Multi-Query Optimization of Sliding Window Aggregates by Schedule Synchronization*

Lukasz Golab† Kumar Gaurav Bijay‡ M. Tamer Özsu§

Technical Report CS-2006-26
August 2006

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

†David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, lgolab@uwaterloo.ca. Now at AT&T Labs, Florham Park, New Jersey, USA.

‡Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India, gauravk@cse.iitb.ac.in. Work done while the author was visiting the University of Waterloo.

§David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, tozsu@uwaterloo.ca.

Abstract

Data stream systems process persistent queries, typically posed over sliding windows and re-evaluated periodically as the windows slide forward. Due to their long-running nature, a number of similar persistent queries may run in parallel at any given time, therefore multi-query optimization is particularly important. In traditional multi-query optimization, one of the primary goals is to detect common parts across multiple queries issued at the same time and perform the common task only once. Another related goal is to check if a new query may be answered using the answer of a related query which has been computed previously (and is stored as a materialized view). In this paper, we argue that in the context of periodically re-evaluated queries, multi-query optimization requires an additional step beyond common sub-expression matching. This is because queries that have been identified as similar may be re-evaluated with different frequencies and therefore may be scheduled at different times. Thus, the additional step must attempt to synchronize the re-execution times of similar queries in order to take advantage of computation sharing.

The solution presented in this paper focuses on periodically-evaluated aggregates over sliding windows of various lengths, which are a common class of persistent queries used for monitoring purposes. The proposed solution assumes that users specify an upper bound on the interval between re-evaluations of their queries and is based upon the following insight: it may be cheaper to re-execute some queries more often if their re-execution schedules can be synchronized with those of similar queries, thereby amortizing the computation costs. We also show that additional schedule synchronization is possible when the system is forced to lengthen the desired re-execution intervals during periods of overload. Our solutions are based upon a variant of the earliest-deadline-first algorithm that views persistent queries are periodic tasks. Experimental results show significant improvements in system throughput due to increased resource sharing.

1 Introduction

Data stream management systems (DSMSs) process on-line data such as sensor measurements, IP packet headers, stock quotes, or transaction logs. Usually, only a sliding window of recently arrived data is available at any given time for two reasons: to avoid memory overflow and to emphasize recent data that are likely to be more useful. In addition to traditional one-time queries evaluated over the current state of the sliding window(s), users pose persistent (continuous) queries that monitor the streaming data. These monitoring queries often compute sliding window aggregates, such as road traffic volume via sensors embedded in a motorway, network bandwidth usage statistics, or recent behaviour of stock prices.

For all but the simplest queries, it may be infeasible for a DSMS to compute up-to-date results whenever a new stream tuple arrives or an old tuple expires from its window. Additionally, users may find it easier to deal with periodic output rather than a continuous output stream [6, 11]. Consequently, much of the DSMS research assumes that answers are updated periodically [1, 8, 10, 12, 27, 29, 34, 40]. For example, consider a query that monitors the median packet length over a stream of IP packet headers (call it S), computed every two minutes over a ten-minute window. Using syntax similar to the CQL stream query language [4], this query may be specified as follows.

```
Q1: SELECT MEDIAN(length)
    FROM S [WINDOW 10 min SLIDE 2 min]
```

The current workload may include many persistent queries similar to Q_1, but having additional `WHERE` predicates, different window sizes, or different periods (in the remainder of this paper, we will use the expressions "`SLIDE` interval" and "period" interchangeably). For instance, the workload may also include the following query.

```
Q2: SELECT MEDIAN(length)
    FROM S [WINDOW 14 min SLIDE 3 min]
```

These two queries may be posed by different users (e.g., network engineers), or by the same user, who wishes to summarize the network traffic at different granularities and may simultaneously ask similar aggregate queries over different window lengths.

Other examples of applications that issue many aggregate queries in parallel over different window lengths include detecting bursts of unusual activity identified by abnormally high or low values of an aggregate such as `SUM` or `COUNT` [39, 41]. A burst of suspicious activity may be a denial-of-service attack on a network or a stock with an unusually high trading volume. Since the length of the burst is typically unknown apriori (e.g., a short-lived burst could produce several unusual values in a span of a few seconds, whereas a longer-term burst would have to generate a hundred suspicious packets over the last minute), a series of aggregates over a range of window lengths needs to be monitored.

Given that many persistent (and likely similar) queries are expected to run concurrently, multi-query optimization is particularly important. Traditionally, one of the primary goals of multi-query optimization in relational systems has been to detect common parts across multiple queries issued at the same time and perform the common task only once [33]. A related issue occurs when a new query is posed, but the answer of a related query has been computed previously and is now stored as a materialized view. In this case, it would be beneficial for the new query to re-use the materialized view rather than computing it from scratch [18]. In our context, an application of a reasonable set of sub-expression and view matching rules, suitably augmented with DSMS-specific details such as sliding window lengths, identifies that Q_1 and Q_2 are similar because the shorter window required by Q_1 is contained in the longer window maintained by Q_2 [6, 40].

In this paper, we argue that multi-query optimization of periodically-refreshed queries issued to a DSMS requires an additional step. For instance, even if Q_1 and Q_2 can be identified as similar, they have different periods and therefore it is not obvious whether they can in fact share computation. Thus, the additional step, which presents a novel challenge in joint evaluation of periodically-refreshed persistent queries, is to enable computation sharing among similar queries, even if they are re-evaluated with different frequencies and therefore get scheduled at different times.

For a motivating example, consider a time axis with a possible execution sequence of Q_1 and Q_2 from above illustrated in Figure 1 (a). The two queries are both due for a refresh every six minutes (the least-common-multiple of their `SLIDE` intervals) and may be executed together at those times. One way to do this is to save any temporary state used during the calculation of Q_1, process the remaining four minutes of data, and merge the computations to arrive at the answer for Q_2. In effect, over two-thirds of the work is amortized across two queries. However, now suppose that the semantics of the `SLIDE` clause are defined as an upper bound on the interval between two re-evaluations of the corresponding query (this assumption will be justified

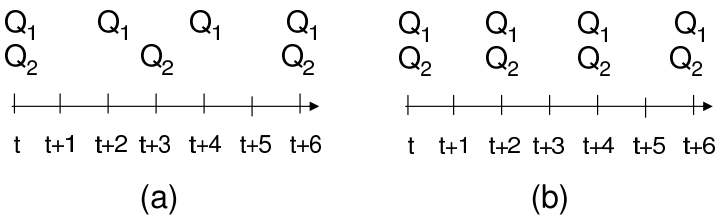

Figure 1: Two possible ways to schedule queries Q_1 and Q_2.

in Section 3). In this case, it is possible to always schedule Q_2 when Q_1 is due for a refresh, as shown in Figure 1 (b). Clearly, more computation is now shared, but the savings in processing time may be defeated by the work expended on updating Q_2 more often than necessary.

