Granularity Reduction in Temporal Document Databases

Kjetil Nørvåg [1]

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

Abstract

With rapidly decreasing storage costs, temporal document databases are now a viable solution in many contexts. However, storing an ever-growing database can still be too costly, and as a consequence it is desirable to be able to physically delete old versions of data. Traditionally, this has been performed by an operation called *vacuuming*, where the oldest versions are physically deleted or migrated from secondary storage to less costly tertiary storage. In temporal *document* databases on the other hand, it is often more appropriate to remove *intermediate* versions instead of removing the *oldest* versions. We call this operation *granularity reduction*. In this paper we describe the concept of granularity reduction, and present six strategies for selecting the document versions to eliminate. Three of the strategies have been implemented in the V2 temporal document database system, and in this context we discuss the cost of applying the strategies.

Key words: Temporal databases, document databases, vacuuming, granularity reduction

1 Introduction

Nowadays, most documents are produced electronically on computers, and more and more of these documents are stored in some kind of database management system. Previously, it has been quite common that only the last version of a document has been stored. However, with rapidly decreasing storage costs, it is now more often affordable also to keep the previous versions of the documents in the databases. An example of an application of such systems is *temporal XML/web warehouses*. In these systems it should be possible to query for individual pages or web sites as

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

they were at a particular time T, query for all versions of pages that contained one or more particular words at a particular time T, etc.

Versions of documents in document databases have traditionally been stored and retrieved using the document name and the version number. However, with more powerful systems we can take advantage of the temporal aspect, and make it possible to retrieve documents based on predicates involving both *document contents* and *validity time*, and in this way satisfy the query types mentioned above. A system supporting these features is called a *temporal document database system*, and we have in previous papers described the design and implementation of such a system: the V2 temporal document database system [11]. We have also presented algorithms for querying temporal XML databases [10], where queries can be on structure as well as text content.

However, even though storage cost is decreasing, storing an ever-growing database can still be too costly in many cases. A large database can also slow down the speed of the database system, for example because of the size of the indexes. As a consequence, it is desirable to be able to physically delete old document versions. Traditionally, this has been performed by a process called *vacuuming*, where the oldest versions are physically deleted, or migrated from secondary storage to less costly tertiary storage.

If we compare changes between versions in relational (or object) databases and document databases, we can make an important observation: the *relative* changes between document versions is in general much smaller than the relative changes between tuple versions. For example, if a tuple contains 4 fields and one field is changed, we can consider 25% of the tuple changed. However, we can assume that as much as 25% change between two *document* versions is less frequent. This has also been verified from the multi-versioned web pages we have included in our test data. For example, information and course web pages from our university are typically created and then updated as new information comes available or errors are detected. The same applies to pages from newspapers on the web. The actual articles usually only change because of corrections, while the portal pages (the main pages of a newspaper) change many times a day, but changes are very small (typically, when a new event happens and a new article is submitted to the system). A third example is typical web homepages. Updating is often an iterative process: changes are made, the page is viewed, feedback received, and new changes are applied. The conclusion we draw from this observation, is that the difference in information content between versions often is small, so that we in a temporal document database should consider removing intermediate versions instead of removing the oldest versions. We call this process *granularity reduction*, which can be considered a special case of vacuuming that is particularly suitable for temporal document databases (for relational/object databases other alternative techniques can be more applicable, we will describe some of these in Section 2). The advantage of granularity reduction compared to traditional vacuuming is that more of the knowledge is preserved, and we can still perform queries and retrieve useful results based on the oldest versions.

The use of granularity reduction can be illustrated by the following example: Assume we have a large number of document versions created during a relatively 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*. It is likely that if a document has a high number of updates during a short time period, the changes are not large, and in many cases also individually insignificant. 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. An example of a simple strategy for granularity reduction is to remove versions in a way that leaves *at most one version per day* in the database.

In this paper, we describe six strategies for selecting the document versions to eliminate, and discuss relative merits of these strategies. We discuss the cost of applying the different strategies, and for the three strategies we consider most interesting we also provide performance numbers based on their implementation in the V2 temporal document database system [11].

The organization of the rest of this paper is as follows. In Section 2 we give an overview of related work. In Section 3 we provide the context and general assumptions we make by giving an overview of the V2 temporal document database system. In Section 4 we describe our strategies for selecting which document versions to eliminate. In Section 5 we describe granularity reduction in combination with tertiary memory migration. In Section 6 we study granularity reduction cost. Finally, in Section 7, we conclude the paper.

2 Related work

A number of techniques have been proposed to reduce the amount of storage needed for storing data in temporal and versioned databases. The techniques can be divided into two categories:

- Lossy: Complete versions of items or part of the information stored in the items are removed. Techniques include simple deletion, variants of vacuuming (including data expiration), use of summary data, and granularity reduction.
- Non-lossy: All information in all versions is retained. Techniques include (non-lossy) compression, use of deltas, and compacted-archive techniques.

