Knowing When to Slide – Efficient Scheduling for Sliding Window Processing

Ali Salehi
Ecole Polytechnique
Fédérale de Lausanne (EPFL)
Switzerland
ali.salehi@epfl.ch

Mehdi Riahi
Isfahan University
of Technology
Iran
m.riahi@ec.iut.ac.ir

Sebastian Michel
Ecole Polytechnique
Fédérale de Lausanne (EPFL)
Switzerland
sebastian.michel@epfl.ch

Karl Aberer
Ecole Polytechnique
Fédérale de Lausanne (EPFL)
Switzerland
karl.aberer@epfl.ch

***Abstract*—We consider sliding window query execution scheduling in stream processing engines. Sliding windows are an essential building block to limit the query focus at a particular part of the stream, based either on value count or time ranges. These so called sliding window predicates specify the execution condition for the query. Due to the often massive amount of registered queries, efficient algorithms to check these predicates are essential. While there exists a comprehensive set of works on the stream processing techniques, the actual algorithms to intelligently decide on the sliding behaviors is not extensively addressed in the existing works. In this paper we propose a set of algorithms for managing and sharing sliding decisions. This work introduces the concept of the batch sliding and sliding graphs to improve the sliding decision of the stream processing engines. Our algorithms can be efficiently used in large-scale stream processing systems where data arrives at high rates and a large number of user queries are registered to these data streams. Our evaluation results show the suitability of this approach in the real world applications.**

I. INTRODUCTION

With the advent of new sensing devices and recent advances in the wireless sensor network technology, the demand for providing a large scale stream processing platform for processing data produced by these devices is higher than ever before. We witness an immense increase of sensing devices, ranging from web cams to sophisticated sensor network deployments consisting of hundreds of wireless sensors capable of measuring a physical phenomena such as temperature, movement, light, and several others. Today, commercial entities are playing an active role in the streaming world by publishing real time data ranging from financial information (e.g., stock ticks) to entertainment information such as real time scores for a soccer match. These streaming data can be produced either by real sensors such as RFID readers (e.g., tracking parcels on a web site) or virtual sensors with no direct connection to physical world (e.g., network traffic).

In order to deal with the streaming data, the standard way is to specify a query with at least two extra properties associated with it, window size and sliding value. The window size is used to limit the actual data used for the processing (execution) to a certain range in time or number of values. The sliding predicate is introduced to specify the execution condition for the query. The execution of the query is triggered whenever the sliding condition is satisfied implying a possibly infinitely long periodic execution of the query.

For instance, one can express the interest of obtaining the average of a temperature sensor over the last 10 minutes, and doing so periodically every 2 minutes, by simply providing the window size of 10 minutes and sliding value of 2 minutes to the stream processing engine. As indicated before, each time the sliding condition is satisfied (e.g., 2 minutes passed from the previous execution) the actual action, computing the average over the last 10 minutes, is performed. Note that in some research papers the execution of the action is also called *movement of the sliding window*.

While there exist a comprehensive set of related work both on stream processing techniques and middlewares, the actual algorithms to intelligently decide on the sliding behaviors while the streaming data are arriving to the system has not been sufficiently addressed in the existing works. However, the problem becomes severe in scenarios with thousands of users registered to hundreds of high frequency data stream. One of our motivation applications, which suffers from the same issue, is called *NexTick*[15] which is a real time stock tick processing application architected to identify variety of trends by performing multiple technical analysis (TA) over the market to identify the best entry and exit points (lots of queries). NexTick is designed to be used on standard PCs and laptops and sends its recommendation messages in real time as the market moves which means it has to very efficiently handle the processing and notification events. NexTick itself is using many ideas from our generic stream processing engine GSN[1][4][5]. Our second motivation use case is initiated by the environmental scientists whom we are collaborating in the context of the *Swiss Experiment*[2]. In Swiss Experiment project, we face many different challenging questions, most of which we addressed through the GSN platform [1][4][5] (explained in more detail in section 2).

In above applications scenarios (among many other similar high demanding use cases) having algorithms to efficiently use the resources in order to decide *when and which* queries have to be executed can save both processing time and memory consumption, as verified later in the paper.

In this work, we provide a set of algorithms which can be used to efficiently decide on the processing time of the queries in the stream processing engine. We introduce a new query organization technique based on the sliding attributes of the queries and we provide algorithms for performing *batch sliding*. This work can be specifically useful for popular and

high rate streams such as stock ticks, sensor values for a renowned location (e.g., snow height in a popular skiing resort in winter) in addition to resource constrained environments such as mobile phones and PDAs.

The rest of this paper is organized as follows. Section II presents our motivation scenario. Section III presents the related work. Section IV presents the stream processing model which we consider in the paper. Section V presents a discussion on scalability issues. Section VI presents algorithms. Section VII presents the evaluation results of the algorithms and finally we conclude in Section VIII.

II. Motivation Scenario

As explained in the previous section, we are involved in two different high rate stream processing applications, the *NexTick* and the *Swiss Experiment*. While aforementioned applications appear at the first glance to be fundamentally different (one is using market information while the other one is monitoring the physical world), in our design, architecturally, they are exhibiting the same behavior. Both applications deal with the integration and processing of high rate data, and achieving that in a highly efficient way to support effective decision making process (e.g., buy or sell actions or sending alerts in the case of environmental monitoring sensors).

