On Concurrency Control in Sliding Window Queries over Data Streams*

Lukasz Golab[1], Kumar Gaurav Bijay[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
gauravk@cse.iitb.ac.in

Abstract. Data stream systems execute a dynamic workload of long-running and one-time queries, with the streaming inputs typically bounded by sliding windows. For efficiency, windows may be advanced periodically by replacing the oldest part of the window with a batch of new data. Existing work on stream processing assumes that a window cannot be advanced while it is being accessed by a query. In this paper, we argue that concurrent processing of queries (reads) and window-slides (writes) is required by data stream systems in order to allow prioritized query scheduling and improve the freshness of answers. We prove that the traditional notion of conflict serializability is insufficient in this context and define stronger isolation levels that restrict the allowed serialization orders. We also design and experimentally evaluate a transaction scheduler that efficiently enforces the new isolation levels.

1 Introduction

A Data Stream Management System (DSMS) executes two types of queries—*long-running* and *snapshot*—whose input streams are typically bounded by sliding windows. Long-running queries return updated answers periodically and often involve complex aggregation for monitoring purposes. Snapshot queries are analogous to traditional database queries in that they can be submitted to the DSMS at any time, are executed once, and return an answer over the current state of the inputs. Snapshot queries may be used to obtain further details in response to a change in the result of a long-running query.

Previous work on sliding window query processing [1-5] and stream query languages [6-8] assumes that windows slide periodically by replacing the oldest part of the window with a batch of fresh data. A periodically-sliding window can be modeled as a circular array of sub-windows, each spanning an equal time interval for time-based windows (e.g., a ten-minute window that slides every minute) or an equal number of tuples for tuple-based windows (e.g., a 100-tuple

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

Y. Ioannidis et al. (Eds.): EDBT 2006, LNCS 3896, pp. 608–626, 2006.

window that slides every ten tuples). We define a *window update* as the process of replacing the oldest sub-window with newly arrived data, thereby sliding the window forward by one sub-window. We will use the terms window update, window movement, and window-slide interchangeably.

As the windows slide forward, a DSMS executes a dynamic workload of long-running and snapshot queries. Suppose that query execution involves accessing a window, one sub-window at a time (we will discuss this in more detail in Sect. 2). Combined with periodic window movements, we can model DSMS data access in terms of two atomic operations: sub-window scan (read) and replacement of the oldest sub-window with new data (write). Thus, a window update is a single write operation, whereas a query is a sequence of sub-window read operations such that each sub-window is read exactly once.

A window may slide while being accessed by a query, resulting in a *read-write conflict*. Consider a sequence of operations illustrated in Fig. 1 (a), where the processing times of window updates (U) and queries (Q_1, Q_2, and Q_3) are shown on a time axis. This represents an ideal scenario, where it is possible to execute all three queries between every pair of window updates, thereby avoiding read-write conflicts. However, the system environment, such as the query workload, stream arrival rates, and availability of system resources, can change greatly during the lifetime of a long-running query. Thus, a more realistic sequence is shown in Fig. 1 (b), where Q_2 takes longer to execute than expected. Q_3 is still running when the second update is ready to be applied, causing a delay in performing the update, and, in turn, causing another read-write conflict when Q_3 is re-executed and the third update is about to take place.

It may appear that read-write conflicts can be prevented by increasing the time interval between window updates, i.e., the sub-window size. However, all sub-windows must have the same size so that the overall window size is fixed at all times. Therefore, either the system must be taken off-line to re-partition the entire window, or two sets of sub-windows must be maintained during the transition period until the window "rolls over" and all the sub-windows have the

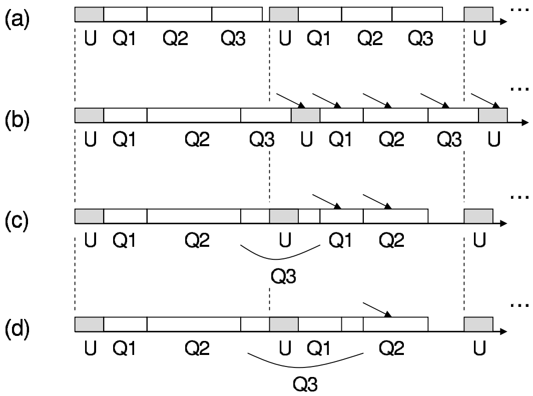


Fig. 1. Examples of query and window update sequences in a DSMS

new size. The first case is inappropriate for an on-line DSMS, whereas the second solution does not immediately eliminate read-write conflicts until the transition period is over.

Existing data stream solutions avoid read-write conflicts by serially executing queries and window movements. In other words, a query locks the window that it is scanning in order to prevent concurrent window movements. Interleaved execution of updates while a window is being scanned by a query is advantageous, provided that the following issue is resolved. Consider suspending the processing of Q_3 in order to perform a window update, as in Fig. 1 (c). Recall that each query is assumed to perform a sequence of atomic sub-window reads, therefore it may be interrupted after it has read one or more sub-windows. It must be ensured that when resumed, Q_3 can correctly read the updated window state. If so, then the answer of Q_3 is slightly delayed (by the time taken to perform the update), but it is more up-to-date because it reflects the second update as well as the first. Otherwise, we are worse off than in Fig. 1 (b), because the answer of Q_3 is delayed, but it is still not up-to-date. Another example is illustrated in Fig. 1 (d), where Q_3 is suspended not only to perform a window update, but also to run Q_1 immediately afterwards. This is desirable if Q_1 is an important query that requires an immediate and up-to-date answer.

This paper studies concurrency control issues in a DSMS with periodic window movements, periodic executions of long-running queries, and on-demand snapshot querying. Our goal is to provide query scheduling flexibility and guarantee up-to-date results. The particular contributions of this paper are as follows.

