Storage optimization for query processing over data streams

TANG Xiang-hong†, LI Guo-hui, CHEN Gang, DU Xiao-kun
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, P. R. China

Received 6 April 2010; received in revised form 29 April 2010

Abstract: A defining characteristic of continuous queries over on-line data streams, possibly bounded by sliding windows, is the potentially infinite and time-evolving nature of their inputs and outputs. For different update patterns of continuous queries, suitable data structures bring great query processing efficiency. In this paper, we proposed a data structure suitable for weak non-monotonic update pattern in which the lifetime of each tuple is known at generation time, but the length of lifetime is not necessarily the same. The new data structure combined the ladder queue with the feature of weak non-monotonic update pattern. The experiment results show that the new data structure performs much better than the traditional calendar queue in many cases.

Keywords: calendar queue; ladder queue; query processing; sliding windows

1 Introduction

In many recent data-intensive applications, data management applications such as financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, and sensor networks raise great challenges to traditional database systems. One major challenge is to process long-running queries (known as continuous queries) in real time over high-volume data streams. So as to focus on latest data and to avoid potentially infinite streams overflow, 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). Online continuous queries processing over sliding windows has recently become a concerned research area [1].

Many researchers [2-7] aimed at improving performance of continuous queries that run over the time-based sliding windows. The structure for intermediate data dominates the efficiency of queries processing. Golab and Özsu [8] proposed a classification that defines four types of update characteristics of continuous queries. Based on the classification, they further stated the notion of update-pattern-aware query processing and designed data structures for storing intermediate state, depending on the update patterns of their inputs and outputs. Golab, Garg, and Özsu [9] proposed data structures and query semantics based on a division of the sliding window into sub-windows. By classifying windowed operators, based on their method of execution, they designed two types of windowed indices. At the same time, indexing methods that took advantage of the temporal order in which old answers expire for efficient maintenance and querying of the results over sliding windows were presented [10], so as to allow the queries efficiently access into a relevant subset of the materialized result. Besides, Golab, Prahladka, and Özsu [11] studied the maintenance of time-evolving data over a sliding window that may be spooled to disk for off-line
analysis. Recently, Mortar was proposed as a distributed stream processing platform for building very large queries across federated systems. It presented a cleanslate design for best-effort network processing [12]. Li et al. [13] introduced a new architecture out-of-order processing (OOP) to avoid ordering constraints in stream systems. It freed stream systems from the burden of order maintenance by using explicit stream progress indicators. Boten, Alonso, and Fischer [14] presented a general purpose storage management framework for data stream management systems. It used information about the access patterns of streaming applications to tune and customize the performance of the storage manager. Krämer and Seeger [15] proposed a consistent and powerful operator algebra for data streams, and the algebra ensured that continuous queries had well-defined and deterministic results. Patroumpas and Sellis [16] studied the existence of monotonic-related semantics in windowing constructs towards a more efficient maintenance of their changing contents, and they offered more insight towards design and implementation of stream processing mechanisms by investigating update patterns observed in typical window variants as well as their impact on typical query operators.

Although Refs. [8] and [17] mentioned the data structure for the result of weak non-monotonic queries and employed the calendar queue, the calendar queue [18] has a few fatal defects in the weak non-monotonic queries environment where the lifetime of each result tuple is known at generation time and is not necessarily in the same length. Due to the expiration time of intermediate result items of weak non-monotonic queries under highly skewed distribution, the items tend to fall only in a few buckets with long sublists and many empty buckets exist. Long sublists make enqueue operations expensive since each enqueue entails a sequential search, whereas excessive traversal of empty buckets can increase the process of dequeue operations [19].

Furthermore, the input of a sliding window is not smooth or steady, which fluctuates greatly sometimes. Fluctuation occurs when there are many successive enqueues or dequeues that lead to frequent resizing of the number of buckets in the calendar queue. During a resizing operation, the calendar queue calculates its new bucket-width, re-initializes the multi-list to some number of buckets and the new bucket-width, and then recopies the old multi-list to the new one. This resizing operation is very costly [20]. Consequently, the data structure of intermediate results becomes a bottleneck of the performance of weak non-monotonic queries.

Our purpose was to introduce a novel data structure to overcome above flaws to achieve perfect performance of weak non-monotonic queries. We characterized statistical properties of expiration time of intermediate results during weak non-monotonic queries processing. Based on these properties, we slightly modified the traditional ladder queue [21] by allowing an item to carry a hidden item list that was transparent to the bucket array and altered the algorithm of bucket width. Such a simple redesign led to significant performance improvement in weak non-monotonic queries circumstances. We described a sliding windows model using algebraic expressions and deduced that the new data structure can work better than the calendar queue.

2 System model and assumptions

Using the algebraic expression framework [22], we described a formal definition model of sliding windows. First, the scope function of the sliding window (\(\text{scope}(\tau)\)) was defined as

\[
\text{scope}(\tau) = \begin{cases} 
\tau_0, & \text{if } \tau_0 \leq \tau < \tau_0 + \omega \mod ((\tau - \tau_0), \beta) = 0, \\
\tau_0 + \omega, & \text{if } \tau \geq \tau_0 + \omega \mod ((\tau - \tau_0), \beta) = 0, \\
\text{scope}(\tau - 1), & \text{if } \mod ((\tau - \tau_0), \beta) \neq 0,
\end{cases}
\]

where \(\tau \in \Gamma\) is the time instant that a continuous query is initially submitted specifying a sliding window \(W\); similar to the domain of natural numbers, \(\Gamma\) is regarded as an ordered and infinite set of discrete time instants; \(\omega\) denotes the invariable temporal extent of this window; and \(\beta\) is its progression step. \(\tau_0\) and \(\tau \in \Gamma\) are expressed in timestamp values. For the sake of clarity, all parameters may be considered as natural numbers according to the definition of the time domain \(\Gamma\); so, the scope function is evaluated at discrete time instants of \(\Gamma\).

