Indexing the Results of Sliding Window Queries*

Lukasz Golab† Piyush Prahladka‡ M. Tamer Özsu†

Technical Report CS-2005-10
May 2005

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

†School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {lgolab,tozsu}@uwaterloo.ca

‡Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India, piyush@cse.iitb.ac.in. Work done while the author was visiting the University of Waterloo.

Abstract

A popular method of bounding the memory requirements of queries over data streams is to use a sliding window, where old data are continuously removed as new data arrive. One problem that has not been addressed previously concerns indexing the results of sliding window queries. This is a noteworthy problem because data stream systems often materialize common sub-expressions or final results of similar queries, therefore it is important to allow the queries efficient access into a relevant subset of the materialized result. In this paper, we design and evaluate indexing methods that take advantage of the temporal order in which old answers expire for efficient maintenance and querying of the results. Our techniques allow sharing of indexed results among similar queries, adapt to fluctuating stream arrival rates, and are experimentally shown to perform updates over twice as fast as the existing sliding window indices.

1 Introduction

Sliding windows are used to limit the memory requirements of *continuous queries* over infinite data streams. In the sliding window model, queries access the N most recent tuples (*count-based* windows) or tuples which have arrived in the last S time units (*time-based* windows). So restricted, many otherwise intractable queries may be answered in finite memory. For example, a join of two infinite streams may require the inputs to be stored in their entirety, but the state associated with a sliding window join is confined to finite windows.

This paper studies indexing the results of sliding window queries. We consider applications that intercept massive amounts of streaming data, perform some light-weight processing on-the-fly, and periodically append newly arrived data to a disk-based archive. The archive maintains a sliding window of recent results and facilitates more complex off-line processing. Examples include network traffic analysis, where the archive is mined by an Internet Service Provider (ISP) in order to discover usage patterns and plan changes to the network infrastructure; transaction logging, where point-of-sale purchase data are streamed into a data warehouse to examine customer behaviour and identify credit card fraud; and networks of sensors that measure physical phenomena such as air temperature, where the recent data could be used to discover trends and make predictions.

To motivate the need for materializing query results in the above applications, note that the archive is expected to be accessed asynchronously by a large number of continuous queries. Therefore, materializing shared sub-expressions of similar queries is a particularly suitable optimization technique; see, e.g., NiagaraCQ [5], PSoup [3], and STREAM [15]. For example, a join result of two streams of network traffic (arriving from different links or sources) may be maintained and periodically probed by queries involving the join, but having different selection predicates and aggregation functions. This reduces query response times because the join is not re-computed from scratch; it suffices to scan the materialized join result, extract tuples matching the selection predicate, and compute the aggregation function over the matching tuples.

This approach is beneficial only if queries can efficiently access relevant subsets of the materialized result. Otherwise, sequentially scanning a large shared result may be less efficient than separate processing of each query. One way to ensure that continuous queries can quickly access the data of interest is to build appropriate indices over the shared results. However, the index maintenance cost must not be so high as to defeat its purpose. Reducing the index maintenance overhead is particularly challenging in the context of sliding window queries, because new data continually arrive and old data expire as the window slides forward. Even if updates are performed periodically by buffering the streams, it may still be too expensive to index the evolving data, especially if the entire index must be scanned during every update.

A solution for indexing disk-resident sliding windows with fast update and access times was presented in [17]. However, this technique is not compatible with the results of sliding window queries, as we show in this paper (Section 2). Thus, we propose a more appropriate solution for indexing the results of queries over time-based windows (Section 3) as well as count-based windows (Section 4). In Section 5, our techniques are experimentally evaluated under various conditions and are shown to perform updates over twice as fast as those in [17], with no penalties in access times. We compare our contributions with related work in Section 6 and conclude in Section 7.

2 Preliminaries

We begin by outlining our system model and discussing the requirements of indices over the results of sliding window queries. We also explain the ideas behind the sliding window indices from [17] and show that they were not designed to handle the results of sliding window queries.

2.1 System Model and Requirements

A data stream is a sequence of relational tuples with a fixed schema. Each tuple obtains a non-decreasing timestamp, call it *ts*, upon arrival at the system; see [1, 18] for suggestions on dealing with timestamp assignment at the data source and out-of-order arrival. Incoming tuples are buffered in main memory, possibly processed in real-time by a set of simple queries, and periodically appended to a disk-based archive for processing by complex queries. The archive is responsible for maintaining a sliding window over the stream as well as any query results that have been chosen for materialization (deciding which sub-results to materialize and adapting to the changing query workload are two orthogonal problems that we do not pursue here; see, e.g., [2, 5]). A particular execution of a query over the archive is assumed to be correct with respect to the state of the windows as of the most recent update time of the archive.

We assume that queries evaluated in real-time are those which can be computed incrementally by scanning only the newly arrived tuples. Examples include selection with a simple boolean predicate and distributive aggregation [7, 10] such as *sum*. More complex queries that need to examine the entire window during re-execution are restricted to operate over the archive (e.g., non-distributive aggregates such as quantiles or top-k of the most frequent attribute values).

The system executes two types of queries: continuous and ad-hoc. Continuous queries are re-evaluated periodically until turned off by the user, whereas ad-hoc queries may be posed at any time and are executed only once over the current state of the stream (or archive). The ad-hoc query workload is unpredictable, therefore the sub-results stored in the archive may or may not be helpful. In the remainder of this paper, we will focus on complex continuous queries over the archive.

An index over the results of a (sub)query must support efficient bulk updates and two access types: probes (retrieval of tuples with a particular search key value or range), and scans of the

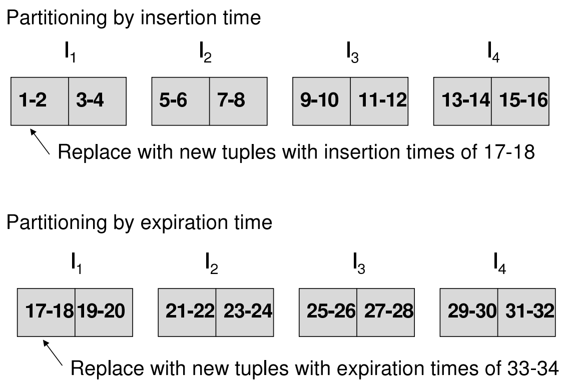
Partitioning by insertion time
I1
I2
I3
I4
1-2
3-4
5-6
7-8
9-10
11-12
13-14
15-16
Replace with new tuples with insertion times of 17-18
Partitioning by expiration time
I1
I2
I3
I4
17-18
19-20
21-22
23-24
25-26
27-28
29-30
31-32
Replace with new tuples with expiration times of 33-34

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

entire index. Index probes correspond to queries that access the shared sub-result and extract tuples matching a final selection predicate. Index scans are performed by queries that read the entire sub-result in particular order, such as joins, duplicate elimination, or order-dependent aggregation such as quantiles. The index directory (e.g., hash table or search tree) is assumed to fit in main memory and contains pointers to the data, stored as buckets of tuples with the same search key. The data are assumed to reside on disk.

2.2 Previous Work on Indexing Individual Sliding Windows

There has been previous work on indexing sliding windows on disk, with the goal of allowing fast bulk-updates and probes [17]. In addition to performing periodic updates, a possible improvement relies on exploiting the temporal order of insertions and deletions from a sliding window. Given a (time-based) window size of S, each tuple expires exactly S time units after it has been inserted into the window, meaning that tuples expire in the same order as that in which they were inserted [9]. This suggests clustering a sliding window index according to the tuple arrival times, in which case insertions are appended to the new end of the window and deletions occur from the old end (and the rest of the window does not change). However, a key search would require a scan of the entire index.

In order to balance the access and update times, [17] 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. On top, the window is partitioned by insertion time. On the bottom, an equivalent partitioning is shown by expiration time; the window size of 16 is added to each tuple'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, 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.

The existing partitioned indices for sliding windows include *DEL* (delete), *REINDEX*, and

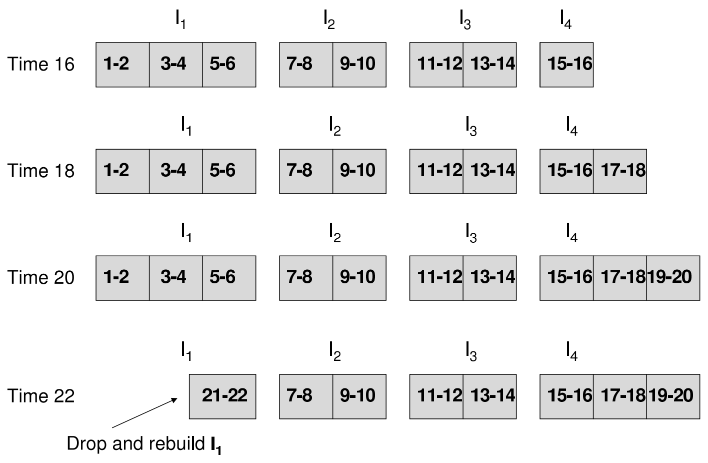

Figure 2: Updating a *WATA* index.

WATA (wait-and-throw-away) [17]. The first two were illustrated in Figure 1. *REINDEX* always stores tuples with the same search key contiguously in one dynamically-sized bucket. Thus, it must recluster updated sub-indices. *DEL* allocates multiple fixed-size buckets for each key (usually not contiguously), therefore updates may cause additional buckets to be created or existing buckets to be deleted, if empty. *REINDEX* is faster to probe because only one bucket is accessed to find all tuples with a given search key, but *DEL* may be updated more quickly because sub-indices are not reclustered. Additionally, *DEL* incurs some space overhead for pre-allocating buckets that may never fill up.

WATA is shown in Figure 2 for the same parameters as in Figure 1 (a window of 16 minutes refreshed every 2 minutes and split into four sub-indices). At times 18 and 20, insertions are appended to I_4, but deletions from I_1 are postponed until time 22, when I_1 is dropped and rebuilt with new data. Thus, rather than deleting from I_1 gradually, we wait and throw it away when every tuple inside it has expired. The advantage of *WATA* is that updates are fast. However, it has the longest probe times because expired tuples must be identified and skipped, and it requires additional space for storing expired tuples.

