Resource Sharing in Continuous Sliding-Window Aggregates

Arvind Arasu Jennifer Widom

Stanford University
{arvinda,widom}@cs.stanford.edu

Abstract

We consider the problem of resource sharing when processing large numbers of continuous queries. We specifically address sliding-window aggregates over data streams, an important class of continuous operators for which sharing has not been addressed. We present a suite of sharing techniques that cover a wide range of possible scenarios: different classes of aggregation functions (algebraic, distributive, holistic), different window types (time-based, tuple-based, suffix, historical), and different input models (single stream, multiple substreams). We provide precise theoretical performance guarantees for our techniques, and show their practical effectiveness through experimental study.

1 Introduction

We consider *continuous-query* based applications involving a large number of concurrent queries over the same data. Examples of such applications include *publish-subscribe* systems (such as Traderbot [29]) that allow a large number of users to independently monitor *published* information of interest using *subscriptions*. Another example is intrusion detection, where a large number of *rules* are used to continuously monitor system and network activity [24, 30]. In these applications, subscriptions and rules are continuous queries.

Handling each continuous query separately is inefficient, and may be infeasible for large numbers of queries and high data rates. Queries must be handled *collectively*, by exploiting similarities and *sharing resources* such as computation, memory, and disk bandwidth among them. Numerous papers [8–10, 14, 25] have highlighted the importance of resource sharing in continuous queries.

One avenue for resource sharing is based on detecting and exploiting common subexpressions in queries, related to traditional multi-query optimization [27, 28]. However, we are interested in more basic sharing—at the operator level. If several queries use different instances of operators belonging to a same class, it is sometimes possible to process all the operator instances more efficiently using a single generic operator. A good example of operator level sharing is processing many range predicates using a single *predicate index* [11, 16, 25]. The predicate index identifies all the indexed range predicates that a given tuple satisfies more efficiently than the naïve method that checks each predicate separately.

Previous work on operator-level sharing has focused primarily on filters, which are stateless and have simple semantics. In this paper, we address operator-level sharing for an important class of operators based on *aggregations over sliding windows*, detailed in the next subsection. The importance of these operators has been identified before [12], but the sharing problem is largely unstudied. (The problem is considered just briefly in PSoup [10]. PSoup is discussed in detail in Section 3.1.3.) We show that there exist significant opportunities for sharing in sliding-window aggregates, and by exploiting them we obtain dramatic improvements in performance over the naïve, non-sharing approaches.

1.1 Sliding-Window Aggregates

Continuous queries typically work on unbounded *data streams*, rather than static data sets. For many applications, recent elements of a stream are more important than older ones. This preference for recent data is commonly expressed using *sliding windows* over a stream [7]. The size of a window is often specified using a time interval (*time-based*), but may use number of tuples instead (*tuple-based*). The window usually ends at the current time, called a *suffix* window, but nonsuffix (*historical*) windows may also be used.

An *aggregation over a sliding window (ASW)* operator continuously applies an aggregation function over the contents of a sliding window. The aggregation value changes over time as the window slides. ASW is widely recognized as a fundamental operation over streams, and is the subject of numerous previous papers [4, 10, 12, 18].

Example 1.1 Consider a stream of stock trades, and two ASW operators. One operator asks for the "average number of shares in the last 10,000 stock trades" (a suffix, tuple-based sliding window), while the other asks for "average number of shares in trades between 1 and 2 minutes

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

ago" (a time-based historical window). Assuming the windows may overlap, using our techniques we can share state and computation between these two operators. In practice our techniques are most important when there are hundreds or thousands of such operators, as would occur in Traderbot [29] or other publish-subscribe systems. □

Although historical windows are less frequent than suffix windows, most of our algorithms are no simpler for the suffix-only case. In those cases where suffix-only is simpler, we explicitly say so (e.g., Algorithm L-INT in Section 3.1.2).

1.1.1 Sliding-Window Aggregates on Substreams

Some streams can be partitioned logically into *substreams* [31], each identified by a *key*. For example, our stock trade stream can be partitioned into substreams based on the stock symbol (key). Similarly, a stream of network packets can be partitioned into substreams, one per *flow* [15].

With partitioning, a useful primitive is the application of an ASW operator over each substream [31]. This operation conceptually produces a dynamic relation: at any point in time, there is one tuple for each substream in the relation, containing the current answer of the ASW operator for that substream. The sharing techniques that we develop for general ASW operators are applicable for substreams as well. However, we show that when an ASW operator over substreams is followed by a range predicate over the conceptual dynamic relation—a class of operators we call *SubStream Filters (SSF)*—there are additional sharing possibilities.

Example 1.2 Consider the class of operators "find all stocks whose trading volume during a certain sliding window exceeds a certain threshold." These are SSF operators. If we have many such operators, with different thresholds and windows, our techniques let us share resources by maintaining a single index-like structure over all the different ASW operators and range predicates. □

1.1.2 Additional Examples

We briefly illustrate that realistic queries are composed of aggregations over sliding windows. The following two queries were taken directly from the Traderbot site [29].

1. **NASDAQ Short-term Downward Momentum:** Find all NASDAQ stocks between $20 and $200 that have moved down more than 2% in the last 20 minutes and there has been significant buying pressure (70% or more of the volume has traded toward the ask price) in the last 2 minutes.

2. **High Volatility with Recent Volume Surge:** Find all stocks between $20 and $200 where the spread between the high tick and the low tick over the past 30 minutes is greater than 3% of the last price and in the last 5 minutes the average volume has surged by more than 300%.

Both queries apply ASW operators over stock substreams. Query 1 uses a `SUM` aggregate over a 2-minute suffix sliding window. Query 2 uses `MAX` (high-tick), `MIN` (low-tick), and `AVG` aggregates over two different windows ("last 5 minutes" and "last 30 minutes"). Various relational operators like filters and joins are performed over the dynamic relations output by the ASW operators.

1.1.3 Output Model

We assume that an ASW or SSF operator does not actively stream its current answer, but instead produces answers only when requested. We call such a request an *answer lookup* (or simply *lookup*). We chose the lookup model over the streaming model for several reasons:

1. The lookup model is more general, i.e., it can emulate the streaming model.

2. In many cases, the current ASW result is needed at most when new stream tuples arrive, and certainly not at every time instant (e.g., correlated aggregates [17]).

3. Reference [10] cites several applications that prefer to periodically refresh continuous query answers, rather than keep them fully up-to-date.

1.2 Space-Update-Lookup Tradeoff and Our Approach

Any algorithm for processing a large number of continuous queries with the lookup model involves three *cost parameters*: the memory required to maintain state (*space*), the time to compute an answer (*lookup time*), and the time to update state when a new stream tuple arrives (*update time*). There is a tradeoff among these three costs and generally no single, optimal solution.

For example, we can make lookups efficient by maintaining up-to-date answers for all queries. However, this approach has a high update cost, since arrival of a new stream tuple potentially requires updating answers of all the queries, and a large space requirement, since current answers for all the queries need to be stored. Alternatively, we can maintain a single historical snapshot of the input that is large enough to compute the answer for any given query, but defer actual answer computation to lookup time. This approach has small update and space costs, but potentially high lookup cost.

These two approaches are appropriate only for the extreme scenarios—very high update rates compared to lookups, or vice-versa. Many applications lie between the extremes. Our techniques are designed to capture a wide range of these "in between" scenarios, by performing partial answer computation at update time and using the partial results to compute the final answer at lookup time. Furthermore, our partial answer schemes are designed specifically so that partial answers can be shared by a large number of queries.