Each tuple in the data stream \(S\) has a supplementary attribute \(A_{\tau}\), which is designated as the timestamp of tuples and takes its ever-increasing values from \(\Gamma\). For every time instant \(\tau \in \Gamma\), the qualifying tuples are included in the window state [22]:

80
\( W(S, \tau, \tau_0, \omega, \beta, \delta) = \{ s \in S(\tau) : s, A_i \in \text{scope}(\tau) \} \),

where parameters \( \omega, \beta, \) and \( \delta \) are actually the sizes of time intervals (\( \omega, \beta, \) and \( \delta > 0 \)).

We proposed an assumed system architecture to realize the above formal model (Fig. 1). Tuples from input stream are saved in buffer of the memory. Until the sliding window steps forward, tuples in the memory are marked with the same timestamp and put in the sliding window. The lifetimes of all tuples over the sliding window are the length of the sliding window. The queries operations such as join execute over multiple sliding windows and produce the weak non-monotonic intermediate result where the lifetimes of tuples are known and different.

3 Distribution of weak non-monotonic intermediate result

Based on the above formal definition and assumed system architecture, the intermediate result tuples of weak non-monotonic queries have a few features. Their expiration times are determined by the minimal expiration time of their source tuples, and the order of their expiration times is not the same as their input timestamp. Besides, many tuples in the intermediate result have the same expiration time, which show discrete distribution. For example, \( W_1(S_2, \tau, \tau_0, \omega, \beta, \delta_1) \) and \( W_2(S_2, \tau, \tau_0, \omega_2, \beta_2, \delta_2) \) respectively denote different sliding windows. If \( W_1 \otimes W_2 \), the expiration time of the result tuples are discretely distributed in several time instants. The set \( Q_1 = \{ m : m \in \Gamma \text{ and } m \leq \tau \text{ and } (m - \tau_0) \mod \beta_1 = 0 \} \) and \( Q_2 = \{ m : m \in \Gamma \text{ and } m \leq \tau \text{ and } (m - \tau_0) \mod \beta_2 = 0 \} \) denote the time instant set of the right boundary of \( W_1 \) and \( W_2 \), respectively. Let \( \tau_1^T \) be the maximum in \( Q_1 \), and \( \tau_2^T \) be the maximum in \( Q_2 \).

\( Q^L_1 = \{ m : m \in Q_1 \text{ and } m \geq \tau_1^T - \omega_1 + 1 \} \) denotes the set of input timestamp contained in sliding window \( W_1 \), \( Q^L_2 = \{ m : m \in Q_2 \text{ and } m \geq \tau_2^T - \omega_2 + 1 \} \) denotes the set of input timestamps contained in sliding window \( W_2 \), \( Q^T = \{ n : \forall m \in Q^L_1 \text{ and } n = m + (\omega_1 \beta_1 \times \beta_1) \} \) denotes the set of expiration times of tuples contained in sliding window \( W_1 \), and \( Q^T_2 = \{ n : \forall m \in Q^L_2 \text{ and } n = m + (\omega_2 \beta_1 \times \beta_2) \} \) denotes the set of expiration times of tuples contained in sliding window \( W_2 \).

Let \( R^L_1 \) be the minimum item in \( Q^T_1 \), \( R^L_2 \) be the minimum item in \( Q^T_2 \), \( R^L \) be the maximum item in \( Q^T \), and \( R^L_2 \) be the maximum item in \( Q^T_2 \). \( R^T = \text{Min}(R^L_1, R^L_2) \), and \( R^T = \text{Min}(R^L_1, R^L_2) \) are the expiration time of intermediate result tuples have the distribution range as

\[
R = \{ m : m \geq R^T_1 \text{ and } m \leq R^T_2 \text{ and } m \in Q^T_1 \} \cup \{ n : n \geq R^T_1 \text{ and } n \leq R^T_2 \text{ and } n \in Q^T_2 \}.
\]

**Fig. 1 Assumed system architecture**
As we observed above, the expiration time of the intermediate result of weak non-monotonic queries shows distribution characteristics. Can we exploit such an observation to improve the performance of maintaining intermediate result algorithms? Then, we described a novel queue structure with a modified version of the ladder queue that used distribution characteristics to maintain queries result.

4 Sluggish ladder queue

A sluggish ladder queue consists of three components like the traditional ladder queue: Bottom, Ladder and Top [21]. However, the structure of the sluggish ladder queue is different with that of the traditional ladder queue. First, the Top list in the sluggish ladder queue is a double-linked list. We call such a list a trunk tuple list, and a tuple on a trunk tuple list is a trunk tuple. In contrast to the traditional ladder queue, a trunk tuple on the Top list keeps two pointers to the head and tail of another doubly-linked list, which is called a branch tuple list. Tuples on a branch tuple list have the same expiration time as a corresponding trunk tuple. Second, the bucket arrays in rungs resemble the bucket arrays in a traditional ladder queue but tuples are organized in a different manner. In each bucket, all the tuples that have been mapped into it are organized into a double-linked list of trunk tuples or a double-linked list of branch tuples. Third, the Bottom list consists of a sorted double-linked list of trunk tuples [1].

