DyST: Dynamic and Scalable Temporal Text Indexing

Kjetil Nørvåg* and Albert Overskeid Nybø
Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway

Abstract

An increasing number of documents in companies and other organizations are now only available electronically, and exist in several versions updated at different times. In order to provide efficient support for temporal text-containment queries (query for all versions of documents that contained one or more particular words at a particular time) temporal text-indexes are needed. In this paper we present DyST*, a dynamic and scalable temporal text index. The goal of DyST is to provide the same efficiency in terms of search cost and space usage as the previous approaches developed for small and medium document databases, while at the same time providing only logarithmically increasing search cost for very large databases. We present the architecture of DyST and describe how inserts and searches are performed. Based on a prototype we will also present an evaluation of performance based on real-life temporal documents.*

1 Introduction

An increasing amount of documents in companies and other organizations is now only available electronically, and exist in several versions updated at different times. These documents can be in a number of formats like plain text, HTML, XML, Microsoft Word, Adobe PDF, etc. Although many organizations already have searchable repositories or intranet search engines that can be used to retrieve documents based on keywords search, in order to provide efficient support for temporal text-containment queries (query for all versions of documents that contained one or more particular words at a particular time) temporal text-indexes are needed.

A natural consequence of storing multiple versions of documents is increased database size, which in general also implies a larger index size. In previous approaches to temporal text indexing the search cost has increased proportional to index size. While this has been acceptable for medium sized document databases (up to a few gigabytes), this is not acceptable for larger databases. Another desired property is dynamic updates, i.e., all updates from a transaction are persistent as well as immediately available. This contrasts to many other previous systems that only perform bulk-updating of the text index at regular intervals in order to keep the average update cost lower.

In this paper we present *DyST*, a dynamic and scalable temporal text index. The goal of DyST is to provide the same efficiency in terms of search cost and space usage as the previous approaches for small and medium document databases, while at the same time providing only logarithmically increasing search cost for very large databases. This is achieved by having two layers of indexes: an ITTX/ND interval-based temporal text index [12] used as term index, and additional *temporal posting subindexes*(TPI) for frequently occurring terms. In DyST all new entries are first inserted into the ITTX/ND. Only when the posting list (a term and the documents where it occurs) of a term reaches a certain size a TPI is created and the contents of the posting list migrated to the TPI. Subsequent inserts for the term are applied to the ITTX/ND, and the contents of the posting list is moved in batch when the posting list reaches a certain size. In this way the impact of higher insert cost of the TPI is reduced. The fact that TPIs are only created when beneficial also means that the impact of higher space usage of TPI compared to posting lists is reduced. Based on a prototype we will also present an evaluation of performance based on real-life temporal documents. This evaluation also gives new knowledge on the behavior of the Time Index+ [6] which the TPI is based on.

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 present previous approaches to temporal text indexing and Time Index+. In Section 4 we describe the DyST temporal text index. In Section 5 we evaluate the performance of DyST. Finally, in Section 6, we conclude the paper.

*Email of contact author: Kjetil.Norvag@idi.ntnu.no

2 Related work

There has been a large amount of research on indexing temporal data in context of traditional data types, see [13] for an extensive survey. However, the traditional temporal indexing methods are not directly applicable to temporal text indexing. A multidimensional indexing technique, for example an R-tree [4] variant, can be used to index the postings in a 3-dimensional space, with time, word, and document version identifier as the dimensions. However, multidimensional indexes like the R-tree are best suited for indexing data that exhibits a high degree of natural clustering in multiple dimensions. This is not the case for temporal text indexing, and one of the results would be a high degree of overlap in the search regions of the non-leaf nodes. In general, traditional multiversion/temporal indexes are problematic in our context because they assume key/value pairs, but in our context the value is a list where the items have different validity time. Treating a complete list as one version is not feasible

We are only aware of three indexing approaches that directly address the issue of temporal text-indexing: The Anick and Flynn approach [1], V2 [10], and ITTX [9, 12]. In the proposal of Anick and Flynn, the timestamp as version identifier. This is not applicable for transaction-based document processing where all versions created by one transaction should have same timestamp. In order to support temporal text-containment queries, they based the full-text index on bitmaps for terms in current versions, and delta change records to track incremental changes to the index backwards over time. This problem of this approach is the costly recreation of previous states. In the V2 approach each unique term in a document version requires a separate posting in the text index. Although the size of the postings are small and the approach performs well for moderately sized databases and databases with few small-granularity updates, the fact that the size of the text index increases proportional to the size of the document version database may be a problem for databases where the update characteristics are different. ITTX solves the problem of both the Anick and Flynn and the V2 approaches, and will be presented in more detail in Section 3.

