Wave-Indices: Indexing Evolving Databases

Narayanan Shivakumar, Hector Garcia-Molina
Department of Computer Science
Stanford, CA 94305.
{*shiva, hector*}*@cs.stanford.edu*

Abstract

In many applications, new data is being generated every day. Often an index of the data of a past window of days is required to answer queries efficiently. For example, in a warehouse one may need an index on the sales records of the last week for efficient data mining, or in a Web service one may provide an index of Netnews articles of the past month. In this paper, we propose a variety of *wave indices* where the data of a new day can be efficiently added, and old data can be quickly expired, to maintain the required window. We compare these schemes based on several system performance measures, such as storage, query response time, and maintenance work, as well as on their simplicity and ease of coding.

Keywords: Indexing, sliding windows, temporal databases

Note to Referees: This paper is an extended version of a prior conference publication in ACM SIGMOD'97 with the same title. The material has been extended significantly in the following ways:

1. The conference version deals exclusively with the case when data sizes across different days is the same. We have now updated Section 3.3 and Section 6 to distinguish between *index length* and *index size*, so we can support wave indices with little overhead for the case we have non-uniform data sizes.

2. We have expanded Section 5 to include new analysis of our techniques for additional performance measures.

3. The conference version motivated our techniques using examples. This version also includes formal descriptions of the algorithms using a pseudocode format in Appendix A, so that practitioners can immediately adopt the techniques we propose in this paper.

4. The prior version included theorems and claims without proof, and we include proofs for these in Appendix B.

1 Introduction

In today's world, large amounts of data are constantly being generated every day, and often applications require an index into the data of some past *window* of days. For example, a Web search engine may provide an index for the past 30 days of Netnews articles, or a financial institution may keep an index of the stock market trades of the past 7 days. Each day, a batch of new data must be added to the index, and data older than the window should be removed.

There are at least three (interrelated) reasons why such sliding window indexes are useful. The first is that the application semantics require a sliding window. For example, if credit card bills can be contested for say up to 90 days, company agents may need to have fast access to the bills of exactly the past 90 days. A second reason is that user interest in data may wane over time. For instance, a stock market analyst may only want to look at recent trades, while a Netnews reader may not be interested in old data. So even if one could build an index for all the data, it would be less useful because it would give the user more information than he wants. A third reason is to reduce storage costs. For example, until recently the Stanford University library maintained only the past 5 years of Inspec, a commercially available bibliography of technical papers. Clearly, at Stanford we were interested in older papers, but the library chose to provide fast index service for only the recent papers, and slower access (look through the stacks) for the rest. In this case, the sliding window index is a cache of what hopefully are the most frequently accessed papers.

Sliding window indexes have been in use for many years, but the tremendous volumes of data that are today being generated in some applications makes it worthwhile to study these indexes carefully. In particular, Internet search engines such as Altavista [Alt], Infoseek [Inf] and Dejanews [Dej] are indexing ever-growing numbers of Web pages, Netnews articles, and other information. In Data Warehousing and On-line Analytical Processing (OLAP), huge volumes of sales, banking, and other transactions are being recorded and analyzed. In our own case, we were motivated to study indexes because of our implementation of the Stanford Copy Analysis Mechanism (SCAM) [SGM96, Ros96]. SCAM registers and indexes large numbers of digital documents collected from the Internet, and allows publishers and authors to search for illegal copies of their work. In SCAM we decided to index only documents collected over the past one or two weeks, both because interest in improper copying decreases over time, and because we could not afford more storage. In the rest of the high volume applications we have mentioned, there is often a similar need for indexing a window of days.

One obvious solution for indexing a window is to keep a single conventional index, and every day

Day	New Data	Operation	Index File (I_1)	Index File (I_2)
10	$+ d_1, \ldots, d_{10}$	Start	$\{ d_1, d_2, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$
11	$+ d_{11}$	Delete d_1 from I_1	$\{ d_2, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$
		Add d_{11} to I_1	$\{ d_{11}, d_2, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$
12	$+ d_{12}$	Delete d_2 from I_1	$\{ d_{11}, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$
		Add d_{12} to I_1	$\{ d_{11}, d_{12}, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$
13	$+ d_{13}$	Delete d_3 from I_1	$\{ d_{11}, d_{12}, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$
		Add d_{13} to I_1	$\{ d_{11}, d_{12}, d_{13}, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$

Table 1: Deletion based index maintenance ($W = 10$).

to delete the old data and insert the new batch of data into it. However, there are other interesting ways to maintain an index on a window of days, and we will see that they may have important advantages. To motivate, we now consider examples of a few such techniques in Tables 1, 2 and 3. In these examples, the techniques index a window of W days and partition the data across multiple indexes. To service queries all indexes will be accessed. The first row in each table is a "start" case where data of the first W days is indexed. On any subsequent day i, we need to index *new data* d_i into the required window. To do so, we execute the listed operations (under *Operation*). The columns labeled *Index* show the days that are covered by each index after the operations are executed. Some ways of maintaining an index of a window of days are:

1. *DEL:* We illustrate *DEL* in Table 1 with $W = 10$ and two indexes, I_1 and I_2. On the tenth day, data of the first five days is indexed into I_1 and data of the next five days is indexed into I_2. When data d_{11} is available on the 11^{th} day, we first delete d_1 from I_1. We then index d_{11} into I_1. Similarly with subsequent days. *DEL* is similar to the obvious solution mentioned above, except that it uses multiple indexes. Note that *DEL* maintains *hard* windows in that it indexes exactly the last W days (unlike *WATA*, one of the schemes we consider below).

2. *REINDEX:* We illustrate *REINDEX* in Table 2 with $W = 10$ and two indexes, I_1 and I_2. On the tenth day, data of the first 10 days is indexed into I_1 and I_2 as in *DEL*. When data d_{11} is available on the 11^{th} day, we replace the expired d_1 in I_1 with d_{11}. We perform this by rebuilding index I_1 with data d_2, d_3, d_4, d_5 and d_{11}. Similarly with subsequent days. *REINDEX* also maintains hard windows.

3. *Wait and Throw Away (WATA):* We illustrate *WATA* in Figure 3 with $W = 10$ and four indexes. On the tenth day, we index data of the first three days into I_1, data of the next three days into I_2, data of the subsequent three days into I_3 and data of the tenth day into I_4. When data d_{11} is available on the 11^{th} day, we add it to I_4. Similarly for d_{12}. When data d_{13} is available on the 13^{th} day, we first throw away I_1. We then create a new index I_1, and finally add d_{13} to it. The next day we add d_{14} to I_1, and so on.

Day	New Data	Operation	Index File I_1	Index File I_2
10	$+ d_1, \ldots, d_{10}$	Start	$\{ d_1, d_2, d_3, d_4, d_5\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
11	$+d_{11}$	Reindex d_2, d_3, d_4, d_5, d_{11}	$\{ d_2, d_3, d_4, d_5, d_{11}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
12	$+d_{12}$	Reindex d_3, d_4, d_5, d_{11}, d_{12}	$\{ d_3, d_4, d_5, d_{11}, d_{12}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
13	$+d_{13}$	Reindex d_4, d_5, d_{11}, d_{12}, d_{13}	$\{ d_4, d_5, d_{11}, d_{12}, d_{13}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
14	$+d_{14}$	Reindex d_5, d_{11}, d_{12}, d_{13}, d_{14}	$\{ d_5, d_{11}, d_{12}, d_{13}, d_{14}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
15	$+d_{15}$	Reindex d_{11}, d_{12}, d_{13} d_{14} d_{15}	$\{ d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
16	$+d_{16}$	Reindex d_7, d_8, d_9, d_{10}, d_6	$\{ d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{d_7, d_8, d_9, d_{10} d_{16} \}$

Table 2: Reindexing based index maintenance ($W = 10, n = 2$).

Day	New Data	Operation	Index File I_1	Index File I_2	Index File I_3	Index File I_4
10	$+d_1, \ldots, d_{10}$	Start	$\{ d_1, d_2, d_3 \}$	$\{ d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{ d_{10} \}$
11	$+ d_{11}$	Add d_{11} to I_4	$\{ d_1, d_2, d_3 \}$	$\{ d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{ d_{10}, d_{11} \}$
12	$+ d_{12}$	Add d_{12} to I_4	$\{ d_1, d_2, d_3 \}$	$\{ d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{ d_{10}, d_{11}, d_{12}\}$
13	$+ d_{13}$	Drop I_1	-	$\{ d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{ d_{10}, d_{11}, d_{12} \}$
		Create $I_1 = \phi$	$\{ \}$	$\{ d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{ d_{10}, d_{11}, d_{12} \}$
		Add d_{13} to I_1	$\{ d_{13} \}$	$\{ d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{ d_{10}, d_{11}, d_{12} \}$
14	$+ d_{14}$	Add d_{14} to I_1	$\{ d_{13}, d_{14} \}$	$\{ d_4, d_5, d_6 \}$	$\{ d_7, d_8, d_9\}$	$\{ d_{10}, d_{11}, d_{12} \}$

Table 3: WATA based index transitions ($W = 10, n = 4$).

Notice that in *WATA* we occasionally maintain data older than the required window. For example on days 11 and 12, data of d_1 is still indexed in I_1 even though it is no longer required as part of the window. *WATA* maintains *soft* windows. Such soft windows may be acceptable in certain applications. For instance, in case of Altavista it is probably acceptable to maintain a soft window of up to 35 days while the required window is only 30 days. Such soft windows may also be acceptable for statistical or trend analyses.

We now briefly consider some of the advantages of the schemes, as presented in the examples. Note however that in this paper we will propose enhancements to these sample schemes, as well as additional schemes, so our comments should be taken as first indication of what might be good or bad about a scheme. We will have a more detailed and formal analysis of the schemes in Sections 5 and 6. Some of the advantages of the schemes in the example are:

- **Bulk Insert/Delete:** In *WATA*, deletions are performed in bulk by throwing away a whole index. If there are a substantial number of deletes, this may be more efficient than deleting an entry at a time (as in *DEL*). For instance in a commercial relational database such as Sybase, it takes a few milli-seconds to throw away an index irrespective of the index size. On the other hand, deleting an entry at a time takes time proportional to the number of deletes. Similarly, it may be efficient to reindex data, like *REINDEX* does, if there are a lot of inserts and the index does not cover too many other days. This is because *incremental* indexing schemes [FJ92, TGMS94] may be expensive.

- **Better Structured Index:** Even though *REINDEX* may sometimes be more costly because it rebuilds indexes from scratch, this rebuilding can often lead to a better structured index (e.g., less fragmentation and contiguous layout on disk). Such an index could lead to more efficient query processing. Thus, we can trade off more index build time for better query performance. This may be another reason to prefer *REINDEX* over *DEL* or *WATA*.
- **Simpler Code:** With *REINDEX* and *WATA*, we do not need complex index deletion code [Jan95]. This could be a great advantage if we are implementing our system from scratch. Also *REINDEX* does not require complex concurrency control since updates and queries are operating on a different set of indexes. We will later consider the case when *shadow* indexes are used to avoid concurrency control code in all the schemes.
- **Legacy Systems:** Some information retrieval indexing packages such as WAIS [Pfe] and SMART [SB], do not implement deletes at all. If we need to use some such package or a legacy system to maintain a window of days, we may have to use one of the new schemes such as *REINDEX* or *WATA*.
- **Query Performance:** Clearly, having multiple indexes creates more work for queries, as they must perform several searches. However in "data analysis" scenarios where query volume may be relatively low and data volumes may be high, the high query costs may be amortized by the savings under some of the categories listed above. Furthermore, if multiple disks and computers are available, the queries across indexes can be easily parallelized. Also in some queries may be constrained to search over a subset of the indexed days, in which case fewer indexes may be searched.

In this paper we use the term *wave index* to refer to a collection of n "conventional" indexes that provide access to a window of W consecutive time intervals ($1 \leq n \leq W$). We use the term "day" to refer to each time interval, although in general time intervals need not be 24 hours.

In the first part of this paper (Sections 2 and 3), we propose six different wave indexing algorithms and three ways for performing updates within each algorithm. In particular, we formalize *DEL, REINDEX* and *WATA*, propose *REINDEX*$^{+}$, *REINDEX*$^{++}$ that improve *REINDEX*, and finally describe *RATA*, a hybrid of *REINDEX* and *WATA*. Each of the above algorithms differ in (1) how the first W days are initially split across the n indexes, (2) how the wave index is modified when a new day's data is available, and (3) whether they maintain "hard" or "soft" windows.

In the second part of this paper (Section 5), we evaluate each of our proposed schemes for a variety of system performance measures. Through our evaluations we attempt to answer questions such as the following: (1) Given a new day's worth of data, how fast can a scheme index the data and make it available for querying? (2) How does the scheme perform as the query/update mix changes? (3) How much overall disk activity is required for maintaining a window and for servicing queries during a day? (4) How much disk space is required to index the data? (5) Does a scheme require complex code for deletion, or for concurrency control? (6) Can the scheme be implemented on top of "widely available" index structures, or is special code required?

In the final part of the paper we consider three "case studies" and show how different wave indexes may be appropriate in each scenario. The scenarios considered are our own SCAM service that indexes Netnews articles for copy detection, a generic Web search engine such as Altavista that indexes the same articles for general user queries, and a representative TPC-D benchmark [TPC] query in a warehousing context. For each scenario we measure realistic parameters whenever possible (e.g., the volume of Netnews articles in a day), and make educated guesses when it is not possible (e.g., how many copy detection queries will be submitted to SCAM when it is operational). We believe that our results provide useful insights into the tradeoffs between the wave index schemes, and can help an application designer in selecting a wave index.

