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 amount of documents in companies and other organizations is 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 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.

Keywords: Temporal databases, document databases, text indexing, temporal indexes

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 in order to keep the average update cost lower only perform bulk-updating of the text index at regular intervals.

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 logarithmically increasing

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

search cost for very large databases. This is achieved by having two layers of indexes: an ITTX/ND index 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 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 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+ index[4] 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 define the data model behind our document database and indexing, and present text-indexing terminology. In Section 4 we present previous approaches to temporal text indexing. In Section 5 we describe the DyST temporal text index. In Section 6 we evaluate the performance of DyST. Finally, in Section 7, we conclude the paper.

2 Related work

There has been a large amount of research on indexing temporal data in context of traditional data types, see [12] for an extensive survey. However, as explained in detail in [9], the traditional temporal indexing methods are not directly applicable to temporal text indexing.

We are only aware of three indexing approaches that directly address the issue of temporal text-indexing: The Anick and Flynn approach [1], V2 [9], and ITTX [7] and the improvements presented in [11]. These approaches will be presented in more detail in Section 4.

Related to the task of temporal full-text indexing is the indexing of temporal XML documents [5]. 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, and this is described in more detail in [6].

The inverted file indexes used as basis in our work is based on traditional text-indexing techniques, see for example [14].

For measuring various aspects of performance in text-related contexts, a number of document collections exists. The most well-know example is probably the TREC collections [13], which includes text from newspapers as well as web pages. Another example is the INEX collection [3] which contains 12000 articles from IEEE transactions and magazines in XML format. We are not aware of any temporal document collections suitable for our purpose, so in our experiments we used collections made from periodically crawling a number of web sites, and temporal document collections created by our TDocGen synthetic temporal document generator [10].

3 Preliminaries

In this section we describe document and time models, and present the terminology used for text indexing.

3.1 Document and time models

In our data model we distinguish between documents and document versions. A temporal document collection is a set of document versions $V_0...V_n$, where each document version V_i is one particular version of a document D_i. Each document version was created at a particular time T (more than one version of different documents can have been created at T), and we denote the time of creation of document version V_i as T_i. A particular document version is uniquely identified by the combination of document name N_j and T_i.

Documents consist of a number of terms (words) w_i, where w_i is an element in the vocabulary set W, i.e., $w_i \in W$. There can be more than one occurrence of a particular term in a document version.

When indexing documents, it is possible to either view documents as 1) an ordered list $V = [w_0, w_1, ..., w_k]$ and index terms and their position in the document, or 2) as a set of unique terms $V = \{w_0, w_1, ..., w_k\}$ and index a particular term only once for each document. Although the list view makes more powerful searches possible, it also results in much higher space usage for the indexes. In order to keep the space usage as low as possible we will in this paper use the set view, i.e., only index unique terms. However, if a list view is desired, this can be achieved by storing a list of positions together with the document/version identifier in the indexes.

The time model is a linear (non-branching) time model, time advances from the past to the future in an ordered step-by-step fashion. The time axis can be viewed as an ordered sequence of instants, where at some instants an event happens. In the context of this paper, an event is a document creation, deletion, or update. More than one event can happen at a particular time instant.

A document version exists (is valid) from the time it is created (either by creation of a new document or update of the previous version of the document) and until it is updated (a new version of the document is created) or the document is deleted.

3.2 Basic temporal text-indexing techniques

The basic lookup operation in non-temporal text indexing is to retrieve the document identifiers of all documents that contain a particular term w. The most common access method for text indexing is the inverted file, which is also the basis of our approaches.

An inverted file index is a mapping from a term w to the documents $D_1, D_2, \ldots, D_j$ where the term appears. Inverted files are also the basis of our approaches. In the inverted file index, a *posting list* $PL = (w, D_1, D_2, \ldots, D_m)$ is created for each index term, where w is the term, and D_i are the document identifiers of the documents the term appears in. The tuple $P = (w, D_i)$, i.e., an index term and a document identifier, is called a *posting*.

In the case of temporal text indexing, a posting conceptually consists of $P = (w, D_i, t_s, t_e)$, i.e., an index term, a document identifier, and the start- and end-timestamp of the period in which the document D_i contained the term w.

4 Previous approaches to temporal/versioned text indexing

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 V2 index (V2) [8, 9], the interval-based temporal text index (ITTX) variants [7, 11], and the index proposed by Anick and Flynn index [1].

