V2: A Database Approach to Temporal Document Management

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

E-mail: `Kjetil.Norvag@idi.ntnu.no`

Abstract

Temporal document databases are interesting in a number of contexts, in general document databases as well as more specialized applications like temporal XML/Web warehouses. In order to efficiently manage temporal document versions, a temporal document database system should be employed. In this paper, we describe the V2 temporal document database system, which supports storage, retrieval, and querying of temporal documents. We also give some performance results from a mini-benchmark run on the V2 prototype.

1 Introduction

In order to efficiently manage temporal document versions, a temporal document database system should be employed. In this paper, we describe an approach to temporal document storage, which we have implemented in the V2 temporal document database system. Important topics include temporal document query processing, and control over what is temporal, how many versions, vacuuming etc., something that is necessary for practical use.

We have in a previous project studied the realization of a temporal XML database using a *stratum* approach, in which a layer converts temporal query language statements into conventional statements, executed by an underlying commercial object-relational database system. That project demonstrated the usefulness of a temporal XML databases in general, and gave us experience from actual use of such systems. The next step is using an *integrated* approach, in which the internal modules of a database management system are modified or extended to support time-varying data. This is the topic of this paper, which describes V2, a temporal document database system. In V2, previous versions of documents are kept, and it is possible to search in the historical (old) versions, retrieve documents that was valid at a certain time, query changes to documents, etc.

Although we believe temporal databases should be based on the integrated approach, we do not think using special-purpose temporal databases is the solution. Rather, we want the temporal features integrated into existing general database systems. In order to make this possible, the techniques used to support temporal features should be compatible with existing architectures. As a result, we put emphasis on techniques that can easily be integrated into existing architectures, preferably using existing index structures[1] as well as a query processing philosophy compatible with existing architectures.

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 describe an example application that originally motivated the work of this project. In Section 4 we give an overview of our approach and our assumptions. In Section 5 we describe the operations supported by V2. In Section 6 we describe the architecture for management of temporal documents used in V2. In Section 7 we describe the architecture and implementation of V2. In Section 8 we give some performance results. Finally, in Section 9, we conclude the paper and outlines issues for further work.

2 Related work

In order to realize an efficient temporal XML database system, several issues have to be solved, including efficient storage of versioned XML documents, efficient indexing of temporal XML documents, and temporal XML query processing. Storage of versioned documents is studied by Marian et al. [7] and Chien et al. [3, 4, 5]. Chien et al. also considered access to previous versions, but only snapshot retrievals. Temporal query processing is discussed in [8].

[1]History tells us that even though a large amount of "exotic" index structures have been proposed for various purposes, database companies are very reluctant to make their systems more complicated by incorporating these into their systems, and still mostly support the "traditional" structures, like B-trees, hash fi les, etc.

An approach that is orthogonal, but related to the work presented in this paper, is to introduce valid time features into XML documents, as presented by Grandi and Mandreoli [6].

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. The implementation is a stratum approach, built on top of Oracle 8.

3 Example application

In order to motivate the subsequent description of the V2 approach, we first describe an example application that motivated the initial work on V2: *a temporal XML/Web warehouse* (Web-DW). This was inspired by the work in the Xyleme project [12]: Xyleme supported monitoring of changes between a new retrieved version of a page, and the previous version of the page, but no support for actually maintaining and querying temporal documents.

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 regularly retrieving these pages and storing them in the warehouse, and at the same time keeping the old versions, we should be able to: 1) retrieve the sites/pages valid at a particular time t, 2) retrieve all pages that contained one or more particular word at a particular point in time t, and 3) ask for changes, for example retrieve all pages that did not contain "Bin Laden" before September 11. 2001, but contained these words afterwards.

It should be noted that a temporal Web-DW based on remote Web data poses a lot of new challenges, for example 1) consistency issues resulting from the fact that it is not possible to retrieve a whole site of related pages at one instant, and 2) versions missing due to the fact that pages might have been updated more than once between each time we retrieve them from the Web.

