Minimizing Latency and Memory in DSMS: a Unified Approach to Quasi-Optimal Scheduling

Yijian Bai
Computer Science Department
University of California, Los Angeles
bai@cs.ucla.edu

Carlo Zaniolo
Computer Science Department
University of California, Los Angeles
zaniolo@cs.ucla.edu

ABSTRACT

Data Stream Management Systems (DSMSs) must support optimized execution scheduling of multiple continuous queries on massive, and frequently bursty, data streams. Previous approaches on optimizing memory consumption or response time (i.e., latency) usually produce very different algorithms. In this paper, we extend the popular chart-partitioning procedure, which was previously used for memory optimization on simple operator paths, to minimize latency as well as memory on complex query-graphs with tuple-sharing forks. Furthermore, we test the performance of algorithms that only assume knowledge of the average behavior of tuples and operators, against a theoretical one that assumes detailed knowledge on the behavior of individual tuples. These experiments show that the practical algorithms closely approximate the performance of the optimal ones.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*

General Terms

Algorithms, Design, Performance

Keywords

Data Stream Management Systems, Operator Scheduling

1. INTRODUCTION

Data Stream Management Systems (DSMS) must support efficiently multiple continuous queries on massive and bursty data streams, and provide fast (near realtime) response upon tuple arrival [1, 15]. As multiple queries, each consisting of multiple operators, must share CPU time and other system resources, we need algorithms to optimize the scheduling of their complex query plans in order to achieve critical objectives, such as the minimization of memory and response time. Much previous research work has been focusing on this difficult problem. In particular, the Chain [2] algorithm to optimize memory consumption in operator chains has raised much interest because of its simplicity and its intuitive appeal. Memory can become a critical resource in bursty arrivals and overload situations, but fast response is a permanent concern for DSMSs which are often used to provide real-time or near real-time responses. Several approaches have thus been proposed to optimize latency and related objectives [9, 16, 7, 17]. However, we are still lacking a simple unifying approach which can be used for both latency and memory optimization as needed to simplify the design of actual systems. In this paper we provide such a unifying approach based on a generalization the well-known Chain algorithm. The generalization of this approach on more complex query graphs, e.g., those containing tuple-sharing fork structures, represent the second contribution of this paper. The third contribution is that we validate the existing practical algorithms, where only the average behavior of tuples is known, against the theoretically optimal one where the actual behavior of each individual tuple is assumed to be known. While the "average behavior" assumption is present in most previous work, the degradation w.r.t. optimality caused by this assumption has not been previously explored.

Related Work: Extensive DSMS research has resulted in a number of prototype systems [12, 13, 6, 10], for which the optimization of continuous queries has provided a major research topic [12, 5, 7]. Much of the efforts have been given to scheduling algorithms for memory or latency minimization. For latency minimization, [20] first proposed rate-based optimization for continuous queries, while [16, 7, 9] each proposed rate-based algorithm variants, and [17] discussed a *slowdown* metric and tuple-sharing optimization under this metric. For memory optimization, [7] devised a Min-Memory algorithm that is based on memory reduction rates, while [2] proposed to restructure the query graph based on progress chart partitioning. Other efforts related to continuous query optimization include the design of adaptive query processing engine [5, 8], methods for Quality of Service (QoS) control and load-shedding [11, 19], and various approximation methods. While the practical algorithms for latency and memory optimization usually are based on average-cost heuristics, no study so far provided any theoretically optimal algorithm or compared the performance of the practical algorithms against the optimal ones. Moreover, we propose new memory minimization algorithm for the tuple-sharing *fork* structure, and unify memory and latency minimization using a generalization of the charting-partitioning algorithm.

2. PROBLEM DEFINITION

The Query Graph The query operator graph has been widely used in DSMSs to describe the scheduling of continuous queries that are composed of multiple basic operators. In general, the nodes of the graph denote query operators and the arcs denote the buffers connecting them. Figure 1 shows an example where the graph is a simple path. The directed arc from Q_i to Q_j represents a *buffer*, whereby Q_i adds tuples to the tail of the buffer (production) and Q_j takes tuples from the head of the buffer (consumption). In addition to the actual query operators, the graph also contains *source* nodes and *sink* nodes as shown in Figure 1, which are filled/removed by external input/output wrappers, respectively.

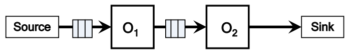
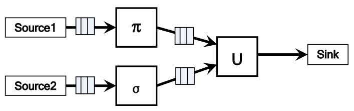
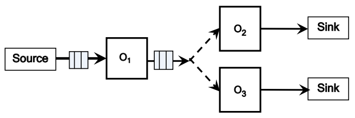


Figure 1: Simple Path Query

Union and join operators have multiple inputs, as shown in Figure 2(a). On the other hand, when two or more operators read from the same input, a *fork* structure is created (Figure 2(b)). Observe that the multiple operators following the fork share the same input tuples. Each consuming operator points to the tuple that it will consume next (i.e., the front of its private queue)[1].



