assigned as integers increasing by one for each version, so that in the index we do not have to store the VIDs. An array of timestamps is sufficient, one for each VID. In practice, a number of versions will have the same timestamp (created by the same transaction), and we do not really need every timestamp as we use the TIVID index only to determine a maximum bound. Thus, in the implementation we only store the timestamp of every i^{th} version. When the TIVID index is realized as an array, i has to be chosen when the database is created and kept constant. In practice it is unlikely that a really bad value for i will be chosen (values like 100 or 1000 are typical values that can be used). However, considering the consequences and the fact that the TIVID index will still be very small, we decided in our implementation to store both the time *and* the VID so that i can be changed later if necessary. The size of the TIVID index will be relatively small compared to the rest of the text index and the version database, so it should still be small enough to always be resident in main memory.

Assuming the postings for each word in the text index are ordered on VID, it is possible to use the VID retrieved from the TIVID index as a stop condition when retrieving postings from the text index, and not only afterwards when the posting list is processed. This will on average reduce the number of postings to retrieve to 50%. This is especially beneficial in the case of frequently occurring words in combination with query for a time not close to time t_{Now}.

4.2.3 VIDPI index

The TIVID index only halves the number of versions that has to be checked during the time-select phase. In addition, the TIVID index is not very useful in queries for timestamps close to $t = t_{Now}$. The reason is that the TIVID index only gives an upper bound of possible matches, and the reduction in the number of possible matches is reduced as timestamps get closer to $t = t_{Now}$. When time $t = t_{Now}$ in the query, there is no reduction at all in the number of possible matches. One step further from the TIVID index approach, is to store the *validity period* of every version in a separate index. We call this index a VIDPI index, and it maps from VID to period. That is, each tuple in the VIDPI index contains a VID, a start timestamp, and an end timestamp. In the V2 prototype the VIDPI index is implemented as a number of chunks, where every chunk is an array where we only store the start VID to which the period tuples are relative. In this way we reduce the space while still making delete operations possible.

Even for a large number of versions the VIDPI index is relatively small. It makes it possible to avoid any lookups in the version database and the document name index in order to determine the validity time of the version. However, the maintenance cost is higher: in addition to storing the VID and timestamp at transaction commit time by appending to the index, every document update involves updating the end timestamp in the previous version, from U.C. (until changed) to a new value

which is the start (commit) timestamp of the new version. However, considering the initial text indexing cost for every version this cost is not significant in practice (see Section 5.3.1).

Similar to the case when using a TIVID index, it is possible reduce the number of postings to be retrieved from the text index if we can determine the maximum VID that can possibly be an answer to the query. In the case of a TIVID index this did not incur any additional cost, because the lookup in the TIVID index in order to determine the maximum VID would have to be done in any case (that was the purpose of the TIVID index). In the case of the VIDPI index, determining the maximum VID is more expensive. Instead of one lookup as in the TIVID index case, a number of lookups using binary search is necessary in order to find the maximum VID. However, if we assume the VIDPI index is resident in main memory, this is not a problem.

4.2.4 Temporal text index

Our first approach to solving the temporal text containment problem was actually to try to develop a temporal text index. Possible approaches include:

- Store the time together with the posting.
- Consider time, word, and VID as three dimensions and index accordingly in a multidimensional index structure.
- Store a separate temporal subindex in the text index for each index term.

4.2.4.1 Posting-time approach The basic approach is to store time together with the posting. However, for several reasons we do not consider this to be a good alternative:

- The size of the index would increase considerably, even though timestamps also could be encoded as differences (similar to our storage of VIDs in the text index, note however that using a granularity of seconds in timestamps, 16 bits would only be enough for a difference of 18 hours).
- In order to be useful, both the start and end timestamps are needed. This will result in a more expensive document update.

4.2.4.2 Indexing dimensions A multidimensional indexing technique, for example an R-tree variant, can be used to index the postings in a 3-dimensional space, with time, word, and VID as the dimensions. However, R-trees are best suited for indexing data that exhibits a high degree of natural clustering in multiple dimensions [23]. This is not the case in our context, and one of the results would be a high degree of overlap in the search regions of the non-leaf nodes. Although a *segment*

R-tree can reduce this problem, it will have a higher insert cost [23]. The fact that we do not know the end time of a new word occurrence further complicates the use of an R-tree (although this problem can be solved [4]). Another problem using an R-tree is that words would have to be replicated, something that would considerably increase the size of the tree. In general, it is better having a "less efficient" access methods where most of the structure can stay in main-memory buffers, than a "more efficient" access method where more disk accesses are needed in practice.

