Indexing Time-Evolving Data With Variable Lifetimes*

Lukasz Golab
School of Comp. Sci.
Univ. of Waterloo, Canada
lgolab@uwaterloo.ca

Piyush Prahladka†
Google, Inc.
Bangalore, India
piyushp@google.com

M. Tamer Özsu
School of Comp. Sci.
Univ. of Waterloo, Canada
tozsu@uwaterloo.ca

Abstract

Many applications store data items for a pre-determined, finite length of time. Examples include sliding windows over on-line data streams, where old data are dropped as the window slides forward. Previous research on management of data with finite lifetimes has emphasized on-line query processing in main memory. In this paper, we address the problem of indexing time-evolving data on disk for off-line analysis. In order to reduce the I/O costs of index updates, existing work partitions the data chronologically. This way, only the oldest partition is examined for expirations, only the youngest partition incurs insertions, and the remaining partitions "in the middle" are not accessed. However, this solution is based upon the assumption that the order in which the data are inserted is equivalent to the expiration order, which means that the lifetime of each data item is the same. We motivate the need to break this assumption, demonstrate that the existing solutions no longer apply, and propose new index partitioning strategies that yield low update costs and fast access times.

1 Introduction

A traditional database stores data items assumed to be valid indefinitely, or at least until modified by a user or application. However, many applications deal with data that are valid for a pre-determined, finite length of time [19]. Data stream processing is one example, where sliding windows are defined on the inputs in order to avoid memory overflow and emphasize recently arrived data. For instance, a weather monitoring application may track the average temperature and humidity reported by various stations over the last hour. This involves maintaining a one-hour sliding window, wherein each data point has a *lifetime* of one hour before it expires.

Research on data stream management typically assumes that sliding windows are stored in main memory for fast online processing; e.g., [1, 2, 4, 12, 13, 15]. In contrast, this paper studies the maintenance of time-evolving data that may be spooled to disk for off-line analysis. We consider applications that monitor data generated by one or more sources, perform light-weight processing on-the-fly, and periodically append new data to a disk-based archive. The archive is responsible for removing expired data and facilitating complex off-line queries that are too expensive to be done on-line. Examples include network traffic analysis, where the archive is mined by an Internet Service Provider (ISP) in order to discover recent usage patterns and plan changes in the network infrastructure [3]; transaction logging, where recent point-of-sale purchase records or telephone call logs are examined for customer behaviour analysis and fraud detection [6, 14]; and networks of sensors that measure physical phenomena such as temperature and humidity, where recent observations are used to discover trends and make predictions [7].

As in traditional database applications, query performance can be improved if appropriate indices are built. However, maintaining a disk-based index over time-evolving data is challenging because new data must be continually inserted and old data deleted. One way to reduce the I/O complexity of index maintenance is to perform periodic batch-updates. Additionally, it is desirable to avoid bringing the entire index into memory during every update. This can be done by partitioning the data so as to localize updates to a small number of disk pages. For example, if an index over a sliding window is partitioned chronologically [9, 20], then only the youngest partition incurs insertions, while only the oldest partition needs to be checked for expirations (the remaining partitions "in the middle" are not accessed).

Chronological partitioning is based upon the assumption that the order in which data are inserted is equivalent to the expiration order, which means that the lifetime of each data item is the same. This assumption holds if the application

*This research is partially supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Communications and Information Technology Ontario (CITO).

†Work done while the author was visiting the University of Waterloo.

	Equal lifetimes	Variable lifetimes
Memory	FIFO queue [13]	Calendar queue [13]
Disk	Wave index [20]	

Table 1. Classification of previous work on maintenance of time-evolving data.

only maintains sliding windows, but we may also choose to store (indexed) materialized results derived from one or more base windows, such as those of a sliding window join [12, 15]. If many queries over the archive compute the same join, then materializing the join result removes the need for each interested query to compute it from scratch[1]. Instead, the index may be used by each interested query to efficiently extract relevant data for further processing. However, a newly arrived item may join with several items from the other window(s), of which some have arrived recently and others are about to expire. Given that a join result expires when at least one of the base data items expires from its window, the results of a join operation may have different lifetimes [13, 16]. Consequently, the insertion order of the results is different from their expiration order.

In addition to introducing variable lifetimes by way of materialized results, sources may explicitly assign different lifetimes to the data that they generate. For example, one sensor may produce temperature measurements every ten minutes (giving each value a lifetime of ten minutes before being replaced with a new value), whereas another sensor may report humidity values every fifteen minutes. Similarly, various sources may be polled explicitly with different frequencies. For instance, the humidity sensor may require more energy to compute and/or transmit a new value than the temperature sensor, and should therefore be polled less often in order to save battery power [17].

As illustrated in Table 1, previous work on storing time-evolving data may be classified according to two criteria: main memory versus secondary storage, and equal versus variable lifetimes of the data items. To the best of our knowledge, this paper is the first to address the most challenging of the four scenarios: disk-based indexing of data items having variable lifetimes. In the remainder of this paper, Section 2 explains our assumptions and the limitations of previous work, Section 3 presents our index partitioning techniques, Section 4 experimentally shows the advantages of our solutions in terms of index update and access times, Section 5 compares the contributions of this paper with related work, and Section 6 concludes the paper with suggestions for future research.

[1]Deciding which sub-expressions to materialize is an orthogonal problem that we do not pursue here; see, e.g., [2, 5] for possible solutions in the context of data streams and sliding windows.