4 General overview and assumptions

In previous work on temporal database, including temporal XML documents [7, 8], it has been assumed that it is not feasible to store complete versions of all documents. The proposed answer to the problem has been to store delta documents (the changes between two documents) instead. However, in order to access an historical document version, a number of delta documents have to be read in order to reconstruct the historical versions. Even in the unlikely case that these delta versions are stored clustered, the reconstruction process can be expensive, in terms of disk accesses cost as well as CPU cost. As a result, temporal queries can be very expensive, and not very applicable in practice. We take another approach to the problem, based on the observation that during the last years storage capacity has increases at a high rate, and at the same time, storage cost has decreased at a comparable rate. Thus, it is now feasible to store the complete versions of the documents. Several aspects make this assumption reasonable:

- In many cases, the difference in size between a complete version and a delta version is not large enough to justify storage of delta versions instead of complete document version. For example, deltas are stored in a format that simplifies reconstruction and extracting change-oriented information, a typical delta can in fact be larger than a complete document version [7]. Even a simpler algorithm for creating deltas of document can generate relatively large deltas for typical document. The main reason for this, is that changes between document versions can be more complex than typical changes between fixed-size objects or tuples. For example, sections in a document can be moved, truncated, etc.

- Even though many documents on the web are very dynamic, and for example change once a day, it is also the case that in many application areas, documents are relatively static. When large changes occur, this is often during site reorganization, and new document names are employed.

Instead of storing delta documents, we will rely on other techniques to keep the storage requirements at a reasonable level: 1) compression, 2) granularity reduction [10], and vacuuming.

Although we argue strongly for not using the delta approach in general, we also realize that in some application areas, there will be a large number of versions of particular document with only small changes between them, and at the same time a small amount of queries that require reconstruction of a large number of versions. For this reason, we will in the next version of V2 also provide diff-based deltas as an option for these areas.

Document names and document version identifiers. A document is identified by its *document name*, i.e., every time a document with a given name is inserted, and there is already stored a document with the same name, the new document is considered a new version of the stored document.

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 a logical object identifier (OID) used in object databases.

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*.

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 (a tombstone is written to denote a logical delete operation) for each document.

5 Supported operations

In this section we summarize the most important user operations supported by V2 through the V2 API. A more detailed description of the operations can be found in [9].

Document insert, update, and delete. V2 is a general document database system, and in some of our application areas exact round-trip[2] of documents is required. As a result, we use a *FileBuffer* as the basic access structure, which is the intermediate place between the outside world (remote web page or local file), and the document version in the database. Thus, a document can be inserted into the FileBuffer from an external source, inserted into the version database from the FileBuffer, inserted into the FileBuffer from the version database, or written back to an external destination, for example a file.

Retrieving document versions. In order to retrieve a particular version into the FileBuffer from the version database, operations for retrieving the current version as well as the version valid at time t exist. These operations will be sufficient for many applications. However, in order to support query processing a number of operators will be needed. Included are operators for temporal text-containment queries (i.e., queries for document versions containing a particular word or set of words), the Allen operators [1] (i.e.: `before, after, meets, overlaps,` etc.) and operators for granularity reduction, vacuuming, compression, and deletion.

6 An architecture for management of temporal documents

In order to support the operations in the previous section, careful management of data and access structures is important. In this section, we present the architecture for management of temporal documents as implemented in V2.

As a starting point for the discussion, it is possible to store the document versions directly in a B-tree, using *document name* and *time* as the search key. Obviously, this solution has many shortcomings, for example, a query like "list the names of all documents stored in the database" or "list the names of all documents with a certain prefix stored in the database" will be very expensive. One step further, and the approach we base our system on, is to use 1) one tree-based index to do the mapping from name and time to VID, and 2) store the document versions themselves separately, using VID as the search key.

6.1 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. It is very useful to be able to query all documents with a certain prefix, for example `http://www.idi.ntnu.no/grupper/db/*`. In order to support such queries efficiently, the document name should be stored in an index structure supporting prefix queries, for example a tree-structured index.

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 and whether the document is temporal or not is stored (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*. This includes the timestamp and whether the actual version is compressed or not.