Fig. 2 shows the basic structure of a sluggish ladder queue. In the Top list, there are 18 tuples in total but only 6 trunk tuples here. Hence the number of trunk tuple can be much smaller than the total number of tuples in the queue. This is a key idea of this data structure. We described tuple manipulation operations on a sluggish calendar queue here. The data structure is extended from the traditional ladder queue. Thus, tuple manipulation operations on these two data structures have a lot of similarities. In the following discussion, we will highlight the places where the sluggish ladder queue differs from the traditional ladder queue. Ref. [21] presented details of traditional ladder queue. Stability is an important property of sliding window query algorithms. A stable tuple manipulation algorithm preserves the enqueuing order of any two tuples carrying the same timestamp when they are dequeued. Stable tuple manipulation algorithms are often desirable because they facilitate the result of queries. We thus strictly impose the stability requirement upon the sluggish ladder queue.

Before we present a detailed description of the sluggish ladder queue algorithm, we assume the number of sliding windows as NumSW.

4.1 Enqueue operation

There are two cases.

One case occurs when the Top, Ladder and Bottom are empty. When it is coming, the first tuple is inserted in the Top list as a trunk tuple, but the second tuple and following tuples are different from the first tuple during insertion. When we insert a new tuple into the Top list \( (T_{new}) \), we first check whether it can be added onto the trunk tuple list or the branch list. We use parameter \( \alpha \) to constrain the number of trunk tuples as we want to check against the new tuple. Starting from the tail of the Top list, we iteratively compare the expiration timestamps of the last \( \alpha \) trunk tuples against that of the new tuple. If we find one that has the same expiration timestamp as the latter, we stop the iteration and insert the new tuple into the tail of the branch tuple list which the former points to. If we fail to find the proper trunk tuple among the last \( \alpha \) trunk tuples, we insert the new tuple into the tail of Top list as a trunk tuple. We ensure that tuples bearing the same expiration timestamp appear on the branch tuple list in a first-in first-out (FIFO) manner. In our implementation, we set the default value of \( \alpha \) to be 3.

The other case occurs when all Top, Ladder and Bottom are not empty. Based on the expiration time of the new tuple, the tuple is put in the sublist of some bucket by the method of the traditional ladder queue. The new tuple is inserted in the sublist as a trunk tuple or branch tuple following the way of the first case.

4.2 Dequeue operation

When the deadline of tuple is coming, tuples in Top are transferred to the first rung of ladder like the traditional ladder queue. However, the bucket-width of first rung, denoted by Bucketwidth_1, is obtained differently with the traditional ladder queue.

The new tuple is inserted in the sublist as a trunk tuple or branch tuple following the way of the first case.

\[
\text{Bucketwidth}_1 = \text{MinPS},
\]

where MinPS refers to the minimum progress step of sliding windows, and MinPS = Min\{\(\beta_1, \beta_2, \ldots, \beta_{\text{NumSW}}\}\}, in which \(\beta_i\) denotes the progress step of the \(i\)-th
sliding windows. The reason of using Eq. (1) to determine the bucketwidth is to assure that the tuples with the same expiration time cluster in a bucket and the trunk list is shorter because of the branch list.

When the number of trunk tuples in a bucket exceeds a preestablished threshold value called THRES, a spawning action will be fired. A new bucketwidth of the rung needs to be calculated. The bucketwidth is different with the traditional ladder queue.

\[
\text{Bucketwidth}_{i+1} = \frac{\text{Bucketwidth}_i}{\text{NumSW}},
\]

where \( i > 1 \) and \( \text{Bucketwidth}_i \) represents the bucketwidth of the \( i \)-th rung.

**Legend:**
- T: A trunk tuple
- S: A branch tuple
- X: An invalidated empty bucket where all events have been dequeued before.
- U: A valid bucket may contain an unsorted linked list. Event inserted into these buckets will simply be appended and not sorted.
- A: A bucket used to contain an unsorted linked list
- C: A current dequeue bucket where all events have been sorted and transferred to the Bottom linked list.

**Fig. 2** A sluggish ladder queue
In Eq. (1), the bucketwith of Rung [1] is evaluated by MinPS. The tuples with different expiration timestamps from a sliding window must be distributed in different buckets. That is, two tuples have different expiration timestamps and they must come from different sliding windows. The number of tuples with different expiration timestamps in a bucket is no more than NumSW. Thus, it is enough for the number of the buckets of the new child rung.

