Norwegian University of Science and Technology
Technical report IDI-TR-10/2002

Design, Implementation, and Performance of the V2 Temporal Document Database System

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

The advent of large amounts of data on the web has closed the gap between the document storage and the database communities. In this paper, this work is continued by the description of the foundations for temporal document databases. We describe functionality and operations/operators to be supported by such systems, and more specifically we describe the architecture for management of temporal documents used in the prototype of 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.

Keywords: Temporal databases, document databases, XML, query processing

1 Introduction

One of the advantages of XML is that the document itself contains information that is normally associated with a schema. This makes it possible to do more precise queries, compared to what has been previously possible with unstructured data. It also has advantages for long-term storage of data: even though the schema has changed, the data itself can contain sufficient information about the contents, so that meaningful queries can be applied to the data. This is of increasing importance as storage costs are rapidly decreasing and it feasible to store larger amounts of data in databases, including previous versions of data. If documents as well as DTDs/schemas are stored in a suitable temporal database system, it is also possible to make queries based on the actual DTD/schema that was valid at the time a particular document was stored. In order to efficiently manage the temporal 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 in order to be able to control the storage requirements.

We have previously in the TeXOR 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 [21]. We could in this way rapidly make a prototype that supported storage and querying of temporal XML documents. However, we consider a stratum approach only as a short-term solution. As a result of using a stratum approach, some problems and bottlenecks are inevitable. For example, no efficient time index for our purpose were available, and query optimization can be a problem when part of the "knowledge" is outside the database system. Retrieving data from the database and process it in the TeXOR middleware would not be a good idea if we want to benefit from the XML query features supported by the object-relational database management system.

The TeXOR 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 data 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

Until the advent of XML, document storage and database management were to a large extent two separate research areas. A good overview of document data models is given in [15].

The recent years have brought many papers on query on structured and semistructured documents (for example [1]), in particular in the context of XML documents (for example [2]), and several systems for managing and querying such documents have been implemented, for example Lore [23] and Xyleme [27].

A model for representing changes in semistructured data (DOEM) and a language for querying changes (Chorel) was presented by Chawathe et al. in [5, 6]. Chorel queries were translated to Lorel (a language for querying semistructured data), and can therefore be viewed as a stratum approach.

[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 files, etc.

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. [14] and Chien et al. [7, 9, 8]. Chien et al. also considered access to previous versions, but only snapshot retrievals. Temporal query processing is discussed in [18, 19].

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 [12].

Also related to our work is work on temporal object database systems. In the area of object databases (ODBs) with integrated support for temporal objects, we are only aware of one prototype: POST/C++ [26]. In addition, some temporal ODBs built on top of non-temporal ODBs (i.e., stratum approach) exists. One example is TOM, built on top of O_2 [24]. The architecture of the object-relational database system POSTGRES [25] also made it possible to store and retrieve historical versions of objects/tuples.

Another approach to temporal document databases is the work by Aramburu et al. [4]. 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 [27]: 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:

- Retrieve the sites as they were at a particular point in time.
- Retrieve pages valid at a particular time t.
- Retrieve all versions of pages that contained one or more particular word at a particular point in time t.
- 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. A more thorough discussion of these aspects can be found in [20].

4 General overview and assumptions

In previous work on temporal database, including temporal XML documents [14, 18], 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 [14]. 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:

- Compression of documents. Typical compression rates using low-cost compression algorithms on simple text is in the order of 25-50% of the uncompressed size (depending of the document size, typical examples from our experiments are 50% for small documents less than 1 KB, improving to approx. 20% for larger documents of 3 KB and more). Compression of raw XML will give even better compression due to the redundancy of tag names. However, XML documents are typically stored in a more optimized form in a database, using integer identifiers instead of text tags. Thus, the compression potential will be somewhat lower than for raw XML.

- When we have a large number of document versions created during relatively a short time period, it is possible that after a while we do not really need all these versions. For example, in most cases we will probably not ask for the contents of a document as it was *at a particular time of the day* half a year ago, we will ask for the contents for *a particular day*. Thus, after a while it is possible to reduce the granularity of the document versions that are stored, without reducing the usefulness of the database. It is likely that if a document has a high number of updates during a short time period, these changes are not large, and in many cases also individually insignificant. It should be noted that reducing the granularity is not always appropriate. For example, in the context of newspapers on the web, reducing the granularity from 1 day to 1 week means we keep only every 7th version, and the problem here is that the intermediate versions are just as important as the ones left in the system.

- Another strategy for reducing the number of version stored, is the traditional vacuuming process, where all historical document versions created before a certain time t are physically deleted and can not be accessed again (in contrast to logical deletion, which simply creates a tombstone version without removing previous versions).

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.

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

This makes it also possible to use the VID as contents in another document. This can be useful in a number of cases. In the extreme, it can also be used to actually employ V2 as a temporal object database, storing objects, with references to other objects, as documents. If text indexing is disabled (this can be done dynamically), this could make V2 just as efficient as a purpose-built temporal object database.

4.2 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*.

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. 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 reappear later). This contrasts to most object database systems, where an object identifier can not be reused. In a document database system, it is also possible that a document is deleted by a mistake, and with temporal support it can be brought to life again by retrieving a historical version and rewriting this as a new current version.