4.2.4.3 Temporal subindexes A separate temporal subindex for each index term in the text index can be maintained. The temporal index can be based on one of the well-known temporal access methods, for example the TSB-tree [14]. Again there is a possibility of much replication and the problem of ever-increasing end-time for current versions. However, what we consider to be the most important problem with this approach is that it is difficult to integrate into existing systems, making it less attractive in practice.

4.2.5 Separate current-version text index

In many application areas, queries on current version documents can be assumed to dominate. In this case, a separate text index for words occurring in current version documents can be beneficial. The advantages of this approach are that the current-version text index will be smaller than the index that indexes all document versions, and there is a higher probability that the relevant parts of the index will be resident in memory, so that few disk accesses will be needed. However, there are also some possible disadvantages with this approach:

(1) Using two indexes means duplicating the indexed key (word) and their associated overhead.
(2) In many temporal queries, both indexes have to be searched (for example, when we ask for a version valid at time t, this can be an historical version, or it can be the last/current version).
(3) The maintenance cost is much higher. For example, when a document is updated, the entries reflecting the words in the previous document have to be moved to the index that indexes historical versions. If the postings are stored sorted (which is useful for, e.g., zig-zag joins), the insert operation is more complicated than when entries are just appended, and it is also more difficult to maintain a high page-fill factor.

Whether or not using a separate current-version text index is beneficial depends heavily on the document update rates (note: *not* the create rate) and the fraction of queries that are to current-version documents. We believe the separate current-version text index can be useful in many application areas, together with one of the other approaches. Thus, we consider the question of having a separate current-

version text index or not to be orthogonal to the main issue of this paper, and in order to emphasize the main topic of this paper we do not discuss it further.

4.2.6 Discussion

Based on the discussion above, we consider a VIDPI index in combination with a text index to be most efficient for queries. However, with small documents (or other contexts, e.g., relational or object databases), it could increase the update cost more than desired. In that case, the TIVID index could be an alternative. An important advantage of using an approach based on the TIVID or VIDPI index is that in many systems, the complete TIVID/VIDPI index and the text index can be kept in main memory. The size of text index plus the TIVID or VIDPI index will still be less than 10% of the size of the version database in typical cases. By efficient use of these indexes, only the document versions that are actually wanted need to be retrieved, and normally the most frequently accessed of these can be kept in a buffer.

5 Performance

The performance of a system can be compared in a number of ways. For example, benchmarks are useful both to get an idea of the performance of a system or a technique, as well as comparing the system with similar systems. However, to our knowledge there exist no benchmarks suitable for temporal document databases. An alternative technique that can be used to measure performance is the use of actual execution traces. However, again we do not know of any available execution traces (this should not come as a surprise, considering that this is relatively new research area). In order to do some measurements of our system, we have created an execution trace based on the temporal web warehouse described in Section 1.

5.1 Acquisition of test data and execution trace

In order to get a reasonable amount of test data for our experiments, we have used data from a set of web sites. The available pages from each site are downloaded once a day, by crawling each site starting with the site's main page. This essentially provides an insert/update/delete trace for our temporal document database.

In general, many web sites have relatively static pages, and the main cost is the first loading of pages. The cost of storing this kind of sites can be considered equal to loading the first set of pages as described above. The maintenance cost for such sites is only marginal (although determining whether a page has changed or not incurs a non-marginal cost). However, other sites are more dynamic. We wanted to

study maintenance of such sites, and in order to achieve this, we used a set of sites that we expected to be relatively dynamic. This included local pages for courses at our university, and a number of online versions of major newspapers (here the main pages are dynamic, but the articles themselves fairly static, although some of them are actually edited after publication).

The initial set of pages was of a size of 91 megabytes (MB) (approximately 10000 web pages). An average of approximately 500 web pages were updated each day, approximately 300 web pages were removed (all pages that were successfully retrieved on day d_i but not available on day d_{i+1} were considered deleted), and approximately 350 web new pages were inserted.

The average size of the updated pages was relatively high, resulting in an average increase of the version database of 45 MB for each set of pages loaded into the database.