Thanks to strong acceptance of the Wireless Sensor Networks technology, more and more applications and user groups outside the core technology of wireless sensor networks started to benefit from this technology. One of the major user bases are in the environmental science. Wireless sensors bring environmental scientists the opportunity of fine grain monitoring of the physical phenomenon and that explains to some extent why environmental scientists are among the early adapters of this new technology. Using wireless sensor networks, not only the environmental scientists can address their research questions, but also computer and communication scientists can identify new opportunities to design more efficient and reliable systems.

In this section, we focus on the Swiss Experiment project as our motivation use case (very similar arguments can be applied to NexTick). In the context of Swiss Experiment, computer science researchers work closely in an inter-disciplinary collaboration with environmental scientists across multiple research centers in Switzerland. Interestingly, most researchers from the environmental science side are from different sub domains including snow and avalanche research, water quality, earthquake, understanding rapid mass movements, climate change, weathering, soil formation and ecosystem evolution. As one can imagine, introducing systems to address needs of over 10 different subgroups (although all are related to environmental science) can be both very interesting and challenging in nature.

Once the sensor data is captured (e.g., by means of wireless or satellite links) and transferred to the storage system deployed across the relevant research institute, it gets streamed to GSN[4][5] as our generic stream processing platform. GSN offers several services such as data sharing between multiple GSN servers, data storage, processing and filtering. Scientists can use GSN to express the processing logic and have GSN taking care of storage of the events, distributing the notification messages among other services (depicted in figure 1[1]).

The queries posted by the scientists in the Swiss Experiment, are including but not limited to data aggregation, statistical analysis (median, average, max and min), data quality estimation (through median average deviation) and analysis of extreme values (extreme statistics). Each of the aforementioned calculations has to be preformed in several time granularities (e.g., 15 seconds, 1 minute, 15 minutes, etc.) on the data streams generated by each sensor on every station. For instance, a wind sensor is deployed on each station with 8 other sensors (e.g., sun radiation, snow height, etc.), typically generate streaming data at 50 hertz. Considering an environmental research center such as Swiss Federal Institute for Snow and Avalanche Research in Davos with over 150 scientists, all of whom are interested in different aspect of above calculations and considering the sheer amount of sensor data piped into this institute (around 500 weather stations deployed around Switzerland) one can easily observe that the need for scalable management of the resources on the underlying stream processing system is not only beneficial but also essential to achieve reasonable response time. Note that the output of the GSN stream processing engine is typically delivered through reports and emails. For applications which need immediate attention of the scientists, GSN can deliver its output through sending SMS or making VoIP phone calls (through using Text-To-Speech engines). Applications which may need immediate attention are including but not limited to sending early warning (forecasting) and detecting the broken sensors (by comparing the patterns of the sensor data measured by a station to its previous measurements and other stations covering an overlapping region).

III. Related Work

As of today, there exists a few dozen of stream processing engines developed by different research groups. STREAM [6], Aurora [3] and TelegraphCQ [8] are some of the existing stream processing systems which support sliding windows on data streams. The different types of windows have been classified in [12] based on works in [11]. Processing data using the sliding concept is also recently added to commercial databases thanks to the new notations in SQL99 for specifying logical and physical windows on relations. Data stream query languages, such as CQL [7], StreaQuel [8], GSQL [10], and AQuery [13], often define their own notion of windows based on SQL99.

The work in this paper is also highly relevant to the data aggregation sharing methods introduced in [14] and [9].

In [9] the authors exploit similarities among the queries in order to share resources. The focus of the work is on

[1]The Wannengrat deployment consists of a set of solar powered weather stations deployed on the Wannengrat mountain in Davos, Switzerland. The stations are communicating through a long range point to point wireless network and through GPRS satellite link (the backup link).

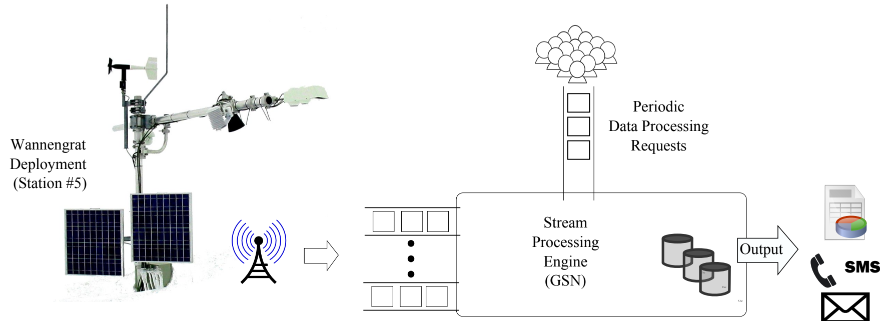

Fig. 1. Data Acquisition, Processing and Management Chain in the Swiss Experiment Project.

sharing streaming aggregate queries with different periodic windows without any up-front multi query optimization which are relying on complex static analysis. Authors introduce *Shared Data Shards* for sharing the processing cost among queries with different window sizes and predicates. The Shared Data Shards itself is actually the combination of two other optimizations, the *Shared Time Slices* approach which is designed for sharing processing cost among continuous queries with differing window size and the *Shared Data Fragments* approach for sharing processing resources among queries with different predicates.

The [14] proposes a technique called *panes* which reduces both the space and computation cost of evaluating sliding-window queries by sub-aggregating and sharing computation. The paper divides overlapping windows into disjoint panes, computes sub-aggregates over each pane, and compute window-aggregates by *rolling up* the sub-aggregates to compute the window-aggregates. The concept of sub-aggregation and super-aggregation is used originally by the ROLLUP operator in SQL 99 specification and the data cube operator to express aggregates at different granularities.