Operator Type	Aggregation Function	Update Time	Lookup Time	Space
ASW	Algebraic, Distributive	$O(1)$	$O(\log W)$	$O(N_{max})$
ASW	Algebraic, Distributive	$O(\gamma)$	$O(1)$	$O(\gamma N_{max})$
ASW	Subtractable	$O(1)$	$O(1)$	$O(N_{max})$
ASW	Holistic: `QUANTILE`	$O(\log N_{max})$	$O(\log^2 W \log\log W)$	$O(N_{max} \log N_{max})$
SSF	`COUNT`	$O(\frac{1}{\epsilon} \log \lvert\mathcal{K}\rvert)$	$O(\lvert\mathcal{K}_o\rvert)$	$O(\frac{1}{\epsilon} \log \lvert\mathcal{K}\rvert \log N_{max})$
SSF	`SUM`	$O(\frac{1}{\epsilon} \log \lvert\mathcal{K}\rvert \log A_m)$	$O(\lvert\mathcal{K}_o\rvert)$	$O(\frac{1}{\epsilon} \log \lvert\mathcal{K}\rvert \log(N_{max} A_m))$
SSF	`MAX`, `MIN`	$O(\log^2 N_{max})$	$O(\log^2 N_{max} + \lvert\mathcal{K}_o\rvert)$	$O(N_{max} \log N_{max})$

Figure 1: Summary of results. N_{max} = Max left end of a window; W = width of lookup window; $|\mathcal{K}|$= # substreams; $|\mathcal{K}_o|$= output size for SSF operators; ϵ = error parameter; γ = "spread" of window widths; A_m = Max value of aggregated attribute.

1.3 Summary of Contributions

General

Our high-level contribution is that of identifying the resource sharing possibilities in sliding-window aggregations and devising a suite of algorithms to exploit them. We precisely formulate the sharing problem for this class of operators, indicate the basic tradeoffs involved, and study techniques for a wide variety of scenarios. While many of the sharing techniques are our original contributions, some are a synthesis of existing ideas from other fields such as data structures and computational geometry. The entire suite of results is summarized in Table 1.

ASW Operators

For ASW operators, we present general sharing techniques based on properties of aggregation functions. As in [20], we divide aggregation functions into *distributive*, *algebraic*, and *holistic*.

1. For distributive and algebraic aggregates, we present two techniques: one has low update and space costs, while the other has low lookup cost.

2. We identify a subclass of distributive and algebraic aggregates, which we call *subtractable*, and show that they can be handled more efficiently than the general class.

3. Since there is no common property for the class of holistic aggregates (they are defined as aggregates that are not algebraic), we cannot have a general technique. However, we present sharing techniques for the well-known holistic `QUANTILE` aggregate.

SSF Operators

For SSF operators, one simple approach is to use ASW sharing techniques for processing the ASW suboperators corresponding to each substream. At SSF lookup time, we perform an ASW lookup for each substream and return those substreams that satisfy the range predicate. Therefore, the SSF lookup operation depends linearly on the number of substreams. For specific aggregation functions (`COUNT`, `SUM` over positive values, `MAX`, `MIN`) and window types (suffix, time-based), we show that we can achieve a lookup cost that is sublinear in the number of substreams, without significantly increasing update cost. Our techniques for `SUM` and `COUNT` are *approximate*: they slightly relax the range predicate. However the approximation is controlled by a parameter ϵ, which can be made as small as desired at the expense of increased space and update time. Further, exact answers can be obtained by postprocessing.

1.4 Outline of Paper

Section 2 presents formal definitions and notation. Sections 3 and 4 present our algorithms for resource sharing in ASW and SSF operators, respectively. Section 6 contains a thorough experimental evaluation of the algorithms. Section 7 covers related work, and Section 8 contains our conclusions and directions for future work.

All proofs and some of the algorithms (ASW algorithms for time-based windows, SSF algorithms for `MAX` and `MIN`) have been omitted due to space constraints. These can be found in the full version of the paper [5].

2 Formal Preliminaries

Streams and Substreams

A *stream* S is a sequence of tuples arriving at a continuous query processing system. Tuples of S are timestamped on arrival using the system clock. At a given point in time, we refer to the number of tuples of S that have arrived so far as the *current length* of S.

Stream S may be partitioned into *substreams* based on a set of *key attributes*. We assume a single key attribute; generalizing is trivial. Let K denote the key attribute and $\mathcal{K}$ the domain of K. Every value $k \in \mathcal{K}$ identifies a unique substream, which we denote S_k, and k is called the *key* of S_k. All tuples of S with a value k for attribute K belong to the substream S_k, and their relative ordering within S_k is the same as their relative ordering in S.

Sliding Windows

Consider a stream S with the sequence of tuples $s_1, s_2, \ldots$ and corresponding timestamps $\tau_1, \tau_2, \ldots$. For any two nonnegative integers $N_L > N_R$, $S[N_L, N_R]$ denotes a tuple-based sliding window over S: When the current length of S is r, $S[N_L, N_R]$ contains the set of tuples $\{s_i \mid \max\{r - N_L + 1, 1\} \leq i \leq (r - N_R)\}$. Similarly, for any two time

periods $T_L > T_R$, $S[T_L, T_R]$ denotes a time-based sliding window over S: When the current time is τ, $S[T_L, T_R]$ contains the set of tuples $\{s_i \mid (\tau - T_L + 1) \leq \tau_i \leq (\tau - T_R)\}$. Note that we use the same notation for time-based and tuple-based windows. The type of window is usually clear from context and our different conventions for numbers (e.g., N, N_L, N_i) and time intervals (e.g., T, T_L, T_i). Whenever the window type is not important to the discussion, we abuse this naming convention further and denote a generic window over S by $S[X_L, X_R]$.

Aggregation Functions

We use the classification from [20] that divides aggregation functions into three categories: *distributive*, *algebraic*, and *holistic*. Let X, X_1, and X_2 be arbitrary bags of elements drawn from a numeric domain. An aggregation function f is *distributive* if $f(X_1 \cup X_2)$ can be computed from $f(X_1)$ and $f(X_2)$ for all X_1, X_2. An aggregation function f is *algebraic* if there exists a "synopsis function" g such that for all X, X_1, X_2: (1) $f(X)$ can be computed from $g(X)$; (2) $g(X)$ can be stored in constant memory; and (3) $g(X_1 \cup X_2)$ can be computed from $g(X_1)$ and $g(X_2)$. An aggregation function is *holistic* if it is not algebraic. Among the standard aggregates, `SUM`, `COUNT`, `MAX`, and `MIN` are distributive, `AVG` is algebraic, since it can be computed from a synopsis containing `SUM` and `COUNT`, and `QUANTILE` is holistic.

ASW and SSF Operators

An ASW operator over a stream S with window specification $[X_L, X_R]$ and aggregation function f over attribute $S.A$ is denoted $f_A(S[X_L, X_R])$. At any point in time, its current answer is obtained by applying the aggregation function f over the values of attribute A of all the tuples that are currently in the window $S[X_L, X_R]$. An SSF operator is denoted by: $\{k \in \mathcal{K} \mid f_A(S_k[X_L, X_R]) \in (v_1, v_2)\}$, where $f_A(S_k[X_L, X_R])$ denotes the ASW operator that is applied to each S_k and (v_1, v_2) denotes the predicate range.

Example 2.1 The ASW operator $\texttt{AVG}_A(S[120, 0])$ continuously computes the average value of attribute A over S tuples in the last 120 time units. The SSF operator $\{k \in \mathcal{K} \mid \texttt{SUM}_A(S_k[300, 0]) \in (1000, \infty)\}$ continuously computes all substreams for which the sum over attribute A values in the last 300 time units is greater than or equal to 1000. Using a SQL-like syntax (e.g., in CQL [3]) this SSF operation is expressed as follows:

```
Select K, SUM (A)
From S [Range 300]
Group By K
Having SUM(A) > 1000
```

3 ASW Operators

In this section, we present our algorithms for collectively processing a set of ASW operators. Sharing resources is not possible between operators over different streams, or between operators with different aggregated attributes, since their input data is completely different: there is no benefit to processing them collectively. Sharing is sometimes possible between operators with different aggregation functions (e.g., `AVG` and `SUM`) over the same input stream and aggregated attribute. However, for presentation clarity, we do not address this special case.