4.1 The V2 temporal text-index

A document version stored in V2 is uniquely identified by a *version identifier* (VID), and stored in a B-tree-based document-version index. 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 index is append-only and always compact. A document is identified by a *document name*. A *document name index* is used to provide mapping from document name to the VID and timestamps of its versions.

The terms in the document versions are indexed by variants of inverted lists, which essentially provides a mapping from a term to the VIDs of all document versions containing the term. One very important aspect about the inverted list variant used in V2 is that the VID can be coded very efficiently, so that the average number of bytes for each posting is close to 2. In order to support efficient temporal text-containment queries, a separate index called *VIDPI* is employed. The VIDPI provides the mapping from VID to validity period (start- and end-timestamp), which is the timestamp of the document version identified by VID and the timestamp of the next version (or time of deletion) of the particular document.

Temporal text-containment queries using the VIDPI-index-based approach can be performed by the following two-step algorithm:

1. A text-index query using the text index that indexes all versions in the database. The result is a set of VIDs of all document versions containing the particular term.

2. A time-select operation selects the actual versions (from stage 1) that were valid at the particular time or time period. For this purpose the VIDPI is used. One lookup is needed for each of the VIDs returned in stage 1.

This algorithm generally performs very well, because even for large document databases the VIDPI index can be assumed to be resident in main-memory.

4.2 The ITTX/ND temporal text-index

One problem with the V2 index is that each unique term in a document version requires a separate posting in the text index (although as mentioned above, the size of the posting is small). This makes the size of the text index proportional to the size of the document version database. 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, we can reduce the size of the text index 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, instead of the version identifiers used in the V2 index. Given a version of a document with DVID=v, then the next version of the same document has DVID=v+1. In contrast to a VID that uniquely identifies a document version stored in the system as was the case in V2, 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.

When a new document version with DVID$_i$ is inserted, and it contains a term that did not occur in the previous version, a (w,DID,DVID$_i$,DVID$_j$) tuple is inserted into the index. DVID$_i$ is the DVID of the inserted version, but DVID$_j$ is set to a special value UC (until changed). In this way, 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 (a similar technique for avoiding text index updates is also described in [1]). Only when a new version of the document that does not contain the term is inserted, the tuple has to be updated. One of the main reasons why the VIDPI was very attractive in the context of V2, is that storing the time information in the VIDPI is much more space efficient than storing the timestamps replicated many places in the text index (once for each term). However, when intervals are used, one timestamp for each start- and end-point of the intervals is sufficient, and the increase in total space usage, compared with using a VIDPI index, should be less than what is the case in V2. It could also be more scalable, because the V2 approach is most efficient when the VIDPI index can always be resident in main memory. To summarize, in the ITTX as presented in [7] (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$_j$>).

During a temporal text-containment query using the implemented version of 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, t). In order to determine the actual versions, a separate lookup in the document name index is necessary. The DVDIs 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, called *ITTX/ND* [11], 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).

4.3 Anick and Flynn approach

In addition to our indexing approaches, i.e., the V2 index, ITTX, and ITTX/ND, the only research work we are aware of that directly focuses on access methods for general temporal document querying is the proposal from Anick and Flynn [1] on how to support versioning in a full-text information retrieval system. In their proposal, the current version of documents are stored as complete versions, and backward deltas are used for historical versions. This gives efficient access to the current (and recent) versions, but costly access to older versions. They also use 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 approach has the same advantage and problem as the delta-based version storage: efficient access to current version, but costly recreation of previous states is needed. It is also difficult to make temporal zig-zag joins (needed for multi-term temporal text-containment queries) efficient.

5 The DyST temporal text index

The V2 index and the ITTX variants described earlier in this paper has proven to 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