An optimal partitioning of a $WATA$ index is as follows [17]. Let n be the number of sub-indices, r be the time interval between two consecutive updates, and p be the number of times that a window is updated until it rolls over completely. The first $(p-1) \bmod (n-1)$ sub-indices have size $\lceil \frac{p-1}{n-1} r \rceil$, all but one of the remaining sub-indices have size $\lfloor \frac{p-1}{n-1} r \rfloor$, and the remaining sub-index has size r.

2.3 Limitations of Previous Work

The above technique relies on the fact that tuples expire from the window at predictable times and in order of arrival. However, the majority of sliding window queries may expire their results in different order [9]. Let $Q(\tau)$ be the answer of a continuous query Q at time τ. At any time, $Q(\tau)$ must be equal to the output of a corresponding one-time relational query whose input is the current state of the streams and sliding windows referenced in Q[1] [9, 11, 13]. Q is *monotonic* if $Q(\tau) \subseteq Q(\tau + \epsilon)$ for all τ and all $\epsilon \geq 0$. In other words, Q is monotonic if its result set only incurs insertions over time, e.g., a join of two infinite streams. All sliding window queries are *non-monotonic* because their results expire over time as the windows slide forward. In [9], we developed the following categorization of sliding window queries based upon the order in which their results expire[2].

Weakest non-monotonic queries do not reorder incoming tuples during processing and therefore expire results in generation order. Selection over a single window is one example; it drops some tuples, but the relative order of the surviving tuples does not change. We do not study these queries any further because their results may be indexed using the techniques from [17].

Weak non-monotonic queries do not expire results in generation order, but have the property that the expiration time of each result tuple can be determined at generation time. Examples include sliding window joins, where a result tuple expires if any one of the joined tuples expires from its window. That is, the expiration time of a join result can be calculated as the minimum of the expiration times of the individual tuples participating in the result. This means that the "lifetime" of join results may vary from one time unit up to the length of the smallest window referenced in the join. For instance, if a newly arrived tuple joins with a tuple from another window that is about to expire, then the result will have a short lifetime. Consequently, the expiration order of the results is different from their generation order, though the expiration times may still be calculated via timestamps. Indexing the results of weak non-monotonic queries will be discussed in Section 3.

Strict non-monotonic queries have the property that at least some of their results expire at unpredictable times. One example is negation over two sliding windows, where some results expire due to the movement of the sliding windows, but others expire according to the operator semantics. Suppose that a result tuple with attribute value v is in the answer set of a negation over two windows, $W_1 - W_2$. That is, there are tuples with value v in W_1 but not in W_2. If at any time a W_2-tuple arrives with value v, result tuples with value v must be deleted. These premature deletions are usually implemented as *negative tuples* that are generated by the negation operator and propagated through the query plan [11]. Note that those result tuples which are not deleted by arrivals of matching W_2-tuples expire in predicable order, as in weak non-monotonic queries. We will deal with this case in Section 3.5.

A partitioned index similar to that in Figure 1 is inappropriate for sliding window query results where the insertion order differs from the expiration order (i.e., for all but the weakest non-monotonic queries). First, suppose that we partition the data by generation time, as in the top of Figure 1. At time 18, only I_1 is accessed in order to insert newly generated results, as before. However, all three sub-indices need to be scanned to determine which results have expired (it is no longer the case that only the results generated between times one and 2 expire at time 18). Similarly, partitioning the index according to the result deletion times, as in the bottom of Figure 1, means that all the expired tuples 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 the new results generated between times 17 and 18 will expire between times 33 and 34).

[1]In our context of periodic window updates, we require the answer to reflect the state of the sliding windows as of the most recent update, as discussed in Section 2.1.

[2]For now, all windows are assumed to be time-based. We will address count-based windows in Section 4.

3 Indexing the Results of Queries over Time-Based Windows

As seen in the previous section, a partitioned sliding window index balances two requirements: clustering by search key for efficient probing and by tuple arrival time so that updates are confined to one sub-index. Indexing the results of sliding window queries involves three conflicting requirements: clustering by search key for efficient probing, by arrival time for efficient insertions, and by expiration time for efficient deletions. In this section, we propose a solution for indexing time-evolving data whose generation order is different from the expiration order. The idea, which we call a doubly partitioned index, is to simultaneously divide the data by generation and expiration time. We begin by discussing indexing the results of weak non-monotonic queries and present extensions to strict non-monotonic queries in Section 3.5.

3.1 Variants of Doubly Partitioned Indices

A simple example of a doubly partitioned index (we will present an improved variant shortly) is shown in Figure 3 for a result over a 16-minute window updated every 2 minutes, as in Figure 1. Each of the four sub-indices contains a directory on the search key. Note that the ranges of insertion and expiration times are chronologically divided into two partitions each, creating a total of four sub-indices. For example, at time 16, I_1 stores results generated between times one and 8 that will expire between times 17 and 18. The update illustrated on top takes place at time 18, inserts new tuples into I_1 and I_2, and deletes expired tuples from I_1 and I_3. The next update occurs at time 20 and is shown at the bottom, with circled intervals highlighting changes made by the previous update. Again, new results are inserted into I_1 and I_2, and expired tuples are deleted from I_1 and I_3. The situation is similar for the next two updates in that I_4 does not have to be accessed. 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 partitioning the insertion and expiration times chronologically is that the sub-indices may have widely different sizes. In Figure 3, at time 16, I_2 stores tuples 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 result tuples that remain in the answer set for a time longer than S_{SW}. As a result, the other sub-indices are large and their update costs may dominate the overall maintenance cost.

We address this problem by adjusting the timestamp intervals spanned by each sub-index. The improved technique, which we call round-robin partitioning, is illustrated in Figure 4 for the same parameters as in Figure 3 (result over 16-minute windows updated every two minutes). Rather than dividing the insertion and expiration time ranges chronologically, it distributes updates in a round-robin fashion such that no sub-index experiences two consecutive insertions or expirations. For instance, the update illustrated on top of Figure 4 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 is illustrated on the bottom of Figure 4, with circled intervals marking changes made by the previous update. It 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

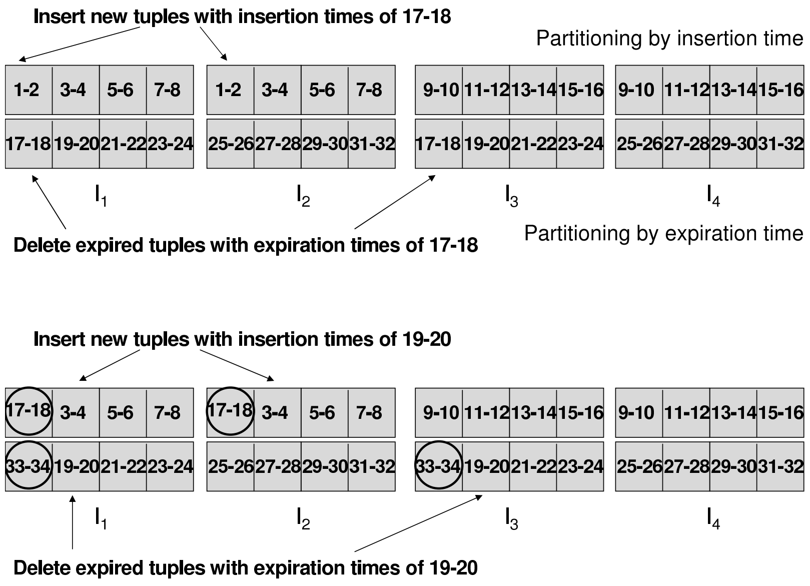
Insert new tuples with insertion times of 17-18
Partitioning by insertion time
1-2
3-4
5-6
7-8
9-10
11-12
13-14
15-16
17-18
19-20
21-22
23-24
25-26
27-28
29-30
31-32
I1
I2
I3
I4
Delete expired tuples with expiration times of 17-18
Partitioning by expiration time
Insert new tuples with insertion times of 19-20
33-34
Delete expired tuples with expiration times of 19-20

Figure 3: Example of a doubly partitioned index, showing updates at times 18 (top) and 20 (bottom).

sub-indices have similar sizes. In Appendix A, we prove the following theorem.

Theorem 1 *Given a constant rate of insertion into the result, the average variance of sub-index sizes using round-robin partitioning is lower than the average variance of sub-index sizes using chronological partitioning.*

Doubly partitioned indices are compatible with two bulk-update strategies. In what we call *Rebuild*, updated sub-indices are reclustered such that all the tuples with the same search key are contiguous. In what we call *NoRebuild*, sub-indices consist of pre-allocated buckets and insertions may create new buckets, whereas deletions may cause some buckets to be partially empty. These are similar to *REINDEX* and *DEL* from [17], but note that our techniques partition the data by insertion and expiration time, and assign updates to sub-indices in a round-robin fashion rather than chronologically. Moreover, we can apply an update strategy resembling *WATA*, but only for the insertion times[3]. An example is shown in Figure 5 for the same parameters as in Figures 3 and 4; the sizes of the generation time partitions have been chosen as outlined in Section 2.2. The update illustrated on top (at time 18) does not access I_1 and I_2, and neither does the next update shown on the bottom (at time 20). In fact, I_1 and I_2 will not be updated until time 30, at which point all of their tuples will have expired and will be replaced with newly arrived tuples. Meanwhile, all the insertions up to time 30 are routed to I_3 and I_4. We call this index type *LazyNoRebuild* or *LazyRebuild*, depending on the clustering technique. Note that chronological partitioning is only used for insertion times, whereas expiration times are

[3]A *WATA* index divided by expiration time does not make sense because insertions would need access to each sub-index. However, if every sub-index must be accessed, we may as well remove expired tuples in the oldest sub-index at the same time.