The most basic *lossy* technique is simply deleting some of the data in the database. A second technique (which can also be viewed as a special case of deletion) is vacuuming. Frequently, vacuuming by cut-off points has been assumed. In this case, it is specified that data that is older than a certain age (or created before a particular time) should be considered inaccessible and can be removed. Example of a systems supporting this strategy are POSTGRES [17] and our V2 system [11]. Vacuuming by cut-off points is also supported by the TSQL2 temporal query language.

A more detailed description of vacuuming and associated concepts is given by Jensen in [7]. He argues for disciplined vacuuming, so that during subsequent queries it is possible to know that data that could affect the result of queries is missing. This approach is further developed by Skyt et al. [15], who establish a foundation for the correct processing of queries and updates against vacuumed databases.

Related to vacuuming is data expiration in data warehouses, where the goal is to still be able to correctly answer a fixed set of queries. In [6], Garcia-Molina et al. describes a framework for system-managed removal of warehouse data that avoids affecting the user-defined views. In [19], Toman presents a technique for automatic expiration of data in a historical data warehouse that preserves answers to a known and fixed set of first-order queries.

An approach with conceptual similarities to granularity reduction, is to aggregate old data/create summary data. For example, data reduction in dimensional data warehouses can be achieved by aggregating data to higher levels in the dimensions, as described by Skyt et al. [16].

The most basic *non-lossy* technique is compression of documents. This technique is in particular useful for document databases where the size of the compressed data is typically only 20% of the size of the uncompressed data [11]).

Another technique that is frequently used is to store only one of the versions of a document as a complete version, and the other versions as delta versions (or edit scripts) that can be used to transform the complete version to one of the other original versions (this approach is also often called a *diff* approach). One of the most well-known systems employing this technique is the RCS version control system [18]. Other examples are described in [4] and [8].

In the last technique, which we denote *compacted archive*, all versions of a document are stored in one archive. An example of such a system is presented by Buneman et al. [3]. There different versions of a timestamped XML document are stored in one archive document. Elements in the archive are timestamped so that retrieval for a particular version can be performed. The compacted archive approach is particularly useful when keys can be identified. Another example of using a compacted archive techniques is presented by Wang and Zaniolo [20].

In general, it is possible to combine the use of lossy and non-lossy techniques. For example, granularity reduction can be combined with compact archiving; the granularity reduction can be applied to versions implicitly stored in the archive document, thus reducing the total size.

3 An overview of the V2 temporal document database system

Three of the presented strategies for selecting the document versions to eliminate have been implemented into V2. In order to make this paper self-containing, and provide the context for the rest of this paper, we give in this section a short overview of V2. For a more detailed description, and a discussion of design choices, we refer to [11].

V2 provides support for storing, retrieving, and querying temporal documents. For example, it is possible to retrieve a document stored at a particular time T, retrieve the document versions that were valid at a particular time T and that contained one or more particular words. In contrast to many existing systems that support versioning of documents, time is an integrated concept in V2, and is efficiently supported by the query operators.

3.1 Data and time model

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

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

The time model in V2 is a linear time model (time advances from the past to the future in an ordered step-by-step fashion). However, in contrast to most other transaction-time database systems, V2 does support reincarnation, i.e., a (logically) deleted version can be updated, thus creating a non-contiguous lifespan, with possibility of more than one tombstone for each document.

3.2 Design and implementation

The current prototype is essentially a library, where accesses to a database are performed through a V2 object, using an API supporting the operations and operators

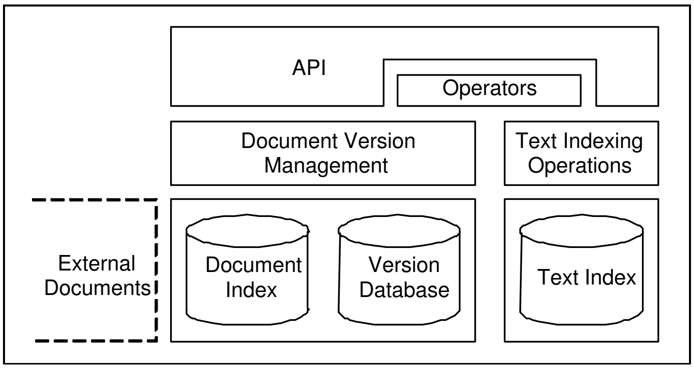


Fig. 1. The V2 prototype architecture.

as described in [11]. The bottom layers are built upon the Berkeley DB database toolkit [13], which we employ to provide persistent storage using B-trees.