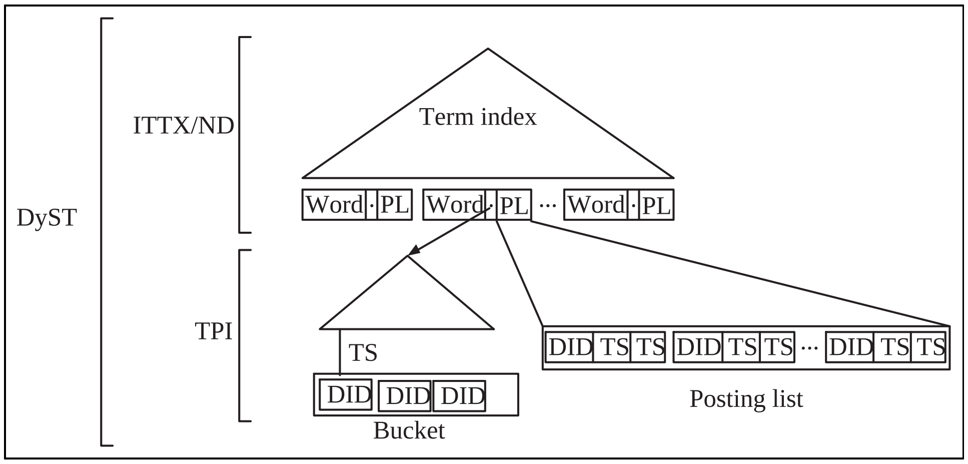


Figure 1: High-level overview of the DyST index architecture.

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 V2 and ITTX 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).

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. TPI is based on the Time Index+ [4], and we start with a brief introduction to Time Index+, followed by a more detailed description of TPI and the interaction between TPIs and the top-level ITTX/ND index.

5.1 Time Index+

The Time Index+ [4] 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 continous 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 [2], the predecessor of TI+. In order to reduce the redundancy in the original Time Index, TI+ employs three different kinds of continous 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 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.

For a more detailed description of TI+ we refer to [4].

Example. In order to show how data is stored in TI and TI+ structures, we will now illustrate the result after inserting the 7 intervals illustrated in Figure 2(a).

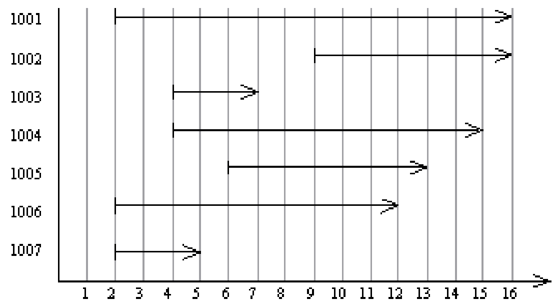
Figure 2(b) shows the intervals inserted into the original TI. As is evident, the SCs contain relatively many identifiers. Several identifiers are also duplicated, e.g., 1001 are stored in 5 different buckets, 1006 and 1004 in 3 buckets, and 1005 in two buckets.

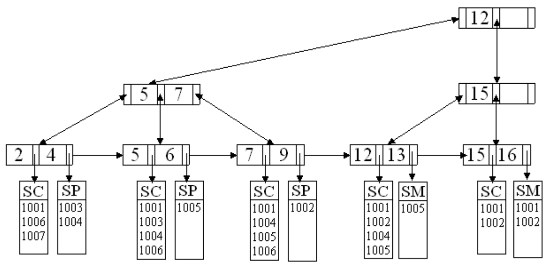
The TI+ introduced shared SCs (SCS) in order to reduce redundancy. The effect of the SCSs is illustrated in Figure 2(c), for example, the identifier 1001 is now only stored in 4 buckets, down from 6 buckets in TI. Note that in TI the end timestamp did not have to be explicitly stored when it was equal to the main indexing point, by omitting it from the SC we knew it was not valid anymore. However, in TI+ it is necessary to store these end timestamps as well, in SMs associated with the main indexing points as illustrated in Figure 2(c). Thus, for some identifiers like 1003 and 1007, we do not reduce redundancy compared to the TI. This problem mainly occurs in this example because of relatively short intervals, and in a more realistic application the positive effect of the shared buckets will be larger. In the case of a larger tree covering a higher number and longer intervals, the size of the continous buckets will increase from left to right of the tree.

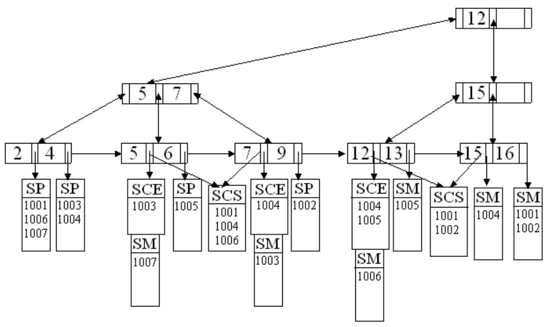
The real gain from using TI+ comes with the use of internal SCs. As can be noted in Figure 2(c), there are identifiers covering intervals larger than one subtree (1001 and 1006). These identifiers can be stored in an SCI instead of in all the leaf nodes of the actual subtree. This is illustrated in Figure 2(d).

