No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams

Jin Li[1], David Maier[1], Kristin Tufte[1], Vassilis Papadimos[1], Peter A. Tucker[2]

[1]Portland State University
Portland, OR, USA
{jinli, maier, tufte, vpapad} @cs.pdx.edu

[2]Whitworth College
Spokane, WA, USA
ptucker@whitworth.edu

ABSTRACT

Window queries are proving essential to data-stream processing. In this paper, we present an approach for evaluating *sliding-window* aggregate queries that reduces both space and computation time for query execution. Our approach divides overlapping windows into disjoint *panes*, computes sub-aggregates over each pane, and "rolls up" the pane-aggregates to compute window-aggregates. Our experimental study shows that using panes has significant performance benefits.

1. Introduction

Many applications need to process streams, for example, financial data analysis, network traffic monitoring, and telecommunication monitoring. Several database research groups are building *D*ata *S*tream *M*anagement *S*ystems (DSMS) so that applications can issue queries to get timely information from streams. Managing and processing streams gives rise to challenges that have been extensively discussed and recognized [3, 4, 6, 7, 12].

An important class of queries over data streams is sliding-window aggregate queries. Consider an online auction system in which bids on auction items are streamed into a central auction processing system. The schema of each bid is: <item-id, bid-price, timestamp>. For ease of presentation, we assume that bids arrive in order on their timestamp attribute. (We are actively investigating processing disordered data streams) Query 1 shows an example of a sliding-window aggregate query.

Query 1: "Find the maximum bid price for the past 4 minutes and update the result every 1 minute."

```
SELECT max(bid-price)
FROM bids[WATTR timestamp
          RANGE 4 minutes
          SLIDE 1 minute]
```

In the query above, we introduce a window specification with three parameters: RANGE specifies the window size, SLIDE specifies how the window moves, and WATTR specifies the windowing attribute on which that the RANGE and SLIDE parameters are defined. The window specification of Query 1 breaks the bid stream into overlapping 4-minute sub-streams that start every minute, with respect to the timestamp attribute. These overlapping sub-streams are called *sliding windows*. Query 1 calculates the max for each window, and returns a stream with schema <max, timestamp>, where the timestamp attribute indicates the time when the max value is generated (the end of the window). Sliding window aggregate queries allow users to aggregate the stream at a user-specified granularity (RANGE) and interval (SLIDE), and thus provide the users a flexible way to monitor streaming data.

Current proposals for evaluating sliding-window aggregate queries buffer each input tuple until it is no longer needed [1]. Since each input tuple belongs to multiple windows, such approaches buffer a tuple until it is processed for the aggregate over the last window to which it belongs. Each input tuple is accessed multiple times, once for each window that it participates in.

We see two problems with such approaches. First the buffer size required is unbounded: At any time instant, all tuples contained in the current window are in the buffer, and so the size of the required buffers is determined by the window range and the data arrival rate. Second, processing each input tuple multiple times leads to a high computation cost. For example in Query 1, each input tuple is processed four times. As the ratio of RANGE over SLIDE increases, so does the number of times each tuple is processed. Considering the large volume and fast arrival rate of streaming data, reducing the amount of required buffer space (ideally to a constant bound) and computation time is an important

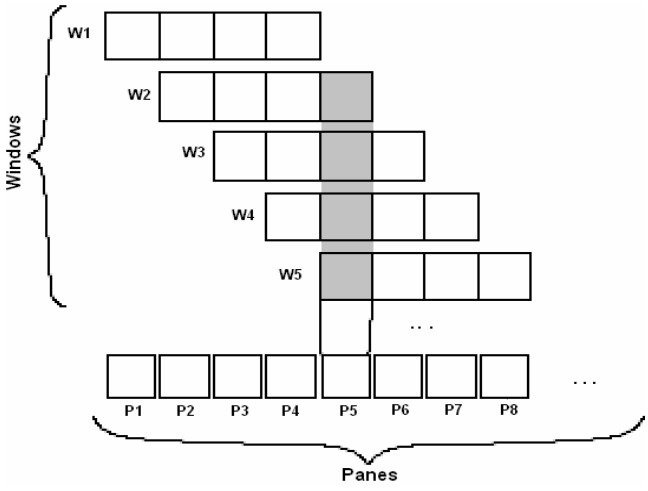

Figure 1: Windows Composed of Four Panes

issue.