Related to the task of temporal full-text indexing is the indexing of temporal XML documents [7]. In that case the focus is on improving path queries. It should be noted that temporal full-text indexes like the ones presented in our paper can also be used to improve performance of temporal XML queries [8]. Also indexing schemes for supporting queries in multiversion XML databases have been proposed [2].

A number of versioned/temporal document databases exist where support for scalable temporal text-indexing would be very rewarding. One project that is particularly interesting because of its size and number of users, is the Internet Archive [5]. The Internet Archive aims at storing the history of web pages that are available at the Internet and supports retrieving pages that were valid at a particular time. However, because of the cost they do not yet support text-containment queries. Similar projects have also been initiated in many countries by the national libraries, and efficient support for temporal text-containment queries would be very welcome in these projects.

3 Background

In order to make this paper self-containing, and provide the context for the rest of this paper, we will in this section give a short overview of the interval-based temporal text index (ITTX) [9, 12] and the Time Index+ [6].

3.1 The ITTX/ND temporal text-index

In a document database with several versions of each document, the size of the text index can be reduced by noting the fact that the difference between consecutive versions of a document is usually small: frequently, a term in one document version will in also occur in the next (as well as the previous) version. Thus, the size of the text index can be reduced by storing term/version-range mappings, instead of storing information about individual versions. In order to benefit from the use of intervals, *document version identifiers* (DVIDs) are used in the ITTX,. Given a version of a document with DVID=v, then the next version of the same document has DVID=v+1. Note that different versions of different documents can have the same DVID, i.e., the DVIDs are not unique between different versions of different documents. In order to uniquely identify (and to retrieve) a particular document version, a *document identifier* (DID) is needed together with the DVID, i.e., a particular document version in the system is identified by (DID||DVID). In this way, consecutive versions of the same document that contain the same term can form a range with no holes.

Conceptually, the text index that use ranges can be viewed as a collection of (w,DID,DVID$_i$,DVID$_j$)-tuples, i.e., a term, a document identifier, and a DVID range. Note that for each document, there can be several tuples for each term w, because terms can appear in one version, disappear in a later version, and then again reappear later. Note that if this term is also included in the next version of this document, the tuple does not have to be modified. This is an important feature, only when a new version of the document that does not contain the term is inserted, the tuple has to be updated. To summarize, in the ITTX as presented in [9] (w,DID,DVID$_i$,DVID$_j$,t_s,t_e) was stored in the index

(where t_s and t_e are the start- and end-timestamps of the interval [DVID$_i$,DVID$_j$>).

It can be noted that the size of each posting period in ITTX as described in [9] is quite large, so we will now present an improved variant that will be used later in this paper. During a temporal text-containment query using the original ITTX, a lookup in the text index returns for each document where the term appears, an interval of versions (DVID,DVID) and a time period (t_1, t_2). In order to determine the actual versions, a separate lookup in the document name index is necessary. The DVIDs can be used to reduce the amount of work during the lookup in the document name index, but are not strictly necessary. It is possible to omit explicit storage of the DVID interval in the index, and instead having a DID together with the time interval. In this variant, which we call *ITTX/ND* [12], logical (w,DID,t_s,t_e) tuples are stored (but note that when a set of tuples have the same term w, w is physically only stored once).

3.2 Time Index+

The Time Index+ [6] (TI+) is an indexing structure that indexes objects identified by an identifier and their associated time intervals (object in this context can be any item identified by an identifier, including documents and document versions). It provides efficient support for temporal queries of the kind "given a timestamp t (or a time range $[t_1, t_2 >$), find the identifiers of all objects valid at that time (or time range). In our context, this can be used to index the valid time of postings, i.e., the interval when a term occurred in a particular document.