(a) Simple Union



(b) Tuple-sharing *Fork*

Figure 2: Complex Topologies

In general, query graph in a DSMS can have several *query components*, where each component is a DAG. E.g., Figure 1, Figure 2(a) and Figure 2(b) each shows a *query component.* If we ignore the direction of edges, then each *query component* is a connected component in the underlying query graph.

The scheduling and execution algorithms on simple components and operator paths already exist in the literature (e.g., in [2, 7, 4]). We first briefly discuss them within the context of our chart-area-minimization based method framework, and then later build upon these methods to handle tuple-sharing forks.

Optimization Objectives. First, we clarify an important assumption in DSMS: the *Arrival-Order Constraint* which prescribes that the processing order of tuples on the same operator is exactly the same in which they arrived in its input buffer. This assumption is introduced to respect the time-dependent semantics of data streams. In this paper we consider the following estimates of costs.

1. Output Response Time (ORT). The *output response time*, or *output latency*, is measured on output tuples that exit the system (i.e., reach the *sink* nodes). The ORT for an output tuple is the time-span from the moment when all information required to derive the output tuple becomes available, to the time when the output tuple is actually created.

2. Memory-Based Cost. This cost measures the memory required for storing the bytes in the buffers over time. When an operator produces into a *sink* node where the tuples exit the system, the cost of an output tuple is normally considered zero [2].

Tuple Processing Charts. We can use a tuple delivery chart to illustrate the tuple processing progress. Suppose we plot output generation, then the tuple count starts from zero and ends at time T with the last tuple, the N-th tuple (Figure 3(a)). However, the cost is better demonstrated in terms of the number of tuples not yet delivered. Thus we have a different chart, where the tuple count starts from N and is decreased at any point in time by the number of tuples so far delivered (Figure 3(b)), we call this chart the Output Completion Chart (or, the OC Chart). The total area under this curve represents the total amount of time tuples stay inside the DSMS system. Thus, minimization of the area under the curve leads to the minimization of average latency.

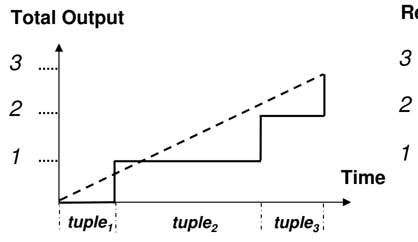
We can also plot memory reduction, where the vertical axis would be the total size, instead of total count, of tuples, which gives the Memory Reduction Chart (the MR Chart) (same as in [2]). Similar to latency optimization, minimization of area under this curve leads to minimization of memory consumption over time. Thus, the scheduling methods for both latency minimization and memory minimization used in this paper are based on a common objective—to achieve area-minimization on these progress charts, which becomes especially useful when handling tuple-sharing forks, as discussed later.

Cost Estimates, Averaging, and Smoothing. In a DSMS system, the true processing cost of any particular tuple is generally not known until it is finished, therefore such information is not available at scheduling time. However, the costs of tuples can generally be closely approximated by the average cost of tuples on the same operator. If we assume that all tuples behave in a similar way and require a time equal to the average time τ for processing, then the curve in Figure 3(b) becomes a straight line and the area S under our curve is equal to $S = \frac{1}{2}\, \tau\, N^2$ (Figure 3(c)), where N is total number of output tuples. To incorporate average *selectivity* σ (one input tuple produces σ output tuple on the average), the curve will now be vertically scaled by σ—the curve intersects the vertical axis at point $\sigma \times N$. Under these assumptions, the estimate for the total ORT cost is $\sigma\frac{1}{2}\tau N^2$.

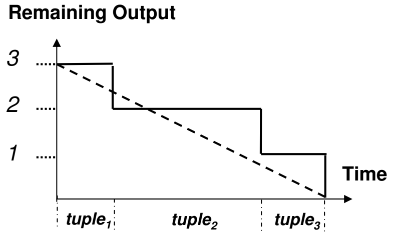
3. TUPLES, QUEUES, AND COMPONENTS

For memory reduction charts, cost of memory in bytes can be estimated to be $M = \beta \times S = \beta\frac{1}{2}\tau N^2$ where β is

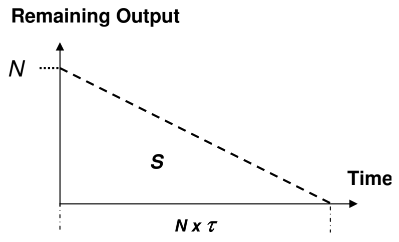
[1]The DSMS removes the tuples that have been used by all consuming operators.



(a) 3 Tuples: Positive Slope



(b) 3 Tuples: Negative Slope



(c) Many Tuples

Figure 3: Output Completion Chart for One Operator

the number of bytes (i.e., the cost in memory) reduced from input to output.