We propose a new approach using *panes* for evaluating sliding-window aggregate queries that reduces the required buffer size by sub-aggregating the input stream and reduces the computation cost by sharing sub-aggregates when computing window aggregates. We sub-aggregate the stream over *non-overlapping* sub-sub-streams, which we call *panes*; then we aggregate over the pane-aggregates to get window-aggregates. Figure 1 illustrates how panes are used to evaluate Query 1. The stream is separated into 1-minute panes; each 4-minute window is composed of four consecutive panes. In Figure 1, w1 – w5 are windows and w3 is composed of panes p3 – p6. Each pane contributes to four windows; for example, p5 contributes to w2 through w5. To evaluate Query 1, we calculate the max for each pane; the max for each window is computed by finding the max of the maxes of the four panes that contribute to that window.

Intuitively, panes benefit the evaluation of sliding-window aggregates as long as there are "enough" tuples per pane. As we will discuss later, assuming the RANGE and SLIDE of a given sliding-window query are on the same data attribute (e.g., Query 1), the number of tuples per pane is determined by the RANGE, SLIDE, and the stream arrival rate. The type of windowing attribute (e.g., timestamp or sequence number) normally does not influence the performance with panes. Also, given a sliding-window aggregate query, the benefit of using panes normally increases as the number of tuples in each pane increases (i.e., as the average data arrival rate increases).

However, there is a particular type of sliding-window aggregate query used in some DSMSs [1] that panes do not help: In such a query, the window slides on every tuple. Thus, the SLIDE is fixed as *every tuple*. Query 2 is such a query expressed in our window specification.

Query 2: "Find the max bid price for the past 4 minutes."

```
SELECT max (bid-price)
FROM bids [WATTR timestamp
           RANGE 4 minutes
           SLIDE 1 tuple]
```

In Query 2, each input tuple defines a window and the query outputs the highest bid (max) in the last four minutes each time an input tuple arrives. The window operator in SQL-99 defines windows in a similar way. We call this type of window a *slide-by-tuple* window. More generally, *tuple-based* window is a window that slides by a fixed number of tuples (for example, "produce a new result every ten tuples"); a slide-by-tuple window is a special case of a tuple-based window in which the window slides by exactly *one* tuple. While panes can be beneficial for tuple-based windows, the benefit vanishes as the SLIDE approaches one tuple. However, for many stream applications, such as network-traffic monitoring where the data arrival is rapid, producing a result every time an input tuple arrives is neither realistic nor desirable. We believe that most window aggregates over high-volume streams will use user-specified granularity (RANGE) and interval (SLIDE) such as Query 1, and will thus benefit from panes.

This paper is organized as follows: Section 2 discusses related work; Section 3 describes how to use panes to evaluate sliding-window aggregate queries; Section 4 presents experimental results; and Section 5 concludes.