Periods (the time from one particular point in time to another, for example January 1st to June 30th) as well as intervals (duration, for example “one month”) are also supported. A version is essentially valid in the period from the value of its timestamp, until the timestamp of the next version. If the version is the current version, it is valid from the value of its timestamp *until changed* (U.C.). U.C. is a special value in the data model (and implementation).

In valid-time temporal databases, it is useful to be able to support different granularities in timestamps. In a transaction-time temporal database this is less interesting, so that in V2 the same granularity is employed for the whole timeline (i.e., from database creation time until *Now*).

4.3 Timestamp management

Traditionally, the commit time has been used as the timestamp in transaction-time temporal databases. However, in some situations, the timestamp of a transaction has to be decided *before* commit time. One situation where this is the case, is when the query in a transaction needs the timestamp of itself in a query. How to manage this problem, is described by Jensen and Lomet in [13].

Another situation, which is the one we expect to be most important for our context, is the write timestamp. In general, we want to have the timestamp stored together with the data. Again, if the timestamp is decided at commit time, this is not a problem, and the problem of storing the time information together with the data and at the same time avoiding early decision of the timestamp can for example be solved by 1) keeping an *intention list* with all the updates, and apply these at commit time, or 2) writing the transaction identifier instead of timestamp during the transaction, and do the update later (in this way the problem with large intentions lists is avoided, and it can also be performed in a lazy way, cf., e.g., [17]).

Traditionally, transactions are assumed to be relatively short in duration, and mostly create a moderate amount of data *and* index entries. The solutions above can be satisfactory for the data itself (the documents) in our context, however, they are less appropriate in the case of index entries which includes the timestamp: 1) in the case of a document database there will a large number of index entries created for each document, and 2) if this is an index where one of the indexing attributes is the timestamp (for example a temporal text index) the transaction identifier is not sufficient.

Timestamps will in our system only be used for retrieval/querying purposes, and not, e.g., concurrency control. For this reason, we do not consider it critical *when* the timestamp is decided, as long as all document versions stored in one transaction are assigned the same timestamp. On the other hand, the amount of index entries that can be created for each document makes the traditional approaches unsuitable. Thus, we have to decide the transaction timestamp when the first document is stored in a transaction.

The problem and consequences of early timestamp decision can be illustrated by the following example: Assume that two transactions T_1 starting at time $t = 1$, and transaction T_2 starting at time $t = 2$. Further assume that transaction T_2 reads and updates a document D, and then commits. The problem now is if T_1 then reads and updates the same document D, the new version is now assigned a lower timestamp than the previous version. This is clearly an anomaly that can not be accepted. It should be noted that the situation can not be solved by locks alone. Even if T_2 in this example locked the document D, the problem still remains if T_2's locks are released when T_2 commits.

Assuming that concurrency control is still based on locking, the problem can partly be solved if no documents with a lower timestamp than that of T_2 are allowed to update the document after T_2 has released its lock. This can also be combined with a pessimistic approach where a *write set* has to be declared when a transaction starts. As is obvious, we are now moving into the domain of timestamp-based concurrency control, and the unfortunate side effect of possibly more transaction aborts than when using locking protocols. However, in practice this should be less of a problem. Concurrent document management can be managed by using check-out/check-in protocols, resulting in very short update transactions.

4.4 Metadata

In a document database system, there are metadata on several levels. There is the traditional "database" metadata, describing characteristics of the database itself and storage details, and there is *document metadata*, for example *author*, *topic*, etc. Document metadata should normally be under control of the user, and we assume it is stored separately, for example in a separate document. By using persistent VIDs, it is also possible to associate versions directly with metadata. Associated metadata can also be DTDs and schemas, themselves stored as documents. This aspect facilitates temporal DTDs and schemas.

5 Supported operations

In this section we summarize the most important user operations supported by V2 through the V2 API. In additions, operations exists for the most important database management tasks, for example creating and removing databases, as well as transaction management operations like begin, commit and abort transaction.

The current prototype is a one-user version, and transactional support is only partial; logging and recovery have not yet been enabled (although this can easily be incorporated due to the fact that V2 uses Berkeley DB for the low-level database operations). However, it is still required that the begin/commit transaction operations are used in order to set the correct timestamp on the data. The commit transaction operation also ensures that the updated data and index entries are safe on disk after the commit operation has completed.

5.1 Document insert, update, and delete

When working with XML documents, DOM trees are often used in order to access and manipulate the documents while in main memory. In our application area, where exact round-trip[2] of documents is required we use another approach. As the basic access structure, we use the *FileBuffer*, which is the intermediate place between the outside world (remote web page or local file), and the version in the database. Thus, a document can be:

- Inserted into the FileBuffer from an external source, for example file, user input, or a remote Web source. It is possible to specify that a document should only be stored if it has changed from the previous version.
- Inserted into the version database from the FileBuffer. When a new document is stored, i.e., one that does not have a previous version in the database, the system is informed by the user/application whether this is 1) a document where old versions should be stored after updates, i.e., a temporal document, or 2) a document where only the last version should be stored, i.e., a non-temporal document.
- Inserted into the FileBuffer from the version database.
- Written back to an external destination, for example a file.