In the terminology of TI+, an indexing point is the point of time when an object version interval starts or ends. Logically, TI+ stores for each indexing point the identifiers of all objects valid at that time. The indexing points can be totally ordered and stored in a B-tree, i.e., an entry in the tree is an indexing point and a pointer to a bucket containing the identifiers of all objects valid at the indexing point. In order to reduce redundancy, an incremental scheme is used when physically storing the identifiers. Instead of storing the identifiers of all valid objects in the bucket, this is only done for the first entry t_i of each leaf node. This first entry is called *main indexing point*. The first bucket is called a continuous bucket (SC) and contains the identifiers of objects valid at the previous indexing point (in the left neighbor node) that are still valid. For the other indexing points t_j in a leaf node, a plus bucket (SP) and a minus bucket (SM) is maintained. The SP contains identifiers of objects with start time t_j, and the SM contains identifiers of objects with end time t_j. Thus, a logical bucket LB_j for time t_j is equal to $LB_j = (SC \cup (SP_i \cup ... \cup SP_j)) - (SM_i \cup ... \cup SM_j)$.

The aspects of TI+ described so far are essentially those of the original Time Index [3], the predecessor of TI+. In order to reduce the redundancy in the original Time Index, TI+ employs three different kinds of continuous buckets: shared buckets (SCS) and exclusive buckets (SCE) at leaf node level, and continuation buckets (SCI) at internal node level. Shared buckets (SCS) and exclusive buckets (SCE) are used instead of the original SC. An SCS is shared between an odd-even[1] pair of leaf nodes, and each leaf node has a separate SCE. Essentially, the SCS contains the identifiers of objects valid through an interval longer than the time interval covered by the pair of leaf nodes, while an SCE contains 1) the entries for objects valid at the previous indexing point and that are still valid (in the case of an odd node) or 2) the entries valid at the previous indexing point and still are valid but was not valid at the first indexing point in the node to the left (even node). The use of SCS and SCE reduces redundancy to approximately the half, but the most important factor in reducing redundancy in TI+ is the introduction of the SCIs. An SCI is associate with the root of a subtree, and contains the identifiers of objects whose validity time spans the subtree.

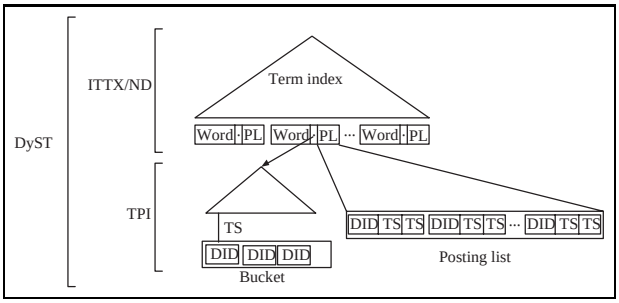

Figure 1. High-level overview of the DyST index architecture.

4 The DyST temporal text index

The indexes presented in the previous section can perform well for moderately large document databases, up to a few gigabytes. However, because the whole posting lists have to be read during a search, the search cost increases approximately linearly with the database size. This gives unacceptable high cost in the case of very large databases, and in order to solve this problem we developed the DyST temporal text index. The goal of DyST is to provide the same efficiency in terms of search cost and space usage as the previously presented indexes for small and medium document databases, while at the same time providing logarithmically increasing search cost for very large databases. This goal is achieved by having two layers of indexes: an ITTX/ND index indexing medium and infrequently occurring terms, and additional temporal posting subindexes (TPI) for the frequently occurring terms (see Figure 1).

[1] Odd/even based on numbering leaf nodes staring with the first/left.

In the DyST a new entry is first inserted into the ITTX/ND, and only when the posting list of a term reaches a certain size a TPI is created and the contents of the posting list migrated to the TPI. Subsequent inserts for the term are performed to the ITTX/ND, and the contents of the posting list is migrated in one batch operation when the posting list reaches a certain size. In this way the impact of higher insert cost of the TPI is reduced. The fact that TPIs are only created when beneficial also reduces the impact of higher space usage of TPI compared to traditional posting-list storage.

We will now give a more detailed description of DyST, giving a detailed description of TPI and the interaction between TPIs and the top-level ITTX/ND index.