In order to handle timestamps in a distributed environment, [16] proposes a flexible heartbeat technique for application-defined time in a DSMS (Distributed Stream Management System). The proposed solution can handle time skew between streams, out-of-order data items within streams, and latency in streams reaching the data stream management systems.

In stream processing systems, continuous queries are executed whenever the sliding occurs. While all the aforementioned works provide different insights in defining and using window and sliding parameters on the streaming data, the actual window processing and sliding is always considered to be handled in a per stream bases thus there is no optimization performed on the way stream processing engines deal with these parameters. To clarify the difference, for instance, in the window-data aggregation works mentioned above, authors focus on sharing data in the aggregate queries at the execution time of the queries, our work is focused on introducing data structures and algorithms to optimize the sliding decisions (sliding decisions simply act as the triggers for the actual execution of a continuous queries). To the best of our knowledge, this work is the first attempt to optimize the way stream processing engines are handling the sliding parameters in large scale environments.

Our work exploits the similarities between the sliding specifications of the queries to plan a smart execution schedule. Once the query is scheduled to be executed, the optimizations in the above papers can be applied to share data and processing costs among the queries. We address both the time and count based sliding behaviors which are independent of the actual streaming data and execution plan. The ideas in this paper are designed so that they can be applied easily to any stream processing engine including those mentioned above. We assume that data elements arrive in the correct order to the sliding manager component. For handling out of order data elements, we can benefit from the techniques introduced by [16].

IV. System Model

In this section we briefly review the system model that we consider in our work. As we aim at introducing a new layer of optimization to stream processing systems, our goal is to be as general as possible.

Figure 2 represents a sample stream processing engine which uses two wireless sensor networks as its input streams. The first network is based on the TinyOS platform while the second network uses hand-held RFID readers communicating through WiFi with a base station (typically a wireless router). In contemporary stream processing engines, there exists a data interface component which is typically connected to a permanent storage such as a relational database, and a load shedder to monitor the current input rate in order to adapt the existing resources (which are consumed by the stream producers and users queries) with the preferred quality of service level.

In standard stream processing systems, users can post their queries to the system in order to get notified about occurrence of specific conditions and patterns on the data streams. These queries are stored inside the query repository that closely interacts with the query planner which in turn uses the stream statistics directory to generate an optimal execution plan for each query. The stream statistics directory contains the most recent statistical information (e.g., income rate and stream data size) regarding all the streams registered to the system.

Once the streaming data arrives, the sliding manager uses the query repository to generate a list containing all queries for which the sliding is imminent. This list is called the candidate query list. The query list is then delivered to the query scheduler which schedules the queries for execution by using arbitrary scheduling algorithms. Since we post a list of queries for the scheduler, it can consider batch processing of the queries. The queries are evaluated in the query execution unit that emits the results to the output delivery module. The notification system and output delivery module communicate with each other to notify registered users about new events. This model implies the essential role of the sliding manager in a stream processing system. Our focus in this work is on the sliding manager module of the stream processing engines. We propose algorithms to intelligently manage the sliding windows, thus improving the processing time and reducing the memory overhead.

V. Scalability Discussion

In this section we would like to show how other fields which are facing major scalability issues (in the stream processing context) can benefit from this work. Consider a financial market data processing applications. In these systems (e.g., NexTick[15]) the quick response time has the utmost importance. The actual processing of the market ticks are done in two stages. The first stage involves identifying which queries to execute at a given time. The second stage involves the actual execution process. In a typical setup of 10,000 symbols (the NYSE and Nasdaq together) with 100 different types of technical analysis (TA) performed on the price movements plus having these TAs preformed at multiple (typically 20) time granularities (e.g., 1 second, 5 seconds, 15 seconds, 1 minute, etc.), one can easily see how the scalability can become a real issue. Just in the aforementioned application, one has to deal with over 20 Million queries. Of course each query has to be also processed which itself implies huge lag between the decision time and the stock tick arrival time.

As in these setups the queries are posted to the system before data streams arrive to the system. For instance, the number of queries and TAs are typically fixed before the market opens, we normally can have the sliding graph already prepared. Now, whenever a tick is delivered to the system from the market, a typical naive approach has to evaluate all the registered queries to know which one to slide (e.g., 20 million queries in this case). Our approach fixes this stage by providing data structure and algorithms to intelligently handle the sliding values. The execution process of queries is orthogonal to this stage simply because in typical scenarios like above, we can easily use data grids and grid computing techniques to have the processing cost under our control. Note that in these applications, our algorithms for the query candidate construction phase combined with other optimization techniques at the execution time (likes those introduced by [14] and [9]) can provide a comprehensive toolkit to handle the performance issues in these kind of applications.

VI. Algorithms

In the stream context, there exist two types of sliding actions, time based and (tuple) count based. The time based sliding implies execution of the query in predefined and (possibly fixed) intervals. The count based sliding is used for triggering the query execution once a certain amount of data items (tuples) have been arrived to the stream processing engine. The amount of the data on which the query is evaluated is specified through the window property. The window property can be also specified using time or tuples. Given the above stream processing constructs one can come up with four different combinations listed below:

1) Count based window, count based slide (CBW-CBS)
2) Time based window, count based slide (TBW-CBS)
3) Time based window, time based slide (TBW-TBS)
4) Count based window, time based slide (CBW-TBS)

For sliding windows which have a count based slide (the first and second types), the case of *slide=1* is considered as a special case. In this case we simply do the sliding on arrival of each new tuple. There are also two different types of time which should be handled differently, *local time* and *remote time*. If the system time is used as the timestamp of tuples, we say these tuples are using local time. If the timestamp is set by the remote data source, we say tuples have remote time. Since each of these two types of time requires different treatment, sliding windows with time based slide are further divided into local time based and remote time based and therefore, we will have six different sliding window types: Two so called *count based sliding windows*, two so called *local time based sliding windows*, and in addition two *remote time base sliding windows*. In the following subsections we will explain different algorithms used for management of sliding for these three groups of sliding windows.

