On Indexing Sliding Windows over On-line Data Streams*

Lukasz Golab[1], Shaveen Garg[2], and M. Tamer Özsu[1]

[1] School of Computer Science, University of Waterloo, Canada.
{lgolab,tozsu}@uwaterloo.ca
[2] Department of Computer Science and Engineering, IIT Bombay, India.
shaveen@cse.iitb.ac.in

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. By classifying windowed operators according to their method of execution, we motivate the need for two types of windowed 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. We propose and evaluate indices for both of these cases and show that our techniques are more efficient than executing windowed queries without an index.

1 Introduction

Data management applications such as Internet traffic measurement, sensor networks, and transaction log analysis, are expected to process long-running queries (known in the literature as *continuous queries*) in real time over high-volume data streams. In order to emphasize recent data and to avoid storing potentially infinite streams in memory, the range of continuous queries may be restricted to a sliding window of the N most recent items (count-based windows) or 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.

Windowed queries are re-evaluated periodically, yet the input streams arrive continuously, possibly at a high rate. This environment, in which insertions and deletions caused by high-speed data streams heavily outweigh query invocations, contrasts traditional DBMSs where queries are more frequent than updates. In light of this workload change, we pose the following questions in this paper. Given the usefulness of indices in traditional databases, is it beneficial to index sliding

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

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 as 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. We divide the sliding window into n sub-windows, called *Basic Windows* [16]. To ensure constant sliding window size, each basic window stores the same number of tuples (count-based windows) or spans an equal time interval (time-based windows). Each sub-window may store individual tuples, aggregated summaries, or both. When the newest basic window fills up, it is appended to the sliding window, the oldest basic window is evicted, and queries are 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. Finally, we assume that sliding windows are stored in main memory.

1.2 Related Work

Broadly related to our research is the work on data stream management; see [8] for a survey. Windowed algorithms are particularly relevant as simple queries may become non-trivial when constrained to a sliding window. For example, 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 [16] may be used to decrease memory usage and query processing time by storing summary information rather than individual tuples in each basic window. For instance, storing the maximum value for each basic window may be used to incrementally compute the windowed maximum.

In the basic window approach, results are refreshed after the newest basic window fills up. *Exponential Histograms* (EH) have been proposed in [4] to bound the error caused by over-counting those elements in the oldest basic window which should have expired. The EH algorithm, initially proposed for counting the number of ones in a binary stream, has recently been extended to maintain a histogram in [13], and to time-based windows in [2].

Recent work on windowed joins includes binary join algorithms for count-based windows [11] and heuristics for load shedding to maximize the result size of windowed joins [3]. In previous work, we addressed the issue of joining more than two streams in the context of time-based and count-based windows [9].

Our work is also related to main memory query processing, e.g. [5, 12], especially the domain storage model and ring indexing [1]. However, these works test a traditional workload of mostly searches and occasional updates. Research

in bulk insertion and deletion in relational indices is also of interest, e.g. [6, 15], though not directly applicable because the previous works assume a disk-resident database. Moreover, *Wave Indices* have been proposed in [14] for indexing sliding windows stored on disk. We will make further comparisons between our work and wave indices later on in this paper.

1.3 Contributions

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. Our specific contributions 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 evaluate 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.

1.4 Roadmap

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

2 Physical Storage Methods for Sliding Windows

2.1 General Storage Structure for Sliding Windows

As a general window storage structure, we propose a circular array of pointers to basic windows; implementing individual basic windows is an orthogonal problem, as long as the contents of basic windows can be accessed via pointers. When the newest basic window fills up, its pointer is inserted in the circular array by overwriting the pointer to the oldest basic window. To guarantee constant window size, a new basic window is stored in a separate buffer as it fills up so that queries do not access new tuples until the oldest basic window has been replaced. Furthermore, since the window slides forward in units of one basic window, we 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. when performing a windowed join, tuples which should have expired may be joined; however, this error is bounded by the basic window size), but space usage is decreased. Our data structure is illustrated in Fig. 1.