If DOM-like operations should be needed, a DOM-tree can be created from the contents of the FileBuffer.

In order to reduce the storage requirements, documents can be compressed. A database can contain both compressed and uncompressed documents. Whether to use compression or not can be changed any time, so that different versions of the same document can have different compression characteristics. Whether a document is compressed or not, is in general transparent to the user: when the FileBuffer is accessed, the view is always the original uncompressed document. Use of compression

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

can significantly increase the time needed to store/retrieve documents, so that whether to employ compression or not is clearly a space/time tradeoff. It should be noted that although it at first sight might seems obvious that documents that are infrequently accessed should be compressed, while frequently accessed documents should be stored uncompressed, this is not necessarily the case. For example, compressing documents to half the size can result in twice as many documents in the page buffers, thus increasing buffer hit ratio.

A delete in a temporal database system is different from a non-temporal databases system: data is not physically deleted, but information is recorded about the time the logical deletion occurred, this timestamp is stored in a *tombstone*. This is also the case when deleting a document in V2: the stored historical versions of the deleted document are still available for retrieval after the logical deletion. On the other hand, if a document at creation time was declared non-temporal, it is physically deleted as the result of the delete operation, and related data in the text index is also removed.

5.2 Retrieving document versions

In order to retrieve a particular version into the FileBuffer from the version database, one of the following operations can be used:

- Retrieve current version.
- Retrieve the version valid at time t.

These operations will be sufficient for many applications. However, in order to support query processing a number of operators will be needed. They will be described in more detail in Section 5.5.

5.3 Space-reducing operations

We employ three types of space-reducing techniques in V2:

- Traditional vacuuming: Delete all non-current versions created before a certain time t.
- Granularity reduction: Consider all non-current versions created before a certain time t, and delete all versions of an document that are closer than time t_t to the next (more recent) version that is kept. The versions to be removed are decided by a simple algorithm were we start with the most recent version D_i, which is to be kept. If the difference of the timestamps of version D_i and the previous version D_{i-1} is less than t_t, then version D_{i-1} is to be removed. If the difference is larger than t_t, D_{i-1} is kept. Next, the difference of timestamps of version D_{i-2} and the next version that is kept (D_{i-1} or D_i in this case) is calculated, and if smaller than t_t, version D_{i-2} is removed. In the final result, there are no versions left that are closer in time to each other than t_t.
- Compression: Compress all non-current uncompressed versions created before a certain time t.

5.4 Importing and exporting data

Sometimes a large amount of documents are to be retrieved from files and stored into the database. This can conveniently be achieved by using the import function provided by V2, which imports all documents stored in a certain directory (this is applied recursively, so that files under subdirectories are also imported). V2 also supports exporting all documents stored in the database to files.

5.5 Query operators

Items in temporal and document databases have associated metadata that can be used during query processing to filter out a subset of items for subsequent query processing (this metadata would normally be stored as ordinary data in other databases). The result of execution of the operators can either be used in subsequent "traditional" query processing employing traditional operators like selection, projection etc., or the VIDs can be used to retrieve the actual document versions from the version database.

In order to support metadata-based query processing, a number of operators will be needed. We now describe some of the most important operators currently supported by V2, together with the data type of input and output of the operators. In the following discussion we use the following notation for operators and the data types of the input and output of the operators:

- *docname* is a document name/file name.
- *period* is a time period.
- *vid* is a version identifier.
- t_s is a timestamp.
- (x,y) denotes a tuple containing one element of type x and one element of type y.
- $\{x\}$ denotes a set of elements of type x.
- `op`: $x \rightarrow y$ denotes an operator `op` with input x and output y.

and the operators are as follows:

- `contain_words`: $\{word\} \rightarrow \{vid\}$
  Returns the document identifiers of all document versions containing all the words in the set $\{word\}$.
- `time_select`: $(\{vid\}, time) \rightarrow \{(vid, period)\}$
  `time_select`: $(\{vid\}, period) \rightarrow \{(vid, period)\}$
  Returns the document identifiers of all document versions in the input set that were valid at a particular time or during a particular period, together with the validity period of the versions. If the time has the value *Now*, all document versions that are currently valid are returned (this is in general a cheaper operation than a general `time_select` operation).
- `VIDinfoT`: $\{vid\} \rightarrow \{(vid, t_s, docname)\}$
  Returns the document names of the particular document version identified by their version identifier, together with the timestamp of the version.
- `VIDinfoP`: $\{vid\} \rightarrow \{(vid, period, docname)\}$
  Returns the document names of the particular document version identified by their version identifier, together with the period of validity of the version. It should be noted that although this operator is close in functionality to the `VIDinfoT` operator, it is more costly to evaluate.
- `cur_docnames`: $(prefix) \rightarrow \{docname\}$
  Returns the name of all current (non-deleted) documents with name starting with *prefix*.

- `all_docnames`: $(prefix) \rightarrow \{docname\}$
  Returns the name of all documents with name starting with *prefix*, including the name of deleted documents.