It should be emphasized that this is a very simple example with relatively small buckets and short intervals, and that in practice the space usage reduction from using SCS and SCIs will be much larger. The improvements will be studied in more detail in Section 6.

In order to illustrate a search in the TI+ in Figure 2(d), assume that we want to know the identifiers of all objects valid at time $t = 10$. We then follow the path from the root down to the largest indexing


(a) Intervals.


(b) TI


(c) TI+ employing SCE and SCS.

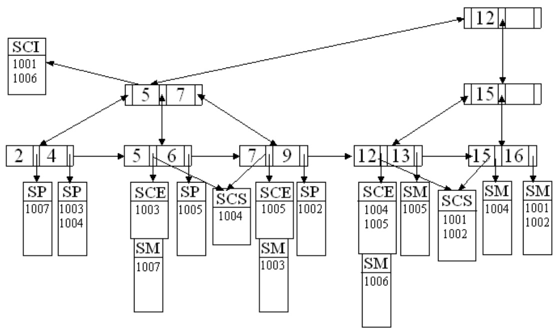

(d) TI+ also employing SCI.

Figure 2: Example of storing intervals in TI/TI+ structures.

point that is equal to or smaller than t. While traversing the tree, identifiers in SCIs are also collected and stored in the result set R. So, entering the first internal node, we set $R = \{1001, 1006\}$. Entering the leaf node, we insert the contents in the SCS and the SCE, giving $R = \{1001, 1006, 1004, 1005\}$. For each indexing point in the leaf node, as long as the indexing point is smaller than or equal to t, we add identifiers in the SP buckets to R, while removing identifiers in SM buckets from R. In this example, 1002 is added, giving $R = \{1001, 1002, 1004, 1005, 1006\}$ as result. The correctness of this result can be verified from Figure 2(a).

5.2 Basic TPI

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 (see Section 4.1) 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 (see Section 4.1) together with the DID, this would approximately double the index size, and increase the total cost.

5.3 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 6 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.

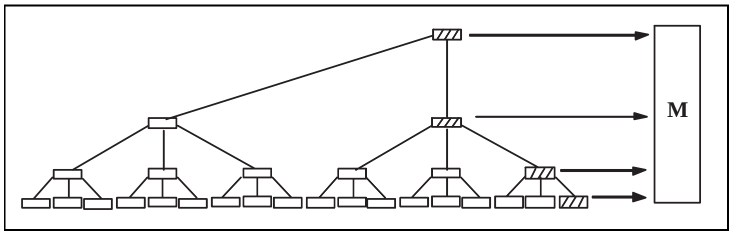


Figure 3: Retrieving right hand path of TPI (hatched nodes).

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.

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

5.3.2 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 3. 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

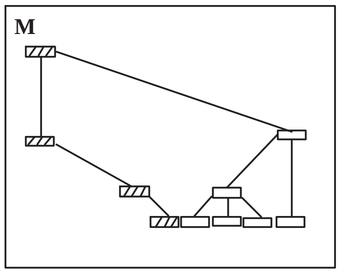


Figure 4: Nodes in main memory after creating new nodes. Hatch nodes are the ones retrieved from disk, the others are new nodes.

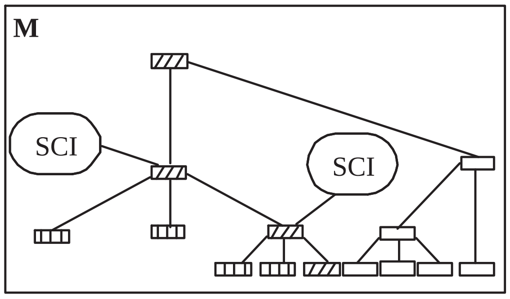


Figure 5: Nodes in main memory after creating SCI. Vertical hatched nodes denotes child nodes that have to be read.

is illustrated in Figure 4, 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 5). 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 5 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 6, using SCIs on the lowest level(s) only does not significantly increase the TPI size.