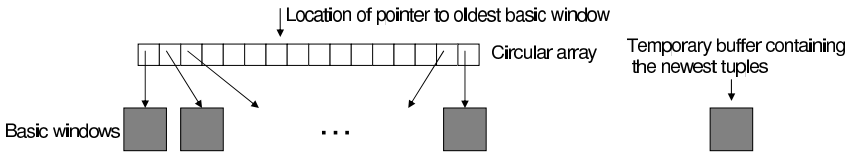


Fig. 1. Sliding window implemented as a circular array of pointers to basic windows

2.2 Storage Structures for Individual Basic Windows

The simplest storage method for individual basic windows is a linked list of tuples, abbreviated as LIST[3]. Again, implementing tuple storage is orthogonal to the basic window data structure. 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). In fact, if basic windows contain tuples with the same attribute value, we can aggregate 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 have 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 using AGGR structures, 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.

2.3 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 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 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.

[3] Time-based windows usually do not store the same number of tuples in each basic window, so we cannot use another circular array.

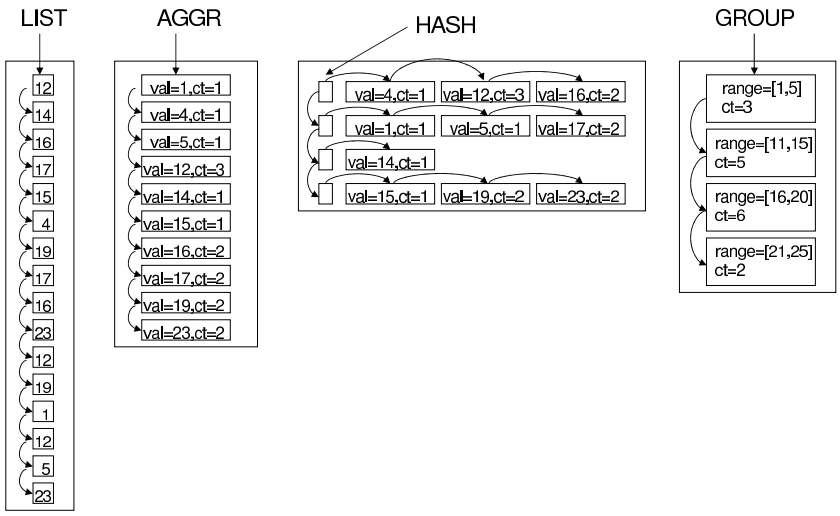


Fig. 2. Our four basic window data structures, assuming that attribute values are stored inside tuples (rather than in an index to which the tuples point, as in the *domain storage* model [1])

In general, window maintenance with the basic window approach is faster than advancing the window upon arrival of each new tuple. However, there is a delay between the arrival of a new tuple and its inclusion in the query result. If t_b is the time span of a basic window, the contribution of a new tuple to the answer will be reflected in the result with a delay of at least t_b and at most $2t_b$—the query must be re-evaluated before the next basic window fills up in order to keep up with the stream. If, however, we are falling behind, we may continue to advance the window after the newest basic window fills up, but we re-execute queries every k basic windows. We may also evaluate easy or important queries often and others more rarely, and adjust the query re-evaluation frequency dynamically in response to changing system conditions.

3 Classification of Sliding Window Operators

We now present a classification of windowed relational operators, revealing that many interesting operators require access to the entire window during query re-execution. We will show that two types of window indices may be useful: those which cover set-valued queries such as intersection, and those which cover attribute-valued queries such as joins.

3.1 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, each tuple is processed on-the-fly, 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, e.g. we could store a sliding window of the result of a selection predicate.
4. In the windowed input and windowed output mode, we define windows over the input streams and materialize the output as a sliding window. For example, we can maintain a sliding window of the results of a windowed join.