- `scancurrent`: $(prefix) \rightarrow \{(docname, vid, period)\}$
  Returns name, VID and validity period (effectively timestamp and U.C.) for all current version (i.e., all non-deleted documents) with prefix *prefix*,.

- `scanall`: $(prefix) \rightarrow \{(docname, vid, period)\}$
  Returns name, VID and validity period for all documents (including deleted documents) with name starting with *prefix*.

- `docversions`:
  $(\{docname\}) \rightarrow \{(docname, vid, period)\}$
  `docversions`:
  $(docname, period) \rightarrow \{(docname, vid, period)\}$
  Returns the history of a document, alternatively only the versions of a document valid during a particular time period.

Although types are denoted as sets above, in the implementation they are actually "ordered sets", suitable for `next/previous` functions as well as more general iterators.

V2 also supports the Allen operators [3], i.e.: `before, after, meets, met_by, overlaps, overlapped_by, during, contains, starts, started_by, finishes, finished_by,` and `equal`. These operators take as input a set of VIDs and their validity periods, together with a time period. The output is the VIDs for which the condition is true:

- `AllenOp`: $(\{(vid, period)\}, period) \rightarrow \{(vid, period)\}$

In addition to query operators, there are also operators for granularity reduction, vacuuming and deletion. This includes the following operators:

- `delete_versions`: $\{vid\} \rightarrow \emptyset$
  Delete all document versions with VID as in the input set.

- `granularity`: $(\{docname\}, t_s) \rightarrow \{vid\}$
  For each document in the input set, returns the VIDs of versions of the document that are closer in time than t_s to the . The resulting set of VIDs can for example be used as input for the `delete_versions` operator.

In order to support "programmatic" access, rather than queries on collections, we also provide operators that can be considered syntactic sugar, and which functionality could be provided by application of the operators above. However, for efficiency reasons they are implemented directly. Examples include direct functions for retrieving the current version, retrieving the VID of the current version of a document, etc.

It should be noted that the set of operators is not a minimal set. For example, the result of `scanall` can also be generated by a combination of `all_docnames` and `docversions`. However, in several cases the execution can be more efficient for the "combined" operator, instead of using a combination of two or more other operators.

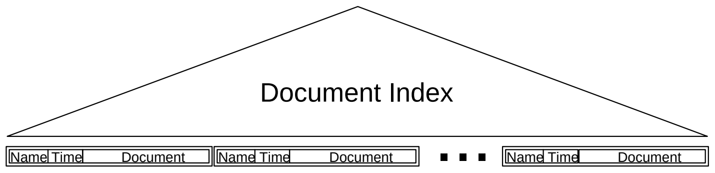


Figure 1. Document, document name, and time are stored in one structure (a rather inefficient solution!).

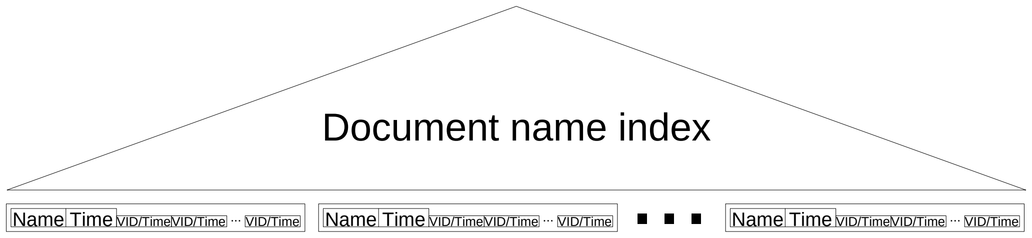

Document name index
Name Time VID/Time VID/Time ··· VID/Time
Name Time VID/Time VID/Time ··· VID/Time
Name Time VID/Time VID/Time ··· VID/Time

Figure 2. Document name index. Each record consists of metadata common to all versions of a document, followed by metadata specific to the particular versions.

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 (see Figure 1). 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 (see Figure 2). For each document, the following metadata is currently stored:

- Document name.
- Whether the document is temporal or not, 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*:

- Timestamp.
- 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 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 (it should be noted that in order to benefit from prefix compression in internal nodes, it is important that the document name comes first in the key) 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.

[3]We assume that queries for the size of an object 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.

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 [16] 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 object 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

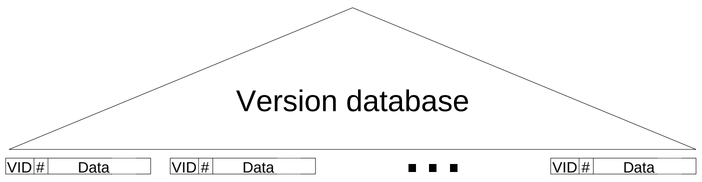


Figure 3. Version database. Each record consists of a VID, a chunk number, and a part (chunk) of the document itself.

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 (see Figure 3).

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 (as discussed in Section 4.3). 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.

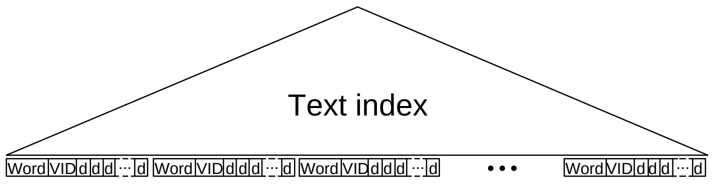


Figure 4. Improved chunk-based full-text index. Each chunk contains the indexed word, the VID of the first document version, and a number of distances encoding the VIDs.