4.3 Trunk list and branch list

The expiration timestamp of result tuples clustered at a few time instants. That is, many tuples have the same expiration timestamps. In the traditional ladder queue, infinite spawning of rungs can occur if the number of tuples exceeds THRES and the timestamps associated with all these tuples are all identical. This causes all the tuples to be always enqueued in one bucket irrespective of the number of rungs spawned. To solve the shortcut of ladder queue, a tree structure including a trunk list and a few branch lists is proposed in the ladder queue. In the tree structure, every tuple in the trunk list called trunk tuple points to a double-linked list named the branch list. A tuple in the branch list is called the branch tuple. The expiration time of the trunk tuple and its branch tuples are all identical. In the sluggish ladder queue, a trunk tuple and the branch tuples in its branch list that it points to are regarded as an element of the queue. The adoption of trunk tuple lists and branch tuple lists brings improvement of the following performance.

a) Overcome the weakness of traditional ladder queue that infinite rungs appear when many tuples with the same expiration time join in a queue. As many result tuples of a weak non-monotonic query have the same expiration time, it causes infinite spawning of rungs in the traditional ladder queue. After spawning, tuples are grouped into a few trunk tuples or their branch lists according to their expiration time, and the number of trunk tuples with the same expiration timestamp decreases because of employment of trunk tuple lists and branch tuple lists. Therefore, infinite spawning of rungs will not occur.

b) Degrade the scale of processing time complexity. Based on distribution characteristics of expiration timestamp of weak non-monotonic intermediate result and use of special tree structure, many tuples with identical expiration time are treated as a trunk tuple in the sluggish ladder queue. The number of trunk tuples is much smaller than that of the whole tuples in the queue. Thus, the time complexity of processing is reduced greatly.

c) Decrease the use of space. As the number of trunk tuples is small and most trunk tuples have different expiration time, the maximum number of rungs and the number of buckets are smaller. For this reason, the space complexity of sluggish ladder queue drops.

Stability is pursued by many queries over sliding windows. A stable query is to ensure that the dequeueing order of tuples with the same expiration time appear in a FIFO manner. For the stable order of tuples with the same expiration time, both trunk tuple lists and branch tuple lists of the sluggish ladder queue use double-linked lists. In the double-linked list, it is easy not only to enqueue a tuple but also to dequeue a tuple.

5 Algorithm analysis

In this section, we analyzed the event manipulation operations of a sluggish calendar queue from aspects of correctness and performance.

5.1 Correctness analysis

Correctness of the sluggish ladder queue is related to whether causality errors occur. We defined causality errors as situations in which tuples are dequeued from the sluggish ladder queue out of their expiration timestamp order. Occurrence of causality errors in this setting indicates incorrectness of tuple manipulation operations and is thus not allowable. Based on the algorithm description in Section 4, we can easily prove the following theorem.

Theorem 1. Sluggish ladder queues do not produce causality errors.

Proof. \( T_1 \) and \( T_2 \) are two tuples with different expiration timestamps. The expiration timestamps of \( T_1 \) and \( T_2 \) are \( \text{Exs}T_1 \) and \( \text{Exs}T_2 \), respectively. \( \text{Exs}T_i < \text{Exs}T_j \).

\( D \) represents the interval between \( \text{Exs}T_1 \) and \( \text{Exs}T_2 \). That is, \( D = \text{Exs}T_2 - \text{Exs}T_1 \) and \( D > 0 \).

MaxRng represents the maximum number of rung spawned in the sluggish ladder queue.

Bucketwidth[i], for 1 \( \leq i \leq \text{MaxRng} \), denotes the bucket width of Rung [i].
In dequeue operation, \( T_1 \) and \( T_2 \) are transferred to the Ladder from the Top. If \( D \geq \text{Bucketwidth}[i] \), \( T_1 \) and \( T_2 \) are distributed to a different bucket of Rung [1] and the number of bucket including \( T_1 \) is smaller than that including \( T_2 \). Thus, \( T_1 \) is transferred to the Bottom before \( T_2 \). Thus, \( T_1 \) is dequeued from the queue earlier than \( T_2 \).

If \( D \geq \text{Bucketwidth}[i] \), \( T_1 \) and \( T_2 \) are distributed to different buckets of Rung [1] and the number of bucket including \( T_1 \) is smaller than that including \( T_2 \). Then, \( T_1 \) is transferred to the Bottom before \( T_2 \). Thus, \( T_1 \) is dequeued from the queue earlier than \( T_2 \).

If \( D \geq \text{Bucketwidth}[i + 1] \leq \text{Bucketwidth}[i] \) for \( 1 \leq i \leq \text{MaxRng} - 1 \), \( T_1 \) and \( T_2 \) are distributed to different buckets of Rung \([i+1]\) and the number of bucket including \( T_1 \) is smaller than that including \( T_2 \). Thus, \( T_1 \) is dequeued from the queue earlier than \( T_2 \).

If \( D < \text{Bucketwidth}[\text{MaxRng}] \), \( T_1 \) and \( T_2 \) are distributed into a bucket of Rung \([\text{MaxRng}]\) and both transferred to the Bottom at the same time. In the Bottom, \( T_1 \) and \( T_2 \) are sort and the position of \( T_1 \) is before that of \( T_2 \) for \( \text{Exs} T_1 < \text{Exs} T_2 \). Thus \( T_1 \) is dequeued from the queue earlier than \( T_2 \).

During the enqueue operation, if \( T_1 \) and \( T_2 \) are inserted into different components of queue, \( T_1 \) is inserted into a lower component than \( T_2 \) because \( \text{Exs} T_1 < \text{Exs} T_2 \). For example, \( T_1 \) is inserted into the Top and \( T_2 \) is into the Ladder or the Bottom. In this situation, \( T_1 \) is transferred to the Bottom before \( T_2 \) and is dequeued from the queue first. If \( T_1 \) and \( T_2 \) are inserted into a component of queue, there are two cases discussed. When \( T_1 \) and \( T_2 \) are both inserted into the Top or the Ladder, this case is discussed as the above dequeue operation and \( T_1 \) is dequeued from the queue first in the result. When \( T_1 \) and \( T_2 \) are both inserted into the Bottom, \( T_1 \) must be before \( T_2 \) in the Bottom because tuples in the Bottom ordered by expiration timestamps and \( \text{Exs} T_1 < \text{Exs} T_2 \). Therefore, \( T_1 \) is dequeued from the queue first.