Motivated by the above issues, this paper presents a multi-query optimization solution for sliding window aggregates, whose goal is to synchronize the re-execution schedules of similar queries. Our specific contributions are as follows.

- We formalize the meaning of the `SLIDE` clause used in data stream query languages.
- We model sliding window queries as periodic tasks that must be scheduled according to their deadlines. We demonstrate that the earliest-deadline-first (EDF) algorithm [36] may be used as a starting point in the design of a DSMS query scheduler.
- We propose a cost-based heuristic for deciding whether a query should be re-executed more often than necessary if its execution times can be synchronized with those of a similar query (as in Figure 1 (b)).
- The workload of a DSMS, both in terms of persistent and one-time queries, is expected to fluctuate over time. Moreover, it is well-known that many types of data streams have bursty arrival rates [24, 28, 32, 41]. Therefore, it is likely that a DSMS may experience periods of temporary overload. In response, we discover additional sharing opportunities that arise during overload and exploit them to minimize the impact of overload on system throughput. Suppose that Q_1 is due for a refresh at time t and Q_2 at time $t+1$, but the system was unable to run Q_1 at time t. We have two choices at time $t+1$: clear the backlog by executing Q_1 and other late queries before moving on to Q_2, or recognize that Q_1 and Q_2 are similar and execute them together before moving on to other late queries (in which case Q_2 is much cheaper to evaluate because it can re-use the answer computed by Q_1). We show that looking for similar queries in the "late query set" and the "currently-scheduled query set" increases overall throughput.

The remainder of this paper is organized as follows. Section 2 introduces our system model and assumptions. Section 3 discusses the semantics of periodically re-evaluated persistent queries. Section 4 presents our scheduling techniques synchronizing the re-execution times of similar queries. Our solutions are evaluated experimentally in Section 5 and compared to related work in Section 6. Section 7 summarizes the paper and suggests directions for future work.

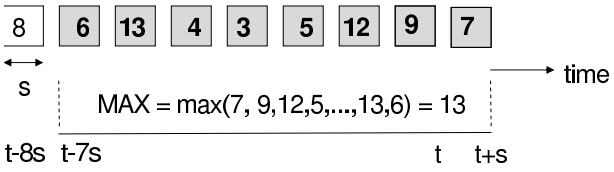

Figure 2: Answering a MAX query using a MAX synopsis.

2 Preliminaries

2.1 Single Query Evaluation

A data stream is assumed to be a sequence of timestamped relational tuples with a fixed schema, possibly buffered to ensure timestamp order. Queries are posed over sliding windows specified in units of time; a window of length w time units means that a stream tuple with timestamp ts *expires* at time $ts + w$. We consider periodically-evaluated monitoring queries, each of which runs for a specified *lifetime*, and includes a set of selection and group-by conditions, followed by an aggregate function. The aggregate may be one of the following: `SUM`, `COUNT`, `AVG`, `MAX`, `MIN`, `COUNT DISTINCT`, `PERCENTILE`, or `TOP k` (the last three may be exact or approximate).

Queries are evaluated by scanning appropriate *synopses* that summarize the underlying window; one window may be linked to a number of synopses corresponding to various types of aggregates over various attributes. A synopsis is associated with three parameters: the type of aggregate used, s, which is the time between two consecutive synopsis updates[1], and b, which defines the longest window covered by the synopsis as bs. The window of length bs is partitioned into b non-overlapping intervals of length s each and each of these intervals stores summary data. Every s time units, the synopsis is updated to reflect the new state of the underlying window as it slides forward. For example, a `MAX` synopsis stores the value of the maximum element in each interval and is illustrated in Figure 2 for $b = 8$. Let t be the time of the last synopsis update. At time $t + s$, the oldest interval $(t - 8s, t - 7s]$ now summarizes expired tuples (of which the largest tuple has value 8) and is removed. In its place, a new interval $(t, t + s]$ is appended[2]. More precisely, the maximum value of all tuples in the new interval, namely 7, is added to the synopsis. As illustrated, query evaluation proceeds by reading the maximum values stored in the synopsis and returning the overall maximum. This type of synopsis is a generalization of the data structures proposed in [6, 40].

The approach of pre-aggregating each interval in the synopsis is suitable for distributive aggregates [17] such as `SUM`, `COUNT`, `MAX`, and `MIN`. An aggregate f is distributive if, for two disjoint sets X and Y, $f(X \cup Y) = f(X) \cup f(Y)$. For holistic aggregates [17] (an aggregate f is holistic if, for two multi-sets X and Y, computing $f(X \cup Y)$ requires space proportional to $X \cup Y$), each interval must store additional information. For instance, storing the frequency counts of

[1]This means that the SLIDE intervals of queries using the synopsis must be multiples of s; this will be formalized in Section 3.

[2]Note that newly arrived tuples are buffered until the newest interval fills up, therefore the aggregate value over the newest interval can be pre-computed on-the-fly as new tuples arrive.

values occurring in each interval may be used for quantiles, `TOP k`, and `COUNT DISTINCT`. As before, query execution involves scanning each interval, computing the overall frequency counts in the entire window, and producing the final answer. Furthermore, as new tuples arrived and are buffered, frequency counts of the newest interval may be pre-computed on-the-fly.

If the space complexity of storing frequency counts in each sub-window turns out to be prohibitive, then approximate answers of non-distributive aggregates may be computed with the help of sketches. In this case, each interval in the synopsis stores a sketch, which is a compact approximation of the underlying distribution of values [5]. For example, approximate `COUNT DISTINCT` queries can be computed using Flajolet-Martin (FM) [15] or Alon-Matias-Szegedy (AMS) [3] sketches, whereas approximate quantiles and `TOP k` queries can use Count-Min (CM) sketches [13]. In all cases, pre-computing the sketch of the newest interval and sketch merging are straightforward because sketches are distributive.

