Lazy Data Structure Maintenance for Main-Memory Analytics over Sliding Windows

Chang Ge
University of Waterloo
Waterloo, Ontario, Canada
c4ge@uwaterloo.ca

Lukasz Golab
University of Waterloo
Waterloo, Ontario, Canada
lgolab@uwaterloo.ca

ABSTRACT

We address the problem of maintaining data structures used by memory-resident data warehouses that store sliding windows. We propose a framework that eagerly expires data from the sliding window to save space and/or satisfy data retention policies, but lazily maintains the associated data structures to reduce maintenance overhead. Using a dictionary as an example, we show that our framework enables maintenance algorithms that outperform existing approaches in terms of space overhead, maintenance overhead, and dictionary lookup overhead during query execution.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

Keywords

Main-memory analytics; Sliding windows; Dictionary encoding

1. INTRODUCTION

Main-memory data warehousing and analytics are becoming possible due to the dropping price of RAM, and becoming necessary due to the need to reduce latency and extract insight from data "at the speed of business" [6, 7]. However, main memory size remains limited as compared to disk, therefore space-saving techniques such as sliding windows (e.g., [2, 3]) and compression (e.g., [1, 4, 9]) are critical in memory-resident data warehouses. Sliding windows provide an intuitive mechanism to discard old data over time, which are no longer of interest to users and/or must be deleted to satisfy data retention requirements. Here, we assume that windows are defined by *length* (e.g., the most recent 60 minutes of data) and a *slide interval* (e.g., slides every 5 minutes).

However, maintaining a table with a sliding window constraint can be expensive due to the constant need to insert new data and remove expired data. Since main-memory analytics must return up-to-date results with low latency, data ingest and maintenance overhead must be kept to a minimum. A common trick is to partition the table by time such that a new partition is added and the oldest partition is dropped whenever the window slides [3, 5, 8]. For example, a table with a 60-minute window that slides every 5 minutes consists of 12 partitions of length 5 minutes each. However, what is less clear is how to efficiently maintain auxiliary data structures over partitioned sliding windows, such as indices or dictionaries. This is exactly the problem we study in this paper.

Consider a dictionary that maps words to integers to enable dictionary encoding and compression – a popular technique in column-store systems [1, 4, 7]. In dictionary encoding, the words appearing in a column are stored in the database as integer IDs, which take up less space. Dictionary lookups must be performed during query execution to translate words to their corresponding IDs and vice versa. There may be a separate dictionary for each column.

One way to implement dictionaries in the context of sliding windows is to build a separate dictionary per partition, as illustrated in Figure 1. Here, the window contains four partitions, and each dictionary contains only those words which appear in its partition. The advantage of this approach is very fast maintenance. Whenever the window slides (e.g., at time $i + 1$, as illustrated in Figure 1), it suffices to build a new dictionary for the new partitionand drop the oldest partition and its dictionary. However, query performance suffers, especially for queries that access most or all of the window, because multiple dictionaries must be probed. Furthermore, since the same word may appear in multiple partitions, the space overhead of the dictionaries may be high.

On the other hand, we can maintain a single dictionary for the entire window and rebuild it whenever the window slides, as shown in Figure 2. Here, the dictionary contains all the words appearing in the current window. This approach uses less space than one-dictionary-per-partition because only one mapping per word needs to be stored. It also gives better query performance since only one dictionary needs to be accessed during query execution. However, the dictionary maintenance costs may be much higher: whenever the window slides, we need to add new words that have appeared in the newest partition, remove words that no longer appear in the window, and re-sort the dictionary.

1.1 Our Contributions

We present a new framework for maintaining data structures associated with partitioned sliding windows, and instantiate it in the context of dictionary encoding. The proposed framework combines the advantages of the two approaches described above while avoiding their drawbacks. That is, 1) the maintenance overhead is much lower than that of a single data structure per window, and not much higher than that of one data structure per partition, and 2) the space and query execution overhead is much lower than that of one data

DOLAP'13, October 28, 2013, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2513190.2513203