A. Sliding Graph

In this part, we start by describing the proposed sliding window management strategies and then we continue toward the concrete algorithms. The problem is to develop a method to reduce the time required for checking each sliding value to see whether its window must be slid or not. The straightforward method is to test all streams on arrival of each tuple. An obvious improvement is grouping those streams which have the same sliding value. This can already greatly reduce the number of comparisons on arrival of each new tuple.

We further improve the time by introducing a graph structure for sliding groups. This graph is based on the fact that in continuous queries that are issued by users, the sliding value of windows are often factors of each other. Suppose that after grouping of sliding windows, we have the following (count based) sliding groups: 2, 4, 8, 24, 15, 12, 5, 22, 3, 11, and 9. We know, for example, that 2 divides 2, 4, 8, 12, 24, and 22; also 3 divides 3, 9, 12, 15, and 24. These values are put in a directed graph such that for each edge the start node's value divides the end node's value and there is no other node in between them. If after the construction of

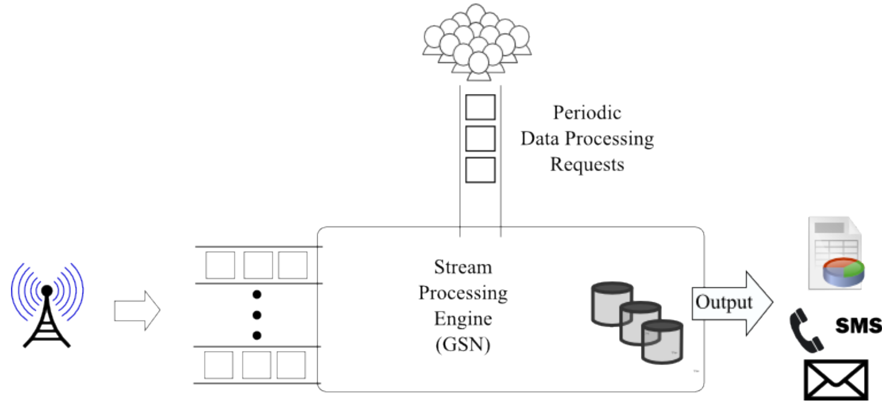

Fig. 2. General Model For Large Scale Stream Processing

the graph there is more than one node without any edges, a dummy parent node is created which uses the greatest common divisor (*gcd*) of its children as its sliding value. The only node without any incoming edges, which may be a dummy node, is called *root node*. Edges in the graph are either *strong* or *weak*; an incoming edge to a node is strong if its start value is the greatest among the start values of the other incoming edges. All other edges are weak edges and are used to simplify modifications to the graph. The resulting graph is called the *sliding graph*.

Figure 3 shows the sliding graph for the sample sliding groups mentioned above. Strong edges are expressed with solid line arrows and weak edges are depicted using dashed line arrows. As it can be seen in the figure, if we remove weak edges from the graph, what remains is a tree which is called *sliding tree* which we use it in our sliding algorithms.

B. Sliding for Count based Sliding Windows

Count based sliding is the simplest among three sliding window types. After constructing the sliding graph, we only need to keep track of the number of tuples received so far. On arrival of each new tuple Algorithm 1 is executed to create the candidate query list.

Algorithm 1

1: $tc \Leftarrow tc+1$ {updating current number of received tuples}
2: **if** tc mod $rootNode.slide = 0$ **then**
3: Push $rootNode$ to stack S
4: **while** S is not empty **do**
5: Pop n_s from S
6: Add requests of n_s to the candidate query list
7: **for** each child node n_{ch} of n_s **do**
8: **if** tc mod $n_{ch}.slide$ = 0 **then**
9: Push n_{ch} to S
10: **end if**
11: **end for**
12: **end while**
13: **end if**

In Algorithm 1 we actually perform a pre-order traverse of the sliding tree. The search is stopped at each node that its sliding value is not a divisor of the current tuple count. In this way we can eliminate a (possibly) large number of unsuccessful sliding value testing and hence reducing the execution time. For example, in the sample graph in Figure 3, when the tuple count is an odd number after testing divisibility by 2, the nodes 4, 8, 12, and 24 no longer need to be tested, while nodes 9, 15, and 22 are tested if the tuple count is divisible by 3, 5, or 11, respectively.

Window sizes are not considered in the algorithms and window size checking is left to the query execution system. If window sizes were considered in the sliding algorithms, the sliding groups might become very limited because we cannot put the sliding windows with the same sliding values and different window sizes into the same sliding group. Algorithm 1 does not consider the dynamic behavior of addition and deletion of queries to/from the system. It only uses the provided sliding graph. A separate simple algorithm is needed to update the sliding graph when a new query is introduced to the system or when an existing query leaves it. Since new sliding windows could be added to the sliding graph when the system is running, the first sliding round of new sliding windows may not be accurate, which is generally acceptable. After the first round, Algorithm 1 and other algorithms which are described in the next sections work as expected. For example, if 958 tuples have received so far and a new count based sliding window with sliding value of 10 is added to the sliding graph, this sliding window is scheduled for its first execution just after the arrival of two new tuples. After this first sliding round, the sliding window will be scheduled for execution after each new 10 tuples. The maximum performance of the algorithms is when the system is in a relatively steady state and there are a large number of registered queries on input streams.