Only Mode 1 does not require the use of a window; our storage techniques are directly applicable in all other cases. Furthermore, Modes 2 and 4 may require one of two index types on the input windows, as will be explained next.

3.2 Incremental Evaluation

An orthogonal classification considers whether an operator may be incrementally updated without accessing the entire window. Mode 1 and 3 operators are incremental as they do not require a window on the input. In Modes 2 and 4, we distinguish three groups: incremental operators, 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 (e.g. basic windows) and storing a partial aggregate for each partition. For example, we may compute the windowed sum by storing a 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 instance, we can incrementally compute a windowed intersection 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 each window. The third group includes operators such as join and some non-distributive aggregates such as variance, where the entire window must 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.

4 Indexing for Set-Valued Queries

We begin our discussion of sliding window indices with five techniques for set-valued queries on one attribute. This corresponds to Mode 2 and 4 operators that are not incremental without an index, but become incremental when a histogram

is available. Our 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 stored in an external data structure to which tuples point.

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

The L-INDEX is 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 Fig. 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; we will deal with attribute-valued indices, which need pointers from the index to the tuples, in the next section.

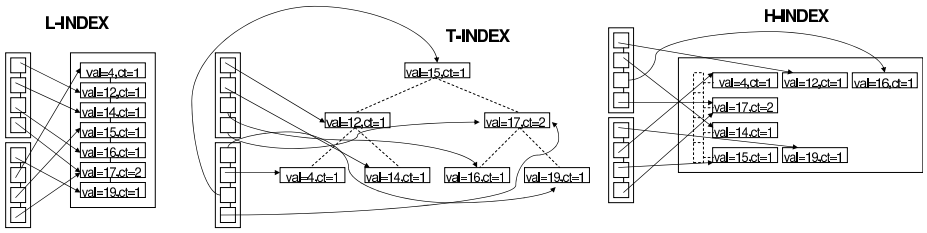
Insertion in the L-INDEX proceeds as follows. For each tuple arrival (LIST) or each new distinct value (AGGR and HASH), we scan the index and insert a pointer from the tuple (or AGGR node) to the appropriate index node. This can be done as tuples arrive, or after the newest basic window fills up; the total cost of each is equal, but the former spreads 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. To delete from the L-INDEX, we scan the oldest basic window as it is being removed, follow pointers to the index, and decrement the counts. There remains the issue of deleting an attribute value from the index when its count drops to zero. We will discuss this problem separately in Sect. 4.4.

4.2 Domain Storage Index as a Search Tree (T-INDEX) or Hash Table (H-INDEX)

Insertion in the L-INDEX is expensive because the entire index may need to be traversed before a 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 [10] is shown in Fig. 3 (b); we will assume that a self-balancing tree is used whenever referring to the T-INDEX in the rest of the paper. A hash table index is another alternative, with buckets containing linked lists of values and frequency counts, sorted by value. An example is shown in Fig. 3 (c).

4.3 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



(a) (b) (c)

Fig. 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

G-INDEX node that stores the label for the group; only one copy of the label is stored. However, with a small number of groups, we may choose not to use the domain storage model and instead store group labels in each basic window. At a price of higher space usage, we benefit from simpler maintenance 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 changes which 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.

4.4 Purging Unused Attribute Values

Each of our indices stores frequency counts for attribute values or groups thereof. A new node must be inserted in the index if a new value appears, but it may not be wise to immediately delete nodes whose counts reach zero. 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 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.

4.5 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 the G-INDEX, h is the number of hash buckets in HASH and the H-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. 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. The G-INDEX is expected to require the least space, 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 requires a hash directory. The per-tuple time complexity can be divided into four steps (except the GROUP implementation, which will be discussed separately):

1. Maintaining the underlying sliding window, as derived in Sect. 2.3.
2. Creating pointers from new tuples to the index, which depends on two things: how many times we scan the index and how expensive each scan is. The former costs 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 in the basic window). The latter costs D for the L-INDEX, $\log D$ for the T-INDEX, and $\frac{D}{h}$ for the H-INDEX.
3. 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. 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, ignoring the cost of periodic purging of zero-count values.