The architecture of V2 and its main modules are illustrated in Fig. 1. The main storage structures are:

- Version database: The actual document versions are stored in the version database. Document versions can be stored compressed, and this is transparent for the users.
- Document index: As mentioned with respect to the data model, a document is identified by a document name. Conceptually, the document 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 (i.e., whether previous versions should be kept when a new version of the document is inserted into the database) is stored. For each document *version,* some metadata is stored in structures called *version descriptors*: 1) timestamp and 2) whether the actual version is stored compressed or not. The document index is also used to provide the mapping from a document to the VIDs of its versions.
- Text indexing: A text-index module based on variants of inverted lists is used in order to efficiently support text-containment queries, i.e., queries for document versions that contain a particular word (or set of words). The system also supports several indexing methods that can be used to improve the performance of temporal text-containment queries. These indexing methods are described in more detail in [12].

4 Granularity reduction

Granularity reduction is the process of removing a number of document versions. The process can conceptually be divided into two parts: 1) determine the set G of versions that should be removed (in practice a set of VIDs), and 2) physically remove the versions. Granularity reduction can be applied to a single document only, or to a set of documents (possibly all documents in the database, including those that are logically deleted).

There are many possible strategies for selecting the document versions to eliminate, and in this paper we will concentrate on the following:

(1) Naive.
(2) Time.
(3) Periodic.
(4) Similarity.
(5) Change.
(6) Relevance.

Before we describe the strategies in more detail, it can be useful to have in mind that what will be the best strategy for a given application, depends very much on whether the content is mostly document-centric (most often documents meant for human consumption, like books, papers, etc., and can be various formats like plain text, HTML, XML, or even proprietary formats like MS Word), or data-centric (often XML documents meant for computer consumption). A system should typically support several of the strategies, but the decision on what strategy to use in a particular application context has to be based on knowledge of the contents, and can not be determined automatically by the system. Thus, the database administrator/user has to decide which strategy to use.

We will now describe the strategies in more detail. We denote the timestamp of a document D_i as T_i. The granularity reduction is applied to all the c versions of a document created before a certain time T_G, i.e., all versions with a timestamp $T_i \leq T_G$. For simplicity we always keep version D_c (in order to avoid special cases when D_c is the current version, which shall never be eliminated). We denote this set of candidate versions $D = D_1, \ldots, D_c$, where $T_{i-1} < T_i$ for all $1 < i \leq c$. The goal is to identify the set G containing the versions D_i that should be removed because of granularity reduction.

The versions in D are versions of the same document. If granularity should be applied to a number of documents j (which could even be all documents in the database), the actual granularity reduction algorithm should be applied separately on all j documents, creating a set G_k for each document. The final result set is then the union of the individual result sets, i.e., $G = G_1 \cup \ldots \cup G_j$.

4.1 Naive granularity reduction

The most basic strategy to determine the set G containing the versions that should be removed, is to remove every R document version. Note that in the special case of $R = 1$ all candidate versions are removed, and this equals vacuuming all versions with $T \leq T_G$. This also applies to the other strategies, where special cases exist that will result in removal of all candidate versions.

The algorithm for naive granularity reduction is outlined in Algorithm 1, and the result of applying this algorithm is illustrated in Fig. 2b. The problem with this strategy can be illustrated with an example where many versions are created close in time, for example during a revision process. Using this basic strategy, many of these versions will be kept, while other versions, already covering large time ranges, will be removed.

Algorithm 1 Naive granularity reduction.

```
$G = \emptyset$
$k = (c + 1) - R$
while $k > 0$ do
  add $D_k$ to $G$
  $k = k - R$
end while
```