In order to keep the size of the document name index as small as possible, we do not store the size of the document version in the index, because this size can efficiently be determined by reading the document version's meta-chunk (a special header containing information about the document version) from the version database.[3]

6.1.1 Managing many versions

For some documents, the number of versions can be very high. In a query we often only want to query 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

[2]Exact round-trip means that a document retrieved from the database is exactly the same as it was when it was stored.

[3]We assume that queries for the size of a document are only moderately frequent. If this assumption should turn out to be wrong, the size can easily be included in the version descriptor in the document name index.

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 inserted into this new chunk.

The key for each chunk is the document name and *the smallest timestamp of an entry in* the next chunk *minus one time unit.* The reason for this can be explained as follows:

1. One version is valid from the time of its timestamp until (but not including) the time of the timestamp of the next version. Thus, a chunk covers the time from the timestamp of its first version descriptor until the timestamp of the first version descriptor in the next chunk.

2. In the B-tree library we base our system on, the data item (chunk) with the smallest key larger than or equal to the search key is returned.

The document metadata is replicated in each chunk in order to avoid having to read some other chunk in order to retrieve the metadata. In the current version of V2, the only relevant replicated metadata is the information on whether the document is temporal or not.

6.1.2 One vs. two indexes

When designing a temporal index structure, we have to start with one design decision, namely choosing between 1) one temporal index that indexes both current and historical versions, or 2) two indexes, where one index only indexes current or recent versions, and the other indexes historical versions.

The important advantage of using two indexes is higher locality on non-temporal index accesses. 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 or a two-index approach 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 one index that indexes only current document versions, and one index that only indexes historical versions is potential high update costs: when a temporal document is updated, both indexes have to be updated. This could be a bottleneck. To avoid this, we 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.

When keeping recent versions in the current version index, we trade off lookup efficiency with increased update efficiency. However, it should be noted that this should in most cases not affect the efficiency of access to non-temporal documents: We expect that defining documents as non-temporal or temporal in general will be done for collections of documents rather than individual documents. This will typically also be documents with a common prefix. These will only have a current version descriptor in the index, and the leaf nodes containing the descriptors will have the entries for many documents, and should in many cases achieve high locality in accesses.

6.1.3 Lookup operations on the document name index

In a temporal document database, a typical operation is to retrieve the current version of a particular document, or the version valid at a particular time. Using the architecture described in this section, this will involve a lookup in the document name index in order to find the VID of the document version, followed by the actual retrieval of the document version from the version database.

Retrieving the current version simply involves a search in the current version document name index. When retrieving the version valid at time t, the version descriptor can be in either the 1) historical or 2) current version index. In which index to start the search, depends on whether we expect that most such retrievals are to recent (current version index) or older (historical version index) versions. The best strategy also depends on the number of versions of most documents. In V2 it is possible for the user to give hints about this when requesting the search. When query processing is added to the system, this decision can be made by the query optimizer. One simple strategy that should work in many cases is simply to 1) maintain statistics for hit rates in the indexes versus the age of the requested document, and 2) use this statistics to start searching in the historical version index if the time searched for is less than current time minus a value t. Note that even if there is no entry for the document in the current version index, it is possible that it exists in the historical version index. The reason for this is that information about temporal documents that have been (logically) deleted is moved to the historical version index even if the chunk is not full (because we want the current version index only to contain information about non-deleted documents).

6.2 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 most significant part of the version key is the VID. 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 is makes it easy to retrieve all versions inserted during a certain VID interval (which can be mapped from a time interval). One interesting use 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 some situations, for example during execution of some queries, we get VIDs as results. In this way, we are able to retrieve the actual document versions. However, often information about the documents is requested at the same time, for example the document name. For this reason, some metadata is stored in a separate header, or *meta-chunk*. In this way, it is easy to do the reverse mapping from VID to document name. Currently we also store the timestamp in the meta-chunk, because we decide the timestamp at an early stage anyway. However, if we later should use another approach for timestamp management, this can be changed. The meta-chunk also contains the size of the document version.

