Norwegian University of Science and Technology
Technical Report IDI-TR-1/2003

Algorithms for Granularity Reduction in Temporal Document Databases

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

ISSN: 1503-416X

Abstract

With rapidly decreasing storage costs temporal document databases is 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. Traditionally, this has been performed by an operation called *vacuuming*, where the oldest versions are physically deleted (or migrated from secondary storage to cheaper tertiary storage). However, in temporal *document* databases it is more appropriate to remove intermediate versions instead of removing the oldest versions. We call this operation *granularity reduction*. In this paper we describe six approaches to granularity reduction, and discuss advantages and disadvantages of these approaches. Three of the approaches have been implemented into the V2 temporal document database system, and in this context we discuss the cost of applying the approaches.

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, often only the last version of a document was stored. However, with rapidly decreasing storage costs, it now more often affordable to also 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 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 [9]. We have also presented algorithms for querying temporal XML databases [8], where queries can also 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 cheaper 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. The conclusion we draw from this observation, is that we in a temporal document database should be able to remove intermediate versions instead of removing the oldest versions. We call this process *granularity reduction*, which can be considered as a special case of vacuuming that is particularly applicable 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, these 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 makes *at most one version per day* left in the database.

In this paper, we describe six approaches to granularity reduction, and discuss advantages and disadvantages of these approaches. We discuss the cost of applying the approaches, and for the three approaches we consider most interesting we also provide performance numbers based on their implementation in the V2 temporal document database system [9].

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 approaches to granularity reduction. 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

Frequently, vacuuming by cut-off points has been assumed. In this case, it is specified that data that is older than a certain time should be considered inaccessible and can be removed. An example of a system supporting this strategy is POSTGRES [14], and vacuuming by cut-off points is also provided by the TSQL2 temporal query language.

A more detailed description of vacuuming and associated concepts is given by Jensen in [5]. 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. [12], which 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 [4] Garcia-Molina et al. describes a framework for system-managed removal of warehouse data that avoids affecting the user-defined views. In [15] 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. [13].

3 An overview of the V2 temporal document database system

Three of the granularity reduction algorithms described in this paper have been implemented into the V2 temporal document database system. In order to make this paper self-containing, and provide the context for the rest of this paper, we give in this section an overview of V2. For a more detailed description, and a discussion of design choices, we refer to [9].

3.1 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 the object identifier in an object database.

3.2 Time model and timestamps

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

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

3.3 Functionality

V2 provides support for storing, retrieving, and querying temporal documents. For example, it is possible to retrieve a document stored at a particular time T, 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 of V2, and is efficiently supported by the 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 the tuples/objects in relational/object databases). The result of execution of the operators (i.e., VIDs, document names, timestamps, periods, etc.), 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. V2 supports a number of operators, including operators for returning the VIDs of all document versions containing one or more particular words, support for temporal text-containment queries, as well as the Allen operators [1], i.e., *before, after, meets,* etc.

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

3.4 Design and implementation

The current prototype is essentially a library, where accesses to a database are performed through a V2 object, using an API supporting the operations and operators described previously. The bottom layers are built upon the Berkeley DB database toolkit [10], which we employ to provide persistent storage using B-trees.

The architecture of V2 is illustrated in Figure 1, and the main modules are the version database, document name index, document version management, text index, API layer, operator layer, and optionally extra structures for improving temporal queries. We will now give an overview of the storage-related modules.

3.4.1 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

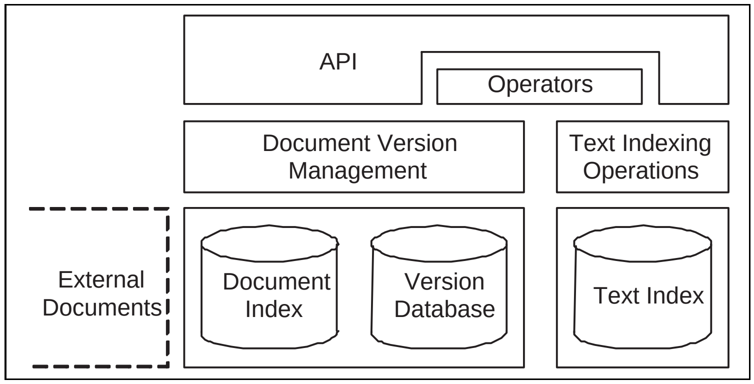


Figure 1: The V2 prototype architecture.

user/application) in a tree structure, where the concatenated VID and chunk number is used as the search key. The VID is essentially a counter, and given the fact that each new version to be inserted is given a higher VID than the previous versions, the document version tree index is append-only. An interesting side effect of this is that it is easy to later retrieve all versions inserted during a certain VID interval (which can be mapped from a time interval). One application of this feature is reconnect/synchronization of mobile databases, which can retrieve all versions inserted into the database after a certain VID (last time the mobile unit was connected).