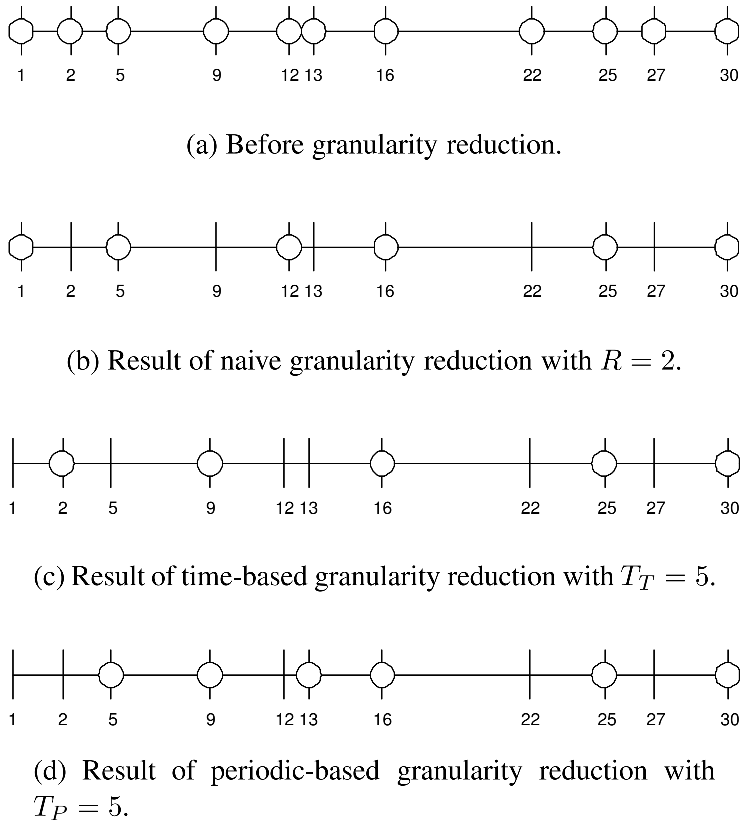
4.2 Time-based granularity reduction

The most straightforward *and useful* strategy to granularity reduction is time-based granularity reduction, where versions that are closer in time than T_T are deleted. The algorithm for time-based granularity reduction is outlined in Algorithm 2, and the result of applying this algorithm is illustrated in Fig. 2c.

Algorithm 2 Time-based granularity reduction.

```
$G = \emptyset$
$T_{prev} = T_c$
for $k = c$ downto $2$ do
  if $(T_{prev} - T_{k-1}) < T_T$ then
    add $D_{k-1}$ to $G$
  else
    $T_{prev} = T_k$
  end if
end for
```

We emphasize 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 that 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.



(a) Before granularity reduction.

(b) Result of naive granularity reduction with $R = 2$.

(c) Result of time-based granularity reduction with $T_T = 5$.

(d) Result of periodic-based granularity reduction with $T_P = 5$.

Fig. 2. Illustration of the different results of applying *naive*, *time-based,* and *periodic-based* granularity reduction on a document. The horizontal line is the time line, numbers denote time, and a circle denotes a document version stored in the database. A vertical line indicates a removed version.

4.3 Periodic-based granularity reduction

In the periodic-based strategy, the versions to keep are the ones created at a periodic interval, for example keeping one version from every Sunday. In that case, the starting point T_G should be a Sunday, and T_P equal 7 days. With this strategy, at most one version is kept for each period T_P, and it is also possible that the same version could be valid for more than one period.

The algorithm for periodic-based granularity reduction is outlined in Algorithm 3. The result of applying this algorithm is illustrated in Fig. 2d. Here we can also see the different result when applying the period-based strategy, compared to the time-based strategy.

Algorithm 3 Periodic-based granularity reduction.

```
$G = \emptyset$
$t = T_G - T_P$
$k = c - 1$ {Keep version $D_c$ because it covers $T_G$}
while $k > 0$ do
  if $T_k > t$ then
    add $D_k$ to $G$
  else
    if $T_k = t$ then
      $t = t - T_P$
    else
      $t = t - T_P * \lceil (t - T_k)/T_P \rceil$
    end if
  end if
  $k = k - 1$
end while
```

4.4 Similarity-based granularity reduction

The strategies we have described so far for selecting versions to be eliminated are simple strategies that do not consider the actual contents of the versions. A more adaptive and "intelligent" strategy is the similarity-based approach, which uses the actual similarity between document versions to decide which versions should be removed. The similarity between two document versions D_i and D_j can be computed by a similarity function $sim(D_i, D_j)$. Two identical documents will have $sim = 1$, and two documents that have no similarity , for example, does not have any terms in common, will have $sim = 0$. If the similarity between two document versions D_i and D_{i+1} is greater than a threshold d, document version D_i can be removed, based on the assumption that small changes are probably error corrections or similar. The algorithm for similarity-based granularity reduction is outlined in Algorithm 4.

Algorithm 4 Similarity-based granularity reduction.

```
$G = \emptyset$
$D_{prev} = D_c$
for $k = c$ downto 2 do
  if $sim(D_{prev}, D_{k-1}) > d$ then
    add $D_{k-1}$ to $G$
  else
    $D_{prev} = D_k$
  end if
end for
```

Using similarity-based granularity reduction, it might be difficult to know what is a reasonable value for the threshold d, and it is also possible that reasonable values differ from one document to another. In order to solve this problem, it is possible to use an adaptive process where the amount of data reduction is specified, for example that only a fraction p of the versions should be left after reduction. This can be achieved as follows:

(1) In the first step, all document versions are read and the document vectors are created.
(2) In the second step, Algorithm 4 is applied with different values of d until only a fraction p of the document versions are left, i.e., $|G| = pc$.