- By modeling window movements and queries as transactions consisting of atomic sub-window reads and writes, we extend concurrency theory to cover queries over periodically-advancing windows. We show that conflict serializability is not sufficient in the presence of interleaved queries and window movements because some serialization orders produce incorrect answers.
- We propose two isolation levels that are stronger than conflict serializability in that they restrict the permissible serialization orders.
- We design a transaction scheduler that efficiently enforces the desired isolation levels. The main idea is to exploit the access patterns of queries and window updates. The scheduler is proven to be optimal in the sense that it aborts the smallest possible number of transactions while allowing immediate (optimistic) scheduling of window updates.
- We perform an experimental evaluation of the transaction scheduler under various query workloads and system parameters, showing improved query freshness and response times with a minimal drop in throughput.

The remainder of this paper is organized as follows. Section 2 explains our system model and assumptions. Section 3 defines new isolation levels for DSMS transactions, and Sect. 4 presents a transaction scheduler for enforcing them. Section 5 presents experimental results, Sect. 6 compares the contributions of this paper to previous work, and Sect. 7 concludes the paper.

2 System Model and Assumptions

2.1 Data and Query Model

A data stream is assumed to consist of relational tuples with a fixed schema. Without loss of generality, we assume that each stream is bounded by a time-based window. A window of time-length nt is stored as a circular array of n sub-windows, each spanning a time-length of t (each window may have different values for n and t). Every t time units, the oldest sub-window is replaced with a buffer containing incoming tuples that have arrived in the last t time units. Additionally, the DSMS may materialize intermediate results of selected queries or sub-queries, e.g., sliding window joins [2], which may also be stored as arrays of sub-windows [9]. We assume that t is significantly larger than the time taken to perform a window update (otherwise, the system would spend all of its time advancing the windows rather than executing queries).

Given that long-running queries are used for monitoring purposes, they typically compute aggregates over a single window or a join of several windows; a selection predicate may precede the aggregate and a group-by condition may follow it. Each long-running query Q also specifies its desired re-execution frequency. The frequency must be a multiple of t, i.e., Q will be scheduled for re-execution every m window updates, where $1 \leq m < n$. The DSMS attempts to execute all the queries with the desired frequencies, but it cannot guarantee that this will be the case at all times due to unpredictable system conditions.

Queries are executed using one of two techniques (a discussion of other possible evaluation methods and justification of our choice may be found in the extended version of this paper [10]). First, a default access plan scans the entire window (or windows), one sub-window at a time, and computes the query from scratch. Second, aggregates may be computed by accessing a summary, which contains pre-aggregated values for each sub-window. An example is illustrated in Fig. 2, showing a summary that stores the maximum of all the values in each sub-window. A single scan of the summary, from youngest sub-window to oldest, may be used to compute the maximum over windows of different lengths. As illustrated, *max1* is the maximum over a window of size $6t$, which is re-used to compute the maximum over a window of size $10t$ (*max2*). The size of a summary depends on the type of aggregate. Associative aggregates, such as `MAX` and `MIN`, require one value per sub-window (in case of group-by, a separate value is stored for each group). Non-associative aggregates, such as median, top-k, and `COUNT DISTINCT`, need access to the frequency counts of all the distinct values

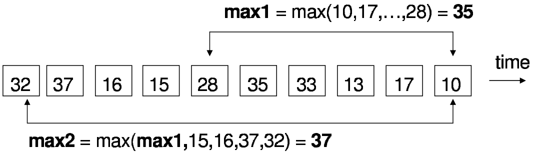

max1 = max(10,17,...,28) = **35**
32
37
16
15
28
35
33
13
17
10
time
max2 = max(**max1**,15,16,37,32) = **37**

Fig. 2. Window summary for computing the maximum

on each sub-window. Alternatively, approximate answers to complex aggregates may be computed by storing summaries that contain estimates of the distribution of values in each sub-window. Examples include Count-Min sketch [11] and Flajolet-Martin sketch [12]. In all cases, query evaluation involves scanning and merging each sub-window summary, from youngest sub-window to oldest.

2.2 System Architecture

The assumed system architecture is illustrated in Fig. 3. Let $w[i]$ denote the replacement of the ith sub-window with newly arrived data, for $0 \le i \le n-1$. Each data stream generates periodic write-only transactions T_j in subscript order, defined as $T_j = \{w_j[j \bmod n]\}$. They are processed by the transaction manager, which immediately propagates updates to all the summaries and materialized results that reference this window (e.g., new tuples are passed to the join operator, which probes the other window and generates new join results). For each stream, the transaction manager initially executes T_0 through T_{n-1} to fill up the windows. Thereafter, each T_j has the effect of moving the window forward by one sub-window. In order to ensure that queries have access to the latest data, the transaction scheduler executes each T_j as soon as a buffer is full.

Snapshot queries are executed by scanning a suitable summary, if available, or accessing the underlying window(s). Answers are returned in the form of a table. Long-running queries are re-executed periodically throughout their *lifetimes* and generate a stream of updated answers. A new long-running query is inserted into the query manager, or may be rejected if the system is overloaded. The query manager then determines an appropriate execution strategy for the new query, e.g., whether an existing summary may be used or a new summary should be built, and whether the new query may be merged into a group of similar queries for shared processing. The design of the query manager is an orthogonal topic, which we pursue in separate work. In this paper, we define an interface between the query manager and the transaction scheduler, which consists of read-only transactions corresponding to re-execution of one or several similar

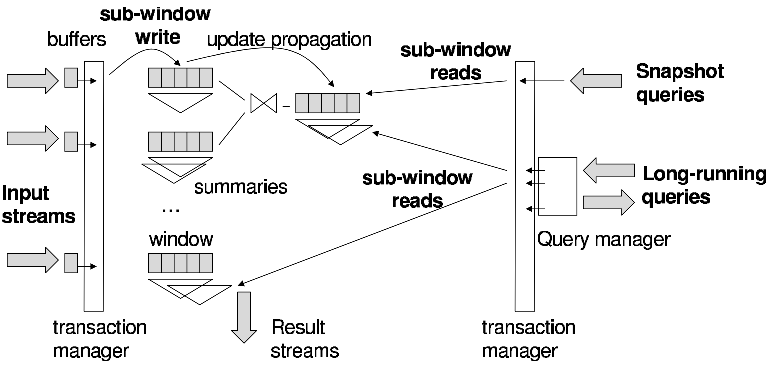


Fig. 3. Assumed system architecture

queries. We define $r[i]$ be a scan (read) of the ith sub-window, or its summary, for $0 \leq i \leq n-1$ (without loss of generality, in the rest of the paper, we will refer to either of these as a sub-window). A snapshot query or a particular re-execution of one or more long-running queries is a read-only transaction T_{Qk}, defined as $T_{Qk} = \{r_{Qk}[0], r_{Qk}[1], \ldots, r_{Qk}[n-1]\}$. That is, each T_{Qk} performs a scan of a window, sub-result, or summary, by reading each sub-window exactly once (queries over windows shorter than nt may be defined similarly). We assume that sub-windows may be read in arbitrary order.