2 Preliminaries

The problem addressed in this paper concerns indexing a time-evolving set of data items with associated lifetimes, such that index lookups and periodic updates may be done efficiently. We assume that the lifetime, and therefore the expiration time, of each item is known, but the lifetimes of various items may have different lengths, up to some predetermined upper bound. New data are continually generated by one or more sources and buffered in main memory between index updates. During an update, new items which have arrived since the last update are inserted and items whose lifetimes have expired must be deleted. This involves bringing one or more pages into memory, updating them, and writing them back to disk. Two access types must be supported: probes (retrieval of items having a particular search key value or range), and scans of the entire index. Probes may be performed by queries that access a shared materialized result and extract a relevant subset of the data for further processing. Scans are performed by complex queries that must examine the entire data set in order to update their answers.

Index probes and scans may be done efficiently if the index is clustered on the search key. On the other hand, if the index stores data whose insertion order is equivalent to the expiration order, then chronological clustering leads to efficient updates—insertions are appended to the new end and deletions occur at the old end. However, the disadvantage of chronological clustering is that records with the same search key may be scattered across a very large number of disk pages, causing index probes to incur a prohibitively high number of disk I/Os.

In order to balance the access and update times, a sliding window index has been proposed in [20] that chronologically divides the window into n equal partitions, each of which is separately indexed and clustered by search key. An example is shown in Figure 1, where a window of size 16 minutes that is updated every 2 minutes is split into four sub-indices: I_1, I_2, I_3, and I_4. Triangles indicate directories—each associated with a single sub-index—which could be B+-trees, R-trees, or any other data structure as appropriate. Rectangles represent data records, which are stored on disk. On the left, the window is partitioned by insertion time. On the right, an equivalent partitioning is shown by expiration time; the window size of 16 is added to each item's insertion time to determine the expiration time. As illustrated, an update at time 18 inserts newly arrived tuples between times 17 and 18 (which will expire between times 33 and 34) into I_1, at the same time deleting tuples which have arrived between times one and 2 (or which have expired between times 17 and 18). The advantage of this approach is that only one sub-index is affected by any given update; for instance, only I_1 changes at times 18 and 20,

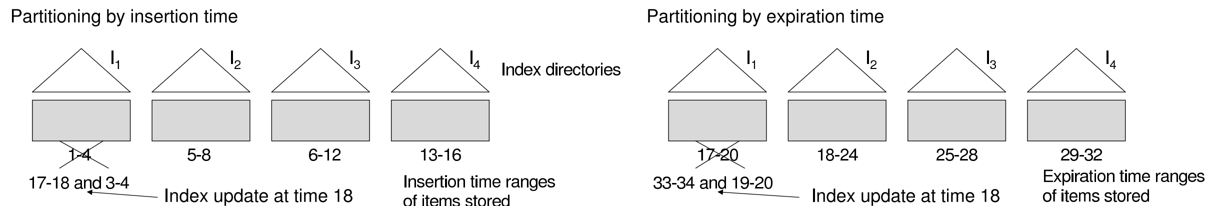

Figure 1. Two equivalent illustrations of a partitioned sliding window index.

only I_2 will change at times 22 and 24, and so on. The tradeoff is that access times are slower because multiple sub-indices are probed to obtain the answer.

A partitioned index similar to that in Figure 1 is inappropriate for disk-based storage of data with variable lifetimes. First, suppose that we partition the index by insertion time, as on the left of Figure 1. At time 18, only I_1 is accessed in order to insert new items, as before. However, all four sub-indices need to be scanned in order to determine which records have expired (it is no longer the case that only the items inserted between times one and 2 expire at time 18). This may require a large number of disk I/Os and cause unacceptably slow updates. Similarly, partitioning the index according to deletion times, as in the bottom of Figure 1, means that all the expired items at time 18 can be found in I_1, but there may be insertions into every sub-index (it is not the case that all records inserted between times 17 and 18 will expire between times 33 and 34). Again, all the sub-indices may need to be read into memory during index updates.

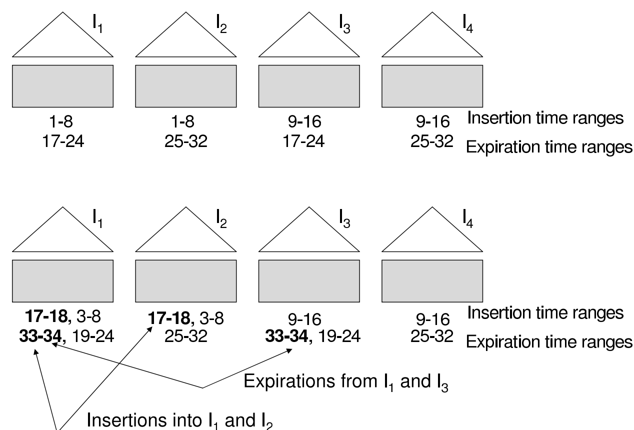

Figure 2. Example of a doubly partitioned index, showing an update at time 18 (bottom).

3 Proposed Solution

As seen in the previous section, a partitioned index balances two requirements: clustering by search key for efficient probing and by insertion (or expiration) time so that updates are confined to a single sub-index. Disk-based indexing of time-evolving data with variable lifetimes involves three conflicting requirements: clustering by search key for efficient probing, by insertion time for efficient insertions, and by expiration time for efficient deletions. In this section, we propose a solution, which we call *doubly partitioned index*, that reconciles these three constraints. The idea is to simultaneously partition the index on insertion and expiration times.