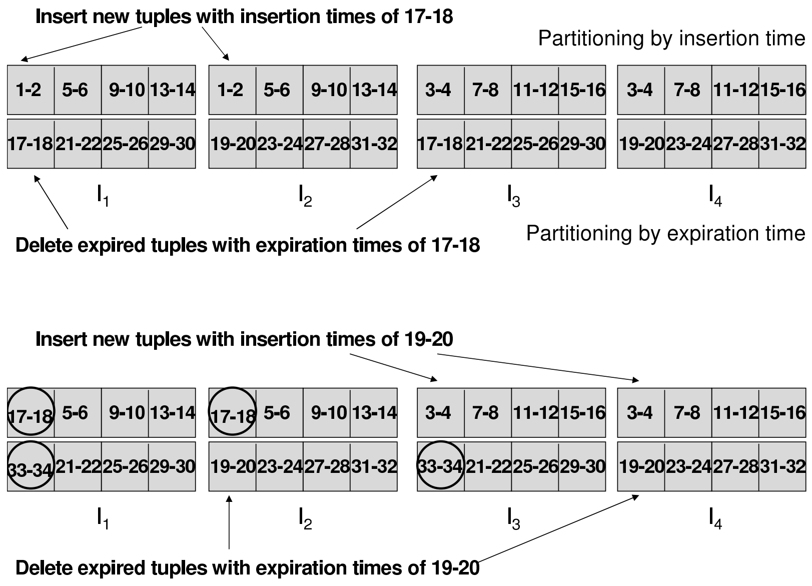
Insert new tuples with insertion times of 17-18
Partitioning by insertion time
1-2
5-6
9-10
13-14
1-2
5-6
9-10
13-14
3-4
7-8
11-12
15-16
3-4
7-8
11-12
15-16
17-18
21-22
25-26
29-30
19-20
23-24
27-28
31-32
17-18
21-22
25-26
29-30
19-20
23-24
27-28
31-32
I1
I2
I3
I4
Delete expired tuples with expiration times of 17-18
Partitioning by expiration time
Insert new tuples with insertion times of 19-20
17-18
17-18
33-34
33-34
Delete expired tuples with expiration times of 19-20

Figure 4: Example of a round-robin doubly partitioned index, showing updates at times 18 and 20.

Table 1: Summary of Doubly Partitioned Indexing Techniques

	Insertion Times	Deletion Times	Recluster?
Rebuild	*RR*	*RR*	Yes
NoRebuild	*RR*	*RR*	No
LazyRebuild	*Chr*	*RR*	Yes
LazyNoRebuild	*Chr*	*RR*	No

partitioned in a round-robin fashion to ensure that at least some of the sub-indices have similar sizes.

Table 1 summarizes the doubly partitioned indices in terms of the partitioning strategies of the insertion and deletion times, and the decision to recluster the data after updates. *RR* denotes round-robin partitioning and *Chr* denotes chronological partitioning. Appendix B contains algorithms for all the doubly partitioned indexing techniques discussed in this section.

3.2 Cost Analysis

Let n be the number of sub-indices, G be the number of partitions of generation (insertion) times, and E be the number of partitions of expiration times ($n = G \times E$). The number of sub-indices accessed during each update is $G + E - 1$ for *Rebuild* and *NoRebuild*, and $G + E - 2$ for *LazyRebuild* and *LazyNoRebuild*. The latter two update one fewer sub-index because sub-indices associated with the oldest insertion times are not accessed (recall Figure 5). To choose optimal

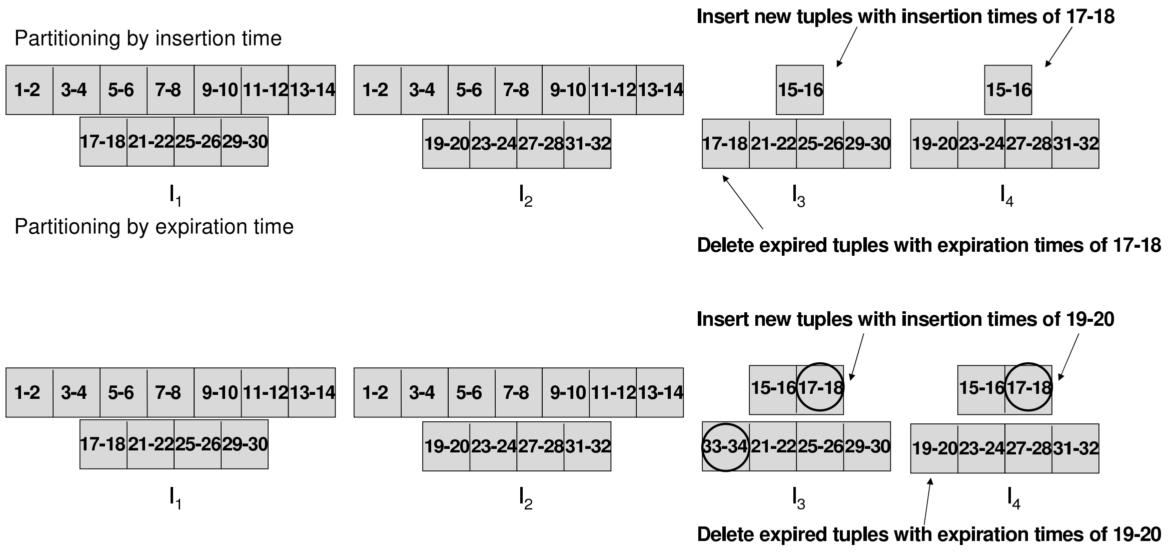
Partitioning by insertion time
Partitioning by expiration time
Insert new tuples with insertion times of 17-18
Delete expired tuples with expiration times of 17-18
Insert new tuples with insertion times of 19-20
Delete expired tuples with expiration times of 19-20
1-2
3-4
5-6
7-8
9-10
11-12
13-14
17-18
21-22
25-26
29-30
19-20
23-24
27-28
31-32
15-16
33-34
I1
I2
I3
I4

Figure 5: Example of a *LazyRebuild* or *LazyNoRebuild* index, showing updates at times 18 and 20.

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 in-memory 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, *Rebuild* with $G = E = 2$ updates three out of four sub-indices, but increasing G and E to four means that only seven of the sixteen sub-indices are accessed during updates. We expect that increasing n initially decreases update times, but eventually a breakpoint may be reached where the individual sub-indices are small and making any further splits is not helpful. The breakpoint value of n should therefore 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 the fastest to query, but slowest to update (especially as the data size grows, because rebuilding large indices may be expensive). Access into *NoRebuild* is slower because tuples 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. Finally, *LazyRebuild* and *LazyNoRebuild* should have the fastest update times (they always update one fewer sub-index than the other methods), but the slowest query times and highest space usage (they may store some expired tuples). For example, the index in Figure 5 does not delete expired tuples from I_1 and I_2, therefore it requires up to 50 percent more space than an equivalent *Rebuild* or

NoRebuild index (this occurs when I_1 and I_2 are nearly full with expired tuples and are about to be rebuilt). In general, the space overhead of *LazyRebuild* and *LazyNoRebuild* is less than or equal to $\frac{1}{G}$.

3.3 Dealing with Fluctuating Stream Conditions

The preceding discussion assumed uniform query result generation rates, in which case Theorem 1 guarantees similar sub-index sizes with round-robin partitioning. However, in the worst case, the result generation 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 result generation rate to arise from two factors: either the stream arrival rates are bursty or the attribute value distribution changes. Furthermore, the change may be persistent for several updates, or short-lived between two consecutive updates. In both cases, we expect the round-robin partitioning techniques 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 a query begins to produce more (or fewer) results, 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 results could be inserted into a large sub-index, which would become even larger.

3.4 Sharing Results Among Similar Queries

For scalability reasons, data stream systems are expected to evaluate similar queries with different window sizes together rather than forcing each query to probe the index separately [12]. That is, one index probe is performed per group of similar queries, followed by a filtering step that distributes results among the queries as appropriate. Doubly partitioned indices are compatible with this requirement by allowing the extraction of sub-results over windows with smaller sizes. In the simplest case of shortening each window by the same amount, we ignore result tuples with the oldest expiration times during index probes. For example, a join result of windows of size 20 can be used by a join of windows of size 16 by omitting results that are scheduled to expire within the next four time units. Similarly, a join result of three windows of size 16, 18, and 20 can be used by a join of windows of size 14, 16, and 18 by discarding results scheduled to expire within the next two time units.

If the windows are not shortened by the same amount, our indexing techniques still apply. However, it is necessary to store additional timestamps within result tuples, corresponding to windows whose sizes change across queries. For instance, consider using a join result over j windows of size 20 each to compute a join of the first $j-1$ windows of size 20, and the last window of size 10. Doing so requires storing three timestamps per result tuple: generation time, expiration time, and the timestamp of the participating tuple from the last window. The tradeoff is that storing additional timestamps could offset the advantages of sharing—the extra memory used by timestamps may be higher than maintaining two separate results. We intend to study this issue in future work.

3.5 Note on Strict Non-Monotonic Queries

As discussed in Section 3.1, some results of strict non-monotonic queries expire naturally as determined by timestamps, but others may expire prematurely if they cease to satisfy the query semantics. In terms of index partitioning, this implies that deletions may occur in any sub-index at any time. Our solution is to defer premature deletions from indices which are not scheduled to be updated during a particular time interval. For example, in Figure 4, I_4 is not updated at time 18, but it is updated at time 20, so we propose to postpone updates of I_4 caused by premature expirations at time 18 until time 20. Meanwhile, we store a list of invalidated key values and refer to this list when probing I_4 to ensure that invalidated tuples therein are ignored by queries. Relative to indexing the results of weak non-monotonic queries, this modification results in slightly longer probe times and higher memory usage due to storing invalidated tuples.

4 Extensions to Count-Based Sliding Windows

The order in which queries over count-based windows expire their results is more difficult to characterize because expirations depend on the arrival rates of the input streams. We now present a modification of our indexing techniques that applies in this case, provided that the variation in the arrival rates can be bounded.

4.1 Motivation

Maintenance of individual count-based windows is simple: not only are tuples expired in the same order as they arrive, but each insertion triggers exactly one expiration. However, this is not the case for query results over count-based windows—even a simple query such as selection does not contain the same number of results at all times. Furthermore, the nature of timestamps in count-based windows severely complicates the analysis of expiration patterns.

Assume that new tuples on each stream are assigned consecutive integer “timestamps” that determine their relative position in the window. Consider a join of three count-based windows of size 50 each, W_1, W_2, and W_3, with all windows storing tuples with timestamps 1 through 50 at the current time. Suppose that a result tuple t consists of three individual tuples from the three input windows, having timestamps of one, two and two, respectively. To determine when t expires, we verify if at least one of its constituent tuples has expired from the corresponding window. Thus, t expires either when tuple with timestamp 51 is inserted into W_1, or when tuple with timestamp 52 is inserted into W_2, or when tuple with timestamp 52 is inserted into W_3, whichever comes first. In other words, we need one new insertion into W_1, or two new insertions into W_2 or W_3. However, we do not know when these new tuples will arrive. In the general case, result expiration times cannot be predicted, meaning that queries over count-based windows are strict non-monotonic (recall Section 2.3). This is different from queries over time-based windows, where timestamps are defined in terms of absolute time and result expiration times are independent of the arrival rates. As a result, it may not be possible to design a partitioned index over count-based window query results that does not have to be scanned in its entirety during every bulk-update.