The cost of the G-INDEX over GROUP consists of inserting into the GROUP structure (g per tuple), and merging the delta-file with the index, which is g per basic window, or $\frac{g}{b}$ per tuple (this is the merging approach where the domain storage model is not used). Table 1 summarizes the per-tuple time complexity of maintaining each type of index with each type of basic window implementation, except the G-INDEX. The G-INDEX is expected to be the fastest, while the T-INDEX and the H-INDEX should outperform the L-INDEX if basic windows contain multiple tuples with the same attribute values.

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

	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	$O(\frac{d}{h} + \frac{d}{b}D)$	$O(\frac{d}{h} + \frac{d}{b}\log D)$	$O(\frac{d}{h} + \frac{d}{b}\frac{D}{h})$

We now compare our results with disk-based wave indices [14], which split a windowed index into sub-indices, grouped by time. This way, batch insertions and deletions of tuples are cheaper, because only one or two sub-indices need to be accessed and possibly re-clustered. We are not concerned with disk accesses and clustering in memory. Nevertheless, the idea of splitting a windowed index (e.g. one index stores the older half of the window, the other stores the newer half) may be used in our work. The net result is a decrease in update time, but an increase in memory usage as an index node corresponding to a particular attribute value may now appear in each of the sub-indices.

5 Indexing for Attribute-Valued Queries

In this section, we consider indexing for attribute-valued queries, which access individual tuples, i.e. Mode 2 and 4 operators that are not incremental. Because we need to access individual tuples that possibly have more than one attribute, 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.

5.1 Windowed Ring Index

To extend our set-valued indices to cover attribute-valued queries, we add pointers from the index to some of the tuples. We use a ring index [1], which links all tuples with the same attribute value; additionally, the first tuple is pointed to by a node in the index that stores the attribute value, while the last tuple points back to the index, creating a ring for each attribute value. One such ring, built on top of the L-INDEX, is illustrated in Fig. 4 (a). 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 a LIST of four tuples. Shaded boxes indicate tuples that have the same attribute value of five and are connected by ring pointers. We may add new tuples to the index as they arrive, but we must ensure that queries do not access tuples in the newest basic window until it is full. This can be done by storing end-of-ring pointers, as seen in Fig. 4, identifying the newest tuple in the ring that is currently active in the window.

Let N be the number of tuples and D be the number of distinct values in the sliding window, let b be the number of basic windows, and let d be the average number of distinct values per basic window. When a new tuple arrives, the index is scanned for a cost of D (assuming the L-INDEX) and a pointer is created from the new tuple to the index. The youngest tuple in the ring is linked to the new tuple, for a cost $\frac{N}{D}$ as we may have to traverse the entire ring in the worst case. When the newest basic window fills up, we scan the oldest basic window and remove expired tuples from the rings. However, as shown in Fig. 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 to the start of the ring (now pointing to the next oldest tuple). Thus, deletion costs $\frac{N}{D}$ 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 advanced. The total maintenance cost per tuple is $D + 2\frac{N}{D} + \frac{D}{b}$.

5.2 Faster Insertion with Auxiliary Index (AUX)

Insertion can be cheaper with the auxiliary index technique (AUX), which maintains a temporary local ring index for the newest basic window, as shown in Fig. 5 (a). When a tuple arrives with an attribute value that we have not seen before