Therefore, our algorithms are designed for collectively processing a set of ASW operators with the same input stream, aggregation function, and aggregated attribute. The only difference between different operators is the sliding window specification. One exception is the `QUANTILE` aggregation function, where we allow the quantile parameter to be different; see Section 3.3.

Due to space constraints, we only present algorithms for the case where all the windows are tuple-based. Algorithms that handle time-based windows are described in the full version of the paper [5].

Notation: For the rest of this section, let $o_1, \ldots, o_n$ denote the input set of operators. Let S denote the common input stream, f the common aggregation function, and A the common aggregated attribute. For these operators, only the sequence of attribute A values are relevant, which we denote $a_1, a_2, \ldots$. We call each a_i a stream *element*. Further, let each $o_i = f_A(S[N_{Li}, N_{Ri}])$.

Intervals: The notion of an *interval* over positions of S is useful to describe our algorithms. The interval $I = (l, r)$ $(l \leq r)$ denotes the positions $l, l+1, \ldots, r$ of S, and the elements $a_l, \ldots, a_r$ *belong to* interval I. For an interval I, $f(I)$ denotes the aggregation over the elements of S belonging to I.

For each algorithm, we specify: (1) the state that it maintains; (2) an operation $\textsc{Update}(a_{m+1})$ that describes how the state is updated when element a_{m+1} arrives; and (3) an operator $\textsc{Lookup}(I)$ that describes, for certain intervals I, how $f(I)$ can be computed using the current state. $\textsc{Lookup}(I)$, as the name suggests, is used to perform answer-lookups for operators o_i: when the current size of S is m, the current answer for o_i can be obtained using $\textsc{Lookup}(I)$ where $I = (m - N_{Li} + 1, m - N_{Ri})$. Therefore, for correctness, when the current length of S is m, we require that $\textsc{Lookup}(I)$ correctly compute $f(I)$ for all intervals $I = (m - N_{Li} + 1, m - N_{Ri})$ $(1 \leq i \leq n)$; it may or may not compute $f(I)$ correctly for other intervals.

3.1 Distributive, Algebraic Aggregates

In this section, we present two algorithms, B-INT and L-INT, for the case where f is distributive or algebraic. For presentation clarity, we assume that f is distributive; the generalization to the algebraic case is straightforward.

Both algorithms are based on a simple, but fairly general approach: For certain intervals I, precompute $f(I)$ and store it as part of the state. The basic intuition behind this step is that, since f is distributive, the precomputed aggregate values can be used to compute lookups more efficiently. For example, $f(101, 200)$ and $f(201, 300)$ can be used to compute $f(101, 300)$, and therefore, $\textsc{Lookup}(101, 300)$. More generally, $f(I)$ can

potentially help compute LOOKUP(I') for any I' that contains I.

For what intervals I should we precompute $f(I)$? Selecting more intervals for precomputation is likely to improve lookup efficiency, but at the cost of space and update time—a manifestation of the space-lookup-update tradeoff discussed in Section 1.2. Also, any precomputed aggregate $f(I)$ loses its utility eventually, once all the windows of o_i slide past I. (In fact, the answer to this question of which intervals to precompute is not very obvious even for processing a single operator, i.e., $n = 1$.)

Next we present our two algorithms, which are essentially two different schemes for dynamically selecting the intervals I to precompute, along with the details of how $f(I)$ aggregates for selected intervals are (pre)computed and used for lookups.

3.1.1 B-INT Algorithm

Our first algorithm is called B-INT (for *Base-Intervals*), since it precomputes aggregate values $f(I_b)$ for intervals I_b that belong to a special class called *base-intervals*. Intuitively, base-intervals form a "basis" for intervals: any interval can be expressed as a disjoint union of a small number of base-intervals. Using this property, any $f(I)$ can be computed using a small number of precomputed $f(I_b)$ values. At any point in time, B-INT stores $f(I_b)$ values for only recent or "active" base-intervals—only these are potentially useful for future lookups of the operators $o_1, \ldots, o_n$.

Figure 2 abstractly illustrates the state maintained by algorithm B-INT, when the current length of S is m. The active base-intervals, for which the B-INT precomputes aggregate values, are shown as solid rectangles. The base-intervals which are not active are shown using dotted rectangles. The figure also shows how the aggregate value for a lookup interval is computed using precomputed aggregates for active base-intervals.

Definition 3.1 (Base-Interval) An interval I_b is a base-interval if it is of the form $(2^\ell i + 1, 2^\ell(i+1))$ for some integer $i \geq 0$, in which case it is called a *level-ℓ* base-interval. □

For example, $(385, 512) = (2^7 \cdot 3 + 1, 2^7 \cdot 4)$ is a level-7 base-interval. A level-ℓ base-interval has a width 2^ℓ and is a disjoint union of exactly two level-$(\ell - 1)$ base-intervals.

Base intervals turn out to be the same concept as *dyadic intervals*, which have been used in [19] for approximate quantile computation over updatable relations. In the context of sliding windows, in related work [4] we have used base (dyadic) intervals for computing approximate statistics over sliding windows.

The following theorem, whose proof is straightforward, formally states that any interval can be expressed as a union of a small number of base-intervals.

Theorem 3.1 *Any interval $I = (l, r)$ of width $W = (r - l + 1)$ can be expressed as a disjoint-union of $k = O(\log W)$ base-intervals of the form $I_{bi} = (l_i, r_i)$ $(1 \leq i \leq k)$, where $l_1 = l$, $r_k = r$, and $r_i = l_{i+1} - 1$, $(1 \leq i < k)$. Given interval I, the intervals $I_{b1}, \ldots, I_{bk}$ can be determined in $O(k) = O(\log W)$ time.*

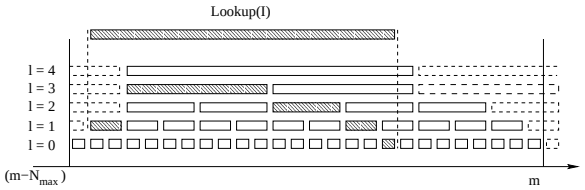

Figure 2: Base-intervals used by B-INT when current length of S is m

STATE:
For each currently active base-interval I_b, store $f(I_b)$.

UPDATE (a_{m+1}):
1. Compute and store $f(a_{m+1})$.
2. Let $z =$ number of trailing 0's in binary representation of $m + 1$.
3. For $i = 1$ to z do
   /* assert: $(m-2^i+2, m+1)$ is a base-interval */
4. If $(l, r) = (m - 2^i + 2, m + 1)$ is active
5. Compute $f(l, r)$ using $f(l, \frac{l+r-1}{2})$ and $f(\frac{l+r+1}{2}, r)$ and store it.
6. For each I_b that ceases to be active, discard $f(I_b)$.

LOOKUP(I):
1. Express I as a union of base-intervals $I_{b1}, \ldots, I_{bk}$ as in Theorem 3.1.
2. Compute and return $f(I)$ using $f(I_{b1}), \ldots, f(I_{bk})$.

Figure 3: Algorithm B-INT

Example 3.1 The interval $(1, 43)$ can be expressed as a union of base-intervals $(1, 2)$, $(3, 4)$, $(5, 8)$, $(9, 16)$, $(17, 32)$, $(33, 40)$, $(41, 42)$, $(43, 43)$. □

Active Intervals: Let $N_{max} = \max_i(N_{Li})$ denote the "earliest" left end of a window in $o_1, \ldots, o_n$. When the current size of S is m, we call an interval $I = (l, r)$ *active* if $(l > m - N_{max})$ and $(r \leq m)$. Intuitively, an interval is active at some point of time, if it is completely within the last N_{max} positions of the stream.