We kept the temporal snapshots from the web sites locally, so that insertion to the database was essentially loading from a local disk. In this way, we isolated the performance of the database system, excluding external factors as for example communication delays.

For our experiments, we used a computer with a 1.4 GHz AMD Athlon CPU, 1 gigabytes (GB) RAM, and 3 disks each with capacity 18.4 GB. One disk was used for the operating system (FreeBSD) and the V2 system, one for storing the database files, and one for storing the test data files (the web pages). The version database and the text index had separate buffers, and the size of these were explicitly set to 100 MB and 200 MB, respectively.

5.2 Measurements

We now describe the various measurements we have done.

Loading and updating. The first part of the tests is loading data into the database. Loading data into a document database is an expensive operation, mainly because of the text indexing. Our text indexes are dynamic; i.e., they are immediately updated. This implies that in the case of frequent commit operations, the result will be a very low total update rate. However, for our intended application areas we expect large amounts of data to be loaded in bulk in each transaction. For example, in the case of the web warehouse application we assume commit is only performed between each loaded site, or set of sites. We load all the updates for one day of data in one transaction. For each parameter set, we measure the cost in terms of used time for a number of operations. The most important measurements, which are discussed in this paper, are:

- The update cost for the last transaction, when the last set of documents is applied to the system. This cost will increase with increasing database size. With small database sizes, the buffer space is larger than the size of the database. This makes operations like retrieval of previous versions for comparison purposes cheap. As the database increases and it does not fit completely in main memory any more, the update cost increases.
- After the initial updates based on test data, we also insert a total of 10000 new pages with document names not previously found in the database, in order to study the cost of *inserting* into a database of a certain size, compared to updates (when inserting, no comparison with a previous version is necessary).
- In order to study the cost of insertions of individual documents, when only one document is inserted during one transaction, we also insert a number of documents of different sizes from 800 bytes to 30 KB, using a separate transaction for each. As a measure of this cost, we use the average time of the insertion of these documents. Because we have not enabled logging, the result is that every disk page changed during the transaction has to be forced to disk. This is a relatively costly operation.

Query and retrieval. After loading the database, we performed tests to measure the basic query performance. The operations are text lookup of single words, with some additional temporal operations as described below. In this paper, we give the results for two categories of words:

- Frequently occurring words. We define a "frequently occurring word" as a word that appears in at least 10% of the documents. In our database, all postings for one frequently occurring word typically occupies in the order of 10 disk pages.
- Moderately frequent words. We define a "moderately frequent word" as a word that occurs in approximately 0.5% of the documents. In our database, all entries for a moderately frequent word fit in one disk page (but are not necessarily stored in one disk page, because the chunks can be in two different pages).

It should be noted that we have also studied a third category of words, *infrequently occurring words*, where each word only exists in a few documents. Due to the fact that these results give no new insight into the problem, we do not comment further on this category.

For each query we used different words, and for each query type we used several of the words and used the average query cost as the result. In practice, in an application a set of such basic operations will be used, and only the resulting documents are to be retrieved. Thus, for each query we do not retrieve the documents; we are satisfied when we have the actual VIDs available (the retrieval of the actual versions is orthogonal to the issue we study here). The query types used were:

- AllVer: All document versions, current as well as historical, that contain a particular word.

- TSelCurrent: All currently valid document versions that contain a particular word.
- TSelMid: All document versions valid at time t that contained a particular word. As the value for time t we used the time when half of the update transactions have been performed. We denote this time $t = t_{Mid}$.
- PSelF: All document versions that contained a particular word and were valid sometime in the period from the first insertion into the database and until $t = t_{Mid}$.
- PSelL: All document versions that contained a particular word and were valid some time in the period from $t = t_{Mid}$ and the current time $t = t_{Now}$.

For all containment queries involving time, the meta-chunk of the actual versions has to be retrieved when we have no additional time indexes (TIVID or VIDPI indexes).

