On Indexing Sliding Windows over On-line Data Streams*

Lukasz Golab
School of Comp. Sci.
University of Waterloo
lgolab@uwaterloo.ca

Shaveen Garg
Dept. of Comp. Sci. and Eng.
IIT Bombay
shaveen@cse.iitb.ac.in

M. Tamer Özsu
School of Comp. Sci.
University of Waterloo
tozsu@uwaterloo.ca

University of Waterloo Technical Report CS-2003-29
September 2003

*This research is partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Abstract

We consider indexing sliding windows in main memory over on-line data streams. Our proposed data structures and query semantics are based on a division of the sliding window into sub-windows. When a new sub-window fills up with newly arrived tuples, the oldest sub-window is evicted, indices are refreshed, and continuous queries are re-evaluated to reflect the new state of the window. By classifying relational operators according to their method of execution in the windowed scenario, we show that many useful operators require access to the entire window, motivating the need for two types of indices: those which provide a list of attribute values and their counts for answering set-valued queries, and those which provide direct access to tuples for answering attribute-valued queries. For the former, we evaluate the performance of linked lists, search trees, and hash tables as indexing structures, showing that the high costs of maintaining such structures over rapidly changing data are offset by the savings in query processing costs. For the latter, we propose novel ways of maintaining windowed ring indices, which we show to be much faster than conventional ring indices and more efficient than executing windowed queries without an index.

1 Introduction

Many current and emerging data management applications, among them network traffic measurement, sensor networks, and transaction log analysis, are expected to process long-running queries (known in the literature as *continuous queries* [4, 19]) in real time over high-volume data streams. In order to emphasize recent data and to avoid storing potentially infinite streams in memory, data stream management systems may require some continuous queries to operate over sliding windows, e.g. Aurora [1] and TelegraphCQ [3]. In the sliding window model, the system stores only the N most recent items (count-based windows) or only those items whose timestamps are at most as old as the current time minus T (time-based windows). For example, an Internet traffic monitoring system may calculate the average Round Trip Time (RTT) over a sliding window to determine an appropriate value for the TCP timeout. A sliding window RTT average is appropriate because it emphasizes recent measurements and acts to smooth out the effects of any sudden changes in network conditions.

Continuous queries over sliding windows are re-evaluated periodically, yet the input streams arrive continuously, possibly at a high rate. Hence, this environment, in which insertions and deletions caused by high-speed data streams heavily outweigh query invocations, is a complete opposite of the situation in traditional DBMSs where queries are more frequent than updates. In light of this drastic workload change, we pose the following questions in this paper. Given the usefulness of indices in traditional databases, is it also beneficial to index sliding windows over on-line data streams or will the cost of maintaining indices over volatile data negate their advantages? Further, how can we exploit the update patterns of sliding windows to design more efficient indices?

1.1 System Model and Assumptions

We define a data stream to be a sequence of relational tuples with a fixed schema. Each tuple has a timestamp that may either be implicit (generated by the system at arrival time) or explicit (inserted by the source at creation time), and is used to determine the tuple's position in the stream. However, we do not include the timestamp attribute in the schema of the stream. Following Zhu and Shasha [20], we divide the sliding window into n sub-windows, called *Basic Windows*. To ensure constant sliding window size, each basic window should either store the same number of tuples (count-based windows) or span an equal time interval (time-based windows). Inside each

sub-window, we may store individual tuples, some aggregate information about attribute values, or both. When the newest basic window fills up, it is appended to the sliding window, the oldest basic window is evicted, and the continuous query may be re-evaluated. This allows for inexpensive window maintenance as we need not scan the entire window to check for expired tuples, but induces a "jumping window" rather than a gradually sliding window. Therefore, the maximum basic window size is limited by the latency requirements of a particular query or application. Finally, we assume that sliding windows are stored in main memory to accept fast stream arrival rates and ensure timely processing of continuous queries.

1.2 Contributions

Our contributions in this paper are as follows.

- Using the basic window model as a window maintenance technique and as a basis for continuous query semantics, we propose main-memory based storage methods for sliding windows.
- We classify relational operators according to their evaluation techniques over sliding windows, and show that two types of indices are potentially useful for speeding up windowed queries: set-valued and attribute-valued indices.
- We propose and experimentally assess the performance of indexing techniques for answering set-valued windowed queries as well as novel techniques for maintaining a windowed ring index that supports attribute-valued queries.

To the best of our knowledge, this paper is the first to propose storage structures and indexing techniques specifically for main-memory based sliding windows.

1.3 Roadmap

In the remainder of the paper, Section 2 reviews previous work, Section 3 outlines physical storage methods for sliding windows, Section 4 classifies relational operators in the windowed scenario and motivates the need for indices, Section 5 proposes indices for set-valued queries, Section 6 discusses ring indices for attribute-valued queries, Section 7 presents experimental results regarding index performance, and Section 8 concludes the paper with suggestions for future work.

2 Related Work

Broadly related to our research is the recent work on data stream management systems; see Golab and Özsu [10] for a survey. Sliding window algorithms are particularly relevant as many queries are easy to compute over an infinite stream, but difficult when constrained to a sliding window. For instance, computing the maximum value in an infinite stream requires $O(1)$ time and memory, but doing so in a sliding window with N tuples requires $\Omega(N)$ space and time. The main issue is that as new items arrive, old items must be simultaneously evicted from the window and their contribution discarded from the answer. The basic window technique [20] may be used to decrease memory usage and query processing time by storing summary information rather than individual tuples in each basic window. For example, storing only the maximum value for each basic window allows us to compute the windowed maximum by scanning the basic window summaries and choosing the overall maximum. This method has recently been extended to detecting bursts of similar tuples by means of multiple layers of overlapping basic windows [21].

In the basic window approach, results are refreshed after the newest basic window fills up. Datar et al. [7] propose *Exponential Histograms* (EH) to bound the error caused by over-counting those elements in the oldest basic window which should have expired. Their algorithm provides an approximate answer at all times using poly-logarithmic space in the size of the sliding window. The EH algorithm, initially proposed for counting the number of ones in a binary stream, has recently been extended to maintaining a histogram by Qiao et al. [15], and to time-based windows by Cohen and Strauss [5].

Recent work on sliding window joins includes Kang et al. [13], who give join algorithms for count-based windows and show that NLJs are usually the slowest, except when one or both streams are very fast, in which case the cost of maintaining indices is too high. In previous work, we addressed the issue of joining more than two streams in the context of time-based and count-based windows [11]. We showed that index performance highly depends on the frequency of tuple expiration and that hash indices are more efficient in a basic window-like approach of batch insertions and deletions. Heuristics for semantic load shedding to maximize the result size of windowed joins are given by Das et al. [6].

Our work is also related to research in main memory query processing, e.g. [8, 14], but these works test a traditional workload of mostly searches and occasional updates. In particular, our research uses the domain storage model and ring indexing, which recently also drew attention in small-footprint databases, e.g. [2]. Research in bulk insertion and deletion in indices is also of interest, e.g. [9, 16, 18], though not directly applicable because the previous works assume a disk-resident database.

Finally, the problem of indexing sliding windows, albeit stored on disk and updated off-line, has been tackled by Shivakumar and Garcia-Molina [17]. The main idea in their algorithms, called *Wave Indices*, is to split the index into several parts so that deletions and insertions do not affect the entire index. Maintaining clustered order on disk as well as temporarily storing parts of the index in main memory are also discussed. We will make further comparisons between our work and Wave Indices later on in this paper.

3 Physical Storage Methods for Sliding Windows

In this section, we propose a general data structure for sliding windows using the basic window model. As in traditional databases, our goal is to specify a default storage method and access plan, and allow for the possibility of building one or more indices over the window.