4.5 Change-based granularity reduction

The similarity strategy does not consider the structure of the documents, only the words they contain. In order to compare documents based on structure, a change/diff-based approach can be used. Using this strategy, the difference between two versions are calculated using a function *diff(*D_i, D_j) that calculates the difference between the document versions D_i and D_j. A difference of $d = 0$ means that the documents are identical, and a high difference means the document versions have few (or no) similarities. If the difference between two document versions D_i and D_{i+1} is less than a threshold d, document version D_i can be removed, based on the same assumption as before, i.e., that small changes are probably error corrections or similar. The algorithm for change-based granularity reduction is outlined in Algorithm 5.

Algorithm 5 Change-based granularity reduction.

```
$G = \emptyset$
$D_{prev} = D_c$
for $k = c$ downto 2 do
  if diff($D_{prev}, D_{k-1}$) $< d$ then
    add $D_{k-1}$ to $G$
  else
    $D_{prev} = D_k$
  end if
end for
```

The *diff* function can be based on an algorithm that produces an edit script (a set of basic edit operations that will transform a document D_i into document D_j, examples of operations are *insert line* and *delete line*). The result of the *diff* function can for example be a weighted sum of the operations, because an operation inserting or deleting a line should contributed more than an operation that simply inserts (or deletes) a single letter. Using this result directly is difficult, because the value when applied to large documents will typically be larger than the value when applied to small document. Normalizing the result into the range $[0.0, 1.0]$ is difficult, but dividing the result by the number of lines or number of characters of the document gives a partial normalization, although not limited to values $v \leq 1.0$.

The basic *diff* function considers the documents as simple text. This is similar to the *diff* command in Unix. However, for HTML and XML documents, diff algorithms that consider the hierarchical structure are more appropriate, and as a result the system should support more than one algorithm for the *diff* function. For example, the basic *diff* function can use the same algorithm as used by many *diff* command implementations [9]. In addition, an improved *diff* function can be provided to be used for XML documents, for example one of the algorithms proposed by Cobena et al. [5], or by Wang et al. [21]. It should be noted that in the context of granularity reduction, speed is likely to be more important than a high-quality diff algorithm that could give an optimal/minimal result. This should be considered when comparing possible algorithms.

The adaptive technique described for similarity-based granularity reduction can also be applied for change-based granularity reduction.

4.6 Relevancy-based granularity reduction

The ideal goal in granularity reduction is *to keep those versions that are most relevant* and as such contribute most knowledge to subsequent queries. Thus, in granularity reduction based on relevancy, all document versions with a relevancy rank below a threshold r are removed. This is illustrated by Algorithm 6.

Algorithm 6 Relevancy-based granularity reduction.

```
$G = \emptyset$
for $k = 1$ to $c$ do
    if $rank(D_k) < r$ then
        add $D_k$ to $G$
    end if
end for
```

The relevance of a document is traditionally calculated *with respect to some query* (or set of queries) Q. However, at granularity reduction time, we do not in general know the future queries. Thus, the problem is *to find a relevancy measure without the exact knowledge of* Q. We can distinguish between plain text documents, and documents that can also contain links (for example HTML or XML documents).

Relevance of plain text documents. In traditional information retrieval systems, as well as Web search engines, result documents are ranked by relevance based on a particular query. One possible approximation in our context is to maintain statistics over the most common search words and search phrases. The rank of a document version D_i can be calculated by using the similarity measure between the document version and the most frequent k search phrases S_k:

$$rank(D_i) = \sum_k sim(D_i, S_k)$$

Other traditional text document ranking algorithms can also be used, for example based on statistical measures such as query term frequency and inverse document frequency of the term. An alternative or extension to future query prediction is to use metadata like, for example, length of the document or language.

It is also possible to base relevance of a document version on the number of previous retrievals, for example by keeping a retrieval counter for each document version. The counters should be normalized with respect to the age of document versions (this can be achieved by regularly decrementing all counters). In this way, the counters reflect the actual access pattern. When a document version is inserted, the counter is initialized to the current average of counter values. On every access, the counter is incremented by one (if it already has the maximum possible value, it should not be changed). The cost of maintaining the counters is marginal. For example, in a 10 GB database with an average document size of 20 KB, the number of document versions is 500,000. Assuming 4 B for each counter, only 2 MB of main memory is needed to store the counters (for performance reasons they should always be main-memory resident). The counters should be stored on disk, but as it is not critical if some updates to the counters are lost after a crash it is not necessary to update the disk during each counter update or after each commit; regularly checkpointing the counters to disk should suffice (this can be done in one efficient write operation, which in this example should take less than 100 ms using a modern disk).

Relevance of documents containing links. It is possible to employ additional techniques to determine relevance for documents containing links, for example in the context of a temporal web warehouse. These techniques can be based on the page-ranking algorithms that are used for web pages. For example, the Google strategy can be used, where the rank of a document is computed as a combination of the PageRank [2] which is based on links, and the relevance of document with respect to query words.