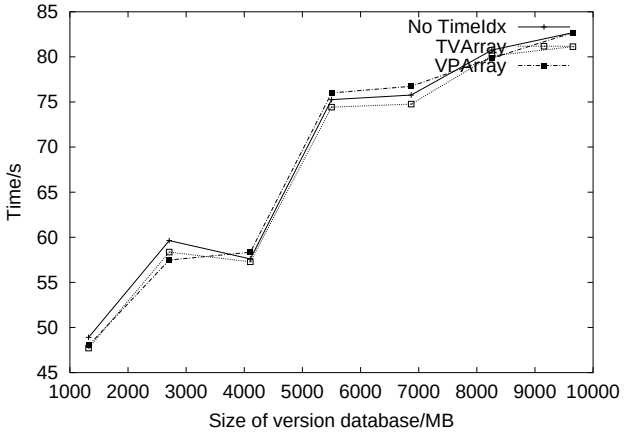
5.3 Results

We now summarize some of the measurement results. The sizes given on the graphs are the sizes of the version databases in MB, and the measured time is given in seconds. As described, the first transaction loads 10000 documents, giving a total database size of 91 MB, and each of the following transactions increases the size with an average of approximately 45 MB. The final size of the version database is 9.7 GB (or 2.0 GB if we enable compression).

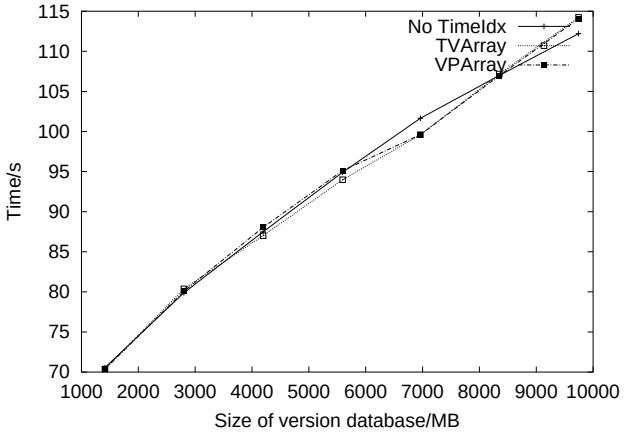
5.3.1 Load and update costs

The initial loading of the documents into the database was most time consuming, as every document is new and have to be text indexed. The loading time of the first set of documents (inserting 10000 documents) was on average 55 s. At subsequent updates, only updated or newly created documents have to be indexed. In this process, the previous versions have to be retrieved in order to determine if the documents have changed. If a document has not changed, the new version is not inserted. Figure 2a shows this update cost for different database sizes. The cost of updating/inserting a set of web documents increases with increasing database size, because the probability of finding the previous version in the main memory buffer is reduced. The same applies to pages in the text index where postings have to be inserted. It is interesting (but comes as no surprise) that employing a TIVID or VIDPI index incurs only a marginal extra cost.

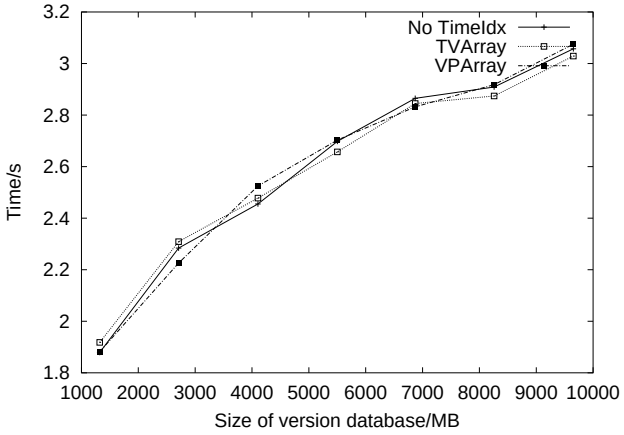
If a document does not already exist in the database (the name is not found in the document name index), there is no previous version that has to be retrieved. However, it is guaranteed that the document has to be inserted and indexed. This is



(a) Update set of web pages.



(b) Insert new set of web pages.



(c) Insert single document.

Fig. 2. Load and update performance.

more expensive on average. This is illustrated in Figure 2b which shows the cost of inserting a set of new documents into the database as described previously.

Figure 2c illustrates the average cost of inserting a single document into the database, in one transaction. The cost increases with increasing database size because pages in the text index where postings have to be inserted are not found in the buffer. It also illustrates well that inserting single documents into a document database is expensive, and that bulk loading, with a number of documents in one transaction, should be used when possible.