3.1 Double Partitioning

A simple example of a doubly partitioned index (we will present an improved variant shortly) is shown in Figure 2, given that the lifetimes of all the data records are at most 16 minutes and that updates are performed every 2 minutes. As before, each sub-index contains a directory on the search key and stores data records on disk, clustered by search key. However, the ranges of insertion and expiration times are now chronologically divided into two partitions each, creating a total of four sub-indices. As illustrated, at time 16, sub-index I_1 stores data items inserted between times one and 8 that will expire between times 17 and 24 (the other three sub-indices may be described similarly). The update illustrated on the bottom of Figure 2 takes place at time 18, inserts new items into I_1 and I_2, and deletes expired items from I_1 and I_3. Observe that I_4 does not have to be accessed during this update, or during the next three updates at times 20, 22, and 24. Then, the next four updates at times 26, 28, 30, and 32 will insert into I_3 and I_4, and delete from I_2 and I_4 (I_1 will not be accessed). In general, increasing the number of partitions leads to more sub-indices not being accessed during updates, thereby decreasing the index maintenance costs.

The flaw with chronological partitioning of the insertion and expiration times is that the sub-indices may have widely different sizes. Recall Figure 2 and note that at time 16, I_2 stores items that arrived between times one and 8 and will expire between times 25 and 32. That is, I_2 is empty at this time because there are no items whose lifetimes are larger than 16. As a result, the other sub-indices are large and their

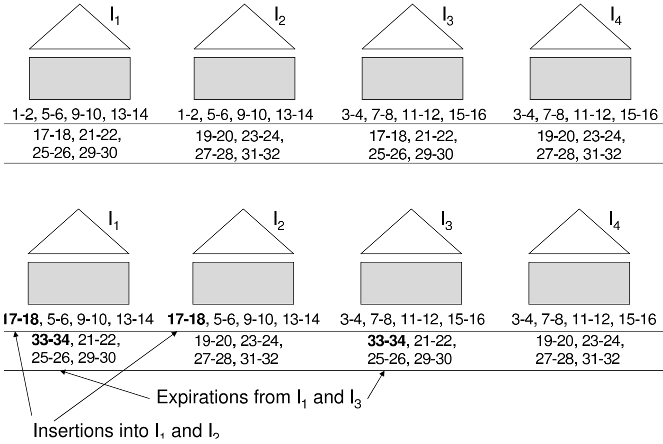

Figure 3. Example of a round-robin doubly partitioned index, showing an update at time 18 **(bottom).**

update costs may dominate the overall maintenance cost. We address this problem by adjusting the intervals spanned by each sub-index. The improved technique, which we call *round-robin partitioning*, is illustrated in Figure 3 for the same parameters as in Figure 2 (items have lifetimes of up to 16 minutes and index updates are done every two minutes). The two rows of intervals underneath each sub-index correspond to the insertion time and expiration time ranges, respectively. As can be seen, rather than dividing the insertion and expiration time ranges chronologically, round-robin partitioning distributes updates in a round-robin fashion such that no sub-index experiences two consecutive insertions or expirations. For instance, the update illustrated on the bottom of Figure 3 takes place at time 18, inserts new tuples into I_1 and I_2 and expires tuples from I_1 and I_3. The next update at time 20 inserts new tuples into I_3 and I_4, and deletes old tuples from I_2 and I_4. The fact that consecutive updates are spread out over different sub-indices ensures that the sub-indices have similar sizes. As we will experimentally show in Section 4, this property translates to more efficient index updates.

Doubly partitioned indices are compatible with two bulk-update strategies. In what we call *Rebuild*, updated sub-indices are completely rebuilt and reclustered, such that all the records with the same search key are stored contiguously in one dynamically-sized bucket (spanning one or more contiguous disk pages). In what we call *NoRebuild*, each sub-index allocates multiple fixed-size buckets for each search key (usually not contiguously). Therefore, updates may cause additional buckets to be created or existing buckets to be deleted, if empty.

Let n be the number of sub-indices, G be the number

Algorithm 1 Round-robin doubly partitioned index

Initial stage at time τ

1 **for** $i = 0$ to $\frac{S}{rG} - 1$
2 **for** $j = 0$ to $G - 1$
3 I_{jE+1} through $I_{(j+1)E}$ are assigned an insertion time range of $\tau - S + (iG + j)r + 1$ to $\tau - S + (iG + j + 1)r$
4 **for** $i = 0$ to $\frac{S}{rE} - 1$
5 **for** $j = 0$ to $E - 1$
6 $I_{j+1}, I_{E+j+1}, \ldots, I_{(G-1)E+j+1}$ are assigned an expiration time range of $\tau + S + (iG + j)r + 1$ to $\tau + S + (iG + j + 1)r$
7 Insert initial result tuples to appropriate sub-indices