Due to the problem of predicting future queries, it can be expected that the first five strategies to granularity reduction in general will give a more desirable and predictable result than using relevancy-based granularity reduction. However, if we want to perform data reduction on documents where only one version of each document exists (the algorithms above are intended for reducing the number of versions of a particular document), the first five strategies are not applicable. In that case, the relevancy-based strategy can prove useful.

4.7 Combining strategies

In very large document databases where the cost of using similarity- or change-based granularity reduction on all document versions is too costly, it is sometimes possible to use a low-cost granularity reduction strategy (naive, time-based, or period-based) as a filtering step before applying one of the more costly strategies in order to reduce the processing cost. However, this has the risk of eliminating some important versions in the filtering step, thus should *only be performed when particular knowledge about the temporal versioning pattern is known.* An example of a page where a combined strategy can be employed, is a web page providing patches for the XYZ suite of software. The page is updated with new patches the second Tuesday every month, but some months there are no new patches made available, and it can also happen that small error corrections are made to the web page other times as well (typically added information). A possible combined strategy to apply to this page is to first employ periodic-based granularity reduction, followed by change-based granularity reduction.

5 Granularity reduction and tertiary memory

Until now, we have assumed that document versions identified during the granularity reduction process should be physically removed from the system. However, another use of granularity reduction (and vacuuming in general), is to move data from secondary storage to less expensive and larger tertiary memory (for example tape or optical storage).

If moving data to tertiary storage, there is a choice whether the indexes that index the data on tertiary storage should be stored on tertiary storage, or if the indexes on secondary storage should be used to index the actual data that have been moved to tertiary storage. In a document database, the text index is the most space-consuming component in addition to the documents themselves. However, the space usage of a space-efficient text index can be expected to be less than 5% of the document size, so that it is in many cases feasible to keep the indexes on secondary storage: k GB of disk can be used to index $20k$ GB of documents on tertiary storage. In our context, there is an additional reason why we see this as beneficial when possible: text-index operations are very costly. If the index entries should also be migrated to tertiary storage, they have to be removed from one text index, and inserted into another text index. By keeping them on secondary storage this costly operation is avoided, only the mapping between VID and physical location needs to be updated. Updating the mapping requires one update in the document index, however, because the mapping information for the versions of a particular document is clustered together, the average cost will be much less than one index node. Finally, it should be mentioned that with the inexpensive high-capacity disks available nowadays, using tertiary storage for on-line documents will probably not be very common in the future.

6 Granularity reduction cost

The cost of granularity reduction can be divided into two parts. The first is the cost of determining the versions to be removed (creating the set G of VIDs as described previously), and the second is the cost of physically removing the versions. Both costs depend on which strategy is used. The granularity reduction strategies can be classified into two categories:

(1) Strategies that only consider the version metadata (timestamp) and where it is not necessary to load the documents themselves. This is the case for the naive-, time-, and periodic-based strategies. The algorithms behind these strategies are also computationally cheap. Denoting the average versions of a document as n, the time complexity of these algorithms is $O(n)$.

(2) Strategies that actually compare the contents of the document versions, and where the document versions will always have to be retrieved. This is the case of the similarity-, change-, and relevancy-based approaches. The CPU cost can also be significant, especially in the case of a change-based strategy (but in this context it should be noted that the cost can be much lower if the document versions are stored as delta documents instead of complete documents). The time complexity for the similarity-based approach as implemented in our prototype is $O(n * (k \log w + w))$, where k denotes the number of words in a document version, and w denotes the vocabulary size for two documents to be compared. The complexity when using change- and relevance-based approaches will depend on which change/relevancy algorithm is used.

The cost of actually removing the document versions is a function of the number of elements in G.

In order to give an idea of the cost involved in deciding removal candidates in granularity reduction, as well as the cost of removing the document versions, we have implemented three of the strategies into the V2 temporal document database system: the time-, periodic-, and similarity-based strategies. These were chosen because 1) they represent both of the categories described above, 2) they have a predictable performance in terms of quality of document version selection, and 3) it is intuitive for the users/administrators of such a system why particular versions are selected for removal. We will now give some performance results from the granularity reduction of a temporal document database. The period-based strategy has the same cost as the time-based strategy, and is therefore omitted from the following discussion.

Integration of granularity reduction in V2. The selection of document versions to be removed during granularity reduction is performed by the Document Version Management layer (see Fig. 1), but are also supported by particular granularity reduction operators so that granularity reduction can also be part of a more general operation/query. In the case of the naive, time- and periodic-based strategies the set G of document versions to eliminate is created based on information in the document index. In the case of the similarity-, change-, and relevancy-based strategies the actual document versions have to be retrieved from the version database as well.