2.2 Multi-Query Evaluation

Recall Figure 2 and observe that the synopsis illustrated therein may be used to answer `MAX` queries over windows with lengths s, $2s$, ..., $8s$. In particular, suppose that two `MAX` queries are due for re-execution at the same time, one of which references a window of length $6s$ and the other references a window of length $8s$. As the latter is being computed by scanning the synopsis, the former may be answered "for free"—we stop after reading the first six intervals (from youngest to oldest) and return the maximum so far. Similarly, if `SUM` and `COUNT` are being computed over some attribute, then `AVG` over the same attribute can simply divide the two answers already computed (without having the re-scan the `SUM` and `COUNT` synopses). Furthermore, suppose that two `MAX` queries have just been answered, one of which computing the maximum packet length over a stream of TCP packets and the other computing the maximum length of UDP packets. Clearly, taking the maximum of the two answers immediately yields an answer to a `MAX` query over the length of all TCP and UDP packets combined.

The above examples motivate the need for synchronizing the re-execution times of similar queries, in which case a significant fraction of the overall computation may be amortized across multiple queries. In general, we assume that the DSMS contains a set of rules for grouping similar queries that may be executed together. The definition of this rule set is orthogonal to the scheduling solutions that will be presented in this paper[3].

2.3 Dealing with Changes in the Query Workload

For completeness, we now briefly discuss what happens when the query workload changes. First, in addition to persistent queries, a DSMS may accept one-time queries that are not registered in advance. If appropriate synopses exist, then one-time queries may be answered efficiently. Otherwise, we may reject the query, or the system may be configured to store the actual windows in addition to synopses. In this case, one-time queries may be answered via sequential scan of the window(s). Second, a persistent query whose lifetime expires may be removed from the system, but its synopsis may still be in use by another query. Conversely, if appropriate synopses exist (or the windows are stored), then a newly registered query may begin execution immediately.

[3]See, e.g., [20, 27, 35, 38] in the context of queries with group-by and aggregation.

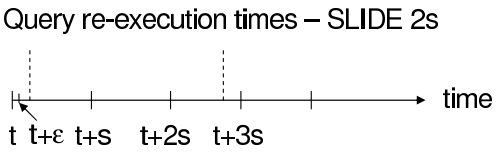

Figure 3: Semantics of periodic query re-execution.

An interesting case occurs when a matching synopsis is found, but its window length is too short for the new query. If so, then we can extend the time span of the synopsis by not deleting its oldest intervals for several updates. Note that this causes a delay before the query begins generating output. If all else fails, then we may build an appropriate synopsis for the new query from scratch, but it will take one window length before the synopsis fills up with data.

3 Semantics of Periodic Query Re-Execution

Prior to discussing query scheduling, we formalize the definition of the `SLIDE` clause used to determine how often a persistent query is to be re-executed. Consider query Q accessing one or more synopses. As described in Section 2, a synopsis is updated when its buffer completes pre-aggregating the current time interval, say $(t-s, t]$ (recall that s is the interval between synopsis updates). This means that at time t, all the buffers that have been computing aggregates over this particular interval are ready to send pre-aggregated values to their synopses. After all the updates have taken place at some time $t+\epsilon$, the synopses reflect the state of their window(s) as of time t. We assume that $\epsilon << s$, i.e., there should be ample time between synopsis updates for executing queries.

Let Q have a `SLIDE` interval of $2s$ and let Q's synopses have update times $t + is, i \in \mathbb{N}$. A time line is illustrated in Figure 3, showing a re-execution of Q some time between $t + \epsilon$ and $t + s$. The answer of Q reflects the state of its synopses (and the underlying windows) as of time t. We define a `SLIDE` interval of ns to mean that the number of times Q's synopses are updated between consecutive re-executions *should* be no more than n. Hence, the next re-evaluation of Q should reflect the state of the synopses as of time $t + 2s$ at the latest, therefore it should be done before time $t + 3s$, as illustrated.

We make two practical remarks regarding the above definition. First, by defining only an upper bound on the number of synopsis updates between query re-executions, it is assumed that queries may be refreshed more often than specified. This assumption has two consequences. First, if the system is lightly loaded, then it may in fact be possible to re-execute queries more often; had the definition assumed a rigid re-execution interval, the system would experience periods of idle time during underload. Second, allowing a query to be refreshed sooner than required enables the synchronization of its re-execution times with those of a similar query, as illustrated in Figure 1 (b). Again, had the definition required a fixed re-execution interval, resource sharing would have been significantly limited, as illustrated in Figure 1 (a).

The second remark involves using the words *should* rather than *must* in the definition of periodic re-execution. This accommodates periods of overload, which are likely to occur due to fluctuations in the query workload and stream arrival rates. More precisely, it is assumed that

for each periodic query, the DSMS must follow the above definition whenever possible, but is permitted to break it if necessary. Consequently, the definition allows at least the following two solutions for handling overload. First, the DSMS may drop a fraction of queries when overload is detected and block users from re-registering the dropped queries until the overload has subsided. Second, the DSMS may continue to re-execute all of its queries during overload, but temporarily increase all of their periods. The second solution is assumed in the remainder of this chapter as it ensures fairness across the query workload.

Finally, it may be argued that some users are interested in tracking events that are expected to occur regularly, say every three minutes, and would therefore insist that their queries be refreshed exactly every three minutes. However, these situations are different from the queries discussed in this paper in the following two ways. First, tracking a specific event is not a periodic query, but rather a continuous query that keeps listening for new input and immediately reacts upon observing the specified event (e.g., by raising an alarm). Second, event tracking corresponds to selection queries, possibly with complex predicates, rather than aggregation over sliding windows, which is the focus of this paper.

4 Multi-Query Scheduling

We are now ready to present our scheduling solutions for synchronizing the re-execution times of queries that have been identified as similar (by means of some set of rules, as discussed in Section 2). We begin by designing an earliest-deadline-first scheduler for periodically-executed queries (Section 4.1) and we then extend it to schedule similar queries together (Section 4.2). Next, we present two improvements for synchronizing the schedules of similar queries (Sections 4.3 and 4.4).

4.1 Earliest-Deadline-First Scheduling

Figure 3 suggests the following execution sequence: at time t, we update all the synopses that are waiting for aggregates over the new interval $(t-s, t]$, find all the queries due for a refresh, and attempt to execute them before the synopses are updated again. However, only shared sets of synopses need to be synchronized (for example, if the average is computed by dividing the answers obtained from `SUM` and `COUNT` synopses, then these two synopses must be updated at exactly the same time intervals). Thus, while the sequence that started at time t is running, another sequence, corresponding to a different group of synopses and queries, may begin. Without loss of generality, the remainder of this section deals with "local" scheduling of one task sequence, containing queries that access the same group of synopses. Our solutions are compatible with any underlying "global" scheduler, e.g., allocating a weighted time slice to each local scheduler.