3.1 General Storage Structure for Sliding Windows

As a general sliding window storage structure, we propose a circular array of pointers to basic windows (the number of basic windows is fixed to keep the window size constant). We treat the issue of implementing individual basic windows in an orthogonal way, as long as the contents of a basic window can be accessed by following its pointer. When the newest basic window fills up, its pointer is inserted in the circular array by overwriting the pointer to the oldest basic window. In order to guarantee constant window size, a new basic window should be stored in a separate buffer as it is filling up so that queries do not have access to these new tuples until the oldest basic window has been replaced. Furthermore, since the window slides forward in units of one basic window, we propose to remove timestamps from individual tuples and only store one timestamp per basic window, say the timestamp of its oldest tuple. With this approach, some accuracy is lost (e.g. if we perform a window join, we introduce additional error by possibly joining tuples which

should have expired; however, this error is bounded by the basic window size), but space usage is decreased. The data structure is illustrated in Figure 1.

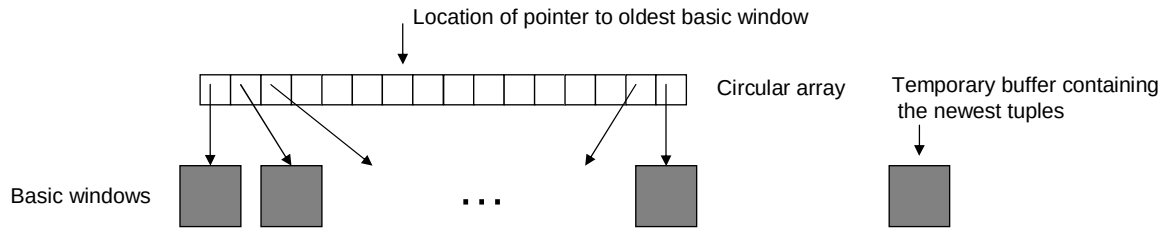


Figure 1: Diagram of our sliding window data structure containing a circular array of pointers to basic windows.

3.2 Storage Structures for Individual Basic Windows

The simplest storage method for individual basic windows is a linked list of tuples, abbreviated as LIST[1]. Again, the way in which we implement tuple storage is orthogonal to the basic window data structure: a tuple could consist of a list of attribute values or a list of pointers to an index that stores the values (the latter is known as the *domain storage model* [2]). Moreover, the tuples inside a basic window could be linked in chronological order (newest tuple at the tail) or in reverse chronological order (newest tuple at the head).

If basic windows contain multiple tuples with the same attribute value, we can save space by aggregating out one or more attributes, leading to three additional storage methods. In AGGR, each basic window consists of a sorted linked list of frequency counts for every distinct value appearing in this basic window. If tuples consist of more than one attribute, a separate AGGR structure is needed for each attribute. Alternatively, we may retain a LIST structure to store attributes which we do not want to aggregate out and use AGGR structures for the remaining attributes. Since inserting tuples in an AGGR structure may be expensive, it is more efficient to use a hash table rather than a sorted linked list. That is, (every attribute that we want to aggregate out in) each basic window could be a hash table, with each bucket storing a sorted linked list of counts for those attribute values which hash to this bucket. We call this technique HASH. Finally, further space reduction may be achieved by building an AGGR structure, but only storing counts for groups of values (e.g. value ranges) rather than distinct values. We call this structure GROUP. Figure 2 illustrates the four basic window implementations, assuming that each tuple has one attribute and that tuples with the following values have arrived in the basic window: $< 12, 14, 16, 17, 15, 4, 19, 17, 16, 23, 12, 19, 1, 12, 5, 23 >$.

3.3 Window Maintenance and Query Processing

Let d be the number of distinct values and b be the number of tuples in a basic window, let g be the number of groups in GROUP, and let h be the number of buckets in HASH. Assume for simplicity that b and d do not vary across basic windows. The worst-case, per-tuple cost of inserting items in the window is $O(1)$ for LIST, $O(d)$ for AGGR, $O(\frac{d}{h})$ for HASH, and $O(g)$ for GROUP. The space

[1]In count-based windows where basic windows always store the same number of tuples, we could have used another circular array within each basic window. However, to make the structure applicable to time-based windows, we have to account for the fact that we do not know how many tuples a particular basic window will have when full.

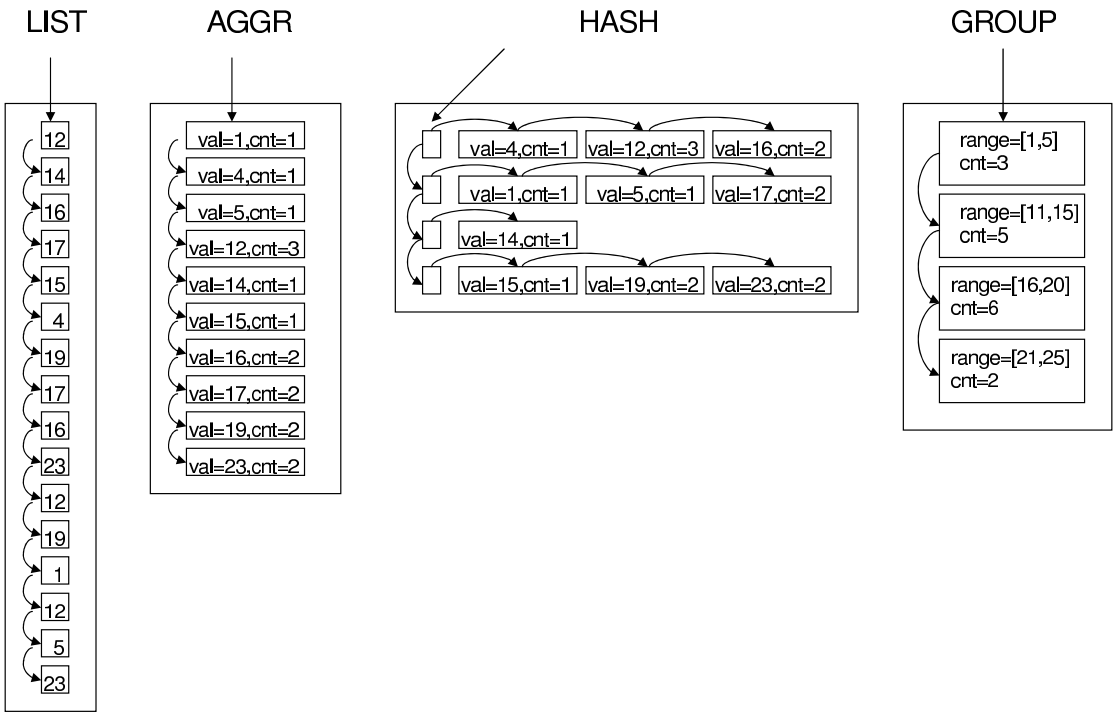


Figure 2: Our four basic window data structures, assuming that attribute values are stored inside tuples (rather than in an index to which the tuples point, as in the domain storage model). The hash function in HASH is modular division of the attribute value by the number of buckets (four).

requirements are $O(b)$ for LIST, $O(d)$ for AGGR, $O(d+h)$ for HASH, and $O(g)$ for GROUP. Hence, AGGR and HASH save space over LIST, but are more expensive to maintain, while GROUP is efficient in both space and time at the expense of lost accuracy. In general, window maintenance with the basic window approach is faster than advancing the window upon arrival of each new tuple (in the latter, there is the additional cost of scanning the window upon every tuple arrival to check for expired tuples). The downside is the presence of a delay between the arrival of a new tuple and its inclusion in the query result. Let t be the time when the newest basic window has filled up and is about to be attached to the sliding window, let t_b be the time span of a basic window, and let d be the maximum delay that a tuple may have incurred on its way to the system. At time $t+d$, all the tuples that belong in the newest basic window have arrived, at which time the sliding window may be advanced and the continuous query re-evaluated. Thus, we guarantee that the contribution of any new tuple to the answer will be reflected in the query result with a delay of at least t_b+d and at most $2t_b+d$—we must re-evaluate the query before the next basic window fills up in order to keep up with the stream. If, however, we are falling behind, we may reduce the query re-evaluation frequency. In the background, we continue to advance the window after the newest basic window fills up, but we re-execute the query every two, three, or k basic windows. In fact, we may split the query workload by evaluating easy (or important) queries often and others more rarely. We may also adjust the query re-evaluation frequency dynamically in response to changing system conditions.