Figure 3 contains the formal description of B-INT. When the current size of S is m, LOOKUP(I) computes $f(I)$ for all intervals $I = (m - N_{Li} + 1, m - N_{Ri})$ $(1 \leq i \leq n)$ that correspond to lookups of operators $o_1, \ldots, o_n$. By definition, any such interval $(m - N_{Li} + 1, m - N_{Ri})$ is active, and therefore each interval $I_{b1}, \ldots, I_{bk}$ in Step 1 of LOOKUP(I) is active as well, implying that $f(I_{bi})$ is stored as part of the state.

Conceptually, UPDATE(a_{m+1}) computes $f(I_b)$ for all base-intervals I_b that become newly active and adds it to the current state (Steps 1–5), and discards $f(I_b)$ for all intervals that cease to be active (Step 6). UPDATE(a_{m+1}) always introduces at least one new base-interval: (a_{m+1}, a_{m+1}). In general, if 2^z denotes

the largest power of 2 that divides $(m+1)$, then UPDATE(a_{m+1}) introduces $z+1$ new base-intervals. One obvious technique for computing $f(I_b)$ is to do so from scratch, using the elements of S that belong to I_b. A more efficient technique is to compute it recursively, using aggregate values corresponding to base-intervals of the next lower-level, as shown in Step 5 of UPDATE(a_{m+1}).

Theorem 3.2 *Algorithm* B-INT *requires $O(N_{max})$ space, has an amortized update time complexity of $O(1)$, and has a worst-case lookup time complexity of $O(\log W)$, where W denotes the width of the lookup interval.*

3.1.2 L-INT Algorithm

Our second algorithm for distributive and algebraic aggregates, called L-INT (for *Landmark Intervals*), uses an interval scheme based on certain *landmarks* or specific positions of the stream. L-INT is more efficient than B-INT for lookups, but its update and space costs are higher. Further, L-INT is more input-specific: while B-INT depends only on N_{max}, L-INT, in addition to N_{max}, depends on the distribution of window widths.

We first present L-INT for a special case, where all the window widths are close to equal. Specifically, we assume that they are within a small constant factor c of each other, i.e., $W_{max}/W_{min} \leq c$, where $W_{max} = \max_i(N_{Li} - N_{Ri})$ denotes the maximum width, and $W_{min} = \min_i(N_{Li} - N_{Ri})$, the minimum width of a window. For this special case, L-INT is optimal: it uses $O(N_{max})$ space and has $O(1)$ update and lookup time. (Clearly, we cannot do better than $O(1)$ update and lookup time and $O(N_{max})$ space.) Then, we show how we can extend L-INT to handle the general case.

Definition 3.2 (**Landmark Interval**) Landmark intervals are defined for two width parameters $W_{min} \leq W_{max}$. A landmark interval is of the form $(\alpha W_{min}, \alpha W_{min} + d)$ or of the form $(\alpha W_{min} - d, \alpha W_{min} - 1)$, for some $\alpha \geq 0$ and $d \leq W_{max}$. □

We call stream positions of the form αW_{min} $(\alpha \geq 0)$ *landmarks*. A landmark interval is one that begins at or ends just before a landmark, and has a width less than W_{max}. For example, if $W_{max} = 2000$ and $W_{min} = 1000$, intervals $(75, 999)$, $(1762, 2999)$, and $(5000, 6542)$ are landmark intervals, while $(5000, 7162)$ is not, since its width is greater than 2000. Figure 4 schematically illustrates landmark intervals that begin or end at the landmark αW_{min}.

The following theorem states that any interval with width between W_{min} and W_{max} can be expressed as a union of at most two landmark intervals.

Theorem 3.3 *Any interval $I = (l, r)$, such that $W_{min} \leq (r - l + 1) \leq W_{max}$, can be expressed as a disjoint union of at most two landmark intervals defined for parameters W_{min} and W_{max}.*

Example 3.2 Let $W_{max} = 2000$ and $W_{min} = 1000$. The interval $(3257, 5164)$ can be expressed as a union of landmark intervals $(3257, 3999)$ and $(4000, 5164)$. □

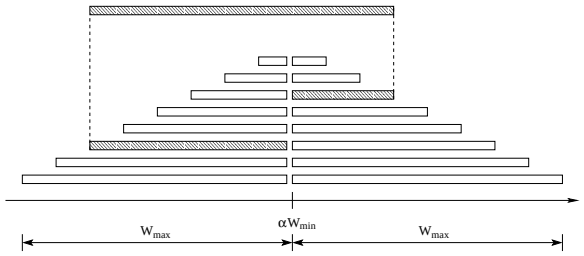

Figure 4: Landmark intervals that begin or end at the landmark αW_{min} for some $\alpha \geq 0$

STATE:
1. For each currently active landmark interval I_l, store $f(I_l)$.
2. For each currently active element a_i, store $f(a_i)$.

UPDATE (a_{m+1}):
1. Compute and store $f(a_{m+1})$.
2. If $m+1 = \alpha W_{min}$
3. For $d = 2$ to W_{max}
4. Compute $f(m+1-d, m)$ from $f(m+2-d, m)$ and $f(a_{m+1-d})$ and store it.
5. For each β such that $m+1-W_{max} \leq \beta W_{min} \leq m$
6. Compute $f(\beta W_{min}, m+1)$ from $f(\beta W_{min}, m)$ and $f(a_{m+1})$
7. Drop $f(I_l)$ for intervals I_l that cease being active.

LOOKUP(I):
1. Express I as a union of landmark intervals I_{l1} and I_{l2} as in Theorem 3.3
2. Compute and return $f(I)$ using $f(I_{l1})$ and $f(I_{l2})$.

Figure 5: Algorithm L-INT for $W_{max}/W_{min} \leq c$

Figure 5 contains the formal description of L-INT for the special case of $W_{max}/W_{min} \leq c$. Using a reasoning similar to that for algorithm B-INT, we can argue that algorithm L-INT is correct, i.e., it can be used to compute lookups corresponding to the operators $o_1, \ldots, o_n$. The update operation is also similar to that of B-INT: it computes $f(I_l)$ for all landmark intervals I_l that newly become active, and discards $f(I_l)$ for intervals I_l that cease to be active.

Theorem 3.4 *Algorithm* L-INT *presented in Figure 5 requires $O(N_{max})$ space, has an amortized update time of $O(1)$, and has a worst case lookup time of $O(1)$.*

Extending L-INT algorithm for the general case is straightforward: partition the set of operators $o_1, \ldots, o_n$ into γ partitions $P_1, P_2, \ldots, P_\gamma$, such that for the operators belonging to each partition, the property $W_{max}/W_{min} \leq c$ is satisfied. Use γ instances of the special-case version of the L-INT algorithm (Figure 5) to process these partitions independently. This extended algorithm requires $O(\gamma N_{max})$ space, has an update cost of $O(\gamma)$ and a lookup cost of $O(1)$.

For any set of operators $o_1, \ldots, o_n$, we can always define a partitioning scheme with $\gamma = O(\log N_{max})$. How-

ever, for many real-world applications, it seems natural to expect the window widths to be clustered around a few values. For such applications, γ could be significantly smaller than $\log N_{max}$.

Further, if all the operators $o_1, \ldots, o_n$ have suffix windows, or even "approximately" suffix windows, we can reduce the space required from $O(\gamma N_{max})$ to $O(N_{max})$. A tuple-based window $[N_L, N_R]$ is approximately suffix if $(N_L - N_R)$ is comparable in value to N_L.

3.1.3 PSoup Algorithm

PSoup [10] proposes a different algorithm for distributive and algebraic aggregates, that uses an augmented n-ary search tree. Each leaf of the search tree contains an element of the stream (only the last N_{max} elements need to be stored), and the leaves are ordered based on insertion times of the elements. Each internal node stores the value of the aggregation function computed over the descendant leaves (elements) of that node. This algorithm has an update and lookup cost of $O(\log N_{max})$. Both B-INT and L-INT algorithms perform asymptotically better than PSoup for at least one of update or lookup operations. Also, note that the lookup cost of B-INT depends only on the window size W of the operator involved in the lookup (the lookup cost is $O(\log W)$), and is independent of N_{max}. In Section 6, we compare empirically the performance of PSoup against our algorithms.