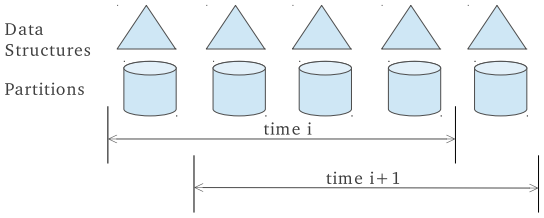

Figure 1: An example of one data structure per partition

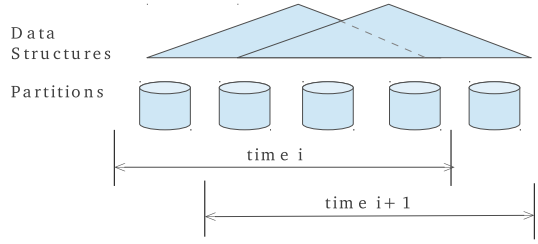

Figure 2: An example of one data structure per window

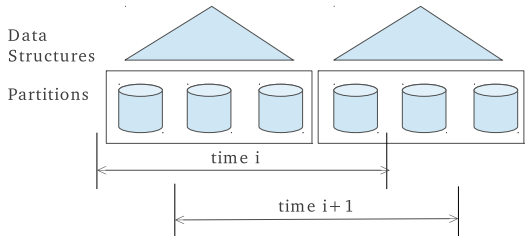

Figure 3: An example of partition grouping

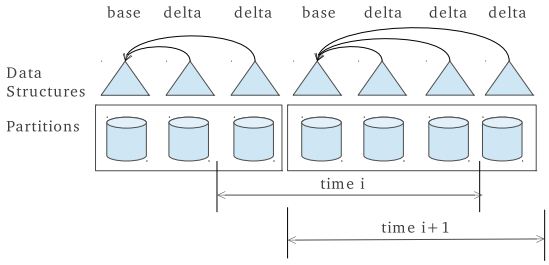

Figure 4: An example of improved partition grouping

structure per partition, and not much higher than that of a single data structure per window. The key idea is to *eagerly* expire data from the sliding window (to save space and satisfy data retention policies), but *lazily* expire the data structures (to reduce the maintenance overhead).

We implement the proposed idea by clustering partitions into groups and maintaining one data structure such as a dictionary per group (not one per partition or one for the whole window). A simple example is shown in Figure 3; we will describe an improved method shortly. The window size is 4 partitions, as before, and the group size is 3 partitions. At time i, the window contains the first four partitions, three of which use the dictionary on the left and one uses the dictionary on the right. At time $i + 1$, the window moves forward and the oldest partition is dropped. The dictionary on the right needs to be updated to include new words that have appeared in the new partition. While we could also update the dictionary on the left and remove words that have only appeared in the just-dropped partition, we can choose not to. Instead, we leave the left dictionary as-is and we will drop it only when all of the partitions it refers to are dropped—this is exactly what we mean by lazy maintenance. This strategy incurs a slight space overhead because the oldest dictionary may now store words that no longer appear in the window. However, the dictionary maintenance overhead will be lower than that of the single-dictionary method, and queries will be faster than with one-dictionary-per-partition since fewer dictionaries will have to be probed.

We then propose an adaptive method for clustering partitions into groups. Rather than fixing a group size, we start a new group (with its own dictionary) depending on the overlap between the words appearing in the group and those present in a new partition. Furthermore, even if we decide that a newly created partition should be placed in an existing group and should reuse an existing dictionary, we do not rebuild that dictionary. Instead, we create a small delta dictionary for words that have not appeared before and we reuse the existing dictionary of that group for other words.

An example is shown in Figure 4, showing one group with three partitions and one with four partitions. The first dictionary of each group is a "base" dictionary that contains exactly those words which appear in the first partition of that group. The remaining dictionaries within a group are "delta" dictionaries that only store additional words (not present in the base dictionary) that appear in their corresponding partitions. When the window moves forward at time $i+1$, a new delta dictionary is created for the newest partition. Additionally, the oldest (left-most) base dictionary is dropped, as is the delta dictionary corresponding to the partition that has just expired from the window (the right-most partition of the left group).

1.2 Related Work

There has been a great deal of work on sliding window processing (see, e.g., [2, 3]), especially in the context of real-time data stream management systems [3]. Furthermore, dictionary encoding is commonly used in column-store OLAP databases (see, e.g., [4, 7, 9]). However, to the best of our knowledge, this is the first work on efficient implementation of data structures such as dictionaries in a main-memory sliding-window context.