In practice, granularity-reduction can be performed in two phases as described in Section 4, or the versions can be removed immediately, as soon as they are identified. The advantage of removing the versions as soon as they are identified is that the relevant metadata is already resident in main memory. However, the removal of items in the text index is a much more costly operation than removing the actual version. In order to reduce this cost it is advantageous to remove a number of versions in batch so that disk-arm movement can be reduced. This is the approach used in our implementations. It is in some cases also necessary to preprocess the set G before the versions are removed. This can be the case when adaptive methods or a combination of granularity-reduction strategies are used.

In V2 we use the vector space model [14] for similarity measures, more specifically the popular *cosine similarity measure*. This measure is known to perform well, and is relatively inexpensive to compute. Using the vector space model, a document version D_i is represented by a vector

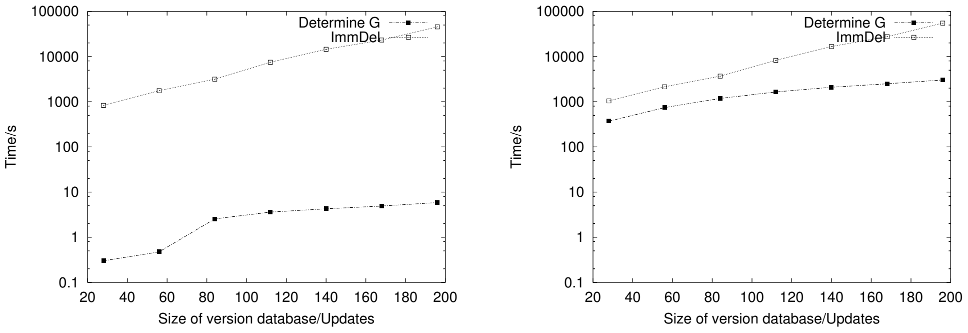
$$d_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,t})$$

where each $w_{i,j}$ is a weight (number of occurrences in our case) for the term/word j in the document version (the same word should of course have the same position in the two vectors representing two documents to be compared). The similarity between two document versions D_i and D_j can be expressed as [1]:

$$sim(D_i, D_j) = \frac{d_i \cdot d_j}{|d_i||d_j|} = \frac{\sum_k w_{i,k} * w_{j,k}}{\sqrt{\sum_k w_{i,k}^2} * \sqrt{\sum_k w_{j,k}^2}}$$

Using this measure, the extreme values are $sim = 1$ for two identical documents, and $sim = 0$ for two documents that share no terms. In the database the document versions are essentially stored sequentially in VID order. If more than one document is to be granularity reduced at the same time, it can be advantageous to create several of the G sets in parallel. In this way, more clever scheduling of the retrieval of the document versions can be employed to reduce disk-arm movement.

Test data. 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 have been downloaded once a day, by crawling each site starting with the site's



(a) Time-based.

(b) Similarity-based.

Fig. 3. Cost of determining granularity reduction set G (“Determine G”) and the cost of granularity reduction with immediately delete (“ImmDel”) for the time- and similarity-based granularity reduction strategies. The size of the database is given as updates (each day is one update), and after 200 days the size of the version database is 9.5 GB.

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 91 MB (approximately 10,000 web pages). An average of approximately 500 web pages were updated each day, approximately 300 web pages were removed (all pages that were successfully retrieved on day d_i but not available at day d_{i+1} were considered deleted), and approximately 350 new pages were inserted. Thus, on average approximately 850 versions were inserted into the database for each day. The average size of the updated pages was relatively high (the raw pages that are stored in the database also contain HTML formatting information), resulting in an average increase of the version database of 45 MB for each set of pages loaded into the database.

System configuration. 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 the operating system (FreeBSD) and the V2 system, one disk for storing the database files, and the third disk was used for storing the test data files (the web pages). We configured the system so that the version database and the text index had separate buffers, and the size of these was explicitly set to 100 MB and 200 MB, respectively.

Results. In the experiments we measured the cost of determining the granularity reduction set G when all historical document versions are considered. This cost is illustrated in Fig. 3 for the time- and similarity-based granularity reduction strategies (as noted, the cost of the periodic-based strategy is the same as the time-based strategy). For both strategies, the figure show the cost of determining the set G only, and the total cost of determining G and removing the document versions. Note that the total granularity reduction cost of the strategies can not be compared to each other based on the numbers in the graphs, because each of the strategies resulted in a different number of documents being removed: approximately 115,000 document versions using the time-based strategy, versus approximately 150,000 documents when using the similarity-based strategy. The resulting database size after performing granularity reduction was approximately 30% of the original size when using the time-based strategy, and from 30% (down from 1.2 GB to 0.37 GB) for the small-sized database, to 15% (down from 9.5 GB to 1.4 GB) for the largest sized database when using the similarity-based strategy with $d = 0.7$.