Above all, \( T_1 \) is dequeued from the queue earlier than \( T_2 \), if \( \text{Exs} T_1 < \text{Exs} T_2 \). Thus, Theorem 1 is proven.

5.2 Stability analysis

Stability is an important property that the sluggish ladder queue pursues. If a queue is stable, it is assured that the dequeueing order of any two tuples carrying the same expiration timestamp is the same as their enqueueing order.

**Theorem 2.** Sluggish ladder queues are stable.

**Proof.** \( T_1 \) and \( T_2 \) are two tuples with the same expiration timestamp but \( T_1 \) is inserted into the sluggish ladder queue before \( T_2 \).

In the Top, there are two cases. One case is occurred when \( T_1 \) and \( T_2 \) are both inserted into a branch list. It is obvious that \( T_1 \) must be ahead of \( T_2 \) in the branch list, because a new coming tuple is added at the tail of the branch list. The other case is met when \( T_1 \) and \( T_2 \) are inserted into two different branch lists. In this situation, it is observed that the trunk tuple of \( T_1 \) is in front of the trunk tuple of \( T_2 \) in the Top list. Thus, the Top is a stable structure for the tuples inserted.

When the tuples are transferred into the Ladder, positions of \( T_1 \) and \( T_2 \) are not changed. That is, \( T_1 \) is in front of \( T_2 \) if \( T_1 \) and \( T_2 \) are both in a branch list. If \( T_1 \) and \( T_2 \) belong to two different branch lists in the Top, they must be assigned to a bucket in the Ladder because of their identical expiration timestamps. Starting from the head of the Top list, trunk tuples in the Top list are transferred into the tail of the bucket sublist using Eq. (2). Trunk tuples of \( T_1 \) are before those of \( T_2 \) in the sublist of a bucket or the branch list of \( T_1 \) links to the tail of the branch list of \( T_2 \), and they merge into a new branch list. It is obvious that \( T_1 \) is ahead of \( T_2 \) in the branch list. In the result, the Ladder structure is stable for tuples.

In the Bottom, for the stable sorting algorithm is adopted based on Section 4, the Bottom structure is also stable.

Above all, sluggish calendar queues are stable.

5.3 Performance analysis

In the traditional ladder queue, there are epochs that the ladder queue will always operate with a new set of operating parameters tailored for tuples in the new epoch. Based on Sections 2 and 3, a weak non-monotonic
intermediate result is a data set where the number of tuples enqueued in the result is approximately equal to the number of tuples being dequeued. Then each dequeue operation will advance the current time stamp.

On average, the quantity of results reserves a constant on the whole. Thus, epochs occur in sluggish ladder queues for the weak non-monotonic intermediate result. A mult-epoch consists of some 1-epochs. The performance of 1-epoch determines that of the mult-epoch. We theoretically analyzed the performance of 1-epoch in sluggish ladder queues compared with that in traditional calendar queues.

Complexity of 1-epoch of sluggish ladder queues is discussed as follows.

**Lemma 5.1.** The average number of rungs in a 1-epoch sluggish ladder queue must be bounded by a constant, and the average number of rungs in a 1-epoch sluggish ladder queue is no more than the average number of rungs in a 1-epoch traditional ladder queue with the same operation parameters.

**Proof.** Let \( N \) represent the total number of tuples in a sluggish ladder queue. We use \( \eta \) to denote the average number of tuples with the same expiration timestamp. \( N_i \) represents the total number of trunk tuples in a sluggish ladder queue. \( N_i \approx N / \eta \), and \( \eta > 1 \).

If the trunk tuple and its branch tuples are treated as an element, the sluggish ladder queue with \( N \) tuples is transformed into a traditional ladder queue with \( N_i \) elements. Based on Theorem 3.2 in Ref. [21], the average number of rungs in a 1-epoch traditional ladder queue is bounded by a constant. Thus, the average number of rungs in a 1-epoch sluggish ladder queue must be bounded by a constant.

**Lemma 5.2.** As more levels of child rungs are spawned, the probability that a bucket in the lowest child rung of sluggish ladder queues has no element is higher than that of the traditional ladder queues with the same operation parameters.

**Proof.** We defined \( B_i \) as a random variable that the bucket \( B_i \) or \( B_{sp} \) in Rung \([i] \) of the traditional ladder queue contains a certain number of events ranging from 0 to \( N \).

\( B_i \) represents a random variable that the bucket \( B_i \) or \( B_{sp} \) in Rung \([i] \) of the sluggish ladder queue contains a certain number of events ranging from 0 to \( N \).

\( \theta_i \) represents the bucketwidth of Rung \([i] \). Based on Eq. (2), \( \theta_{i+1} = \frac{\theta_i}{\text{NumSW}} = \frac{\theta_i}{\text{NumSW}^i} \). Note that the analysis is for the 1-epoch traditional ladder queue and sluggish ladder queues where \( \theta_i \) is a constant throughout.