2 Preliminaries

In this section we outline the basic index structures used in this paper, we describe how these are updated, and we define the operations to manage wave indexes. Note that most of the ideas in this paper are applicable to all classes of index structures, but for concreteness here we will focus on one specific class we now describe.

Figure 1 illustrates the basic index structures. The data we need to index consists of *records*. For instance, r_1 and r_2 in the figure are records. Each of the records has a *search field*, F, upon which an index is being built. Each record may have multiple values for F, for example a record may have values "War" and "Peace" for its title field. Similarly, an employee record may have values $D55$ and $D57$ in its "department" field. The index consists of a *directory* and associated *buckets*. The directory is a search structure (e.g., a B+Tree or a hash table) that given a search value, v, identifies a bucket b. Bucket b contains a pointer p_i for each record r_i having search value v. In b, each pointer p_i may have additional associated information a_i. For example, in an Information Retrieval context, with each p_i we can store the byte offset of value v in field F of r_i. In a relational database context, with each p_i we may store additional attributes of r_i to speed up searches. For some of the indexing schemes we use here, we require a *timestamp* for each a_i, which denotes the day r_i was inserted. We refer to p_i and its associated information as an *entry*.

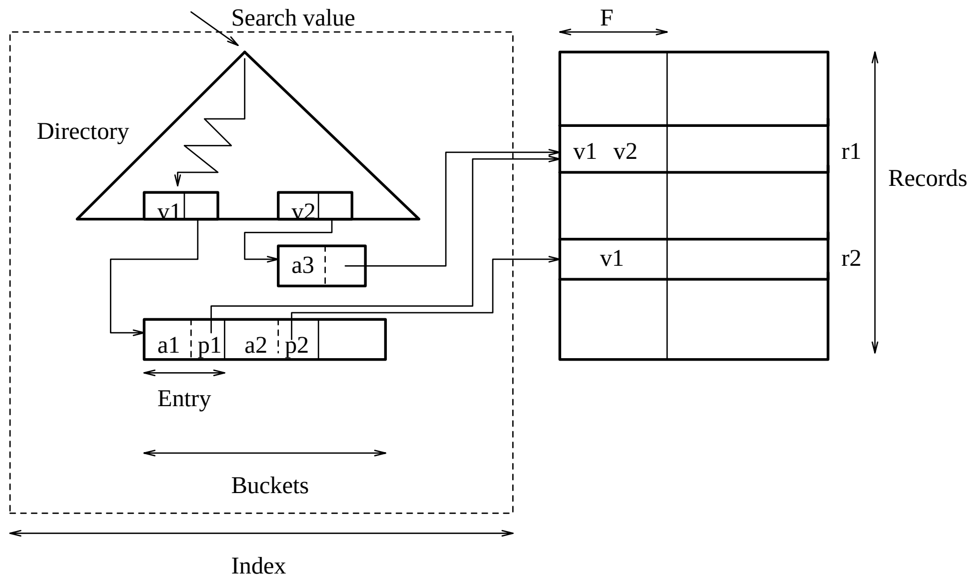

Figure 1: Basic index structures.

For simplicity we assume that the directory is in memory, and the buckets are on disk. We define an index to be *packed* if each of its buckets uses a minimal amount of space to store entries (without room for growth), and all its buckets are allocated contiguously on disk. If an index is not packed, we still assume that entries within a given bucket are contiguous. In some applications, packed indexes may be preferable since they save space, and are efficient for queries that scan the whole index. For example, queries that compute some aggregate such as *sum, min* or *max* typically scan the whole index. If the index entries are packed contiguously on disk, the query can be efficiently executed by scanning (with a single disk seek) the entries from the first bucket until the last bucket, and computing the aggregate.

A *wave index* on a search field F is used to search a collection of *days*, where each day contains the records generated during a particular time period (typically 24 hours). The time periods covered by the wave index should be contiguous. The days are partitioned into disjoint *clusters* and an index on F is built for each. Each individual index is termed a *constituent* index. The set of constituent indexes is termed the wave index, Θ. In the rest of the paper, when we use “index” we refer to a constituent of a wave index.

2.1 Update techniques

Suppose that we have an index on a set of records and the records change, or records are added to or deleted from the set. To update the index to reflect this batch of updates, we can use one of the following three techniques. (In this paper we assume updates for a day are performed as a batch. This usually leads to better performance, mainly due to memory caching.)

1. **In-Place Updating:** For each update the directory and/or buckets are modified in-place. If there is not enough space in the bucket, then the bucket can be copied to a new location and allocated more space. To decide how much space to add, we could use techniques proposed in [FJ92]. This updating technique requires concurrency control to prevent queries from reading inconsistent data. Typically the resulting index is not packed even if the original one was packed.

2. **Simple Shadow Updating:** First make a copy of the index, and then for each update modify the new copy of the index in-place. Finally, the new index replaces the old version in the wave index. The main advantage of this technique is that queries can be serviced using the old index, while the new index is being updated. Hence no concurrency control is required. The corresponding disadvantage is that more space is required than with in-place updating while the new day is being indexed.

3. **Packed Shadow Updating:** This technique is similar to the simple shadow technique except that the resulting index is packed. Although this technique works in general, here we describe it when the updates consist of a set of inserted records, and records to be deleted are those with an expired timestamp. First we build a temporary index for the new records to be inserted. We then scan the buckets of the index to be updated, copying them to a new contiguous location, but in the process deleting entries with expired timestamps, and leaving enough space in each new bucket copy to accomodate entries for the inserted records. Then we scan the temporary index, and append each bucket to the appropriate bucket in the new index, if one exists. If not, that bucket represents a new search value not present in the old index. We append such buckets after the last bucket in the new index. Finally we update the directory to reflect the new search values, and the new index replaces the old version in the wave index.

2.2 Operations on a Wave Index

In describing our wave index algorithms, we use the following primitive functions. For simplicity we use integers to refer to days. Thus the days indexed by I in a wave index Θ can be represented by a set of integers, referred to as the *time-set* of the index.

1. **Wave index update operations:**

(a) **AddIndex(I, Θ):** Given a wave index Θ and an index I, this operation adds I to the set of constituent indexes in Θ.

(b) **DropIndex(I, Θ):** Given a wave index Θ and an index I, this operation first removes I from Θ. It then deletes all index entries in I (i.e., reclaims space).

2. **Constituent index update operations:**

(a) **BuildIndex(*Days*):** Given *Days*, a set of integers, this operation builds a packed index for the batch of records in those days. i.e., for the cluster identified by *Days*. We assume here that a packed index is achieved by scanning the *Days* records and counting the number of entries needed in each bucket. Then contiguous buckets of the appropriate size are allocated on disk.

(b) **AddToIndex(*Days*, I):** Given *Days*, a set of integers, and an index I, this operation incrementally adds the batch of entries for *Days* records to I. This can be achieved using any one of techniques in Section 2.1. Thus if in-place or simple shadow updating is used, the resulting I will not be packed. If packed shadows are used, then I is replaced in the wave index by a new packed index.

(c) **DeleteFromIndex(*Days*, I):** Given *Days*, a set of integers, and an index I this operation incrementally deletes entries for *Days* records from I. Like $AddToIndex$, this can also be performed using any of the three techniques in Section 2.1. Again if in-place or simple shadow updating are used, I will not be packed. If packed shadow updating is used, I will be packed.

Note that *BuildIndex* and *AddToIndex* can often be used to achieve the same goal. However the performance can be very different. For instance, let a cluster have five days worth of data and suppose that we already have an index for the first four days. We can construct an index for the 5-day cluster either by adding the the fifth day to the existing index, or by building the index from scratch for the 5 days. The former option is typically less expensive than the latter. However, unless packed shadowing was used in the former, the latter will be more efficient for scan queries since the resulting index is packed. On the other hand, if we do not have the initial 4-day index, it is typically more efficient to do a *BuildIndex* rather than a series of *AddToIndex* operations.

3. **Access operations:**

We expect four kinds of queries to access the wave index. They are *IndexProbe*, *SegmentScan*, *TimedIndexProbe* and *TimedSegmentScan*. To illustrate, consider a set of daily sales records for the past year, indexed by the sales person. Let us assume that each index entry contains, in addition to a pointer to full sales record, the amount and date of sale (i.e., when the record was inserted.) A query that looks at all sales entries for a given salesperson, $S1$, will be executed as an *IndexProbe*, which probes the index with search value $S1$. A query that looks at sales entries of $S1$ for the past month will be executed as a *TimedIndexProbe*, which is an *IndexProbe* restricted to entries with a date in the past month. A query to compute aggregate yearly sales by sales person for the store will be executed as a *SegmentScan*, which scans all buckets of the index. A query to compute aggregate sales for the past month will be executed as a *TimedSegmentScan*, which is a *SegmentScan* restricted to entries inserted in the past month. As we shall see now, *IndexProbe* and *SegmentScan* can be expressed as *TimedIndexProbe* and *TimedSegmentScan* respectively.

(a) **TimedIndexProbe(Θ, T_1, T_2, s):** Given a wave index Θ, times T_1 and T_2 and search value s, this operation retrieves buckets of entries for v inserted between day T_1 and T_2. It does this by probing a subset of constituent indexes in Θ whose clusters have days more recent than T_1 and older than time T_2. For each such index, buckets for s are retrieved, and entries with insert time in the desired range are selected. Note that if we restrict timed queries to only refer to time intervals that correspond to the cluster intervals, then bucket entries do not need insertion times. That is, all entries for s in the indexes in the T_1, T_2 range will be relevant. When $T_1 = -\infty$ and $T_2 = \infty$, this operation is equivalent to an *IndexProbe* that probes all indexes.

(b) **TimedSegmentScan(Θ, T_1, T_2):** Given a wave index Θ and times T_1 and T_2, this operation retrieves all entries inserted between day T_1 and T_2. It does this by scanning buckets of all constituent indexes in Θ whose clusters have days more recent than time T_1 and older than time T_2. When $T_1 = -\infty$ and $T_2 = \infty$, this operation is equivalent to a *SegmentScan* that scans all buckets in all indexes.

3 Building Simple Wave Indexes

In this section, we review the simple algorithms to build wave indices that we presented in Section 1. Let d_i refer to the i^{th} days' data, and d_{new} refer to a new day's data. Let Θ be the wave index being maintained.

3.1 Deletion *(DEL)*

We briefly motivated *DEL* in the Introduction with Table 1. In *DEL*, we initially index W/n days of data[1] each in indexes $I_1, I_2, \ldots, I_n$. We then make $I_1, I_2, \ldots, I_n$ constituent indexes of Θ. Every day when d_{new} is available, we delete entries of d_{new-W} from I_j that indexed d_{new-W}. Then we insert entries for d_{new} to I_j. The deletion and insertion can be performed using one of the update techniques proposed in Section 2.1. We present the formal *DEL* algorithm in Appendix A as Figure 12.

DEL maintains hard windows. If in-place or simple shadow updating are used, *DEL* requires code to implement incremental deletion in both the directory and the buckets. Also the resulting index is not packed. If packed shadow updating is used, the resulting index is however packed.

3.2 Reindexing (*REINDEX*)

We briefly motivated *REINDEX* in the Introduction with Table 2. The operations performed at each step in the example is actually a *BuildIndex*. We formally present the *REINDEX* algorithm in Appendix A as Figure 13.

REINDEX maintains hard windows, and the resulting index is packed. However this technique requires reindexing W/n days worth of data every day. In Section 4 we propose several schemes that reduce the work done while building the index.

3.3 Wait and Throw Away (WATA)

We briefly motivated the WATA approach in the Introduction with Table 3. Recall that this algorithm uses a lazy form of deletion by throwing away an entire index only when all its entries have expired. Clearly there are several ways to implement this type of lazy deletion. For example, Table 4 presents a scheme that is slightly different from the one in Table 3, for the same $W = 10$, $n = 4$. We see that on the 10^{th} day the example in Table 4 forms different clusters for the four indexes than we had earlier. While there are a variety of measures we can use to evaluate different WATA-based schemes, we concentrate on the following measures for the purposes of this paper:

- **Length of index:**

  For some applications, we may prefer a WATA scheme that gives us the "tightest" soft window. For instance, if we are computing the average revenue and standard deviation for the past

[1] In the formal algorithm in Appendix A, we handle the case when W/n is not an integer.

week from the sales relation of the company, we may prefer a WATA scheme with small soft windows for more accuracy.

In Table 4, we see that the total number of days indexed on days 11, 12, 13 is 11, 12 and 13, respectively. We define the *length* of the wave index so constructed to be 13, the maximum number of days stored in the index at any time. Similarly the length of the index constructed in Table 3 is 12, since the total number of days indexed on days 11, 12, 13 is 11, 12 and 10. Since the example in Table 3 has a smaller length, it indexes fewer extra days thereby providing a "tighter" window.

- **Size of index:**

  In many applications, we may prefer to use a WATA scheme that incurs the least space overhead due to lazy deletion. That is, we need to minimize the total *index size*, where index size is the maximum storage required for maintaining the wave index.

  When the size of data to be indexed is the same from day-to-day, minimizing index size corresponds to minimizing index length. However, the size of data to be indexed can vary dramatically across days. For example, the number of daily Usenet postings in popular newsgroups varies dramatically depending on the day of the week. In Figure 2 we report the total number of daily postings in September 1997 across about 10,000 popular newsgroups subscribed to by the Stanford Computer Science department's NNTP server. We see that the number of postings on the second Wednesday is about $110,000$, while on Sundays, the number of postings falls to around 30000. We will discuss algorithms to minimize the index size in case of non-uniform data sizes later in this section.