Periodic update stage at time $\tau + jr$, $j = 1, 2, \ldots$

1 **for** each sub-index with expiration time range of $\tau + (j - 1)r + 1$ to $\tau + jr$
2 Replace above range with $\tau + (j - 1)r + 1 + S$ to $\tau + jr + S$
3 Delete tuples with expiration times of $\tau + (j - 1)r + 1$ to $\tau + jr$
4 **for** each sub-index with insertion time range of $\tau + (j - 1)r + 1 - S$ to $\tau + jr - S$
5 Replace above range with $\tau + (j - 1)r + 1$ to $\tau + jr$
6 Insert new result tuples to appropriate sub-indices

of partitions of generation (insertion) times, and E be the number of partitions of expiration times ($n = G \times E$)[2]. Furthermore, let S be the upper bound on the lifetimes of data items and r be the time interval between two consecutive index updates. Algorithm 1 implements the round-robin doubly partitioned index and contains two stages: the initial stage and the periodic update stage. We start with a set of data that are assumed to be valid at some time τ. In the initial stage, we make a partitioning of the insertion and expiration times, insert the data records into the appropriate sub-indices, and build the corresponding sub-index directories. In the update stage, we periodically access some of the sub-indices in order to adjust the insertion and expiration times that they span, insert and/or delete tuples in the appropriate sub-indices according to the insertion and expiration times, and update the corresponding sub-index directories. A detailed implementation of insertions and deletions is not shown in the algorithm as this depends on the clustering technique (*Rebuild* versus *NoRebuild*). Similarly, specific details concerning directory updates are omitted since the algorithm is compatible with a wide range of directory data structures.

[2] For example, in Figures 2 and 3, $G = 2$, $E = 2$, and $n = G \times E = 4$.

3.2 Cost Analysis

The number of sub-indices accessed during each update is $G + E - 1$. To choose optimal values for G and E with respect to the number of sub-index accesses, we minimize $G + E$ given that $G \times E = n$ and $G, E \geq 2$, which yields G=E=$\sqrt{n}$. For simplicity, we assume that n is a perfect square.

Increasing n increases the space requirements (each sub-index requires its own directory on the search key) and leads to slower query times because index scans and probes need to access all n sub-indices. Additionally, more individual sub-indices are accessed during updates as n increases. However, the sub-indices are faster to update because they are smaller, and the fraction of the data that need to be updated decreases. For instance, setting $G = E = 2$ (as in Figures 2 and 3) means that three of the four sub-indices are scanned during updates, but increasing G and E to four means that only seven of sixteen sub-indices are accessed. As will be shown in Section 4, increasing n initially decreases update times, but eventually a breakpoint is reached where the individual sub-indices are small and making any further splits is not helpful (note that the breakpoint value of n is expected to be higher for larger data sets).

The other part of the maintenance and query costs is contributed by the operations done after a sub-index is accessed. Fixing G and E, *Rebuild* should be faster to query (only one bucket is accessed to find all records with a given search key), but slower to update (especially as the data size grows, because rebuilding large indices may be expensive). Access into *NoRebuild* is slower because records with the same search key may be scattered across many buckets, but *NoRebuild* should be faster to update because individual updates are less costly than rebuilding an entire sub-index. Furthermore the total size of *NoRebuild* may be larger than *Rebuild* because some pre-allocated buckets may not be full.

3.3 Handling Fluctuating Stream Conditions

Round-robin partitioning creates sub-indices with similar sizes if the amount of new data arriving between updates does not change. However, in the worst case, the data rate may alternate between slow and bursty periods, causing the round-robin allocation policy to create some sub-indices that are very large and some that are very small. The algorithmic solution in this case is to randomize the update allocation policy. In practice, though, we expect random fluctuations in the data rate. Furthermore, the change in the data rate may be persistent for several index updates, or short-lived between two consecutive updates. In both cases, we expect round-robin partitioning to adapt to the new conditions. Given a persistent change, round-robin update allocation ensures that updates are spread out across the sub-indices. Thus, if the number of new data items increases (or decreases), then each sub-index will in turn get larger (or smaller), until all the sub-indices have similar sizes again. Using chronological partitioning, the same sub-index would receive a number of consecutive updates and become either much larger or much smaller than the others. If a change is short-lived, it is also better to begin with equal sub-index sizes. Otherwise, a burst of new data could be inserted into a large sub-index, which would become even larger.

4 Experiments

This section contains an overview of our implementation (Section 4.1) and experimental results. Sections 4.2 through 4.4 present results of experiments with a small data set of approximately 500 Megabytes (this corresponds to data generated over a time of 500000 time units, with an average of one record generated per time unit). In Section 4.5, we investigate index performance over larger data sets with sizes of up to 5 Gigabytes (i.e., data produced over a time of 5 million time units, with an average of one record generated per time unit). Our experimental findings are summarized in Section 4.6.

4.1 Implementation Details

We implemented the doubly partitioned indices (*Rebuild* and *NoRebuild*) using Sun Microsystems JDK 1.4.1, and tested them on a Linux PC with a Pentium IV 2.4Ghz processor and 2 Gigabytes of RAM. For comparison, we also implemented two chronologically-partitioned indexing strategies from [20] (recall Figure 1): *REINDEX*, which is similar to *Rebuild* in that it reclusters sub-indices after updates, and *DEL*, which is similar to *NoRebuild* as it maintains multiple fixed-size buckets per key. Both *REINDEX* and *DEL* may be partitioned by insertion time (abbreviated *R-ins* or *D-ins*, respectively) or by expiration time (abbreviated *R-exp* or *D-exp*). We will refer to the indexing techniques by their abbreviations, followed by the value of n (number of sub-indices) or values of G and E (number of partitions of insertion and expiration times, respectively), e.g., *R-ins*4 or *Rebuild*2x2.