\( \mu \) is the finite mean of the jump random variable that defines the expiration time increment distribution of the \( N \) tuples.

\( q_{i,j} \) denotes the limiting probability that a bucket in Rung \([i] \) of the traditional ladder queue has exactly \( j \) tuples enqueued inside.

\( q_{i,j} \) denotes the limiting probability that a bucket in Rung \([i] \) of the sluggish ladder queue has exactly \( j \) tuples enqueued inside.

Based on the proof of Lemma 5.2, the sluggish queue with \( N \) tuples can be treated as a traditional ladder queue with \( N_i \) elements, and \( N_i \approx N / \eta \).

According to Eq. (8) [21], the following probability expressions are valid.

\[
P(B_i = 0) = q_{i,0} = \frac{\mu}{\mu + N \theta_i / \text{NumSW}^i},
\]

and

\[
P(B_i = 0) = q_{i,0} = \frac{\mu}{\mu + N_i \theta_i / \text{NumSW}^i}.
\]

\( N_i \approx N / \eta \), and \( P(B_i = 0) = q_{i,0} > P(B_i = 0) = q_{i,0} \).

It is important that the structure including the trunk list and the branch list resolves a pathological scenario where more than 50 tuples have the same expiration timestamp and uncontrolled rung spawning may occur.

**Lemma 3.4.** Let \( L \) represent a random variable that counts the number of rungs spawned in a 1-epoch traditional ladder queue. \( P \) denotes the probability of something occurrence. Then for \( n > 1 \), \( P(L = n) = q_{n+1,0} - q_{n,0} \).

**Proof.** No less than \( n \) rungs in a 1-epoch traditional ladder queue means that a bucket in Rung \([n] \) of the 1-epoch traditional ladder queue has no less than 1 tuple. That is, the following probability expression is valid.

\[
P(L \geq n) = 1 - q_{n,0}.
\]

The probability that a 1-epoch traditional ladder queue has no less than \( n \) rungs includes the probabilities that a 1-epoch traditional ladder queue has \( n \) rungs and has more than \( n \) rungs.
\( P(L \geq n) = P(L = n) + P(L \geq n + 1) \).

Thus,
\[
P(L = n) = P(L \geq n) - P(L \geq n + 1) =
\]
\[
(1 - q_{n,h}) - (1 - q_{n+1,h}) = q_{n+1,0} - q_{n,h}.
\]

**Lemma 3.5.** Let \( L \) represent a random variable that counts the number of rungs spawned in a 1-epoch queue. \( P(L = n) \) denotes the probability of the traditional ladder queue with \( n \) rungs, and \( P(L = n) \) denotes the probability of the sluggish ladder queue with \( n \) rungs. Then for \( n > 1 \), \( P(L = n) \geq P(L = n) \).

**Proof.** There are two cases.

One case is that tuples have different expiration time in the traditional ladder queue. It is obvious that the sluggish ladder queue and traditional ladder queue have the same enqueueing and dequeueing operations. Thus, \( P(L = n) = P(L = n) \).

The other case is that some tuples have the same expiration time in the traditional ladder queue. For using a tree structure including a trunk list and a few branch lists, many tuples with the same expiration time are treated as an element in the sluggish ladder queue. In the identical condition, the number of elements in the sluggish ladder queue is smaller than that in the traditional ladder queue. The probability that the new rung is spawned in the sluggish ladder queue is smaller than that in the traditional ladder queue. Thus \( P(L = n) > P(L = n) \).

Above all, \( P(L = n) \geq P(L = n) \).

**Remark.** From Section 3, it is common that many tuples of the weak non-monotonic intermediate result have the same expiration time. So the sluggish ladder queue applies to weak non-monotonic intermediate result effectively.

**Theorem 3.1.** The average number of rungs in a 1-epoch sluggish ladder queue, containing \( N \) tuples, is no more than the average number of rungs in a 1-epoch traditional ladder queue.

**Proof.** Let \( E(L) \) denote the average number of rungs in a 1-epoch traditional ladder queue. \( E(L) \) is used to represent the average number of rungs in a 1-epoch sluggish ladder queue.

\[
E(L) = \sum_{j=1}^{\infty} jP(L = j),
\]

and

\[
E'(L) = \sum_{j=1}^{\infty} jP'(L = j).
\]

Based on Lemma 3.5, \( E'(L) \leq E(L) \).

**Corollary 3.2.** The average time complexity of 1-epoch sluggish ladder queue is no more than that of 1-epoch traditional ladder queue.

**Proof.** The cost of the 1-epoch sluggish ladder queue is gauged by considering the cost of a dequeue operation and the cost of an enqueue operation. If the costs of these basic two operations in the 1-epoch sluggish ladder queue are \( O(1) \), the cost of the 1-epoch sluggish ladder queue will be \( O(1) \). We analyzed costs of these basic two operations of a 1-epoch sluggish ladder queue and compared them with that of a 1-epoch traditional ladder queue.