2. Related Work

Panes sub-aggregate the input stream, and in particular, the sub-aggregates are then shared by the aggregation of multiple windows (super-aggregation) of a single query to reduce both computation time and buffer usage. The concept of sub-aggregation and super-aggregation is used by the ROLLUP operator in SQL-99 and the data cube operator [8] to express aggregates at different granularities over stored data. The ROLLUP operator provides an efficient and readable way to express such queries and is most often used for aggregating data along a hierarchy, for example, city, state, and country. However, the ROLLUP operator functions on stored data and handles only slide-by-tuple windows. Holistic aggregate (e.g., quantile and heavy-hitter) evaluation in Gigascope [5] uses fast, light-weight sub-aggregation to reduce data for super-aggregation where expensive processing is performed. However, Gigascope only supports *tumbling* (non-overlapping) window queries. As such, Gigascope does not share sub-aggregates among multiple windows. Arasu and Widom [2] propose two algorithms, B-Int and L-Int, for shared execution of multiple sliding-window aggregates with the same aggregate function but different window sizes. Their algorithms maintain a data structure that stores the sub-aggregates over the active part of the stream at many different granularities. When a user polls a query, the aggregate over the current window is computed by looking up the constituent sub-aggregates stored in the data structure, and aggregating those values. B-Int and L-Int share a data structure among multiple queries to reduce computation cost, at the cost of increased buffer space usage. These algorithms do not support periodic result generation—results must be generated by polling.

3. Panes

In this section, we first describe the evaluation of sliding-window aggregate queries using panes. Then, we discuss in detail how panes are used for different types of aggregates. We use the online auction system introduced in Section 1 as our working scenario. For ease of presentation, we only discuss time-based windows, but the techniques can be easily extended to tuple-based windows.

3.1 Evaluating Queries with Panes

To evaluate a sliding-window aggregate query using panes, the query is decomposed into two sub-queries: a pane-level sub-query, *PLQ*, and a window-level sub-query, *WLQ*. The PLQ is a *tumbling-window aggregate query*, which separates the input stream into non-overlapping panes, and produces a *pane-aggregate* for each pane. The WLQ is a sliding-window query over the result of the PLQ that returns a *window-aggregate*.

Figure 2 shows the query plan for Query 1 using panes. This query, a sliding-window max, is decomposed into a tumbling-window max for the PLQ and a sliding-window max for the WLQ. The PLQ aggregates the input stream into a pane-max for each pane, and its output schema is <pane-max, pane-timestamp>, where pane-timestamp equals the timestamp value of the last tuple contributing to the pane. The WLQ runs over the stream produced by the PLQ, uses the pane-timestamp attribute as the windowing attribute, and every minute computes the max over the last four minutes. Each window of the WLQ contains four tuples.

To use panes, given a sliding-window aggregate query, the PLQ and WLQ (i.e., their window specifications and their aggregate functions) need to be determined. The PLQ and WLQ aggregate functions depend on the aggregate function of the original query. For example in a sliding-window count, the PLQ is a count, and the WLQ is a sum; for a sliding-window max, both the PLQ and WLQ use the max aggregate. Given the original query, the window specifications of both sub-queries are also determined—the intuition is that the size of the panes in the PLQ is the largest possible size for sub-aggregation such that the sub-aggregates can be used by the WLQ to compute window aggregates. Therefore, given a sliding-window aggregate query, the RANGE, as well as the SLIDE, of the PLQ is the greatest common divisor of the RANGE and SLIDE of the query: *pane-range* = *pane-slide* = *GCD*(*RANGE, SLIDE*). The WLQ has the same RANGE and SLIDE as the original query. The number of panes per window is *RANGE*/*pane-range*. Also, as in Figure 2, for time-based queries, the PLQ's windowing attribute is the windowing attribute of the original query, and the windowing attribute of the WLQ is the pane-timestamp attribute. From the discussion above, it is clear that the PLQ and WLQ of a given query can be constructed automatically. Also note that the implementation of panes, as shown in Figure 2, uses only window aggregate operators—it does not require any new query operators for panes.