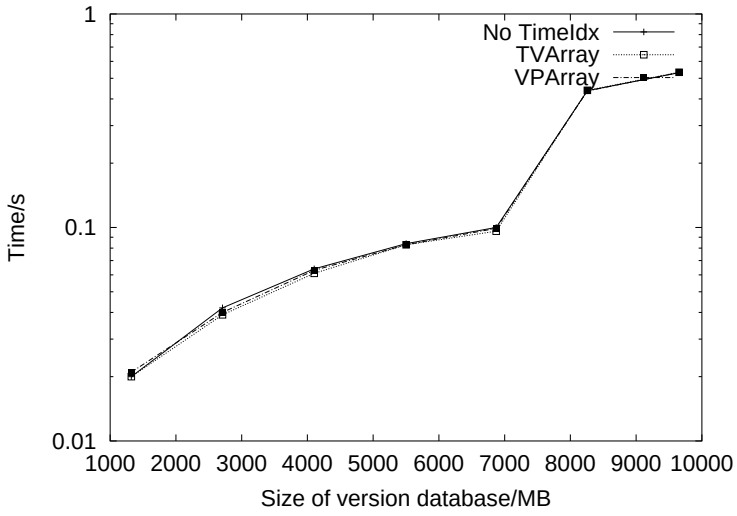
5.3.2 Query costs

Figure 3 illustrates the cost of retrieving the VIDs of all document versions containing a particular word. As expected, the cost increases with increasing database size. The main reason for the cost increase with smaller database sizes is a higher number of versions containing the actual word, resulting in an increasing number of VIDs to be retrieved. When the database reaches a certain size, only parts of the text index can be kept in main memory, and the result is reduced buffer hit probability (as is evident by the sharp increase when the database size reached 7 GB).

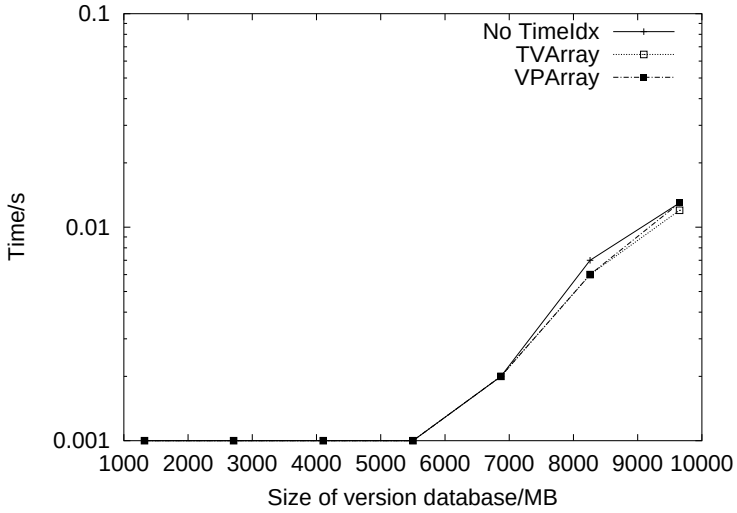
Figure 4 illustrates the average cost of retrieving the VIDs of all document versions valid at time $t = t_{Mid}$ that contained a particular word. The first thing to note is the improvement from using a TIVID index compared to no time index (basic approach). As expected, for larger databases, the query cost is reduced to half of the original cost. However, the real improvement comes from using the VIDPI index. For frequently occurring words, the query cost is reduced to 0.4% of the original cost. The VIDPI index will in general be small enough to fit in main memory, so the main cost here (60%) is to actually retrieve the VIDs from the text index. This also illustrates the potential benefits of developing a good temporal text index (this is left as further work), where only a subset of the postings had to be retrieved. For moderately frequent words, the performance gain using the VIDPI index is even higher.

As noted in Section 4.2.3, the TIVID index is not very useful in queries for timestamps close to $t = t_{Now}$. This is illustrated in Figure 5, where the query is for all current versions containing a particular word. The costs for TIVID index and using no time index overlaps, as expected.

Figure 6 shows the cost for queries for document versions valid sometime during a time period and that contained a particular word. Figure 6a shows the cost of a query for the period from first insertion until the mid point, and Figure 6b shows the cost of a query for the period from the midpoint until t_{Now}. In the first case, the TIVID index reduces the cost to the half, but in Figure 6b we again see the shortcomings of the TIVID index in queries where the time is close to or equal to t_{Now}.