In terms of related systems, PowerDrill [4] is a column-store system that supports partitioning, but only one dictionary is used (similar to Figure 2). SAP HANA [9] is a column-store database that supports partitioning and data lifecycle management; e.g., it supports separate partitions within a single logical table for frequently-changing data and static data. However, data expiration was not discussed. Similarly, data and/or data structure expiration was not discussed in other column-store projects such as MonetDB/DataCell [6], C-store [1, 10] and Vertica [5].

Our framework is conceptually similar to Wave Indices [8], which support lazy sliding window maintenance similar to Figure 3. However, Wave Indices were designed for disk-based data warehouses and they lazily maintain the window itself in addition to indices; i.e., space overhead is much less of an issue, and, at some points in time, the window may be longer than required. Furthermore, Wave Indices assume a fixed group size. In contrast, our solution adaptively chooses an optimal group size, and decouples the maintenance of data with the maintenance of indices and other data structures. This way, we can eagerly expire data from the window to save space and satisfy data retention policies, but lazily expire data from the associated data structures in order to reduce their maintenance overhead.

2. OUR SOLUTION

We assume that (tables containing) sliding windows of recent data are divided into time-based horizontal partitions, and that data arrive in batch-mode, with the batch size equal to the partition length (e.g., one minute or 5 minutes). Each tuple is assumed to contain a timestamp, which can be assigned by the system upon arrival or by the source upon generation. Of course, source-assigned timestamps raise the possibility of out-of-order arrivals, in which case late tuples may have to be merged into existing partitions. We assume that the attribute values of tuples will not be modified after they arrive, and we focus on efficient dictionary maintenance under insertion of new data (partitions) and expiration of old data (partitions) as the window slides over time. Dictionary implementation (e.g., a hash table) is orthogonal to the ideas presented in this paper.

We will refer the the approach illustrated in Figure 1 as *multiple-dictionaries* or *multiple*, and the approach from Figure 2 as *single-dictionary* or *single*. We will refer to our approach as *lazy*.

2.1 Dictionary Maintenance Operations

Recall Figure 4. In our approach, partitions are divided into groups; clearly, the number of partitions must be larger than the number of groups. When the window slides forward and a new partition is ready to be inserted, we have two choices. One choice is to start a new group, in which case a *base* dictionary is created, which contains all the words occurring in the new partition. We then use this dictionary to encode the data in the new partition. Alternatively, we can add the new partition to the current group, which already contains a base dictionary and possibly one or more delta dictionaries. In this case, we create a new delta dictionary for the new partition, which contains only those words which are not in this group's *base* dictionary. We then encode data in the new partition using the existing base dictionary and the new delta dictionary.

Additionally, whenever the window slides forward, we drop the oldest partition and its delta dictionary. However, we do not always drop the oldest base dictionary. Returning to Figure 4, at time i, we still need the oldest (leftmost) base dictionary for the currently-oldest (leftmost) partition inside the window. However, at time $i+1$, we can drop the oldest partition from the window, its delta dictionary, and the oldest base dictionary. Note that for efficiency, we never rebuild any (base or delta) dictionaries; we only create or drop them.

In contrast, *single* rebuilds its dictionary whenever the window slides, whereas *multiple* builds a new dictionary for the new partition and drops the oldest dictionary with the oldest partition. Thus, we expect the maintenance overhead of *lazy* to be much lower than that of *single* and nearly as low as that of *multiple*. Furthermore, the space overhead of *lazy* should be lower than *multiple* and not much higher than *single*, provided that we make good decisions about whether to start a new group or join an existing group, as will be discussed shortly.

2.2 Dictionary Lookup

We now discuss the overhead of dictionary lookup during query evaluation. Suppose we are querying the whole window for records containing some word w. For each group of partitions, *lazy* first probes the group's base dictionary to obtain the ID for w. If w does not appear there, then w is not in the first partition of this group, but it may be in other partitions of this group. Thus, we need to check each delta dictionary for w. In the best case, the number of dictionary lookups is equal to the number of groups; if this number is small, then the dictionary lookup overhead is not much higher than that of *single* (more dictionaries need to be probed, but they are smaller than the single large dictionary maintained by *single*). In the worst case, we make as many lookups as there are partitions, but some of these are lookups into very small delta dictionaries. Thus, the worst-case lookup overhead of *lazy* is still better than the average case of *multiple*. A similar analysis holds for queries that only access a sub-interval of time within the window.