4.2 Solution

The general problem appears intractable; however, we can at least estimate the expiration times of result tuples based on the observed stream arrival rates. Although the estimates may be incorrect because the arrival rates may fluctuate, we can attempt to use our doubly partitioned indices and periodically insert (and delete) result tuples into particular sub-indices based on their guessed expiration times. To do this, each result tuple stores as many timestamps as there are windows referenced in the query, denoting the relative positions of all the base tuples in their windows. All of these timestamps are needed to determine if the result tuple has in fact expired. The guessed expiration timestamps are first used to select the sub-index into which new results should be inserted, and then to select which sub-indices are due for an update. The problem is that deletions become more complicated and may require access to more sub-indices than before.

To see why expirations are more difficult, consider a *Rebuild* or *NoRebuild* index from Figure 4. When an update takes place at time 18, newly generated tuples with insertion times of 17 through 18 are added to I_1 and I_2, as before. However, it is not immediately clear which result tuples should be examined for possible expiration. First, suppose that the stream arrival rates speed up, causing some result tuples to expire sooner than we had originally estimated. At time 18, instead of deleting result tuples with expiration times between 17 and 18, we may need to examine tuples with estimated expiration times between 17 and, say 20. On the other hand, if the arrival rates slow down, some result tuples with estimated expiration times of 17 to 18 may not expire until the next update at time 20, as illustrated in Figure 6. This is why it is necessary to assume a bounded variation in the stream arrival rates. This way, each update may be associated with a range of guessed expiration times within which we are sure to find all the expired tuples. For simplicity, we assume that it is sufficient to scan the expiration range assigned to the current update, as well as the previous and next update. For example, the update illustrated in Figure 6 will scan the expiration time range of 19 and 20, as well as 17 to 18 and 21 to 22.

Given that a range of guessed expiration times may be computed for each update, the remaining problem with storing the results of count-based window queries in a doubly partitioned index is that expirations may need to access more sub-indices than before. Returning to the above examples, note that all four sub-indices in Figure 4 need to be accessed if the arrival rates increase and the examined expiration times at time 18 are no longer 17 to 18, but rather 17 to 20. This is different from time-based windows, where even if the arrival rates fluctuate, tuples which should be deleted at time 18 are known to have expiration times of 17 to 18. Similarly, all four sub-indices in Figure 6 may need to be accessed at time 20 if the arrival rates slow down. This is because the update at time 20 expires old tuples from different sub-indices than the previous update at time 18.

As seen above, round-robin partitioning does not cope well with errors in estimating the expiration times. Chronological partitioning may be better suited, but, as discussed in Section 3.1, it produces sub-indices with different sizes, even if the stream arrival rates are fixed. Fortunately,

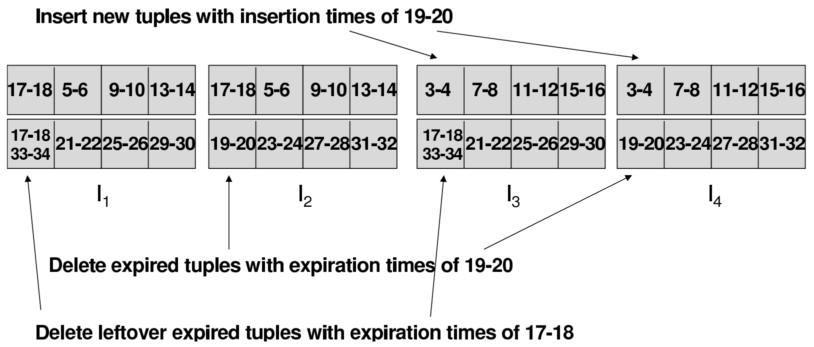

Figure 6: Updating a doubly partitioned *Rebuild* or *NoRebuild* index over count-based window query results. The illustrated update occurs at time 20.

Table 2: Summary of All Doubly Partitioned Indexing Techniques

	Insertion Times	Deletion Times	Recluster?
Rebuild	*RR*	*RR*	Yes
NoRebuild	*RR*	*RR*	No
LazyRebuild	*Chr*	*RR*	Yes
LazyNoRebuild	*Chr*	*RR*	No
CountRebuild	*RR*	*Chr*	Yes
CountNoRebuild	*RR*	*Chr*	No

in the context of queries over count-based windows, chronological partitioning is needed only for expiration times because the type of partitioning done on the insertion times does not affect deletions. Thus, our proposed solution retains round-robin partitioning for generation times to help ensure that the sub-index sizes are similar. The resulting structure, which we call *CountRebuild* or *CountNoRebuild* depending on the clustering technique, is illustrated in Figure 7, for the same parameters as in Figure 3 (16-minute window split into four sub-indices and updated every two minutes). An update at time 18 is shown on the top, assuming that there are no changes in the stream arrival rates (that is, assuming that expired tuples at time 18 all have guessed expiration times of 17 to 18). On the bottom, we show an update at time 20, given that the arrival rates may have increased or decreased. In either case, it is usually possible to localize expirations to two sub-indices due to chronological partitioning of the expiration times (but not always, e.g., all four sub-indices need to be accessed in order to scan the expiration time range of 23 to 26). In the context of count-based windows, updating *CountRebuild* and *CountNoRebuild* should therefore incur fewer sub-index reads and writes than *Rebuild* and *NoRebuild*, at the expense of more variation in sub-index sizes.

Table 2 summarizes all of the proposed indexing strategies, including those presented in Section 3.1 for the results of queries over time-based windows. As in Table 1, we list the partitioning strategies for generation and expiration times (*RR* denotes round-robin partitioning and *Chr* denotes chronological partitioning), and whether or not updated sub-indices are reclustered. We

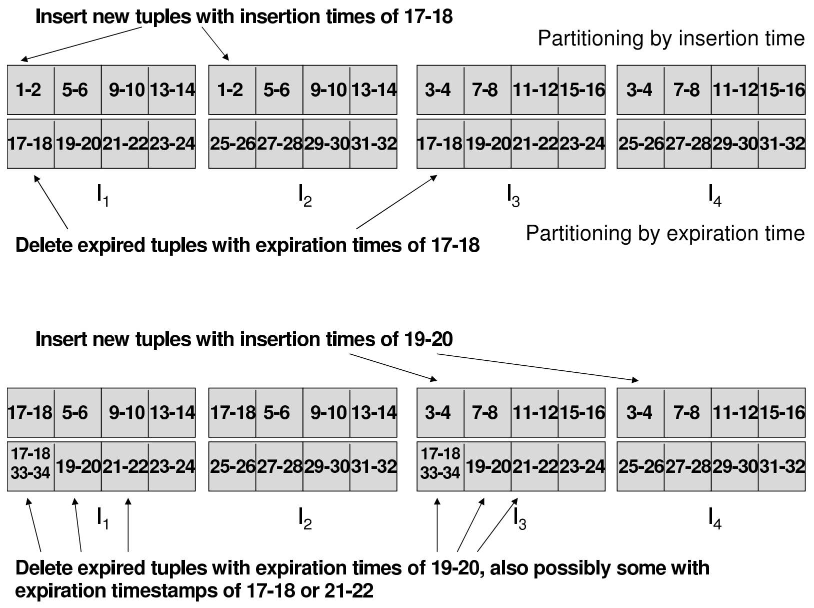

Figure 7: Example of a round-robin/chronological partitioning strategy for indexing results of queries over count-based windows (*CountRebuild* or *CountNoRebuild*), showing updates at times 18 and 20.

do not consider adapting *LazyRebuild* and *LazyNoRebuild* to queries over count-based windows because these would require chronological partitioning of both insertion and expiration times, thereby causing exceedingly large variance of sub-index sizes. Appendix B contains an algorithm for *CountRebuild* and *CountNoRebuild*.

5 Experiments

This section contains an overview of our implementation (Section 5.1) and experimental results. Sections 5.2 through 5.4 present results of experiments with a small window size of approximately 500 Megabytes (this corresponds to 500000 time units, with an average of one result tuple generated per time unit). In Section 5.5, we investigate index performance over larger windows with sizes of up to 5 Gigabytes (i.e., 5 million time units, with an average of one result tuple per unit time). We then present performance results for strict non-monotonic queries (Section 5.6), and queries over count-based windows (Section 5.7). Our experimental findings are summarized in Section 5.8.

5.1 Implementation Details

We implemented the doubly partitioned indices (*Rebuild*, *NoRebuild*, *LazyRebuild*, *LazyNoRebuild*, *CountRebuild*, and *CountNoRebuild*) 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 the indices from [17], split by insertion or expiration time: *REINDEX*

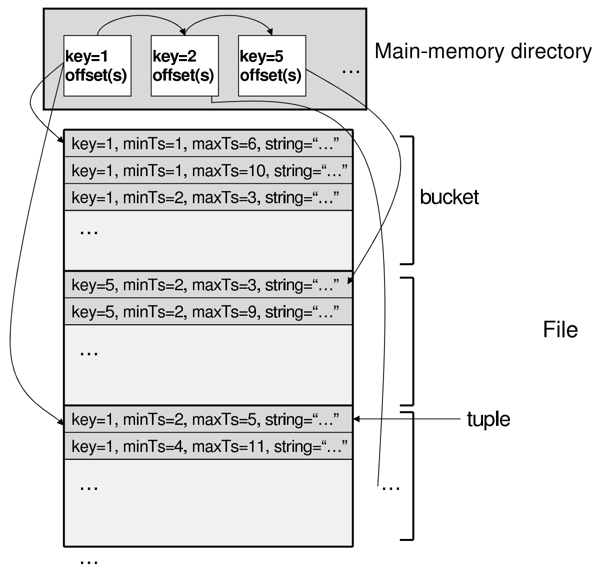
key=1
offset(s)
key=2
offset(s)
key=5
offset(s)
...
Main-memory directory
key=1, minTs=1, maxTs=6, string="..."
key=1, minTs=1, maxTs=10, string="..."
key=1, minTs=2, maxTs=3, string="..."
...
bucket
key=5, minTs=2, maxTs=3, string="..."
key=5, minTs=2, maxTs=9, string="..."
...
File
key=1, minTs=2, maxTs=5, string="..."
tuple
key=1, minTs=4, maxTs=11, string="..."
...
...
...