4 Classification of Sliding Window Operators

In order to motivate the need for indexing sliding windows, we now present a classification of windowed relational operators, which will reveal that many interesting operators require access to the entire window during query re-execution. We will also show that two types of indices may be useful: those which cover set-valued queries such as intersection, and those which cover attribute-valued queries such as joins.

4.1 Input and Output Modes

We define two input types and two output types—windowed and non-windowed—which give rise to four operator evaluation modes:

1. In the non-windowed input and non-windowed output mode, we process each tuple as it arrives and immediately return it in the output stream if it satisfies the operator, e.g. selection.

2. In the windowed input and non-windowed output mode, we store a window of tuples for each input stream and return new results as new items arrive, e.g. sliding window join.

3. In the non-windowed input and windowed output mode, we consume tuples as they arrive and materialize the output as a sliding window. All the operators from Mode 1 apply here, e.g. we may want to store a sliding window of the result of a selection predicate.

4. In the windowed input and windowed output mode, we proceed as in Mode 2 except that rather than streaming new results to the user, we materialize the output as a sliding window. For example, we may evaluate a windowed join and store the result as a window. This entails producing new results whenever a new tuple arrives, storing these results in the view, and expiring stale tuples from the view and from the base windows. Another example is the windowed sort, which only makes sense in this mode.

Only Mode 1 does not require the use of a window, either for the inputs or for materialized views. In all other cases, the basic window model and our storage techniques are directly applicable. Moreover, Modes 2 and 4 may require one of two index types on the input windows, as will be explained next.

4.2 Incremental Evaluation

An orthogonal classification considers whether or not an operator may be incrementally updated in the basic window model without accessing the entire window. Mode 1 and 3 operators are incremental as they do not require a window on the input. As for operators working in Modes 2 and 4, we distinguish three groups: incremental operators, non-incremental operators that become incremental if a histogram of the window is available, and non-incremental operators. The first group includes aggregates computable by dividing the data into partitions and storing a partial aggregate for each partition (basic window). For example, we may compute the sum of all the items in the window by storing one cumulative sum and partial sums for each basic window; upon re-evaluation, we subtract the sum of the items in the oldest basic window and add the sum of the items in the newest basic window. The second group contains some non-distributive aggregates (e.g. median) and set expressions. For example, we can incrementally compute a set intersection of two windows by storing a histogram of attribute value frequencies: we subtract frequency counts of items in the oldest basic window, add frequency counts of items in the newest basic window, and

re-compute the intersection by scanning the histogram and returning values with non-zero counts in both windows. Without such an index, we could use a hashing or a sort-merge algorithm that would need to scan the entire window. Finally, the third group includes operators such as join and some non-distributive aggregates such as variance, where the entire window has to be probed to update the answer. Thus, many windowed operators require one of two types of indices: a histogram-like summary index that stores attribute values and their multiplicities, or a full index that enables fast retrieval of tuples in the window. We devote the remainder of this paper to a study of both of these index types in the context of sliding windows.

5 Indexing for Set-valued Queries

We begin our discussion of sliding window indices by describing five main-memory indexing structures for set-valued queries on one attribute. From the previous section, this corresponds to Mode 2 and 4 operators that are not incremental without an index, but become incremental when a histogram is available. The indices consist of a set of attribute values and their frequency counts, and all but one make use of the domain storage model. In this model, attribute values are not stored inside tuples, but in an external data structure to which tuples point. When a tuple is retrieved from memory, its pointer(s) must be followed to look up the attribute value(s).

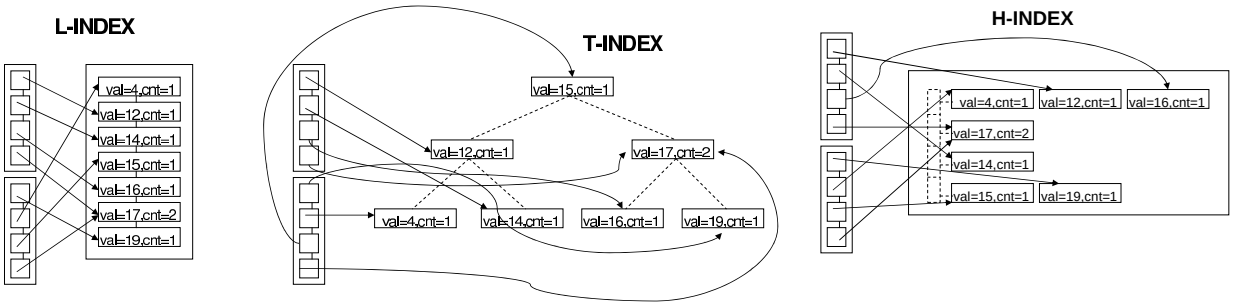
5.1 Domain Storage Index as a List (L-INDEX)

The L-INDEX consists of a linked list of attribute values and their frequencies, sorted by value. It is compatible with LIST, AGGR, and HASH as the underlying basic window implementations. When using LIST and the domain storage model, each tuple has a pointer to the entry in the L-INDEX that corresponds to the tuple's attribute value—this configuration is shown in Figure 3 (a). When using AGGR or HASH, each distinct value in every basic window has a pointer to the corresponding entry in the index. There are no pointers directed from the index back to the tuples because set-valued queries only need to know which attribute values are present in the window (we will deal with attribute-valued indices, which need pointers from the index to the tuples, in the next section).

Insertion in the L-INDEX proceeds as follows. As the newest basic window fills up, for each tuple arrival (LIST) or each new distinct value (AGGR and HASH), we scan the index and insert a pointer from the tuple (or AGGR node) to the appropriate index node. This can be done eagerly as tuples arrive, or lazily after the newest basic window fills up. The cost of each is equal, so it is better to do so eagerly and spread out the computation. However, we may not increment counts in the index until the newest basic window is ready to be attached, so we must perform a final scan of the basic window when it has filled up, following pointers to the index and incrementing counts as appropriate. Deletion from the L-INDEX is simple: there exists a pointer from each tuple (or each distinct value) to the appropriate value in the index. Therefore, as an old basic window is being removed, we scan it, follow pointers to the index, and decrement the counts. There remains the issue of deleting an attribute value from the index when its count reaches zero, but we discuss it separately in Section 5.5.

5.2 Domain Storage Index as a Tree (T-INDEX)

Insertion in the L-INDEX is expensive: the entire index may need to be traversed before a particular value is found. This can be improved by implementing the index as a search tree, whose nodes store values and their counts. An example using an AVL tree [12] is shown in Figure 3 (b). Another



(a) (b) (c)

Figure 3: Illustration of the (a) L-INDEX, (b) T-INDEX, and (c) H-INDEX, assuming that the sliding window consists of two basic windows with four tuples each. The following attribute values are present in the basic windows: $< 12, 14, 16, 17 >$ and $< 19, 4, 15, 17 >$.

benefit of a T-INDEX is that lookups are also cheaper, as long as we use a self-balancing tree (we will assume so whenever referring to the T-INDEX in the rest of the paper). The procedure for maintaining the T-INDEX is the same as for the L-INDEX.