3.2 Subtractable Aggregates

An algebraic aggregation function f is *subtractable* if its synopsis function g has the following property: for any bags $X_1 \subseteq X_2$, $g(X_2 - X_1)$ is computable from $g(X_1)$ and $g(X_2)$. Among the standard algebraic aggregation functions `SUM`, `COUNT`, and `AVG` are subtractable, while `MAX` and `MIN` are not. For instance, `SUM` is subtractable since $\texttt{SUM}(X_2 - X_1) = \texttt{SUM}(X_2) - \texttt{SUM}(X_1)$, if $X_1 \subseteq X_2$.

For subtractable aggregates, we present a simple algorithm called R-INT (for *running intervals*) that has $O(1)$ update and lookup cost. For presentation clarity, we assume that f is subtractable and distributive, i.e., $f(X_2 - X_1)$ is computable from $f(X_2)$ and $f(X_1)$, whenever $X_1 \subseteq X_2$. Generalizing to the case where f is algebraic and subtractable is straightforward.

A *running interval* is an interval of the form $(1, r)$, whose left end is at the beginning of S. Any interval (l, r) can be expressed as a difference of two running intervals: $(1, r) - (1, l-1)$. R-INT (Figure 6) is based on this observation. R-INT stores aggregate values corresponding to currently active running intervals, and uses these to compute lookup answers. When the current length of S is m, a running interval $(1, r)$ is *active*, if $m - N_{max} \leq r \leq m$. It can be easily shown that R-INT uses $O(N_{max})$ space and has $O(1)$ update and lookup time.

3.3 Quantiles

The quantile aggregation function is specified using a parameter $\phi \in (0, 1]$ and is denoted $\texttt{QUANTILE}(\phi)$. The output of $\texttt{QUANTILE}(\phi)$ for a bag of N elements is the element at position $\lfloor \phi \cdot N \rfloor$ in a sorted sequence of these elements.

STATE:
1. For each currently active running interval I_r, store $f(I_r)$.

UPDATE (a_{m+1}):
1. Compute $f(1, m+1)$ using $f(1, m)$ and $f(a_{m+1})$.
2. For each I_r that is no longer active, discard $f(I_r)$.

LOOKUP$(I = (l, r))$:
1. Compute and return $f(l, r)$ using $f(1, l-1)$ and $f(1, r)$.

Figure 6: Algorithm R-INT

We briefly sketch one algorithm (called B-INT-QNT) for processing ASW operators with quantiles (possibly with different parameters ϕ) that uses the base-intervals defined in Section 3.1.1: Corresponding to each active base-interval I_b store a sorted array of the elements of S that belong to I_b. To perform LOOKUP(I) for a quantile parameter ϕ, express I as a union of base-intervals $I_{b1}, \ldots, I_{bk}$, using Theorem 3.1, and compute $\texttt{QUANTILE}(\phi)$ using the sorted arrays corresponding to $I_{b1}, \ldots, I_{bk}$. In general, given a set of p sorted arrays of length $\leq q$, we can compute any quantile over all the elements in the sorted arrays in $O(p \log p \log q)$ time, using a greedy algorithm.

4 SSF Operators

We now consider the problem of processing a collection of SSF operators $o_1, \ldots, o_n$. As in Section 3, we assume that all operators have the same input stream S, the same aggregation function f, the same aggregated attribute A. So they differ only in their window specification and range predicate. Therefore each operator o_i has the form $\{k \in \mathcal{K} \mid f_A(S_k[X_{Li}, X_{Ri}]) \in (v_{li}, v_{hi})\}$.

A simple strategy for processing these operators is as follows: For each substream S_k, process the set of ASW suboperators $f_A(S_k[X_{Li}, X_{Ri}])$ $(1 \leq i \leq n)$ using the algorithms of Section 3. To perform a lookup for operator o_i, perform a lookup on the ASW suboperator $f_A(S_k[X_{Li}, X_{Ri}])$ for each substream S_k, and return those substreams for which the lookup output lies within (v_{li}, v_{hi}). Clearly, the SSF lookup cost for this approach depends linearly on $|\mathcal{K}|$, the number of substreams.

In this section, we present algorithms for certain combinations of aggregation functions (`COUNT`, `SUM` over positive values) and window types (suffix, time-based) that have lookup cost sublinear in $|\mathcal{K}|$. (The full version of the paper [5] contains algorithms for `MAX` and `MIN` for suffix, time-based windows.) We call our algorithms CI-COUNT and CI-SUM. CI stands for *collective index*) since, conceptually, the algorithms can be thought of as a collection of search indexes, one for each ASW suboperator.

Notation: Throughout this section, let $a_{k1}, a_{k2}, \ldots$ denote the sequence of attribute A values for substream S_k. As before, we call them *elements* of S_k. Let $\tau_{k1}, \tau_{k2}, \ldots$ denote the timestamps of these elements. As in Section 3, for each algorithm we present: (1) the state that it maintains; (2) an operation UPDATE(a_{km}, τ_{km}) that describes

how the state is modified when element a_{km} with timestamp τ_{km} arrives on substream S_k; and (3) an operation LOOKUP$(\tau, T, (v_1, v_2))$ that describes how the current answer for the SSF operator $\{k \in \mathcal{K} \mid f_A(S_k[T, 0]) \in (v_1, v_2)\}$ can be computed using the current state.

4.1 CI-COUNT

CI-COUNT is an approximate algorithm for processing a collection of SSF operators of the form $o_i = \{k \in \mathcal{K} \mid \texttt{COUNT}_A(S_k[T_i, 0]) \in R\}$, where R is a one-sided range condition of the form (v, ∞) or $(0, v)$. The approximation produced by CI-COUNT for LOOKUP(τ, T, R) is as follows. If K_{co} denotes the correct output, the output K_{ao} produced by CI-COUNT has the following guarantees:

1. The approximate output K_{ao} is a superset of the exact output K_{co}.

2. The current ASW answer for each substream in the approximate output satisfies a relaxed version of the range condition R. Specifically, if R is of the form (v, ∞), for every key $k \in K_{ao}$, $\texttt{COUNT}_A(S_k[T, 0]) \in (v(1-\epsilon), \infty)$, for some *approximation parameter* $\epsilon \in (0, 1)$. Similarly, if R is of the form $(0, v)$, for every key $k \in K_{ao}$, $\texttt{COUNT}_A(S_k[T, 0]) \in (0, v(1+\epsilon))$.

The approximation parameter ϵ in the above guarantees can be made as small as desired, but decreasing ϵ increases the required space and update cost: both grow linearly in $\frac{1}{\epsilon}$.

Although CI-COUNT supports only approximate lookups, it can be used along with our ASW algorithms for performing exact lookups. Specifically, we can compute the correct answer K_{co} from the approximate output K_{ao} by checking for each $k \in K_{ao}$ whether $\texttt{COUNT}_A(S_k[T, 0]) \in R$ using an ASW-lookup.

We first present a non-parameterized version of CI-COUNT that yields a fixed $\epsilon = 0.75$, and then describe how this algorithm is modified to produce the parameterized version. Also, for clarity, we assume that the range conditions of all the operators are of the form (v, ∞). Handling range conditions of the form $(0, v)$ is a straightforward generalization. In the rest of this subsection, we abbreviate LOOKUP$(\tau, T, (v, \infty))$ as LOOKUP(τ, T, v).

4.1.1 Non-Parameterized CI-COUNT

To get an intuition for CI-COUNT, consider the LOOKUP(τ, T, v). This operation seeks all substreams S_k which have received more than v elements in the last T time units. An alternate, but equivalent, view of this operation is that it seeks all substreams S_k, for which the v^{th} element from the end has a timestamp greater than $\tau - T$. Based on this observation, one idea for improving the efficiency of lookup is as follows: Maintain an index over the timestamp values of the v^{th} element from the end of all the substreams. Use this index to determine, in $O(\log |\mathcal{K}|)$ time, all the substreams for which the timestamp of the v^{th} element from the end is greater than $\tau - T$.