Figure 8: The structure of one sub-index in *NoRebuild*, *LazyNoRebuild*, *CountNoRebuild*, *DEL*, and *WATA*.

(*R-ins* or *R-exp*), *DEL* (*D-ins* or *D-exp*), and *WATA* (split by insertion time only). In this section, we 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 simulate one window of results from a join of m windows using a specified distribution of search key values (uniform or Power Law). Next, we generate periodic updates using the same key distribution and insert them into the index, at the same time removing expired tuples, until the window rolls over. 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.

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 tuples are 1000 bytes long and contain an integer search key, integer insertion and expiration timestamps, as well as a string that abstractly represents the contents of the tuple. The structure of one sub-index in *NoRebuild*, *LazyNoRebuild*, *CountNoRebuild*, *DEL*, and *WATA* is illustrated in Figure 8, showing the directory with offset pointers to locations of buckets in the file (we also store how many tuples are in each bucket as not all buckets are full). *Rebuild*, *LazyRebuild*, *CountRebuild*, 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 [6] in the context of skewed distributions and is orthogonal to this work.

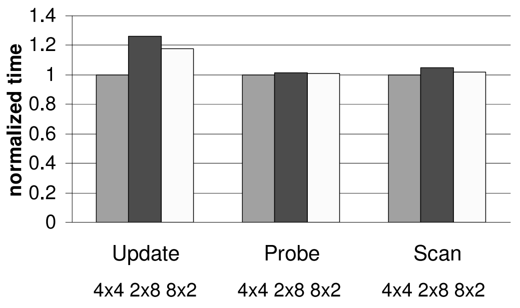

Figure 9: Relative performance of *Rebuild*4x4, *Rebuild*2x8, and *Rebuild*8x2.

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 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 search tree, and is orthogonal to this work. Third, the number of tuples per bucket in *NoRebuild*, *LazyNoRebuild*, *CountNoRebuild*, 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). Finally, we update the results 36 times until the window rolls over. Varying the update frequency demonstrates a clear tradeoff between the total update time and the average answer latency [8], and is not discussed further.

5.2 Optimal Values for G and E

We begin by validating our results from Section 3.2 regarding the optimal assignment of values for G (number of partitions of generation times) and E (number of partitions of expiration times) given a value for n (number of sub-indices). Figure 9 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 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.

5.3 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. One exception is that *LazyNoRebuild*2x2 and *LazyRebuild*2x2 are not tested due to their excessive space overhead of up to fifty percent (recall Section 3.2). We also 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 10, 11, and 12 show the average update, probe, and scan times, respectively, as functions of n (number of sub-indices). Figure 10 additionally shows the update times of doubly partitioned indices with chronological splitting on both levels (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, *LazyNoRebuild* performs the fastest updates, followed by *LazyRebuild*, *NoRebuild*, and *Rebuild*. Conversely, *Rebuild* has the lowest probe and scan times, followed by *NoRebuild*, *LazyRebuild*, and *LazyNoRebuild*.

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 window size in this experiment, 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 all the search keys are 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 *REINDEX*, *DEL*, and *WATA*). This confirms our hypothesis from Section 3.2 regarding the effects of the two factors influencing 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*, *D-ins*, and *WATA* start increasing for smaller values of n than those for our two-level techniques: the existing techniques access all n sub-indices during updates, whereas our techniques only access $O(G + E) = O(\sqrt{n})$ sub-indices.

Given a fixed value of n, *Rebuild* and *R-ins* both have the lowest space requirements, followed by *NoRebuild* and *D-ins*, and then by *WATA*, *LazyRebuild*, and *LazyNoRebuild*. *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). Additionally, *WATA*, *LazyRebuild*, and *LazyNoRebuild* may store expired tuples (recall Section 3.2). As

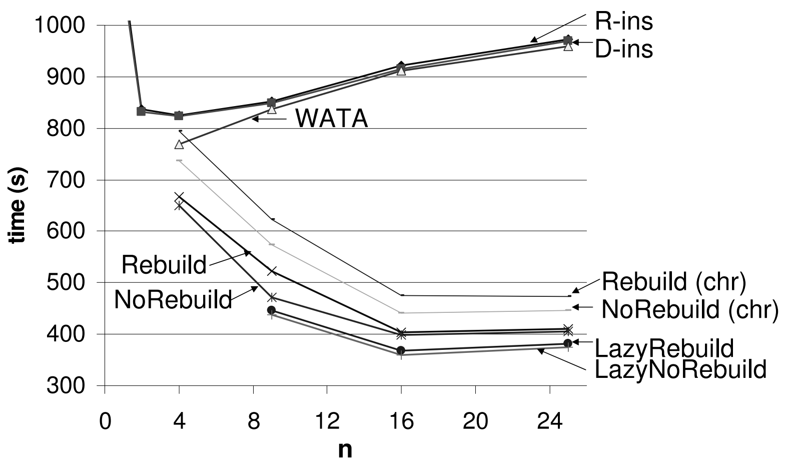

Figure 10: Update times of index partitioning techniques given a small window size.

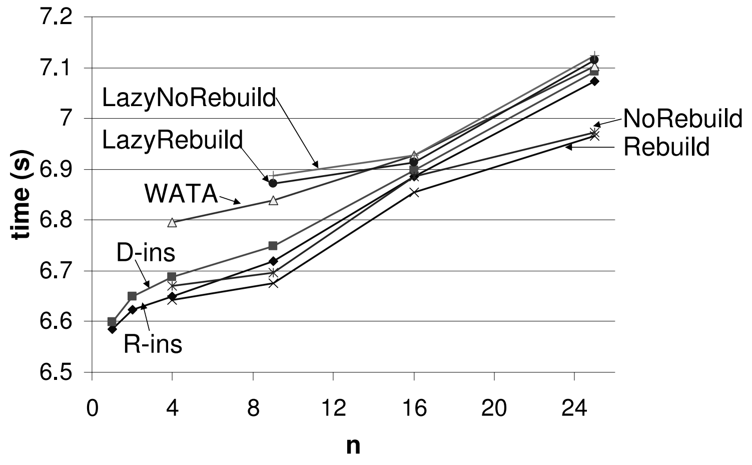

Figure 11: Probe times of index partitioning techniques given a small window size.

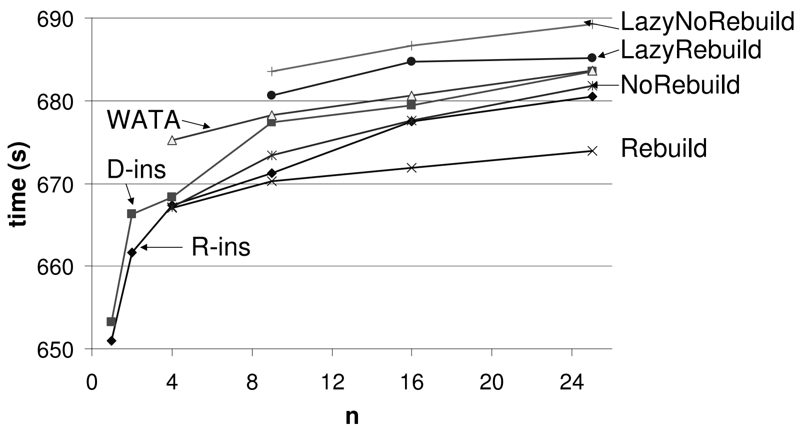

Figure 12: Scan times of index partitioning techniques given a small window size.

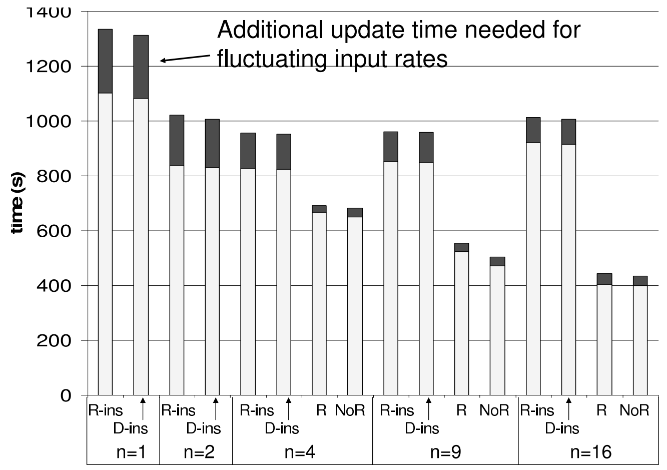

Figure 13: Effect of fluctuating stream conditions on index update performance.

n increases, all techniques require more space to store the individual 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 12, 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.

5.4 Adapting to Fluctuating Stream Conditions

In this experiment, the result generation rate varies randomly by a factor of up to four. We set the total amount of results 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 13 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 result generation rate.

Doubly partitioned indices are more adaptable to changing input 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 result generation 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.

5.5 Scaling up to Large Window Sizes

This test investigates the behaviour of our techniques when indexing large amounts of data. The average update, probe, and scan times as functions of n are shown in Figures 14, 15, and 16, respectively. Compared to results using a smaller window (Figures 10, 11, and 12), doubly partitioned indices are now up to three times as fast to update as the existing techniques. As before, updates are the fastest for *LazyNoRebuild* and *LazyRebuild*, followed by *NoRebuild* and *Rebuild*, whereas *Rebuild* is the fastest to probe and requires the least space, followed by *NoRebuild*, *LazyRebuild*, and *LazyNoRebuild*. 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 11 are less than one percent and roughly three percent, respectively. This is the expected outcome of using a larger window size: *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 14 and 10 is the behavior of update times as n grows. In Figure 10, there is a turning point (at $n = 16$ for *NoRebuild*, *Rebuild*, *LazyNoRebuild*, and *LazyRebuild*) after which update times do not decrease. This is not the case in Figure 14, 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 $n \leq 25$.

5.6 Index Performance with Strict Non-Monotonic Queries

There is an overhead associated with maintaining indexed results of strict non-monotonic queries, namely storing and keeping track of invalidated tuples that have not yet been deleted. The time needed to maintain a linked list of invalidated keys during updates (sorted by search key) and the additional time to scan this list during index probes do not contribute significantly to the overall cost. According to our tests, the update and index probe overhead was on the order of five percent. The overhead is modest because only some of the result tuples expire prematurely, whereas the others expire in the same way as in weak non-monotonic queries and can be handled easily by our indexing techniques.