4.1 Basic TPI

TPI is based on the Time Index+ [6] (see Section 3.2). For the sake of presentation, we will in the following distinguish between the *TPI search tree* and *the associated buckets*, even though buckets will in general be stored together with their associated nodes. We use the term *node* exclusively to mean the search tree structures containing the key/pointer(s). The TPI search tree is a monotonic B+-tree variant where all leaf nodes except the one currently being inserted into are full, and the tree is not necessarily completely balanced on right side.

In search trees used for traditional indexing, the size of a node is usually a multiple of operating system or disk pages, i.e., 4 KB or more. There is a tradeoff between having as many key/pointer pairs on the pages in order to have large fan-out/few levels in tree, versus locality of accesses on disk pages and average access cost. In the context of TPI these aspects are still important, but there is also the issue of being able to represent as many identifiers as possible in the SCIs, thus reducing the number of identifiers redundantly stored in a number of SCSs. Using small nodes, which effectively results in subtrees covering shorter intervals, increases the number of identifiers that can be stored in the SCIs. This aspect is clearly contradictory with the goal of large fan-out, and a tradeoff has to be done here as well.

We have found that it is in general beneficial to use smaller nodes than in other indexes, but because a node is stored together with its associated buckets, the size of the total unit to be stored will still be high enough to justify page-based storage.

The contents of the TPI must be compatible with the contents of the index on the layer above, i.e., the ITTX/ND. A posting in the ITTX/ND contains a document identifier (DID) and a time interval (the size of the timestamps in the DyST indexes is 4 byte). The posting itself does not tell directly which document version that contains the term in the given interval; in order to determine this a lookup in the document name index is performed. The DID is also what is stored in the TPI. Although the lookup in the document name index could be avoided by storing the DVID together with the DID, this would approximately double the index size, and increase the total cost.

4.2 Inserting posting lists

Efficient storage and management of the TPI can be challenging. We will now outline some of the details on how this is done.

The size of the TPI nodes and buckets can vary a lot, from just a fraction of a disk page to a number of disk pages. For this reason we considered using record-oriented storage of the structures instead of page-based. However, the TPI buckets are always stored together with their associated nodes, and as the results in Section 5 will show this gives a total size that will be larger than disk pages. The efficiency of page-based storage outweighs the amount of internal fragmentation on pages that will occur. The only exception is internal nodes without SCI buckets attached. These nodes, which have constant size, will be stored on sub-pages.

The DyST index structure is as mentioned two-layered with an ITTX/ND on top of a number of TPI indexes, and inserts are always performed on the ITTX/ND. Only when the size of a posting list is over a certain threshold $N_{createTPI}$ a TPI index is created for the term, and the postings migrated to the TPI. Subsequent inserts are again done to the ITTX/ND, and when the number of postings for the term reaches a threshold $N_{migrateTPI}$ those postings are migrated.

For small and medium-sized posting lists the ITTX/ND approach is most efficient because the TPI has higher space usage. However, the whole posting list has to be read during a search, so that when the posting list reaches a certain size a tree-based structure like the TPI becomes more efficient. This crossover point determines the values of $N_{createTPI}$ and $N_{migrateTPI}$. In practice, $N_{createTPI}$ will be larger than $N_{migrateTPI}$. The reason is that when the added update cost due to the use of TPI is also taken into account, using posting lists is also beneficial a bit beyond the crossover point.

4.2.1 TPI creation

The size of a posting list at the time of TPI creation will be small compared to available main memory, so the process of TPI creation is easy and efficient. When the TPI is first created for an index term the whole posting list in the ITTX/ND can be read into main memory (and removed from the ITTX/ND), a TPI tree is created, and the resulting tree is written to disk. The nodes are written to disk together with their associated buckets. The nodes are writ-

ten in an order that minimizes subsequent search and update cost, this will be discussed in more detail below.

4.2.2 TPI update

Update of the TPI will be performed in batch, a large number of postings are migrated from the ITTX/ND to the term's TPI. When updating a TPI, one of the two following approaches can be used, depending on the size of the TPI.

When the TPI is small, the most efficient approach is to simply read the whole TPI from disk, recreate it in main memory, and write it back. By clustering relevant parts of the TPI together, the cost of future TPI searches can be reduced.