Now suppose a query only needs to access a single partition. Here, both *single* and *mutliple* require one dictionary lookup, but *lazy* may require one base dictionary lookup, and possibly an additional delta dictionary lookup. However, since the delta dictionary should be small, the lookup overhead of *lazy* should be not much higher than that of *multiple* and lower than *single* (the base and delta dictionaries together should be much smaller than the single dictionary for the entire window).

2.3 Partition Grouping

We now discuss when to create a new base dictionary for a new partition and when to add it to an existing group. The simplest thing to do is assume a fixed group size; e.g., in Figure 3, every group contains three partitions. We call this approach *lazy-fixed*. However, a better approach is to dynamically decide based on the overlap of the words in the new partition with those in the base dictionary of the current group. We call this approach *lazy-dynamic*.

Let f be the fraction of queries in the workload that access all or most of the window, and $1-f$ be the fraction of queries that access only one partition of data. Assume that most queries accessing only one partition of data will access the most recent partition, i.e., we assume that new data are hotter than old data. There may be other ad-hoc queries that access individual older partitions, but we will not focus on optimizing them in this paper since they are likely to have lower latency requirements than those on new data. Our objective is to minimize the worst-case number of dictionary lookups for a given value of f.

Suppose that at the current time, the newest (current) group contains $k-1$ partitions, for some $k >= 2$. That is, there is one base dictionary and $k-2$ delta dictionaries. Let c_i be the lookup cost into the ith dictionary from the current group. Let c_{other} be the total lookup cost of all the dictionaries from all older groups. Let c_k be the lookup cost of the dictionary corresponding to the new (kth) partition assuming that it joins the current group (i.e., that it will have its own delta dictionary), and let c_k^* be the lookup cost of the dictionary corresponding to the current partition assuming that a new group will be created and a new base dictionary will be built.

Let c_{add} be the worst-case number of dictionary lookups assuming that the newly created partition is added to the current group, i.e., assuming that it becomes the kth partition in the current group:

$$c_{add} = f \times (c_{other} + c_1 + c_2 + \ldots + c_k) + (1-f) \times (c_1 + c_k) \quad (1)$$

That is, in the worst case, a query over the entire window needs to access all the dictionaries from older groups (for a cost of c_{other}) and all the dictionaries from the current group, including the new delta dictionary built for the newest partition. On the other hand, a query against only the newest partition may have to access the base dictionary of the current group (for a cost of c_1) plus the delta dictionary of the newest partition (for a cost of c_k).

Now let c_{new} be the worst-case number of dictionary lookups assuming that the new partition will start a new group with a new base dictionary:

$$c_{new} = f \times (c_{other} + c_1 + c_2 + \ldots + c_{k-1} + c_k^*) + (1-f) \times c_k^* \quad (2)$$

Here, in the worst case, a query over the entire window will access all the dictionaries from older groups (c_{other}), all $k-1$ dictionaries from the most recent group (c_1 through c_{k-1}), plus the new base

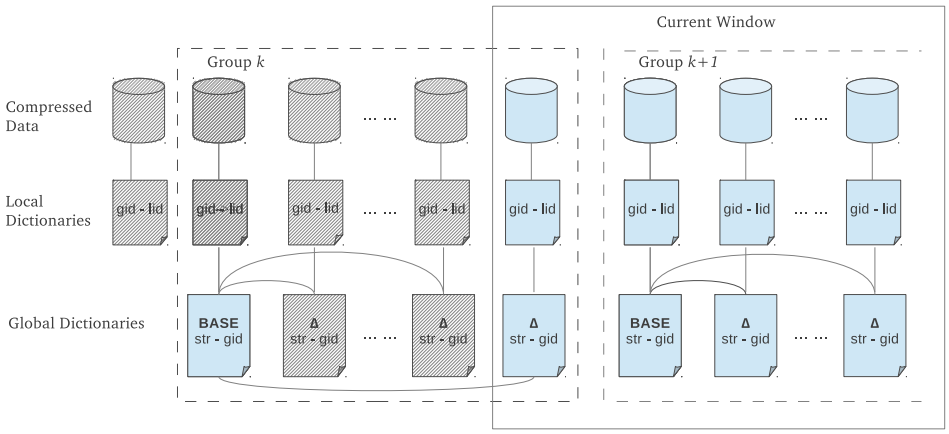

Figure 5: Implementation of *lazy-fixed* and *lazy-dynamic* using double dictionary encoding