In Figure 16 of Appendix A we propose one instance of WATA termed WATA*. For this algorithm, we can show the following (proof in Appendix B).

Theorem 3.1 *(Index length)*

WATA* *is an optimal algorithm to construct a* WATA *wave index with the smallest index length.*
□

It is easy to see that to construct an optimal WATA index for index size, we need complete information of data sizes of all future days. Since such information is typically not available, we need to design an *online* algorithm that adds a new day's data to the wave index based only on the current day's data and the currently indexed days. However we can use the $WATA^*$ algorithm for minimizing index size as well, due to the following property.

Theorem 3.2 *(Index size)*

Day	Operation	Index File I_1	Index File I_2	Index File I_3	Index File I_4
10	$I_1 \leftarrow$ BuildIndex({ 1, 2, 3, 4 })				
	$I_2 \leftarrow$ BuildIndex({ 5, 6, 7 })				
	$I_3 \leftarrow$ BuildIndex({ 8, 9, 10 })				
	$I_4 \leftarrow \{\ \}$	$\{ d_1, d_2, d_3, d_4 \}$	$\{ d_5, d_6, d_7 \}$	$\{ d_8, d_9, d_{10} \}$	$\{\ \}$
11	AddToIndex({11}, I_4)	$\{ d_1, d_2, d_3, d_4 \}$	$\{ d_5, d_6, d_7 \}$	$\{ d_8, d_9, d_{10} \}$	$\{ d_{11} \}$
12	AddToIndex({12}, I_4)	$\{ d_1, d_2, d_3, d_4 \}$	$\{ d_5, d_6, d_7 \}$	$\{ d_8, d_9, d_{10} \}$	$\{ d_{11}, d_{12} \}$
13	AddToIndex({13}, I_4)	$\{ d_1, d_2, d_3, d_4 \}$	$\{ d_5, d_6, d_7 \}$	$\{ d_8, d_9, d_{10} \}$	$\{ d_{11}, d_{12}, d_{13} \}$
14	$I_1 \leftarrow \phi$	$\{\ \}$	$\{ d_5, d_6, d_7 \}$	$\{ d_8, d_9, d_{10} \}$	$\{ d_{11}, d_{12}, d_{13} \}$
	AddToIndex({14}, I_1)	$\{ d_{14} \}$	$\{ d_5, d_6, d_7 \}$	$\{ d_8, d_9, d_{10} \}$	$\{ d_{11}, d_{12}, d_{13} \}$

Table 4: Another example of index transitions based on WATA (W = 10, n = 4).

WATA* *is an online algorithm to construct wave indices with index size no more than twice the index size of any optimal* WATA *algorithm that has complete knowledge of future data sizes. That is, the* competitive ratio *[MR95] of* WATA* *is* 2.0 *for the index size measure.* □

Kleinberg et al [KMRV97] recently extended our work in WATA index construction in the following directions. They proposed an optimal WATA algorithm, for the case when they have complete knowledge of data sizes in the future. For the online problem, they improved the competetive ratio of our WATA algorithm to $\frac{n}{n-1}$, for n indices by assuming they know the total maximum index size ever possible in the future, ahead of time. Recall that our WATA* is "purely" online in that it assumes no such information, while still providing a competetive ratio of 2.0.

All WATA algorithms maintain soft windows and thereby use more space to store the extra days of data. However, they do relatively little work each day, and index deletion code is not needed. Also, once a new day's data is available, it takes only the time of one *AddToIndex* before the new data is available for querying. However, *TimedSegmentScans* may be less efficient due to the entries of days older than the window, but that are part of the soft window. Another potential disadvantage of WATA is that it requires at least two constituent indexes to be efficient. To see this, consider the case when there is only one constituent index. In that case, each new day has to be added to the single index, and at no point will all data in the index expire to allow the removal of the index. Hence, the constituent index will then keep growing forever. For this reason, we require at least two constituent indexes for WATA.

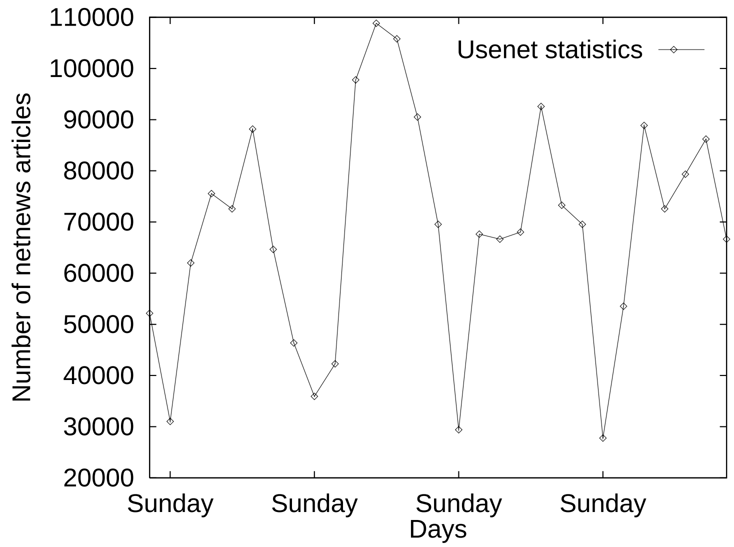

Figure 2: Number of Usenet postings per day in September 1997.

4 Enhancing Simple Wave Indices with Temporary Indices

In this section, we enhance the simple wave-indexing algorithms of Section 3 by constructing temporary indices. These enhanced schemes improve important performance measures such as average maintenance work, and time to add new data at the cost of using more disk space.

4.1 Improved reindexing (*REINDEX*$^+$)

This scheme enhances *REINDEX* by reducing the average work required in maintaining a wave index. To motivate *REINDEX*$^+$, we reconsider the example for *REINDEX* in Table 2. Note that index entries for d_{11} are recomputed every day from day 11 to day 15. Similarly, index entries for d_{12} are recomputed every day from day 12 to day 15. Similarly for d_{13} and d_{14}. Instead *REINDEX*$^+$ maintains a temporary index, *Temp*, to avoid recomputing these index entries every day.

In Table 5, we present an example of how *REINDEX*$^+$ works with $W = 10$ and $n = 2$. In this table (and in subsequent tables) we drop column *New Data* and assume that on day i ($i > W$), data d_i is available to be indexed. We add column *Temp* to show the current entries in *Temp*. On the 10^{th} day, the first five days are indexed in I_1 and the next five days are indexed in I_2 (as in *DEL* and *REINDEX*). In addition, an empty index, *Temp* is created. On the 11^{th} day when new data d_{11} is available, the cluster of I_1 should contain days d_{11}, d_2, d_3, d_4, and d_5. For this, we first index d_{11}

Day	Operation	Index I_1	Index I_2	Temp
10	Temp $\leftarrow \phi$ $I_1 \leftarrow$ BuildIndex($\{1,2,3,4,5\}$) $I_2 \leftarrow$ BuildIndex($\{6,7,8,9,10\}$)	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$	ϕ
11	*Temp*, $I_1 \leftarrow$ BuildIndex($\{11\}$) AddToIndex($\{2,3,4,5\}, I_1$)	$\{d_{11}, d_2, d_3, d_4, d_5\}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$	$\{d_{11}\}$
12	AddToIndex($\{12\}$, Temp) $I_1 \leftarrow$ Temp AddToIndex($\{3,4,5\}, I_1$)	$\{d_{11}, d_{12}, d_3, d_4, d_5\}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$	$\{d_{11}, d_{12}\}$
13	AddToIndex($\{13\}$, Temp) $I_1 \leftarrow$ Temp AddToIndex($\{4,5\}, I_1$)	$\{d_{11}, d_{12}, d_{13}, d_4, d_5\}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$	$\{d_{11}, d_{12}, d_{13}\}$
14	AddToIndex($\{14\}$, Temp) $I_1 \leftarrow$ Temp AddToIndex($\{5\}, I_1$)	$\{d_{11}, d_{12}, d_{13}, d_{14}, d_5\}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$	$\{d_{11}, d_{12}, d_{13}, d_{14}\}$
15	$I_1 \leftarrow$ Temp AddToIndex($\{15\}, I_1$) Temp $\leftarrow \phi$	$\{d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$	ϕ
16	*Temp*, $I_2 \leftarrow$ BuildIndex($\{16\}$) AddToIndex($\{7,8,9,10\}, I_2$)	$\{d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{ d_{16}, d_7, d_8, d_9, d_{10} \}$	$\{ d_{16} \}$

Table 5: Example of index Transitions in *REINDEX*$^+$ (W = 10, n = 2).

into *Temp*. We then copy *Temp* into I_1 so I_1 contains entries for d_{11}. Then we incrementally add d_2, d_3, d_4 and d_5 into I_1. On the 12^{th} day after new data d_{12} is available, the cluster of I_1 should contain days d_{11}, d_{12}, d_3, d_4, and d_5. For this, we first add new data d_{12} to *Temp*. We then copy *Temp* into I_1 so I_1 contains entries for d_{11} and d_{12}. Finally we incrementally add d_3, d_4 and d_5 to I_1. Similarly for subsequent days. Observe that between days d_{11} and d_{15} we are incrementally indexing progressively fewer days. This reoccurs between days d_{16} and days d_{20} and so on. We can see that the average number of days indexed per transition by *REINDEX*$^+$ during index build is about half that of *REINDEX*. The *REINDEX*$^+$ algorithm is formally described in Appendix A as Figure 14.

REINDEX$^+$ maintains hard windows. If we use in-place or simple shadow updating to update the constituent indexes, the resulting index is not packed. If we use packed shadow updating instead, the resulting index is packed. Every day, this scheme on the average reindexes about half the number of days that *REINDEX* does. It achieves this by using additional space to store a temporary index, *Temp*. Also like *REINDEX*, it does not require code for deleting from an index.

4.2 Further improved reindexing (*REINDEX*$^{++}$)

This scheme improves *REINDEX*$^{+}$ by reducing the time to index new data and making new data available sooner for querying. We achieve this by performing most of the work required in maintaining the wave index before the data is available. For this, we use a few temporary indexes $(T_1, T_2, \ldots)$ and increase our storage requirements.

We explain how *REINDEX*$^{++}$ works using the example in Table 6 with $W = 10$ and two indexes, I_1 and I_2. On the 10^{th} day, we index the first five days in I_1 and the next 5 days in I_2. Then we build temporary indexes $T_0, T_1, \ldots, T_4$ as follows. We initialize T_0 to an empty index, and we create T_1 with day 5. Then we copy T_1 to index T_2, and incrementally add day 4 to it, so T_2 contains days 4 and 5. Similarly for T_3 and T_4, so that T_3 contains days $3, 4, 5$ and T_4 contains days $2, 3, 4, 5$, as shown in column *Temp*. On the 11^{th} day, add d_{11} to T_4. Then rename T_4 as I_1 so that I_1 now contains days $2, 3, 4, 5$ as well as 11. Queries can start accessing data of d_{11} at this point much faster than if *REINDEX* were used. We then add d_{11} to T_3 so T_3 now contains days $3, 4, 5$ as well as day 11. Indexes T_2, T_1 and T_0 remain unchanged (they are not shown in order to reduce the size of the table.) On day 12, we add d_{12} to T_3 so it contains days $3, 4, 5, 11$ as well as day 12. As earlier, we rename T_3 as I_1 and queries can start accessing data of d_{12} at this point. We then add d_{11} and d_{12} to T_2 to be used the next day. Indexes T_1 and T_0 remain unchanged. Similary for days 13 and 14. On day 15 we reinitialize T_0, T_1, $\ldots, T_4$ for the next set of days. We formally present the algorithm for *REINDEX*$^{++}$ in in Appendix A as Figure 15.

REINDEX^{++} maintains hard windows. Like *REINDEX*$^{+}$, the constituent indexes are packed only if packed shadow updating is used. Notice that in *REINDEX*$^{++}$ we are doing marginally additional amount of work compared to *REINDEX*$^{+}$. On any given day, we are adding the new day's data to about half the indexes which is the work done in *REINDEX*$^{+}$. In addition on days 10, 15, $\ldots$, we incrementally index 4 days of data. In general, we would incrementally index W/n days of data every W/n days. Clearly this work can be spread across the W/n days. Hence *REINDEX*$^{++}$ performs about the same amount of work as *REINDEX*$^{+}$, but reduces the time to index a new day's data.