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, i.e., queries for document versions containing a particular word (or set of words).

In our context, we consider it necessary to support dynamic updates of the full-text index, so that all updates from a transaction are persistent as well as immediately available. This contrasts to many other systems that base the text indexing on bulk updates at regular intervals, in order to keep the average update cost lower. In cases where the additional cost incurred by the dynamic updates is not acceptable, it is possible to disable text-indexing and re-enable it at a later time. When re-enabled, all documents stored or updated since text indexing was disabled will be text-indexed. The total cost of the bulk-updating of the text-index will in general be cheaper than sum of the cost of the individual updates.

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 $\langle$*word,VID*$\rangle$ 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. Two examples of codings that can be used for this purpose are Elias encoding [10] and variable length encoding [11]. 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, as illustrated in Figure 4, 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.

In order to support temporal text-containment queries better, we work with implement a temporal text index into V2, but currently temporal text-containment queries have to be performed by an ordinary text-containment query, followed by temporal selection of the matching VIDs that represent the document versions.

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 [22], 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 probably 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. With the modular design of the V2, the adaption should be relatively easy. Using a native XML storage

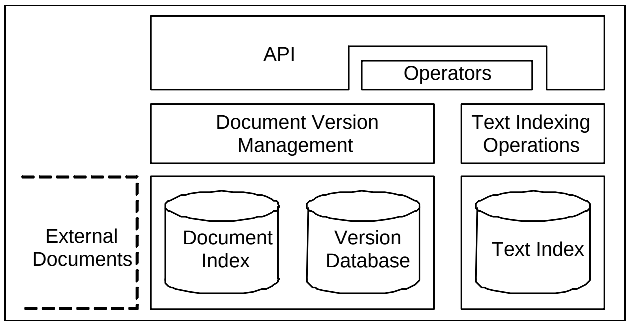


Figure 5. The V2 prototype architecture.

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 architecture of V2 is illustrated in Figure 5, and the main parts are:

1. Version database, as described in Section 6.2.
2. Document name index, as described in Section 6.1.
3. Document version management, employing the document name index and version database.
4. An optional text index (optional in the sense that it can be disabled, it is also possible to disable it and re-enable it at a later time, this can be useful during bulk loading), as described in Section 6.3. The text-index module also supports functions like stop-word management.
5. API layer.
6. Operator layer, supporting the operators described in Section 5.5.

The system also provides functions for compacting the version database and text index, in order to increase the fill factor of the disk pages where documents and text indexes are stored. This is particularly useful when a dynamic period with many updates and deletes is followed by a relatively long static period.

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.

In general, many web sites have relatively static pages, and the main cost is the first loading of pages. The performance for storing this kind of sites can be considered comparable to loading the first set of pages as described above, and the maintenance cost for such sites is only marginal (although the cost of determining whether a page has changed or not incur a certain 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 expect to be relatively dynamic. This included local pages for the university courses, and a number of online versions of major newspapers (both in the Norwegian and English languages).

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.

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.

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. This time will increase with increasing database size, because we in the beginning have more buffer space than the size of the database. This makes operations like retrieval of previous version for comparison purposes cheap. As the database increases and we can not store everything in main memory any more, the update time increases.
- 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 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).

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 lack of space and the fact that these results does no give any 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 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 used were:

- AllVer: All document versions 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 insert 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 $t = t_{Mid}$ time and the current time $t = t_{Now}$.

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 load 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).

8.4 A study of optimal chunk size values

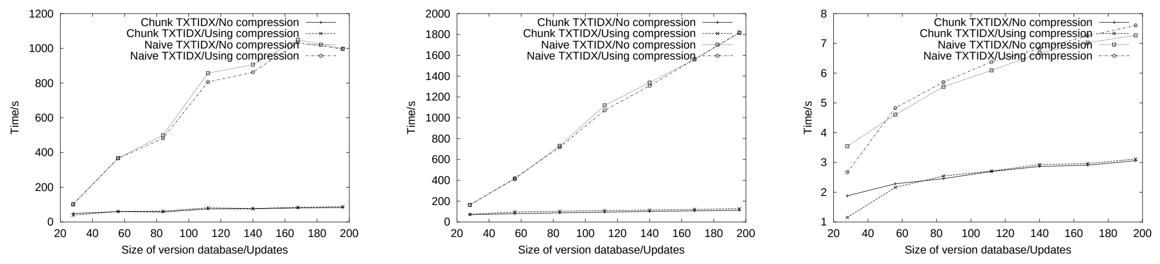
V2 has some tunable parameters where a bad choice of values could affect the performance. In order to see how sensitive the system is to the value of these parameters, we have studied the performance with different values for the chunk size in the version database and the text index.

Chunk size in the version database: It was expected that a low value would negatively affect performance because this increases the number of chunks, and there is some overhead for every chunk. On the other hand, a too large chunk size would also have a negative effect, because when there is not space for a chunk on a page, the system starts on a new page, and the free space on the previous page will remain unused. We studied the performance with different chunk sizes in the range from 200 to 2000 bytes. It showed that for all chunks sizes in the range 200-500 bytes the performance was comparable and suitable choices.