Now, suppose we have multiple operators in our system, each with an input queue. There are two possible models for the scheduling problem:

- **Tuple-based scheduling**. Under this model, the scheduling algorithm consider the cost of each tuple in the queues that feed into the n operators. A queue is selected according to some optimization criteria and a tuple is removed from the queue and processed on the operator. This operation is repeated, until all queues are empty. This approach considers exact cost of each tuple and has the potential of guaranteeing optimality (when optimal algorithms are available). However, it is normally unfeasible in practice. Furthermore, in DSMS where there are many small tuples, the scheduling overhead may be prohibitive. Thus existing algorithms generally adopt the component-based approach, discussed next. Later we will return to the tuple based approach to provide a provably optimal scheduling solution for time optimization. Then through experiments we will show that the practical component-based solutions are only mildly inferior to the optimal solution.
- **Component-Based scheduling**. In this model, we schedule execution on the basis of the *average cost* of the tuples on an operator. This is the practical approach of choice used in most existing studies. We next briefly review methods for simple components and operator paths, then extend them to tuple-sharing forks.

Simple Components A *simple component* consists of a single operator consuming the tuples from a single buffer. Suppose we have two simple components, component A and component B (shown in Figure 4(a)). Figure 4(b) and Figure 4(c) show the ORT costs for components A and B, respectively. Then Figure 4(d) and Figure 4(e) demonstrate two different schedules: in one schedule component A is scheduled first, and in the other schedule component B is scheduled first. Let τ_A denote the average time used by the tuples in A, and τ_B for that of B. Also, let N_A denote the total number of tuples processed for A, and N_B for that of B. Now consider the cost of first processing A, then B. As shown in Figure 4(d), the total cost for both components is:

$$S_{AB} = S_A + S_B + \tau_A N_A \times N_B$$

If we instead process B before A we obtain (Figure 4(e)):

$$S_{BA} = S_A + S_B + \tau_B N_B \times N_A$$

Clearly, A before B is better (has a smaller total area under the curve) iff $\tau_A < \tau_B$, and vice versa. Thus a component-based schedule will always select the component with greater slope. The result can be easily extended to multiple components, where we will then always schedule the components by decreasing slope (i.e., increasing average cost).

The slopes for different optimization criteria are as follows:

Output Generation Rate: For ORT optimization, the slope on the output completion chart represents output generation rate. I.e., suppose the average processing cost for an input tuple is τ and the average number of outputs generated is σ (the selectivity), then the slope is the number of output tuples generated in unit time: $R = \frac{\sigma}{\tau}$.

Memory Reduction Rate: For memory optimization, the slope on the memory reduction chart represents the memory reduction rate. I.e., again suppose the average processing cost for an input tuple is τ and the average number of outputs generated is σ (the selectivity). Moreover, suppose each input tuple has memory size M_{in}, and each output tuple has memory size M_{out}. Then the slope is the memory reduction achieved in unit time[2]:

$$R = \frac{M_{in} - \sigma \times M_{out}}{\tau}$$

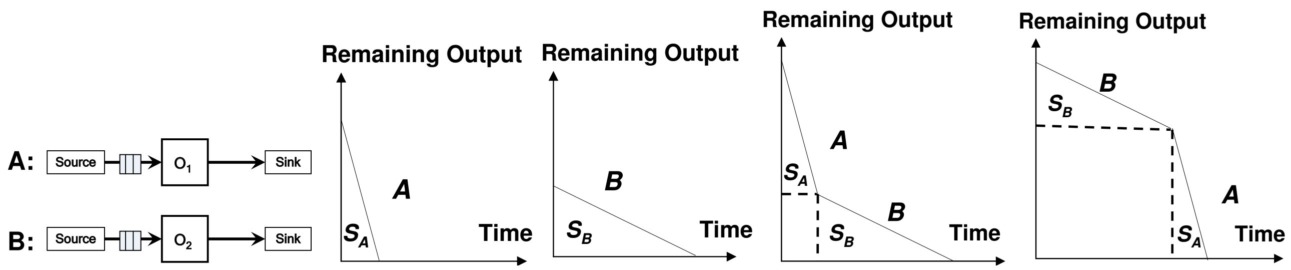
3.1 Inter-Component Scheduling

Now consider the case of multiple components, where each component contains multiple operators and we perform a two-level scheduling—first an *intra-component* scheduling step, followed by an *inter-component* scheduling step.

In the execution of each component, we will assume that each tuple in the input buffer is completely processed through the component, in a depth-first fashion (DFS). Thus in Figure 1, each tuple from the input buffer is first processed by operator O_1, and the tuples so generated are then processed by O_2. Only when this operation is completed, we return to O_1 which takes the next tuple in the input buffer and so on. In the fork situation of Figure 2(b), each of the tuple produced by O_1 is then processed by O_2 and O_3 (as discussed in more details in the next section)[3].