Each test consists of an initial building stage and an update stage. In the building stage, we populate the index using records with randomly generated lifetimes and search key values (the latter are generated from a uniform or Power Law distribution). The total data size in the initial stage varies from 500 Megabyte to five Gigabytes. Next, we generate periodic updates using the same lifetime and search key distribution, and insert them into the index, at the same

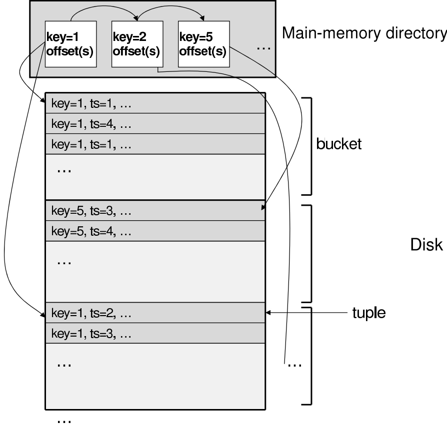

Figure 4. Structure of an individual sub-index.

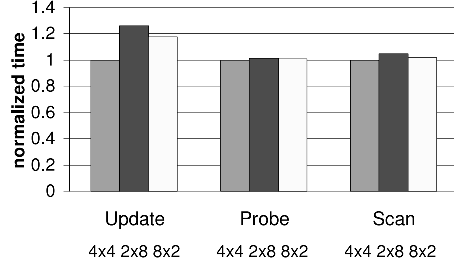

Figure 5. Relative performance of *Rebuild*4x4, *Rebuild*2x8, and *Rebuild*8x2.

time removing expired tuples. After each update, we perform an index probe (retrieving tuples having a randomly chosen search key value) and an index scan, and report the average processing time for each operation. We perform 36 updates until the amount of new data generated equals the initial data size. This corresponds to an index update frequency of roughly 14000 to 140000 time units, with a relative data rate of one record per time unit.

Each indexing technique consists of an array of sub-indices, with each sub-index containing a main-memory directory (implemented as a linked list sorted by search key) and a random access file storing the results. The file is a collection of buckets storing tuples with the same search key, sorted by expiration time. Individual data records are 1000 bytes long and contain an integer search key, an integer expiration timestamp, as well as a string that abstractly represents the contents of the data. The structure of an individual sub-index in *NoRebuild* and *DEL* is illustrated in Figure 4, showing the directory with offset pointers to locations of buckets in the file (note that there may be more than one bucket per search key in case of overflow). We also store how many records are in each bucket as not all buckets are full. *Rebuild* and *REINDEX* are structured similarly, except that one variable-size bucket is maintained for each search key.

A number of simplifications have been made to focus the experiments on the relative performance of doubly partitioned indices. First, bucket sizes are not adjusted upon overflow; this issue was studied in [8] in the context of skewed distributions and is orthogonal to this work. Instead, we implemented a simple strategy that allocates another bucket of the same size for the given key. We also ignore the fact that empty buckets should be garbage-collected periodically by compacting the file, because this operation adds a constant amount of time to the maintenance costs of each indexing technique. Second, the number of search key values is fixed at 100 in order to bound the length of the directory. Otherwise, query times may be dominated by the time it takes to scan a long list; handling a larger set of key values can be done with a more efficient directory, such as a B+-tree, and is orthogonal to this work. Third, the number of tuples per bucket in *NoRebuild* and *DEL* is based upon the initial distribution of key values in the building stage, such that each sub-index contains an average of 2.5 buckets per search key. We found this value to be a good compromise between few large buckets per key (which wastes space because many newly allocated buckets never fill up) and too many buckets (which results in slower query times).

4.2 Optimal Values for G and E

We begin by validating our results from Section 3.2 regarding the optimal assignment of values for G and E given a value for n. Figure 5 shows the normalized update, probe, and scan times for *Rebuild*4x4, *Rebuild*2x8, and *Rebuild*8x2, given a uniform distribution of search key values; other index types and a Power Law distribution of key values give similar results. *Rebuild*4x4 performs best in terms of scan and probe times, though the difference is negligible because all three techniques probe the same number of sub-indices to obtain query results and all the sub-indices have roughly equal sizes. The average update time of *Rebuild*4x4 is approximately 20 percent lower than the other techniques because the number of sub-indices updated by *Rebuild*4x4 is 7, versus 9 for the other two strategies. Notably, *Rebuild*8x2 can be updated faster than *Rebuild*2x8 because tuples inside buckets are ordered by expiration time, and therefore deletions are simple (tuples are removed from the front of the bucket) but insertions are more complex (whole bucket must be scanned). Since the number of insertions is determined by the number of partitions in the lower level, the technique with a smaller value of E wins.

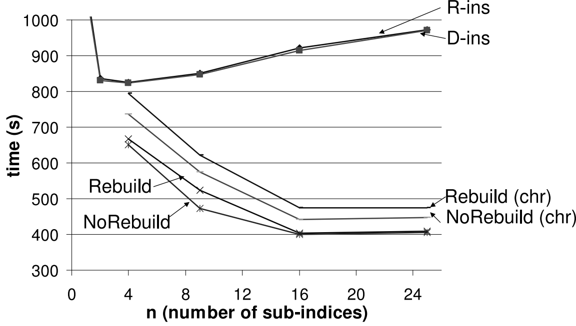

Figure 6. Update times of index partitioning techniques given a small window size.

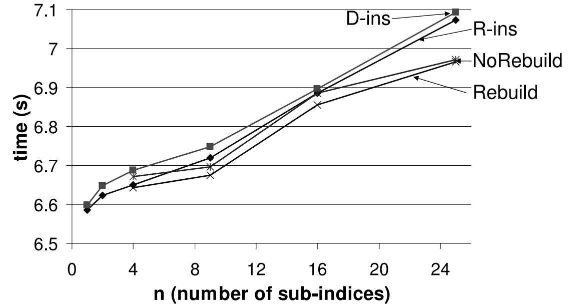

Figure 7. Probe times of index partitioning techniques given a small window size.