dictionary created for the newest partition (c_k^*). A query over the latest partition will only access the newly created base dictionary.

It follows from equations (1) and (2) that

$$c_{add} > c_{new} \Leftrightarrow 1 - f > \frac{c_k^* - c_k}{c_1}. \quad (3)$$

To understand the intuition behind this result, let us make the simplifying assumption that c_1 is roughly equal to c_k^* when compared to c_k, i.e., that these two base dictionaries are equally large and costly to probe compared to the much lower probing cost of the delta dictionary c_k. Equation (3) then simplifies to

$$c_{add} > c_{new} \Leftrightarrow \frac{c_k}{c_k^*} > f. \quad (4)$$

Note that $\frac{c_k}{c_k^*}$ is the proportion of words in the new partition that do not overlap with the current base dictionary, i.e., the access cost of partition k's delta dictionary compared to the access cost of a new base dictionary for partition k. For example, if there are no or very few new words in partition k (i.e., c_k is close to zero) then c_{add} will be lower than c_{new} and we will add partition k to the current group. This makes sense since the new delta dictionary will be very small. On the other hand, if the kth partition contains all new words, then $c_k^* = c_k$ and therefore we will create a new group for partition k. Again, this makes sense since adding partition k to the current group would create a large delta dictionary – so we may as well start a new group. At the same time, if we have more one-partition queries in the workload (i.e., f goes down), then c_{new} is more likely to be lower. This makes sense since such queries would only have to access the new base dictionary instead of the existing base dictionary and possibly also the new delta dictionary.

Equation (3) leads to an algorithm for choosing whether to create a new group or add a new partition into the existing group based on the proportion of whole-window vs. one-partition queries and the proportion of new words appearing in the new partition. We will evaluate the performance of this algorithm in the next section.

3. EXPERIMENTS

We implemented C++ prototypes of *single*, *multiple*, *lazy-fixed* and *lazy-dynamic*. We implemented *single*, *lazy-fixed* and *lazy-dynamic* using double dictionary encoding similar to [4], in which each dictionary maps words to global integer IDs, and each partition additionally has a local dictionary that maps global IDs to local IDs; finally, the data are encoded using local IDs. Dictionaries were implemented as hash tables.

Figure 5 shows the implementation of *lazy* with double dictionary encoding, with global dictionaries at the bottom (with word to global ID mappings) and a layer of local dictionaries in the middle (with global ID to local ID mappings). Figure 5 assumes a sliding window of four partitions, with the most recent (k + 1st) group containing three partitions and the previous (kth) group containing 4 partitions. At the current time, the youngest partition from group k is still within the current window, so we still need its delta dictionary as well as group k's global dictionary. Other global and local dictionaries (and partitions) have been dropped and are greyed out.

Our test machine is equipped with an AMD 3.3GHz 6-core CPU and 16GB of RAM. The OS is Ubuntu 12.04 LTS with kernel version 3.2.0. As input, we used the Twitter data set from snap.stanford.edu/data/twitter7.html, and we used dictionary encoding to store the words inside the tweets. We did not apply any further compression methods to the data. For simplicity, we set each partition to equal roughly 8Mb in size, which amounts to over 56,000 tweets and 135,000 unique words. We tested windows of size 200, 400 and 800 partitions, and we ran each experiment by feeding 6000 partitions to our prototype system.