Panes reduce both required buffer space and computation cost. The two major features of panes are that 1) the PLQ is a tumbling-window query: Each input tuple belongs to only one window, so each tuple is processed only once as it arrives and does not need to be buffered; and 2) the WLQ does less processing and buffering since it processes pane-aggregates instead of tuples. Although each pane-aggregate is processed multiple times by the WLQ, the overall computation cost for the query is normally reduced, because the number of panes in a window is usually much fewer than the number of tuples in a window. For example in Query 1, each input tuple is processed once to produce a pane-max. Then, each pane-max is used in the computation of four windows and is accessed four times, because each pane-max contributes to four windows. Generally, the number of tuple accesses here is much fewer than that of accessing each input tuple four times. In addition, by sub-aggregating the input stream, the PLQ significantly reduces the amount of input data for the WLQ, and thus the required buffer size for evaluating the query. Since we assume that tuples arrive in order, the WLQ only buffers the pane-aggregates contributing to the current window, and so the buffer size required by the WLQ as well as by the whole query is determined by the number of panes in a window. For example in Query 1, the WLQ buffers four max values—this is the only buffering required by panes to evaluate Query 1.

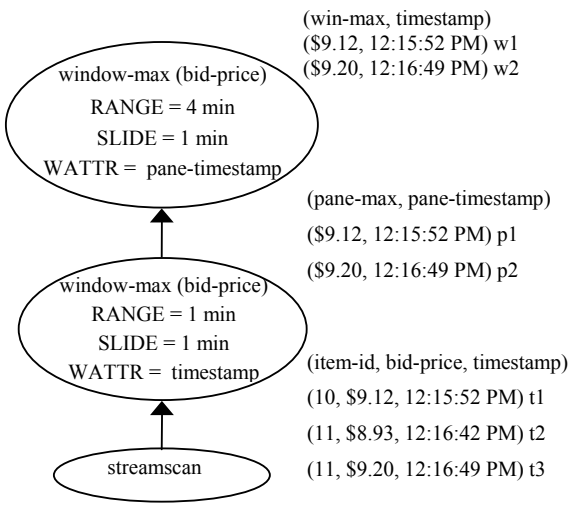

Figure 2: Using Panes to Evaluate Query 1

3.2 Different Types of Aggregates

We introduce two properties of aggregate functions that affect the evaluation of sliding-window aggregates.

3.2.1 Holistic

Suppose an aggregate function F over a dataset X can be computed from a "sub-aggregate" function L over disjoint datasets $X_1, X_2, \ldots, X_n$, where $\bigcup_{1 \le i \le n} X_i = X$ and a "super-aggregate" function S to compute $F(X)$ from the sub-aggregates, $L(X_i)$, $1 \le i \le n$.

$$F(X) = S(\{L(X_i) \mid 1 \le i \le n\})$$

As defined by Gray *et al*. [8], an aggregate function F is holistic if for any possible sub-aggregate functions L there is no constant bound on the size of storage needed to store the result of L. For example, median, quantile, and mode are holistic.

We call aggregates that are not holistic *bounded* aggregates. The term bounded encompasses the distributive and algebraic terms defined by Gray *et al*. [8]; the distinction between distributive and algebraic is unnecessary in our work. For example, average is bounded: The function L records *count* and *sum*; the function S adds the respective components and then divides to produce the global average. Other common examples of bounded aggregates include count, max, sum, variance, and center-of-mass.

3.2.2 Differential

Assume that there exist two datasets X and Y such that $Y \supseteq X$. Aggregate F is *differential*[1] if there exist such functions L, H and J that satisfy the conditions that $F(Y - X)$ can be computed from $L(Y)$ and $L(X)$ and $F(Y)$ can be computed from $L(Y - X)$ and $L(X)$ as below:

$$F(Y–X) = H(L(Y), L(X))$$
$$F(Y) = J(L(Y–X), L(X)).$$

We also require that $|L(X)| < |X|$.

For example, count is differential as shown below.

$$count(Y–X) = count(Y) – count(X)$$
$$count(Y) = count(Y–X) + count(X).$$

Based on the sub-aggregate function L, we further categorize differential aggregate functions. If the result of L can be stored with constant storage, F is *full-differential*. For example, count, average and variance are full-differential. A full-differential aggregate function must necessarily be bounded. Otherwise, if the result of L cannot be stored with constant bound, F is *pseudo-differential*, for example, the heavy-hitter aggregate that finds the frequently occurring items. Max is an example of an aggregate that is neither full-differential nor pseudo-differential.