3 Conflict Serializability in the Context of Sliding Window Queries

3.1 Serializability and Serialization Orders

We begin by analyzing the isolation level requirements of queries over periodically-sliding windows. Due to space constraints, we assume that queries access a single window and deal with materialized sub-results in the extended version of this paper [10]. First, we define the possible types of conflicts arising from concurrent execution of transactions. A conflict occurs when two interleaved transactions operate on the same sub-window and at least one of the operations is a write. Clearly, a read-write conflict occurs whenever T_j interrupts T_{Qk}, as in Fig. 1 (c) and (d). This is because each T_{Qk} reads every sub-window, including the sub-window overwritten by T_j. Since we assumed that window movements are executed immediately, we can ignore write-write conflicts. The traditional method for dealing with conflicts requires an execution history H to be serializable. We show that serializability is insufficient in our context using the following example.

Assume a sliding window partitioned into five sub-windows, numbered zero through four, with sub-window zero being the oldest at the current time. Consider the following four histories—H_a, H_b, H_c, and H_d—with c_j or c_{Qk} denoting that transaction T_j or T_{Qk}, respectively, has committed (we omit the initial transactions T_0 through T_4 that fill up the window).

$$\begin{aligned}
H_a &= r_{Q1}[0]\ w_5[0]\ c_5\ w_6[1]\ c_6\ r_{Q1}[1]\ r_{Q1}[2]\ r_{Q1}[3]\ r_{Q1}[4]\ c_{Q1} \\
H_b &= r_{Q1}[0]\ w_5[0]\ c_5\ r_{Q1}[1]\ w_6[1]\ c_6\ r_{Q1}[2]\ r_{Q1}[3]\ r_{Q1}[4]\ c_{Q1} \\
H_c &= r_{Q1}[4]\ w_5[0]\ c_5\ r_{Q1}[3]\ r_{Q1}[2]\ r_{Q1}[1]\ w_6[1]\ c_6\ r_{Q1}[0]\ c_{Q1} \\
H_d &= r_{Q1}[4]\ w_5[0]\ c_5\ r_{Q1}[0]\ r_{Q1}[3]\ r_{Q1}[2]\ w_6[1]\ c_6\ r_{Q1}[1]\ c_{Q1}
\end{aligned}$$

Each history represents interleaved execution of a read-only transaction T_{Q1} and two window movements, T_5 and T_6. Note that H_c and H_d reorder the read operations within T_{Q1}; we will say more about ordering atomic operations in Sect. 4. The associated serialization graphs are drawn in Fig. 4. The direction of the edges corresponds to the order in which conflicting operations are serialized. In particular, there are two pairs of conflicting operations in each schedule: $r_{Q1}[0]$ and $w_5[0]$, and $r_{Q1}[1]$ and $w_6[1]$. Note that all four graphs are acyclic, therefore all four histories are serializable, but their serialization orders are different.

Let us analyze the data read by T_{Q1}. For each history, consider the state of the sliding window shown in Fig. 5, where the first sub-windows on the left

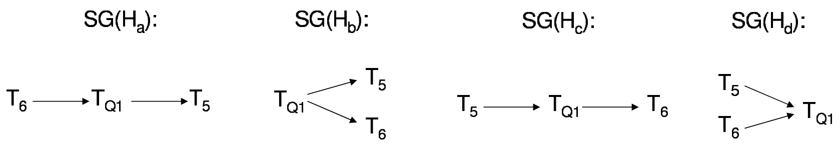


Fig. 4. Serialization graphs for H_a, H_b, H_c, and H_d

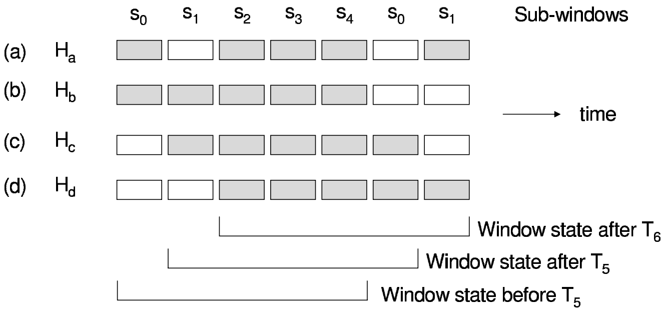


Fig. 5. Differences in the results returned by T_{Q1} in H_a, H_b, H_c, and H_d

(s_0 through s_4) correspond to the initial state of the window after T_0 through T_4 were executed. Next, T_5 advances the window forward by one sub-window, which may be thought of as overwriting the old copy of sub-window s_0 (on the far left) with a new copy, appended after s_4. Thus, the state of the window after T_5 commits is represented by the contiguous sequence of sub-windows $\{s_1, s_2, s_3, s_4, s_0\}$. Then, T_6 advances the window again by appending a new copy of s_1 on the far right and implicitly deleting the old copy of s_1 on the left. Hence, the state of the window after T_6 commits is equivalent to the contiguous sequence of sub-windows $\{s_2, s_3, s_4, s_0, s_1\}$. Shaded sub-windows represent those which were read by T_{Q1} in each of the four histories, as explained next.