5.3 Domain Storage Index as a Hash Table (H-INDEX and UH-INDEX)

Another alternative is to use a hash table with buckets containing linked lists of counts of those attribute values which hash to this bucket. Insertion and deletion proceed as in the L-INDEX. Buckets can be sorted by value (H-INDEX; see Figure 3 (c) for an example) or unsorted (UH-INDEX). The motivation for the latter is that old items are removed from the index upon expiration, so it may be better to always insert newly observed attribute values at the head of each bucket. Based on the H-INDEX example in Figure 3 (c), an equivalent UH-INDEX would have the value 12 first in the first bucket, followed by 16 and 4, and the value 19 first in the last bucket. This would be beneficial if tuples with the same attribute values arrive (at least somewhat) contiguously in one batch and do not repeat outside of this batch. We will experimentally compare the performance of the H-INDEX and the UH-INDEX later on in the paper.

5.4 Grouped Index (G-INDEX)

The G-INDEX is an L-INDEX that stores frequency counts for groups of values rather than for each distinct value. It is compatible with LIST and GROUP, and may be used with or without the domain storage model. Using domain storage, each tuple (LIST) or group of values (GROUP) contains a pointer to the node in the G-INDEX that stores the label for the group. This is space efficient because only one copy of the label is stored. In this case, index maintenance is identical to the L-INDEX. However, when using GROUP with a small number of groups, we may choose not to use the domain storage model and instead store the group labels inside each GROUP structure for each basic window. The space usage is admittedly higher, but the advantage is a simpler maintenance procedure as follows. When the newest basic window fills up, we merge the counts in its GROUP structure with the counts in the oldest basic window's GROUP structure, in effect creating a sorted delta-file that contains all the changes that need to be applied to the index. We then merge this delta-file with the G-INDEX, adding or subtracting counts as appropriate. In the remainder of the paper, we will assume that domain storage is not used when referring to the

G-INDEX.

5.5 Purging Unused Attribute Values

Each of the indices defined in this section stores frequency counts for attribute values or groups thereof. If a new value appears, a new node must be inserted in the index. Similarly, when a count reaches zero, its node should be deleted. However, since we are dealing with volatile data, it may not be wise to delete zero-count nodes immediately. This is especially important if we use a self-balancing tree as the indexing structure, because delayed deletions should decrease the number of required re-balancing operations. The simplest solution is either to clean up the index every n tuples, in which case every nth tuple to be inserted invokes an index scan and removal of zero counts, or to do so periodically. Another alternative is to maintain a data structure that points to nodes which have zero counts, thereby avoiding an index scan. In any case, there is a trade-off in that if we clean up too rarely, indices will become large and scanning them will take more time when inserting tuples and processing queries.

5.6 Analytical Comparison

We will use the following variables: d is the number of distinct values in a basic window, g is the number of groups in GROUP and in G-INDEX, h is the number of hash buckets in HASH, H-INDEX and UH-INDEX, b is the number of tuples in a basic window, N is the number of tuples in the window, and D is the number of distinct values in the window. To simplify the analysis, we assume that d and b are equal across basic windows and that N and D are equal across all instances of the sliding window. In terms of space usage, G-INDEX is expected to be cheapest, especially if the number of groups is small, followed by the L-INDEX. The T-INDEX requires additional parent-child pointers while the H-INDEX and the UH-INDEX require a hash directory. As for the per-tuple time complexity, we may divide it into four steps for all basic window implementations except GROUP, which will be discussed separately:

1. The cost of maintaining the underlying sliding window, as derived in Section 3.3.

2. The cost of creating pointers from newly arrived tuples to the index. This cost depends on two things: how many times we scan the index and how expensive each scan is. For the former, the cost is 1 for LIST, and $\frac{d}{b}$ for AGGR and HASH (in AGGR and HASH, we only make one access into the index for each distinct value present in the basic window). For the latter, the cost is D for the L-INDEX, $\log D$ for the T-INDEX, and $\frac{D}{h}$ for the H-INDEX and the UH-INDEX.

3. The cost of scanning the newest basic window when it has filled up, following pointers to the index, and incrementing the counts in the index. This is 1 for LIST, and $\frac{d}{b}$ for AGGR and HASH.

4. The cost of scanning the oldest basic window when it is ready to be expired, following pointers to the index, and decrementing the counts. This costs the same as in Step 3. We ignore the cost of purging value-count pairs with zero counts from the index since this can be done periodically, as discussed above.

The cost of the G-INDEX with basic windows implemented as GROUPs consists of the cost of insertion in the GROUP structure at g (per tuple), and the cost of creating a delta-file and merging it with the index, which is g per basic window, or $\frac{g}{b}$ per tuple (recall that this is the

merging approach where the domain storage model is not used). Table 1 summarizes the worst-case per-tuple time complexity of maintaining each type of index (corresponding to each column) with each type of basic window implementation (corresponding to each row), except the G-INDEX. In general, the G-INDEX is expected to be the fastest, while the T-INDEX, the H-INDEX, and the UH-INDEX are expected to outperform the L-INDEX if basic windows contain multiple tuples with the same attribute values.

	L-INDEX	T-INDEX	H-INDEX
LIST	$O(D)$	$O(\log D)$	$O(\frac{D}{h})$
AGGR	$O(d + \frac{d}{b}D)$	$O(d + \frac{d}{b}\log D)$	$O(d + \frac{d}{b}\frac{D}{h})$
HASH	$O(\frac{d}{h} + \frac{d}{b}D)$	$O(\frac{d}{h} + \frac{d}{b}\log D)$	$O(\frac{d}{h} + \frac{d}{b}\frac{D}{h})$

Table 1: Per-tuple cost of maintaining each type of index using each type of basic window implementation.

At this point, we compare our results with Wave Indices [17]. The main idea that is applicable to our work is splitting a windowed index into sub-indices so that insertion of a batch of tuples is cheaper (only one part of the index needs to be accessed). This technique is orthogonal to our work and may be used in conjunction with any of our indices. The net result is a decrease in processing time, but a corresponding increase in memory usage as an index node corresponding to a particular attribute value may now appear in each of the sub-indices. Wave Indices are also helpful if we want to store the sliding window on disk. In this case, our circular array may still be used (and stored in main memory), while the basic windows may be stored on disk and divided into partitions, as outlined in [17].

6 Indexing for Attribute-valued Queries

We now deal with the issue of indexing for attribute-valued queries, which need to access individual tuples. Such queries involve operators which we classified in Section 4 as Mode 2 and 4 operators that are non-incremental. For example, we may execute a join of two windows using an index on the join attribute, followed by some operations on a different attribute of the join result. We need to access individual tuples that possibly have more than one attribute, therefore LIST is the only basic window implementation available as we do not want to aggregate out any attributes. In what follows, we will present several methods of maintaining an attribute-valued index, each of which may be added on to any of the index structures proposed in the previous section.