**Dequeue cost.** When there is already a Bottom list, a fixed cost of dequeue in the 1-epoch sluggish ladder queue like a 1-epoch traditional ladder queue is \( O(1) \) irrespective of \( N \). When there is no Bottom list, the worst-cost of dequeue in a 1-epoch traditional ladder queue is \( O(N(c + 1)) \), where \( c \) is the number of rungs spawned. In the sluggish ladder queue, there will be \( \text{NumSW} \) buckets in each spawned rung. Based on Eq. (1), there are \( \frac{\text{MaxTS} - \text{MinTS}}{\text{MinPS}} \) buckets in total to be traversed for Rung [2], where \( \eta \) denotes the average number of tuples with the same expiration timestamp. The analysis is the same for subsequent rungs. The total cost (worst-case) for transferring \( N \) tuples during dequeue operations from the Top to Rung [1] is \( \frac{\text{MaxTS} - \text{MinTS}}{\text{MinPS}} \), from Rung [1] to Rung \([c^{'}]\) is \( (c^{'} - 1) \frac{N+1}{\text{THRES} \times \eta} \text{NumSW} \), and from Rung \([c]\) to the Bottom is \( \frac{(N+1) \times \text{NumSW}}{\text{THRES} \times \eta} \), where \( c^{'} \) is the number of rungs spawned in the 1-epoch sluggish ladder queue. The total worst-cost (from the Top to the Bottom) during all dequeues is \( \frac{\text{MaxTS} - \text{MinTS}}{\text{MinPS}} + c^{'} \times \frac{(N+1) \times \text{NumSW}}{\text{THRES} \times \eta} \). For \( \frac{\text{MaxTS} - \text{MinTS}}{\text{MinPS}} < N \),
NumSW < THRES, and $\eta > 3$ in weak non-monotonic intermediate result, the following expression is valid:

$$\left[\frac{\text{MaxTS} - \text{MinTS}}{\text{MinPS}}\right] + c \times \frac{(N+1) \times \text{NumSW}}{\text{THRES} \times \eta} < N + c(N+1).$$

If $c$ denotes the average number of rungs spawned in the 1-epoch traditional ladder queue and $c$ represents the average number of rungs in the 1-epoch sluggish ladder queue, the above expression is shown as follows based on Theorem 3.1.

$$\left[\frac{\text{MaxTS} - \text{MinTS}}{\text{MinPS}}\right] + c \times \frac{(N+1) \times \text{NumSW}}{\text{THRES} \times \eta} < N + c(N+1).$$

Thus, a dequeue operation involving rung spawning in the 1-epoch sluggish ladder queue is $O(1)$ amortized and the cost of a dequeue operation in the 1-epoch sluggish ladder queue is smaller than the cost of a dequeue operation in the 1-epoch traditional ladder queue.

**Enqueue cost.** The cost of enqueue is incurred by three cases.

The first case is that the tuple is inserted into the Top list. The cost of enqueue consists of finding the insertion position and appending the tuple into the tail of trunk list or branch list. Based on the description of enqueue operation, finding the insertion position needs no more than three comparisons. The cost of dequeuing a tuple into Top list is $O(1)$.

The second case is that the tuple is inserted into the Ladder structure. The enqueue cost is mainly spent on searching the appropriate insertion level. Based on Lemma 5.1, this enqueue cost is also bounded, for the average number of rungs is bounded by a constant.

The last case is that the tuple is inserted into the Bottom list. The cost is gauged by searching the position in the ordered list. The search cost is also bounded as the number of tuples in the Bottom which is always limited to 50.

**Corollary 3.2.** The 1-epoch sluggish ladder queue has no more than $O(N)$ total memory usage and the total memory usage of the 1-epoch sluggish ladder queue is smaller than that of the 1-epoch traditional ladder queue.

**Proof.** As shown in Eq. (1), Rung $[1]$ requires

$$\left[\frac{\text{MaxTS} - \text{MinTS}}{\text{MinPS}}\right] < (N)$$

buckets on a transfer of tuples from the Top and thus the first rung has an no more than $O(N)$ memory usage. Each subsequent child rung requires $O(\text{NumSW})$ memory space. Since the average number of rungs is bounded and less than that of 1-epoch traditional ladder queue (see Lemma 5.1 and Theorem 3.1), the 1-epoch sluggish ladder queue’s memory consumption is therefore bounded by no more than $O(N)$ and smaller than that of 1-epoch traditional ladder queue.

**Corollary 3.3.** The multi-epoch sluggish ladder queue is theoretically an $O(1)$ queue structure.

**Proof.** The multi-epoch sluggish ladder queue is a special multi-epoch traditional ladder queue where many tuples with the same expiration time are treated as a element by trunk lists and branch lists. Based on Ref. [21], the multi-epoch ladder queue is theoretically an $O(1)$ queue structure. Thus, the multi-epoch sluggish ladder queue is theoretically an $O(1)$ queue structure.

6 Experiment and performance measurement

For a fair comparison, we implemented the sluggish ladder queue, partitioned data structure (like the traditional calendar queue) [8] and the traditional ladder queue in the update-pattern-aware query processor using Microsoft Framework 2.0 Query inputs consist of network traffic data obtained from the Internet Traffic Archive [23] as well as Ref. [8]. Each tuple in the trace consists of the following fields: system-assigned timestamp, session duration, protocol type, payload size, source IP address, and destination IP address. Furthermore, result tuples have an additional timestamp used to determine the expiration time. Testing was performed on a Windows XP laptop with Intel® Core 2 Duo processors of 1.4 GHz and 1 giga byte of RAM. In our experiments, the trace that contained wide-area TCP connections between the Lawrence Berkeley Laboratory and the rest of the world [24] was used and it was broken up into several logical streams based on the destination IP addresses.