To reflect the `SLIDE` semantics defined in Section 3, each persistent query is modeled as a task T_i with period $n_i s$. We denote the rth re-execution of T_i as T_i^r and assign a *time-until-deadline* to each task, denoted $d(T_i)$, as follows. After T_i^r is done, we set $d(T_i) = n_i$. After each update of the synopses, we set $d(T_i) = d(T_i) - 1$. T_i^r is said to *execute on-time* if $d(T_i) \geq 0$ when it is done. For example, in Figure 3, $d(T_i) = 2$ after the query is executed, $d(T_i) = 1$ in the time interval $(t+s+\epsilon, t+2s]$, $d(T_i) = 0$ at time $t+2s+\epsilon$ until the next re-execution of the query

(i.e., the query is executed on-time), and $d(T_i) = 2$ again after the re-execution.

Due to unpredictable system conditions and dynamic query workloads, it may not be possible to reliably estimate how long it will take to execute each T_i^r, nor is it possible to schedule all tasks off-line. Consequently, we have chosen the on-line earliest-deadline-first (EDF) algorithm [36] as a starting point for a (local) query scheduler. A high-level pseudo-code is presented as Algorithm 1; for now, each query is scheduled separately. We maintain a task queue $Q(T)$ containing the current query workload. New queries are translated into new tasks and added to $Q(T)$; queries whose lifetimes have expired are removed. We assume that an update notification is sent when the interval currently being pre-aggregated in the buffers fills up and the synopses are due for an update. For clarity, Algorithm 1 and the remaining scheduling algorithms that will follow assume that queries and window-slides are executed in isolation. Nevertheless, the DSMS concurrency control techniques presented in [16] are fully compatible with the scheduling algorithms described here.

Algorithm 1 Local scheduler

Input: task periods n_i, task queue $Q(T)$
Local variables: array $d(T_i)$, initially $d(T_i) = n_i$

```
1  loop
2    if update notification arrives
3        update all synopses
4        decrement d(T_i) for all tasks T_i
5    end if
6    if Q(T) is not empty
7        execute task T_i with the lowest value of d(T_i)
8        reset d(T_i) = n_i
9    end if
10 end loop
```

The following observations regarding Algorithm 1 are worth noting:

- Even if T_i is re-executed late, we set $d(T_i) = n_i$ when it is done. The reasoning behind this is that lateness may imply system overload, therefore attempting to make up for the late re-execution by scheduling the next re-execution early could make the overload worse.
- Ties in line 7 may be broken arbitrarily. Alternatively, each query may be assigned a user-defined priority, in which case the highest-priority task (of all the tasks whose deadline is zero) is executed first.
- Algorithm 1 adaptively adjusts the query periods in response to system load. In underload, tasks may be executed before their $d(T_i)$-values reach zero. During overload, the algorithm attempts to clear the backlog by executing queries with the largest negative value of $d(T_i)$. Since tasks may be executed when their $d(T_i)$-values are negative, their periods are lengthened implicitly. This behaviour may be thought of as a form of automatic *load shedding*.

4.2 Scheduling Multiple Queries Together

Suppose that there are q groups of similar queries, call them $G_1, G_2, \ldots, G_q$, that may be efficiently re-executed together if they happen to be scheduled for re-execution at the same time. Each group may contain queries having different `SLIDE` intervals and `WINDOW` lengths. For the remainder of this section, suppose that one particular group, G_1, contains seven queries, each computing `MAX` over the same window and same attribute, and using a single `MAX` synopsis with $s = 1$ minute. The `WINDOW` and `SLIDE` parameters of the queries are as follows.

```
Q1: ... [WINDOW 10 min SLIDE 2 min]
Q2: ... [WINDOW  5 min SLIDE 2 min]
Q3: ... [WINDOW  6 min SLIDE 2 min]
Q4: ... [WINDOW 15 min SLIDE 3 min]
Q5: ... [WINDOW 12 min SLIDE 3 min]
Q6: ... [WINDOW 20 min SLIDE 5 min]
Q7: ... [WINDOW 30 min SLIDE 5 min]
```

To incorporate multi-query scheduling into Algorithm 1, we let each group G_i be a single task T_i, with period n_i equal to the shortest period among its queries. We call this technique *aggressive scheduling*. As illustrated in Figure 4 (a), aggressive scheduling executes all seven queries in G_1 every two minutes. Another simple technique jointly executes similar queries only if they are due for a refresh at the same time. This can be achieved by splitting each group G_i into sub-groups, $G_{i,1}, G_{i,2}, \ldots, G_{i,q_i}$, containing queries with the same `SLIDE` interval. Furthermore, each task $T_{i,j}$ corresponds to all the queries in $G_{i,j}$, all of which are always executed together. We call this technique *conservative scheduling*. In our example, G_1 is partitioned into $G_{1,1}$ containing Q_1, Q_2 and Q_3 (with $n_{1,1} = 2$ minutes), $G_{1,2}$ containing Q_4 and Q_5 (with $n_{1,2} = 3$ minutes), and $G_{1,3}$ containing Q_6 and Q_7 (with $n_{1,3} = 5$ minutes). The resulting schedule is shown in Figure 4 (b).

Algorithm 2 summarizes conservative scheduling. Line 10 ensures that queries across sub-groups $G_{i,j}$ of the same group G_i are executed together when the `SLIDE` intervals of different sub-groups happen to coincide. For instance, every six minutes, $G_{1,1}$ and $G_{1,2}$ may be executed together.

The meaning of "joint execution" in line 10 depends upon the types of queries present in the group. In our example all the queries compute the same aggregate over the same attribute, but their window lengths are different. Therefore, joint execution means that the shared synopsis is scanned and answers over shorter windows are computed along the way (recall Section 2.2).

At this point, we remark that the following optimization applies to Algorithm 2 and its improvement that will be discussed later in this section. If query Q is already registered with the system and another query, Q', arrives that is identical to Q (including the same window size) except that its period is longer, then Q' can share the result of Q. In other words, if we are already computing the same query (Q) more often, then we may as well shorten the period of Q' and re-use the results generated by Q. Note that when the lifetime of Q expires, we may need to reset the period of Q' to its original value and possibly find a new sub-group $G_{i,j}$ for Q'.