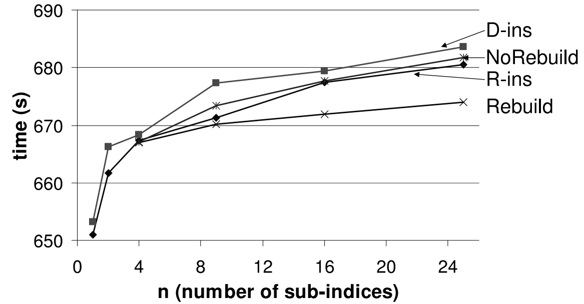

Figure 8. Scan times of index partitioning techniques given a small window size.

4.3 Performance of Doubly Partitioned Indices

We now compare our doubly partitioned indices with the existing algorithms. As per the previous experiment, we test our techniques only with the following splits: 2x2, 3x3, 4x4, and 5x5. We omit results for *R-exp* and *D-exp* because these always incur longer update times than *R-ins* and *D-ins*. As before, this is because insertions are more expensive than deletions if buckets are sorted by expiration time, therefore splitting an index by expiration time forces insertions into every sub-index. Moreover, we only report results with search keys generated from a uniform distribution for this and all remaining experiments. Results using a Power Law distribution (with the power law coefficient equal to unity) produce similar relative results, except that *NoRebuild* and *DEL* are slower to update due to their simple bucket re-allocation mechanism.

Figures 6, 7, and 8 show the average update, probe, and scan times, respectively, as functions of n (number of sub-indices). Figure 6 additionally shows the update times of doubly partitioned indices with chronological partitioning (denoted by chr) in order to single out the benefits of round-robin partitioning. Even chronological partitioning outperforms the existing strategies by a factor of two as n grows, with round-robin partitioning additionally improving the update times by ten to 20 percent. As explained in Section 3.2, *NoRebuild* is faster to update than *Rebuild*, but is slower to probe and scan.

The update overhead of *Rebuild* relative to *NoRebuild* is roughly five percent for $n < 9$ and decreases to under two percent for large n. The relative savings in index probe times of *Rebuild* are less than one percent. This is because we use a relatively small data size in this experiment (roughly 500 Megabytes), meaning that the individual sub-indices are small and can be rebuilt quickly. Additionally, all the buckets with a particular search key may be found with a small number of disk accesses, even if the buckets are scattered across the file. Hence, probing *NoRebuild* is only slightly more expensive than probing *Rebuild*, where records with the same search keys are found in the same bucket. In general, *Rebuild* and *NoRebuild* perform probes slightly faster than *R-ins* and *D-ins* because the sub-indices in our techniques have similar sizes, and therefore we do not encounter "bad cases" where probing one or more large sub-indices inflates the access cost.

As n increases, the access times grow because more sub-indices must be probed separately, whereas update times decrease initially, but begin growing for $n \geq 25$ (or $n \geq 9$ for *R-ins* and *D-ins*). As mentioned in Section 3.2, this is due to two factors influencing the update costs: as n increases, the amount of data to be updated decreases, but the number of individual sub-index accesses increases. The latter is the reason why the update costs of *R-ins* and *D-ins* start increasing for smaller values of n than those for our doubly partitioned techniques: the existing techniques access all n sub-indices during updates, whereas our techniques only access $G + E - 1 = 2\sqrt{n} - 1$ sub-indices.

Given a fixed value of n, *Rebuild* and *R-ins* both have the lowest space requirements, followed by *NoRebuild* and *D-ins*. *NoRebuild* and *D-ins* incur the overhead of pre-allocating buckets which may never fill up (the exact space

penalty depends on the bucket allocation strategy, which is orthogonal to this work). As n increases, all techniques require more space in order to store sub-index directories.

To measure the overhead associated with our indexing techniques, we also tested chronological and round-robin partitioning without indices. Update times were down by approximately 20 percent because index directories did not have to be updated and files were not reclustered. However, probes and scans both required a sequential scan of the data, and took approximately the same amount of time as index scans in Figure 8, namely on the order of 600 seconds. Thus, our indexing techniques incur modest update overhead, but allow probe times that are two orders of magnitude faster than sequential scan.

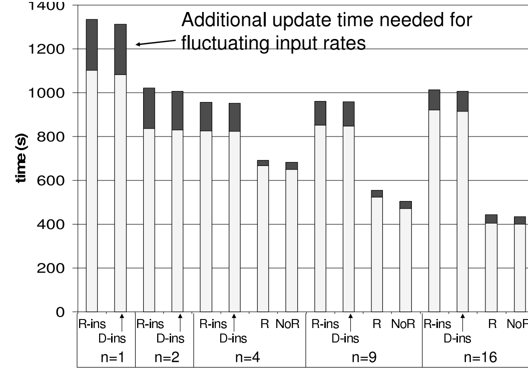

Figure 9. Effect of data rate fluctuations on index update performance.

4.4 Fluctuating Stream Conditions

In this experiment, the data rate varies randomly by a factor of up to four. We set the total amount of data items generated to be approximately the same as in the previous experiment in order to enable a head-to-head comparison. The average access times did not exhibit any interesting changes aside from being slower by several percent, therefore we focus on the update times, as illustrated in Figure 9 for selected techniques (R and NoR denote *Rebuild* and *NoRebuild*, respectively). The darkened portion of each bar corresponds to the increase in update time caused by the fluctuating data rate.