We found that *lazy-dynamic* created groups ranging from 100 to 800 partitions, with delta dictionaries that were 50 to 200 times smaller than base dictionaries. To measure the effects of sub-optimal (too large or too small) group sizing, we tested two variants of *lazy-fixed*: *lazy-fixed-20* with fixed groups of 20 partitions and *lazy-fixed-2000* with fixed groups of 2000 partitions. We assumed that the proportion of analytics queries that scan the whole window is 0.98 and the proportion of queries that access a single partition is 0.02. Due to space constraints, we omit the results of experiments with other values of these parameters; they resulted in a change of the optimal group size, but led to the same relative performance rankings of the tested algorithms.

The goal of our experiments is to compare our techniques to the two straightforward approaches in terms of 1) maintenance overhead, 2) space usage and 3) dictionary lookup times.

3.1 Dictionary Maintenance Overhead

Figure 6 shows the total dictionary maintenance time for different window sizes; the bars correspond to *single*, *multiple*, *lazy-fixed 20*, *lazy-fixed 2000*, and *lazy-dynamic*, in that order. *Single* is over 15 times slower than the other techniques for a 200-partition window and 30 times slower for a 800-partition window due to the

Figure 6: Comparison of dictionary maintenance times

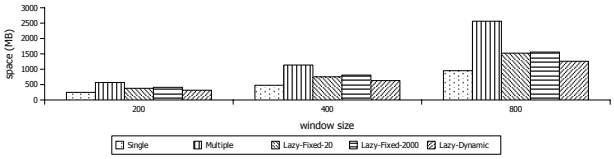

Figure 7: Comparison of dictionary space requirements

overhead of rebuilding the entire dictionary whenever the window slides. The overhead of our algorithms is nearly as low as that of *multiple*, whose maintenance overhead is negligible.

3.2 Space Overhead

Figure 7 shows the average space requirements for each technique, averaged over the duration of the experiment, which inserts 6000 partitions into the warehouse. The bars correspond to the same algorithms in the same order as those in Figure 6. *Single* wins since it maintains only one dictionary and eagerly removes words that no longer appear in the window. *Multiple* has the highest space overhead because many words are repeated in multiple dictionaries. Our techniques, especially *lazy-dynamic* use only slightly more space than *single*. Furthermore, they noticeably outperform *multiple*, especially for large windows with many partitions where *multiple* may need to store the same mapping many times. *Lazy-dynamic* beats *lazy-fixed* as it can adaptively choose between creating a small delta dictionary when the new partition contains few new words, and creating a new base dictionary when there are many new words in the new partition.

3.3 Dictionary Lookup Overhead

We now evaluate the dictionary lookup overhead for two types of queries: those that access all the partitions in the window and those that access only the most recent partition. For each type, the query was a simple selection of a particular column where the column value is equal to some search word. Since we are only evaluating dictionary lookup overhead, these simple queries are sufficient to illustrate the behaviour of the different algorithms.

For both types of queries, *single* always requires one dictionary lookup into the single dictionary that it maintains; however, the dictionary spans the entire window and may be very large and therefore costly to probe. *Multiple* always requires one lookup for each partition required by the query. The number of dictionary lookups for *lazy* depends on the distribution of words across the base and delta dictionaries. In the best case of searching a popular word, the number of lookups equals the number of groups, i.e., we find the word we are looking for in a base dictionary and we do not need to probe any delta dictionaries. A less efficient scenario occurs in case of rare words, where we do not find the search word in a base dictionary and we need to probe all the delta dictionaries. Similarly, if the search word does not appear in the window at all, we still need to search for it in all the dictionaries.

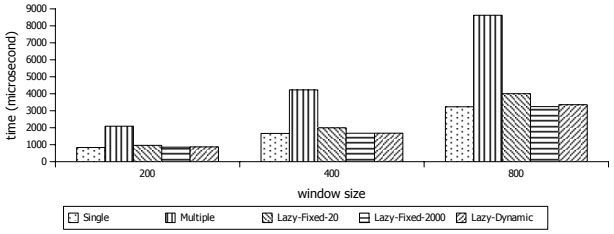

Figure 8: Comparison of dictionary lookup overhead assuming popular search words

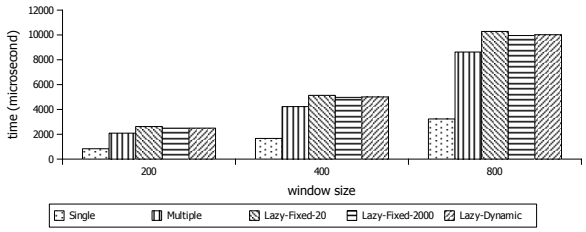

Figure 9: Comparison of dictionary lookup overhead under worst-case conditions for lazy

3.3.1 Queries Scanning the Whole Window

Figures 8 to 10 illustrate the dictionary lookup times under different conditions for various randomly chosen search words. Again, the bars correspond to the same algorithms in the same order as those in Figure 6 and Figure 7.