4.3 Reindex and Throw Away (RATA)

We now propose a variant of WATA to maintain hard windows. RATA is similar to WATA except that it uses additional temporary indexes to simulate deleting old entries. We explain RATA with the example in Table 7. In the example, we use the notation T_i for temporary indexes that replace some constituent index I_j on day i. On the 10^{th} day, RATA indexes the first ten days in the same way as WATA. In addition, it also builds additional temporary indexes, T_{11} and T_{12} so that T_{11}

Day	Operation	Index I_1	Index I_2	Temp
10	$I_1 \leftarrow$ BuildIndex($\{1,2,3,4,5\}$) $I_2 \leftarrow$ BuildIndex($\{6,7,8,9,10\}$) $T_0 \leftarrow \phi$, $T_1 \leftarrow$ BuildIndex($\{5\}$) $T_2 \leftarrow T_1$, AddToIndex($\{4\}, T_2$) $T_3 \leftarrow T_2$, AddToIndex($\{3\}, T_3$) $T_4 \leftarrow T_3$, AddToIndex($\{2\}, T_4$)	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{ d_6, d_7, d_8, d_9, d_{10} \}$	$T_0 = \phi$, $T_1 = \{d_5\}$ $T_2 = \{d_5, d_4\}$ $T_3 = \{d_5, d_4, d_3\}$ $T_4 = \{d_5, d_4, d_3, d_2\}$
11	AddToIndex($\{11\}, T_4$) Rename T_4 as I_1 AddToIndex($\{11\}, T_3$)	$\{d_5, d_4, d_3, d_2, d_{11}\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$T_4 = \{d_5, d_4, d_3, d_2, d_{11}\}$ $T_3 = \{d_5, d_4, d_3, d_{11}\}$
12	AddToIndex($\{12\}, T_3$) Rename T_3 as I_1 AddToIndex($\{11, 12\}, T_2$)	$\{d_5, d_4, d_3, d_{11}, d_{12}\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$T_3 = \{d_5, d_4, d_3, d_{11}, d_{12}\}$ $T_2 = \{d_5, d_4, d_{11}, d_{12}\}$
13	AddToIndex($\{13\}, T_2$) Rename T_2 as I_1 AddToIndex($\{11, 12, 13\}, T_1$)	$\{d_5, d_4, d_{11}, d_{12}, d_{13}\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$T_2 = \{d_5, d_4, d_{11}, d_{12}, d_{13}\}$ $T_1 = \{d_5, d_{11}, d_{12}, d_{13}\}$
14	AddToIndex($\{14\}, T_1$) Rename T_1 as I_1 AddToIndex($\{11, 12, 13, 14\}, T_0$)	$\{d_5, d_{11}, d_{12}, d_{13}, d_{14}\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$T_1 = \{d_5, d_{11}, d_{12}, d_{13}, d_{14}\}$ $T_0 = \{d_{11}, d_{12}, d_{13}, d_{14}\}$
15	AddToIndex($\{15\}, T_0$) Rename T_0 as I_1 $T_0 \leftarrow \phi$, $T_1 \leftarrow$ BuildIndex($\{10\}$) $T_2 \leftarrow T_1$, AddToIndex($\{9\}, T_2$) $T_3 \leftarrow T_2$, AddToIndex($\{8\}, T_3$) $T_4 \leftarrow T_3$, AddToIndex($\{7\}, T_4$)	$\{d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$T_0 = \{d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$ $T_0 = \phi$, $T_1 = \{d_{10}\}$ $T_2 = \{d_{10}, d_9\}$ $T_3 = \{d_{10}, d_9, d_8\}$ $T_4 = \{d_{10}, d_9, d_8, d_7\}$
16	AddToIndex($\{16\}, T_4$) Rename T_4 as I_2 AddToIndex($\{16\}, T_3$)	$\{d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{d_{10}, d_9, d_8, d_7, d_{16}\}$	$T_4 = \{d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$ $T_3 = \{d_{10}, d_9, d_8, d_{16}\}$

Table 6: Example of index transitions in *REINDEX*$^{++}$ (W = 10, n = 2).

State	Operation	Index I_1	Index I_2	Index I_3	Index I_4	Temp
10	$I_1 \leftarrow$ BuildIndex($\{1,2,3\}$) $I_2 \leftarrow$ BuildIndex($\{4,5,6\}$) $I_3 \leftarrow$ BuildIndex($\{7,8,9\}$) $I_4 \leftarrow$ BuildIndex($\{10\}$) $T_0 \leftarrow$ BuildIndex($\{3\}$) $T_1 \leftarrow T_0$, AddToIndex($\{2\}, T_0$)	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}\}$	$T_0 = \{d_3\}$ $T_1 = \{d_3, d_2\}$
11	AddToIndex($\{11\}, I_4$) Drop I_1, Rename T_0 as I_1	$\{d_3, d_2\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}\}$	Unchanged: T_1
12	AddToIndex($\{12\}, I_4$) Drop I_1, Rename T_1 as I_1	$\{d_3\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$	
13	$I_1 \leftarrow$ BuildIndex($\{13\}$) $T_0 \leftarrow$ BuildIndex($\{6\}$) $T_1 \leftarrow T_0$, AddToIndex($\{5\}, T_1$)	$\{d_{13}\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$	$T_0 = \{d_6\}$ $T_1 = \{d_6, d_5\}$
14	AddToIndex($\{14\}, I_1$) Drop I_2, Rename T_0 as I_2	$\{d_{13}, d_{14}\}$	$\{d_6, d_5\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$	Unchanged: T_1

Table 7: Example of index transitions in RATA

indexes d_3 and d_2, and T_{12} indexes d_3. On day 11, RATA indexes d_{11} in I_4 like WATA. Then it drops I_1 and replaces I_1 with T_{11} which contains entries for d_3 and d_2. The wave index thereby indexes d_2 through d_{11}. Similarly for subsequent days. We present the formal RATA* algorithm based on *WATA** in Appendix A as Figure 17. It it easy to see that we can extend RATA to enhance any WATA-based algorithm.

RATA performs more work than WATA but maintains hard windows. However it takes the same time as WATA to index a new day's worth of data after it is available. For instance, on day 13 additional work is done to build temporary indexes T_0 and T_1 to be used on subsequent days. However the operation $T_1 \leftarrow$ `BuildIndex({ 6 })` can be performed on day 11 since it depends only on d_6 which is already available on day 11. Similarly, $T_{15} \leftarrow T_{14}$ and `AddToIndex({5},` T_{14}) can be performed on day 12 since they depend on T_{14} and d_5, which are available after day 11 and 5 respectively. Hence if we use the above optimization, we would never need to index more than two days of data on any given day. The formal algorithm in Figure 17 does not show this optimization to keep the exposition simple.

5 Analytic Comparison of Wave Indexing Schemes

In the last few sections we proposed six algorithms to build wave indexes and three different ways for performing updates with each algorithm. We now present a simple analysis of the schemes. For our analysis, we assume that our n constituent indexes are stored on one disk. In case of multiple disks, our analysis can be extended in a similar fashion, but is not shown here. We consider in

Section 8 a few trends we expect in case of multiple disks.

Since our goal in this section is to identify general trends rather than to predict accurate performance numbers, we now propose some "coarse" parameters to compare our wave indexing schemes. The parameters we propose below are of three types (and sometimes of more than one type): (1) parameters that depend on the *hardware* used (such as disks used), (2) parameters that depend on the specific *application* (such as the average number of *TimedIndexProbes*), and (3) *implementation* parameters that depend on the how certain algorithms are implemented (such as which incremental indexing scheme is used).

1. **Disk Parameters:** Let $seek$ be the time to perform one seek. Let $Trans$ be the transfer speed in blocks per second to transfer disk blocks from disk to memory. These are both hardware parameters.

2. **Space Parameters:** For ease of analysis, we assume that the data size of all days is the same. It is easy to extend our analysis for the case of non-uniform data sizes. Let S be the space required to store a packed index of one day. Let S' be the space required to store a non-packed index of one day. We assume that the space required to store a packed index for d days is $S * d$, and the space to store a non-packed index for d days is $S' * d$.

   The parameter S is an application parameter since it depends on the size of data. The parameter S' depends on the application as well as on the implementation of incremental indexing. In this paper for concreteness, we assume we index incrementally using the *CONTIGUOUS* scheme of Faloutsos and Jagadish [FJ92]. Essentially, the *CONTIGUOUS* scheme allocates contiguous space for each search value. Each new index entry for a value is appended into the corresponding allocated space. When the allocated space is consumed, the scheme allocates a larger space which is g (growth factor) times larger than the previous space. It then copies over the index entries to the new space, and releases the old space. Similarly for deletion. Different implementations may use different g values and this clearly affects the value of S'.

3. **Constituent Index Operation Parameters:** Let Add be the time to incrementally index one day's data. Let Del be the time to incrementally delete one day's data from an index. Let $Build$ be the time to build an index of one day's data.

   All three depend on the application. Clearly the larger the amount of data in an application, the more expensive is each operation. All three depend on the implementation as well. For instance in *CONTIGUOUS*, if the initial space allocated for a new bucket is small, the time to add and delete is large because a lot of time is spent in copying the old bucket to a new location to allow for future growth.

4. **Update Technique Parameters:** Given an unpacked index for one day, let CP be the time to copy all buckets of that index into memory, and then flush them to another location on disk. Given a packed index for one day, let $SMCP$ be the time to copy all buckets of the index into memory, delete entries with expired timestamps, and then flush packed buckets to another location on disk. Both CP and $SMCP$ depend on the size of the data to be copied, and hence are application parameters.

5. **IndexProbe Parameters:** Given an index for one day, let c be the average size of a bucket (in disk blocks) for some random search value. We assume that the size of the bucket for d days is $d{*}c$. Let $Probe_{num}$ be the number of *TimedIndexProbes* and $Scan_{num}$ be the number of *TimedSegmentScans* in a day. Recall that *TimedIndexProbes* and *TimedSegmentScans* access between 1 and n constituent indexes depending on the specified time ranges. Let $Probe_{idx}$ and $Scan_{idx}$ be the average number of indexes a *TimedIndexProbe* and *TimedSegmentScan* access. All the above parameters are application parameters.

Some of the important performance measures we consider for each scheme are:

1. **Space Utilization:** First, we consider how much space is required to store the required window of days, i.e., during system *operation.* We also consider how much additional space is required when a new day is being indexed, i.e., during index *transitions.* This measure helps system administrators in deciding how many disks to buy, for instance.

2. **Query Response Time:** We consider how long it takes to execute *TimedIndexProbes* and *TimedSegmentScans*. In cases where users are sitting at a terminal waiting for a response, it is important to keep this measure low.

3. **Transition Time:** We consider how soon after a new day's data is available it is part of the wave index and ready for querying. In cases like the stock market where decisions may be made based on the new data, it may be critical to keep the transition time low. This measure may not be quite as important in data mining queries which look at general trends, for instance.

4. **Pre-Transition Time:** We consider how much time is spent each day as pre-computation in preparing temporary indexes. This indicates how long this pre-computation will interfere with user queries.

5. **Total Work:** During the course of the day, we need to index new data, maintain indexes and answer a stream of queries. We try to capture the work done by the system during the day into a single number by estimating resources consumed. We believe one good estimate

of work done is the time to index a given volume of new data, pre-compute new indexes, and in answering a set of user queries as if they were performed one after the other, without parallelism. For this, we first add the transition time and the pre-transition time. We then add the time to perform $Probe_{num}$ timed probes that access $Probe_{idx}$ indexes each, and the time to perform $Scan_{num}$ timed scans that access $Scan_{idx}$ indexes each.

In Table 8 we show the space utilization of the six algorithms if they are implemented with simple shadow updating. To simplify the equations in the table, we define $X = \frac{W}{n}$ and $Y = \frac{W-1}{n-1}$.

We now consider in detail the first two columns that show maximum and approximate average space required during system operation. We estimate the maximum space as follows: we compute the maximum number of days indexed in the constituent indexes as well as in the temporary indexes. We then multiply that number of days by S' (or S in case of *REINDEX*) to obtain the maximum space required. We estimate the average space averaged over the number of transitions in a similar fashion. For instance, we see that *REINDEX+* requires an average of $(W + \frac{X}{2}) * S'$ and a maximum of $(W + \lceil X \rceil - 1) * S'$ space while the system is in operation. This is because the constituent indexes in $REINDEX^{+}$ index W days. In addition, in $REINDEX^{+}$ *Temp* indexes at most $\lceil X - 1 \rceil$ days (Figure 14). However when averaged over time, *Temp* indexes about $\frac{X}{2}$ days.

$REINDEX^{++}$ also stores W days in its constituent indexes. In addition it maintains $\lceil X \rceil$ temporary indexes, and each temporary index T_i $(0 \leq i \leq \lceil X \rceil - 1)$ stores i days. That is, the temporary indexes will store a maximum of $\frac{1}{2} * \lceil X \rceil * \lceil X - 1 \rceil$ days. The average space for $REINDEX^{++}$ can be similary calculated. Similarly, WATA* stores a maximum of $W + \lceil Y - 1 \rceil$ (proved in Appendix B) days since it maintains soft windows. Hence it requires the maximum space indicated in the table. *RATA* maintains temporary indexes similar to $REINDEX^{++}$ (See Figures 15 and 17). *RATA* however stores a maximum of $\lceil Y \rceil$ days in a temporary index rather than $\lceil X \rceil$ days like $REINDEX^{++}$. Hence the difference between the formulae for *RATA* and $REINDEX^{++}$.