(a) Frequent words.



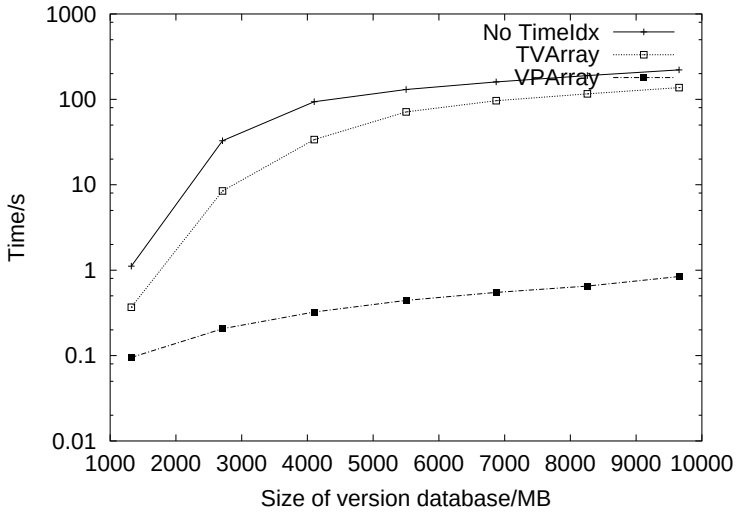
(b) Moderately frequent words.

Fig. 3. Text containment, all versions.

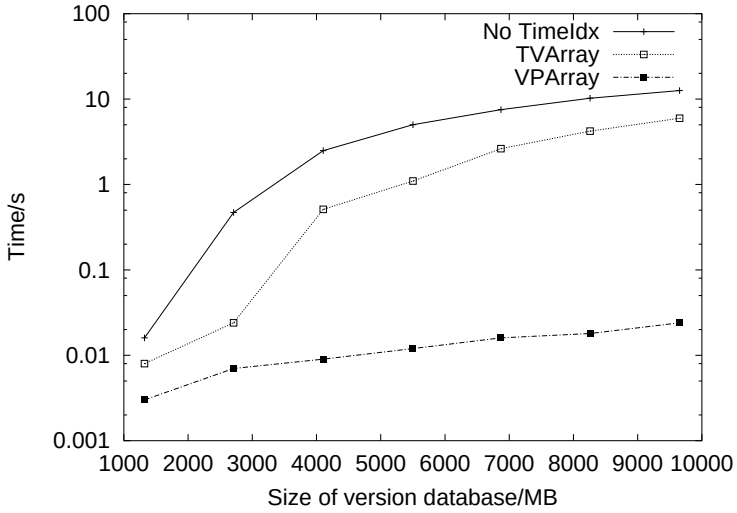
5.3.3 Summary

As can be seen from the figures, using a TIVID or VIDPI index incurs only marginal extra cost. Maintaining the VIDPI index in our context is only marginally more costly than using the more query efficient VIDPI index, so we conclude that with normally sized documents as we have used here, the VIDPI index should be the preferred choice.

It should be noted that in the time measurement the database overhead is included.



(a) Frequent words.



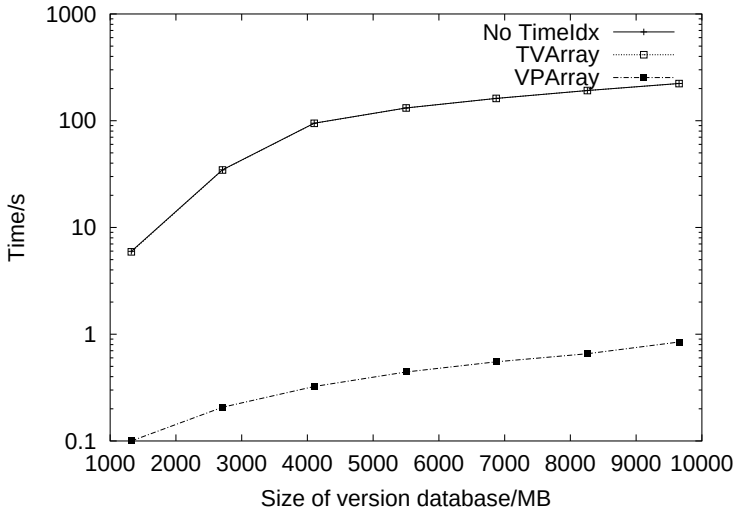
(b) Moderately frequent words.