When updating TPIs over a certain size only the relevant parts are read into memory. Because the TPI is an append-only index (a new posting always has a timestamp that is equal or larger than the previous one) all inserts are applied to the right hand part of the index. Thus, only the right hand path is needed during update, as illustrated in Figure 2 (left). This also reduces the amount of main memory needed in the update process. If the rightmost leaf node is not yet full some of the posting information is inserted into this node, but because of the relatively large amount of postings that are migrated to the TPI in one operation, more leaf nodes have to be created, as well as more internal nodes. This is illustrated in Figure 2 (middle), which shows the contents in main memory after the new nodes have been created.

When subtrees are full so that SCIs can be created, the children of the subtree roots has to be read (illustrated with vertical hatches in Figure 2 (right)). Some of the intervals in these children can be concatenated and moved to the parent node, and in that case those child nodes will have to be updated on disk.

As Figure 2 (right) shows, the strategy as described above requires that a certain number of nodes and their buckets fit in main memory. The number of nodes is essentially controlled by the fan out and level of the tree. The size of the buckets however, can be large in the case of a large number of events at each indexing point. If this occurs, the problem can be alleviated by only using SCI on the lowest level(s) of internal nodes. As will be shown in Section 5, using SCIs on the lowest level(s) only does not significantly increase the TPI size.

5 Evaluation

In the previous sections we have described the structure of DyST and strategies for maintaining and storing data in the index. Now, the question is how this structure will perform when applied to real data: a temporal document collection based on versions of web pages from a number of sites that were collected once a day during a period of several months [10]. The properties of the ITTX/ND on the top level is well-known from earlier studies[12], so we will now concentrate our efforts on a study of the TPI part of the index. We have implemented a main-memory version of TPI which gives us the possibility of studying the resulting size of the TPI, suitable node sizes, the use of internal continuation buckets (SCI), and the resulting access cost. This also makes it possible to determine when the TPI should be employed for a term instead of storing all postings of a term in the ITTX.

Our study will be on individual posting lists, i.e., on one TPI-tree at a time. The contents of these posting lists are based on indexing a temporal document collection that have been created by retrieving the contents of a set of web sites at regular intervals. The collection is described in more detail in [10].

We perform the study on posting lists resulting from different terms. The performance of the TPI depends very much on the occurence statistics of the term it indexes, so in our study we have used a selection of terms with different frequency in the document collection. Due to space constraints we will in this paper only give the results for very frequently occurring terms, for moderately frequent and infrequently occurring terms we refer to the extended version of this paper [11]. The posting list of frequently occurring term contains a total of 43139 posting intervals (of which 27% are without end timestamp, i.e., current terms).

Different time granularities and transaction sizes can affect the indexing on the data. In one extreme, each document inserted into the document database will be in a separate transaction and given a unique timestamp. In another extreme, all documents will belong to the same transaction. The first extreme can make sense, but the second case makes little sense in the context of a temporal document database: if all items stored in the database has the same timestamp the temporal aspect disappears.

Regarding transaction sizes, in the case of ITTX, each interval will be stored as one posting, so this is not an issue in this case. In the context of TPI however, the first extreme case will give a large search tree as result, as well as a large number of plus- and minus buckets with only one entry. However, this does not necessarily make the TPI a bad alternative compared to ITTX/ND. The second extreme can actually be a good case for TPI. The TPI search tree will be small, but the bucket for the one indexing point will be very large. However, the DIDs in the bucket can be efficiently coded if the typical difference between the DIDs is not large, making it possible to store them using a minimum of space. The ITTX can not benefit from several events happening at same time, while TPI will be more efficient when this is the case to a large degree.