3.3 Panes for Different Aggregate Queries

In this section, we discuss using panes to evaluate bounded and holistic aggregates. We also discuss the effects that the differential property and the number of groups defined by GROUP-BY construct have on evaluating sliding-window aggregate queries. In the interest of space, we discuss these two factors for bounded aggregates, but the discussion applies to holistic aggregates as well.

[1] Differential is similar to what Arasu and Widom [2] term as subtractable.

3.3.1 Panes for Bounded Aggregates

As discussed in Section 3.1, when using panes to evaluate sliding-window bounded aggregate queries (e.g., Query 1), the number of required buffers is bounded by the number of panes per window, and the pane-aggregates can be shared by the computation of multiple window-aggregates to reduce overall computation cost.

Given a differential aggregate function, we can exploit that property to further reduce its evaluation cost by computing the aggregate for the current window based on the aggregate of the previous window. Most differential bounded aggregates are full-differential, and so the required buffer size is still bounded when using panes. For example in Query 1, to compute the count over w3 as shown in Figure 1, we can use $count(w3) = count(w2) – count(p2) + count(p6)$. To take the advantage of the differential property, the aggregate operator (in the WLQ) needs to handle tuple expiration, as well as tuple arrival.

The GROUP-BY construct introduces another factor, the number of groups, into the buffering requirement and computation cost. Intuitively, the more groups, the more buffer space and the more computation are needed to evaluate the query. The following query is a sliding-window aggregate query with GROUP-BY.

Query 3: "Count the number of bids made on each auction item for the past 4 mininutes; and update the result every 1 minute."

```
SELECT count(*) FROM bids
GROUP BY item-id [WATTR timestamp
                  RANGE 4 minutes
                  SLIDE 1 minute]
```

Using panes to evaluate Query 3, each group in each pane is aggregated into a <item-id, pane-count, pane-timestamp> tuple by the PLQ. Assuming G groups per pane for the WLQ, a window contains $4*G$ tuples. In addition, the required buffer size for a sliding-window aggregate query is $P * G * sizeof(pane\text{-}aggregate)$ bytes, where P is number of panes per window and *sizeof* (*pane-aggregate*) is the number of bytes to store a pane-aggregate value. The number of groups per pane, G, is important because for each group the PLQ constructs an output tuple and the WLQ processes an input tuple. In the extreme case where every group contains only one tuple, the PLQ does not reduce the number of input tuples for the WLQ and panes provide no benefit. In fact, for a bounded aggregate query with GROUP-BY, the size of the required buffers is bounded only if the number of groups is bounded, and so the distinction between a GROUP-BY bounded aggregate and a holistic aggregate is blurred.

Taking both the number of groups and the differential property of the aggregate function into account, we express the cost per window-aggregate of using panes for sliding-