Fig. 4. Temporal text containment, time selection at time $t = t_{Mid}$.

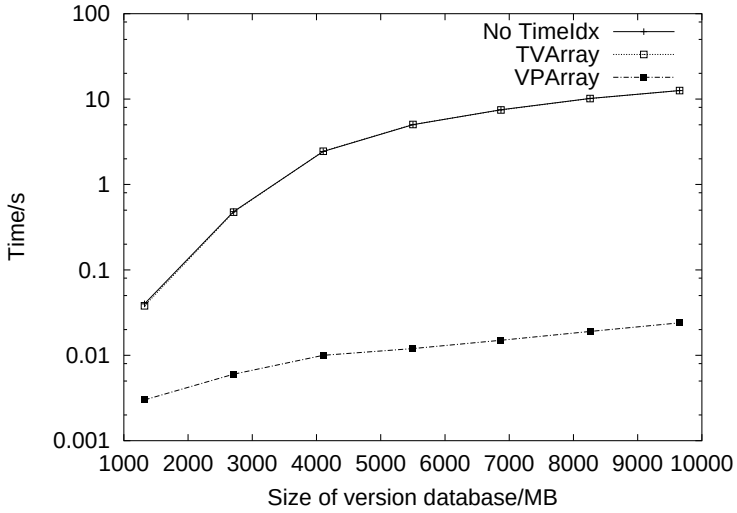
However, the basic cost due to overhead is the same for all alternatives.

6 Conclusions

We have in this paper described how to manage documents and index structures in temporal document databases in a way that makes temporal text-containment querying feasible. We described and discussed different algorithms and index struc-



(a) Frequent words.

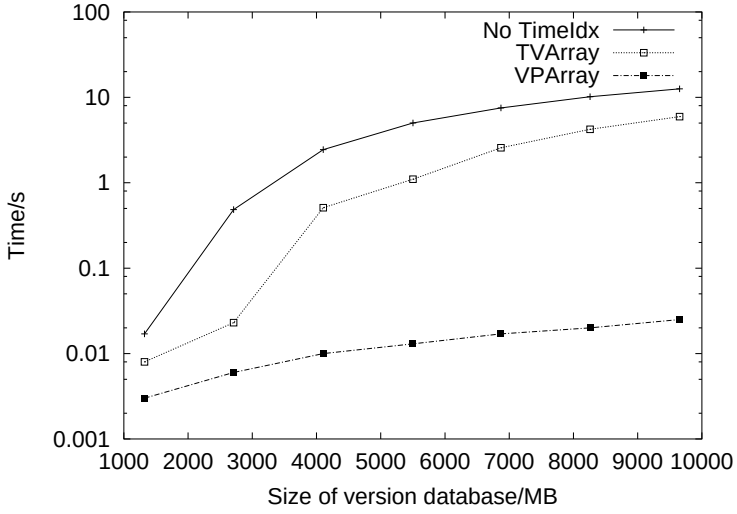


(b) Moderately frequent words.

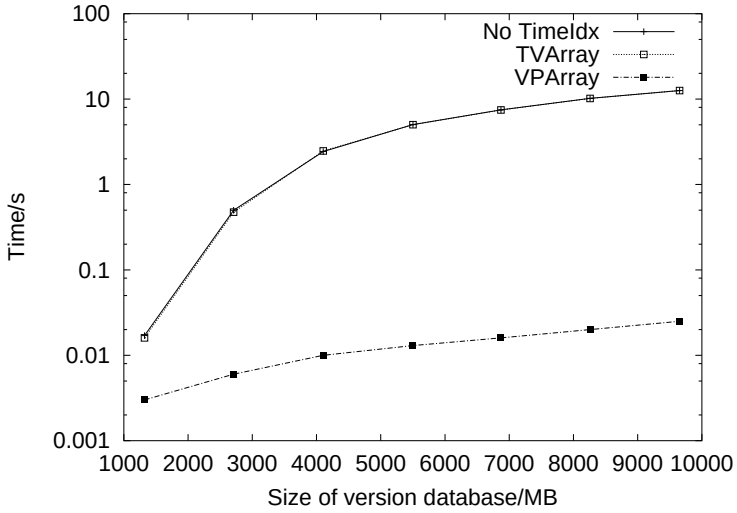
Fig. 5. Temporal text containment, time selection of current versions at time $t = t_{Now}$.

tures that can be used to improve such queries, and why some of the alternatives are not suitable in practice. We implemented three of these structures in the V2 temporal document database system, and based on tests with temporal web data we studied the performance of the index structures.