5.7 Queries Over Count-Based Windows

In the final test, we investigate index performance for queries over count-based windows. As in Section 5.4, the arrival rate is allowed to vary randomly by a factor of up to four. We test *D-ins*, *R-ins*, *Rebuild*, *NoRebuild*, as well as *CountRebuild* and *CountNoRebuild*. Result expiration times are estimated on the basis of the arrival rates at the time that the results were generated. Index scan and probe times do not vary significantly from the reported results for time-based windows, therefore only the update times of selected techniques are shown in Figure 17. These results were generated using a small window size; similar relative results were obtained with larger window sizes. Our best doubly partitioned techniques (*CountRebuild* and *CountNoRebuild*) outperform the existing approaches, but the difference is not as great

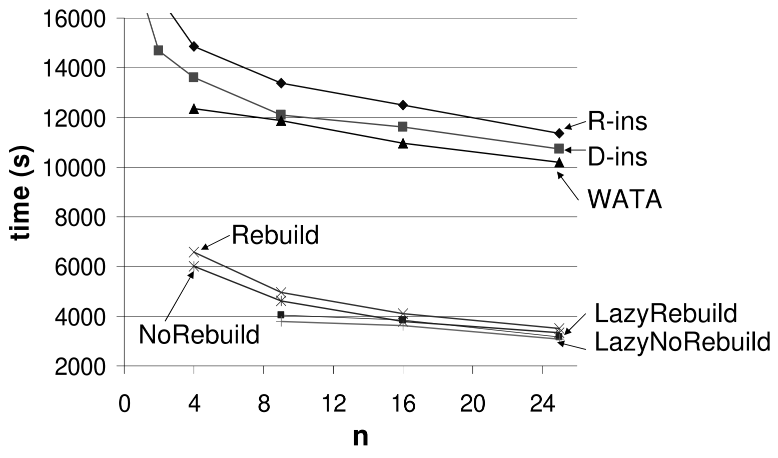

Figure 14: Update times of index partitioning techniques given a large window size.

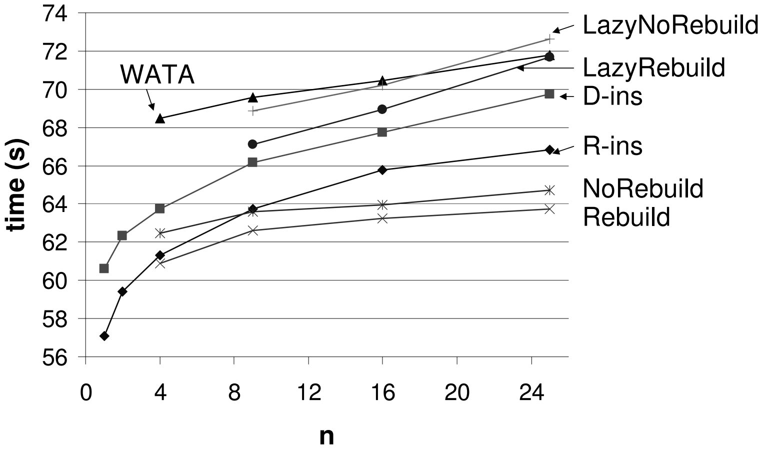

Figure 15: Probe times of index partitioning techniques given a large window size.

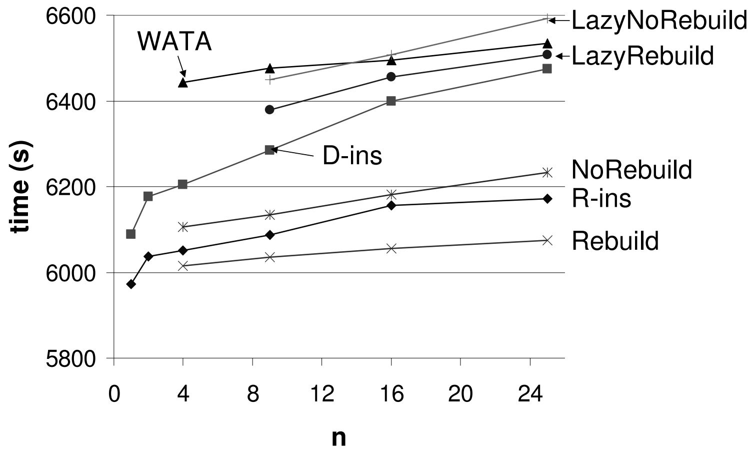

Figure 16: Scan times of index partitioning techniques given a large window size.

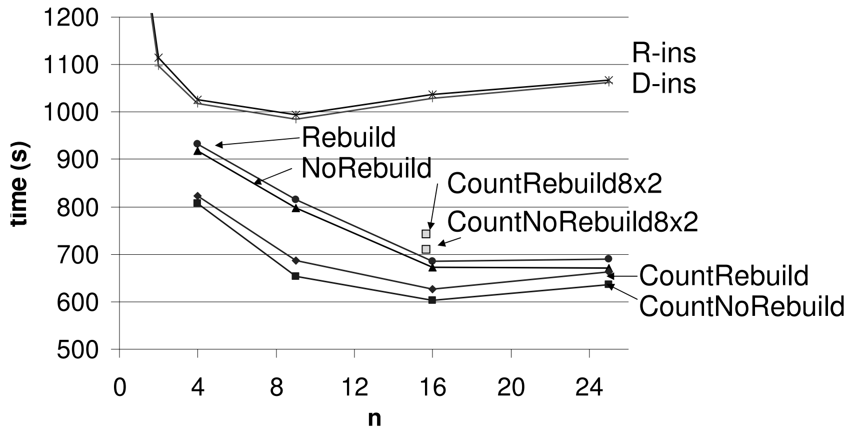

Figure 17: Index update performance over count-based windows.

as with time-based windows (recall Figures 10 and 13). This is because *CountRebuild* and *CountNoRebuild* use round-robin partitioning only for the insertion times, therefore the sub-index sizes are not as uniform as in *Rebuild* and *NoRebuild*. Also note that *CountRebuild*4x4 and *CountNoRebuild*4x4 outperform *CountRebuild*8x2 and *CountNoRebuild*8x2 (*CountRebuild*2x8 and *CountNoRebuild*2x8 are even slower), meaning that setting $G = E$ remains a good choice in the context of count-based windows.

*CountRebuild*2x2 and *CountNoRebuild*2x2 are faster to update than *Rebuild*2x2 and *NoRebuild*2x2, but the difference diminishes as n increases. Moreover, the advantage of *CountRebuild* and *CountNoRebuild* relative to *Rebuild* and *NoRebuild* is greater if the arrival rates fluctuate more widely. The reason is that chronological partitioning is less helpful when the number of sub-indices is large because each sub-index covers a small time span, so it is likely that deletions will require additional accesses into other sub-indices. On the other hand, chronological partitioning is useful for small values of n (and when the arrival rates fluctuate widely), where using round-robin partitioning for the guessed expiration times essentially means that all four sub-indices are always accessed during updates (recall Section 4.2). Finally, update times increase as more count-based windows are referenced in the query because of the increase in the total size of the data being indexed; recall that each result tuple of a count-based window query must store one timestamp for each window referenced in the query.

5.8 Lessons Learned

We make the following general recommendations regarding the best index partitioning strategy. Our guidelines depend on the data size and the number of continuous queries that are executed between updates.

- For a small window size and small number of queries, *NoRebuild*4x4 is a good choice as it incurs low update times. *LazyNoRebuild*4x4 is another good choice, but it has higher memory requirements.
- 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.

- 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). *LazyNoRebuild* is not recommended as its increased memory requirements may be a problem as the data size grows.

- 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.

- In general, the same guidelines apply for queries over count-based windows with bounded variations in the stream arrival rates, except that *CountRebuild* and *CountNoRebuild* should be used instead of *Rebuild* and *NoRebuild*, especially if the fluctuations in the arrival rates are large.

6 Comparison with Related Work

This work is related to indexing time-evolving data, analyzing the expiration order of the results of sliding window queries, and answering sliding window queries using materialized views. Previous work on indexing sliding windows partitions the data by arrival time [8, 17], which, as we have shown, is not compatible with data that do not expire in order of arrival. The expiration patterns of the results of sliding window queries were first analyzed in our previous work on update-pattern-aware query processing [9]. 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 must be used.

Relevant work on evaluating sliding window queries using materialized views includes choosing which intermediate results to materialize in the context of sliding window joins [2], and re-ordering query operators (e.g., pushing down joins) to ensure that results may be shared efficiently among similar queries [4, 5, 14]. These issues are orthogonal to our work on designing appropriate indexing techniques once an intermediate result is chosen for materialization.

The continuous query workload may fluctuate over time, therefore it may be necessary to switch between various indexing techniques at some point. This problem is similar to plan migration in the context of sliding window queries that store state [19]. 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. We intend to study the tradeoffs involved in these two approaches in future work.

As discussed in Section 2.3, results of sliding window queries are associated with time intervals corresponding to their lifetimes in the answer set. This suggests a connection with spatio-temporal interval indexing (see [16] for a survey). The important difference in our work is that we do not explicitly index the actual intervals, but rather we index individual tuples by their search keys. It is possible to build a two-dimensional R-tree index over the time intervals spanned by individual sub-indices, in which case looking up sub-indices which are affected by a given update could be faster. We did not need to employ this optimization in this work because the number of sub-indices we were dealing with was relatively small.

7 Conclusions

In this paper, we proposed and evaluated solutions for indexing the results of sliding window queries, taking advantage of the order in which result tuples are generated and expired. The challenge of accommodating time-evolving data whose generation order is different from the expiration order was solved by simultaneously partitioning the index by insertion time and expiration time. Experimental results showed significant improvements in index update times under various system conditions. Our indexing techniques may be used by data stream systems to speed up sliding window queries over a disk-based archive and to improve scalability by sharing indexed results among similar queries. Future work includes the following issues:

- addressing consistency issues in sliding window indices due to making changes in-place or replacing an entire sub-index with a copy on which updates have been made;
- improving the performance of indices over count-based window queries by providing more accurate estimates of expiration times of result tuples;
- investigating the tradeoffs involved in materializing a shared result for queries over windows with different sizes versus maintaining separate results.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug 2003.

[2] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Proc. 21st Int. Conf. on Data Engineering*, 2005, to appear.

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

[4] J. Chen, D. DeWitt, and J. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. 18th Int. Conf. on Data Engineering*, pages 345–357, 2002.