C. Sliding for Local Time based Sliding Windows

In the simplest way, we use a local timer for each sliding window (or sliding group). The time unit or timer tick of each timer is set to its associated sliding value. This approach requires a large number of timers in the system and leads to more processor and memory usage. However we can use a

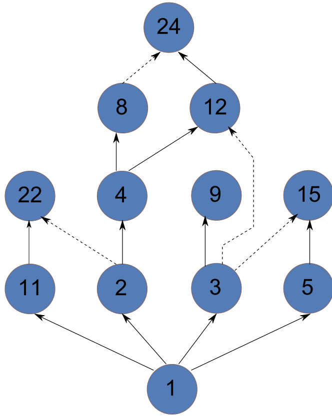

Fig. 3. Sliding graph produced for sample sliding groups

single timer for the all sliding windows defined over a data stream by setting its time unit to the *gcd* of the sliding values. Again we can use Algorithm 1 to reduce the number of slide testing on each timer tick provided that the tuple counter in the algorithm is replaced with a time unit counter. The time unit counter keeps the sum of time unit passed up to now. Suppose that the sliding values in Figure 3 are in seconds, so we can use them as an example for time based sliding windows. The *gcd* of these sliding values, and hence the timer tick, is 1 *sec*. The timer is scheduled to check the sliding windows every 1 *sec*.

D. Sliding for Remote Time based Sliding Windows

Handling remote time based sliding windows is more complicated than local time based sliding windows. The variable delays that networks put on the packets are the main source for the complication of the time management. These delays along with unsynchronized times may lead to out of order receiving of tuples. We assumed that a separate component is responsible for dealing with these out of order tuples. Using any approach to deal with the out of ordering, this subsystem delivers correct ordered tuples to the sliding manager component.

One simple solution is to check the sliding on arrival of each new tuple without using any local timer. On arrival of the first tuple, the *next slide time* is computed for all sliding windows and then the sliding windows are sorted in increasing order of next slide times. For each next tuple, the tuple's timestamp is compared with the updated next slide time of the first sliding window. If it is not greater than the timestamp, the window is slid and its next slide time is updated and then the next window is tested. If the test is not passed, other sliding windows won't be checked. At the end of slide testing, the sliding windows must remain sorted. Algorithm 2 represents these steps.

This algorithm has some drawbacks. First, window sliding may not be done at exact times. The second drawback is the worst case of the first one and it is postponing sliding of windows for a possibly long time. Suppose that the sliding value for a sliding window is 150 *sec* and new tuples arrive each 60±10 seconds. In some cases we must wait for 60 *sec* to receive the next tuple and then slide the window.

Algorithm 2

1: **for** each new tuple, tpl **do**
2: **if** tpl is the first tuple **then**
3: **for** each sliding window, SW **do**
4: $SW.nextSlide \Leftarrow SW.slide + tpl.timestamp$
5: **end for**
6: Sort sliding windows in increasing order of their $nextSlide$s
7: **else**
8: **for** each sliding window, SW **do**
9: **if** $SW.nextSlide \leq t.timestamp$ **then**
10: Add requests of SW to the candidate query list
11: $SW.nextSlide \Leftarrow SW.slide + tpl.timestamp$
12: **else**
13: Ensure sorting of sliding windows
14: Exit for
15: **end if**
16: **end for**
17: **end if**
18: **end for**

A different approach is to synchronize the local clock with the remote clock. The synchronization is based on the timestamps of the new tuples and therefore is an approximate method. The algorithm, which is done for each sliding group, works as follows. By arrival of the first tuple a timer is initialized, the timestamp of the tuple is set as the current time of the timer and the sliding value is set as its time unit. On each timer tick we delay the sliding by a computed value for the delay λ. If a new tuple arrives during this period, we ignore the delay and do the sliding. Each time a new tuple arrives, the value of λ is updated based on the following formula:

$$\lambda = \alpha\lambda + (1-\alpha)delay \tag{1}$$

Where *delay* is the difference of the tuple's timestamp and the current time of the timer, and α is a value between 0 and 1 which specifies the weight of the previous value of λ in the new value. In order to get a more accurate delay, the value of α can be refined during the execution of the algorithm to adjust the fraction of previous value of λ that affects the current estimated delay. Note that in this approach a separate timer is used for each sliding group and also some late tuples may be discarded. It is easy to realize that this approach requires more memory and processing time than the first one. It is up to the designer (or user) to choose between the simplicity of the first algorithm and the better accuracy of the second one.

E. Optimizing the sliding graph

It is possible to have some slide values none of them is a divisor of others but they may have some common divisors. Assume that we have these sliding values: 7, 8, 12, and 20. The sliding graph produced for them has been shown in Figure 4(a). We know that 4 is the (greatest) common divisor of 8, 12, and 20, so we can add a dummy node with slide value

of 4 to the graph to get the sliding graph in Figure 4(b). To see the effect of adding an extra node to the graph, we can compare the number of node testing in these graphs. For λ tuples, the number of node testing is $C_a = 5\lambda$ for graph (a) and $C_b = 3.75\lambda$ for graph (b). Therefore, the number of eliminated nodes to be evaluated will be $C_a - C_b = 1.25\lambda$, at the cost of adding one extra node to the original sliding graph. If the benefit of optimization be more than the cost of extra nodes, we can optimize the graph by adding some dummy nodes to it.