The most important conclusion is that even these simple access structures can give a very high performance improvement. Traditional structures can be used, and the additional update cost is only marginal.



(a) $p = [0, t_{Mid}>$



(b) $p = [t_{Mid;}, t_{Now}>$

Fig. 6. Temporal text containment, period selection for moderately frequent words.

The techniques described in this paper are also applicable for indexing text attributes in temporal relational and object databases. However, there is an important difference that has to be kept in mind: a tuple/object is in general much smaller than a document, and the typical text attribute will have only a few words. As a result, the TIVID/VIDPI index maintenance cost will be more costly compared to the text indexing cost than in the document database case.

Future works include a theoretical analysis of the algorithms and access methods used in this paper. We also plan to study closer what we see as the main bottleneck

in our application area, the document update and insert operations. Due to the text-indexing cost it is in general quite expensive. We believe that further work should focus on reducing these costs, for example using an approach based on structures like the LHAM [17] or the Persistent Cache [18].

Acknowledgments

Many of the basic ideas of this paper were developed during the 9 months the author spent as an ERCIM fellow in the Verso group at INRIA, France, in 2001. The main part was done when the author visited Athens University of Economics and Business in 2002, supported by grant #145196/432 from the Norwegian Research Council.

References

[1] V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying the XML documents of the Web. In *Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval*, 2000.

[2] 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*, pages 98–111, 1992.

[3] 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):987–1011, 2001.

[4] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas. R-tree based indexing of Now-relative bitemporal data. In *Proceedings of 24rd International Conference on Very Large Data Bases*, pages 345–356, 1998.

[5] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 4–13, 1998.

[6] S. S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *TAPOS*, 5(3):143–162, 1999.

[7] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. A comparative study of version management schemes for XML documents (short version published at WebDB 2000). Technical Report TR-51, TimeCenter, 2000.

[8] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In *Proceedings of VLDB 2001*, pages 291–300, 2001.

[9] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Version management of XML documents: Copy-based versus edit-based schemes. In *Proceedings of the 11th International Workshop on Research Issues on Data Engineering: Document management for data intensive business and scientifi c applications (RIDE-DM'2001)*, pages 95–102, 2001.

[10] D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the 13th International Conference on Research and Development in Information Retrieval*, pages 405–411. ACM, 1990.

[11] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, 1975.

[12] A. Fraenkel and S. Klein. Novel compression of sparse bit-strings — preliminary report. In *Combinatorial Algorithms on Words, NATO ASI Series Volume 12*, pages 169–183. Springer Verlag, 1985.

[13] Internet archive. http://archive.org/.

[14] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, pages 315–324, 1989.

[15] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *Proceedings of VLDB 2001*, pages 581–590, 2001.

[16] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proceedings of WWW10*, pages 396–406, 2001.

[17] P. Muth, P. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In *Proceedings of the 24th VLDB Conference*, pages 452–463, 1998.

[18] K. Nørvåg. The Persistent Cache: Improving OID indexing in temporal object-oriented database systems. In *Proceedings of the 25th VLDB Conference*, 1999.

[19] K. Nørvåg. The Vagabond temporal OID index: An index structure for OID indexing in temporal object database systems. In *Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS)*, pages 158–166, 2000.

[20] K. Nørvåg. Algorithms for temporal query operators in XML databases. In *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops, EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Prague, Czech Republic, March 24-28, 2002, Revised Papers.*, pages 169–183, 2002.

[21] K. Nørvåg. V2: A database approach to temporal document management. In *Proceedings of the 7th International Database Engineering and Applications Symposium (IDEAS)*, pages 212–221, 2003.

[22] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.

[23] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[24] W. F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[25] A. Tomasic, H. Garcia-Molina, and K. A. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 289–300, 1994.

[26] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[27] L. Xyleme. A dynamic warehouse for XML data of the web. *IEEE Data Engineering Bulletin*, 24(2):40–47, 2001.