First, consider $SG(H_a)$ and note that H_a serializes T_6 before T_{Q1}, meaning that the window movement caused by T_6 (creation of a new version of sub-window s_1) is reflected in the query. However, H_a serializes an earlier window update T_5 after T_{Q1}, therefore the prior window movement caused by T_5 (creation of a new version of s_0) is hidden from the query. Hence, H_a causes T_{Q1} to read an old copy of s_0 and a new copy of s_1, as illustrated in Fig. 5 (a), which does not correspond to a window state at any point in time. This is because the shaded rectangles do not form a contiguous sequence of five sub-windows. Next, recall that H_b serializes both window movements after T_{Q1}, therefore the query reads old versions of s_0 and s_1, as illustrated in Fig. 5 (b). This corresponds to the state of the window after T_4 commits. By similar reasoning, H_c allows T_{Q1} to read the state of the window after T_5 commits (Fig. 5 (c)), and only H_d ensures that T_{Q1} reads the most up-to-date state of the window that reflects both T_5 and T_6 (Fig. 5 (d)). Again, this is because only $SG(H_d)$ serializes both window movements before T_{Qk}, meaning that T_{Qk} sees both updates.

3.2 Isolation Levels for Sliding Window Queries

Having shown that the serialization order affects the semantics of read-only transactions, we propose two stronger isolation levels that restrict the allowed serialization orders.

Definition 1. A serializable history H is said to be *window-serializable (WS)* if all of its committed T_{Qk} transactions read a true state of the sliding window as of some point in the past or present (i.e., a contiguous sequence of sub-windows is read, as in Fig. 5 (b), (c), and (d)).

Definition 2. A window-serializable history H is said to be *latest-window-serializable (LWS)* if all of its committed T_{Qk} transactions read the state of the sliding window that reflects all the window update transactions that have committed before T_{Qk} commits.

Note that only *LWS* guarantees that queries read the most up-to-date state of the window Motivated by Fig. 4, we prove the following results.

Theorem 1. A history H is window-serializable iff $SG(H)$ has the following property: for any T_{Qk}, if any T_i is serialized before T_{Qk}, then for all T_j serialized after T_{Qk}, $i < j$.

Proof. Suppose that H is *WS*. If all transactions T_{Qk} contained in H incur at most one concurrent window movement, then clearly, $SG(H)$ satisfies the desired property. Otherwise, note that for T_{Qk} to read a sliding window state from some point in the past or present, it must be the case that either T_{Qk} is isolated from all the concurrent window updates, or it only reads the least recent update, or it only reads the two oldest updates, and so on. In all cases, $SG(H)$ contains less recent updates serialized before the query and more recent updates serialized after the query, as wanted. Now suppose that $SG(H)$ satisfies the property that all T_j serialized after any T_{Qk} have higher subscripts than those serialized before T_{Qk}. Let m be the maximum subscript of any transaction T_i serialized before T_{Qk}. It follows that T_{Qk} reads a sliding window state that resulted from applying all the updates up to T_m and therefore H is *WS*. □

Theorem 2. A history H is latest-window-serializable iff $SG(H)$ has the following property: for any T_{Qk}, all concurrent T_i transactions must be serialized before T_{Qk}.

Proof. Suppose that H is *LWS* and let T_{Qk} be any query that incurs at least one concurrent window movement. It follows that T_{Qk} reads a state of the window that results from applying all the concurrent updates. Hence, concurrent window updates must be serialized before queries, as wanted. Now suppose that $SG(H)$ does not contain any links pointing from any T_{Qk} to any T_i. This means that there are no queries that have been interrupted by window updates which the queries then did not see. Hence, H is *LWS*. □

4 Transaction Scheduler Design

4.1 Producing *LWS* Histories

We now present the design of a DSMS transaction scheduler that produces *LWS* histories. Recall from Sect. 2 that write-only transactions T_j must be executed with highest priority so that queries have access to an up-to-date version of the window. Given this assumption, our scheduler executes window movements optimistically and uses *serialization graph testing (SGT)* to abort any read-only transaction that causes a read-write conflict. In general, *SGT* may suffer from high space usage and long running time if many conflicts among many transactions must be tracked over time [13]. Fortunately, in our context, the serialization graph is simple and can be pruned dynamically. In particular, for each currently running T_{Qk}, it suffices to monitor concurrent window movements T_j and ensure that all interleaved T_j are serialized before T_{Qk} (recall Fig. 4). Once T_{Qk} commits, it is guaranteed not to cause *LWS* violations at any point in the future, and therefore its node can be safely deleted from the serialization graph.

The scheduler is summarized as Algorithm 1. Lines 3 and 4 serially execute window movements immediately (technically, line 4 must wait for an acknowledgement that the write operation has been performed). Lines 8 through 11 initialize a bit array B_{Qk} for each newly arrived T_{Qk}, where bit i is set if T_{Qk} has already read sub-window i. Lines 12 through 18 execute read-only transactions, one sub-window scan at a time, and set the corresponding bit in B_{Qk} to true. Again, before committing T_{Ql} in line 17, the algorithm must wait for an acknowledgement of performing the read operation from line 14. Note that Algorithm 1 allows multiple read-only transactions to be executed at the same time in any order (line 13) because they do not conflict with one another. Lines 5 through 7 resolve *LWS* conflicts, as proven below.

Theorem 3. Algorithm 1 produces *LWS* histories.