Since v is a parameter in LOOKUP(τ, T, v), we would need to maintain such an index for every possible v in order to use the above idea. However, doing so would dramatically increase the update cost: every new element a_{ki} of substream S_k changes the timestamp of the v^{th} element from the end for every v, and so requires updating all the indexes.

However, if we are permitted to approximate v, i.e., use a different value v' that is close to v, we can reduce the number of different indexes that need to be maintained, since many values v can use the same approximation v'.

This observation forms the basis for algorithm CI-COUNT: CI-COUNT divides the positions from the end of a substream S_k into different *levels* (not to be confused with levels of base-intervals in Section 3.1.1). It maintains one search index for each level. The index for level-ℓ contains, for each substream S_k, the timestamp of some element that currently belongs to level-ℓ. These indexed timestamps are used for approximate answer lookups.

Definition 4.1 (Level) Let m be the current length of substream S_k. Then the current *level* of a position $p \leq m$ of S_k is defined to be $\lfloor \log_2(m - p + 1) \rfloor$. □

The last position, m, of substream S_k belongs to level-0, the previous two ($m-1$ and $m-2$) to level-1, and so on. In general, 2^ℓ positions belong to level-ℓ.

Figure 7 contains the formal description of algorithm CI-COUNT. The variable TSTAMP$[k, \ell]$ contains the timestamp of the element of S_k that currently belongs to level-ℓ and whose position is a multiple of 2^ℓ. Note that, at any point of time, such an element (if it exists) is unique, since at most 2^ℓ contiguous positions belong to level-ℓ. For each level ℓ, all the TSTAMP$[k, \ell]$ values are indexed using a search tree SEARCHTREE$[\ell]$. In order to perform

STATE:
1. For each substream S_k
2. For each level-ℓ with at least one element
3. Let i be unique position of S_k such that:
4. (a) i currently belongs to level-ℓ, and
5. (b) i is a multiple of 2^ℓ
6. TSTAMP$[k, \ell] = \tau_{ki}$
7. For each level-ℓ
8. Store TSTAMP$[k, \ell]$, for all valid k, using a search tree SEARCHTREE $[\ell]$.

UPDATE(a_{km}, τ_{km}):
1. $p = 1$
2. Let $z =$ number of trailing 0s in binary representation of $m + p$.
3. For $\ell = z$ downto 1 do:
4. TSTAMP$[k, \ell]$ = TSTAMP$[k, \ell - 1]$
5. TSTAMP$[k, 0] = \tau_{km}$

LOOKUP(τ, T, v):
1. Let $\ell = \lfloor \log_2 v \rfloor - 1$
2. Determine $\mathcal{A} = \{k \in \mathcal{K} \mid$ TSTAMP$[k, \ell] \geq \tau - T\}$ using SEARCHTREE $[\ell]$.
3. Return $\mathcal{A}$.

Figure 7: Algorithm CI-COUNT

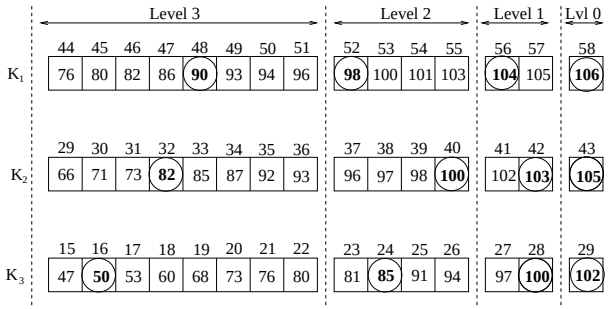

Figure 8: CI-COUNT Example

LOOKUP(τ, T, v), CI-COUNT uses SEARCHTREE$[\ell]$ for $\ell = \lfloor \log_2 v \rfloor - 1$ to determine all substreams S_k such that TSTAMP$[k, \ell] \geq (\tau - T)$.

Example 4.1 Figure 8 shows the timestamps (within boxes) and positions (above boxes) of elements belonging to three substreams. The elements themselves are not shown. The timestamps that are stored in TSTAMP$[k, \ell]$ are circled. For example, TSTAMP$[k_3, 2] = 85$. The search trees over TSTAMP$[k, \ell]$ values are not shown.

Consider LOOKUP$(106, 20, 11)$ which seeks at time 106 all substreams that have received more than 11 tuples in the last 20 time units, i.e., in the time interval $(87, 106)$. Clearly, the correct output is $\{k_1\}$. The same output can also be obtained by checking if the timestamp of the 11th tuple from the end (which is 90, 85, and 68, for S_{k_1}, S_{k_2}, and S_{k_3}, respectively) is ≥ 87.

Since $\lfloor \log_2 11 \rfloor - 1 = 2$, CI-COUNT returns those substream keys k for which TSTAMP$[k, 2] \geq 87$, which is $\{k_1, k_2\}$ for this example. In other words, for each substream, CI-COUNT uses the timestamp of a position at a distance 4–7 from the end of the substream, instead of the timestamp of the position that is at a distance 11, and checks if it is greater than 87. □

We now briefly comment the on update operation. Consider any one particular TSTAMP$[k, \ell]$ for $\ell > 1$. By definition, TSTAMP$[k, \ell]$ stores the timestamp of the element that currently belongs to level-ℓ and whose position is $i2^\ell$ for some i. Clearly, TSTAMP$[k, \ell]$ changes only when this element moves to level-$(\ell+1)$, and the element corresponding to position $(i+1)2^\ell$ enters level-ℓ. Since $(i+1)2^\ell$ is also a multiple of $2^{\ell-1}$, the timestamp of this element would previously have been stored in TSTAMP$[k, \ell - 1]$. Therefore, TSTAMP$[k, \ell]$ can be updated by just copying the previous value of TSTAMP$[k, \ell - 1]$ (Step 4 in UPDATE(a_{km}, τ_{km})).

Lemma 4.1 *Algorithm* CI-COUNT *presented in Figure 7 has approximation parameter $\epsilon = 0.75$: If $k \in \mathcal{K}$ is returned in the output of* LOOKUP(τ, T, v)*, then* COUNT$_A(S_k[T, 0]) \in (v/4, \infty)$ *at time τ. If at time τ,* COUNT$_A(S_k[T, 0]) \in (v, \infty)$*, then k is returned in the output of* LOOKUP(τ, T, v).

Theorem 4.1 *Let T_{max} denote maximum time interval of a sliding window that* CI-COUNT *supports, and let N_{max} denote the current number of elements belonging to all substreams with timestamps in the last T_{max} time-units. The* CI-COUNT *algorithm presented in Figure 7 requires $O(|\mathcal{K}| \log N_{max})$ space, has an amortized update time complexity of $O(\log |\mathcal{K}|)$, and a lookup time complexity of $O(|\mathcal{K}_o|)$, where $|\mathcal{K}_o|$, the number of substreams in the output of lookup. Further, the lookup has an approximation parameter $\epsilon = 3/4$.*

4.1.2 Parameterized CI-COUNT

The technique that we use to parameterize CI-COUNT is well-known and has been suggested before [12]. We only present the main ideas the statement of the results. Consider a simple generalization of the non-parameterized CI-COUNT. The generalized version has p levels of size 1, p levels of size 2, and so on. As before, for each level whose size is 2^ℓ, CI-COUNT stores the timestamp of the element that belongs to the level, and whose position is a multiple of 2^ℓ. We can extend the lookup and update operations presented earlier to the general case in a straightforward manner. The update complexity is now $O(p)$, while the lookup complexity remains unchanged at $O(|K_{ao}|)$.