Figure 8 assumes that the search word was found by the lazy techniques in the base dictionaries, and therefore the delta dictionaries did not have to be accessed. This was the most likely scenario in this set of experiments and it happened whenever we were searching for a popular word, which was very likely to be in every base dictionary. *Multiple* is slow because it needs to access as many dictionaries as there are partitions. *Lazy* probes as many dictionaries as there are groups (it only needs to probe the base dictionary for each group). *Single* only probes one dictionary, but it is not much faster than *lazy*, likely because the one dictionary is large and takes nearly as long to probe as several smaller dictionaries.

Figure 9 shows the total dictionary lookup times in a very rare situation that represents the worst-case scenario for *lazy*: when the search word does not exist in any base dictionary, but it exists in all delta dictionaries. Here, *single* and *multiple* outperform *lazy*. *Single* is the fastest because it only probes one dictionary, while *multiple* performs similarly to *lazy* since both have to access as many dictionaries as there are partitions in the window.

Figure 10 shows the total dictionary lookup times assuming that the search word does not appear anywhere in the window. Here, *single* is very fast because we only have to probe one dictionary; if the word is not in it, we can immediately conclude that it is not in the window. On the other hand, *multiple* probes every dictionary, and so does *lazy*. The overhead of *lazy* is slightly lower than that of *multiple* because some of the dictionaries maintained by lazy (namely the delta dictionaries) are small.

3.3.2 Queries Accessing One Partition

Finally, we examine the dictionary lookup times for queries over a single partition. Results are shown in Figures 11 to 13 for the same three conditions as in the previous experiment: 1) search word found in the base dictionary, 2) search word found in the delta dictionary, and 3) search word not found in the partition. The bars

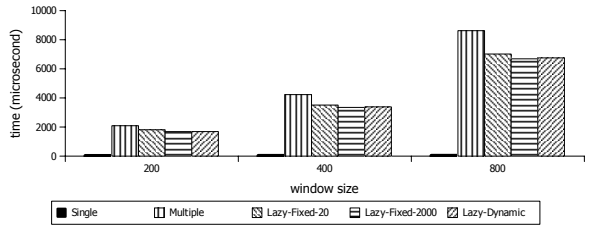

Figure 10: Comparison of dictionary lookup times if the search word does not exist in the window

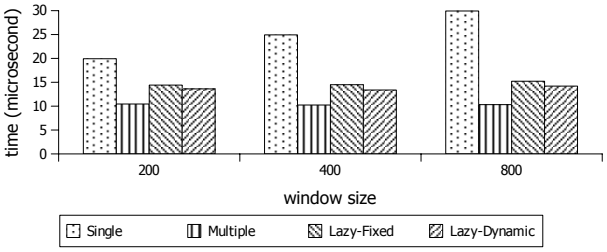

Figure 11: Comparison of dictionary lookup overhead assuming popular search words (single-partition query)

correspond to *single*, *multiple*, *lazy-fixed* and *lazy-dynamic*. We only show one bar for *lazy-fixed* as both *lazy-fixed-20* and *lazy-fixed-2000* had the same performance in this set of experiments. This is because group size does not matter for queries over a single partition; they will always probe the base dictionary, and perhaps also the delta dictionary of the partition being accessed.

Figure 11 assumes that the search word was found in the base dictionary, in which case *multiple* and *lazy* perform very similarly: it suffices to probe the dictionary corresponding to the single partition of interest to the query. *Single* performs worse because its dictionary is much larger than those of *multiple* and *lazy*, and therefore it takes longer to find the mapping for the given search word.