Once the component execution algorithm is specified, the scheduling problem can then be simplified: a component, no matter how complex, behaves as if it is a simple component with a given (average) response time and selectivity. Therefore, the inter-component scheduling step is always a trivial

[2]Note that if the output tuple exits the system (i.e., reaches a *sink* node), then M_{out} is considered as 0.
[3]The DFS processing of structures involving union and joins, such as that of Figure 2(a), was discussed in [4] and, because of space limitations, will thus be omitted here.



(a) Two Simple Components: A and B (b) ORT Cost for A (c) ORT Cost for B (d) A before B (e) B before A

Figure 4: Output Completion Chart for Different Schedules

extension of the simple component case above: the various complex components will simply be scheduled by charting them according to their decreasing slopes.

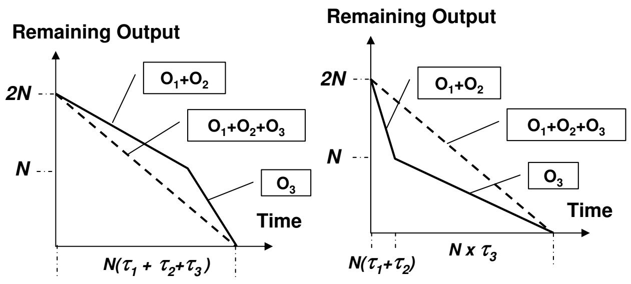
The complication lies in the fact that better optimization may be achieved by *breaking the original components into subcomponents*. The best example of this was provided by the Chain [2] algorithm that optimize memory for path queries, which we will not fully review here for space limitations. In the next sections we generalize the approach of Chain and provide optimization algorithms for both memory and latency on tuple-sharing fork structures.

4. COMPONENT PARTITIONING

Operator paths can be partitioned (segmented) to achieve better memory optimization [2]. For latency minimization, the breaking of a single operator path always makes the response time worse and thus it should never be broken [16]. However, latency optimization on tree shaped graphs containing forks requires partitioning at the fork, and memory optimization for forks is even more complex. Both these cases are discussed next.

Tuple-sharing Fork

The *fork* structure occurs whenever multiple operators share the same input (Figure 2(b)).



(a) DFS - No Partitioning (b) Partitioning at the Fork

Figure 5: Output Completion Chart for Fork

Output Response Time (ORT) Let us begin with the simple case where there are three operators, each with selectivity 1 (Figure 2(b)). Assume that τ_1, τ_2 and τ_3 are the average processing times of the operators O_1, O_2 and O_3, respectively.

For ORT, we will try to pick the output with the highest slope, i.e., the lowest average processing time. Therefore, suppose $\tau_2 < \tau_3$, we would process the first input on O_1 and then push the intermediate tuple through O_2 to get our first output. However, at this point we may either process another input tuple at O_1, or we may choose to process the intermediate tuple at the other branch, O_3. Now there are two possibilities:

- If $\tau_3 < \tau_1 + \tau_2$ (i.e., the slope of O_3 is higher than the combined slope of $O_1 + O_2$), we will process the tuple through O_3. This is basically the DFS strategy applied on the entire fork—for each tuple, we process it through each of its branches sequentially, ordering the branches by non-decreasing slopes (ties broken arbitrarily). This can be shown in the output completion chart in Figure 5(a). In the chart, the dashed lines represent applying DFS on the entire fork, while the solid lines represent process all available tuples on the first branch (O_2) before the second branch (O_3) (i.e., partitioning the fork). Observe that since we are now expecting twice as many output tuples than input tuples, the average processing time for one output tuple is $\frac{1}{2} \times (\tau_1 + \tau_2 + \tau_3)$. The area under the curve for DFS (no fork partitioning) is $N^2 \times (\tau_1 + \tau_2 + \tau_3)$, which is smaller than the area under the solid lines (with fork partitioning).

- If $\tau_3 > \tau_1 + \tau_2$ (i.e., the slope of $O_1 + O_2$ is higher than that of O_3), then we should try to generate the remaining output tuples on the branch $O_1 + O_2$ first. This corresponds to the solid lines in Figure 5(b), where the total area shows a cost of $\frac{N^2}{2}(\tau_1 + \tau_2) + \frac{N^2}{2}\tau_3$. Therefore, in this case we break this component into separate partitions—we apply DFS inside each partition, and use slope-based priority scheduling to order the partitions.

This approach can be generalized to arbitrary operator trees, as discussed next.

Chart Partitioning for Latency Minimization:

In the most general case, the operators have selectivity σ, i.e., on the average, the operator produces σ tuples for each tuple it processes. Moreover, multiple forks can be nested, as in the query graph in Figure 6.