In a temporal document database system, document versions can be stored as *complete* versions, or as a combination of complete versions and *delta* versions. The advantage of only storing complete versions is efficient access to particular versions. On the other hand, if delta versions are stored, one complete version and a number of delta versions will have to be retrieved in order to reconstruct a particular document version.

3.4.2 Document name index

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

We believe that support for temporal data should not significantly affect efficiency of queries for current versions, and therefore either a one-index approach with sub-indexes [7] or a two-index approach where current and historical information is stored in different indexes should be employed. One of our goals is to a largest possible extent using structures that can easily be integrated into existing systems, and based on this we have a two-index approach as the preferred solution. An important advantage of using two indexes is that the current version index can be assumed to be small enough to always fit in main memory, making accesses to this index very cheap. The disadvantage of using one index that indexes only current document versions, and one index that only indexes historical versions, is potentially high update costs: when a temporal document is updated, both indexes have to be updated. This could be a bottleneck. To avoid this, we actually use a more flexible approach, using one index that indexes the most recent n document versions, and one index that indexes the older historical versions. Every time a document is updated and the threshold of n version descriptors in the current version index is reached, all but the most recent version descriptors are moved to the historical version index. This is an efficient operation, effectively removing one chunk from the current version index, and rewriting it to the historical version index.

3.4.3 Text indexing

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

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. In the index, one or more chunks is used for each index word. Each chunk contains the index word, and a number of VIDs. For each VID inserted into the chunk, the size increases, until it reaches its maximum size (we use a value of 400 B). 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 text indexes is reduced by using compression of VIDs.

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).

In practice, granularity-reduction can be performed in two phases as described above, 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. As will be shown later, 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 approaches is used.

There is a number of possible approaches to granularity reduction, 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 approaches in more detail, it can be useful to have in mind that what will be the best approach for a given application, depends very much on whether the documents are 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).

We will now describe the approaches 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 assume that this time range might include the current version, so we always keep the version D_c. 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., $G = G_1 \cup \ldots \cup G_j$.

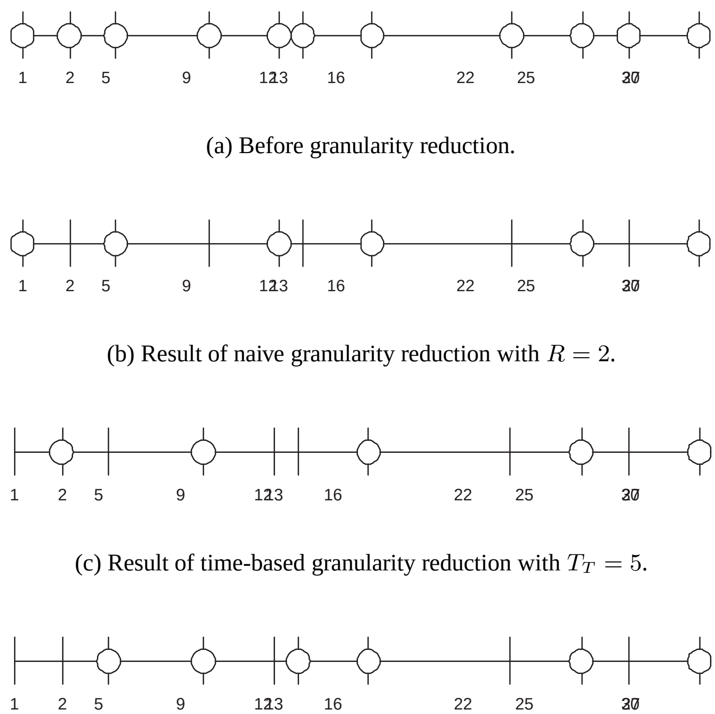
4.1 Naive granularity reduction

The most basic approach to granularity reduction is to remove every R document version, where $R > 1$. 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 approaches, 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 Figure 2b. The problem with this approach can be illustrated with an example where many versions are created close in time, for example during a revision process. Using the basic approach, 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



(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 $T_G = 5$.