For each node n, node optimization O_n and node optimization factor θ_n is defined as follows:

$$O_n = \frac{C_0 - C_1}{C_0} \times \frac{1}{S_n} \tag{2}$$

$$\theta_n = \frac{O_n}{cf(\sigma)} \tag{3}$$

Where C_0 is the number of testing of the children of node n before optimization, C_1 is the number testing of the children of node n after optimization, S_n is the sliding value of the node, σ is the number of extra nodes created for the optimization, and $cf(\sigma)$ is the *cost function* that takes σ and returns the cost of σ node(s) as a real number.

In order to be able to optimize a sliding graph, two other optimization parameters are needed: the node optimization limit θ_l, which is the minimum value of node optimization factor, and the graph optimization limit θ_g, which is the lower bound on optimization of the graph. θ_l is a criterion for measuring the benefit of optimizing a node versus the cost of adding extra nodes. In other words, if optimization factor of a node is less than θ_l, then the cost of adding a node will become more than the benefit we gain from the optimization and this optimization should not be applied to the sliding graph. If more than one node is optimized, we should inspect that whether the sum of this optimization has a lower cost than the resulting benefit or it has a negetive effect on running time of the algorithms. The θ_g parameter is used not only to inspect this issue but also makes a compromise between the number of extra nodes (and memory consumption increment caused by them) and the value of graph optimization. Therefore, for a realistic optimization of a sliding graph, the mentioned parameters should be carefully selected. This selection could be based on the input rate of data, the way the graph is traversed, and the cost of extra nodes in this traversal.

Having these parameters, Algorithm 3 is used to optimize a sliding graph. The algorithm produces (in a greedy manner) an optimized sliding graph for the given parameters. Although it does not always produce the best optimization, it can build near optimal sliding graphs based on the given parameters. In this algorithm *n.sigma* is the number of extra nodes produced for optimizing node n. In order to compute possible optimizations at a node, it is necessary to compute all feasible combinations of child nodes based on their greatest common divisors. Although it is not a difficult problem, it might require heavy computations when a node has more than a specific number of children. In this case we can put a limit on the computation and continue with the best optimization resulted from this limited computation.

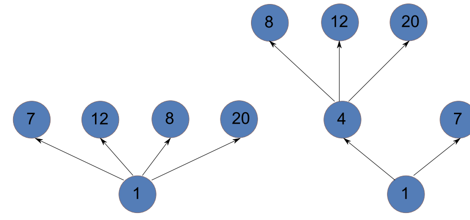

Fig. 4. Original and optimized sliding graphs

Algorithm 3

1: For each node n in the graph compute O_n ($n.opt$) and θ_n ($n.theta$).
2: If more than one optimization is possible for a node, select one optimization that its θ is maximum.
3: Add all nodes to the *nodeList*, except those nodes with $\theta < \theta_l$.
4: Sort *nodeList* in decreasing order of θs.
5: $sumOpt \Leftarrow 0$
6: $sumSigma \Leftarrow 0$
7: **for** each node n in *nodeList* **do**
8:     **if** $((sumOpt + n.opt)/cf(sumSigma + n.sigma)) > \theta_g$ **then**
9:         Optimize n in the graph
10:         $sumOpt \Leftarrow sumOpt + n.opt$
11:         $sumSigma \Leftarrow sumSigma + n.sigma$
12:     **else**
13:         **if** n has multiple optimizations **then**
14:             Among those optimizations with $\sigma < n.sigma$ select one of them with the maximum θ. If there exists such optimization, set it as the node's optimization and insert n in the proper place in *nodeList*
15:         **end if**
16:     **end if**
17: **end for**

VII. Evaluation Results

To evaluate the effect of using sliding graphs in the sliding algorithms, the execution time of our count based sliding algorithm has been compared with the execution time of the basic sliding algorithm which does not use sliding graphs. The basic sliding algorithm, which we call it the *plain algorithm*, checks all sliding groups on arrival of a new tuple. We use the following configuration for the evaluation: The number of continuous queries with sliding windows defined over a single stream is 10000, 5000, 1000, 100, 50, 30, and 10. The maximum values for slides are 200, 800, and 2000. Tuple production rate is 1000 tuples per second and each algorithm runs for 10 seconds (approximately 10000 tuples are produced). Sliding values are generated randomly in the range of [2..maximum sliding value]. We evaluate each algorithm 10

times independently, each time with different sliding values, and then the average execution time of the algorithms is used as the criteria for comparison of execution times. The evaluation was performed on a typical desktop PC with dual core, 2GHz Intel processor with 2MB cache, 1GB memory, running Linux kernel 2.6.24. We used the GSN platform as our stream processing system. The GSN platform and our evaluations have been written in Java language.

Figure 5 shows the result of the evaluation when the maximum value of sliding values is 200. It can be seen that the execution time is improved by the sliding graph algorithm (except for the case of 10 queries). As the number of queries increased the improvement of the sliding algorithm is getting better. As the number of queries decrease, the time required for traversing the graph reduces the effect of using the sliding graph. The results of the evaluation are shown in Figure 6 and Figure 7 when sliding values are less than 800 and 2000, respectively. The execution time of the sliding graph algorithm is between 3 and 3.7 times better than the plain algorithm for 10000 and 5000 queries in Figure 6 and Figure 7, respectively. We can conclude that our sliding graph gives better performance as the number of queries increased and the range of sliding values gets expanded.