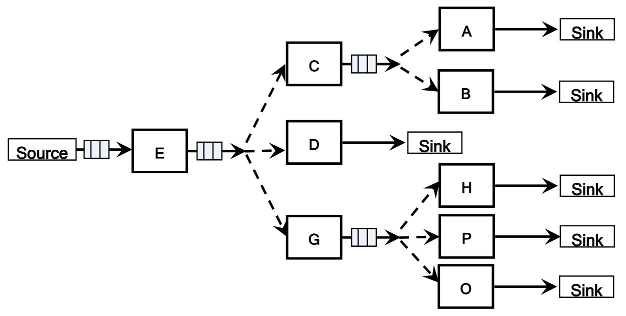
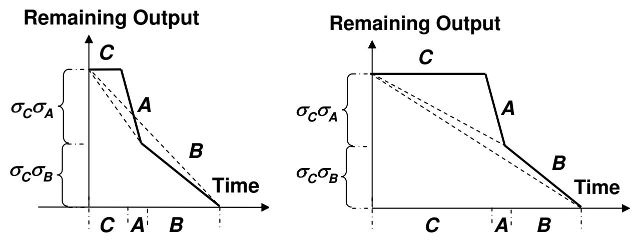


Figure 6: Nested Fork Structure



(a) Short C Segment - Two Partitions

(b) Long C Segment - One Partition

Figure 7: Possible OC Charts for Fork after Operator C in Figure 6

The chart partitioning on nested trees is a bottom-up algorithm that starts from the bottom operators (i.e., the *sink* nodes), and then proceeds upward (by processing the forks whose children have already been processed) until we arrive at root. For example, let us start processing the unordered tree of Figure 6 starting from the fork under operator C. The solid lines in Figure 7(a) give the output completion chart for our fork under the assumption that the output delivery rate (slope) of A is greater than that of B. Observe that segment C, which only produces intermediate tuples and thus has output delivery rate of 0, comes before the other segments for which the rate is always positive. This chart will be called the*unnormalized chart* for the subtree rooted on C.

Before we can proceed and move up in the tree, we must normalize this chart to a convex one using the standard chart partitioning algorithm of Chain. As shown in Figure 7(a), each of the dashed lines in Figure 7(a) correspond to a computation of a slope (the inverse of the cost per output)—therefore we compute the slopes for C+A, C+A+B, and pick the line with the steepest slope to form one partition. This could result into a chart containing one or two partitions, as shown in figure Figure 7(a) (two partitions) and Figure 7(b) (one partition). Observe that a very long C segment will result into a chart with one partition, where C, A, and B are processed in the same partition. The chart with two partitions implies that C and A will be processed together (using DFS), while B will be broken off to a separate partition.

This decision however could soon change, as the addition of E later can have the same effect as a longer C segment. By the same reasoning, we see that once the normalization process coalesces various segments into one partition, they will never need to be separated again later.

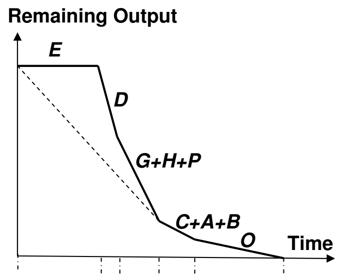


Figure 8: OC Chart for Nested Fork

After processing C, we compute the normalized chart for the subtree rooted on D and that rooted on G (siblings can be processed in any order). We can now move up and find the normalized chart for E. For instance, say that for C we have the one-partition situation of Figure 7(b), and that the subtree rooted in G was partitioned into two parts, $G{+}H{+}P$ and O, respectively. Then the unnormalized chart for the fork after operator E has four independent branches: D, $G+H+P$, $C+A+B$ and O. Now, we order the branches by decreasing slope [4] (Figure 8), then perform chart partitioning and derive the normalized chart (dashed lines) for the subtree rooted at E. Thus we obtain the chart of Figure 8, whereby our query graph would be broken into the following three partitions: $E+D+G+H+P$, $C+A+B$ and O.

Similar ideas can be used for memory minimization on forks. Previously studies in [2] made the assumption that separate buffers are created for each branch emanating from the fork. An efficient implementation would use one shared buffer, which produces a big saving in memory and is used in Stream Mill [4]:

<u>Memory Minimization on Forks</u>:

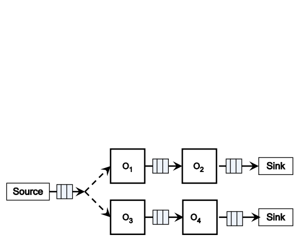
In the latency minimization algorithm above, chart-partitioning breaks the fork, however it never makes sense to break up each individual branch. This, however, is not the case for memory minimization, which may require multiple segmentation points on each branch of the fork.

Let's first consider a simple case, where two paths share the same external input (Fig 9(a)). If we consider each branch as independent paths, the memory reduction progress chart for the branches are shown in Fig 9(b) and Fig 9(c).