As we increase p, the relative error, ϵ, of the generalized version reduces. For example, we can show that relative error for the case $p = 2$ is $\epsilon = 1/2$ instead of $\epsilon = 3/4$ for $p = 1$. In general, we can prove that as p increases the relative error falls roughly as $2/p$. Therefore, by setting p to be roughly $2/\epsilon$, we can achieve any desired relative error ϵ. The results claimed in Table 1 follow directly from this relation between p and ϵ.

4.2 CI-SUM

CI-SUM is derived from CI-COUNT in a straightforward manner: Replace the `SUM` aggregation functions in the input operators with `COUNT` aggregation functions, and process a modified stream S' using algorithm CI-COUNT. Corresponding to every element a_{ki} of S_k, there are a_{ki} copies of the same element in S'_k with the same timestamp τ_{ki}. Any lookup involving the `SUM` aggregation function can be translated into an equivalent lookup involving `COUNT` aggregation function on the modified stream S', and therefore can be processed using CI-COUNT. The only problem with this approach is that naïvely performing an update for each of the a_{ki} copies of an element a_{ki} of the original stream S_k would result in an update operation whose time complexity grows linearly in a_{ki}. However, we can show that the updates corresponding to all the a_{ki} duplicate copies can be collectively performed with $O(\log a_{ki})$ multiplicative overhead.

5 Implementation Issues

We have implemented all of the algorithms in this paper, and we briefly touch on some of the more important implementation issues. All of our algorithms permit an implementation that uses simple arrays. In particular, they do not require pointer manipulations or dynamic memory management. For example, our implementation of Algorithm R-INT (Figure 6) uses an array of N_{max} values (recall that

N_{max} is the earliest left-end of a supported window). Each array location is used to store the value $f(I_r)$ for some active running interval I_r. The array is conceptually treated is a circular buffer, and values $f(1, r)$ and $f(1, r+1)$ are stored in adjacent locations of the buffer. Clearly, this organization lets us access any active $f(I_r)$ in a single memory lookup.

A similar implementation strategy is used for Algorithm B-INT: We use an array of size N_{max} for the level-0 intervals, an array of size $\frac{N_{max}}{2}$ for level-1 intervals, and so on. Further, all common operations in the B-INT algorithm, such as expressing a lookup interval as a union of base-intervals (Step 1 of the Lookup operation in Figure 3) can be implemented using low-level, highly efficient bit-level operations. Details are omitted due to space constraints.

6 Experiments

A wide variety of experiments evaluating the empirical performance of our algorithms can be conducted. Due to space limitations, here we report on three sets of experimental results that are representative of overall performance.

1. *Comparison against alternate approaches*: For SUM, we compare the performance of the R-INT and B-INT algorithms against the PSoup algorithm sketched in Section 3.1.3 and the two naïve extreme approaches discussed in Section 1. We see that our algorithms outperform the alternatives.

2. *Performance of ASW algorithms*: We present raw performance numbers for three basic ASW algorithms, showing they are capable of handling very high lookup and stream update rates (millions of events per second).

3. *Performance of* CI-COUNT: Using real stock trade data, we compare the performance of algorithm CI-COUNT against the alternative approach of processing each ASW suboperator independently as described in Section 4. We show that there exist cases where CI-COUNT provides orders of magnitude improvement in overall performance.

The first two experiments are data-independent, since the performance of none of the relevant algorithms depend on the actual data values, while the third experiment is data dependent. Therefore, we use synthetically generated data for the first two, and real financial data for the third experiment.

All experiments were performed on a 4-processor 700 Mhz. Pentium III machine running Linux with 4 GB of main memory RAM.

6.1 Comparison with alternate approaches

For the SUM aggregation function, we compared the performance of the R-INT and B-INT algorithms against PSoup, as well as against the naïve approaches discussed in Section 1: (1) Materialize the results of all operators at all times (*materialize all*); (2) Maintain the maximum required window of the input stream and perform all answer computations only at lookup time (*materialize none*). Since PSoup [10] does not provide implementation details, we adapted *libavl* [23], a publicly available implementation of red-black (binary) search trees.

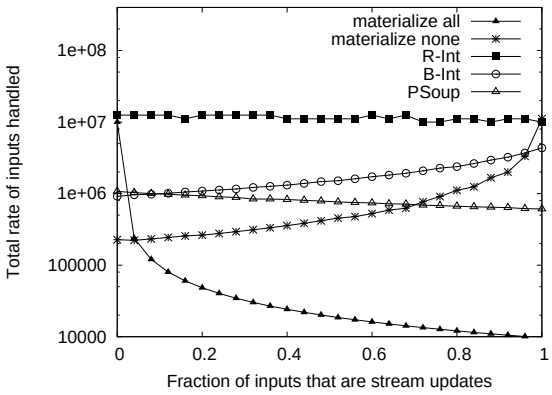

Figure 9: Comparing R-INT and B-INT against alternatives

We used 1000 operators of the form $\texttt{SUM}_A S[N, 0]$ with tuple-based windows varying in size from $N = 0$ to $N = 999$. For each algorithm, we measured the total input rate that it was able to handle. Each input consisted of a mixture of query lookups and stream updates. We constructed different inputs by varying the ratio of updates to lookups. Lookups were picked uniformly at random from one of the 1000 operators.

Figure 9 shows the results. As expected the performance of the full materialization approach is good when update rates are low, while that of the on-demand approach is good when lookup rates are low. However, both approaches deteriorate quickly as we move away from their favorable ends. The performance of the other three algorithms remains relatively stable for all inputs. As expected, R-INT outperforms the other two. The performance of B-INT and PSoup are similar when there are no stream updates, however the performance of PSoup falls and B-INT improves as the ratio of updates in the input increases. This occurs because the actual cost of updates in PSoup is higher than the actual cost of lookups although their asymptotic costs are the same, while for B-INT updates are cheaper than lookups (asymptotically and empirically).

6.2 Performance of ASW algorithms

We present raw performance numbers for three basic algorithms: R-INT (for SUM), B-INT (for MAX), and B-INT-QNT (for QUANTILE). For each one we measured its performance handling updates and handling lookups separately. From these numbers we can easily derive the expected performance for a combined workload. Update handling was measured for different values of maximum window left-end N_{max}, and lookup handling was measured for different values of the query windows (W). For lookups, we set $N_{max} = 100,000$. Individual operator windows were distributed uniformly over the entire permitted range, i.e., we considered both suffix and historical windows.

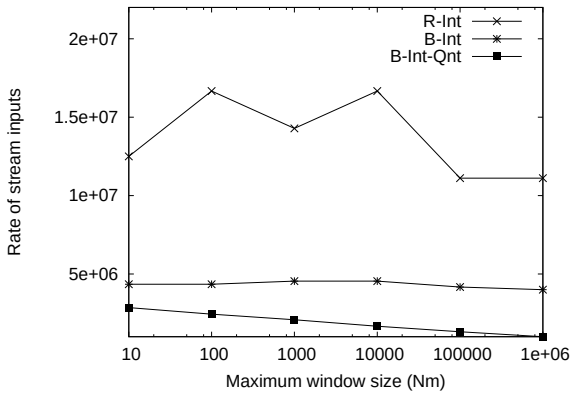

Figure 10: Stream update rates for the ASW algorithms

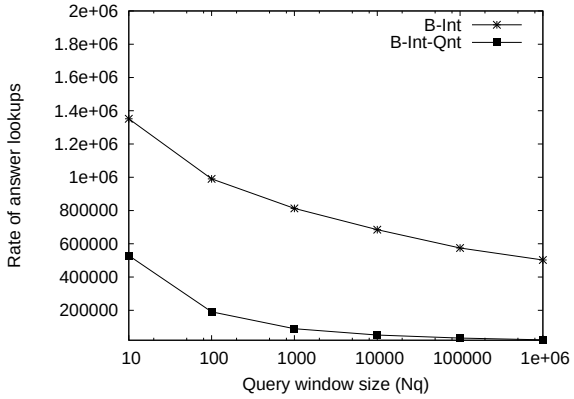

Figure 11: Maximum lookup rates for the ASW algorithms

Figures 10 and 11 present the results, showing that our algorithms handle up to millions of events per second, depending on window sizes. Note that the y-axis in these figures does not go to zero. For example, in Figure 11 B-INT-QNT handles a lookup rate of about 22,000 per second even at the maximum window size (not obvious from the graph). In Figure 11 we deleted the plot for R-INT in order to better depict those for B-INT and B-INT-QNT. The performance of R-INT is uniformly around 10^7 lookups per second. Note that all of these results are for tuple-based windows only. Time-based windows have similar performance characteristics with a slight degradation due to additional overhead.