In Ref. [8], five basic types of queries mainly consist of Join and Distinct operations and the result tuples were updated by typical weak non-monotonic patterns. Thus, the performance of the data structure to store intermediate and final results of Join and Distinct operations is focused on. For simplicity, the performance of partitioned data structure, ladder queue structure and sluggish ladder structure are tested to
store the results of Join and Distinct operations in different experiment conditions. The two operations are shown in Fig. 3.

![Fig. 3 Join and Distinct operations](image)

Query 1 joins tuples from two outgoing links on the source IP or the protocol. The result size of the latter is hundred times as many as the former. Query 2 selects the distinct source-destination IP pairs on an outgoing link.

There are five experimental parameters: sliding window size, sliding window progression step, lazy expiration interval (for operators that maintain state lazily), the rate of arriving tuples, and the number of partitions in the state buffers. Depending on the query, in terms of time, the window size varies anywhere from 1 s to 1 000 s. The rate of tuples arrives on each link between 1 tuple per second and 100 tuples per second. This range allows us to comment on the performance trends of various data structure as the data size grows. For simplicity, the lazy expiration interval is set to 5% of the window size. Finally, the number of state buffer partitions of calendar queue is set to 10, unless otherwise noted. The reported performance figures correspond to the average overall access times (including tuple insertion and expiration) per tuple processed in result tuples.

We began by testing mean access performance of Query 1 in different speeds of tuple arrival. Fig. 4 illustrates the performance of partitioned data structure (like the traditional calendar queue), traditional ladder queue and sluggish ladder queue at the speed of 5 tuples per second. When the result size is relatively small, three data structures nearly perform the same effect because that there are not many tuples in each bucket of partitioned data structure and the cost of insertion or expiration of tuple is lower. The sluggish ladder queue and the traditional ladder queue have a rung. They work effectively nearly as the partitioned data structure. As the window size grows and result tuples expand, our sluggish ladder queue outperforms the other two and is nearly twice as fast as the portioned data structure. In this case, most tuples cluster in few buckets of portioned data structure and the cost of insertion or expiration of tuple becomes expensive. The sluggish ladder queue and the traditional ladder queue spawn new rungs and there are even tuples in new rungs. So the mean cost of access of tuple increase in a low speed. Furthermore, the sluggish ladder queue adopts trunk and branch lists. The number of trunk tuples is less than the number of tuples in traditional ladder queue, and the sluggish ladder queue works better than the traditional ladder queue.

![Fig. 4 Performances of three data structures for Query 1 at the speed of 5 tuples per second](image)

Fig. 5 shows the performance of partitioned data structure, the traditional ladder queue and sluggish ladder queue for result tuples of Query 1 at the speed of 50 tuples per second. As the window size is small, the states of three data structure are similar with Fig. 4. Until the window size is large, the performance of three data structures change. The mean cost of access of tuple stored in partitioned data structure mounts up faster than that at the speed of 5 tuples per second. There is a slight drop in the performance of the traditional ladder queue. Because the result tuples with the same expiration time become popular and then the number of rungs in the traditional ladder queue becomes larger, as the space of tuples inserted in sliding windows speeds up. Using trunk and branch lists, the performance of sluggish ladder queue is little affected.

![Fig. 5 Performances of three data structures for Query 1 at the speed of 5 tuples per second](image)
Fig. 5 Performances of three data structures for Query 1 at the speed of 50 tuples per second

At the speed of 100 tuples per second, the performance of the three data structures for result tuples of Query 1 is shown in Fig. 6. Efficiency of three data structures is obvious. The average cost of access of tuples in the partitioned data structure and the traditional ladder queue climbs quickly. The few buckets in the partitioned data structure have larger tuples and the cost of finding appropriate position is more expensive when a tuple is inserted in the partitioned data structure. The number of rungs spawned in the traditional ladder queue becomes larger and the mean expense of expiration of tuples grows up because more result tuples have the same expiration time.

In Query 2, the distinct source-destination IP pairs are selected. The result tuples of Query 2 were not as many as that of Query 1. The gap between three data structures about access cost is not as large as that of Query 1.

Fig. 7 shows the performance of three data structures of Query 2 at the speed of 5 tuples per second.

Fig. 8 shows the performances of three data structures of Query 2 at the speed of 50 tuples per second. Compared with Fig. 7, the costs of accesses of three data structures all rise. The sluggish ladder queue is superior over the other two in access cost.

Fig. 9 indicates the mean access cost of three data structures for result tuples of Query 2 at the speed of 100 tuples per second. As the window size expanded, a
lot of growth in mean access time of the partitioned data structure appears. The sluggish ladder queue is one order of magnitude faster than the partitioned data structure.

![Graph showing Performances of three data structures for Query 2 at the speed of 100 tuples per second](image)

Fig. 9 Performances of three data structures for Query 2 at the speed of 100 tuples per second

### 7 Conclusions

In this paper, a sluggish ladder queue was proposed to meet the update of weak non-monotonic tuples over sliding windows. The sluggish ladder queue extended the traditional ladder queue by allowing tuples to carry a branch tuple list. In data streams environment, the sluggish ladder queue had $O(1)$ average time complexity and overcame the weakness of limitless spawning of the traditional ladder queue. Experiment results show that the new data structure performs much better than the partitioned data structure and the traditional ladder queue especially in the large window of time with fast flow. We will test sluggish ladder queues over count-based windows and meet the strict non-monotonic result tuples in the future research.

### References


Association, 2008: 405-418.