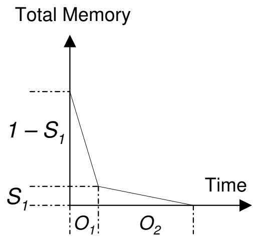
However, since the branches share common input, they are in fact not independent. We make the following observations:

- The first-operator of the first branch (O_1) no longer reduces total memory, due to the fact that the shared tuple can not be removed from memory until all branches consume it. Since a new tuple of size S_1 is produced, the first-operator of the first branch in fact increases memory!

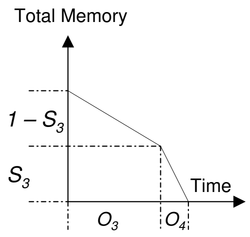
[4]By the nature of the chart partitioning algorithm, ordering partitions by slopes will never violate tuple dependency between operators. For example, in Figure 6, operator O should not be executed before operator G, since it relies on operator G to provide its input tuple. If operators O and G end up in separate partitions, by the partitioning process operator G will always be in a partition that has a higher slope than the partition operator O is in. Since we always order partitions by their slopes, this guarantees that operator G will be executed before operator O.



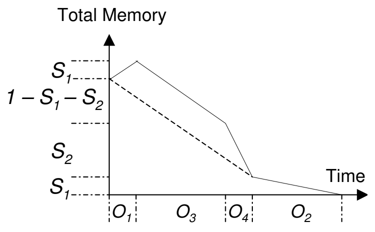
(a) Simple Fork Structure



(b) Memory Reduction of Branch 1



(c) Memory Reduction of Branch 2



(d) Partitioning of the Combined Chart for Both Branches

Figure 9: Memory Reduction Charts for Fork

- The first-operator of the second branch (O_3) in this example still reduces memory at its original rate. Upon completion of this operator, the shared input tuple is consumed by all branches, and thus can be cleared from memory and a tuple of size S_3 is produced. (In forks involving many branches, only the operator in the last branch reduces memory.)
- Every other operator in the branches maintains its original selectivity.

In fact, in a fork with multiple branches, we observe that *the first-operator of every branch, other than that of the last branch, becomes a memory-increasing operator (selectivity $\geq$ 1), even if its original selectivity is less than 1!*

This is a critical observation. Intuitively, the segment at the fork should always contain all the first-operators from every branch—otherwise we achieve no memory reduction at all (the shared input tuple can not be cleared from memory and the segment will certainly have a positive or zero slope)! Therefore for memory optimization, the fork itself should *never* be partitioned.

Now, after all the first-operators from each branch are processed, the shared tuple can be cleared and all the branches become independent paths again—we can simply segment these branches separately using the regular Chain strategy!

Therefore, with these crucial observations, we can present our chart-partitioning algorithm on fork for memory minimization. Intuitively, the algorithm groups all first-operators together first, then add fast-memory-reducing operator segments to this group, until the overall average memory reduction rate can no longer improve.

The following is the chart-partitioning algorithm on fork structures for memory minimization:

1. Starting from the second operator (i.e., leave out the first-operator, which consumes from the shared input), perform Chain on each branch. The branches are thus divided into segments, which we call the *unit-segments*, as they will be the unit of re-ordering later for the overall schedule. We also perform Chain on the commonly shared path before the fork, which generates some unit-segments too. For the example of Fig 9, excluding first-operators O_1 and O_3, operators O_2 and O_4 then each become a unit-segment.
2. Group all the first-operators from individual branches into one segment, and compute a combined memory reduction rate for this segment. We call this segment the *fork-segment*. For our example, the *fork-segment* thus consists of O_1 and O_3.
3. Draw a memory reduction progress chart for the whole fork structure—first draw the shared path before the fork; then draw the *fork-segment* as a group. Then, draw all the unit-segments from all branches (obtained from step 1 above) in the order of non-decreasing memory reduction rates. In our example above, this corresponds to draw O_4 and O_2 (each a unit-segment by itself here) after the *fork-segment*, in that order, as O_4 has a memory reduction rate higher than that of O_2.
4. Perform chart partitioning (i.e., *Chain*) on this chart. Effectively, the chart-partitioning procedure merges some unit-segments into *fork-segment* to achieve the fastest possible memory reduction rate. For our example above, this corresponds to merge O_4 into the *fork-segment*, while leaving O_2 remain as a separate partition, as shown in Fig 9(d).

As an example, after applying the algorithm, the fork in Fig 9(a) is divided into 2 partitions: O_1, O_3 and O_4 form one partition, and O_2 is another partition. This procedure greedily achieves a memory reduction rate that is as large as possible at the fork.