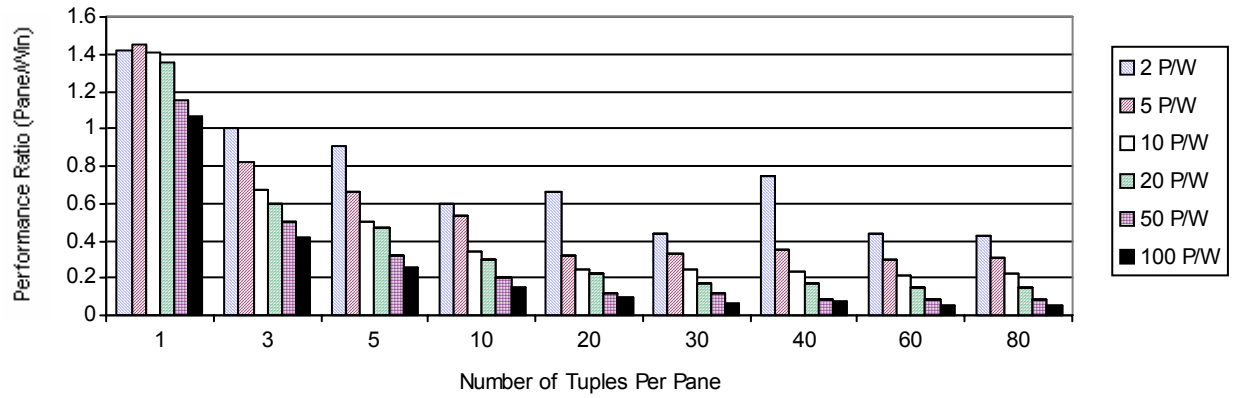

Figure 3: Cost Ratio of the Paned vs. the Windowed Approach

(P/W represents the number of panes per window)

window max and count, $Time_{P\text{-}M}$ and $Time_{P\text{-}C}$. Here, count is used to represent differential aggregates, and max is used to represent non-differential ones.

$$Time_{P\text{-}M} = a*T/P + b*G + c*P*G \quad (3.1)$$

$$Time_{P\text{-}C} = a*T/P + b*G + 2*c*G*SLIDE/GCD(RANGE, SLIDE) \quad (3.2)$$

In the two formulas above, a is the PLQ's cost to process an input tuple, b is the PLQ's cost to generate an output tuple, and c is the WLQ's cost to process a tuple (to insert a tuple into or to remove a tuple from a window); T is the number of tuples per window, P is the number of panes per window, G is the number of groups per pane. In formula (3.2), $SLIDE/GCD(RANGE, SLIDE)$ is the number of panes per slide. For example, when RANGE is 9 minutes and SLIDE is 6 minutes, the number of panes per slide is 2. Then, $2*c*G*\ SLIDE/GCD(RANGE, SLIDE)$ is the cost to compute the aggregate of the current window based on the aggregate of the previous window, that is, the cost to expire old panes and the cost to add new panes. The cost per window of evaluating a sliding-window max and count with current approaches, $Time_{W\text{-}M}$ and $Time_{W\text{-}C}$, are as follows, where a' is the cost to process each tuple (to insert a tuple to or to remove a tuple from a window).

$$Time_{W\text{-}M} = a'*T \quad (3.3)$$

$$Time_{W\text{-}C} = 2*a'*SLIDE*(T/RANGE) \quad (3.4)$$

Using existing approaches to evaluate a sliding-window max, we need to scan the entire window, just as Formula 3.3 indicates. To evaluate a sliding-window count, because count is differential, we can compute the count for the current window based on the count of the previous window by adding one to the previous window-count for each new tuple for the current window and subtracting one for each expired tuple. Comparing Formulas 3.1 and 3.3 (and 3.2 and 3.4), we see that there are some situations in which using panes might not yield performance gains: 1) When the number of groups per pane increases above a certain threshold; 2) when the data arrival rate is so slow that many panes are empty; 3) when the number of panes per window is small.

3.3.2 Panes for Holistic Aggregates

For holistic aggregates, although using panes cannot give us a constant bound on buffer size, it will in many cases reduce the amount of buffer space needed. In addition, the pre-processing of panes can be shared by multiple windows to reduce computation cost. We use heavy hitters as a holistic aggregate example, and use a method that is similar to that used by Gigascope to evaluate this aggregate.

Gigascope, a system for processing network-traffic data, can evaluate heavy hitter queries such as "find the IP sources that most frequently generate packets." To evaluate such queries in Gigascope, multiple alternatives are presented for sub-aggregate and super-aggregate pairs [5]. One option is that the sub-aggregate uses a hash table to record the packet-count for each IP source, and then the super-aggregate uses the hash table entries to update its data structure, called a sketch, for estimating heavy hitters. Although Gigascope only evaluates tumbling windows, we can use a similar method to evaluate heavy hitter queries, such as Query 4.

Query 4: "Over the past 10 minutes, find the ids of the auction items on which the number of bids is greater than or equal to 5% of the total number of bids; update the result every 1 minute."