We now consider the third and fourth columns in Table 8 which report the additional space required during transitions. We estimate the additional space required as follows: if some constituent index needs to be shadowed for updating, we need space to store the shadow index. If some temporary index needs to be updated, we require no additional space since queries are executed only on constituent indexes. Hence the maximum additional space required during transitions is the size of the largest constituent index: this can be computed by multiplying the maximum number of days in a constituent index by S' (S in case of *REINDEX*). We estimate average additional space by averaging additional space requirements over the number of transitions. For instance, $REINDEX^{+}$ requires $\lceil X \rceil * S'$ space during transitions since the maximum number of days in an index is $\lceil X \rceil$. $REINDEX^{++}$ requires no additional space during a transition since it updates only

Measure/ Scheme	Max Space [Operation]	Avg Space [Operation]	Max Space [Transition]	Avg Space [Transition]
DEL	$W * S'$	$W * S'$	$\lceil X \rceil * S'$	$X * S'$
REINDEX	$W * S$	$W * S$	$\lceil X \rceil * S$	$X * S$
REINDEX$^+$	$(W + \lceil X \rceil) * S'$	$(W + \frac{X}{2}) * S'$	$\lceil X \rceil * S'$	$\frac{X}{2} * S'$
REINDEX^{++}	$(W + \frac{1}{2} * \lceil X \rceil * \lceil X - 1 \rceil) * S'$	$(W + \frac{X^2}{6} + \frac{X}{2}) * S'$	0	0
WATA*	$(W + \lceil Y - 1 \rceil) * S'$	$(W + \frac{Y}{2}) * S'$	$\lceil Y \rceil * S'$	$\frac{Y}{2} * S'$
RATA*	$(W + \frac{1}{2} * \lceil Y \rceil * \lceil Y - 1 \rceil) * S'$	$(W + \frac{Y^2}{6} + \frac{Y}{2}) * S'$	$\lceil Y \rceil * S'$	$\frac{Y}{2} * S'$

Table 8: Space utilization of wave indexes that use simple shadow updating ($X = \frac{W}{n}, Y = \frac{W-1}{n-1}$).

Measure/ Scheme	TimedIndexProbe	TimedSegmentScan
DEL	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S'}{Trans})$
REINDEX	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S}{Trans})$
REINDEX$^+$	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S'}{Trans})$
REINDEX^{++}	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S'}{Trans})$
WATA*	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W-1+\frac{1}{2}*\frac{W-1}{n-1}}{n} * \frac{S'}{Trans})$
RATA*	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S'}{Trans})$

Table 9: Query performance of wave indexes that use simple shadow updating.

temporary indexes.

The corresponding space utilization table for the algorithms when we use in-place updating (not shown) will look similar to Table 8 except that the space required by the six algorithms during index transitions will be zero. This is because no space is required for a shadow index. Similarly, the corresponding table for the algorithms when we use packed shadow updating (not shown) will look similar to Table 8, except that all S' will be replaced by S. The only other difference is that *REINDEX*$^{++}$ will now require an average of $X * S$ space since packed shadowing requires temporary indexes to be copied to a new location as well.

In Table 9, we present the time to perform one *TimedIndexProbe* and one *TimedSegmentScan* for the techniques implemented with simple shadow updating. We estimate the time to perform *TimedIndexProbe* as follows: we first compute the time to perform a probe on one index. This we compute by assuming each probe requires one seek followed by a transfer of the corresponding bucket from disk to memory. We then multiply the time to probe one index by $[1, n]$ to indicate the range of possible times a *TimedIndexProbe* can take depending on the specified time ranges. For instance, *REINDEX*$^{++}$ takes time $(seek + \frac{W}{n} * \frac{c}{Trans})$ for probing one index, and therefore

Measure/ Scheme	Precomputation	Transition
DEL	$\frac{W}{n} * CP + Del$	Add
$REINDEX$	0	$\frac{W}{n} * Build$
$REINDEX^{+}$	0	$\frac{W}{n} * CP + \frac{1}{2} * \frac{W}{n} * Add$
$REINDEX^{++}$	$\frac{1}{2} * \frac{W}{n} * Add$	Add
WATA*	0	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Add$
RATA*	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Add$	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Add$

Table 10: Maintenance performance of wave indexes that use simple shadow updating.

$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$ overall.

We estimate *TimedSegmentScan* as follows: we first compute the time to perform a scan of one index. This we compute by assuming each scan requires one seek followed by retrieving buckets of k days, where k is the number of days indexed in the index. Similar to *TimedIndexProbe* we then multiply the time to scan one index by $Scan_{idx}$ to indicate that the actual time depends on the specified time ranges. For instace $REINDEX^{++}$ takes time $(seek + \frac{S'}{Trans} * \frac{W}{n})$ to scan one index, and therefore $Scan_{idx} * (seek + \frac{S'}{Trans} * \frac{W}{n})$ overall.

The table for *TimedSegmentScan* and *TimedIndexProbe* for the algorithms implemented with in-place updating looks identical to Table 9. The corresponding table for the algorithms implemented with shadow updating also looks similar to Table 9 except that all S' are replaced with S.

In Table 10 we present the time it takes each day for adding a new day's data (Transition) and to index data in temporary indexes for future use (Pre-computation). We first consider transition time. In the table we see that, for example, every day $RATA$ copies a temporary index with an average (averaged across time) of $\frac{1}{2} * \frac{W-1}{n-1}$ days to a new location, and adds a new day to the index. This is performed as pre-computation to prepare simulating hard windows for the next few days. Similarly we see for Transition time that every day $RATA$ copies a constituent index with an average (averaged across time) of $\frac{1}{2} * \frac{W-1}{n-1}$ days to a shadow location, and adds the new day to the shadow.

The corresponding table for in-place updating looks similar except for fewer copy operations since additions and deletions are in-place. The corresponding table for packed shadow updating is presented in Table 11. We see that the time taken by the different algorithms is typically less than in simple shadow updating. This is because operations such as deletion are handled as part of the smart copy operation. Also we can show that with packed shadow updating, the incremental insert operations take time $Build$ rather than Add.

Measure/ Scheme	Precomputation	Transition
DEL	0	$\frac{W}{n} * SMCP + Build$
REINDEX	0	$\frac{W}{n} * Build$
REINDEX^{+}	0	$\frac{W}{n} * (SMCP + CP) + \frac{1}{2} * \frac{W}{n} * Build$
REINDEX^{++}	$\frac{1}{2} * \frac{W}{n} * SMCP + \frac{1}{2} * \frac{W}{n} * Build$	$\frac{1}{2} * \frac{W}{n} * SMCP + Build$
WATA*	0	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Build$
RATA*	$\frac{1}{2} * \frac{W-1}{n-1} * SMCP + Build$	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Build$

Table 11: Maintenance performance of wave indexing techniques that use packed shadow updating.

Parameter Type	Parameter	SCAM	WSE	TPC-D
Hardware	$seek$	14 msec	14 msec	14 msec
	$Trans$	10 MBps	10 MBps	10 MBps
Application	S	56 MB	75* MB	600 MB
	c	100 bytes	100* bytes	100* bytes
	$Probe_{num}$	$100,000^*$	$340,000^*$	0^*
	$Probe_{idx}$	n	n	–
	$Scan_{num}$	10^*	0^*	10^*
	$Scan_{idx}$	1	–	n
Implementation (*CONTIGUOUS*)	g	2.0	2.0	1.08
	$Build$	1686 secs	2276 secs	8406 secs
	Add	3341 secs	4678 secs	11431 secs
	Del	3341 secs	4678 secs	11431 secs
	S'	78.4 MB	105* MB	627 MB

Table 12: Parameter values chosen in case study.

6 Case studies

Given the relatively large number of implementation options, parameters, and performance metrics, it is difficult to draw concrete conclusions without looking at particular applications scenarios. In this section we present three application areas (copy detection, web engines, and warehousing), and within those we instantiate particular scenarios (e.g., data size, hardware speeds). For each scenario there are parameters we could directly measure, for example, how many Netnews articles need to be indexed each day for copy detection. Other parameters could be measured via experiments. For example, we evaluated S' by actually implementing the index algorithms and loading data into an index. (The number we obtain is realistic yet specific to our implementation.) However, other parameters values were "educated guesses," for instance, exactly how many queries to expect each day. Hence, the reader should not interpret the results of this section as absolute predictions, but rather as illustrations of performance trends and of the process to follow in selecting a particular wave index scheme. The scenarios we consider are:

1. **SCAM:** SCAM is a research prototype for finding copyright violators. One of the services we provide is to index articles of a set of newsgroups for a week to allow authors to search for recent illegal copies of their articles. In the following experiments for SCAM, we report results only for the case we implement wave indexes using simple shadowing (due to our space constraints here).

2. **Web search engine (WSE):** Several WSEs such as Altavista [Alt], SIFT [YGM95], Infoseek [Inf] and Dejanews [Dej] index Netnews articles in addition to a subset of the World-Wide-Web. We consider how a WSE should index articles for a sliding window of 35 days. In the study of a generic WSE, we report results for the case the indexes are implemented with simple shadowing as well as packed shadowing. (In-place updating is similar to simple shadowing.)

3. **TPC-D:** TPC-D is a benchmark from the Transaction Processing Council [TPC]. The benchmark models a decision support environment in which complex business-oriented queries are submitted against a large database. The queries may access large portions of the database and typically involve various operations such as joins, sorting and aggregation that may implemented with sequential scans and index probes. The benchmark defines two large relations *LINEITEM* and *ORDER*, and six other smaller relations. Similarly 17 queries have been prescribed.

   To simplify our experiments, we consider the following specific scenario. Say we build a wave index on relation *LINEITEM* on the *SUPPKEY* attribute for a window of the past 100 days. Every day the new additions to *LINEITEM* arrive as a batch based on the sales of the day. Let query $Q1$ (specified in the TPC-D benchmark as the "Pricing Summary Report") be the only query that is executed. In our experiments we used the data characteristics (in terms of distribution of tuples, sizes of tables, etc.) prescribed by the TPC-D benchmark. In the following experiments for TPC-D, we report results for the case the indexes are implemented with simple shadowing.

In Table 12 we report specific values we used for different parameters in our case study. The hardware parameters were chosen based on current technology. The application parameters we report are for data of one day. As stated earlier, we chose specific values for application parameters either based on experience, or based on educated guesses (denoted in the table with a $*$). As an example of the former, we computed S for SCAM by building a packed index on about $70,000$ text articles (in a day) and computed the space required. As an example of a guess, we estimated that commercial WSEs index about 100,000 articles per day. (SCAM indexes fewer since our NNTP server subscribes to fewer newsgroups).

We chose implementation parameters for SCAM as follows. First we implemented the *BuildIndex* scheme (as specified in Section 2.2) in C, and measured its running time on a DEC 3000 with an Alpha processor running OSF/1.0 and 96 MB of RAM. We then implemented and measured *AddToIndex* using the *CONTIGUOUS* [FJ92] incremental indexing scheme. To choose a good value for g in *CONTIGUOUS*, we executed *AddToIndex* to index words of one day's Netnews articles for several values of g. Based on the trade off between space consumption, S', and the time spent in copying buckets to new locations, we chose $g = 2$. For $g = 2$, we report S' and *Add* in Table 12. Since *DeleteFromIndex* is symmetric to *AddToIndex*, we assume that *Del* takes the same time as *Add*. The time to execute *BuildIndex* on the Netnews data is reported as *Build*.

In SCAM we expect to service about 100 user queries each day from authors and publishers to check if a given document was available as a Netnews article in the past week. Since for each query we expect to perform 100 *TimedIndexProbes* [SGM96] on the data of the last week, $Probe_{num} = 100,000$ and $Probe_{idx} = n$ ($W = 7$). In SCAM we also offer a *registration* service in which authors submit documents so they can be checked on a daily basis against the current day's Netnews articles. We can check the submitted documents against the current day's articles efficiently with a scan on the current day's index. We estimate (based on expected size of registration database) that we will need to perform about 10 segment scans each day on the current day's index (stored in one index). Hence $Scan_{num} = 10$ and $Scan_{idx} = 1$.

For the WSE, we estimated application and implementation values by scaling the corresponding values in SCAM by $100,000/70,000$ (based on relative number of articles). In a WSE, we expect about $170,000$ queries in a day for Netnews articles. This is roughly 1% of the number of queries per day in Altavista for the more popular web data [Alt]. Since each user query performs an average of two index probes (average length of a query is two words [Alt]) over all data in the window, we estimate $Probe_{num} = 340,000$ and $Probe_{idx} = n$.

For TPC-D, we repeated the experiments we did for SCAM and chose $g = 1.08$. This is because values for *SUPPKEY* in TPC-D are uniformly distributed, while words in SCAM's Netnews articles exhibit skewed Zipfian [Zip49] behavior. We assume about 10 complex analytical queries are run every day over data of the entire window to analyze trends. We assume these queries are executed using a scan over all the indices, and therefore $Scan_{num} = 10$ and $Scan_{idx} = n$.

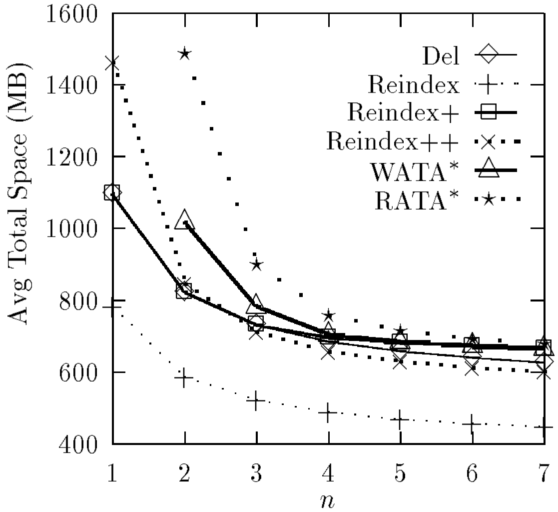

Figure 3: Average space required during SCAM's operation and transition (W = 7).