5.4 Layout on disk

The layout of TPI nodes and buckets on disk is important because it partially determines search and update cost. The problem has two aspects: 1) layout of buckets associated with a node, and 2) layout of nodes (with their buckets) with respect to other nodes (and their buckets).

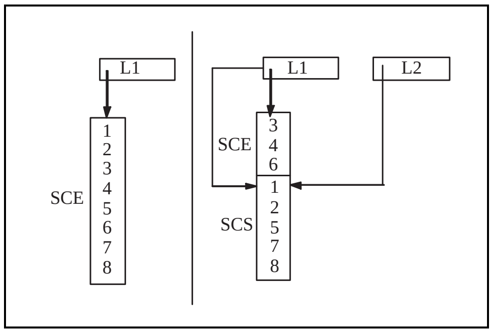


Figure 6: Management of SCE and SCS nodes.

5.4.1 Node/bucket layout

A node will be stored on disk followed by its buckets. Continuation buckets will always be needed during a search, and the SCI and SCE buckets are stored following the node they are associated to. SCS buckets belong to two nodes (an odd-even pair), but have to be stored together with only one of them. In order to make updates easier, an SCS bucket is stored together with the odd node. The reason, as illustrated in Figure 6, is that when a SCS is created, it is essentially a subset of the original SCE bucket at the odd node. In this way, storage space management is easier.

In the case of leaf nodes there will also be incremental buckets (SP and SM). These will be stored following the continuation buckets. During a snapshot search (i.e., search for data valid at a particular time) on average half of these buckets will be needed and have to be read. For typical data sets, the size of a node and its buckets will be small enough to make it most efficient to read it all in one operation. However, data sets with many events for each timestamp results in large buckets, and in that case it is an alternative to only read the buckets that are needed.

The SCI buckets of the internal nodes poses some challenges. When an internal node is first created and stored, it does not in general have an SCI attached. These internal nodes will be relatively small. n order to avoid wasting space, sub-page storage is used for internal nodes without SCI. Only when the subtree rooted at the internal node is full will a SCI bucket is created. It should be noted that the size of an SCI bucket will decrease when SCIs on ancestor nodes are created.

5.4.2 Node layout

Layout of nodes on disk is a challenge for all tree-based index structures. In our index we have the advantage of append-only inserts performed in batch. Our preferred layout is insert-sorted layout, as illustrated in Figure 7. Using this layout writing of nodes is efficient, and search is also fairly efficient. In particular, in the case of range search the actual leaf nodes can be read efficiently. Another alternative is depth-first layout, which has some advantages in the case of low fan-out where it could be advantageous to read all children of a lowest-layer internal node in once operation (see Section 6.1.3).

6 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. The properties of the ITTX/ND on the top level is well-known from earlier studies[11], so we

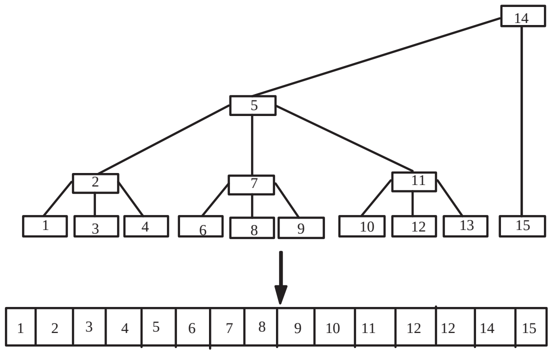


Figure 7: Insert-sorted layout.

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 [9].

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:

- A term that is very frequently occurring. The posting list of the term contains a total of 43139 posting intervals (of which 27% are without end timestamp, i.e., current terms).
- A moderately frequent term, whose posting list contains 7488 posting intervals (of which 13% are without end timestamp).
- A infrequently occurring term, whose posting list contains 4102 posting intervals (of which 14% are without end timestamp).
- A very infrequently occurring term, whose posting list contains only 384 posting intervals (of which 25% are without end timestamp).

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.

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.

However, as will be explained in Section 6.4 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 somewhere 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. The posting intervals are created as follows:

1. When inserting the document collection into the database, each document is stored in a separate transaction, and the timestamps has millisecond granularity. Most actual document inserts/updates will take more than one millisecond, thus giving a result close to the first of the extreme cases above. However, note that delete operations can be performed faster, so that the number of indexing points will be higher than the number of start- and end-timestamps in the posting intervals.