Proof. As per Definition 2, we need to show that all committed read-only transactions T_{Qk} have the property that any window movements T_j that were executed at the same time as T_{Qk} are serialized before T_{Qk}. First, note that the only time that a new *LWS* violation may possibly appear is after a window update T_j commits while one or more T_{Qk} transactions are still running. Furthermore, a *LWS* conflict appears only if any T_j has updated a sub-window (an older copy of) which has already been read by any of the currently running T_{Qk} transactions, in which case T_j would be serialized before T_{Qk}. This occurs if $B_{Qk}[j \bmod n]$ is set for any currently running T_{Qk}. In this case, Algorithm 1 aborts T_{Qk} (line 7), ensuring that all T_{Qk} committed in line 17 satisfy Definition 2. □

Algorithm 1 supports read-only transactions with different priorities, such as snapshot queries or "important" long-running queries (as in Q_1 from Fig. 1 (d)). To do this, we assume that the query manager embeds a priority p within each T_{Qk} and we change line 13 in Algorithm 1 to read: "let T_{Ql} be the transaction in L with the highest value of p". Consequently, if a low-priority T_{Qk} is currently being

Algorithm 1. DSMS Transaction Scheduler

```
let $L$ be the list of currently running $T_{Qk}$ transactions
loop
    if new transaction $T_j$ arrives for scheduling then
        execute $w_j[j \bmod n]$, $c_j$
        for each $T_{Qk}$ in $L$
            if $B_{Qk}[j \bmod n]$ = true then
                execute $a_{Qk}$ (abort $T_{Qk}$)
    elseif new transaction $T_{Qk}$ arrives for scheduling then
        add $T_{Qk}$ to $L$
        for $i = 0$ to $n - 1$
            set $B_{Qk}[i]$ = false
    if $L$ is not empty then
        choose any $T_{Ql}$ from $L$
        execute next operation of $T_{Ql}$, call it $r_{Ql}[m]$
        set $B_{Ql}[m]$ = true
        if no more read operations left in $T_{Ql}$ then
            execute $c_{Ql}$
            remove $T_{Ql}$ and $B_{Ql}$ from $L$
```

executed, then a higher-priority T_{Qm} transaction has the effect of suspending T_{Qk}. This extension does not impact the correctness of Algorithm 1 as it does not introduce any new *LWS* conflicts.

4.2 Optimal Ordering of Read Operations

Given that Algorithm 1 may abort read-only transactions in order to guarantee *LWS*, we want to minimize the required number of aborts. The idea is to shuffle the read operations within T_{Qk} given the following insight. Since aborts occur when a sub-window is being updated but an older version of it has already been read by a concurrent T_{Qk} transaction, we should execute T_{Qk} by first reading the sub-window which is scheduled to be updated the farthest out into the future. More precisely, we define the *time-to-update (TTU)* of a sub-window as the number of window-movement transactions T_j that must be applied until this sub-window is updated. When the scheduler chooses a read-only transaction T_{Qk} to process, it always executes the remaining read operation of T_{Qk} whose sub-window has the highest *TTU* value at the given time. The revised scheduler is shown below as Algorithm 2 (again, adding support for multiple priority levels can be done by changing line 17 to process the highest-priority transaction). There are two main changes. First, lines 6 through 8 update the *TTU* values of each sub-window after every window movement. The newly updated sub-window receives a value of n (it will take n write-only transaction until this sub-window is updated again), whereas the *TTU* values of the remaining sub-windows are decremented. Furthermore, line 18 selects m to be the index of the sub-window which has the highest *TTU* value and has not been read by T_{Ql}.

Algorithm 2. DSMS Transaction Scheduler with TTU

```
let $L$ be the list of currently running $T_{Qk}$ transactions
let $TTU[n]$ be an array of sub-window $TTU$ values
loop
    if new transaction $T_j$ arrives for scheduling then
        execute $w_j[j \bmod n]$, $c_j$
        for $i = 0$ to $n - 1$
            set $TTU[i] = TTU[i] - 1$
        set $TTU[j \bmod n] = n$
        for each $T_{Qk}$ in $L$
            if $B_{Qk}[j \bmod n] =$ true then
                execute $a_{Qk}$ (abort $T_{Qk}$)
    elseif new transaction $T_{Qk}$ arrives for scheduling then
        add $T_{Qk}$ to $L$
        for $i = 0$ to $n - 1$
            set $B_{Qk}[i] =$ false
    if $L$ is not empty then
        choose any $T_{Ql}$ from $L$
        let $m = \text{argmax}_{B_{Ql}[i]=false} TTU[i]$
        execute $r_{Ql}[m]$
        set $B_{Ql}[m] =$ true
        if no more read operations left in $T_{Ql}$ then
            execute $c_{Ql}$
            remove $T_{Ql}$ and $B_{Ql}$ from $L$
```

The idea in Algorithm 2 is similar to the Longest Forward Distance (LFD) cache replacement algorithm [14], which always evicts the page whose next access is latest. LFD is optimal in the off-line case in terms of the number of page faults, given that the system knows the entire page request sequence and that all page faults have the same cost.

Theorem 4. Algorithm 2 is optimal for ensuring *LWS* in the sense that it performs the fewest possible aborts for any history H.

Proof. Let A be the scheduler in Algorithm 2 and let S be any other transaction scheduler that serializes transactions in the same way as A, but only differs in the ordering of read operations inside one or more read-only transactions. That is, S corresponds to Algorithm 1 with some arbitrary implementation of the meaning of "next operation" in line 14. We need to prove that S performs no fewer aborts than A for any history H. Let H_i be the prefix of H containing the first i read operations (interleaved with zero or more write operations, and zero or more commit or abort operations). The proof proceeds by inductively transforming the sequence of read operations produced by S into that produced by A, one read operation at a time. To accomplish this, we let $S_0 = S$ and define a transaction scheduler S_{i+1} that, given S_i, has the following two properties.

1. Both S_i and S_{i+1} order all the read operations in H_i in the same way as A.
2. S_{i+1} orders all the read operations in H_{i+1} in the same way as A and performs no more aborts than S_i in H_{i+1}.

Let $r_k[y]$ be the $(i+1)$st read operation executed by S_i and $r_k[z]$ be the $(i+1)$st read operation executed by S_{i+1}. Due to our assumption that A and S only differ in the ordering of read operations inside read-only transactions, the $(i+1)$st read operations done by S_i and S_{i+1} both belong to the same transaction, call it T_{Qk}. Thus, sub-window $z \pmod n$ has the highest TTU value at this time. Now, if $z = y$ then $S_{i+1} = S_i$ and we are done (property 2 holds). Otherwise, S_{i+1} and S_i differ in the $(i+1)$st read operation. First, suppose that T_{Qk} is not interrupted by any write-only transactions before the next read operation. Then, T_{Qk} is not aborted by S_i or by S_{i+1} in H_{i+1} and we are done (property 2 holds). Next, suppose that T_{Qk} is interrupted by at least one write-only transaction before the next read operation. The remainder of the proof is broken into the following three cases, which collectively prove property 2.