To view the effect of optimizing the sliding graph, each sliding graph has been optimized with two different optimization parameters. The first optimization has been done with $\theta_l = 0$, $\theta_g = 0$ and $cf(\sigma) = \sigma$ and the second optimization has been done with $\theta_l = 0.001$, $\theta_g = 0.01$ and $cf(\sigma) = \sigma$. The first optimization creates the optimal graph without any restriction with the minimum number of extra nodes. As in Algorithm 1, a stack is used when we search the sliding graph. To have a more accurate comparison, the number of comparison operations and stack pushes are calculated. Each sequence of push-pop is estimated to be equal to 10-12 comparison operations and is added to the actual comparison operations. In addition to the previous evaluation parameters, each algorithm is executed for 100000 tuples and the number of comparison operations and stack pushes is calculated.

Figure 8 and Figure 9 show the result of the evaluation for the original sliding graph and the optimized sliding graphs with the two series of parameters mentioned above for sliding values that are less than 200 and 800. These figures show that the optimized sliding graphs reduce the number of operations required in the sliding algorithms that uses the sliding graph. Moreover, the first optimization works better than (or equal to) the second one. In general, we can deduce that by optimizing the sliding graph the execution time of the sliding graph algorithms is reduced, provided that the optimization parameters are correctly chosen. The second parameters have been randomly chosen while in practice one should specify them based on some criteria such as the amount of accessible memory, the graph traversal implementation, and the input rate of data.

Note that in the above evaluation results we only focused on count based sliding. As time is modeled using discrete integer values in the operating systems (e.g., number of milliseconds), one can use the exact same algorithms to handle time based sliding thus the evaluation results also covers the time based sliding queries.

In section VI-D we proposed two different approaches for handling remote time based sliding windows and claimed that the second approach results in more accurate sliding times than the first one (Algorithm 2). To compare the accuracy of the algorithms, a stream source is used which produces a new tuple every 2 minutes. A random delay between 30 and 90 seconds is put on each tuple before sending them to the sliding manager. Two queries are defined on this data stream, one with a slide value of 3 minutes and other with a slide value of 5 minutes. The system runs with both remote time based sliding algorithms and records the sliding times for each of them. Table I and table II show the time intervals between each sliding resulted from each sliding algorithm. As it can be seen in the tables, the second algorithm schedules the sliding windows to slide in more accurate times than the first algorithm. Algorithm 2 needs to wait for arrival of new tuples to decide on sliding while the second algorithm uses a timer to determine the sliding times and tries to synchronize this timer with the timer of the data source.

TABLE I
TIME INTERVALS BETWEEN EACH SLIDING IN REMOTE TIME BASED SLIDING ALGORITHMS WHEN THE SLIDE VALUE IS 3 MINUTES.

Algorithm 2	03:54	03:43	04:18	04:02	03:51	03:54
Using timers	02:56	03:39	02:58	03:06	03:12	03:29

TABLE II
TIME INTERVALS BETWEEN EACH SLIDING IN REMOTE TIME BASED SLIDING ALGORITHMS WHEN THE SLIDE VALUE IS 5 MINUTES.

Algorithm 2	06:03	05:49	06:22	05:48	06:20	05:52
Using timers	05:03	04:47	05:49	05:02	04:58	05:32

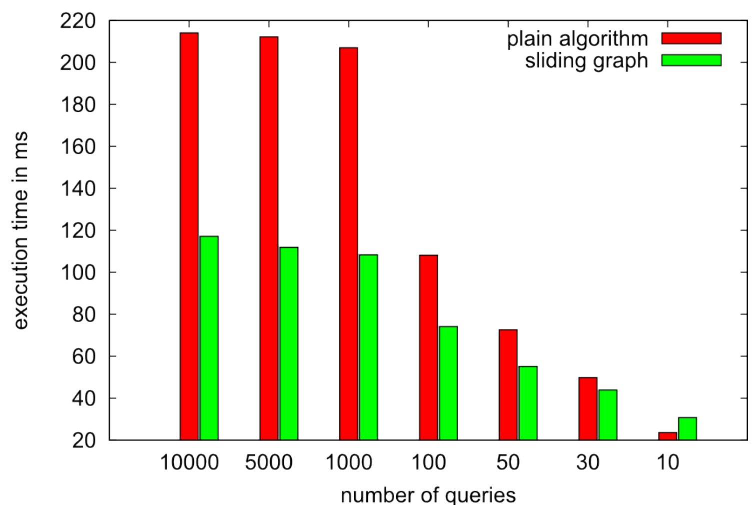

Fig. 5. Average execution times of algorithms with sliding values are ≤ 200.

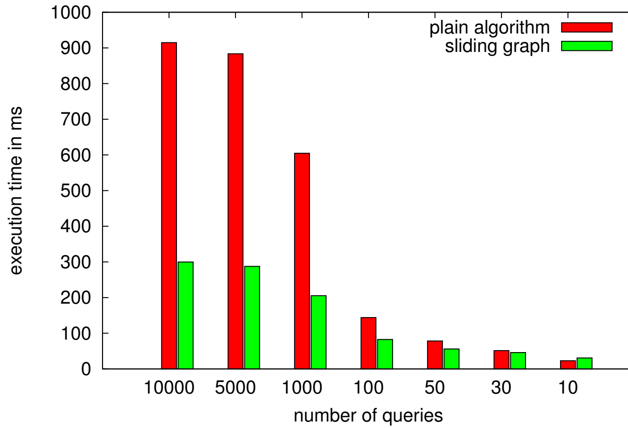

Fig. 6. Average execution times of algorithms when sliding values are $\leq$ 800.