2. In order to increase the number of events at each indexing point, the timestamps are truncated by removing one or more of the least significant digits. In the case of the posting list of the frequent term, removing 3 digits of the timestamp reduced the number of unique timestamps (indexing points) from 74812 to 22696, and removing 4 timestamps reduced the number of unique timestamps to 4677. This means on average 16 events for each indexing point, i.e., interval starts or ends (term introduced or removed from document). We will from now on denote the two test sets created in this way as *G3* and *G4*.

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:

- No use.
- Use when applicable only on 1st level above leaf nodes.
- Use when applicable on all levels.

As described in Section 5.2, 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.

6.1 Very frequent term

6.1.1 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.

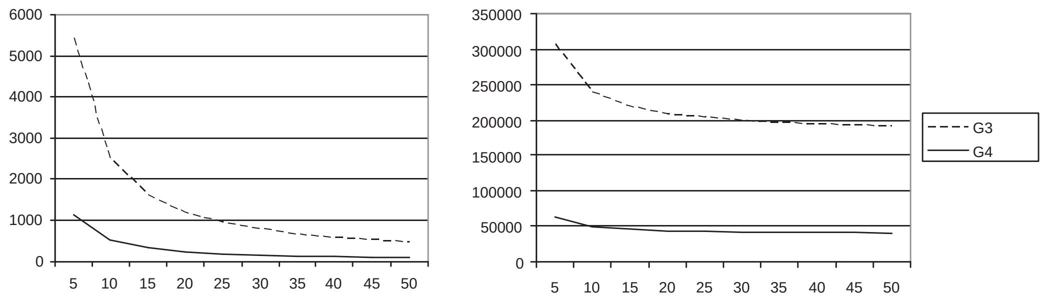


Figure 8: Number of nodes (left) and total space usage (right) for the TPI search tree.

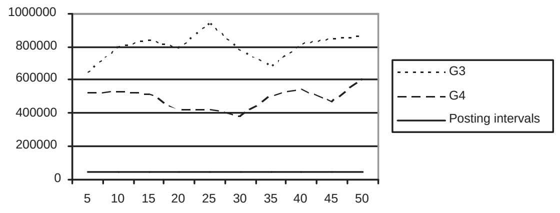


Figure 9: Number of DIDs stored in the TPI.

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 8 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 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.

6.1.2 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 9. 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 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.

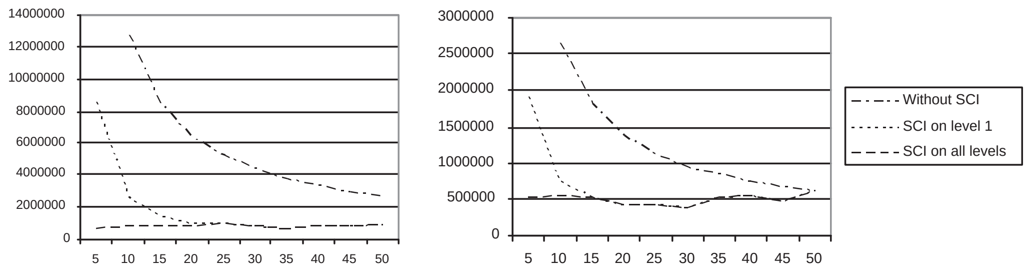


Figure 10: Number of DIDs for different SCI alternatives, G3 to the left, G4 to the right.

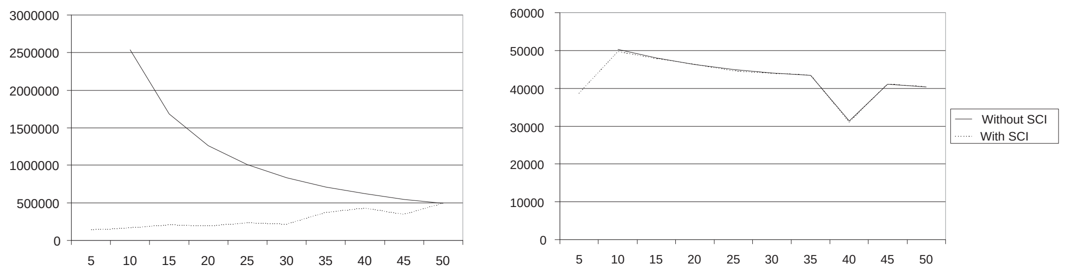


Figure 11: Number of DIDs in the SCS (left) and SCE buckets (right).