We expect most applications to have a pattern some-

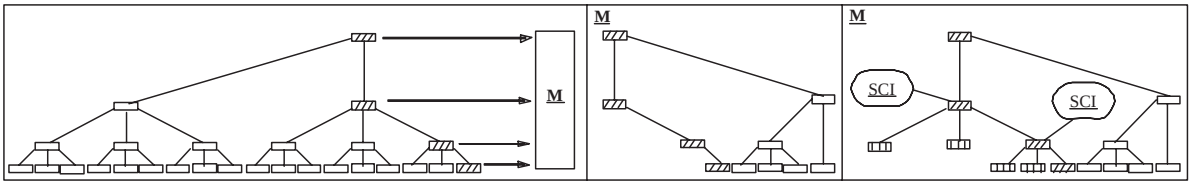

Figure 2. Left: Retrieving right hand path of TPI (hatched nodes). Middle: Nodes in main memory after creating new nodes, hatched nodes are the ones retrieved from disk, the others are new nodes. Right: Nodes in main memory after creating SCI, vertical hatched nodes denotes child nodes that have to be read.

Value of k:	5	10	20	30	40	50
Internal nodes:	68	118	218	318	418	518
Leaf nodes:	54	94	174	254	334	414

Table 1. Size of nodes in TPI search tree.

where between the two extremes outlined above, and we will in this study concentrate on two cases in between the extremes, with (on average) a moderate number of events at each indexing point. We will from now on denote the two test sets as *G3*, having 22696 indexing points, and *G4*, having 4677 indexing points (4677 indexing point means on average 16 events for each indexing point, i.e., interval starts or ends)

We will now study space usage, redundancy, update- and search cost for the selected terms using the TPI. During the study, we have also measured the effect of alternative ways of using SCI buckets: 1) no use, 2) use when applicable only on 1st level above leaf nodes, and 3) use when applicable on all levels.

As described in Section 4.1, nodes in the TPI will be relatively small. In the following, we will study characteristics for different node sizes, given as the node order k (number of indexing points in this case). The size of the nodes with the coding we use in the TPI search tree, excluding contents of buckets, is summarized in Table 1.

5.1 Space usage

Because of the redundancy in the TPI, it can be expected that it will use a larger amount of space than simple posting lists as in the ITTX/ND. We will now study how much the space usage increases.

First of all, in order to be able to compare the approaches we have to calculate the space usage of the posting list in the ITTX/ND. On average 12 byte are needed to represent each posting interval, and the list contains a total of 43139 posting intervals, resulting in a total of 517668 byte used for the whole posting list. Note that the truncation of timestamps only reduces the number of distinct timestamps/indexing points, it does not affect the number of actual intervals.

When using a TPI, the space usage of the search tree will in general be small compared to the space usage of the SM/SP/SCE/SCS/SCI buckets. Figure 3 shows the number of nodes and the corresponding space usage of the nodes. For example, for node size $k = 10$, the space usage of search tree for test set G4 is only 49656 byte. However, 528325 DIDs are stored in the buckets, occupying 2113300 byte. Thus, in total TPI requires 2162956 byte. This is about 4 times as much as the space usage for the ITTX/ND approach.

5.2 Redundancy

A posting list as the one used in ITTX/ND does not contain redundancy, each interval is only represented once. In order to get an idea of the redundancy when using a TPI, the number of stored DIDs is illustrated in Figure 4. The number of posting intervals using the ITTX/ND is included for comparison. The reason for the rough shape of the graphs deserves a couple of comments. For example, at $k = 25$ of test set G3, the number of leaf nodes is 908, so that 3 levels are needed in the tree. The first 26^2 of the nodes form a subtree where SCIs can be employed. However, because SCIs can only be employed for full subtrees, they are not applicable to the rest of the nodes, increasing the redundancy. With $k = 20$ on the other hand, several SCIs at level 1 and also one at level 2 is possible. The result is that more identifiers can be moved up to SCIs, thus reducing the redundancy.

It is also interesting to note that the redundancy (number of DIDs stored) does not change much between different values of k, in average about 490000 for test set G4, and 790000 for test set G3 (for comparison, the number was about 900000 for the original interval set with no truncation). For test set G4, the average number of duplicates of each identifier was 10, with the lowest value of 8 duplicates in the case of $k = 30$.

5.3 Search cost

In the prototype snapshot search and range search has been implemented. We will now present results using test set G4 and employing SCI on all levels. We will also do some comparison with the performance of similar searches employing a posting interval approach as in ITTX/ND.