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

[6] C. Faloutsos and H. Jagadish. On B-tree indices for skewed distributions. In *Proc. 18th Int. Conf. on Very Large Data Bases*, pages 363–374, 1992.

[7] L. Golab, D. DeHaan, E. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, pages 173–178, 2003.

[8] L. Golab, S. Garg, and M. T. Özsu. On indexing sliding windows over on-line data streams. In *Advances in Database Technology — EDBT'04*, pages 712–729, 2004.

[9] L. Golab and M. T. Özsu. Update-pattern aware modeling and processing of continuous queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2005, to appear.

[10] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proc. 12th Int. Conf. on Data Engineering*, pages 152–159, 1996.

[11] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A. Elmagarmid. Efficient execution of sliding window queries over data streams. Technical Report CSD TR 03-035, Purdue University, 2003.

[12] M. Hammad, M. J. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 297–308, 2003.

[13] J. Krämer and B. Seeger. A temporal foundation for continuous queries over data streams. In *Proc. 11th Int. Conf. on Management of Data (COMAD)*, pages 70–82, 2005.

[14] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 972–986, 2004.

[15] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res.*, pages 245–256, 2003.

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

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

[18] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. 23rd ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems*, pages 263–274, 2004.

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

Appendix A

We now prove Theorem 1 from Section 3.1, which we repeat below for convenience.

Theorem 1 *Given a constant rate of insertion into the result, the average variance of sub-index sizes using round-robin partitioning is lower than the average variance of sub-index sizes using chronological partitioning.*

Examples

Recall that n is the number of sub-indices, S is the size of the (time-based) window, and p is the number of times that the window is updated until it rolls over completely. Thus, the window may be thought of as having size p, in units of r, the interval between two consecutive updates. We enumerate the insertion timestamp (ts) and expiration timestamp (call it exp) ranges as one through p. For simplicity, we set $exp = exp - S$ so that the two ranges are the same. Without loss of generality, we suppose that range 1 is the oldest and range p is the youngest. For example, in Figure 2, $n = 4$, $p = 8$, $r = 2$, and the ts and exp ranges of one through two are denoted by 1, through to the ranges of 15 to 16, which are denoted by p. Given this notation, a 2x2 chronologically partitioned index from Figure 3 may be characterized as follows.

- I_1 stores results with ts ranges of one through $\frac{p}{2}$ and exp ranges of one through $\frac{p}{2}$.
- I_2 stores results with ts ranges of one through $\frac{p}{2}$ and exp ranges of $\frac{p}{2}$ through p.
- I_3 stores results with ts ranges of $\frac{p}{2}$ through p and exp ranges of one through $\frac{p}{2}$.
- I_4 stores results with ts ranges of $\frac{p}{2}$ through p and exp ranges of $\frac{p}{2}$ through p.

We will calculate the sub-index sizes in terms of the expected number of result tuples they store. For simplicity, we assume that one unit of size corresponds to the number tuples with some ts range i and some exp range j, such that $1 \leq i, j \leq p$ and $i \geq j$. Due to our assumption of uniform result generation rate, all such range pairs contain the same number of results. For instance, the number of result tuples with $ts = 1$ and $exp = 2$ is the same as the number of those for which $ts = 5$ and $exp = 7$. Given this definition, the initial sub-index sizes of a 2x2 chronologically partitioned index are as follows.

- In I_1, tuples with $ts = 1$ can only have $exp = 1$, therefore there is one unit of them. Tuples with $ts = 2$ can have $exp = 1$ or $exp = 2$, therefore there are two units of results with $ts = 2$. Continuing to $\frac{p}{2}$, there are $\frac{p}{2}$ units of tuples with $ts = \frac{p}{2}$. The total size of I_1 is therefore $1 + 2 + \ldots + \frac{p}{2} = \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}$.
- I_2 is empty at this time.

- In I_3, tuples with $ts = \frac{p}{2} + 1$ can have $exp = 1$ up to $exp = \frac{p}{2}$, therefore there are $\frac{p}{2}$ units of them. The situation is similar for tuples with all the other ts values up to $\frac{p}{2}$, therefore the size of I_3 is $\frac{p}{2}\frac{p}{2} = \frac{p^2}{4}$.
- By the same reasoning as for I_1, the size of I_4 is the same as I_1, at $\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}$.

At all times, the total size of the result is $1 + 2 + \ldots + p$ (because there is one possible exp range for $ts = 1$, two possible exp ranges for $ts = 2$, and so on), which is $\frac{p(p+1)}{2}$. This is equivalent to adding up the sizes of I_1, I_2, I_3, and I_4 from above. Now, for the next $\frac{p}{2}$ updates, I_1 and I_2 incur insertions and some deletions, whereas I_3 and I_4 incur only deletions. After that, during the next $\frac{p}{2}$ updates, the process repeats, with the exception that I_1 and I_2 only incur deletions and I_3 and I_4 are also inserted into. Thus, there is a repeating sequence of $\frac{p}{2}$ distinct steps, during which the sub-indices evolve in size. This sequence may be derived by considering the sub-index sizes at each step, using the same procedure as above. Letting $(|I_1|, |I_2|, |I_3|, |I_4|)$ be the sizes of the four sub-indices at each of the $\frac{p}{2}$ steps, the sequence of sizes is as follows.

$$\left(\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}, 0, \frac{p^2}{4}, \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}\right), \left(\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}, \frac{p}{2}, (\frac{p}{2}-1)\frac{p}{2}, \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}\right), \ldots, \left(\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}, (\frac{p}{2}-1)\frac{p}{2}, \frac{p}{2}, \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}\right)$$

That is, the sizes of I_1 and I_4 do not change, whereas I_2 gains $\frac{p}{2}$ and I_3 loses $\frac{p}{2}$ after each of the $\frac{p}{2}$ updates. After $\frac{p}{2} + 1$ updates, the sub-index sizes are the same as in the first element of the sequence, with the exception that the size of I_2 is $\frac{p^2}{4}$ and the size of I_3 is zero.

A similar analysis may be carried out for a 2x2 round-robin index. For instance, we can characterize the index in Figure 4 as follows.

- I_1 stores results with ts ranges of $1, 3, \ldots, p-1$ and exp ranges of $1, 3, \ldots, p-1$.
- I_2 stores results with ts ranges of $1, 3, \ldots, p-1$ and exp ranges of $2, 4, \ldots, p$.
- I_3 stores results with ts ranges of $2, 4, \ldots, p$ and exp ranges of $1, 3, \ldots, p-1$.
- I_4 stores results with ts ranges of $2, 4, \ldots, p$ and exp ranges of $2, 4, \ldots, p$.

The initial sub-index sizes of a 2x2 round-robin index are as follows.

- In I_1, tuples with $ts = 1$ can only have $exp = 1$, therefore there is one unit of them. Tuples with $ts = 3$ can have $exp = 1$ or $exp = 3$, therefore there are two units of results with $ts = 3$. Continuing to $p - 1$, the possible exp values are $1, 3, \ldots, p-1$, therefore there are $\frac{p}{2}$ units of tuples with $ts = p - 1$. The total size of I_1 is $1 + 2 + \ldots + \frac{p}{2} = \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}$.
- In I_2, there are no tuples with $ts = 1$ because the smallest possible exp value is 2. Tuples with $ts = 3$ can only have $exp = 2$, therefore there is one unit of them. Tuples with $ts = 5$ can have $exp = 2$ or $exp = 4$, therefore there are two units of them. Continuing to $p - 1$, the possible ts values are $2, 4, \ldots, p-2$, therefore there are $\frac{p}{2} - 1$ units of tuples with $ts = p - 1$. The total size of I_2 is $1 + 2 + \ldots + \frac{p}{2} - 1 = \frac{(\frac{p}{2}-1)\frac{p}{2}}{2}$.

- In I_3, tuples with $ts = 2$ can only have $exp = 1$, therefore there is one unit of them. Tuples with $ts = 4$ can have $exp = 1$ or $exp = 3$, therefore there are two units of them. The rest of the analysis is similar to that of I_1, therefore the size of I_3 is the same as the size of I_1, at $\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}$.

- By the same reasoning as for I_1, the size of I_4 is the same as I_1, at $\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}$.

In round-robin partitioning, the sequence of sizes only has the following two states.

$$\left(\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}, \frac{(\frac{p}{2}-1)\frac{p}{2}}{2}, \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}, \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}\right), \left(\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}, \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}, \frac{(\frac{p}{2}-1)\frac{p}{2}}{2}, \frac{\frac{p}{2}(\frac{p}{2}+1)}{2}\right)$$

In fact, the two states are equivalent if we ignore the ordering within them, which is not significant in terms of calculating the variance in sub-index sizes. Thus, there is only one distinct set of sub-index sizes, in which three sub-indices have sizes of $\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}$ and one sub-index has a size of $\frac{(\frac{p}{2}-1)\frac{p}{2}}{2}$.

Generalization

Generalizing our discussion so far to all values of n is straightforward. First, we only consider symmetric splits with $\sqrt{n}$ upper-level and $\sqrt{n}$ lower-level partitions. We also require that each sub-index span at least two refresh intervals (otherwise, each sub-index would store tuples with one particular ts range and one particular exp range, and there would be no difference between chronological partitioning and round-robin partitioning). Thus, we require $p > \sqrt{n}$, meaning that the smallest value that p can take is $2\sqrt{n}$ (in which case, each sub-index spans a time of two refresh intervals of the insertion and generation times).

First, we generalize chronological partitioning. The sequence of sub-index sizes now has $\frac{p}{\sqrt{n}}$ steps, during which the same sub-indices incur insertions. The sub-indices have the following sizes.

- There are $\sqrt{n}$ sub-indices whose sizes are always $1 + 2 + \ldots + \frac{p}{\sqrt{n}} = \frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$.

- Of the remaining $n - \sqrt{n}$ sub-indices, half begin with size zero and half begin with size $(\frac{p}{\sqrt{n}})(\frac{p}{\sqrt{n}}) = \frac{p^2}{n}$. After each of the $\frac{p}{\sqrt{n}}$ updates, the former grow by $\frac{p}{\sqrt{n}}$ and the latter decrease by $\frac{p}{\sqrt{n}}$.

Next, to generalize round-robin partitioning, note that the sequence of sizes now has $\sqrt{n}$ steps, corresponding to applying one update to each of the insertion time partitions. It turns out that the following invariant always holds.