Figure 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.

4.2 Time-based granularity reduction

The most straightforward *and useful* approach to granularity reduction is time-based granularity reduction, where the strategy is to delete versions that are closer in time than T_T. The algorithm for time-based granularity reduction is outlined in Algorithm 2, and the result of applying this algorithm is illustrated in Figure 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
    $prev = T_k$
  end if
end for

In this context 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.

4.3 Periodic-based granularity reduction

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

In the periodic-based approach to granularity reduction, the versions to keep are the ones created at a periodic interval, for example to keep 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 approach, 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 Figure 2d. Here we can also see the different result when applying the period-based approach, compared to the time-based approach.

4.4 Similarity-based granularity reduction

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
    $prev = D_k$
  end if
end for

The approaches to granularity reduction described so far are simple approaches that do not consider the actual contents. A more adaptive and "intelligent" approach is the similarity-based approach, which uses the actual similarity between document versions to decide which versions should be removed.

We use the vector space model [11] for similarity measures, more specifically the popular *cosine similarity measure*. This measure is known to perform well, and is relatively cheap 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:

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

Using this measure, $sim = 1$ for two identical documents, and $sim = 0$ for two documents that share no terms. 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 correction or similar. The algorithm for similarity-based granularity reduction is outlined in Algorithm 4.

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

Algorithm 5 Change-based granularity reduction.

```
G = ∅
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
    prev = D_k
  end if
end for
```

The similarity/vector approach only considers words, and not the structure of the documents. In order to compare documents based on structure, a change/diff-based approach can be used. Using this approach, the difference between two versions are calculated using a function `diff`(D_x, D_y) that calculates the difference between the document versions D_x and D_y. 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.

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_x into document D_y, 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 document, 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 [6], while an improved `diff` function can be provided to be used for XML documents. This improved `diff` can use an XML diff algorithm, for example as proposed by Cobena et al. [3], or by Wang et al. [16]. 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

Algorithm 6 Relevancy-based granularity reduction.

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

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.

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 generally 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 in addition can 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_j:

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

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 as 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 counter for each document version. The counters should be normalized with respect to age of document versions. This is 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 500000. Assuming 4 B for each counter, only 2 MB of main memory is needed to store the counters (they should always be memory resident). It is not critical if some updates to the counters are lost after a crash, so it is sufficient to regularly checkpoint the counters to disk. This can be done in one efficient write operation (which in this example takes less than 100 ms using a modern disk).

Relevance of documents containing links. For documents containing links, for example in the context of a temporal web warehouse, it is possible to employ additional techniques to determine relevance. These can be based on the page-ranking algorithms that are used for web pages, for example using the strategy used in Google, 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 approaches to granularity reduction in general will give a more desirable result than using relevancy-based granularity reduction. However, if we want to perform data reduction on documents where only one version of each document exist (the algorithms above are intended for reducing the number of versions of a particular document), the first five approaches are not applicable. In that case, the relevancy-based approach can prove useful.

4.7 Combining approaches

In very large document databases where the cost of using similarity- or change-based granularity reduction on all document versions is too costly, it can be beneficial to use a low-cast granularity reduction approach (naive, time-based, and period-based) as a filtering step before applying one of the more costly approaches.

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 (in general harddisk) to cheaper 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 also be stored/migrate to 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 that 5% of the document size, so that it is in many cases feasible to keep the indexes on secondary storage. In our context, there is an additional reason why we see this as beneficial when possible: the text-indexing operation is 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 name 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.

6 Granularity reduction cost

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

1. Approaches that only consider the version metadata (timestamp) and where it is not necessary to load the document itself. This is the case for the naive-, time-, and periodic-based approaches. The algorithms behind these approaches are also computationally cheap.

2. Approaches 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. Especially in the case of the change-based approach the CPU cost can also be significant (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).