Figure 12 considers the worst possible (but rare) case for *lazy*, where the partition of interest to the query is always associated with a delta dictionary, and the search word is never in the corresponding base dictionary. In other words, we always have to access the base dictionary, and, after not finding the search word there, we access the delta dictionary. While *multiple* works best in this case (it still only has to probe one dictionary), *lazy* outperforms *single*. Again, this is because the single dictionary is large and costly to probe.

Finally, Figure 13 illustrates the situation in which the search word does not exist in the partition accessed by the query. As before, *multiple* only has to probe one dictionary, so its lookup overhead is the lowest. *Lazy* incurs a slightly higher overhead because in some cases, we need to probe the base dictionary first, and, after not finding the search word there, we need to probe the delta dictionary to ensure that the search word does not appear in the partition of interest. Again, *single* performs worst because of the large size of the centralized dictionary.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a new framework for maintaining data structures associated with main-memory sliding windows. Since sliding windows can be expensive to maintain over time, the key idea of our framework is to reduce maintenance overhead by lazily expiring data structures corresponding to old data. We implemented dictionary data structures as per our framework and experimentally showed their advantages over existing techniques. In future work, we plan to build a storage layer to support efficient sliding window maintenance in main-memory data warehouses. The prototype will be based on the lazy maintenance framework proposed in this paper, and will include dictionary encoding as well as other data structures such as indices and column projections.

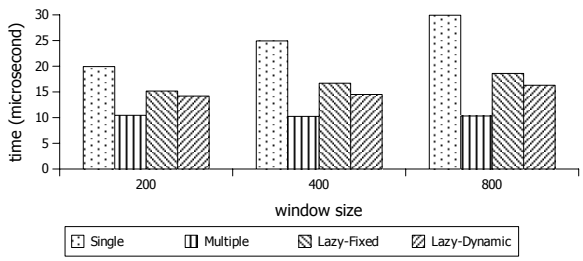

Figure 12: Comparison of dictionary lookup overhead for under worst-case conditions for lazy (single-partition query)

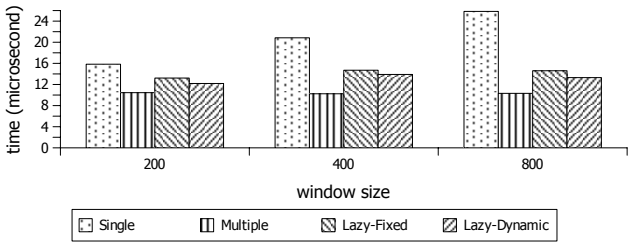

Figure 13: Comparison of dictionary lookup times if the search word does not exist in the window (single-partition query)

5. REFERENCES

[1] Daniel J. Abadi, Samuel Madden, Miguel Ferreira: Integrating compression and execution in column-oriented database systems. SIGMOD Conference 2006: 671-682

[2] Lukasz Golab, Theodore Johnson, J. Spencer Seidel, Vladislav Shkapenyuk: Stream warehousing with DataDepot. SIGMOD Conference 2009: 847-854

[3] Lukasz Golab, M. Tamer Ozsu: Data Stream Management. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2010

[4] Alexander Hall, Olaf Bachmann, Robert Bussow, Silviu Ganceanu, Marc Nunkesser: Processing a Trillion Cells per Mouse Click. PVLDB 5(11): 1436-1446 (2012)

[5] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, Chuck Bear: The Vertica Analytic Database: C-Store 7 Years Later . PVLDB 5(12): 1790-1801 (2012)

[6] Erietta Liarou, Stratos Idreos, Stefan Manegold, Martin L. Kersten: MonetDB/DataCell: Online Analytics in a Streaming Column-Store. PVLDB 5(12): 1910-1913 (2012)

[7] Stephan Muller, Hasso Plattner: An in-depth analysis of data aggregation cost factors in a columnar in-memory database. DOLAP 2012: 65-72

[8] Narayanan Shivakumar, Hector Garcia-Molina: Wave-Indices: Indexing Evolving Databases. SIGMOD Conference 1997: 381-392

[9] Vishal Sikka, Franz Farber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, Christof Bornhovd: Efficient transaction processing in SAP HANA database: the end of a column store myth. SIGMOD Conference 2012: 731-742

[10] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, Stanley B. Zdonik: C-Store: A Column-oriented DBMS. VLDB 2005: 553-564