As described previously, V2 has a document-name index that is divided in a current (or rather *recent*) and historical part. This approach, as has been proposed in the context of traditional temporal databases, could also be used for the version database. However, for several reasons we do not think this is appropriate:

- First of all, considering the typical document size which is much larger than tuples/objects in traditional temporal databases, the locality aspect is less important.

- Second, it would involve more work during update, because we would not only write a new version, but also read and rewrite the previous version (move from current to historical version database). It is also possible to achieve the same in a more flexible and less costly way by actually creating two separate databases, and regularly move old versions to the historical database, for example as a result of vacuuming or granularity reduction processing.

6.2.1 Non-temporal documents

For some documents, we do not want to store their history. When such a *non-temporal* document is updated, the previous version of the document becomes invalid. However, instead of updating the document in-place, we append the new version to the end of the version database. The previous version can be immediately removed from the version database, but a more efficient approach is to regularly sweep through the version database and physically delete old versions, compacting pages (moving contents from two or more almost-empty database pages into a new page), and at the same time vacuum old temporal documents.

There is also another reason for doing updates this way: documents do not have a fixed size, and quite often new versions will be larger than the previous versions. In that case, in-place updating would often result in 1) splitting of pages, 2) writing to overflow pages, or 3) wasted space if space is reserved for larger future versions. All these three approaches are expensive and should be avoided.

6.3 Full-text index

A text-index module based on variants of inverted lists is used in order to efficiently support text-containment queries.

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.

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-index module actually provides three different text-index variants suitable for being implemented inside ordinary B-trees. Each variant have different update cost, query cost, and disk size characteristics:

Naive text-index: This index uses one index record for every posting, i.e., a $(word,VID)$ tuple in the index for each document version containing *word* (although the word is in practice only stored once in the index). The advantage of this index structure is easy implementation, and easy insertion and deletion of postings. The disadvantage is of course the size: in our experiments the disk space of naive text-indexes was close to the size of the indexed text itself.

Chunk-based text index: This index uses one or more chunks for each 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 size (typically in the order of 0.5-2 KB, but should always fit in one index page). 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). The size of this text-index variant will be much lower than the previous variant, because the number of records in the index will be much low, meaning less overhead information. The disadvantage is higher CPU cost because more data has to be copied in memory for each entry added (this is the reason for the chunk size/insert cost tradeoff, giving chunk sizes relatively smaller than maximum size constrained by the index page size). However, the text-index size is much lower than the previous approach. In order to support zig-zag joins, each chunk uses the VID in addition to the index words as the chunk key.

Improved chunk-based text index: Traditionally the size of text-indexes is reduced by using some kind of compression. The 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 can be replaced by the distance $d = d_i - d_{i-1}$. This distance usually requires a lower number of bits for its representation. Given the size of the moderate size of our chunks and the desire to keep complexity and CPU cost down, we use a simpler approach, where 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. It should be noted that this will in practice happen infrequently. When using the improved chunk-based text index, we have in our experiments experienced a typical text-index size of less than 7% of the indexed text. This size can be further reduced if the text-index is compacted (the typical fill-factor of the index in the dynamic case is 67%, but this can be increased to close to 100% with reorganization). This can be useful if the document database is static most of the time, and dynamic only in periods.

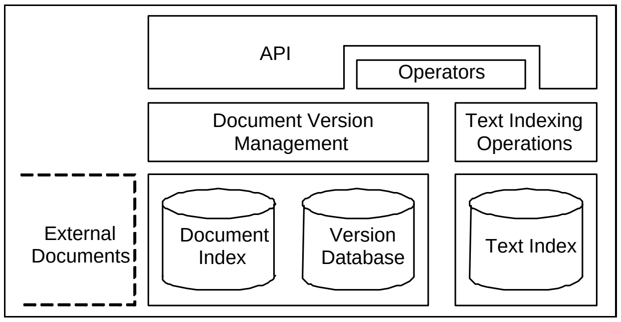


Figure 1. The V2 prototype architecture.

7 Implementation of the V2 prototype

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 in this paper.