As illustrated in Fig. 3a, the cost of determining G using the time-based strategy is low. Even in the case of 9.5 GB of documents it takes less than 5 seconds. The reason is that the document versions themselves do not have to be retrieved, only the document index (approximately 20 MB) has to be scanned. The cost of determining G using the similarity-based strategy is much higher. As illustrated in Fig. 3b, this takes approximately 2,500 seconds. In this case the actual document versions have to be retrieved, i.e., reading 9.5 GB of data. Currently the retrieval of document versions is not fully optimized, so this cost can be reduced by more clever scheduling of the retrieval of the document versions. However, we note that even a best-case number resulting from sequential scan of the database would be in the order of 400 seconds.

The most significant cost of the granularity reduction process is the cost of the actual removal of the document versions (except when only a very small number of document versions are selected for removal, compared to the total number of document versions that have to be considered). As can be seen from Fig. 3, the cost is in the order of 50,000 seconds. The actual cost of removing a document version is the same for all approaches, but with the parameters used in the measurements more versions are selected for removal using the similarity-based strategy, and the removal times increased correspondingly.

Regarding removal of document versions it should be noted that:

- The increase in cost with increasing database size (and size of G) in Fig. 3 is relatively high. The reason is that for smaller database sizes most of the index structures fit in main memory, while for larger databases this is not the case. Thus, for most typical applications, where very large databases compared to main memory can be expected, the cost for large databases in the figure will be most relevant.
- When granularity reduction drastically reduces the size of the database, as is the case here, it can be more efficient to create a new text index of the remaining document versions instead of in-place updating the existing text index.
- In the case of migration of documents to tertiary storage while keeping the text index on secondary storage, the cost will be much lower. The document versions will have to be removed from the version database and the document index will have to be updated with the new physical address of the document versions, but

these operations will not involve much random disk accesses, so the cost will be relatively low.

As can be seen from the performance results, the cost of performing granularity reduction can be quite high. However, granularity reduction of all documents at the same time should not often be necessary, and the process can also be done in parallel with other tasks: in the worst case, only the document versions currently being considered have to be locked.

7 Conclusions and further work

In this paper we have introduced and described granularity reduction of document versions, which can be considered as a special case of vacuuming that is particularly applicable for temporal document databases.

We described algorithms for six strategies for selecting the document versions to eliminate, and discussed advantages and disadvantages of these strategies. We discussed the efficiency of the strategies, and for three of them we also provided performance numbers based on their implementation into the V2 temporal document database system.

For future research, we would like to explore further the issues related to relevancy of documents. As noted, that strategy can be useful for determining individual one-version/non-temporal documents in the database that are candidates for removal. The performance measurements showed that the bottleneck is the maintenance of the text-index when document versions are removed. A more detailed study on this aspect should be performed.

Acknowledgments

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

The author would also like to thank Ståle Smedseng and the anonymous reviewers for useful and constructive comments which have helped to improve the quality and readability of this paper.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *WWW7/Computer Networks*, 30(1-7):107–117, 1998.

[3] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002.

[4] 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.

[5] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[6] H. Garcia-Molina, W. J. Labio, and J. Yang. Expiring data in a warehouse. In *Proceedings of the 24th VLDB Conference*, 1998.

[7] C. S. Jensen. Vacuuming. In R. T. Snodgrass, editor, *The TSQL2 temporal query language*. Kluwer Academic, 1995.

[8] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 581–590, 2001.

[9] W. Miller and E. W. Myers. A file comparison program. *Software – Practice and Experience*, 15(11):1025–1040, 1985.

[10] K. Nørvåg. Algorithms for temporal query operators in XML databases. In *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops, EDBT 2002 Workshops XMLDM, MDDE, and YRWS. Revised Papers*, 2002.

[11] K. Nørvåg. The design, implementation, and performance of the V2 temporal document database system. *Journal of Information and Software Technology*, 46(9):557–574, 2004.

[12] K. Nørvåg. Supporting temporal text-containment queries in temporal document databases. *Journal of Data & Knowledge Engineering*, 49(1):105–125, 2004.

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

[14] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[15] J. Skyt, C. S. Jensen, and L. Mark. A foundation for vacuuming temporal databases. *Journal of Data & Knowledge Engineering*, 44(1):1–29, 2003.

[16] J. Skyt, C. S. Jensen, and T. B. Pedersen. Specification-based data reduction in dimensional data warehouses. In *Proceedings of the 18th International Conference on Data Engineering*, 2002.

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

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

[19] D. Toman. Expiration of historical databases. In *Proceedings of TIME-01*, 2001.

[20] F. Wang and C. Zaniolo. Temporal queries in XML document archives and web warehouses. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic (TIME'2003)*, 2003.

[21] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: an effective change detection algorithm for XML documents. In *Proceedings of the 19th International Conference on Data Engineering*, 2003.