In the first case, suppose that the set of interrupting transactions contains T_y, but not T_z. Given that sub-window $z \pmod n$ has the highest TTU value at this time, and that write-only transactions are generated and serially executed in increasing order of their subscripts, the most recent write-only transaction can have a subscript no higher than $z - 1$. Then, S_i aborts T_{Qk} in H_{i+1}. This is because T_{Qk} has already read an old version of sub-window $y \pmod n$ and therefore T_y would have been serialized after T_{Qk}. However, S_{i+1} does not abort T_{Qk} in H_{i+1}. To see this, observe that T_{Qk} could not have possibly read any of the sub-windows that have just been updated. This is due to the fact that those sub-windows must have lower TTU values than sub-window $z \pmod n$ and must necessarily be scheduled after sub-window $z \pmod n$ by S_{i+1}.

In the second case, suppose that the set of interrupting transactions does not contain T_y or T_z. By the same reasoning, the most recent write-only transaction can have a subscript no higher than $y - 1$. Again, S_{i+1} does not abort T_{Qk} in H_{i+1} because T_{Qk} could not have possibly read any of the sub-windows updated by or before T_{y-1} (they all have lower TTU values than sub-window z (mod n). In terms of satisfying property 2, it does not matter what S_i does in this case.

Finally, in the third case, suppose that the set of interrupting transactions contains both T_y and T_z. Then, both S_i and S_{i+1} abort T_{Qk} in H_{i+1} because both schedulers allow T_{Qk} to read a sub-window that has now been updated. □

5 Experiments

5.1 Implementation Details and Experimental Procedure

We implemented the following transaction schedulers: Algorithm 2 (abbreviated *TTU*), Algorithm 1 (which does not re-order the read operations within transactions, abbreviated *LWS*), a scheduler similar to Algorithm 2 that only enforces window-serializability (abbreviated *WS*), and a scheduler that executes transactions serially (as in current DSMSs, abbreviated *Serial*). The implementation was done in Java 1.4.2, while the experiments were performed on a Pentium-IV PC with a 3 GHz CPU and 1 Gb of RAM, running Linux. The input stream is a sequence of simulated IP packet headers with randomly generated values, e.g., the source and destination addresses have one of one thousand random values, whereas the packet length is a random integer between one and 100. The average data rate is one packet per millisecond, but the specific rate over a particular sub-window is allowed to deviate from the average rate by a factor of up to ten.

We use a long-running query workload representative of an on-line network traffic analysis application (see, e.g., [15, 16]), consisting of top-k queries over the source or destination IP addresses, and percentiles over the total bandwidth consumed by (or directed to) distinct IP addresses. The window sizes referenced by queries are generated randomly between one and n, where n is the total number of sub-windows. Similar aggregates over different window sizes are evaluated together. For simplicity of implementation, long-running queries are executed by scanning the window and building a hash table on the required attribute. Snapshot queries are chosen from a set of simple aggregates over a random subset of the source and destination IP addresses. Each query references the same time-based window, which is stored in main memory.

After initializing the sliding window using a randomly generated input stream, we test each of the four transaction schedulers over an identical query workload. The tests proceed for a time equal to the window length. We then repeat each test five times using different input streams and calculate the average of each measurement being reported. The parameters being varied in (and across) the experiments are the query workload, the window size (controlled via the number of sub-windows), and the length of each sub-window (which controls the frequency of window movements). The following performance metrics are used to evaluate the four transaction schedulers (as illustrated on a time line in Fig. 6).

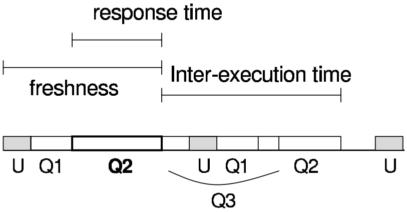


Fig. 6. Freshness, response time, and inter-execution time of query Q_2

- *Query freshness* is the difference between the time that a query reports an answer and the time of the last window update reflected in the answer.
- *Response time* is the difference between the query execution start time and end time. This metric is particularly important for snapshot queries, which are usually time-sensitive.
- *Inter-execution time* of a long-running query is the length of the interval between its re-executions. A DSMS is expected to tolerate slightly longer

inter-execution times if the returned answers are more up-to-date. The motivation for this is that even if we return an older answer earlier, we would have to re-execute the query soon in order to produce an answer that reflects the new state of the window.

5.2 Experiments with Long-Running Queries

We begin by executing *Serial*, *WS*, *LWS*, and *TTU* on a workload consisting of long-running queries and interleaved window movements. We test two sub-window sizes: $t = 1$ sec. and $t = 5$ sec., with the number of sub-windows varied from ten to 100. The number of queries is set to 40 for $t = 1$ sec. and 100 for $t = 5$ sec. For now, we assume that snapshot queries are not posed. We measure the average freshness, inter-execution time, and throughput.

The average query freshness is shown in Fig. 7 (the lower the value, the better). *TTU* and *LWS* clearly outperform *WS* and *Serial* because the first two guarantee latest-window-serializable schedules, where queries have access to an up-to-date state of the window. Freshness deteriorates for all four schedulers as the sub-window size grows to $t = 5$ sec. and window movements become less frequent. Moreover, increasing the number of sub-windows (or equivalently, increasing the window length) generally has an adverse effect on freshness because the query execution times increase. Note that *Serial* performs slightly better than *WS* because *WS* adds to the query execution time by performing concurrent window movements, yet the answer does not reflect any of the updates. Overall, *TTU* provides the best query freshness in all tested scenarios.

The average query inter-execution times are illustrated in Fig. 8. Each cluster of eight bars corresponds, in order, to *Serial*, *WS*, *LWS*, and *TTU* for $t = 1$ sec., followed by *Serial*, *WS*, *LWS*, and *TTU* for $t = 5$ sec. Serial has the best (lowest) inter-execution times because it does not incur the overhead of serialization graph testing, therefore its total query execution time is slightly lower. Notably, *LWS* (corresponding to the third and seventh bars in each cluster) performs the worst because it aborts a significant percentage of transactions (see [10] for full details).