In the current prototype, the bottom layers are built upon the Berkeley DB database toolkit [11], which we employ to provide persistent storage using B-trees. However, we will later consider to use a XML-optimized/native XML storage engine instead. An XML extension to Berkeley DB will be released later this year, and would be our preferred choice sine this will reduce the transition cost. Other alternatives include commercial products, for example Tamino, eXcelon, or Natix. Using a native XML storage should result in much better performance for many kinds of queries, in particular those only accessing subelements of documents, and also facilitate our extension for temporal XML operators. The main parts of the architecture of V2 are illustrated in Figure 1.

8 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 as well as comparing the system with similar systems. However, to our knowledge there exists 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 come as no surprise, considering that this is relatively new research area). In order to do some measurements of our system, we have created a execution trace, based on the temporal web warehouse as described in Section 3.

8.1 Acquisition of test data and execution trace

In order to get some 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 the site starting with the site's main page. This essentially provides an insert/update/delete trace for our temporal document database.

The initial set of pages was of a size of approximately 91 MB (approximately10000 web pages). An average of 510 web pages were updated each day, 320 web pages were removed (all pages that were successfully retrieved on day d_i but not available at day d_{i+1} were considered deleted), and 335 web new pages were inserted. It should be noted that the update/insert rate is relatively high because many of the web sites were web-newspapers/magazines that were updated daily. The average size of the updated pages was relatively high (37.5 KB), resulting in an average increase of 45 MB for each set of pages loaded into the database (with 90% fill factor, this equals 40 MB of new text into the database).

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

For our experiments, we used a computer with a 1.4 GHz AMD Athlon CPU, 1 GB RAM, and 3 Seagate Cheetah 36es 18.4 GB disks. One disk was used for program/OS, one for storing database files, and one for storing the test data files (the web pages). The version database and the text index has separate buffers, and the size of these are explicitly set to 100 MB and 200 MB, respectively. The rest of the memory is utilized by the operating system, mostly as disk page buffers. The database page size is set to 8 KB.

8.2 Measurements

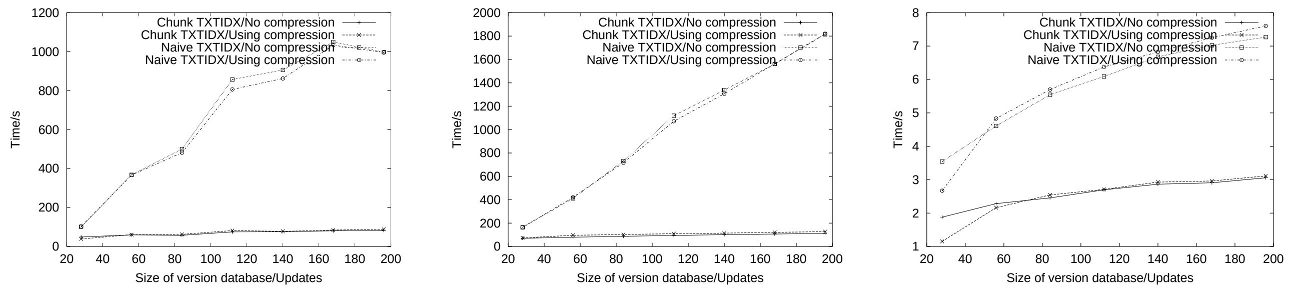
We now describe the various measurements we have done. All have been performed both using the naive and chunk-based text indexes, and both with and without compression of versions. In order to see how different choices for system parameter values like chunk sizes and cache sizes affects performance, we have also run the tests with different document/version/text-index chunk sizes and different cache size for the version database and the text index.