6.1 Windowed Ring Index

A simple extension of our set-valued indices to handle attribute-valued queries is to add pointers from the index to some of the tuples. This can be accomplished by a ring index [2], which consists of linking together all tuples with the same attribute value; additionally, the first tuple is pointed to by a node in the index that stores the actual attribute value, while the last tuple points back to the index, creating a ring for each attribute value. One such ring is illustrated in Figure 4 (a), where it is built on top of the L-INDEX. The sliding window is on the left, with the oldest basic window at the top and the newest (not yet full) basic window at the bottom. Each basic window is implemented as a LIST and has four tuples, connected by pointers drawn on the left. Shaded boxes indicate tuples that have the same attribute value of five and are connected by ring pointers,

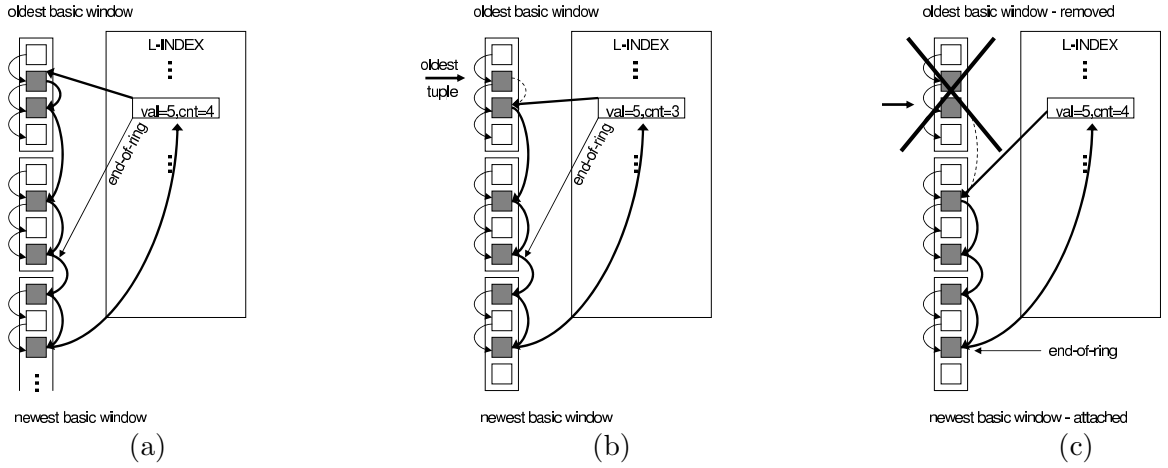


Figure 4: Maintenance of a windowed ring index.

drawn in bold on the right. We may add new tuples to the index as they arrive, but we must ensure that the contents of the newest basic window remain invisible to queries until this window is ready to be appended. This can be done by storing end-of-ring pointers, as seen in Figure 4, which identify the newest tuple in the ring that is currently active in the sliding window.

Let N be the number of tuples and D be the number of distinct values in an instance of a sliding window, let b be the number of basic windows, and let d be the average number of distinct values per basic window. As each tuple arrives and is inserted in the newest basic window, the index is scanned for a cost of D (assuming an underlying L-INDEX) to set up a pointer from the new tuple back to the index. We then find the next newest tuple in the ring and link it with the tuple which has just arrived. This costs $\frac{N}{D}$ in the worst case as we have to traverse the entire ring to find this newest tuple. When the newest basic window fills up, we scan the oldest basic window and remove expired tuples from the rings. However, as shown in Figure 4 (b), deleting the oldest tuple (pointed to by the arrow) entails removing its pointer in the ring (denoted by the dotted arrow) and following the ring all the way back to the index in order to advance the pointer from the index to the start of the ring (drawn in bold and now pointing to the next oldest tuple). Thus, deletion takes time $\frac{N}{D}$ per tuple. Finally, we scan the index and update end-of-ring pointers for a cost of D. Figure 4 (c) shows the completed update with the oldest basic window removed and the end-of-ring pointer moved to its new location. The total maintenance cost per tuple is $D + 2\frac{N}{D} + \frac{D}{b}$, which is quite high.

6.2 Faster Insertion with Auxiliary Index (AUX)

For more efficient insertion, we propose to build a temporary local ring index for the newest basic window as it is filling up, shown in Figure 5 (a). We call this technique the auxiliary index method (AUX). When a tuple arrives with an attribute value that we have not seen before in this basic window, we create a new node in the auxiliary index for a worst-case cost of d. We then link this new node with the appropriate node in the main index for a cost of D. As before, we also connect the previously newest tuple in the ring with the newly arrived tuple for a worst-case cost of $\frac{N}{D}$. However, if another tuple arrives with the same distinct value, we only have to look up this value in the auxiliary index and insert the tuple at the end of the ring. When the newest basic window fills up, advancing the end-of-ring pointers and linking the newest tuples to the main index is cheap as all the pointers are already in place. This can be seen in Figure 5 (b), where dotted lines

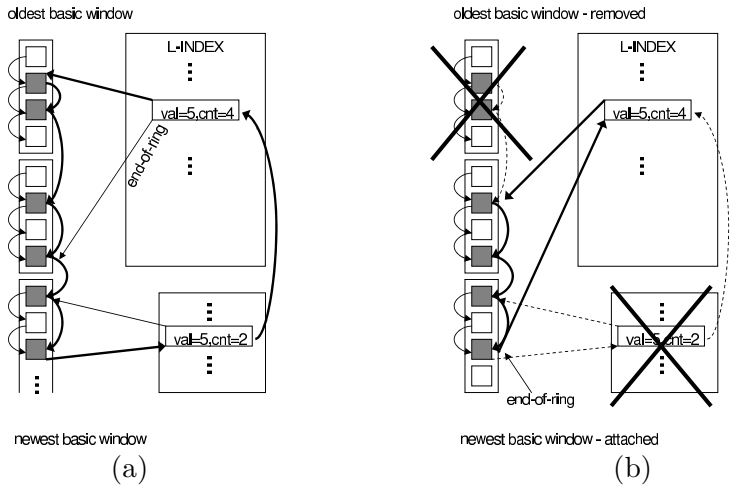


Figure 5: Maintenance of an AUX ring index.

indicate pointers that can be deleted. The temporary index can then be re-used for the next new basic window. The additional storage cost of the auxiliary index is $3d$ because three pointers are needed for every index node, as seen in Figure 5. However, the per-tuple maintenance cost is now lower, at $d + \frac{d}{b}(1 + D + \frac{N}{D})$ for insertion plus $\frac{N}{D}$ for deletion.

6.3 Faster Deletion with Backward-linked Ring Index (BW)

Our next step is to improve the deletion strategy of the AUX method, which traverses the entire ring whenever deleting a tuple. We could shorten the deletion time by storing separate ring indices for each basic window, but this is expensive. Instead, we need a way of deleting all the expired tuples from the same ring at once. However, we cannot simply traverse the ring and delete tuples because we do not store timestamps within tuples, so we would not know when to stop deleting. Fortunately, we can perform bulk deletions without having to store tuple timestamps with the following change to the AUX method: we link tuples in reverse-chronological order in the LISTs and in the rings. This method, to which we refer as the backward-linked ring index (BW), is illustrated in Figure 6 (a). When the newest basic window fills up, we start the deletion process with the youngest tuple in the oldest basic window, as labeled in Figure 6 (b). This is possible because tuples are now linked in reverse chronological order inside a basic window. We then follow the ring (which is also in reverse order) until the end of the basic window and remove the ring pointers. This is repeated for each tuple in the basic window, but some of the tuples will have already been disconnected from their rings. Lastly, we create new pointers from the oldest active tuple in each ring to the index. However, to find these tuples, it is necessary to follow the rings all the way back to the index. Nevertheless, we have decreased the number of times the entire ring must be followed from one per tuple to one per distinct value, without any increase in space usage over the AUX method! The per-tuple maintenance cost of the BW method is $d + \frac{d}{b}(1 + D + \frac{N}{D})$ for insertion (same as AUX), but only $\frac{d}{b}\frac{N}{D}$ for deletion.