6.3 Performance of CI-COUNT

Unlike the algorithms in our first two sets of experiments, the performance of CI-COUNT is highly dependent on actual data and operators, specifically the selectivity of range conditions and the "spread" of aggregation answers across different substreams. Further, picking the right value of ϵ represents a tradeoff between update performance and lookup performance. A detailed study of these issues is left as future work.

Here, we report on one particular experiment for CI-COUNT. We used a one-day stream of real stock trades from the TAQ database, containing approximately 5000 substreams based on ticker symbol. Our queries were synthetically generated by specifying a suffix window ranging from 15 minutes to 3 hours, monitoring stocks over the window with total trades above a given threshold. We selected the threshold to make its selectivity roughly $.03$–$.05$. Lookups and updates were equally interleaved.

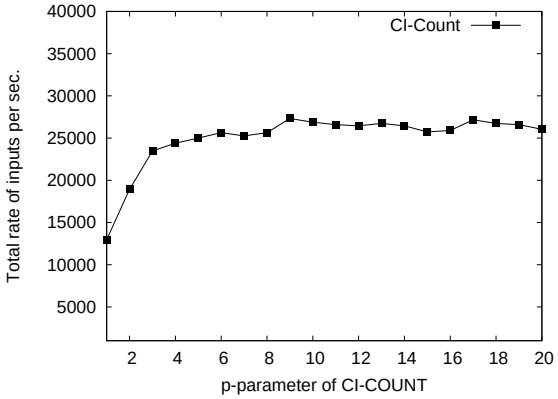

Figure 12: Performance of CI-COUNT for various values of p. The naïve approach processes 1000 inputs/second.

We compared the performance of two approaches: (1) The naïve approach that uses algorithm R-INT to process each substream independently; (2) The approach that uses CI-COUNT in conjunction with R-INT to produce exact answers. For the second approach we varied the parameter p, which directly affects the relative error ϵ.

The naïve approach processes only about 1000 inputs (lookups and updates) per second. In Figure 12 we see that CI-COUNT with an appropriately chosen value of p (or ϵ) processes about $25,000$ inputs per second. Note that the selectivity of the range condition imposes a upper bound on the relative performance of CI-COUNT when compared to the simple approach, suggesting that we can expect greater benefits for more selective range conditions.

7 Related Work

One class of techniques for resource sharing between different queries is based on detecting and exploiting common subexpressions. All of the multiquery optimization techniques for conventional one-time queries, e.g., [27, 28], belong to this class. In the context of continuous queries, similar techniques have been used in the NiagaraCQ system [11]. Recent work in the TelegraphCQ project [9, 25] suggests using the *Eddy operator* [6] for sharing, and argues that since the Eddy operator does not fix a query plan, it exposes greater sharing possibilities.

The second class of techniques, which is the focus of this paper, is sharing resources at the operator level. Most previous work on operator-level sharing has focused on filters. All traditional pub-sub systems, e.g., [1, 16, 21], and some continuous query processing systems, e.g., [11, 25], use variants of predicate indexes for resource sharing in filters. Work on resource sharing for XML-based filters, e.g., [2, 13, 22, 26], also belongs to this class. Recently, reference [14] considers the problem of sharing *sketches* for

approximate join-based processing. As described in Section 3.1.3, PSoup briefly considers the problem of resource sharing and proposes an algorithm for ASW operators.

Lot of research on sliding window aggregates has focused on computing approximate aggregates (statistics) over sliding windows in limited space, e.g., [4, 12, 18]. Our goal in this paper is to compute exact aggregates. For many applications (e.g., applications over financial data) the ability to compute exact answers is crucial. Also, aggregation functions like `MAX` and `MIN` are inherently difficult to approximate [12].

8 Conclusions

We presented new techniques for scalable processing of large numbers of operators based on sliding-window aggregates. Our techniques have precise theoretical guarantees, and they perform extremely well in practice.

We have identified at least two avenues for future work. First, the techniques in this paper represent alternatives, rather than a single "optimal" solution. An optimal solution depends on the exact rates of answer requests and stream updates, and may change as these rates change. Thus, one direction is to consider adaptive techniques that factor in relative answer/update rates. A second direction is to extend the class of operators we handle to include arbitrary filters, both before and after the sliding window is applied.

Acknowledgments

We thank Mayur Datar and Gurmeet Manku for useful feedback.

References

[1] M. K. Aguilera, R. E. Strom, et al. Matching events in a content-based subscription system. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing*, pages 53–61, May 1999.

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the 26th Intl. Conf. on Very Large Data Bases*, pages 53–64, Sept. 2000.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Stanford University, Oct. 2003. `http://dbpubs.stanford.edu/pub/2003-67`.

[4] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, June 2004.

[5] A. Arasu and J. Widom. Resource sharing in continuous sliding window aggregates. Technical Report `http://dbpubs.stanford.edu/pub/2004-15`, Stanford University, 2004.

[6] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[7] B. Babcock, S. Babu, et al. Models and issues in data stream systems. In *Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–16, June 2002.

[8] D. Carney, U. Centintemel, et al. Monitoring streams - a new class of data management applications. In *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, pages 215–226, Aug. 2002.

[9] S. Chandrasekharan, O. Cooper, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the 1st Conf. on Innovative Data Systems Research*, pages 269–280, Jan. 2003.

[10] S. Chandrasekharan and M. J. Franklin. Streaming queries over streaming data. In *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, pages 203–214, Aug. 2002.

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

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

[13] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proc. of the 18th Intl. Conf. on Data Engineering*, pages 341–344, Feb. 2002.

[14] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *Proc. of the 9th Intl. Conf. on Extending Database Technology*, Mar. 2004.

[15] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems*, 21(3):270–313, Aug. 2003.

[16] F. Fabret, H. Jacobsen, et al. Filtering algorithms and implementation for very fast publish/subscribe. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 115–126, May 2001.

[17] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 13–24, May 2001.

[18] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proc. of the 14th Annual ACM Symp. on Parallel Algs. and Architectures*, pages 63–72, Aug. 2002.

[19] A. C. Gilbert, Y. Kotidis, et al. How to summarize the universe: Dynamic maintenance of quantiles. In *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, pages 454–465, Aug. 2002.

[20] J. Gray, S. Chaudhuri, et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, Mar. 1997.

[21] R. E. Gruber, B. Krishnamurthy, and E. Panagos. READY: A high performance event notification system. In *Proc. of the 16th Intl. Conf. on Data Engineering*, pages 668–669, Mar. 2000.

[22] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 419–430, June 2003.

[23] libavl: Library for balanced binary trees. Available at `http://www.gnu.org/directory/GNU/libavl.html`.

[24] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proc. of the IEEE Symp. on Security and Privacy*, pages 146–161, May 1999.

[25] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, June 2002.

[26] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 431–442, June 2003.

[27] P. Roy, S. Seshadri, et al. Efficient and extensible algorithms for multi query optimization. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 249–260, May 2000.

[28] T. K. Sellis. Multiple-query optimization. *ACM Trans. on Database Systems*, 13(1):23–52, Mar. 1988.

[29] Traderbot home page. `http://www.traderbot.com`, 2003.

[30] G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection approach. In *Proc. of the 14th Annual Computer Security Appln. Conf.*, pages 25–38, Dec. 1998.

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