Loading and updating. The first part of the tests is loading data into the system. Loading data into a document database is a heavy operation, mainly because of the text indexing. Our text indexes are dynamic, i.e., are updated immediately. This implies that frequent commit operations will result in a very low total update rate. However, for our intended application areas we expect much data to be loaded in bulk in each transaction. For example, for the web warehouse application we assume commit is only done between each loaded site, or even set of sites. We load all the updates for one day of data in one transaction. In the first transaction, the database is empty, so that approximately 10000 pages are inserted into the system. For each of the following transactions, on average of 510 web pages/documents are inserted, 320 documents logically deleted, and 335 documents inserted, as described above. Note that in order to find out whether a web page has changed or not, the new page with the same name has to be compared to the existing version. Thus, even if only 510+335 document versions are actually inserted, approximately 10000 documents in total actually have to be retrieved from the database and compared with the new document with the same URL during each transaction. For each parameter set, we measure the time of a number of operations. The most important measurements which will discuss below is:

- The update time for the last transaction, when the last set of documents are applied to the system.
- After the initial updates based on test data, we also insert a total of 10000 new pages with documents 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 the previous version is necessary).
- In order to study the cost of inserts of individual documents, when only one document is inserted during one transactions, we also insert a number of documents at different sizes from 800 B to 30 KB, using a separate transaction for each. As a measure of this cost, we use the average time of the insert of these documents. Because we have not enabled logging, every insert result in every disk page changed during the transaction to be forced to disk (this is obviously a relatively costly operation).

Query and retrieval: After loading the database, we do some simple test to measure the basic query performance. The operations are text lookup of a set of words, and with some additional time operations as described below. When searching, we have used three categories of words:

- Frequently occurring words, which typically occurs in more than 10% of the documents. In our database, all entries for one frequently occurring word typically occupies in the order of 10 disk pages.
- Medium frequently occurring words, which occurs in approximately 0.5% of the documents). In our



(a) Update set of web pages.　(b) Insert new set of web pages.　(c) Insert single document.

Figure 2. Load and update performance. The size is given as number of update transactions, each typically increasing the database size with 40-50 MB.

database, all entries for a medium frequently occurring word fit in one disk page (but are not necessarily stored in one disk page, because the chunks can be in two different pages).

For each query we used different words, and for each query type we used several of the words and use the average query time as the result. In practice, 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 presented in this paper were:

- AllVer: All 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 $t = t_{Mid}$.

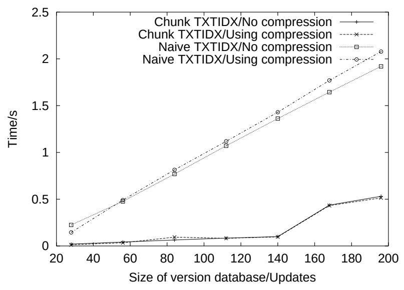
For all text-containment queries involving time, the meta-chunk of the actual versions have to be retrieved when we have no additional time indexes.

8.3 Results

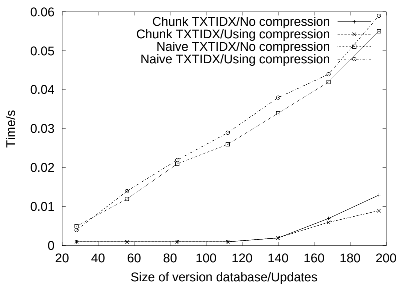
We now summarize some of the measurement results. The size given on the graph is the number of update transactions. As described, the first one loads 10000 documents, giving a total database size of 91 MB, and each of the following increase the size with between 40 and 50 MB. The final size of the version database is 9.7 GB (1.9 GB when compression is enabled). Based on measurements with different chunk size, we found that a chunk size of 400 B was a suitable choice for both the version database and the text index (a tradeoff between overhead and CPU/memory-bandwidth usage).

8.3.1 Load and update cost

The initial loading of the document 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 56 s without compression, and 71 s with compression enabled. 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 document has changed. If it has not changed, it is not inserted. Figure 2a shows this cost for different database sizes. The size is given as number of update transactions. After the initial load, 91 MB of text is stored in the database, and each transaction increases the amount with approximately 45 MB of text, up to a total of 9.7 GB. The cost of updating/inserting a set of web documents increases with increasing database size because of a decreasing buffer hit rate for disk pages containing the previous document version and pages where postings have to be inserted. From the graph we see that the increase in cost using the improved chunk-based text index is much lower than when using the naive text index. One of the main reasons is that more disk pages have to be written back in the case of the naive text index. Also note the last point (at 196) on the graph for naive text index/no compression in Figure 2a. The cost at this point is lower than the cost at the previous point (at 168). This might seem like an error, but the reason is actually that the documents loaded at this point resulted in less updated documents than at the previ-



(a) Frequent words.