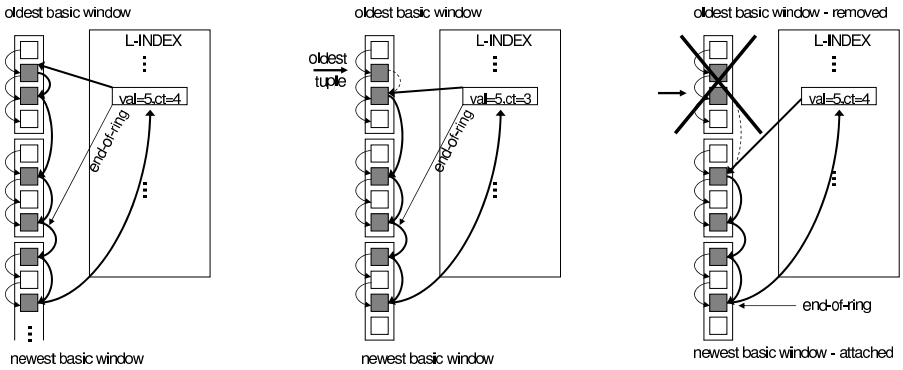


Fig. 4. Maintenance of a windowed ring index

in this basic window, we create a node in the auxiliary index for a cost of d. We then link the 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 cost of $\frac{N}{D}$. If another tuple arrives with the same distinct value, we only look up its value in the auxiliary index and append it to the ring. When the newest basic window fills up, advancing the end-of-ring pointers and linking new tuples to the main index is cheap as all the pointers are already in place. This can be seen in Fig. 5 (b), where dotted lines indicate pointers that can be deleted. The temporary index can 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 Fig. 5. The per-tuple maintenance cost is $d + \frac{d}{b}(1 + D + \frac{N}{D})$ for insertion plus $\frac{N}{D}$ for deletion.

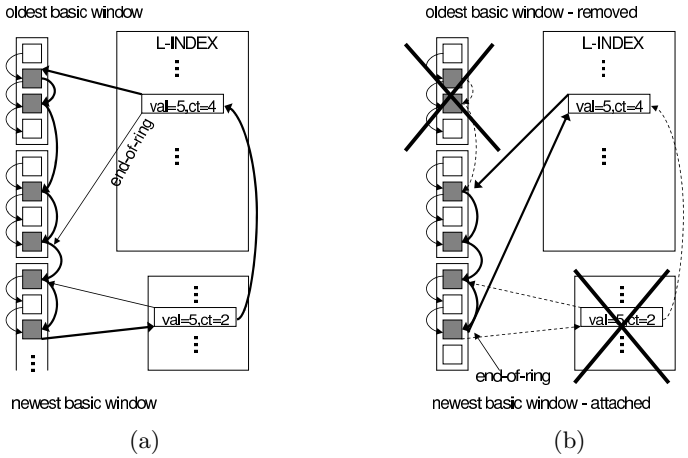


Fig. 5. Maintenance of an AUX ring index

5.3 Faster Deletion with Backward-Linked Ring Index (BW)

The AUX method traverses the entire ring whenever deleting a tuple, which could be avoided by storing separate ring indices for each basic window. However, this is expensive. Moreover, we cannot traverse the ring and delete all expired tuples at once because we do not store timestamps within tuples, so we would not know when to stop deleting. We can, however, perform bulk deletions 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 Fig. 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 Fig. 6 (b). This is possible because tuples are now linked in reverse chronological order. 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 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.