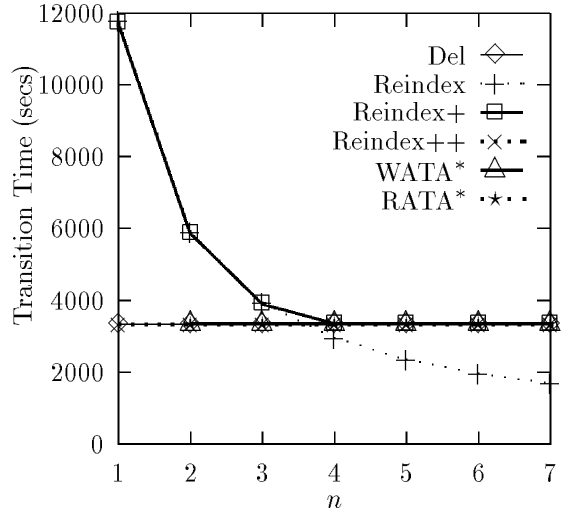

Figure 4: Average transition time for SCAM (W = 7).

We now present a few select graphs to indicate how the wave indexes perform in SCAM, WSE and TPC-D. As we describe these graphs, keep in mind that they illustrate performance metrics (e.g., space, work) and not qualitative measures such as ease of implementation. Recall that even if a scheme outperforms the other others in a given scenario, it may not be advisable either because (1) it requires complex code, or (2) it cannot be implemented with our favorite index package.

In Figure 3 we report the overall space required (averaged across transitions) by SCAM during system operation and transition (sum of column 2 and 4 in Table 3). We see that *REINDEX* requires the minimal amount of space. This is because (1) *REINDEX* maintains packed indexes that consume minimal space, and (2) *REINDEX* does not have any additional temporary indexes like *REINDEX*$^{+}$, *REINDEX*$^{++}$ or *RATA*. We also see that all schemes require less space as n increases. This is because each constituent index stores fewer days as n increases. Hence shadow indexes are smaller during transitions. Also in schemes like *REINDEX*$^{+}$, *REINDEX*$^{++}$ and *RATA*, there are fewer days in each temporary index as n increases. In schemes like *WATA* and *RATA*, the number of days in the soft window also decreases as n increases.

In Figure 4 we report the transition time to index new data in SCAM (column 2 of Table 4). There are two main factors that influence transition time: (1) does the scheme use *BuildIndex* or *AddToIndex* to add the new data? (2) for each scheme, how many days are reindexed using *BuildIndex* or incrementally indexed using *AddToIndex*? For instance, from Table 12 we see that if a scheme executes *BuildIndex* for one day, its transition time (1686 secs) is lower than another scheme that indexes *AddToIndex* (3341 secs) for the same day. However if the first scheme executes *BuildIndex* for 5 days, its transition time ($1686 * 5$ secs) is higher than the second scheme (3341 secs). Since *DEL*, *WATA*, *RATA* and *REINDEX*$^{++}$ execute *AddToIndex* during transitions and always incrementally index one day, we see that their transition times do not depend on n. However recall that *REINDEX* executes *BuildIndex* on $\frac{W}{n}$ days each day, which clearly depends on n. Hence we see that initially ($n \leq 3$) *REINDEX* performs poorly due to the cost of reindexing $\frac{W}{n}$ days each day. But for $n \geq 4$, the cost savings of executing a *BuildIndex* rather than an *AddToIndex* compensates for the cost of reindexing 1 or 2 days each day. *REINDEX*$^{+}$ performs the worst since it executes *AddToIndex* on an average of $\frac{1}{2} * \frac{7}{n}$ days each day.

In Figure 5, we report the total work done during the day by the different schemes in SCAM. The total work is very sensitive to the mix of queries and updates. For example, if we have many queries in a day, it is best to perform more work at update time in order to obtain an index that is better for queries (e.g., packed, small n). In the SCAM scenario, the opposite is true: the number of copy detection queries is relatively small compared to the number of documents indexed.

In Figure 5 again we see that *REINDEX* performs poorly for small n but is very efficient for large n. This is because of the relative cost of reindexing some constituent index each day versus the savings due to using *BuildIndex*, and faster scans due to packed indexes. We see from the figure that the reindexing cost in *REINDEX* dominates for small n, while for large n the savings dominate. We also see that *DEL*, *WATA* and *RATA* are relatively stable since they incrementally add and delete a small constant number of days each day. They increase slowly with n since *TimedIndexProbes* need to probe an increasing number of indexes.

From Figures 3, 4 and 5, we recommend using *REINDEX* for SCAM with $n = 4$ indexes. We recommend $n = 4$ as a compromise value between the following two conflicting factors: (1) as n increases, *REINDEX* performs better than the other schemes and (2) as n increases, the response time of *TimedIndexProbes* increases since more constituent indexes need to be probed. We choose $n = 4$ since we would like to keep the user response time low, and since we see from the graphs that we obtain diminishing returns for our performance measures for $n \geq 4$.

We now consider the performance of our wave indexes for WSE. We observed trends similar to Figure 3 and 4 for the average space during transitions and average transition time for WSE as well as TPC-D (not reported). In Figure 6, we report the total work done by WSE with packed shadowing for $W = 35$. We see that due to significantly higher query volume and window size, *REINDEX* that performed best in SCAM, now in fact performs the worst. *REINDEX* does poorly for small n for the reasons described earlier. But *REINDEX* continues to do poorly even as n increases since the cost savings of reindexing fewer days in a constituent index is offset by the

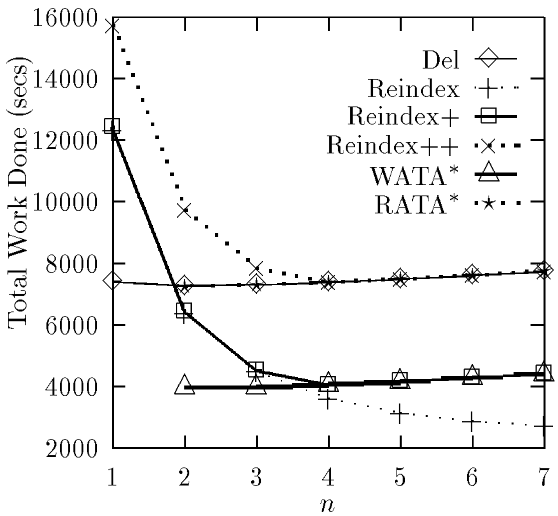

Figure 5: Average work done by SCAM during day (W = 7).

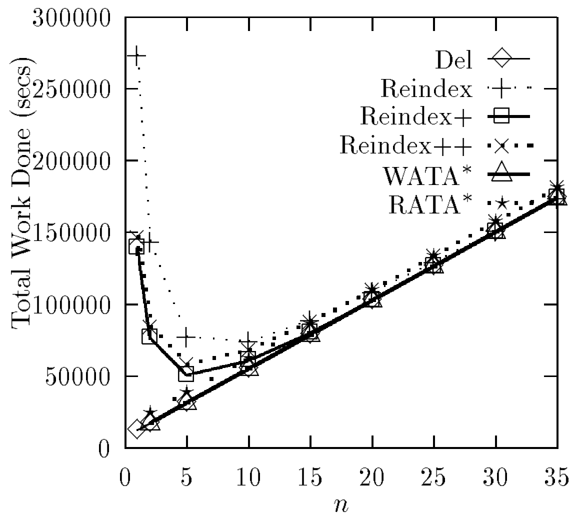

Figure 6: Average work done by WSE during day (W = 35).

increased cost of more probes executed for a *TimedIndexProbe*. In this case *DEL*, *WATA* and *RATA* perform the minimal amount of work when $n \leq 2$. This is because they always perform minimal amount of work in indexing new data, and also because n is small enough to service *TimedIndexProbes* cheaply.

From Figure 6, we recommend using *DEL* ($n = 1$) with packed shadow updating for a WSE. This is because for $n = 1$, the response time for user queries is low. Also, *DEL* performs minimal total work.

Similarly in Figure 7 we report the total work done by the different algorithms in the TPC-D case when packed shadowing is used. (We resized the graph since *REINDEX* performs very poorly.) In this example we see again that *DEL* ($n = 1$) and *WATA** ($n = 2$) perform the best, while *REINDEX* performs the worst. In Figure 8, we report the total work done by the different algorithms in the TPC-D case when simple shadowing is used. While we see similar trends to Figure 7, we see how the work done is significantly less in case of packed shadowing. This is of course because packed shadowing does deletion while copying, and because segment scans are efficient due to the packed constituent indexes. For simple shadowing, we see that *WATA* performs the minimal amount of work among the schemes, and performs less work as n increases. This is because the number of expired days stored in the constituent indexes decreases as n increases, and segment scans are more efficient. Also we would like to point out that *WATA* performs significantly better than *DEL* and *RATA*: *WATA* requires upto $10,000$ seconds (about 3 hours) less time than

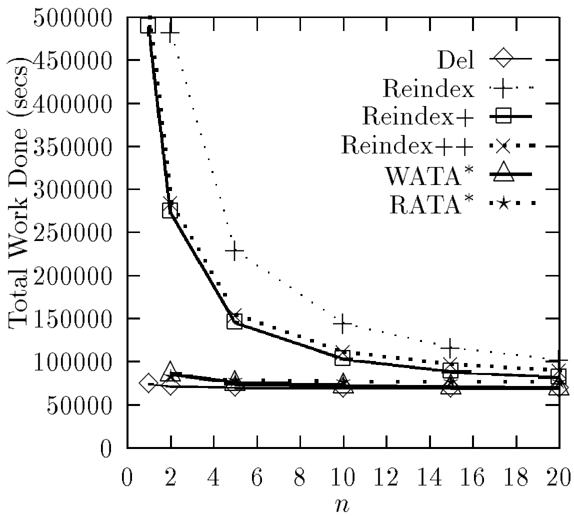

Figure 7: Work done in TPC-D (packed shadowing) during day (W = 100).

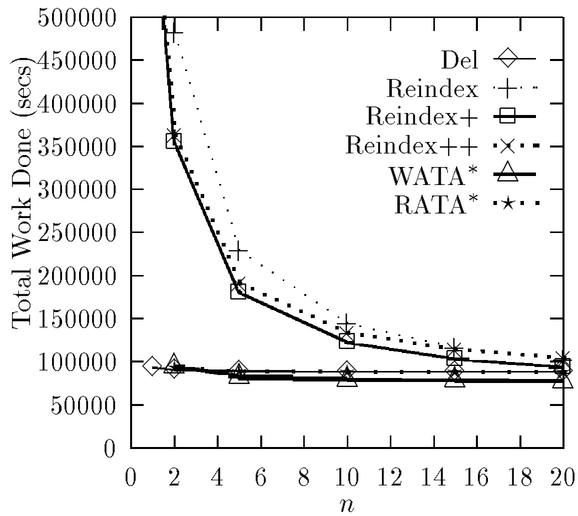

Figure 8: Work done in TPC-D (simple shadowing) during day (W = 100).

both *DEL* and *RATA*. This is not clear from the graph due to the ranges displayed in the vertical axis. In-place updating of course performs like simple shadowing in all measures except it uses less space during index transitions, and is more complex to implement.

From Figures 7 and 8 we recommend the following schemes (in order of preference) to be used for TPC-D. If packed shadowing can be implemented, use *DEL* ($n = 1$) since it has the best user response time and since it performs minimal work. If packed shadowing cannot be implemented (since some legacy system needs to be used), implement *WATA* ($n = 10$). This is because it performs significantly less work (about $9,000$ seconds worth) than *DEL*. Beyond $n \geq 10$ the savings in *WATA* are marginal while increasing query response time. If hard windows are required, we recommend *RATA* ($n = 10$) since it performs the same work as *DEL*, and is not as complex to implement as *DEL*.

In Figure 9 we consider the question of how the schemes scale when the required window size increases from 4 days to 6 weeks. Recall that the reindexing schemes index $O(\frac{W}{n})$ days each day, while *DEL*, *WATA* and *RATA* index a small constant number of days each day. Hence we see that, for a given n, as W increases the three reindexing based schemes do not scale while *DEL*, *WATA* and *RATA* scale very well. So if in SCAM we expect to index (say) a window of 14 days some time in the future, it may be worth the effort now to implement *WATA** rather than *REINDEX*.

If we did expect to index a window of 14 days in the future, we need to consider how much data may have increased by then. In Figure 10 we consider the case in SCAM when the number

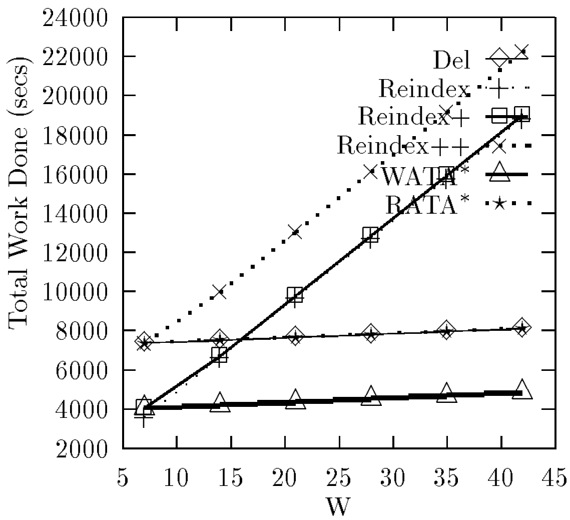

Figure 9: Work done during day by SCAM with W (n = 4).