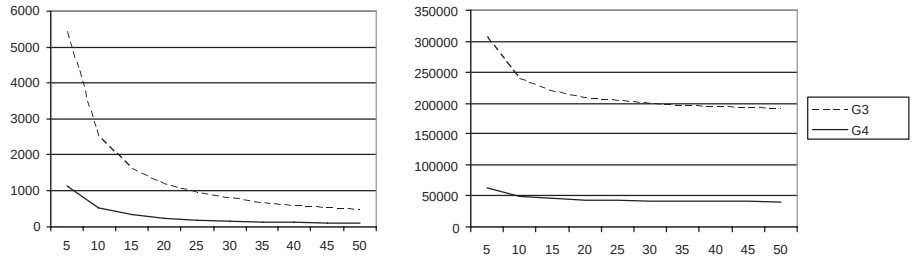

Figure 3. Number of nodes (left) and total space usage (right) for the TPI search tree.

We found that on average, 10283 DIDs had to be read (with standard deviation of 58) when processing a snapshot query. The number was relatively independent of node size. For any node size, it is necessary to read all continuous buckets on the path from the root and to the actual leaf node in order to find all identifiers representing intervals valid on the time of the main indexing point. There is also little difference between using SCI or not. If SCI is not employed, the result is simply that more identifiers have to be read from the SCS/SCE buckets. In the case of small nodes, more identifiers will be in the internal nodes and fewer in the leaf nodes, in the case of larger node sizes, fewer identifiers will be in internal buckets. Still, in the case of large nodes some more identifiers will be read from the continuous buckets. The reason is that more identifiers will be in the continuous bucket and later cancelled out by being in a minus bucket (and hence shall not be in the result set).

Although approximately the same amount of identifiers have to be read from the continuous buckets, the number of incremental buckets (SP/SM) that has to be read differs with different node sizes. In the case of small nodes, only a few incremental buckets will have to be read, and reading of continuous buckets will dominate. In the case of large buckets, in general more indexing points will have to be read in order to reconstruct the logical bucket of an indexing point covered by the leaf node. It should also be mentioned that in general there will be a few more entries in the SP compared to the SM. The reason is that some intervals are not finished yet, the information is still current. For example, in the case of test set G4, each SP contained on average 9 identifiers, and the SMs contained on average 7 identifiers. With node size $k = 50$, this means reading on average 25 indexing points, including 25 SMs and 25 SPs, in total approximately 400 identifiers.

Assuming a physical disk access is needed for each node that has to be read on the path from the root to the leaf node, the height of the tree and the size of the nodes (see Table 1) are important factors affecting the cost of a search. However, since associated buckets are stored together with the nodes (SCI in the case of internal nodes, and SCS/SCE/SP/SM in the case of leaf nodes) and have to be accessed, the node size is less important than the total size of node and associated buckets. This is illustrated on Figure 5, which shows the average size of nodes and associated buckets.

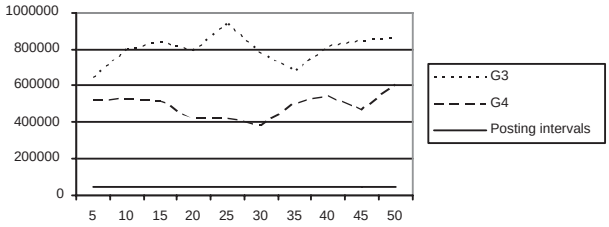

Figure 4. Number of DIDs stored in the TPI.

Using preorder depth-first placement, leaf nodes are always stored sequentially after their parent. If all child nodes are read in the same operation as the parent one disk access is necessary in order to perform the last step in the search. For a node size of $k = 5$, one internal node and 6 leaf nodes will on average have a total size of about 12 KB. In the case of a page size of 4 KB this means that a minimum of 3 consecutive nodes have to be read from disk in addition to the parent node. For a node size of $k = 10$, 50 KB has to be read, and for $k = 20$ about 143 KB. This implies that this strategy is only beneficial when a small node size is used, in other cases it will be better to read the parent and actual leaf node in two separate physical disk operations (however, it should be noted that in many cases the child node will already have been read because the disk and/or file system employs read-ahead). Employing a small node size in order to be able to perform aggressive read-ahead as described is not beneficial in general, because this increases the height of the tree, instead moving the problem to accessing separate internal nodes.