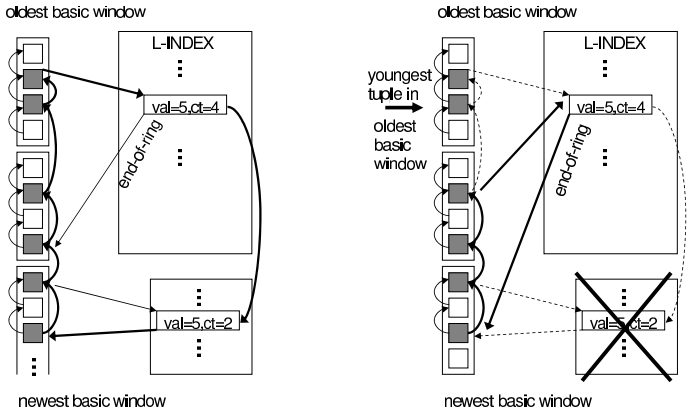


Fig. 6. Maintenance of a BW ring index

5.4 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 solution that

requires additional D bits and D tuples of storage, and slightly increases the query processing time. We use the BW technique, but for each ring, we do not delete the youngest tuple in the oldest basic window. Rather than traversing each ring to find the oldest active tuple, we create a pointer from the youngest tuple in the oldest basic window to the index, as seen in Fig. 6 (b). Since we normally delete the entire oldest basic window, we now require additional storage for up to D expired tuples (one from each ring). Note that we cannot assume that the oldest tuple in each ring is stale and ignore it during query processing—this is true 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).

6 Experiments

In this section, we experimentally 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. Due to space constraints, we present selected results here and refer the reader to an extended version of this paper for more details [7].

6.1 Experimental Setting and Implementation Decisions

We have implemented our proposed indices and basic window storage structures using 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`. Data streams are synthetically generated and consist of tuples with two integer attributes and uniform arrival rates. Thus, although we are testing time-based windows, our sliding windows always contain the same number of tuples. Each experiment is repeated by 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 sliding window size to 100000 tuples, and 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. 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 without noticing a significant difference. We set this value to five times per window roll-over (i.e. once every 20000 tuples). In terms of the number

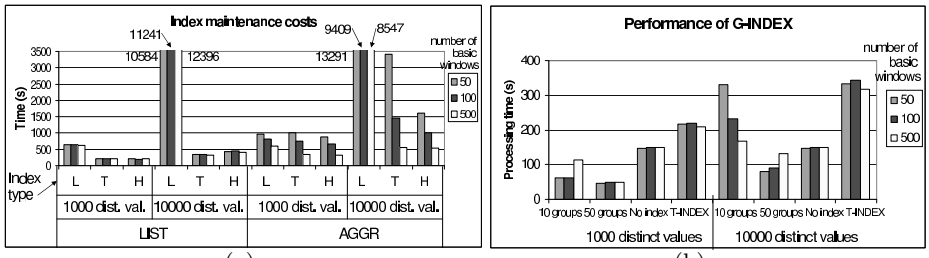
of hash buckets in HASH and the H-INDEX, we observed the expected result that increasing the number of buckets improves performance. To simplify the experiments, we set the number of buckets in HASH to five (a HASH structure is needed for every basic window, so the number of buckets cannot be too large), and the number of buckets in the H-INDEX to one hundred. All hash functions are modular divisions of the attribute value by the number of buckets. The remaining variables in our experiments are the number of basic windows (50, 100, and 500) and the number of distinct values in the streams (1000 and 10000).

6.2 Experiments with Set-Valued Queries

Index Maintenance. Relative index maintenance costs are graphed in Fig. 7 (a) when using LIST and AGGR as basic window implementations; using HASH is cheaper than using AGGR, but follows the same pattern. As expected, the L-INDEX is the slowest and least scalable. 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 (not shown). AGGR fails if there are too few basic windows because it takes a long time to insert tuples into the AGGR lists, especially if the number of distinct values is large. Moreover, 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.

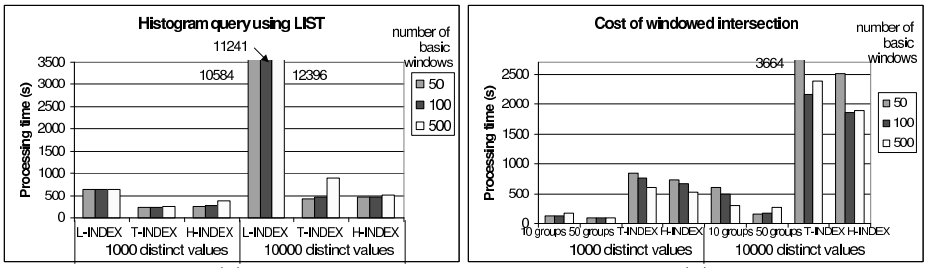
The performance advantage of using the G-INDEX, gained by introducing error as the basic windows are summarized in less detail, is shown in Fig. 7 (b). We compare the cost of four techniques: G-INDEX with GROUP for 10 groups, G-INDEX with GROUP for 50 groups, no index with LIST, and T-INDEX with LIST, the last being our most efficient set-valued index that stores counts for each distinct value. For 1000 distinct values, even the G-INDEX with 10 groups is cheaper to maintain than a LIST without any indices or the T-INDEX. For 10000 distinct values, the G-INDEX with 50 groups is faster than the T-INDEX and faster than maintaining a sliding window using LIST without any indices.