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 G4, and 790000 for G3 (for comparison, the number was about 900000 for the original interval set with no truncation). For G4, the average number of duplicates of each identifier was 10, with the lowest value of 8 duplicates in the case of $k = 30$.

In the numbers presented so far, SCIs has been employed wherever possible. An interesting issue is the actual impact of using SCI. Figure 10 shows the effect of different SCI usage alternatives for G3 (to the left) and G4 (to the right). The figures give the amount of DIDs that have to be stored in the buckets when 1) SCIs are not used, 2) SCIs are only used on the first level of internal nodes (the nodes above the leaf nodes), and 3) SCIs are used wherever applicable. Note that the numbers for $k = 5$ without using SCI are omitted because of the extreme number of stored DIDs that would be the result when using these parameters. Using SCIs is particularly beneficial in the case of long intervals, i.e., intervals that cover more then one leaf node. This is also evident from the figures, for small values of k the reduction in redundancy when using SCI is very high. It can also be seen that except for small value of k, the difference between using SCI on the first level of internal nodes only and all levels is not large. However, the difference can be expected to be larger in a real-life document database system that has been in use for a long time and storing large amounts of documents. This is likely to result in a larger number of indexing points that would make more levels in the tree necessary. The can also be seen from the figures. G3 represents a higher number of indexing points, and the curve for SCI used only on level 1 and curve for SCI on all levels are comparable at a higher value of k than in the case of G4 which has a smaller number of indexing points.

In order to represent a interval in the TPI, at least one entry in a SP bucket and one entry in a SM

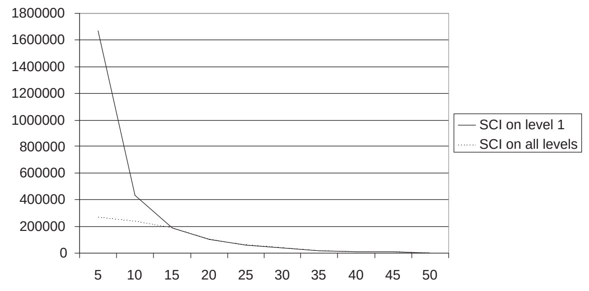


Figure 12: Number of DIDs in SCI.

bucket is necessary. The main part of redundancy occur in the SCS buckets. With small nodes and many indexing points, the number of leaf nodes will increase, and so will the number of SCS buckets. This is illustrated in Figure 11, which shows the number of DIDs in SCS and SCE buckets for G4, both with and without the use of SCI. Without the use of SCI, long-lived intervals will have to be stored in many consecutive SCS buckets. When SCI buckets are employed, the number of DIDs in the SCS increases slightly with increasing node size. The reason for this is that the number of DIDs in SCI buckets are reduced with increasing node size, as a result of fewer levels in the tree and that each internal node covers a much longer interval, which is larger than most intervals stored in the tree (see Figure 12). Note that the use of SCI does not affect the number of DIDs stored in SCE buckets. This is as expected, because only intervals covering a number of SCS can be represented by SCI buckets. Also note that whether SCI is used on only one or more than one level, the number of DIDs stored in SCS buckets will be the same. The reason for employing SCI on higher levels is to reduce the size of the SCI buckets in lower levels of the tree.

6.1.3 Search cost

In the prototype snapshot search and range search has been implemented. We will now present results using 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.

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

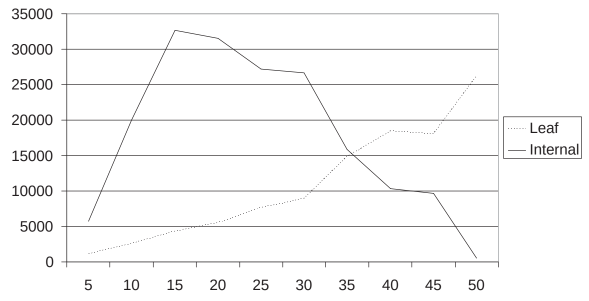


Figure 13: Average size of nodes and their associated buckets.

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 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 13, which shows the average size of nodes and associated buckets.