6.4 Further Improvement: Backward-linked Ring Index with Dirty Bits (DB)

If we do not traverse the entire ring for each distinct value being deleted, we could reduce deletion costs to $O(1)$ per tuple. The following is a possible solution that requires additional D bits and D tuples of storage, and slightly increases query processing time. We use the BW technique, but

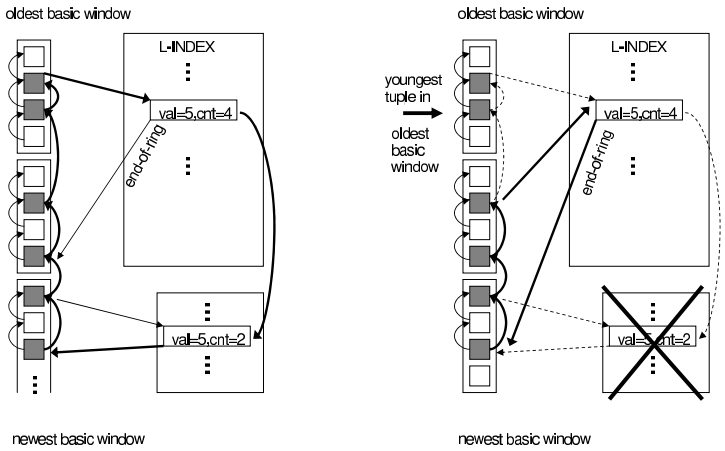


Figure 6: Maintenance of a BW ring index.

for each ring, we do not delete the youngest tuple in the oldest basic window. That is, rather than traversing each ring to find the oldest active tuple, we create a pointer from the youngest inactive tuple (i.e. the youngest tuple in the oldest basic window, as labeled Figure 6 (b)) to the index. Since we would normally delete the entire oldest basic window, we now need additional temporary storage for up to D expired tuples (one from each ring). Using this technique, we could assume that the oldest tuple in each ring is stale and ignore it during query processing. However, this assumption works only for those attribute values which appeared in the oldest basic window. Otherwise, all the tuples in the ring are in fact current. Our full solution, then, is to store "dirty bits" in the index for each distinct value, and set these bits to zero if the last tuple in the ring is current, and to one otherwise. Initially, all the bits are zero. During the deletion process, all the distinct values which appeared in the oldest basic window have their bits set to one. We call this algorithm the backward-linked ring index with dirty bits (DB).

7 Experiments

In this section, we report experimental results concerning the performance of our storage structures and indices. Firstly, we validate our analytical results regarding the maintenance costs of various indices and basic window implementations. Further, we examine whether it is more efficient to maintain windowed indices or to re-execute continuous queries from scratch by accessing the entire window.

7.1 Experimental Setting and Implementation Decisions

We have built a code base consisting of our proposed indices and basic window implementations. We used Sun Microsystems JDK 1.4.1, running on a Windows PC with a 2 GHz Pentium IV processor and one gigabyte of RAM. To test the T-INDEX, we adapted an existing AVL tree implementation from `www.seanet.com/users/arsen/source.html`. For simplicity, the data stream is synthetically generated and consists of tuples with two integer attributes. To generate tuples, we run a continuous for-loop, inside which one tuple per iteration is produced with randomly generated attribute values. Each experiment is repeated by first generating attribute values from a uniform distribution, and then from a power law distribution with the power law coefficient equal

to unity. We set the size of the sliding window to 100000 tuples. In each experiment, we first generate 100000 tuples to fill the window in order to eliminate transient effects occurring when the windows are initially non-full. We then generate an additional 100000 tuples and measure the time taken to process the latter. Note that our tuple generation procedure ignores the variability in tuple interarrival times, so although we are testing time-based windows, each basic window and each instance of the sliding window contain the same number of tuples. We repeat each experiment ten times and report the average processing time.

With respect to periodic purging of unused attribute values from indices, we experimented with values between two and five times per sliding window roll-over. We did not notice a significant difference in index maintenance times, so in our experiments, we set this value to five times per window roll-over (i.e. once every 20000 tuples). In terms of the number of hash buckets in HASH, H-INDEX, and UH-INDEX, we observed the expected result that increasing the number of buckets improves performance. To simplify the experiments, we set the number of buckets in HASH to five (recall that a HASH structure is needed for every basic window, so the number of buckets cannot be too large due to space constraints), and the number of buckets in the H-INDEX and the UH-INDEX to one hundred. All hash functions are modular divisions of the attribute value by the number of buckets. The remaining parameters in our experiments are the number of basic windows (50, 100, and 500) and the number of distinct values in the stream (1000 and 10000). For brevity, we report performance numbers when generating attribute values from a uniform distribution and only mention results of tests with a power law distribution when the outcomes are significantly different.

7.2 Experiments with Set-Valued Queries

7.2.1 Set-Valued Index Maintenance

We first discuss the relative performance of our set-valued indices. Index maintenance costs are graphed in Figure 7 when using LIST and AGGR as basic window implementations (maintenance cost with HASH follows the same pattern as AGGR, except that the times for the former are shorter). As expected, the L-INDEX is by far the slowest and least scalable: in both figures, the bars extend well beyond the range of the graph as the maintenance times exceed 10000 seconds. The T-INDEX and the H-INDEX perform similarly, though the T-INDEX wins more decisively when attribute values are generated from a power law distribution, in which case our simple hash function breaks down, especially for 10000 distinct values. As expected, using AGGR means that performance suffers if there are too few basic windows; in this case, the AGGR structures are long and it takes a long time to insert tuples. This is particularly noticeable as the number of distinct values increases. Comparing the performance of various basic window implementations in Figure 7, LIST is usually more efficient that AGGR. As anticipated, AGGR and HASH only show noticeable improvement when there are multiple tuples with the same attribute values in the same basic window. This happens when the number of distinct values and the number of basic windows are fairly small, especially when values are generated from a power law distribution.

We take a closer look at the performance of the H-INDEX and the UH-INDEX in Figure 8 (a). Each basic window implementation (LIST, AGGR, and HASH) gives rise to six pairs of bars, the first three corresponding to 50, 100, and 500 basic windows with 1000 distinct values in the stream, and the last three to the same with 10000 distinct values. In general, UH-INDEX performs badly for 10000 distinct values, in which case the hash buckets are large and leaving them unsorted causes long insertion times. For 1000 distinct values, the two variants perform very similarly.

The performance advantage of using a G-INDEX (gained by introducing error as the basic

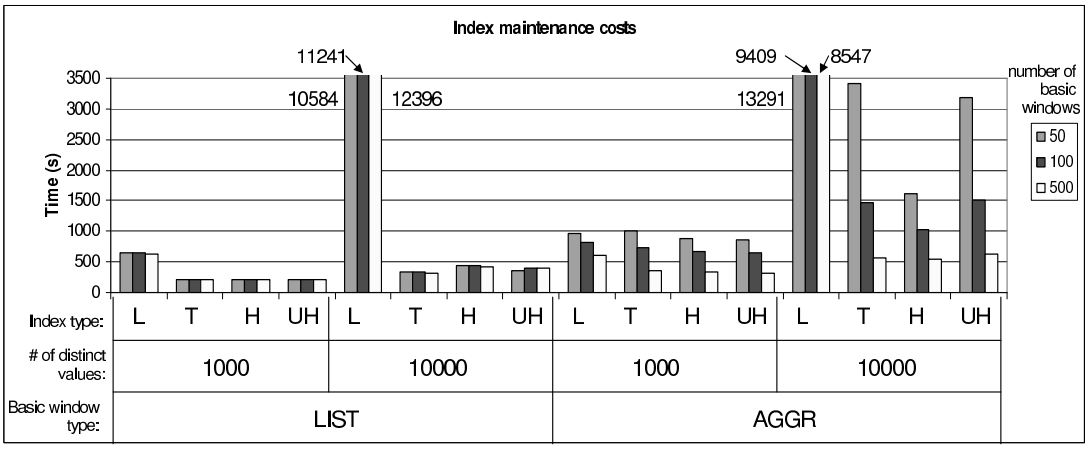


Figure 7: Maintenance costs of the L-INDEX (L), the T-INDEX (T), the H-INDEX (H), and the UH-INDEX (UH).