Algorithm 2 Conservative scheduler

Input: sub-group periods n_i, task queue $Q(T)$
Local variables: array $d(T_{i,j})$, initially $d(T_{i,j}) = n_{i,j}$

```
 1  loop
 2    if update notification arrives
 3        update all synopses
 4        decrement d(T_{i,j}) for all tasks T_{i,j}
 5    end if
 6    if Q(T) is not empty
 7        let v be the lowest d(T_{i,j})-value of any task T_{i,j}
 8        let V(T) = {T_{i,j} | d(T_{i,j}) = v}
 9        choose any task T_{i,j} from V(T)
10        jointly execute T_{i,j} and any other task T_{i,m} in V(T)
11        reset the d(T)-values of all tasks just executed
12    end if
13  end loop
```

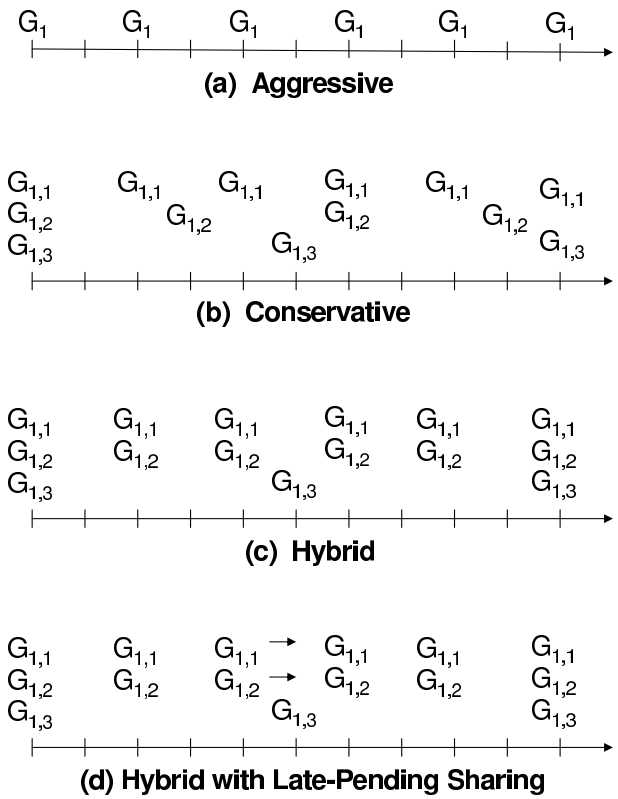

Figure 4: Execution schedules of queries in group G_1 using various scheduling techniques.

4.3 Hybrid Scheduling

The first of two improved scheduling techniques is called *hybrid scheduling*. Using a relative cost model, it determines whether it is more efficient for some sub-groups to be re-executed more often than necessary in order to synchronize their schedules with other sub-groups. The relative cost of a schedule may be computed by adding the synopsis access costs (i.e., how many intervals are accessed and what is needed to combine the pre-aggregated values from two intervals) as well as any post-processing costs (e.g., sorting frequency counts to find the top-k largest ones or combining answers from multiple synopses). Synopsis maintenance costs may be ignored as they do not vary across schedules. We introduce hybrid scheduling using queries Q_1 through Q_7 from Section 4.2; recall that they are partitioned by conservative scheduling into three sub-groups: $G_{1,1}$, $G_{1,2}$, and $G_{1,3}$.

The first step is to calculate the relative cost of a single re-execution of each sub-group. We can compute all queries in $G_{1,1}$ by scanning the ten youngest intervals of their `MAX` synopsis (answers over shorter windows, as needed by Q_2 and Q_3, will be computed along the way). The cost is 9 (comparisons to determine the maximum of ten maximum values stored in each interval). Similarly, the cost of $G_{1,2}$ is 14 and the cost of $G_{1,3}$ is 29. Next, we want to compute the execution cost of $G_{1,1}$ and $G_{1,2}$ incurred by conservative scheduling (recall Figure 4 (b)). Every six minutes (least-common-multiple of $n_{1,1}$ and $n_{1,2}$), both sub-groups are executed together for a cost of 14. Again, while scanning the synopsis to compute the maximum over the 15-minute window needed by Q_4, the answers of other queries are computed along the way. In the interim, $G_{1,1}$ is executed separately twice, for a cost of $9*2 = 18$, and $G_{1,2}$ once for a cost of 14. The total cost of executing queries in $G_{1,1}$ and $G_{1,2}$ is therefore 46 per six minutes, or 7.67 per minute.

Now suppose that $G_{1,2}$ is executed whenever $G_{1,1}$ is due for a refresh. In this case, both sub-groups are executed every two minutes. The cost per minute is $\frac{14}{2} = 7$. Therefore, the best way to execute sub-groups $G_{1,1}$ and $G_{1,2}$ is to schedule them both with a period of two minutes. In total, there are five possibilities: none of the sub-groups change their periods (which corresponds to conservative scheduling and costs 11.63 per minute), $G_{1,2}$ shortens its period to two minutes (11.4 per minute), both $G_{1,2}$ and $G_{1,3}$ shorten their periods to two minutes (14.5 per minute), $G_{1,3}$ changes its period to three minutes (12.67 per minute), and $G_{1,3}$ shortens its period to two minutes (16.83 per minute). Hybrid scheduling chooses the most efficient of these five possibilities, namely reducing the period of $G_{1,2}$ to that of $G_{1,1}$ and always executing queries in these two sub-groups together, as illustrated in Figure 4 (c). Note that dynamic programming may be used to memorize the optimal execution cost of overlapping subsets of the sub-groups in order to prevent duplicate calculations, giving rise to a polynomial-time algorithm for computing the optimal schedule.

Algorithm 2 may be used by hybrid scheduling without any modifications; the only difference is that the periods of some sub-groups are now shorter. Hybrid scheduling is expected to outperform aggressive and conservative scheduling as it performs the right amount of sharing—whenever appropriate, it shortens the periods of some queries in order to synchronize their re-evaluation times with other similar queries. However, hybrid scheduling is more expensive to maintain. As the query workload changes, we need to re-calculate the cost of separate and merged execution of various combinations of sub-groups. This is not expected to be a major source of overhead because our cost model is simple and the optimal schedule may be computed

efficiently.

4.4 Additional Sharing during Overload

We now show that additional computation sharing is possible during overload, when some tasks have negative $d(T_{i,j})$-values. We define the *late set* to contain all tasks $T_{i,j}$ with $d(T_{i,j}) < 0$ and the *pending set* to contain all tasks $T_{i,j}$ with $d(T_{i,j}) = 0$. Suppose that a particular execution of queries in sub-group $G_{1,1}$ is late with $d(T_{1,1}) = -2$. Suppose further that a particular execution of queries in sub-group $G_{1,2}$ is also late, but with $d(T_{1,2}) = -1$. Even though these two sub-groups may be scheduled together (they belong to the same group), Algorithm 2 (conservative scheduling) would not do this; it schedules $G_{1,1}$ and any other tasks with $d(T_{i,j})$-values of -2, before moving on to $G_{1,2}$. This is a missed sharing opportunity, which, if exploited, could help clear the overload faster. We propose an extension of conservative or hybrid scheduling, called *late sharing*, where matching sub-groups in the entire late set, not only those which have the lowest $d(T_{i,j})$-value, are scheduled together.