Chunk size on the text index: Similar to the chunks in the version database, a low chunk size in the text indexes increases the average overhead for each entry, but it also reduces the fragmentation. Another issue for the chunks in the text index, is that each time an entry should be added to a chunk, the chunk has to be copied in and out from the buffers, and thus a large size would increase the CPU usage and use of memory/bus bandwidth. We studied the performance with different chunk sizes in the range of 100 to 2000 bytes. The results show that 400 bytes is a reasonable choice for the chunk size.

8.5 Load and update time

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



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

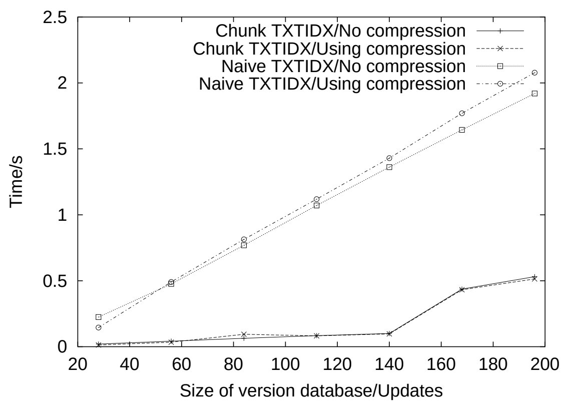
Figure 6. Load and update performance. The size is given as number of update transactions, each typically increasing the database size with 40-50 MB.

have to be retrieved in order to determine if the document has changed. If it has not changed, it is not inserted. Figure 6a 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 6a. 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 previous point.

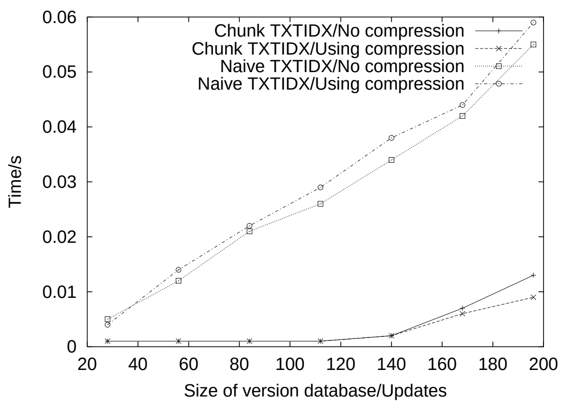
If a document does not already exist in the database (the name is not found in the document name index), there is no previous version that has to be retrieved. However, it is guaranteed that the document has to be inserted and indexed. This is more expensive on average. This is illustrated in Figure 6b, which shows the cost of inserting a set of new documents into the database as described previously.

Figure 6c 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 6, 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



(a) Frequent words.



(b) Medium frequent words.

Figure 7. Text containment, all versions.

is cheap, a reduction of this order can mean the difference between a project that is feasible and one that is not.

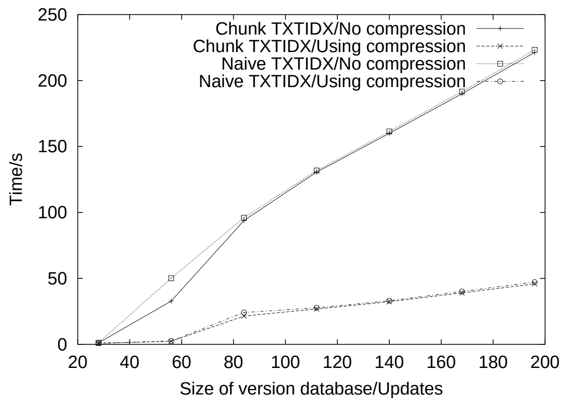
8.6 Query time

Figure 7 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).

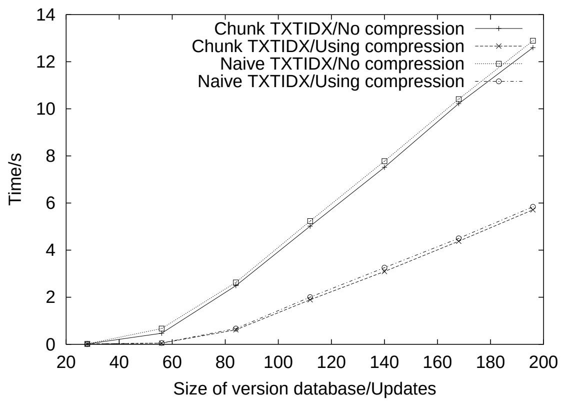
Figure 8 illustrates the average cost of retrieving the VIDs of all document versions valid at time $t = t_{Mid}$ that contained a particular word. Figure 9 shows the cost of queries for all *current versions* containing a particular word. Figure 10 shows the cost for queries for document versions valid sometime during the time period $p = [0, t_{Mid} >$, and that contained a particular word, and Figure 11 shows the cost for queries for document versions valid sometime during the time period $p = [t_{Mid;}, t_{Now} >$ and that contained a particular word, The main cost during a temporal text-containment query using the rather naive approach in this prototype (i.e., text-index lookup followed by time-selection on versions by retrieving the meta-chunk for each possible match), is the retrieval of the meta-chunks. This cost is the same both using naive text-index and chunk-based text index. However, the fact that the chunk-based text index is more space efficient increases the buffer hit ratio and improves performance slightly. This is also the main reason why use of compression improves the performance considerably, it increases the chance that data (and in particular the meta chunks) can be found in main memory.