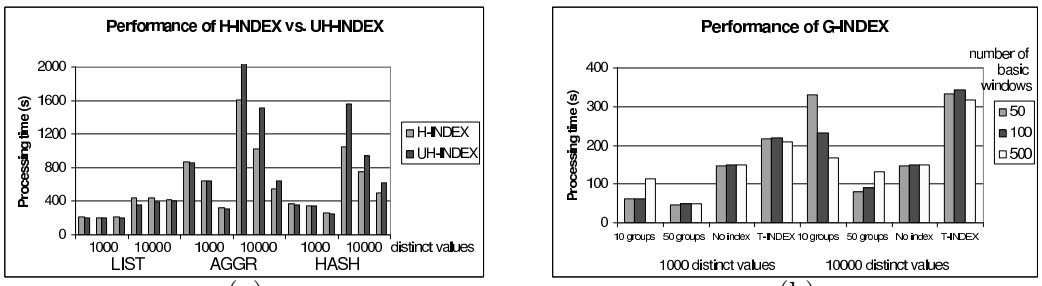


Figure 8: Closer look at (a) the performance of the H-INDEX vs. the UH-INDEX, and (b) the efficiency of the G-INDEX.

windows are now summarized in less detail) is shown in Figure 8 (b). We compare the processing cost of the following four techniques: G-INDEX with GROUP for 10 groups, G-INDEX with GROUP for 50 groups, no index with LIST, and T-INDEX with LIST, the last being our most efficient set-valued index that stores counts for each distinct value. For 1000 distinct values, even a G-INDEX with 10 groups is faster to maintain than a LIST without any indices. G-INDEX is also seen to be more efficient to maintain than T-INDEX. For 10000 distinct values, G-INDEX with 50 groups is again faster than T-INDEX and faster than maintaining a sliding window using LIST without any indices.

7.2.2 Cost of Windowed Histogram Query

Having investigated index maintenance costs, we now use our indices to answer set-valued queries over sliding windows. The first query we test is a windowed histogram, where we wish to return a (pointer to the first element in a) sorted list of attribute values present in the window and their multiplicities. This is a set-valued query that is not incrementally computable unless we have a summary index. We test three basic window implementations: LIST, AGGR, and HASH, as well as three indices: the L-INDEX (where the index contains the answer of the query), the T-INDEX (where an in-order traversal of the tree must be performed whenever we want to produce new results), and the H-INDEX (where a merge-sort of the sorted buckets is needed whenever we

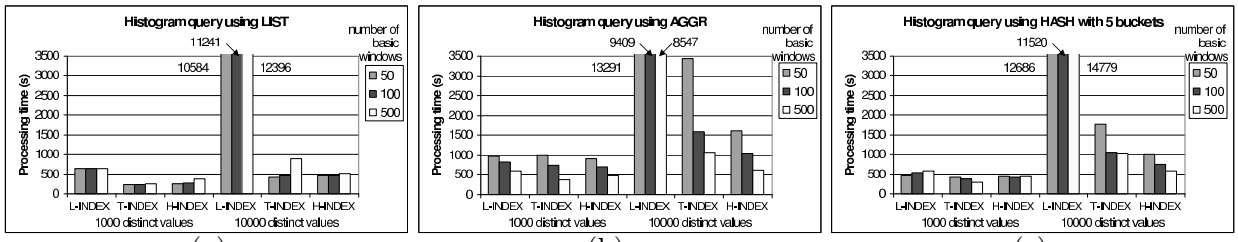


Figure 9: Processing costs (index maintenance and query execution) of a windowed histogram query when using (a) LIST, (b) AGGR, and (c) HASH with five buckets as the basic window implementation.

want new results). We do not test the UH-INDEX because we need the result to be in sorted order. We have also tested the processing time of this query without using any indices in the following two ways: using LIST as the basic window implementation and re-sorting the window from scratch, and using AGGR and merge-sorting the sorted AGGR structures. We found that the former performs several orders of magnitude worse than any indexing technique, while the latter is faster but only outperforms an indexing technique in one specific case, which we will describe shortly. Results are shown in Figure 9. The T-INDEX is the overall winner for each basic window implementation, followed closely by the H-INDEX. The L-INDEX is the slowest, especially when the number of distinct values is large—its processing times extend beyond the range of the graphs. This is because the L-INDEX is very expensive to maintain, despite the fact that it requires no post-processing to produce the answer to the histogram query. As for the best choice of basic window implementation technique, LIST is the fastest if there are many basic windows and many distinct values (in which case there are few repetitions in any one basic window), while HASH wins if repetitions are expected. Moreover, as the number of basic windows increases, answering the histogram query becomes more expensive when using LIST, but cheaper when using AGGR and HASH. This means that basic window maintenance costs outweigh query execution costs in this experiment. Of course, this would not have been the case if our query was more expensive to compute, in which case query execution costs would dominate and the overall cost would increase as the number of basic windows increases.

There was only one case in which a technique that did not use an index was faster than one indexed technique: using AGGR with no index and merge-sorting the AGGR structures was faster than using the L-INDEX over AGGR for fifty basic windows and 10000 distinct values (9183 seconds vs. 13291 seconds)—but much slower than maintaining a T-INDEX or an H-INDEX with the same parameters. This can be explained by noting that with 10000 distinct values, the L-INDEX is very expensive to maintain. Therefore, it is cheaper to only maintain sorted AGGR structures for each basic window and merge-sort them when re-evaluating the query.

7.2.3 Cost of Windowed Intersection

Our second set-valued query is a set intersection of two sliding windows (both of which, for simplicity, have the same size, the same number of basic windows, and the same number of distinct values). Intersection is more expensive than the histogram query from the previous experiment, because whenever re-executing the intersection query, we now have to scan the entire index and return values that appear in both windows. We use the same index for both windows, except that each node in the index now has two counts, one for each window; a value is in the result set of

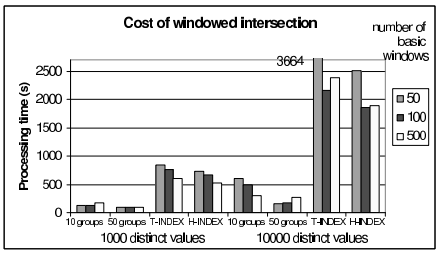


Figure 10: Processing times of a windowed intersection query using G-INDEX over GROUP with 10 and 50 groups, T-INDEX, and H-INDEX.

the intersection if both of its counts are non-zero. However, since it takes approximately equal time to sequentially scan each index, whether it is the L-INDEX, the T-INDEX, or the H-INDEX, the relative performance of each index and basic window implementation is exactly as shown in Figure 7 in the index maintenance section—we are simply adding a constant cost to each technique by scanning the index and returning intersecting values. What we want to show in this experiment is how G-INDEX over GROUP may be used to return an approximate intersection (i.e. return a list of value ranges that occur in both windows) at a dramatic reduction in query processing cost. This is shown in Figure 10, where we compare the G-INDEX with 10 and 50 groups against the fastest implementation of the T-INDEX and the H-INDEX. Interestingly, only G-INDEX over GROUP shows a trend of becoming more expensive as the number of basic windows increases, while the other techniques do the opposite. Again, the reason is that the intersection query is still cheaper to compute than the indices are to maintain. G-INDEX over GROUP, on the other hand, is cheaper to maintain than a full index for two reasons. Firstly, it is cheaper to insert a tuple in a GROUP structure because its list of groups and counts is shorter than an AGGR structure with a list of counts for each distinct value. Also, it is cheaper to look up a range of values in the G-INDEX than a particular distinct value in the L-INDEX.