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 approaches into the V2 temporal document database system: the time-, periodic-, and similarity-based approaches. These were chosen because they 1) represent both the categories described above, 2) they have a predictable performance in terms of quality of document version selection, and 3) they are intuitive for the users/administrators of such a system. We will now give some performance results from the granularity reduction of a temporal document database. The period-based approach has the same cost as the time-based approach, and is therefore omitted from the following discussion.

6.1 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 main page. This essentially provides an insert/update/delete trace for our temporal document database.

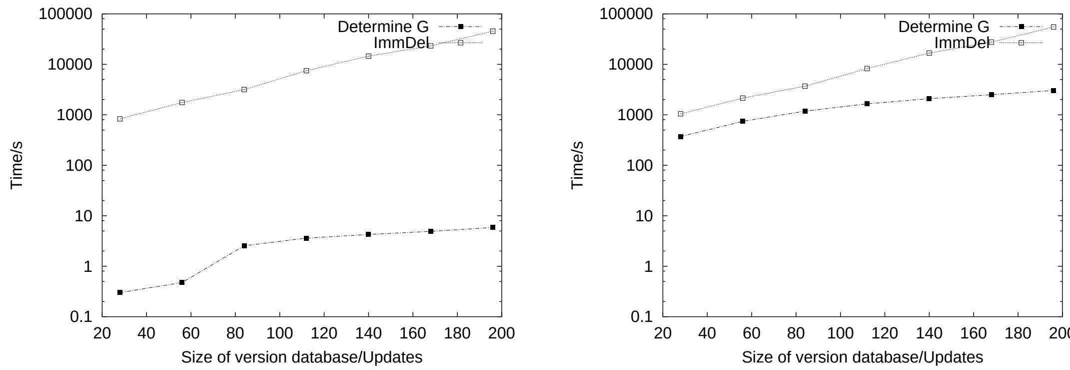
The initial set of pages was of a size of 91 MB (approximately 10000 web pages). An average of approximately 500 web pages were updated each day, approximately 300 web pages were removed (all pages that were successfully retrieved on day d_i but not available at day d_{i+1} were considered deleted), and approximately 350 web 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 contains HTML formatting information), resulting in an average increase of the version database of 45 MB for each set of pages loaded into the database.

6.2 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.

6.3 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 Figure 3 for the time- and similarity-based granularity reduction approaches. The cost of the time- based approach as illustrated in Figure 3a is low because the document versions themselves do not have to be retrieved. When the



(a) Time-based.

(b) Similarity-based.

Figure 3: Cost of determining granularity reduction set G and the cost cost of granularity reduction with immediately delete for the time- and similarity-based granularity reduction approaches. 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.

document versions have to be retrieved, as is the case when using the similarity-based approach, the cost is much higher. This as illustrated in Figure 3b.

The cost of the actual removal of the document versions is much higher than the cost of determining G, as can be seen from Figure 3. Note that the removal cost of the approaches can not be compared to each other based on the numbers on the graphs, because each of the approach resulted in a different number of documents bring removed: The cost of determining G is a function of the total number of historical document versions in the database, the cost of removing the document versions is a function of number of elements in G. It should be noted that the increase in cost with increasing database size (and size of G) in Figure 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 figures will be most relevant.

In the experiments, the resulting database size after performing granularity reduction was approx. 30% when using the time-based approach. For the similarity-based approach the resulting database size after granularity reduction using $d = 0.7$, ranges from 30% (down from 1.2 GB to 0.37 DB) for the small-sized database, to 15% (down from 9.5 GB to 1.4 GB) for the largest sized database.

For both approaches, the removal cost will in most cases be most significant. The exception is of course when only a very small number of document versions compared to the total number of document versions that have to be considered, are to be removed.

It should be noted that 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 name 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.

7 Conclusions and further work

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

We described algorithms for granularity reduction approaches, and discussed advantages and disadvantages of these approaches. We discussed the efficiency of the approaches, 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 approach can be useful for determining individual one-version/non-temporal documents from the system that are candidates for removal.

Acknowledgments

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

References

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

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

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

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

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

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

[7] 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.

[8] K. Nørvåg. Algorithms for temporal query operators in XML databases. In *Proceedings of Workshop on XML-Based Data Management (XMLDM) (in conjunction with EDBT'2002)*, 2002.

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

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

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

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

[13] 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.

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

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

[16] 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.