There are possibilities for even more sharing. Suppose that, in addition to $G_{1,1}$ and $G_{1,2}$ being late, $G_{1,3}$ has a value of $d(T_{1,3}) = 0$ at the current time. It is possible to schedule all three sub-groups together. Even though this shifts some of the system resources away from clearing the backlog of late tasks, it is beneficial in the long run because $G_{1,3}$ will not become late when the window slides again. Thus, the overall system throughput is likely to improve. We call this technique *late-pending sharing* and summarize it in Algorithm 3. It differs from Algorithm 2 in that it defines $V(T)$ to be the union of the late set and pending set during overload (lines 8 and 9). Note that late sharing could be implemented by replacing line 9 with "let $V(T) = \{T_{i,j} \mid d(T_{i,j}) < 0\}$".

A possible schedule produced by hybrid scheduling with late-pending sharing in the context of our example from Section 4.2 is illustrated in Figure 4 (d). As indicated by the arrows, suppose that $G_{1,1}$ and $G_{1,2}$ are late (with $d(T_{1,1}) = d(T_{1,2}) = -1$) and, at the same time, $G_{1,3}$ is pending (i.e., $d(T_{1,3}) = 0$). All queries in all three sub-groups are executed together by hybrid scheduling with late-pending sharing.

5 Experimental Evaluation

5.1 Setting

We implemented prototypes of our scheduling techniques and a simple query processor. The implementation was done in Java 1.4.2 and tests were performed on a Linux PC with a Pentium-IV 3 GHz processor and one Gb of RAM. The input consists of simulated IP packet headers with randomly generated attribute values. Initially, the number of distinct source and destination IP addresses is set to 1000 each, whereas the number of distinct protocols and ports is 100 each. We will discuss experiments on one particular query workload that display the relative differences between the scheduling algorithms. The workload consists of eight groups of persistent queries, each group computing top-k lists and quantiles over the bandwidth usage for one of the following: source IP address, destination IP address, source-destination pairs, protocol, port, and protocol-port pairs. Three workload sizes are considered: five, 10, 20, or 30 queries per group, giving a

Algorithm 3 Hybrid scheduler with late-pending sharing

Input: sub-group periods n_i, task queue $Q(T)$
Local variables: array $d(T_{i,j})$, initially $d(T_{i,j}) = n_{i,j}$

```
1  loop
2    if update notification arrives
3        update all synopses
4        decrement d(T_{i,j}) for all tasks T_{i,j}
5    end if
6    if Q(T) is not empty
7        let v be the lowest d(T_{i,j})-value of any task T_{i,j}
8        if v < 0
9            let V(T) = {T_{i,j} | d(T_{i,j}) ≤ 0}
10       else
11           let V(T) = {T_{i,j} | d(T_{i,j}) = v}
12       end if
13       choose any task T_{i,j} from V(T)
14       jointly execute T_{i,j} and any other task T_{i,m} in V(T)
15       reset the d(T)-values of all tasks just executed
16   end if
17 end loop
```

total number of long-running queries of 40, 80, 160, and 240, respectively. Each query has a randomly generated `WINDOW` length between 20 and 100 and a randomly generated `SLIDE` interval between one and half of its window length. Additionally, one-time queries requesting bandwidth usage statistics for a random source IP address are executed roughly every second.

Each group of queries shares a synopsis storing the appropriate counters. For simplicity, all the synopses are updated at the same time and all queries have the same priority. The queue $Q(T)$ (recall Algorithms 1 through 3) is implemented as a heap sorted by task deadlines. However, rather than storing time-to-deadline values and decrementing them whenever the synopses are updated, it stores the actual deadline times, which need to be revised and re-inserted into the heap only after a task has been executed.

Each experiment begins with a warm-up stage that fills the synopses. Next, each scheduling technique is executed for 200 seconds. This is repeated five times with different random inputs and the results are averaged. We measure the throughput, in queries re-executed per second, as well as the average latency per query, defined by the additional number of times its synopsis is updated between re-executions. For example, if a query requests a `SLIDE` of two seconds (i.e., two synopsis updates between re-executions) and the synopsis always slides three times between each re-execution, then its average latency is one. However, if a query is re-executed too early, then its latency remains at zero. Every experiment is repeated with two average data rates: 1500 and 3000 tuples per second. When using the higher data rate, we also double the number of distinct values of all the packet fields in order to force queries to do more work when generating answers (and cause overload). We omit results of varying the window size as the effects are

Table 1: Abbreviations of scheduling techniques

No-S	No sharing
AS	Aggressive scheduling
CS	Conservative scheduling
HS	Hybrid scheduling
H-LS	Hybrid scheduling with late sharing
H-LP	Hybrid scheduling with late-pending sharing

the same as when the data rates change (in both cases, queries must access more data during re-execution).

Table 1 lists the scheduling techniques and their abbreviations. As a baseline, we implemented a "no sharing" technique that is equivalent to Algorithm 1 with tasks corresponding to sub-groups $G_{i,j}$. That is, no-sharing is similar to conservative scheduling, but does not execute two matching sub-groups together, even if they are due for a refresh at the same time. We also implemented late sharing and late-pending sharing into conservative scheduling, but these always performed worse than when used with hybrid scheduling, and will not be discussed further.

5.2 Results with Low Data Rate

We begin by analyzing the results of experiments with a data rate of 1500 tuples per second. Throughput is shown in Figure 5 and average latency in Figure 6, for 40, 80, 160, and 240 queries. As expected, *No-S* and *CS* have the lowest throughput, with the gap between them growing as the number of queries increases and it is more likely that similar queries with different periods will be due for a refresh at the same time. Similarly, *No-S* and *CS* have high latency—*No-S* is particularly bad—even for a small number of queries, meaning that they cause system overload easily (overload occurs when the average latency is above zero). Note that *HS* alone easily doubles the throughput of *CS* and reduces its latency by an order of magnitude. In particular, we found that *HS* routinely shortened the periods of more than half the sub-groups in order to enable shared computation (recall Section 4.3). That is, if g sub-groups are used by conservative scheduling, it is often the case that hybrid scheduling requires only $\frac{g}{2}$ separately scheduled sub-groups (tasks). Additional improvements in throughput and latency can be gained via *H-LS* and *H-LP*, especially as the number of queries grows and the system falls into overload. Finally, note that *AS* achieves very good throughput, but poor latency. This is because many queries are needlessly refreshed before their deadlines.