When there are multiple nested fork structures, again we apply our algorithm recursively in a bottom-up fashion: we first segment those fork structures closest to the *Sink* nodes—they are simple fork structures that do not contain nested forks. After applying chart-partitioning algorithm on them, the fork structure becomes serialized as if it is a path, and then we can repeat the segmentation process for the upstream fork structures.

Therefore, our algorithm for latency and memory optimization on fork structures is unified as different cases of chart partitioning—the difference between them lies in how to derive the "unnormalized" charts before the partitioning procedure. Multiple partitioning steps may be required on a single component, as shown above.

4.1 Unions and Joins

Extensions of chart partitioning for minimizing memory on union and joins operators was presented in [2], and therefore will not be further discussed here.

The minimization of latency in the presence of union and joins was discussed in [4], where it was shown that special techniques are needed to deal with the *idle-waiting* problem. This problem is caused by timestamp skews across multiple inputs for union/join operators[4, 18, 14]. For example, the union operator performs a sort-merge operation on the timestamp of the tuples. Thus, when one of the inputs becomes empty, tuples on all other inputs of the union will enter an idle-waiting state, until there are new arrivals on the empty input. The best method to solve the idle-waiting problem is on-demand tuple lookup and timestamp generation/propagation [4]. Briefly, when idle-waiting occurs, timestamp information for future arrival tuples are looked-up and propagated on-demand to unblock the waiting. Thus, for latency minimization, we treat the component with union/join as having an average output generation rate and do not normally perform any partitioning—the average output generation rate is computed based on the average behavior after performing timestamp propagation, with the output rate dependent not only on the tuple-processing costs, but also on the actual *tuple arrival rates* on the inputs.

5. TUPLE-BASED SCHEDULING: AN OPTIMAL ALGORITHM

A natural question on the component-based algorithms presented above is that, to which extent they in fact deliver an optimal result, as they assume an average cost for tuples? We now discus optimal scheduling algorithms that instead assume complete knowledge on the cost of individual tuples.

If it were the case that we could pick any tuple to process, regardless of their arrival times, then in scheduling we should always pick the tuple that has the least cost among all waiting tuples. However, in DSMS the extra constraint has to be observed that the tuples on the same input queue have to be processed in their arrival order. Therefore, we can not arbitrarily pick any tuple that has the least cost: we can only select among the head tuples in the queues.

There is an obvious greedy algorithm—we always pick the head tuple with the smallest cost among all current head tuples. However, this greedy algorithm is not optimal. Consider the following example: suppose we have two queues, one has a tuple with cost 10 time units followed by a tuple with cost 1, the other queue has a single tuple with cost 9 time units (we will refer to them as tuple c1, c10 and c9, respectively). If we only compare the head tuples, we would pick tuple c9 first, and the full order of processing would be c9, c10, c1. The output tuple from c9 would have a latency of 9 time units, the output of c10 would have a latency of 19 (9 units waiting, and 10 units processing), and output of c1 would have a latency of 20 (i.e., 9+10+1). The total output latency is thus $9 + 19 + 20 = 48$ time units.

However, under the order c10, c1, c9, which also satisfies the timing constraint, the total output latency is instead only $10 + 11 + 20 = 41$ time units. The problem of the greedy algorithm above is that, it only compares the head tuples, thus one single high-cost tuple (outliers) in a queue may hold back the processing of the other tuples in the same queue, even if the average cost of the other tuples is much lower. Therefore, we have to *look ahead* and have overall knowledge about all the tuples, before deciding on which head tuple to pick.

Maximum-Slope-Segment (MSS) Algorithm:

The Maximum-Sloped-Segment (MSS) algorithm again utilizes chart-partitioning, as follows:

1. We first plot output completion chart for the individual input tuples, where each input tuple generates corresponding number of outputs in a certain amount of time, as illustrate by the example of Figure 10 with two simple components with buffers A and B. Here we assume that we have precise information about tuple costs at scheduling time.

2. Using the chart-partitioning algorithm, we partition each queue of tuples into segments of tuples—we draw a line from the beginning of the first tuple to the end of each of the subsequent tuples, and pick the line with the steepest slope as the first segment, and repeat this process to identify all the remaining segments. (E.g., in the example of output completion chart (the OC Chart) in Figure 10(a), the queue of tuples are partitioned into 2 segments, while in Figure 10(b) the tuples are not segmented.)

3. schedule all tuple segments from all queues in the order of non-increasing slopes, i.e., non-increasing output generation rates. E.g., Figure 10(c) shows the OC chart for the combined schedule for all tuples on the two operators.

Intuitively, the MSS algorithm allows us to consider the costs of tuples beyond the head tuple on the queue. For the example in Figure 10, although $tupleA_1$ has a larger processing time than $tupleB_1$, the average processing time per output on buffer A is compensated by the 3 very fast tuples that follow $tupleA_1$. Therefore we schedule those 4 tuples together as a segment.