Cost of Windowed Histogram. We now use our indices for answering continuous set-valued queries. First, we test a windowed histogram query that returns a (pointer to the first element in a) sorted list of attribute values and their multiplicities. This is a set-valued query that is not incrementally computable without a summary index. We evaluate three basic window implementations: LIST, AGGR, and HASH, as well as three indices: L-INDEX (where the index contains the answer to the query), T-INDEX (where an in-order traversal of the tree must be performed to generate updated results), and H-INDEX (where a merge-sort of the sorted buckets is needed whenever we want new results). We also execute the query without indices in two ways: using LIST as the basic window implementation and sorting the window, and using AGGR and merge-sorting the basic windows. We found that the former performs several orders



(a) (b)

Fig. 7. Index maintenance costs. Chart (a) compares the L-INDEX (L), the T-INDEX (T), and the H-INDEX (H). Chart (b) shows the efficiency of the G-INDEX



(a) (b)

Fig. 8. Query processing costs of (a) the windowed histogram, and (b) the windowed intersection

of magnitude worse than any indexing technique, while the latter is faster but only outperforms an indexing technique in one specific case—merge-sorting basic windows implemented as AGGRs was roughly 30 percent faster than using the L-INDEX over AGGR for fifty basic windows and 10000 distinct values, but much slower than maintaining a T-INDEX or an H-INDEX with the same parameters. This is because the L-INDEX is expensive to maintain with a large number of distinct values, so it is cheaper to implement basic windows as AGGRs without an index and merge-sort them when re-evaluating the query. Results are shown in Fig. 8 (a) when using LIST as the basic window implementation; see [7] for results with other implementations. The T-INDEX is the overall winner, followed closely by the H-INDEX. The L-INDEX is the slowest because it is expensive to maintain, despite the fact that it requires no post-processing to answer the query. This is particularly noticeable when the number of distinct values is large—processing times of the L-INDEX extend beyond the range of the graph. In terms of basic window implementations, 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 (see [7]).

Cost of Windowed Intersection. Our second set-valued query is an intersection of two windows, both of which, for simplicity, have the same size, number

of basic windows, and distinct value counts. We use one index with each node storing two counts, one for each window; a value is in the result set of the intersection if both of its counts are non-zero. Since it takes approximately equal time to sequentially scan the L-INDEX, the T-INDEX, or the H-INDEX, the relative performance of each technique is exactly as shown in Fig. 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 that 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. In Fig. 8 (b), we compare the G-INDEX with 10 and 50 groups against the fastest implementation of the T-INDEX and the H-INDEX. G-INDEX over GROUP is the cheapest 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.

6.3 Experiments with Attribute-Valued Queries

To summarize our analytical observations from Sect. 5, 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 beat BW in terms of index maintenance, but query processing with DB may be slower. In what follows, we only consider an underlying L-INDEX, but our techniques may be implemented on top of any other 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 tuple insertion costs, we first single out tuple deletion costs in Fig. 9 (a). As expected, deletion in 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 window and consequently, more tuples with the same attribute values in each basic window. In this case, we can 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. 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 (not shown).