5.3 Results with High Data Rate

Figures 7 and 8, respectively, graph the throughput and latency achieved with a data rate of 3000 tuples per second. In general, latencies are now higher and throughput is lower for all techniques due to heavier overload. *No-S* and *CS* continue to yield poor throughput and high latency (extremely high latency in case of no-sharing). However, the relative improvement of

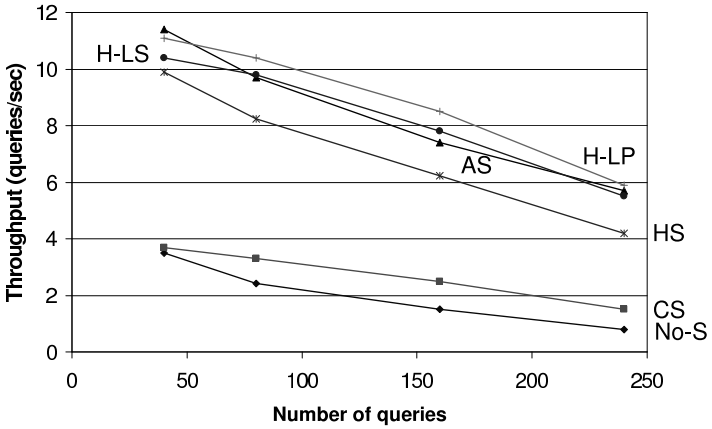

Figure 5: Throughput measurements using a data rate of 1500 tuples per second.

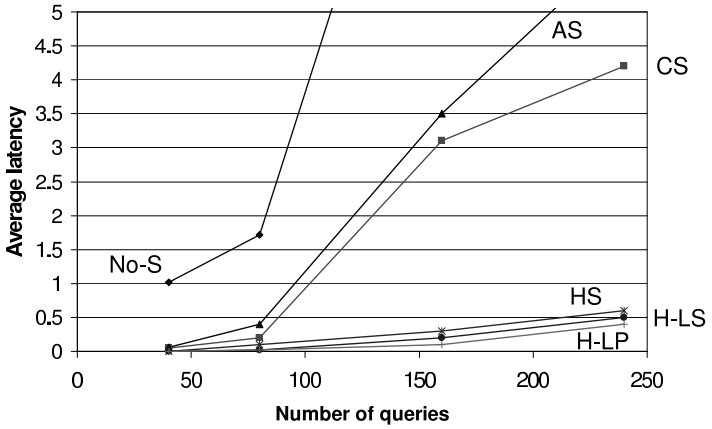

Figure 6: Average latency measurements using a data rate of 1500 tuples per second.

HS is now more modest, even though it continues to shorten the periods of more than half the sub-groups so that the queries within are always re-executed together. On the other hand, *H-LP* more than doubles the throughput of *HS* and is the clear winner in throughput and latency. This is because overload is now more severe, therefore exploiting additional sharing opportunities across queries with different deadlines is crucial. Curiously, the throughput of *AS* drops significantly in this experiment, most likely because it is now more costly to re-execute queries, especially those over long window sizes that would normally be refreshed sporadically (but are done frequently by *AS*).

6 Comparison with Related Work

There has been a great deal of recent interest in data stream management. In terms of multi-query optimization in DSMSs, much of the previous work concentrates on shared execution of filters and joins [11, 14, 19, 26, 30]. These works assume that incoming tuples are processed im-

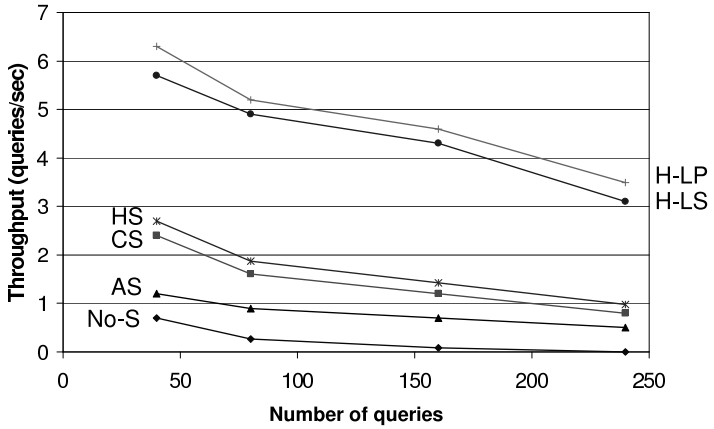

Figure 7: Throughput measurements using a data rate of 3000 tuples per second.

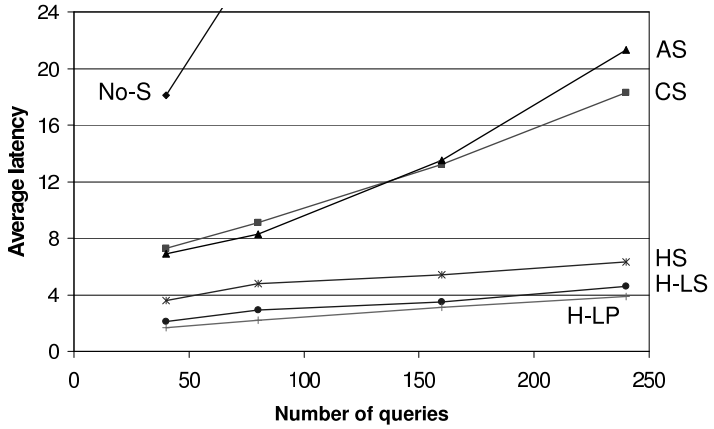

Figure 8: Average latency measurements using a data rate of 3000 tuples per second.

mediately, typically by updating any materialized joins affected by the new tuple and matching the new tuple against a query predicate index. Of the solutions that consider periodic query execution, one deals with sharing state among simple sliding window aggregates [6], another discusses state and computation sharing for queries with selection and sliding window aggregation [27], another discusses state sharing among aggregates having similar sets of group-by attributes [38], while another concentrates on periodic re-evaluation of selections over infinite streams and joins of streams with tables [12]. We are not aware of any work on synchronizing the schedules of similar queries so that they can be executed together.