Doubly partitioned indices are more adaptable to fluctuating data rates than *R-ins* and *D-ins*. *NoRebuild* and *Rebuild* are more significantly affected by fluctuations for larger values of n, whereas *R-ins* and *D-ins* exhibit the worst performance for small values of n. This can be explained as follows. *Rebuild* and *NoRebuild* use round-robin partitioning, meaning that updates are scattered across sub-indices, therefore a large value of n means that it takes longer for the new data rate to take effect in all the sub-indices. On the other hand, *R-ins* and *D-ins* use chronological partitioning, therefore a large value of n means that the sub-indices have shorter time spans and therefore bursty updates spread out faster across the sub-indices. Finally, *Rebuild* and *R-ins* are more resilient to fluctuations than *NoRebuild* and *D-ins* because the latter two use a simple (non-adaptive) bucket allocation technique.

4.5 Scaling up to Large Index Sizes

This test investigates the behaviour of our techniques when indexing large amounts of data (five Gigabytes). The average update, probe, and scan times as functions of n are shown in Figures 10, 11, and 12, respectively. Doubly partitioned indices are now up to three times as fast to update as the existing techniques. Additionally, the gap between the update and query times of *Rebuild* versus *NoRebuild* is now wider. *Rebuild* is two to three percent faster to probe, but between five (for $n = 25$) and nine (for $n = 4$) percent slower to update; the corresponding percentages from Figure 7 are less than one percent and roughly three percent, respectively. This is the expected outcome of indexing a large data set: *Rebuild* becomes slower to update because it must recluster larger sub-indices, whereas *NoRebuild* becomes slower to probe because there are more result tuples with the same search key, spread over multiple buckets and possibly multiple disk pages.

Another difference between Figures 10 and 6 is the behaviour of update times as n grows. In Figure 6, there is a turning point (at $n = 16$ for *NoRebuild* and *Rebuild*) after which update times do not decrease. This is not the case in Figure 10, where update times continue to drop for all tested values of n. This is because the window size, and hence individual sub-index sizes, are larger, therefore the drop in performance caused by making the sub-indices too small is not an issue for $n \leq 25$.

4.6 Lessons Learned

We make the following recommendations regarding the best index partitioning strategy. The guidelines depend on the data size and the expected number of queries to be executed over the archive between updates.

- For a small window size and small number of queries, *NoRebuild*4x4 is a good choice as it incurs low update times.
- For a small window size and large number of queries, *Rebuild*2x2 works best because its probe and scan times are low. The probing times of *R-ins*1, and *R-ins*2 are slightly lower than those of *Rebuild*2x2, but updating *R-ins* is slower.

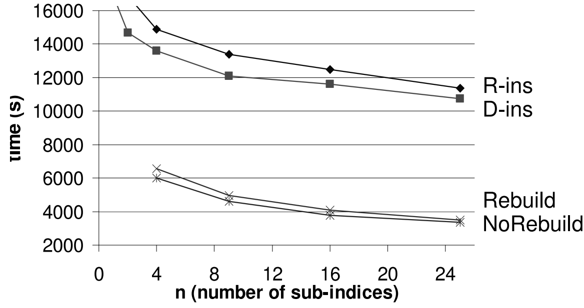

Figure 10. Update times of index partitioning techniques given a large data set.

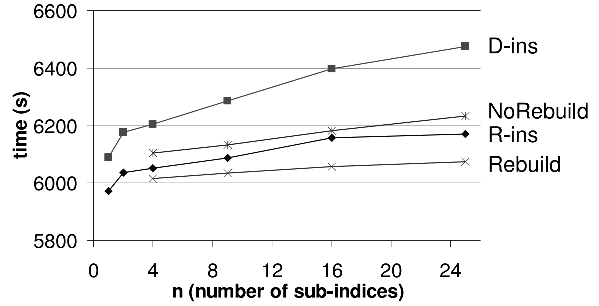

Figure 12. Scan times of index partitioning techniques given a large data set.

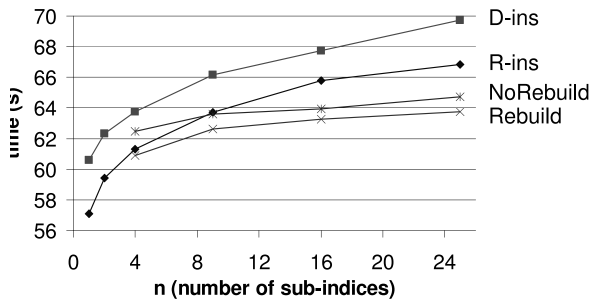

Figure 11. Probe times of index partitioning techniques given a large data set.

- For a large window size and small number of queries, we suggest using *NoRebuild*, but with a smaller value of n than recommended for small window sizes to ensure that probing times are not excessively high (e.g., *NoRebuild*3x3).
- For a large window size and large number of queries, *Rebuild* becomes expensive to update, therefore we recommend *Rebuild*2x2 or *Rebuild*3x3 only if fast probing times are crucial. Otherwise, *NoRebuild*2x2 is a better (more balanced) choice.

5 Comparison with Related Work

This work is closely related to indexing time-evolving and spatio-temporal data, data stream processing, and analyzing the expiration order of queries over sliding windows. We summarize the differences between our contributions and the related work below.

Previous work on indexing sliding windows in secondary storage partitions the data by arrival time [20], which, as we have shown, is not suitable for data that do not expire in order of arrival, i.e., data with variable lifetimes. There has also been previous work on storing large sliding windows on disk and avoiding deletions altogether by materializing multiple append-only prefixes of the window [10]. Again, the underlying assumption in [10] is that the lifetimes of all the data items are equal to the window length, thereby making it inappropriate in the context of date items with variable lifetimes.