Next, we consider tuple insertion and overall query processing costs. We run a query that sorts the window on the first attribute and outputs the second

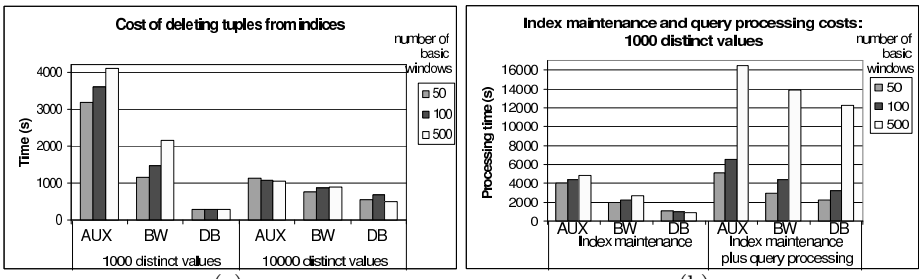


Fig. 9. Performance of AUX, BW, and DB in terms of (a) deleting tuples from the index, and (b) processing an attribute-valued query

attribute in sorted order of the first. This query benefits from our attribute-valued indices because it requires access to individual tuples. In Fig. 9 (b), we graph index maintenance costs and cumulative processing costs for the case of 1000 distinct values; see [7] for results with 10000 distinct values. Since insertion costs are equal for AUX, BW, and DB, total index maintenance costs (first three sets of bars) grow by a constant. In terms of the total query processing cost (last three sets of bars), 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.

6.4 Lessons Learned

We showed that it is more efficient to maintain our proposed set-valued windowed indices than it is to re-evaluate continuous queries from scratch. In terms of basic window implementations, 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. The L-INDEX is slow. Moreover, G-INDEX over GROUP is very efficient at evaluating approximate answers to set-valued queries.

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.

7 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 is affirmative, as long as 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. In future work, we plan to develop techniques for indexing materialized views of sliding window results and sharing them among similar queries. We are also interested in 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. C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez. PicoDMBS: Scaling down database techniques for the smartcard. *VLDB'00*, pp. 11–20.
2. E. Cohen, M. Strauss. Maintaining time-decaying stream aggregates. *PODS'03*, pp. 223–233.
3. A. Das, J. Gehrke, M. Riedewald. Approximate join processing over data streams. *SIGMOD'03*, pp. 40–51.
4. M. Datar, A. Gionis, P. Indyk, R. Motwani. Maintaining stream statistics over sliding windows. *SODA'02*, pp. 635–644.
5. D. J. DeWitt et al. Implementation techniques for main memory database systems. *SIGMOD'84*, pp. 1–8.
6. A. Gärtner, A. Kemper, D. Kossmann, B. Zeller. Efficient bulk deletes in relational databases. *ICDE'01*, pp. 183–192.
7. L. Golab, S. Garg, M. T. Özsu. On indexing sliding windows over on-line data streams. University of Waterloo Technical Report CS-2003-29. Available at `db.uwaterloo.ca/~ddbms/publications/stream/cs2003-29.pdf`.
8. L. Golab, M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.
9. L. Golab, M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. *VLDB'03*, pp. 500–511.
10. E. Horowitz, S. Sahni. Fundamentals of Data Structures. Computer Science Press, Potomac, Maryland, 1987
11. J. Kang, J. Naughton, S. Viglas. Evaluating window joins over unbounded streams. *ICDE'03*.
12. T. J. Lehman, M. J. Carey. Query processing in main memory database management systems. *SIGMOD'86*, pp. 239–250.
13. L. Qiao, D. Agrawal, A. El Abbadi. Supporting sliding window queries for continuous data streams. *SSDBM'03*.
14. N. Shivakumar, H. García-Molina. Wave-indices: indexing evolving databases. *SIGMOD'97*, pp. 381–392.
15. J. Srivastava, C. V. Ramamoorthy. Efficient algorithms for maintenance of large database. *ICDE'88*, pp. 402–408.
16. Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. *VLDB'02*, pp. 358–369.