The total cost of performing a snapshot search in a TPI tree is one disk access for each node along the search path.

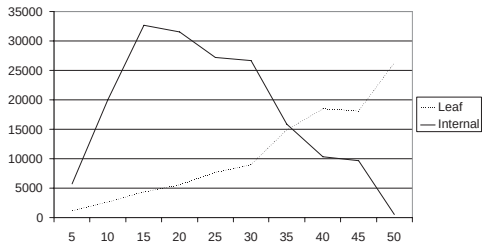

Figure 5. Average size of nodes and their associated buckets.

In the worst case with $k = 15$ this means 4 accesses, each of approximately 30 KB, a total of 120 KB. On a typical modern disk this would take about 40 ms. If a ITTX/ND with posting list was used instead, the posting list would have a size of approximately 505 KB. In the extreme best case, which will only happen if the index has been reorganized so that nodes in the tree are stored in order and are full (the index is B-tree based, meaning that in a dynamic setting the fill factor will typically be 67% or less), the retrieval time would be around 20 ms. However, it is more likely that the posting list is fragmented over a large number of positions on the disk, up the worst case where a disk seek would be necessary for each page (63 disk seeks in this example). The conclusion is that in general a search using a TPI will be much faster than when having to access a large posting list.

A range search is performed by first performing a snapshot search to the start of the range, and then continue the search until the end of the range, adding identifiers from the SPs to the result set. In the case of large ranges, several leaf nodes will have to be read. Except in the case of very long ranges, the number of nodes to be read will be larger than in the case of the posting list.

6 Conclusions

The importance of temporal text-indexing techniques is increasing as the ability to manage timestamped or temporal documents becomes common. In order to be of practical use, such indexes need to have a space usage and search cost that increases less than linearly with the database size. In this paper we have presented DyST, a dynamic and scalable temporal text index that satisfies these properties. We have described the management and interaction of the main ITTX/ND index and the TPI subindexes, and studied the performance of the DySST using a temporal document collection.

From our evaluation study we saw that as expected, TPI has a higher space usage than a posting list approach, requiring up to 4 times as much space. However, given the reduced cost of snapshot search this will in many application areas be an acceptable cost: for our relatively small test database the cost of ITTX/ND is up to 16 times higher than DySST, and because DySST has only logarithmically increasing search cost this factor will only increase with larger database sizes. The cost of most range searches can also be expected to be considerably reduced, because fewer disk operations have to be performed. Also, TPI will only be used for the most frequently occurring terms, so that the total size of the index structure will be much smaller than 4 times the ITTX alternative.

Future work includes 1) a more detailed study on issues related to fine-granularity main-memory buffering of parts of posting lists, and 2) management of temporal documents and indexes in parallel and distributed document databases.

References

[1] P. G. Anick and R. A. Flynn. Versioning a full-text information retrieval system. In *Proceedings of SIGIR'1992*, 1992.

[2] S.-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Efficient complex query support for multiversion XML documents. In *Proceedings of EDBT'2002*, 2002.

[3] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim. The time index: An access structure for temporal data. In *Proceedings of VLDB'1990*, 1990.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, June 1984.

[5] Internet Archive. `http://archive.org/`.

[6] V. Kouramajian, I. Kamel, R. Elmasri, and S. Waheed. The Time Index+: an incremental access structure for temporal databases. In *Proceedings CIKM'1994*, 1994.

[7] A. O. Mendelzon, F. Rizzolo, and A. A. Vaisman. Indexing temporal XML documents. In *Proceedings of VLDB'2004*, 2004.

[8] K. Nørvåg. Algorithms for temporal query operators in XML databases. In *Proceedings of XMLDM'2002*, 2002.

[9] K. Nørvåg. Space-efficient support for temporal text indexing in a document archive context. In *Proceedings of ECDL'2003*, 2003.

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

[11] K. Nørvåg and A. O. Nybø. DySST: dynamic and scalable temporal text indexing. Technical Report IDI 10/2004, NTNU, 2004. Available from `http://www.idi.ntnu.no/grupper/db/`.

[12] K. Nørvåg and A. O. Nybø. Improving space-efficiency in temporal text-indexing. In *Proceedings of DASFAA'2005*, 2005.

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