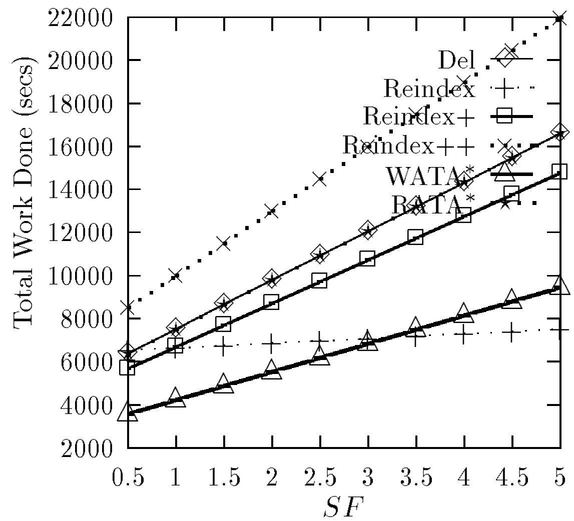

Figure 10: Work done during day by SCAM with SF (W = 14, n = 4).

of netnews articles per day increases from 70,000 to $70,000 * SF$, where $0.5 \leq SF \leq 5$ is the scale factor. We see that *REINDEX* scales the best for this measure since it does not use expensive incremental indexing schemes like *CONTIGUOUS*. However, *WATA** still performs best when $SF \leq 3$. So if we expect the data in the future to increase significantly (i.e., the number of Netnews articles per day becomes $\geq 70,000 * 3$), it may be actually be best to implement *REINDEX* rather than *WATA**! This shows us that before choosing a particular scheme to implement we should consider carefully both (1) whether we may ever want a larger window size, and (2) if so, how much do we expect data to increase by.

Finally we consider how our WATA* scheme performs when we index 200 days worth of Usenet data, collected between June and December 1997. The purpose of this experiment is to understand how much space overhead the WATA* scheme incurs to support lazy deletion. Specifically we are interested in the *index size* ratio, which we define to be the maximum index size ever required by the lazy WATA* scheme divided by the maximum index size ever required if we use eager deletion strategies such as REINDEX. We report this ratio in Figure 11 for the 200 days of data, as n varies for $W = 7$. For instance, when $n = 4$ the index size ratio is 1.24 which indicates the WATA* scheme costs 24% of overhead in storage. We note that the space overhead for WATA* appears tolerable (≤ 1.6), and decreases as n increases – we believe this makes the case stronger for WATA based indexing.

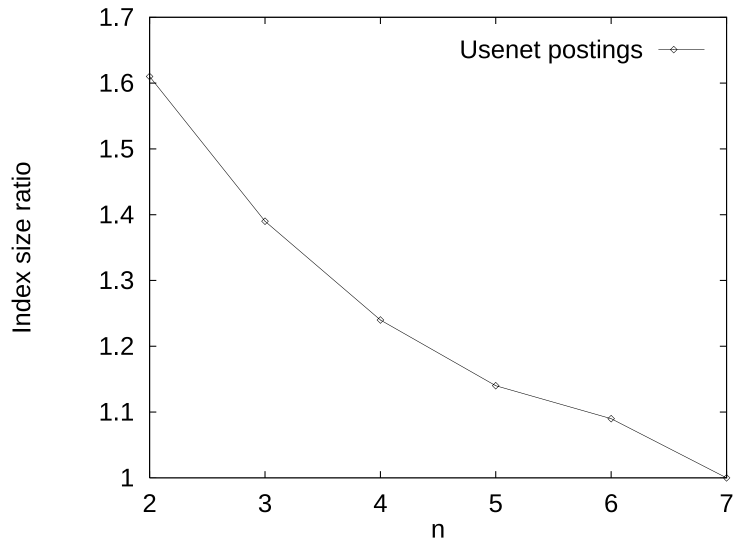

Figure 11: Index size ratio with n ($W = 7$).

7 Related Work

Brown et. al. [BCC94], Cutting and Pedersen [CP90], Faloutsos and Jagadish [FJ92] and Tomasic et. al. [TGMS94] all consider how to incrementally index a growing corpus for fast information retrieval. They do not consider the case where a sliding window of documents is indexed. However their work is orthogonal to us: in fact we can implement *AddToIndex* and *DeleteFromIndex* using any of the above schemes. Indeed in our case study we implemented *AddToIndex* and *DeleteFromIndex* using the *CONTIGUOUS* scheme from [FJ92].

Chandra and Segev [CS93] consider how to manage temporal financial data in the context of an extensible database. Their work is also orthogonal to ours and we expect our schemes will be helpful to index the *calendar* objects and time-series data they consider.

There has been a significant body of work in indexing temporal data. Salzberg and Tsotras [ST94] provide an excellent survey of work in this area. Index structures such as *AP-Trees* [GS93], *Time Index* [EKW91], *Monotonic B+Trees* [EKW91], *Snapshot Index* [TK93], *Segment R-Trees* [KS91] and *Time Split B-Trees* are specific enhancements of well-known index structures such as B+Trees and R-Trees for indexing time-series data. Each of the above is optimized to answer specific kinds of *time-slice* and *range-time-slice* queries [ST94]. Also they handle arbitrary insertion and expiry times of data. However they handle expiry of data by *logical deletion* where data is not physically deleted at time of expiry [ST94]. An asynchronous “vaccuuming” process runs in the background to delete data [KS91, ST94].

Our work differs from the above temporal indexing schemes in that we index sliding windows of data rather than data with arbitrary insertion and expiry times. This assumption helps us carefully organize our indexes and make several performance optimizations. Also our schemes are independent of the underlying index structures used, and hence can be used on top of widely available index structures (such as B+Trees, ISAMs etc.). Hence applications that need sliding windows can implement our schemes using their favorite indexing package. Indeed our schemes can also be used in conjunction with the above temporal index structures and replace the asynchronous deletion process for one of several reasons we identified in this paper (such as efficiency of batched deletes or better structured index).

There has been a lot of work in reorganization of traditional indexes [Wie87], e.g., how to decide when an index has deteriorated so much that it makes sense to throw away the index and rebuild it from scratch. We exploit the semantics of a sliding window to organize our indexes carefully: this helps our schemes (except *DEL* with $n = 1$) to automatically reorganize themselves on a continuing basis.

8 Conclusions and Future Work

Several applications require indexing data of a past window of days. For this we proposed several techniques to build *wave indices*. We then analyzed these schemes and showed experimentally under a variety of scenarios how the schemes perform for different volumes of input data and query patterns. Our results indicate that each of our wave indexing schemes has advantages and could be useful in some specific scenario, depending on what the central performance metrics are, and on how much code we can afford to write.

In the future, we plan to consider how the different wave indices perform when multiple disks are used. In particular, if n matches the number of disks, indexing can be parallelized easily. Also building new constituent indices on separate disks avoids contention. Hence wave indices will have several advantages over monolithic indices when we use multiple disks. Since there are several interesting ways in which a given number of disks can be allocated to the constituent indices, we are planning to evaluate some of the tradeoffs.

Appendix A: Algorithms

Each of the following algorithms have two important states, **Start** and **Transition**. Initially when the wave index is to be built for the 1^{st} W days, the operations in **Start** will be executed. Each subsequent day, the new day's data is indexed using the operations in **Transition**.

Algorithm for $DEL[W, n$: **Integer**]

- **Globals:**
  - $I_1, I_2, \ldots, I_n$: Index// `Constituent indexes`
  - $Days$: Array[1..n] of Set of Integers // `Time-sets of indexes`
- **Start**[$d_1, d_2, \ldots, d_W$: Data]
  Local Variables: low, i, j: Integer
  1. $low = 1$
  2. For all $i = 1, 2, \ldots, W \ mod \ n$
     // 1^{st} W mod n `time-sets have` $\lceil \frac{W}{n} \rceil$ `days`
     (a) $Days[i] \leftarrow \cup_{j=low}^{low+\lceil W/n \rceil}\{j\}$
     (b) $I_i =$*BuildIndex*$(Days[i])$
     (c) $low = low + \lceil W/n \rceil + 1$
  3. For all $i = W \ mod \ n+1, W \ mod \ n+2, \ldots, n$
     // `Other time-sets have` $\lfloor \frac{W}{n} \rfloor$ `days`
     (a) $Days[i] \leftarrow \cup_{j=low}^{low+\lfloor W/n \rfloor}\{j\}$
     (b) $I_i =$*BuildIndex*$(Days[i])$
     (c) $low = low + \lfloor W/n \rfloor + 1$
- **Transition**[d_{new}: Data]
  1. Let I_j be the index containing data of d_{new-W}.
  2. *DeleteFromIndex*(d_{new-W}, I_j)
  3. *AddToIndex*(d_{new}, I_j)

Figure 12: Algorithm for *DEL*

Algorithm for $REINDEX[W, n$: **Integer**]

- **Globals:**
  - $I_1, I_2, \ldots, I_n$: Index // `Constituent indexes`
  - $Days$: Array[1..n] of Set of Integers // `Time-sets of indexes`
- **Start**[$d_1, d_2, \ldots, d_W$: Data]
  1. Same as Start for *DEL*
- **Transition**[d_{new}: Data]
  Local Variables: j: Integer
  1. Let I_j be the index containing data of d_{new-W}.
  2. $Days[j] = Days[j] - \{new - W\} \cup \{new\}$ // `Updating` j^{th} `time-set`
  3. $I_j \leftarrow$*BuildIndex*$(Days[j])$ // `Rebuilding index`

Figure 13: Algorithm for *REINDEX*:

Algorithm for $REINDEX^{+}$[W, n: **Integer**]

- **Globals**
  - $I_1, I_2, \ldots, I_n$: Index `// Constituent indexes`
  - $Temp$: Index `// Temporary index`
  - $Days$: Array[1..n] of Set of Integers `// Time-sets of indexes`
  - $DaysToAdd$: Set of Integers `// Tracks days to add to temporary index`
- **Start**[$d_1, d_2, \ldots, d_W$: Data]
  1. Same as DEL
  2. $Temp \leftarrow \phi$
- **Transition**[d_{new}: Data]

  Local Variables: j: Integer
  1. Let I_j be the index containing data of d_{new-W}
  2. If $Temp = \phi$

     `// As in days 11 and 16 in Table 5`
     (a) $DaysToAdd \leftarrow Days[j] - \{new - W\}$
     (b) $Temp, I_j \leftarrow BuildIndex(d_{new})$
     (c) $AddToIndex(DaysToAdd, I_j)$
  3. Else If $DaysToAdd = \phi$

     `// As in day 15 in Table 5`
     (a) $I_j \leftarrow Temp$
     (b) $AddToIndex(d_{new}, I_j)$
     (c) $Temp \leftarrow \phi$
  4. Else

     `// As in days 12, 13, 14 in Table 5`
     (a) $AddToIndex(d_{new}, Temp)$
     (b) $I_j \leftarrow Temp$
     (c) $AddToIndex(DaysToAdd, I_j)$
  5. $Days[j] \leftarrow Days[j] - \{new - W\} \cup \{new\}$ `// Updating` j^{th} `time-set`
  6. $DaysToAdd \leftarrow DaysToAdd - \{new - W + 1\}$

Figure 14: Algorithm for REINDEX^{+}

Algorithm for $REINDEX^{++}[W, n$**: Integer]**

- **Globals**
  - $I_1, I_2, \ldots, I_n$: Index `// Constituent indexes`
  - $T_1, T_2, \ldots, T_{\lceil W/n \rceil}$: Index `// Temporary indexes`
  - $Days$: Array[1..n] of Set of Integers `// Time-sets of indexes`
  - $DaysToAdd$: Set of Integers `// Tracks days to be added to temporary indexes`
  - $TempUsed$: Integer `// Tracks next temporary index that replaces a constituent index`
- **Initialize**[$\cup_{l=j}^{k}\{d_l\}$: Set of Integers]
  Local Variables: i: Integer
  1. $T_0 \leftarrow \phi$, $T_1 \leftarrow BuildIndex(d_k)$
  2. For i from $j+1$ to k
     (a) $T_{i-j+1} \leftarrow T_{i-j}$
     (b) $AddToIndex(d_{k-(i-j)}, T_{i-j+1})$
  3. $TempUsed = k - j + 1$
  4. $DaysToAdd \rightarrow \phi$
- **Start**[$d_1, d_2, \ldots, d_W$: Data]
  1. Same as DEL
  2. *Initialize(Days[1]* - $\{1\}$*)*
- **Transition**[d_{new}: Data]
  Local Variables: j, j': Integer
  1. Let I_j be the index containing data of d_{new-W}
  2. If $TempUsed = 0$
     `// As in day 10 and 15 in Table 6`
     (a) $AddToIndex(d_{new}, T_0)$
     (b) Rename T_0 as I_j
     (c) Let $I_{j'}$ be the index containing data of $d_{new-W+1}$
     (d) $Initialize$(Days[j'] $- \{new - W + 1\}$)
  3. Else
     `// As in days 11, 12, 13, 14 in Table 6`
     (a) $DaysToAdd \leftarrow DaysToAdd \cup \{new\}$
     (b) $AddToIndex(d_{new}, T_{TempUsed})$
     (c) Rename $T_{TempUsed}$ as I_j
     (d) $TempUsed = TempUsed$ - 1
     (e) $AddToIndex(DaysToAdd, T_{TempUsed})$
  4. $Days[j] = Days[j] - \{new - W\} \cup \{new\}$

Figure 15: Algorithm for $REINDEX^{++}$

Algorithm for $WATA^*[W, n$**: Integer**]