- At any time, there are $\frac{\sqrt{n}(\sqrt{n}+1)}{2}$ sub-indices with size $1 + 2 + \ldots + \frac{p}{\sqrt{n}} = \frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$ and the remaining $n - \frac{\sqrt{n}(\sqrt{n}+1)}{2}$ sub-indices have size $1 + 2 + \ldots + \frac{p}{\sqrt{n}} - 1 = \frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$.

Calculating the Variance of Sub-Index Sizes

Having derived the sequences of possible sub-index sizes, we can now calculate their average variance. Recall that the entire index is expected to have size $\frac{p(p+1)}{2}$. Thus, the mean sub-index size is $\mu = \frac{p(p+1)}{2n}$. Let var_{RR} and var_{Chr} be the average variance of the sub-index sizes for round-robin and chronological partitioning, respectively. That is, we sum up the variance of each possible set of sub-index sizes, and take the average. We begin with var_{RR}, which is simple to calculate due to the above invariant. We have

$$var_{RR} = \frac{\sqrt{n}(\sqrt{n}+1)}{2}\left[\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2} - \mu\right]^2 + (n - \frac{\sqrt{n}(\sqrt{n}+1)}{2})\left[\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2} - \mu\right]^2.$$

After some algebraic simplification, we get

$$var_{RR} = \frac{p^2}{4}(1 - \frac{1}{n}).$$

To obtain var_{Chr}, we add up the variance of the $\sqrt{n}$ sub-indices whose size does not change, and average the possible sizes of the remaining sub-indices. We get

$$var_{Chr} = \sqrt{n}\left[\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2} - \mu\right]^2 + \frac{\sqrt{n}}{p}\left[\left(\frac{n-\sqrt{n}}{2}\right)\sum_{i=0}^{\frac{p}{\sqrt{n}}-1}\left((\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}} - i) - \mu)^2 + (\frac{ip}{\sqrt{n}} - \mu)^2\right)\right].$$

The summation iterates over all the possible sizes of the sub-indices whose sizes change. After expansion of the sum and algebraic simplification, we get

$$var_{Chr} = \frac{p^2}{12\sqrt{n}} - \frac{p^2}{4n} + \frac{p^4}{12n} - \frac{p^4}{12n\sqrt{n}} + \frac{p^2}{6}.$$

Proving the Theorem

We are now ready to prove Theorem 1. Let $d(n,p) = var_{Chr} - var_{RR}$. By substituting the above expressions and by algebraic simplification, we get

$$d(n,p) = \frac{p^2}{12\sqrt{n}} + \frac{p^4}{12n} - \frac{p^4}{12n\sqrt{n}} - \frac{p^2}{12}.$$

By taking the first derivative of d with respect to p and setting $\frac{\partial d(n,p)}{\partial p} = 0$, we find that $d(n,p)$ is monotonically increasing for all values of p such that $p > \frac{\sqrt{2}}{2}\sqrt{n}$. Since we assumed that, $p \geq 2\sqrt{n}$ (each sub-index must span at least two refresh intervals), the smallest value that $d(n,p)$ can take is $d(n, 2\sqrt{n}) = n - \sqrt{n}$. Since $n - \sqrt{n}$ is always positive for $n \geq 4$ (a two-level index has at least four sub-indices), we conclude that the difference in the average sub-index variance of chronological partitioning versus round-robin partitioning is always positive. This completes the proof. □

Appendix B

This section contains algorithms for the proposed variants of doubly partitioned indices. For convenience, we repeat the definitions of all the variables that are used in the algorithms: n is the number of sub-indices, G is the number of partitions of generation times, E is the number of partitions of expiration times, r is the time interval between two consecutive updates, p is the number of times that a window is updated until it rolls over completely, and S is the size of the shortest window in units of time (for queries over count-based windows, S is the estimated size of the shortest window, that is, the minimum over all the windows of the number of tuples divided by the steady-state stream arrival rate). Additionally, in the context of queries over count-based windows, we define R to be the range of expiration times that is guaranteed to contain all the expired tuples (recall Section 4.2).

All algorithms contain two stages: the initial stage and the periodic update stage. Suppose that we want to materialize a (sub)result of some query at some time τ. We assume that all the windows referenced by the query are full at time τ. In the initial stage, we make a partitioning of the generation and expiration times, and insert the initial results of the query into the appropriate sub-indices. In the update stage, we periodically access some of the sub-indices to adjust the insertion and expiration times that they span, and to insert or delete tuples as appropriate (using the *ts* and *exp* values of the result tuples). A detailed implementation of insertions and deletions is not included in the algorithms because that depends on the clustering technique (e.g., *Rebuild* versus *NoRebuild*). The chronological doubly partitioned index is shown in Algorithm 1, *Rebuild* and *NoRebuild* are shown in Algorithm 2, *LazyRebuild* and *LazyNoRebuild* are shown in Algorithm 3, and *CountRebuild* and *CountNoRebuild* are shown in Algorithm 4.

Algorithm 1 Chronological doubly partitioned index

```
Initial stage at time $\tau$
1  for $j = 0$ to $G - 1$
2      $I_{jE+1}$ through $I_{(j+1)E}$ are assigned insertion time range of $\tau - S + j\frac{S}{G} + 1$
       to $\tau - S + (j+1)\frac{S}{G}$
3  for $j = 0$ to $E - 1$
4      $I_{j+1}, I_{E+j+1}, \ldots, I_{(G-1)E+j+1}$ are assigned expiration time range of $\tau + j\frac{S}{E} + 1$
       to $\tau + (j+1)\frac{S}{E}$
5  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
```

Algorithm 2 Round-robin doubly partitioned index (*Rebuild* or *NoRebuild*)

```
Initial stage at time $\tau$
1 for $i = 0$ to $\frac{p}{G} - 1$
2     for $j = 0$ to $G - 1$
3         $I_{jE+1}$ through $I_{(j+1)E}$ are assigned insertion time range of $\tau - S + (iG + j)r + 1$
          to $\tau - S + (iG + j + 1)r$
4 for $i = 0$ to $\frac{p}{E} - 1$
5     for $j = 0$ to $E - 1$
6         $I_{j+1}, I_{E+j+1}, \ldots, I_{(G-1)E+j+1}$ are assigned 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
```

Algorithm 3 *LazyRebuild* or *LazyNoRebuild*

```
Initial stage at time $\tau$
1  for $j = 1$ to $(p-1) \bmod (n-1)$
2      $I_j$ is assigned insertion time range of $\tau - S + (j-1)\lceil \frac{p-1}{n-1} \rceil r + 1$ to $\tau - S + j\lceil \frac{p-1}{n-1} \rceil r$
3  for $j = (p-1) \bmod (n-1) + 1$ to $n-1$
4      $I_j$ is assigned insertion time range of $\tau - S + (j-1)\lfloor \frac{p-1}{n-1} \rfloor r + 1$ to $\tau - S + j\lfloor \frac{p-1}{n-1} \rfloor r$
5  $I_n$ is assigned insertion time range of $\tau - r + 1$ to $\tau$.
6  for $i = 0$ to $\frac{p}{E} - 1$
7      for $j = 0$ to $E - 1$
8          $I_{j+1}, I_{E+j+1}, \ldots, I_{(G-1)E+j+1}$ are assigned expiration time range of
           $\tau + S + (iG + j)r + 1$ to $\tau + S + (iG + j + 1)r$
9  Insert initial result tuples to appropriate sub-indices

Periodic update stage at time $\tau + jr$, $j = 1, 2, \ldots$
 1  Let $J$ be the set of all sub-indices whose expiration time range does not overlap with
    $\tau + (j-1)r$ to $\tau + jr$
 2  Let $K$ be the set of all sub-indices whose insertion time range includes
    $\tau + (j-1)r - S$ to $\tau + jr - S$
 3  Let $L$ be the set of all sub-indices whose insertion time range includes
    $\tau + (j-2)r + 1$ to $\tau + (j-1)r$
 4  if $J$ is empty
 5      for each sub-index $I_v$ with expiration time range of $\tau + (j-1)r + 1$ to $\tau + jr$
 6          Replace above range with $\tau + (j-1)r + 1 + S$ to $\tau + jr + S$
 7          if $I_v$ is not in $K$
 8              Delete tuples with expiration times of $\tau + (j-1)r + 1$ to $\tau + jr$
 9      for each sub-index in $L$
10          Assign insertion time range of $\tau + (j-1)r + 1$ to $\tau + jr$
11          Insert new result tuples to appropriate sub-indices
12  else
13      for each sub-index $I_v$ with expiration time range of $\tau + (j-1)r + 1$ to $\tau + jr$
14          Replace above range with $\tau + (j-1)r + 1 + S$ to $\tau + jr + S$
15          if $I_v$ is not in $L$
16              Delete tuples with expiration times of $\tau + (j-1)r + 1$ to $\tau + jr$
17      for each sub-index in $L$
18          Remove all insertion time ranges and assign range $\tau + (j-1)r + 1$ to $\tau + jr$
19          Remove all results
20          Insert new result tuples to appropriate sub-indices in $L$
```

Algorithm 4 Doubly partitioned index for queries over count-based windows (*CountRebuild* or *CountNoRebuild*)

```
Initial stage at time $\tau$
1 for $i = 0$ to $\frac{p}{G} - 1$
2     for $j = 0$ to $G - 1$
3         $I_{jE+1}$ through $I_{(j+1)E}$ are assigned insertion time range of $\tau - S + (iG + j)r + 1$
          to $\tau - S + (iG + j + 1)r$
4 for $j = 0$ to $E - 1$
5     $I_{j+1}, I_{E+j+1}, \ldots, I_{(G-1)E+j+1}$ are assigned expiration time range of $\tau + j\frac{S}{E} + 1$
      to $\tau + (j + 1)\frac{S}{E}$
6 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 $\tau + (j - 1)r + 1$ to $\tau + jr$
2     Replace above range with $\tau + (j - 1)r + 1 + S$ to $\tau + jr + S$
3 for each sub-index with expiration time range containing range $R$
4     Examine all tuples for expiration and delete tuples that have expired
5 for each sub-index with insertion time range of $\tau + (j - 1)r + 1 - S$ to $\tau + jr - S$
6     Replace above range with $\tau + (j - 1)r + 1$ to $\tau + jr$
7     Insert new result tuples to appropriate sub-indices
```