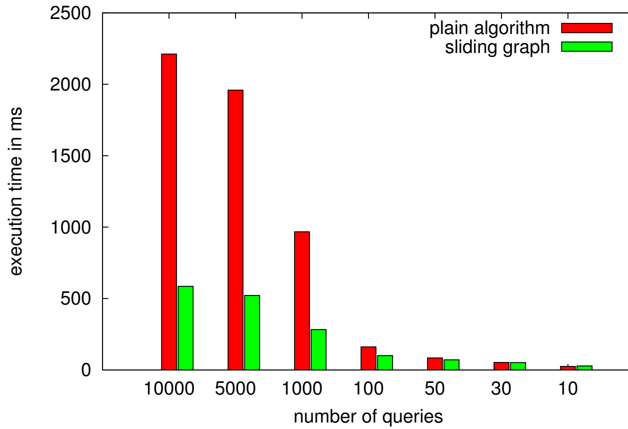

Fig. 7. Average execution times of algorithms when sliding values are $\leq$ 2000.

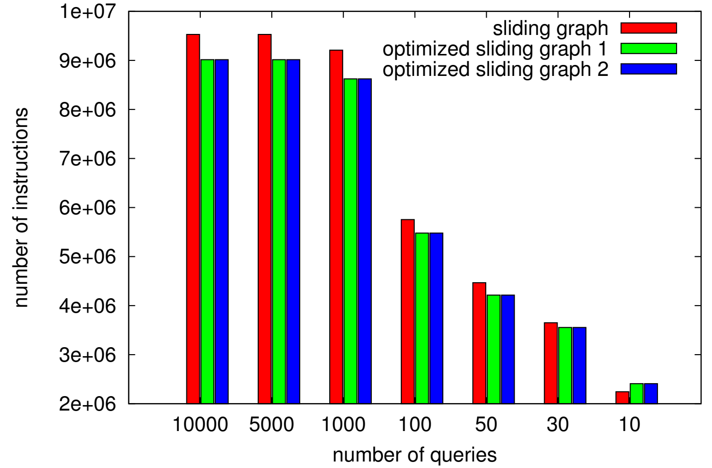

Fig. 8. Comparison of original and optimized sliding graph with sliding values $\leq$ 200.

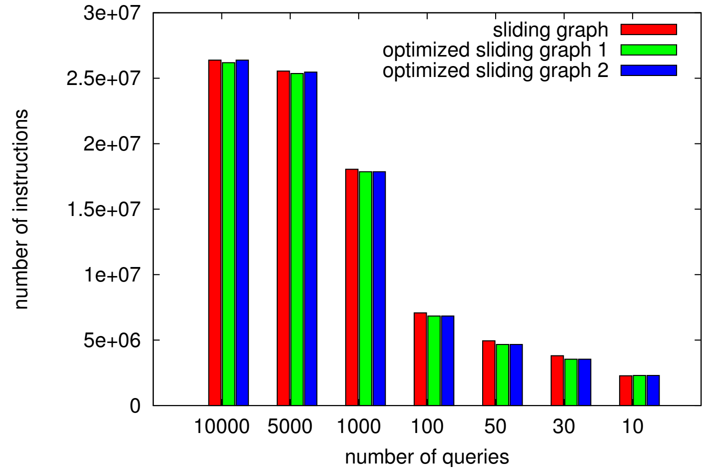

Fig. 9. Comparison of original and optimized sliding graph with sliding values $\leq$ 800.

VIII. Conclusion and Future Work

In this paper we have proposed a set of algorithms and techniques to deal with the management of sliding windows in stream processing systems. The proposed algorithms can be especially used in large-scale data stream processing systems in which there exist a large number of users registered to hundreds of high rate data streams. We address three possible types of sliding windows: tuple based, local time based, and remote time based. The sliding graph concept is introduced to reduce the processing time in sliding managers. Our evaluation results prove the efficiency of the sliding graph in the algorithms.

One interesting feature for the aforementioned stream processing systems is the use of approximate sliding window management. In this approach sliding windows which have close slide values could be grouped in the same sliding group to further reduce the processing time. In local time based sliding windows the number of timers was reduced as a result of using sliding graph. However, it is possible to enhance this algorithm either by merging similar timers which are used for different data streams or by using a few number of global timers in the system for all data streams. If the number of queries is large and data rate is very high, a single processing stream processor system might become overloaded.

Acknowledgments

The authors wish to thank Mathias Bavay and Nicholas Dawes among other people from the Swiss Federal Institute for Forest, Snow and Landscape Research in Davos, Switzerland for helping the authors in adapting the GSN stream processing engine to the environmental science research and IMIS system (network of over 400 weather stations located in Switzerland).

REFERENCES

[1] Global sensor networks (gsn). Website, 2008. http://gsn.sf.net.

[2] Swiss experiment. Website, 2008. http://www.swissexperiment.ch.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[4] K. Aberer, M. Hauswirth, and A. Salehi. A middleware for fast and flexible sensor network deployment. In *VLDB*, pages 1199–1202, 2006.

[5] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *MDM*, pages 198–205, 2007.

[6] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In *SIGMOD Conference*, page 665, 2003.

[7] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[9] S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 2006.

[10] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD Conference*, pages 647–651, 2003.

[11] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD Conference*, pages 13–24, 2001.

[12] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[13] A. Lerner and D. Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.

[14] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[15] A. Salehi. Nextick. Website, 2008. http://nextick.org.

[16] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.