The expiration patterns of the results of sliding window queries were first analyzed in our previous work on update-pattern-aware query processing [13]. However, our previous work assumed that all the data fit in main memory and that intermediate results are partitioned by expiration time. This is sufficient in the main-memory scenario because the goal is to make expirations more efficient (by not having to scan the entire result); insertions may be scattered across the entire result because of the luxury of random access in main memory. In this paper, the fact that the data are stored on disk means that both insertions and expirations must be localized to a small number of sub-indices in order to prevent the entire index from being brought into main memory during each update. Consequently, a doubly partitioned index is more appropriate.

In Section 4.6, our recommendations regarding the best index partitioning strategy depend upon the number of queries executed between index updates. Given that the query workload may fluctuate over time, it may be advantageous to switch to a different indexing technique at some point. This problem is similar to plan migration in the context of sliding window queries that store state [22]. Two possible solutions are either stopping the old plan, migrating the state, and starting the new plan, or running both plans in parallel and discarding the old plan when all the windows roll over. Both strategies are compatible with our indices in that we can migrate from one index type to another either by discarding the old index and building a new index, or maintaining both indices in parallel until the old

index gradually empties out.

In general, maintaining time-evolving data with finite lifetimes is related to spatio-temporal interval indexing (see, e.g., [18] for a survey). The important difference in our work is that we do not explicitly index the actual (insertion or expiration time) intervals, but rather we index individual records by their search keys. It is possible to maintain an interval index over the time intervals spanned by individual sub-indices (see, e.g., [21]), in which case looking up sub-indices which are affected by a given update could be faster. However, we did not need to employ this optimization in this work because the number of sub-indices we were dealing with was relatively small.

6 Conclusions

In this paper, we identified and solved an open problem in the context of maintenance of time-evolving data: indexing data with variable lifetimes in secondary storage. The insight behind our solution was to simultaneously partition the data by insertion and expiration times in order to ensure fast bulk-updates. Experimental results showed significant improvements in index update times as compared to previous work on sliding window indices, especially for large data sets. Our solutions are applicable in a wide range of applications that perform complex off-line mining of streaming data.

We are interested in two directions for future research. First, we want to study the issues and tradeoffs involved in on-line migration from one type of doubly partitioned index to another, e.g., adjusting the number of sub-indices or changing the clustering method from *Rebuild* to *NoRebuild*. Second, we intend to address issues of update consistency due to making changes in-place or replacing an entire sub-index with a copy on which updates have been made. In particular, we plan to extend our recent work on concurrency control in sliding window queries [11] to cover the case of indexing data with variable lifetimes.

References

[1] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, to appear.

[2] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Proc. ICDE Conf.*, pages 118–129, 2005.

[3] Y. D. Cai, D. Clutter, G. Pape, J. Han, M. Welge, and L. Auvil. MAIDS: Mining alarming incidents from data streams. In *Proc. ACM SIGMOD Conf.*, pages 919–920, 2004.

[4] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, Aug 2003.

[5] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Conf.*, pages 379–390, 2000.

[6] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proc. ACM SIGKDD Conf.*, pages 9–17, 2000.

[7] C. Faloutsos. Sensor data mining: Similarity search and pattern analysis. In *Proc. VLDB Conf.*, 2002.

[8] C. Faloutsos and H. Jagadish. On B-tree indices for skewed distributions. In *Proc. VLDB Conf.*, pages 363–374, 1992.

[9] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng. Optimizing refresh of a set of materialized views. In *Proc. VLDB Conf.*, pages 1043–1054, 2005.

[10] V. Ganti, J. Gehrke, and R. Ramakrishnan. DEMON: Mining and monitoring evolving data. *IEEE Trans. Knowledge and Data Eng.*, 13(1):50–63, 2001.

[11] L. Golab, K. G. Bijay, and M. T. Özsu. On concurrency control in sliding window queries over data streams. In *Proc. EDBT Conf.*, pages 608–626, 2006.

[12] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. VLDB Conf.*, pages 500–511, 2003.

[13] L. Golab and M. T. Özsu. Update-pattern aware modeling and processing of continuous queries. In *Proc. ACM SIGMOD Conf.*, pages 658–669, 2005.

[14] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analyzing massive RFID data sets. In *Proc. ICDE Conf.*, 2006, to appear.

[15] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. ICDE Conf.*, pages 341–352, 2003.

[16] J. Krämer and B. Seeger. A temporal foundation for continuous queries over data streams. In *Proc. COMAD Conf.*, pages 70–82, 2005.

[17] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. ACM SIGMOD Conf.*, pages 491–502, 2003.

[18] B. Salzberg and V. Tsotras. A comparison of access methods for time evolving data. *ACM Computing Surveys*, 31(2):158–221, June 1999.

[19] A. Schmidt, C. Jensen, and S. Saltenis. Expiration times for data management. In *Proc. ICDE Conf.*, 2006, to appear.

[20] N. Shivakumar and H. García-Molina. Wave-indices: indexing evolving databases. In *Proc. ACM SIGMOD Conf.*, pages 381–392, 1997.

[21] K.-L. Wu, S.-K. Chen, and P. Yu. Interval query indexing for efficient stream processing. In *Proc. CIKM Conf.*, pages 88–97, 2004.

[22] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proc. ACM SIGMOD Conf.*, pages 431–442, 2004.