Semantics of sliding windows are defined in [29], wherein a periodically-sliding window is represented as a sequence of overlapping extents. A new aggregate value is returned when an extent closes, i.e., when no more tuples will be mapped to it. This definition corresponds to the way we define synopsis updates: when the new interval computed by the buffer fills up, it is inserted into the synopsis. However, our semantics of the `SLIDE` clause separate the underlying synopsis updates from query execution times since it is not possible to execute all

queries instantaneously after every window-slide. To the best of our knowledge, our work is the first to make this important distinction.

There has been recent work on scheduling in DSMSs [7, 9, 23, 31]. These consider scheduling at the level of individual tuples and operators, i.e., choosing which tuple(s) to process at any given time. The goals are to bound the sizes of inter-operator queues or control output latency. To the best of our knowledge, our work is the first to discuss scheduling at the level of whole queries, with the goal of sharing computation among similar queries that may have different periods.

Also related to our scheduling approach is a deadline-based load shedding framework described in [37], where a window-slide and all pending query re-executions are dropped if the system can predict that there is insufficient time before the next window update (i.e., the next deadline) to perform the scheduled tasks. Our approach is different for the following two reasons. First, it does not require a mechanism for predicting the cost of advancing the windows and re-executing queries. Second, query re-executions are never dropped, but rather the periods of all queries are increased during overload. This avoids situations in which a query with a long period has one of its re-executions dropped and must wait a long time for the next refresh.

We adapted the earliest-deadline-first algorithm (EDF) for scheduling sliding window queries. As discussed in Section 4.1, the dynamic nature of the query workload, lack of reliable estimates of task completion times, and desire to prioritize late tasks rather than dropping them eliminate the use of other real-time and job-shop scheduling algorithms, among them rate-monotonic, shortest-time-to-completion, least-slack, and highest-value-first [2, 22]. Given that EDF is known to perform poorly during overload [21], one may wonder why we chose this algorithm as a basis for a DSMS query scheduler. The answer is that EDF performs poorly in terms of the goals of real-time systems, namely completing as many tasks as possible before their deadlines. EDF is not optimal in this case because it gives priority to transactions which will likely not finish on time because their deadlines are very close. However, EDF is a plausible technique in the context of a DSMS, where the goal is to re-execute queries with the desired periods. Consequently, it is better to prioritize queries with earliest re-execution deadlines rather than unnecessarily executing another query that is not due for a refresh yet.

Moreover, it is known that EDF is optimal in terms of minimizing the maximum task lateness [25]. However, one of the assumptions behind the proof is that tasks are executed separately. An interesting area for future work involves finding an optimal scheduling algorithm for the scenario presented in this chapter, namely shared execution of periodic tasks with the possibility of overload. (both in terms of minimizing the maximum task lateness and maximizing system throughput).

7 Conclusions and Future Work

In this paper, we identified and solved a novel problem in the context of multi-query optimization of periodically-refreshed persistent queries: sharing computation among similar queries that have different periods and therefore may be scheduled at different times. We argued that in addition to the traditional multi-query optimization step of common sub-expression matching, a DSMS must ensure that the schedules of similar queries are synchronized in order to take full advantage

of resource sharing. We then designed a query scheduling algorithm based upon the following insight: some queries should be evaluated more frequently than necessary if it means that their re-execution schedules can be synchronized with those of similar queries, thereby amortizing the computation costs. The proposed scheduling algorithm was also extended to uncover additional schedule synchronization opportunities during periods of system overload. Experimental results showed the advantages of our techniques under various system conditions.

In future work, we intend to explore whether our hybrid scheduling technique can be generalized to other scenarios involving periodic tasks, where executing some tasks more often than necessary (i.e., prior to their deadlines) leads to shared computation and increased throughput. We are also interested in extending our solutions to accommodate a wider class of queries, including those consisting of multiple pipelined operators. A possible issue in this context involves separate scheduling of parts of the same query.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug 2003.

[2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1–12, 1988.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 20–29, 1996.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, to appear.

[5] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 286–296, 2004.

[6] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 336–347, 2004.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, 2004.

[8] M. Cammert, J. Krämer, B. Seeger, and S.Vaupel. An approach to adaptive memory management in data stream systems. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 137, 2006.

[9] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 838–849, 2003.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 269–280, 2003.

[11] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, Aug 2003.

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

[13] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 29–38, 2004.

[14] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 551–568, 2004.

[15] P. Flajolet and G. N. Martin. Probabilistic counting. In *Proc. Symp. on Foundations of Comp. Sci. (FOCS)*, pages 76–82, 1983.

[16] L. Golab, K. G. Bijay, and M. T. Özsu. On concurrency control in sliding window queries over data streams. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 608–626, 2006.

[17] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 152–159, 1996.

[18] A. Halevy. Answering queries using views: a survey. *The VLDB Journal*, 10(4):270–294, 2001.

[19] M. Hammad, M. J. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 297–308, 2003.

[20] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 205–216, 1996.

[21] J. Haritsa, M. Carey, and M. Livny. Earliest-deadline scheduling for real-time database systems. In *In Proc. IEEE Real-Time Systems Symp.*, 1991.

[22] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *In Proc. IEEE Real-Time Sytems Symp.*, pages 112–122, 1985.

[23] Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *Proc. British Nat. Conf. on Databases (BNCOD)*, pages 16–30, 2004.

[24] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Disc. and Data Mining*, pages 91–101, 2002.

[25] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[26] S. Krishnamurthy, M. Franklin, J. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 972–986, 2004.

[27] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 623–634, 2006.

[28] W. Leland, M. Taqqu, M. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Trans. on Networking*, 2(1):1–15, 1994.

[29] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 311–322, 2005.

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

[31] Z. Ou, G. Yu, Y. Yu, S. Wu, X. Yang, and Q. Deng. Tick scheduling: A deadline based optimal task scheduling approach for real-time data stream systems. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 725–730, 2005.

[32] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, Jun 1995.

[33] T. Sellis. Multiple-query optimization. *ACM Trans. Database Sys.*, 13(1):23–52, 1988.

[34] N. Shivakumar and H. García-Molina. Wave-indices: indexing evolving databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 381–392, 1997.

[35] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering queries with aggregation using views. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 318–329, 1996.

[36] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[37] S. Wu, G. Yu, Y. Yu, Z. Ou, X. Yang, and Y. Gu. A deadline-sensitive approach for real-time processing of sliding windows. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 566–577, 2005.

[38] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 299–310, 2005.

[39] X. Zhang and D. Shasha. Better burst detection. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 146, 2006.

[40] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 358–369, 2002.

[41] Y. Zhu and D. Shasha. Efficient elastic burst detection in data streams. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Disc. and Data Mining*, pages 336–345, 2003.