(b) Medium frequent words.

Figure 3. Text containment, all versions.

ous point.

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 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.

As can be seen from the graphs in Figure 2, the use of compression only marginally improves performance, and in some cases also reduces the performance in the case of insert/update. However, the retrieving the actual documents, and in particular large documents, the cost will be significant. It is also likely that when using larger databases than the one used in this study, but with the same amount of main memory, the gain from using compression will increase because of the increased hit rate (when using compression, the database will in total occupy a smaller number of pages, thus increasing the hit ratio). However, the main advantage of using compression is the fact that it reduces the size of the version database down to 20% of the original size. This is important: even though disk is cheap, a reduction of this order can mean the difference between a project that is feasible and one that is not.

8.3.2 Query cost

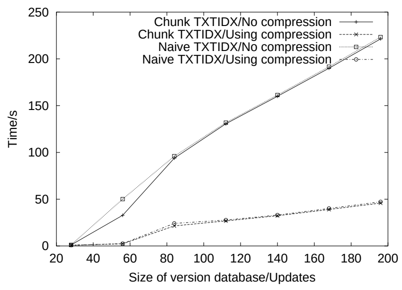
Figure 3 illustrates the cost of retrieving the VIDs of all document version containing a particular word. As expected, the costs increases with increasing database size. The main reason for the cost increase with smaller database sizes is a higher number of versions containing the actual word, resulting 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 after 140 update transactions, which equals a database size of 6.8 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. For more details on other kind of queries, as well as disk space usage, we refer to [9].

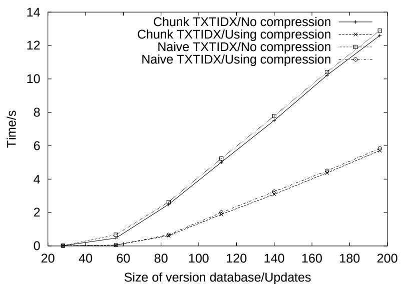
9 Conclusions and further work

We have in this paper described the V2 temporal document database system, which supports storage, retrieval, and querying of temporal documents. We have described functionality and operations/operators to be supported by such systems, and more specifically we described the architecture for management of temporal documents used in the V2 prototype. We also provided a basis for query processing in temporal document databases, including some additional query operators that are useful in a temporal document database. All of what has been described previously in this paper is implemented and supported by the current prototype. This is also what we believe to be one of the most important contributions of this paper; to actually integrate existing aspects of various areas in temporal database management into a working system, capable of managing temporal documents.

We have studied the performance of V2, using a benchmark based on a real-world temporal document collection. The temporal document collection is created by regularly retrieving the contents of a selected set of Web sites. We



(a) Frequent words.



(b) Medium frequent words.

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

have studied document load/update time, as well as query performance using the operators described previously. As we expected, the performance results indicate good performance in the case of large transactions (essentially bulk-loading of data), where an amount of 155 text files/1.7 MB of text is indexed per second.

One of the main reasons for developing this prototype was to identify performance bottlenecks in temporal document databases, as well as have a toolbox to work with in our ongoing work on temporal XML databases. Future work include a temporal browser, which should make it possible in a user-friendly way to ask for a page valid at a particular time, and in the case of Web documents, automatically retrieve and display the page valid at that time when following a link. Such a browser could also make it easier to see changes between versions valid at particular times.

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] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), 1983.

[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. 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.

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

[5] 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 scientific applications (RIDE-DM'2001)*, 2001.

[6] F. Grandi and F. Mandreoli. The valid web: An XML/XSL infrastructure for temporal management of web documents. In *Proceedings of Advances in Information Systems, First International Conference, ADVIS 2000*, 2000.

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

[8] 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.

[9] K. Nørvåg. The design, implementation and performance evaluation of the V2 temporal document database system. Technical Report IDI 10/2002, Norwegian University of Science and Technology, 2002. Available from http://www.idi.ntnu.no/grupper/DB-grp/.

[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.