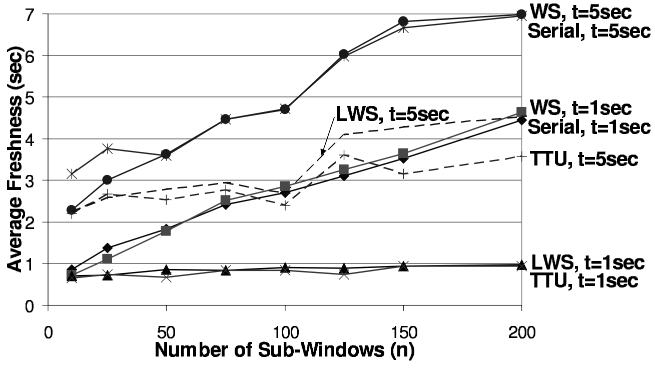


Fig. 7. Comparison of query freshness for Serial, WS, LWS, and TTU

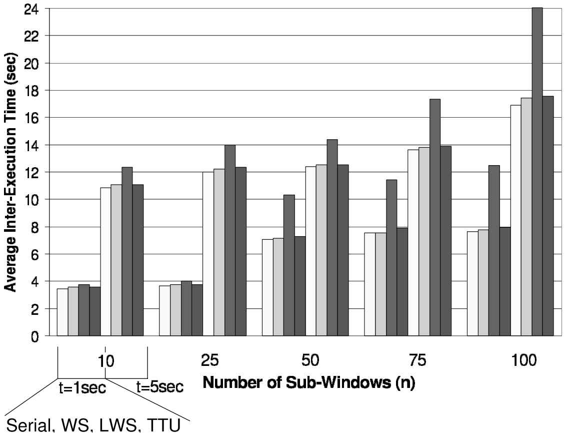


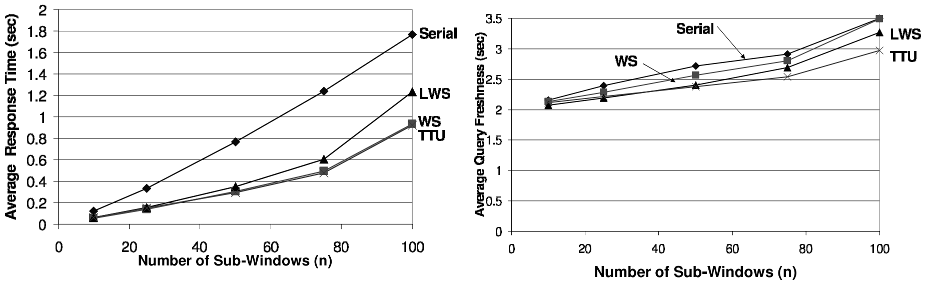
Fig. 8. Comparison of query inter-execution times for Serial, WS, LWS, and TTU

For instance, aborting every second re-execution of a long-running query means that its inter-execution time doubles. In general, increasing the sub-window size to $t = 5$ sec. (and hence, increasing the total window size) leads to longer inter-execution times for all four schedulers as the queries take longer to process. Similarly, increasing the number of sub-windows increases the query evaluation times and therefore negatively affects the inter-execution times. Overall, *Serial* yields the best query inter-execution times, with *WS* and *TTU* following closely behind, whereas *LWS* performs badly due to aborted transactions.

We briefly mention that throughput measurements revealed a very small penalty incurred by *TTU* versus *Serial*—typically below two percent and at most four percent. This is because the serialization graph testing done by *TTU* consists of simple bit operations after each window movement and causes negligible overhead. Furthermore, *TTU* did not abort any transactions in any of the tests. This is because during normal execution, a long-running query does not incur more than one concurrent window update, unless suspended for a long time in order to run a heavy workload of snapshot queries. Since Algorithm 2 ensures that read-only transactions postpone reading the sub-window that is about to be updated until the end, aborts can be easily avoided if the number of concurrent window updates is small. On the other hand, we found that the throughput of *LWS* was always lower than the other techniques because of a high proportion of aborted transactions. Full details may be found in [10].

5.3 Experiments with Long-Running and Snapshot Queries

Next, we report the results of experiments with a mixed workload of long-running and snapshot queries (and concurrent window movements). We fix the sub-window size at five seconds, the number of long-running queries at 100, and the number of snapshot queries per sub-window length at five. Snapshot queries are scheduled at random times with an average time between requests set to one



(a) (b)

Fig. 9. Comparison of snapshot query response time (a) and freshness (b) for Serial, WS, LWS, and TTU

second. We report the average snapshot query response time, and we separately measure the average freshness of snapshot and long-running queries.

Average snapshot query response times are illustrated in Fig. 9 (a). *TTU* and *WS* perform best and yield nearly identical response times. The response times of *LWS* are noticeably longer because it is forced to abort and restart some queries. *Serial* exhibits the worst results because it is unable to suspend a long-running query and execute a snapshot query immediately; in general, *Serial* is inappropriate for any situation involving prioritized scheduling. As the number of sub-windows increases, the response time achieved by each of the four schedulers worsens because it is now more costly to execute each query.

Figure 9 (b) plots the average snapshot query freshness. *TTU* outperforms the other schedulers because it guarantees latest-window serializability and did not abort any read transactions. The performance of *LWS* is somewhat worse because some of the transactions corresponding to snapshot queries are aborted and restarted at a later time. *WS* and *Serial* do not guarantee latest-window serializability and therefore exhibit the worst performance. Overall, *TTU* yields the best results in terms of snapshot query freshness and is tied for best in terms of the response time.

Finally, we separately examine the average freshness of long-running queries in order to verify that the performance edge of *TTU* in the context of snapshot query freshness does not come at a cost of poor long-running query freshness. We found that *TTU* maintains its superiority in producing the most up-to-date results of long-running queries (see [10] for full details).