- **Globals:**
  - $I_1, I_2, \ldots, I_n$: Index `// Constituent indexes`
  - $Days$: Array[1..n] of Set of Integers `// Time-sets of indexes`
  - Z: Array[1..n] of Integers `// Sizes of indexes`
  - $last$: Integer `// Tracks last modified index`
- **Start**[$d_1, d_2, \ldots, d_W$: Data]
  Local Variables: low, i, j: Integer
  1. $low = 1$
  2. For all $i = 1, 2, \ldots, (W-1)\ mod\ (n-1)$,
     `//` 1^{st} $(W-1)mod(n-1)$ `time-sets have` $\lceil \frac{W-1}{n-1} \rceil$ `days`
     (a) $Days[i] \leftarrow \cup_{j=low}^{low+\lceil (W-1)/(n-1) \rceil}\{j\}$
     (b) $I_i = BuildIndex(Days[i])$
     (c) $low = low + \lceil (W-1)/(n-1) \rceil + 1$
     (d) $Z_i = \lceil (W-1)/(n-1) \rceil$
  3. For all $i = (W-1)\ mod\ (n-1)+1, (W-1)\ mod\ (n-1)+2, \ldots, n-1$,
     `// Other time-sets till` $n-1$ `have` $\lfloor \frac{W-1}{n-1} \rfloor$ `days`
     (a) $Days[i] \leftarrow \cup_{j=low}^{low+\lfloor (W-1)/(n-1) \rfloor}\{j\}$
     (b) $I_i = BuildIndex(Days[i])$
     (c) $low = low + \lfloor (W-1)/(n-1) \rfloor + 1$
     (d) $Z_i = \lfloor (W-1)/(n-1) \rfloor$
  4. $Days[i] \leftarrow \{W\}$ `// Last time-set has` W^{th} `day`
  5. $I_n \leftarrow$BuildIndex(b_W)
  6. $last = n$
- **Transition**[d_{new}: Data]
  1. Let I_j be the index containing data of d_{new-W}
  2. If $\sum_{i=1, i\neq j}^{n} Z_i = W - 1$, perform *ThrowAway* else perform *Wait*.
     (a) *ThrowAway:*
         `// Throw away` j^{th} `index`
         i. $DropIndex(I_j)$
         ii. $I_j \leftarrow \phi$
         iii. $I_j \leftarrow$ *BuildIndex(new)*
         iv. $Days[j] \leftarrow \{new\}$, $Z_j = 1$
         v. $last = j$
     (b) *Wait:*
         `// Add new day to last modified index`
         i. *AddToIndex(*d_{new}, I_{last}*)*
         ii. $Z_{last} = Z_{last} + 1$
         iii. $Days[last] = Days[last] \cup \{new\}$

Figure 16: Algorithm for $WATA^*$

Algorithm for $RATA^*[W, n$: **Integer**]

- **Globals:**
  - $I_1, I_2, \ldots, I_n$: Index `// Constituent indexes`
  - $Days$: Array[1..n] of Set of Integers `// Time-sets of indexes`
  - $Z[1..n]$: Integer `// Sizes of indexes`
  - $TempUsed$: Integer `// Tracks next temporary index that will replace a constituent index`
  - $last$: Integer `// Tracks last modified index`
- **Initialize**[$\cup_{l=j}^{k}\{d_l\}$: Set of Integers]

  Local Variables: i: Integer
  1. $T_1 \leftarrow BuildIndex(d_k)$
  2. For i from $j+1$ to k
     (a) $T_{i-j+1} \leftarrow T_{i-j}$
     (b) $AddToIndex(d_{k-(i-j)}, T_{i-j+1})$
  3. $TempUsed = k - j + 1$
- **Start**[$d_1, d_2, \ldots, d_W$: Data]

  Local Variables: low, i, j: Integer
  1. Same as $WATA^*$
  2. $Initialize(Days[1] - \{1\})$
- **Transition**[d_{new}: Data]

  Local Variables: j': Integer
  1. Let I_j be the index containing data of d_{new-W}
  2. If $\sum_{i=1, i\neq j}^{n} Z_i = W - 1$, perform $ThrowAway$ else perform $Wait$.
     (a) *ThrowAway:*
        i. Same steps as $ThrowAway$ in $WATA^*$
        ii. Let $I_{j'}$ be the index containing data of $d_{new-W+1}$
        iii. Initialize($Days[j'] - \{new - W + 1\}$) `// Preparing temporary indexes for next cycle`
     (b) *Wait:*
        i. $AddToIndex(d_{new}, I_{last})$
        ii. $Days[last] = Days[last] \cup \{new\}$
        iii. Drop I_1
        iv. Rename $T_{TempUsed}$ as I_j `// Using temporary index to simulate hard window`
        v. $Days[j] = Days[j] - \{new - W\}$
        vi. $TempUsed = TempUsed$ - 1

Figure 17: Algorithm for $RATA^*$

Appendix B: Goodness of WATA*

We define family $\mathcal{F}$ to be the set of *WATA*-based algorithms that construct and maintain a wave index, Θ, for W days. Recall that these algorithms use only the following operations: (1) *AddToIndex*, to add a day to a constituent index, (2) *DropIndex*, to remove a constituent index from Θ, and (3) *AddIndex*, to add a constituent index to Θ.

Since constituent indices change each day, we use $I_j^{(i)}$ to refer to index I_j on day i. We define $|I_j^{(i)}|$ to be the number of days indexed in I_j, $j = 1, 2, \ldots, n$ on day i. We use $s(I_J^{(i)})$ to denote the corresponding storage required by the index on day i.

Index length measure

We define *length(T)* of Θ on day i to be the total number of days indexed in the constituent indices on day i, i.e., $\sum_{j=1}^{n} |I_j^{(i)}|$. We define *max length* of Θ to be the maximum length of Θ during its life-time, i.e., $max_{i=0}^{\infty} length(i)$. We define *residual length* of Θ to be $max\ length - W$. We define *waste*, $w(I_j^{(i)})$, for each constituent index I_j, $j = 1, 2, \ldots, n$, to be the number of days indexed in I_j that are older than the required window on day i.

We now prove that $WATA^*$ minimizes max length of Θ for a given W and n, among the set of algorithms in $\mathcal{F}$. We do this in two steps. First we consider an abstract algorithm, OPT, in $\mathcal{F}$ that minimizes the max length of Θ: we compute the lower bound on max length for such an optimal algorithm in Theorem 1. Then we show in Theorem 2 that $WATA^*$ achieves the same lower bound, thereby making it optimal.

Theorem 1: Let OPT be an algorithm in $\mathcal{F}$ that minimizes the max length of Θ. Algorithm OPT cannot build a Θ for a given W and n with maximum length less than $W + \lceil \frac{W-1}{n-1} \rceil - 1$.

Proof of Theorem 1: Let C be the residual length of OPT for Θ. By definition, on some day, k, the number of days indexed in Θ by OPT will be $W + C$. Since these $W + C$ days need to be split across the n constituent indices, there must be some index I_{big}, $1 \leq big \leq n$ with

$$|I_{big}^{(k)}| \geq \frac{W+C}{n} \tag{8.1}$$

Also, by definition

$$\begin{aligned} C &= max_{t=1}^{\infty} \sum_{j=1}^{n} w(I_j^{(t)}) && (8.2) \\ &\geq w(I_{big}^{(t)}), t \geq 1 && (8.3) \end{aligned}$$

Observe that every index I_j has to be dropped at some point in time. This is clear since C will otherwise be ∞. Now consider the earliest day, l, $(l \geq k)$ I_{big} is ready to be dropped. On the previous day, $l-1$, $w(I_{big}^{(l-1)})$ is $|I_{big}^{(l)}|-1$ since there is exactly one day in $I_{big}^{(l-1)}$ that does not expire until the next day.

Since I_{big} grows monotonically between day k and day $l-1$, we see that

$$\begin{aligned}
C &\geq w(I_{big}^{(l-1)}) &(8.4)\\
&= |I_{big}^{(l-1)}| - 1 &(8.5)\\
&\geq |I_{big}^{(k)}| - 1 &(8.6)\\
&\geq \frac{W+C}{n} - 1 &(8.7)\\
n*(C+1) &\geq W+C &(8.8)\\
C &\geq \frac{W-n}{n-1} &(8.9)\\
&= \frac{(W-1)-(n-1)}{n-1} &(8.10)\\
&= \frac{W-1}{n-1} - 1 &(8.11)
\end{aligned}$$

That is, no algorithm in $\mathcal{F}$ can have a wave index with max length below $W + \frac{W-1}{n-1} - 1$. Since the max length of Θ should be an integer (number of days is an integer), no algorithm in $\mathcal{F}$ can have a max length below $W + \lceil \frac{W-1}{n-1} - 1 \rceil$. $\square$

Theorem 2: The maximum length of $WATA^*$ (Figure 16) is $W + \lceil \frac{W-1}{n-1} - 1 \rceil$.

Proof of Theorem 2:

The time-sets constructed by OPT have consecutive days. That is, if days i and $i+2$ are assigned to some I_j, then $i+1$ is also assigned to I_j. Also $WATA^*$ drops a constituent index when all days in its time-set has expired. Hence if we maintain a window of W consecutive days, there can be at most one I_j, such that $w(I_j) > 0$.

Also when the wave index is constructed initially, the number of days in each constituent index is at most $\lceil \frac{W-1}{n-1} \rceil$. Observe that $WATA^*$ maintains this throughout the life-time of the wave-index.

Consider the day before some I_j is to be thrown away. At that point, only one day is alive and the other $|I_j| - 1$ have expired. Hence the maximum $w(I_j)$ is $\lceil \frac{W-1}{n-1} \rceil - 1$. Therefore the maximum length of any wave index constructed using $WATA^*$ is $W + \lceil \frac{W-1}{n-1} \rceil - 1$. $\square$

Index size measure

Theorem 3 *WATA** has a competitive ratio of 2.0 with respect to optimal among the algorithms in $\mathcal{F}$.

Proof of Theorem 3: Let M be the maximum value of index size required to store W consecutive days of data, across the entire duration of indexing. Clearly, the optimal algorithm OPT requires at least M storage space to handle the largest window. During this period, WATA* requires M storage as well to store the W days. In addition, WATA* may require an additional S storage to store the residual days that may have expired but are in the same index as some day that has not yet expired. However, this space S clearly cannot exceed M, since M is the maximum storage required for any time window W. Hence $S \leq M$. Therefore, WATA* has a competetive ratio of 2.0.

References

[Alt] AltaVista. Altavista search engine. `http://altavista.digital.com`.

[BCC94] E.W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference of Very Large Databases (VLDB'94)*, pages 192 – 202, Santiago, Chile, September 1994.

[CP90] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the International Conference on Information Retrieval (SIGIR'90)*, pages 405 – 411, Minneapolis, Minnesota, January 1990.

[CS93] R. Chandra and A. Segev. Managing temporal financial data in an extensible database. In *Proceedings of the 19th International Conference of Very Large Databases (VLDB'93)*, pages 302 – 313, Dublin, Ireland, August 1993.

[Dej] DejaNews. Dejanews news reearch service. `http://www.dejanews.com`.

[EKW91] R. Elmasri, Y. Kim, and G. Wuu. Efficient implementation techniques for the Time Index. In *Proceedings of 7th IEEE International Conference of Data Engineering*, pages 102 – 111, 1991.

[FJ92] C. Faloutsos and H. Jagadish. On b-tree indices for skewed distributions. In *Proceedings of the 18th International Conference of Very Large Databases (VLDB'92)*, pages 363 – 374, Vancouver, British Columbia, Canada, September 1992.

[GS93] H. Gunadhi and A. Segev. Efficient indexing methods for temporal relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):496–509, 1993.

[Inf] Infoseek. Infoseek search engine. `http://www.infoseek.com`.

[Jan95] J. Jannink. Implementing deletion in b+trees. *SIGMOD Record*, 24:33 – 38, March 1995.

[KMRV97] J. Kleinberg, R. Motwani, P. Raghavan, and S. Venkatasubramanian. Storage management for evolving databases. In *Proceedings of 38th IEEE Symposium on Foundations of Computer Science (FOCS'97)*, June 1997.

[KS91] C. Kolovson and M. Stonebraker. Segment Indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proceedings of ACM International Conference on Management of Data (SIGMOD'91)*, pages 138–147, May 1991.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Pfe] U. Pfeifer. Freewais-sf Information Retrieval package. In `http://ls6-www.informatik.uni-dortmund.de/ir/projects/freeWAIS-sf/`.

[Ros96] P. E. Ross. Cops versus robbers in cyberspace. *Forbes Magazine*, pages 134 – 139, September 9 1996.

[SB] G. Salton and C. Buckley. The SMART information retrieval system. `http://ftp.cs.cornell.edu/pub/smart`.

[SGM96] N. Shivakumar and H. Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of 1st ACM Conference on Digital Libraries (DL'96)*, Bethesda, Maryland, March 1996.

[ST94] B. Salzberg and V.J. Tsotras. A comparison of access methods for time-evolving data. In *Technical Report: NU-CCS-94-21*, Northeastern University, 1994.

[TGMS94] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of 1994 ACM International Conference on Management of Data (SIGMOD'94)*, Minneapolis, Minnesota, May 1994.

[TK93] V. J. Tsotras and N. Kangelaris. The Snapshot-Index, an I/O optimal access method fpr timeslice queries. *CATT-Technical report 93-68*, December 1993.

[TPC] TPC-Committee. Transaction processing council (TPC). `http://www.tpc.org`.

[Wie87] G. Wiederhold. *File organization for database design*. McGraw-Hill (New York, NY), March 1987.

[YGM95] T. Yan and H. Garcia-Molina. Sift – a tool for wide-area information dissemination. In *Proceedings of USENIX*, 1995.

[Zip49] G.K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley Press, Cambridge, Massachusetts, 1949.