8.7 Disk size

Most of the disk space used for the temporal document storage is for the document versions. The space they occupy, depends on the page fill-factor in the version database. With only insertions into the database the fill factor will be better than 90% (some space is wasted because of fragmentation,

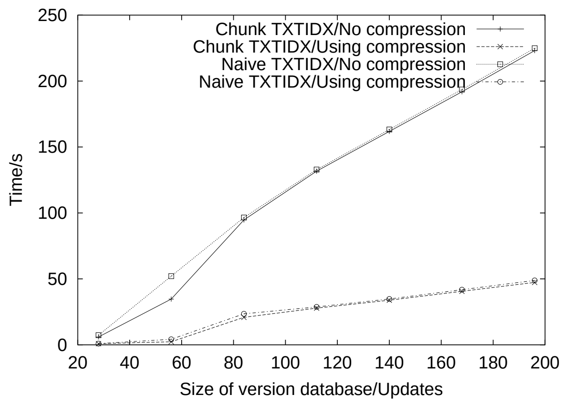


(a) Frequent words.

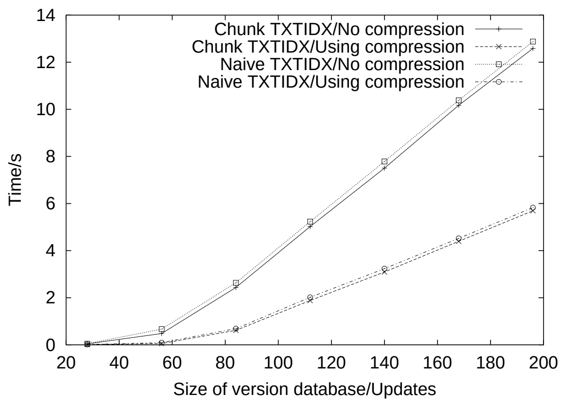


(b) Medium frequent words.

Figure 8. Temporal text containment, time selection at time $t = t_{Mid}$.

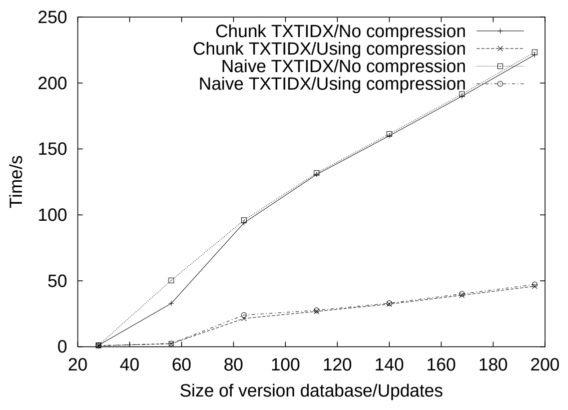


(a) Frequent words.

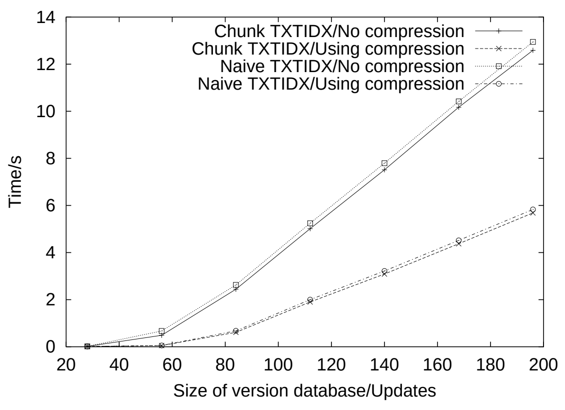


(b) Medium frequent words.

Figure 9. Temporal text containment, time selection of current versions at time $t = t_{Now}$.

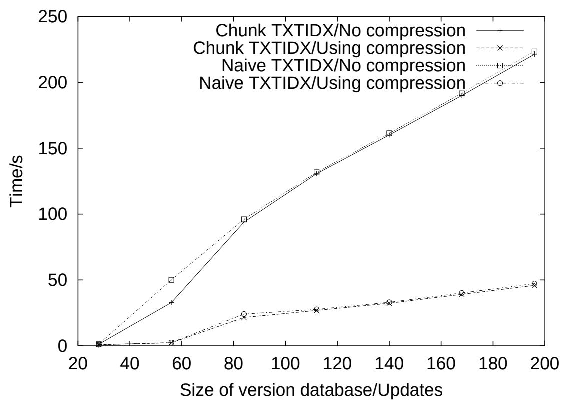


(a) Frequent words

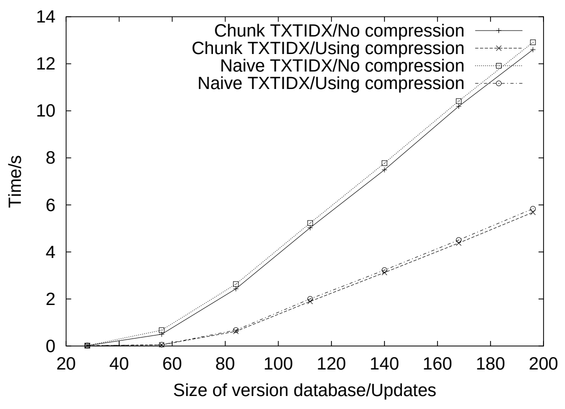


(b) Medium frequent words

Figure 10. Temporal text containment, period selection of $p = [0, t_{Mid} >$.