6 Comparison with Related Work

The concurrency control mechanisms presented in this paper are compatible with any DSMS that employs periodic updates of sliding windows and query results, e.g., [2-8]. Our techniques are also applicable to a system such as PSoup [17], where mobile users connect to a DSMS intermittently and retrieve the latest results of sliding window queries. In our context, these asynchronous requests may be modeled as snapshot queries posed at various times. Given that mobile users may have low connectivity with the system (e.g., via a wireless channel), it is particularly important to guarantee low response times and up-to-date query answers. Our transaction scheduler fulfills both of these requirements.

As discussed in Sect. 2, we assumed an evaluation model in which queries are re-executed by scanning one or more windows or summaries, or a materialized sub-result. Similar techniques were used in [1, 3, 5]. Our procedure for incremental maintenance of materialized join results—using a batch of newly arrived tuples from one window to probe the other window and generate new results—is similar to the lazy multi-way join from [18]. In general, our query model of a final aggregation function applied on a window or materialized sub-result is similar to NiagaraCQ [2], but less expressive than, e.g., Aurora [6] and STREAM [7]. However, we believe that our model is sufficiently expressive for many applications that require long-running queries for monitoring purposes, while at the same time being simple enough to allow straightforward solutions of concurrency control issues.

Our transaction model resembles multi-level concurrency control and multi-granularity locking as it considers a sub-window, rather than an entire window, to be an atomic data object. The novelty of our solution is that the order in which read operations are performed is chosen in such a way as to minimize the number of aborted transactions.

Our transaction scheduler employed serialization graph testing. Other scheduling techniques include two-phase locking and timestamping [13]. However, two-phase locking may not be appropriate in our context because it is not clear how to force a particular serialization order using locks. Moreover, the possible problem with using timestamping for DSMS concurrency control is the difficulty of ensuring latest-window serializability. Suppose that each transaction receives a timestamp when it is passed to the transaction scheduler and that serialization order is determined by timestamps. In this case, any concurrent window update transaction is assigned a higher timestamp than a read-only transaction and is therefore serialized before the read-only transaction. Hence, Algorithm 2 would be forced to abort every read-only transaction that is interrupted by a window movement. A similar issue appears if we want to adapt multi-versioning concurrency control techniques to enforce latest-window serializability, among them snapshot isolation and commit-order preserving serializability [19].

7 Conclusions and Future Work

This paper presented DSMS concurrency control mechanisms that allow a window to slide forward while it, or an associated summary structure, is being scanned by a query. Our solution is based upon a model that views DSMS data access as a mix of concurrent read-only and write-only transactions. We proved that conflict serializability is insufficiently strong to guarantee correct and up-to-date query results, and defined more appropriate isolation levels. We also implemented a transaction scheduler for enforcing the new isolation levels that is provably optimal in reducing the number of aborted transactions. Our scheduler was experimentally shown to improve query freshness and response times while maintaining high transaction throughput.

We are interested in the following two directions for future work. First, we want to extend our query execution model and investigate concurrency control issues in query plans containing an arbitrary number of pipelined window operators. One issue in this context is synchronization among the levels in the pipeline, e.g., updates to the individual windows may take some time as they are propagated up the pipeline to the final query operator. Another problem appears when the same sub-query occurs more than once within a query, in which case our current assumption of queries scanning each window once may not hold (unless the sub-query can be flattened). Second, we want to extend our treatment of DSMS concurrency control to include the semantics of data loss and crash recovery, e.g., loss of data for a particular time interval, which might make it impossible for queries to read a full window.

References

1. Arasu, A., Widom, J.: Resource sharing in continuous sliding-window aggregates. In: Proc. VLDB Conference (2004) 336–347
2. Chen, J., DeWitt, D., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for Internet databases. In: Proc. SIGMOD Conference (2000) 379–390
3. Golab, L., Garg, S., Özsu, M.T.: On indexing sliding windows over on-line data streams. In: Proc. EDBT Conference (2004) 712–729
4. Shivakumar, N., García-Molina, H.: Wave-indices: indexing evolving databases. In: Proc. SIGMOD Conference (1997) 381–392
5. Zhu, Y., Shasha, D.: StatStream: Statistical monitoring of thousands of data streams in real time. In: Proc. VLDB Conference (2002) 358–369
6. Abadi, D., et al.: Aurora: A new model and architecture for data stream management. VLDB Journal **12** (2003) 120–139
7. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: Semantic foundations and query execution. VLDB Journal **14** (2005) to appear
8. Chandrasekaran, S., et al.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Proc. CIDR Conference (2003) 269–280
9. Golab, L., Özsu, M.T.: Update-pattern aware modeling and processing of continuous queries. In: Proc. SIGMOD Conference (2005) 658–669
10. Golab, L., Bijay, K.G., Özsu, M.T.: On concurrency control in sliding window queries over data streams. University of Waterloo Technical Report CS-2005-28. Available at `http://www.cs.uwaterloo.ca/research/tr/cs-2005-28.pdf`.
11. Cormode, G., Muthukrishnan, S.: An improved data stream summary: The count-min sketch and its applications. In: Proc. Latin American Theoretical Informatics Conference (LATIN) (2004) 29–38
12. Flajolet, P., Martin, G.N.: Probabilistic counting. In: Proc. FOCS Conference. (1983) 76–82
13. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
14. Belady, L.: A study of replacement algorithms for virtual storage computers. IBM Syst. J. **5** (1966) 78–101

15. Cormode, G., et al.: Holistic UDAFs at streaming speeds. In: Proc. SIGMOD Conference (2004) 35–46
16. Cranor, C., Johnson, T., Spatscheck, O., Shkapenyuk, V.: Gigascope: High performance network monitoring with an SQL interface. In: Proc. SIGMOD Conference (2003) 647–651
17. Chandrasekaran, S., Franklin, M.: PSoup: a system for streaming queries over streaming data. VLDB Journal **12** (2003) 140–156
18. Golab, L., Özsu, M.T.: Processing sliding window multi-joins in continuous queries over data streams. In: Proc. VLDB Conference (2003) 500–511
19. Weikum, G., Vossen, G.: Transactional Information Systems. Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kauffman (2002)