7.2.4 Lessons Learned

We draw the following general conclusions from our experiments with set-valued indices:

- It is much more efficient to maintain our proposed set-valued indices over sliding windows than it is to re-evaluate continuous queries from scratch.
- Of the various basic window implementation techniques, LIST works well due to its simplicity, while the advantages of AGGR and HASH only appear if basic windows contain many tuples with the same attribute values.
- Of the various indexing techniques, the T-INDEX works well in all situations, though the H-INDEX also performs well in many cases. The L-INDEX is slow.
- We have shown the benefits of using G-INDEX over GROUP as a means of efficiently evaluating an approximate answer to set-valued queries.

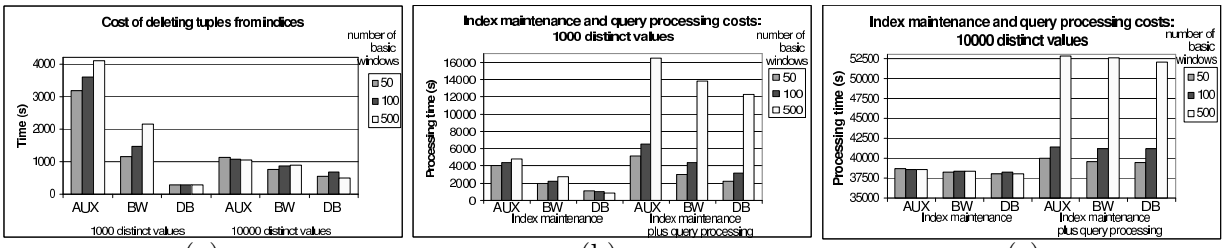
7.3 Experiments with Attribute-Valued Queries

To summarize our analytical observations from Section 6 on attribute-valued indexing, the traditional ring index is expected to perform poorly, AUX should be faster (at the expense of additional

memory usage), while BW should be faster still. DB should be even faster in terms of index maintenance, but query processing may be slower, so it is not clear whether BW or DB is the overall winner. In what follows, we only consider an underlying L-INDEX, but our techniques can also be implemented on top of the T-INDEX, the H-INDEX, or the UH-INDEX (the speed-up factor is the same in each case). We only compare the maintenance costs of AUX, BW, and DB as our experiments with the traditional ring index showed its maintenance costs to be at least one order of magnitude worse than our improved techniques. Since AUX, BW, and DB incur equal costs when inserting tuples in the rings and in the auxiliary index, we first single out the cost of deleting tuples from the rings. This is shown in Figure 11 (a). As expected, DB is the fastest, followed by BW and FW. Furthermore, the relative differences among the techniques are more noticeable if there are fewer distinct values in the sliding window and consequently, more tuples with the same attribute values in each basic window. In this case, we are able to delete multiple tuples from the same ring at once when the oldest basic window expires. In terms of the number of basic windows, FW is expected to be oblivious to this parameter and so we attribute the differences in FW maintenance times to experimental randomness. On the other hand, BW should perform better as we decrease the number of basic windows, which it does. Finally, DB incurs constant deletion costs, so the only thing that matters is how many deletions (one per distinct value) are preformed. In general, duplicates are more likely with fewer basic windows, which is why DB is fastest with 50 basic windows. Similar results were obtained when generating attribute values from a power law distribution, except that duplicates of popular items were more likely, resulting in a greater performance advantage of BW and DB over FW.

We now add tuple insertion costs and query processing costs to determine the fastest ring indexing technique. In this experiment, we have chosen to run a query that sorts the window on the first attribute and outputs the second attribute of each tuple in sorted order of the first. This query requires access to individual tuples, therefore it is a good candidate to make use of our attribute-valued indices, which we defined to be in sorted order of the first attribute. In Figure 11 (b), we graph the index maintenance costs, and the cumulative maintenance and query execution costs for the case of 1000 distinct values in the sliding window. Results for 10000 distinct values are shown in Figure 11 (c); note that the vertical scale begins at 35000 seconds for better readability. Since insertion costs are equal for AUX, BW, and DB, the total maintenance cost simply grows by a constant—it is a rather large constant because we are using an underlying L-INDEX. Consequently, the relative performance differences among our ring indices are far less pronounced in terms of the total cost. However, using the T-INDEX or the H-INDEX decreases maintenance times and shows the advantages of our improved ring indices more clearly. In terms of the total query processing costs, DB is faster than BW by a small margin. This shows that complications arising from the need to check dirty bits during query processing are outweighed by the lower maintenance costs of DB. Nevertheless, one must remember that BW is more space-efficient than DB. Note the dramatic increase in query processing times when the number of basic windows is large and the query is re-executed frequently.

To summarize, our improved attribute-valued indexing techniques are considerably faster than the simple ring index, with DB being the overall winner, as long as at least some repetition of attribute values exists within the window. The two main factors influencing index maintenance costs are the multiplicity of each distinct value in the sliding window (which controls the sizes of the rings, and thereby affects FW and to a lesser extent BW), and the number of distinct values, both in the entire window (which affects index scan times) and in any one basic window (which affects insertion costs). By far, the most significant factor in attribute-valued query processing cost is the basic window size.



(a) (b) (c)

Figure 11: Performance of AUX, BW, and DB in terms of (a) deleting tuples from the index, and processing an attribute-valued query when the sliding window has (b) 1000 or (c) 10000 distinct values.

8 Conclusions and Open Problems

This paper began with questions regarding the feasibility of maintaining sliding window indices. The experimental results presented herein verify that the answer to our question is affirmative, as long as special care is taken to ensure that the indices are efficiently updatable. We addressed the problem of efficient index maintenance by making use of the basic window model, which has been the main source of motivation behind our sliding window query semantics, our data structures for sliding window implementations, and our windowed indices. Future work includes the following problems:

- Indexing materialized views of sliding window query results and sharing them among similar queries.
- Defining a sliding window algebra as well as physical and logical rewritings that could be used in continuous query optimization.
- Implementing additional useful sliding window query operators such as pattern matching and modifying windowed indices to support these operators.
- Developing approximate indices and query semantics for situations where the system cannot keep up with the stream arrival rates and is unable to process every tuple.

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), Aug 2003.

[2] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDMBS: Scaling down database techniques for the smartcard. *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 11–20, 2000.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. *Proc. 1st Biennial Conf. on Innovative Data Syst. Res*, pages 269–280, 2003.

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

[5] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 223–233, 2003.

[6] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 40–51, 2003.

[7] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *Proc. 13th SIAM-ACM Symp. on Discrete Algorithms*, pages 635–644, 2002.

[8] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–8, 1984.

[9] A. Gärtner, A. Kemper, D. Kossmann, and B. Zeller. Efficient bulk deletes in relational databases. *Proc. 17th Int. Conf. on Data Engineering*, pages 183–192, 2001.

[10] L. Golab and M. T. Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[11] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 500–511, 2003.

[12] E. Horowitz, and S. Sahni. Fundamentals of Data Structures. Computer Science Press, Potomac, Maryland, 1987

[13] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[14] T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 239–250, 1986.

[15] L. Qiao, D. Agrawal, and A. El Abbadi. Supporting sliding window queries for continuous data streams. *Proc. 15th Int. Conf. on Scientific and Statistical Database Management*, 2003.

[16] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Sys.*, 1(3):256–267, 1976.

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

[18] J. Srivastava and C. V. Ramamoorthy. Efficient algorithms for maintenance of large database. *Proc. 4th Int. Conf. on Data Engineering*, pages 402–408, 1988.

[19] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 321–330, 1992.

[20] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 358–369, 2002.

[21] Y. Zhu and D. Shasha. Efficient elastic burst detection in data streams. *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2003.