To evaluate Query 4, the PLQ maintains a hash table with (item-id, count) hash entries. At the end of each pane, the non-empty hash table entries are output. The WLQ buffers and uses each hash table entry to update the sketches for multiple windows. Using panes, the PLQ compresses all the bids on an auction item to a single hash entry and reduces required buffer space, similar to the sub-aggregation in Gigascope. In addition, each hash table entry is used by

multiple windows, and thus reduces the overall computation cost. Similar strategies can be applied to evaluate other sliding-window holistic aggregates using panes.

We note that differential holistic aggregate functions are necessarily pseudo-differential. Consider heavy hitters: The count recorded by hash table entries can be summed or subtracted, so the sketch of the current window can be constructed based the sketch of the previous window; but there is no bound on the number of hash entries for each pane.

4. Performance Study

We implemented panes in the publicly-available version of Niagara Internet Query Engine [10], and empirically compared the evaluation of sliding-window aggregate queries with and without panes. Our experiments were conducted on an Intel® Pentium 4® 2.40 MHz machine, running Linux 7.3, with 512MB main memory. Our data generator is loosely based on the XMark data generator [13], and the data size for the experiments was approximately 15.2 MB. We calculated execution time by measuring the query execution time and then subtracting the cost of scanning the input stream, to focus on just the aggregation cost.

In our experiments, we varied the RANGE and the SLIDE parameters of a sliding-window max query, effectively varying the number of tuples per pane, and the number of panes per window (i.e., P/W, as shown by the different columns of each group in Figure 3). Figure 3 shows the ratio of the execution time using panes over the execution time of the current windowed approach (without panes). For example, we see that at 20 tuples per pane and 5 panes per window, the paned option takes about 30% of the time of the non-paned option. We see from Figure 3 that using panes has better performance than the original approach in most cases.

5. Discussion and Conclusion

In this paper, we presented a technique called panes, which reduces both the space and computation cost of evaluating sliding-window queries by sub-aggregating and sharing computation. We discussed using panes to exploit data reduction and computation sharing among multiple window-aggregate computation within a single query. We believe that panes can be extended to improve execution of multiple sliding-window queries over the same stream by sharing panes. We are also working on other aspects of processing streams, including formalization of window semantics, evaluation of window queries and processing disordered streams [9].

6. ACKNOWLEDGMENT

This work was supported by NSF grant IIS 0086002.

7. REFERENCES

[1] A. Arasu, S. Babu, and J. Widom. *The CQL Continuous Query Language: Semantic Foundations and Query Execution.* Stanford University Technical Report, October 2003.

[2] A. Arasu, J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB 2004)*.

[3] B. Babcock *et al*. Models and Issues in Data Stream Systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems (PODS 2002).*

[4] D. Carney *et al*. Monitoring Streams – A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB 2002)*.

[5] G. Cormode *et al*. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on the Management of Data (SIGMOD 2004).*

[6] C. Cranor, T. Johnson, O. Spatashek. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on the Management of Data (SIGMOD 2003)*.

[7] S. Chandrasekaran *et al*. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*.

[8] J. Gray *et al*. Data Cube: A Relational Aggregation Operator generalizing Group-by, Cross-Tab, and Sub Totals. *Data Mining and Knowledge Discovery* 1(1), 1997, 29-53.

[9] J. Li *et al*. Evaluating window aggregate queries over streams. Technical Report, May 2004, OGI/OHSU. http://www.cse.ogi.edu/~jinli/papers/WinAggrQ.pdf

[10] J. Naughton *et al*. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2), 27-33, (June 2001).

[11] U. Srivastava, J. Widom. *Flexible Time Management in Data Stream Systems*. Technical Report 2003-40, Stanford University, Stanford, CA (July 2003).

[12] The STREAM Group. STREAM: The Stanford STREAM Data Manager. *IEEE Data Engineering Bulletin*, 26(1), (March 2003).

[13] XMark Benchmark. http://www.xml-benchmark.org.