Using preorder depth-first placement (see Section 5.4.2), 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. 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

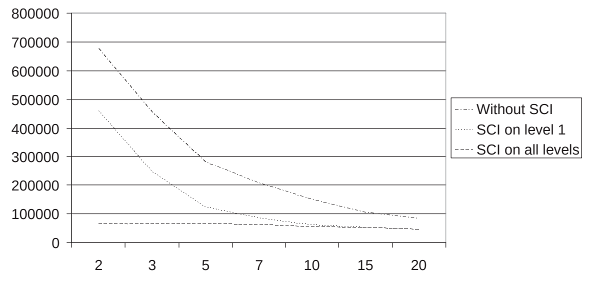


Figure 14: Number of DIDs stored in the TPI for G4.

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.

6.2 Moderately frequent term

The number of postings for the moderately frequent term is about 20% of the postings for the very frequent term, requiring approximately 88 KB stored in an ITTX/ND posting list.

Figure 14 illustrates the number of DIDs that have to be stored using different SCI strategies. The space usage of TPI including the buckets is approximately 238 KB, i.e., 2.7 times more than the space usage of the posting list.

The search cost in the case of a moderately frequent term is still smaller than for the posting lists. With a typical node size the search tree would have 3 levels, meaning 3 accesses needed at most. For the posting list approach, up to 11 disk accesses might be needed assuming 8 KB pages.

6.3 Infrequently occurring terms

In the case of the infrequent and very infrequently occurring terms, the TPI space usage was 48 KB and 6 KB, respectively. Compared to the posting list, the space usage was 1.5 and 1.3. For such terms it is probably not worthwhile to maintain TPI structures, but given the small additional extra cost compared to the posting lists, this can also be used as an argument for using TPI in order to have only one structure in the leaf nodes of the term index. Small TPIs can always be retrieved and stored as complete trees, thus having a low update cost.

6.4 Summary and discussion

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. The cost of most range searches can also be expected to be considerably reduced, because fewer disk operations have to be performed.

The aspect of transaction size was mentioned previously, and as we have seen TPI can benefit greatly in terms of space usage if there are many events at each indexing point. In the other direction, with fewer events at each indexing point than what has been the case in this study, the result will be a larger tree because more indexing points have to be indexed. The size of the search tree part of TPI will increase almost linearly with the number of indexing points. However, it will in many cases be useful to increase the node size in order to reduce the number of DIDs that have to be redundantly stored in continuation buckets.

7 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 DyST using a temporal document collection.

In addition to the application areas like repositories as mentioned in the introduction, the indexing techniques of this paper are also very interesting in the context of web warehouses. This application area can be interesting both in a number of commercial contexts as well as, e.g., national libraries responsible for archiving the web pages for the future.

In many application areas of temporal document databases, there will be a combination of very large databases and high number of users. In order to support acceptable response time, parallel or distributed document databases will be necessary. Our future work includes a closer study on management of temporal documents and indexes in such systems.

References

[1] P. G. Anick and R. A. Flynn. Versioning a full-text information retrieval system. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 98–111, 1992.

[2] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim. The time index: An access structure for temporal data. In *Proceedings of the 16th VLDB Conference*, 1990.

[3] G. Kazai, N. Gvert, M. Lalmas, and N. Fuhr. The INEX evaluation initiative. In *Intelligent Search on XML Data, Applications, Languages, Models, Implementations, and Benchmarks*, 2003.

[4] V. Kouramajian, I. Kamel, R. Elmasri, and S. Waheed. The Time Index+: an incremental access structure for temporal databases. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM)*, 1994.

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

[6] 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.

[7] K. Nørvåg. Space-efficient support for temporal text indexing in a document archive context. In *Proceedings of the 7th European Conference on Digital Libraries (ECDL'2003)*, 2003.

[8] K. Nørvåg. V2: a database approach to temporal document management. In *Proceedings of the 7th International Database Engineering and Applications Symposium (IDEAS)*, 2003.

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

[10] K. Nørvåg and A. O. Nybø. Creating synthetic temporal document collections. Technical Report IDI 6/2004, Norwegian University of Science and Technology, 2004. Available from `http://www.idi.ntnu.no/grupper/DB-grp/`.

[11] K. Nørvåg and A. O. Nybø. Improving space-efficiency in temporal text-indexing. Technical Report IDI 7/2004, Norwegian University of Science and Technology, 2004. Available from `http://www.idi.ntnu.no/grupper/DB-grp/`.

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

[13] Text REtrieval Conference, `http://trec.nist.gov/`.

[14] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.