(a) Frequent words



(b) Medium frequent words

Figure 11. Temporal text containment, period selection $p = [t_{Mid;}, t_{Now} >$.

of there is not space for a chunk on a page, the system starts on a new page, and the free space on the previous page will be unused). If the fill-factor for some reason should drop, for example because of deleting non-temporal documents, the reorganizer can be used to compact the pages and increase the fill factor. When compression was enabled, the size of the version database was reduced from 9.7 GB down to 1.9 GB, which is only 20% of the original size.

The size of the text index in V2 has in our experiments been between 6% and 7% of the indexed text stored in the version database when the improved chunk-based text index was used. When the naive text-index was used, the size of the text index was approx. 22% of the indexed text stored in the version database.

The size of the document index is only marginal compared with the version database and text index, 4 MB for the current version index, and 28 MB for the historical version index.

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. We also identified some additional problems with timestamp management. 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 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. Because we have not yet enabled logging, the performance of small transactions is much lower, due to the fact that all buffers are flushed at transaction commit time.

As can be expected, the performance of V2 depends much on available buffer space. The most critical is the text index. During large transactions, it will have a high number of updates, and with insufficient buffer for the text index pages, a lot of costly and unnecessary random writes would have to be performed before commit time. For a typical modern server this is in many cases not be a problem. However, in order to be truly scalable, better dynamic text indexing techniques are necessary.

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. In this context, we have identified several research issues that we would like to focus on:

- More efficient support for operations that can be assumed to be frequent. One particular example is temporal text-containment queries in the presence of a temporal text index, where we search for documents containing one or more words at a particular point in time. Today, this can be achieved by combining the `contain` and `time_select` operators, but what is desired is efficient temporal text-containment operators like the following:

`containsT`: $(\{word\}, t_s) \rightarrow \{vid\}$
`containsP`: $(\{word\}, period) \rightarrow \{vid\}$

- In order to support efficient querying of temporal XML documents we plan to support the operators presented in [18, 19], which includes queries on element level, including path queries.
- Query language and query processing.
- Temporal join.
- Indeterminacy: This includes treatment of uncertain timestamps, for example when used as a web warehouse as described in the example in Section 3.
- Valid-time support.
- 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.

Finally, we also mention that although V2 is designed and optimized for document storage, it could also be employed in other contexts, including the use as a temporal object database.

Acknowledgments

Many of the basic ideas of this paper was 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] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proceedings of the 19th International Conference on Very Large Data Bases*. Morgan Kaufmann, 1993.

[2] 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.

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

[4] 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.

[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*, 1998.

[6] S. S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *TAPOS*, 5(3), 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, Time-Center, 2000.

[8] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In *Proceedings of VLDB 2001*, 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 scientific applications (RIDE-DM'2001)*, 2001.

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

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

[12] 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.

[13] C. S. Jensen and D. B. Lomet. Transaction timestamping in (temporal) databases. In *Proceedings of the 27th VLDB Conference*, 2001.

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

[15] G. Navarro and R. Baeza-Yates. Proximal nodes: A model to query document databases by contents and sructure. *ACM Transactions on Information Systems*, 15(4), 1997.

[16] 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)*, 2000.

[17] K. Nørvåg. *Vagabond: The Design and Analysis of a Temporal Object Database Management System*. PhD thesis, Norwegian University of Science and Technology, 2000.

[18] 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.

[19] K. Nørvåg. Temporal query operators in XML databases. In *Proceedings of the 17th ACM Symposium on Applied Computing (SAC'2002)*, 2002.

[20] K. Nørvåg. Temporal XML data warehouses: Challenges and solutions. In *Proceedings of the Workshop on Knowledge Foraging for Dynamic Networking of Communities and Economies (in conjunction with EurAsia-ICT'2002)*, 2002.

[21] K. Nørvåg, M. Limstrand, and L. Myklebust. TeXOR: Temporal XML Database on an Object-Relational Database System. In *(submitted for publication)*, 2002.

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

[23] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. D. Ullman, and J. L. Wiener. LORE: a lightweight object repository for semistructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996.

[24] A. Steiner. *A Generalisation Approach to Temporal Data Models and their Implementations*. PhD thesis, Swiss Federal Institute of Technology, 1998.

[25] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th VLDB Conference*, 1987.

[26] T. Suzuki and H. Kitagawa. Development and performance analysis of a temporal persistent object store POST/C++. In *Proceedings of the 7th Australasian Database Conference*, 1996.

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