Note that in the previous figures (e.g., Figure 8) the partitioning is done on the operators, where each operator has a slope representing the average behavior. Here the partitioning is done on tuples that have precise costs.

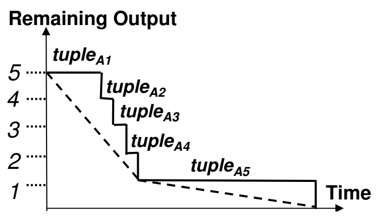
Optimality of the MSS Algorithm The MSS algorithm is optimal under the constraint of tuple processing order. Therefore, the schedule in Figure 10(c) gives minimum area under the output-completion curve (i.e., the total area under all the steps), among all possible schedules that satisfy the tuple-ordering constraint. In fact, we will show that:

1. The slopes for segments of tuples are the determining factor for total area under the curve: placing a higher slope before a lower slope always gives a smaller total area under the curve, than placing a lower slope before a higher slope. This is true regardless of the actual shape of the curve. E.g., in Figure 10(c) the order of the dashed slopes is the critical factor, regardless of the shape of the actual steps.

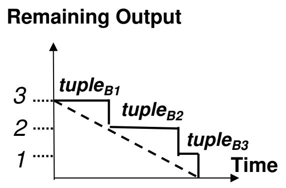
2. The MSS algorithm achieves minimum area under the curve—switching the position of any full segments (or partial segments) of tuples from this schedule would place a lower slope before a higher slope, thus lead to an increased total area under the curve.

The optimality of the MSS algorithm follows from these two properties. First, let us re-visit the operator scheduling example in Figure 4(c) and (d). It is easy to see that the total area under the curve is always smaller if we place a higher slope before a lower one.

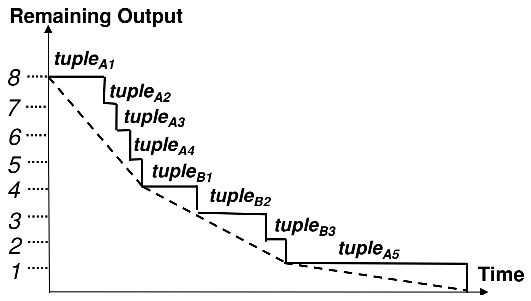
Now consider the output completion chart where individual tuples have different costs (i.e., different steps). Suppose we have two operators A and B, each with two tuples, and



(a) OC Chart for Buffer A



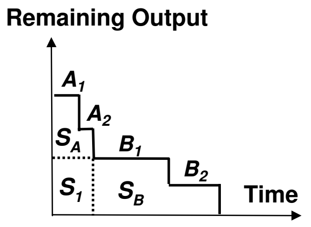
(b) OC Chart for Buffer B



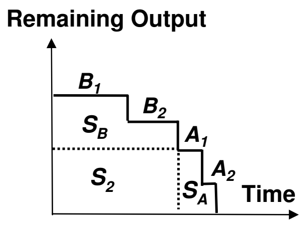
(c) Combined Schedule for Tuples in A and B

Figure 10: Tuple-based Scheduling

each tuple only produces one output. Let τ_{A1} be the processing time of the first tuple at operator A, τ_{B1} be that of first tuple at operator B, etc. Suppose $\frac{\tau_{A1}+\tau_{A2}}{2} < \frac{\tau_{B1}+\tau_{B2}}{2}$, i.e., the 2 tuples at operator A has a higher average slope.



(a) A before B



(b) B before A

Figure 11: Tuple-based Schedules Comparison

Figure 11(a) shows the schedule where the A tuples are scheduled first; compare it to Figure 11(b) where the B tuples are scheduled first. It is clear that, again areas S_A and S_B are shared in both schedules. The difference in total area under the curve lies in the difference of rectangle areas S_1 and S_2. We have:

$$S_1 = (\tau_{A1} + \tau_{A2}) \times N_B = \bar{\tau_A} \times N_A \times N_B$$

Here $\bar{\tau_A}$ is the average processing time per output tuple on A, and N_A is total number of tuples on A, and so on. Similarly, we have for S_2:

$$S_2 = (\tau_{B1} + \tau_{B2}) \times N_A = \bar{\tau_B} \times N_B \times N_A$$

Again, we should schedule A first, iff $\bar{\tau_A} < \bar{\tau_B}$, i.e., iff the tuples on A have a smaller average processing time cost (higher slopes). Therefore, although we no longer make the "average cost" assumption, what determines the total area is still the slopes—scheduling the segments of tuples with larger slopes first would give us a better schedule. The actual shape of steps in the chart is irrelevant, since those areas on the chart are shared by all schedules.

Now, we can prove the optimality by showing that the MSS algorithm gives minimum area under the curve—any reorganization of this schedule would lead to a larger total area under the curve.

There are two potential ways to change the order of the tuples from the optimal schedule:

- If we switch any full segments of tuples, it is clear that we can only move a segment with a lower slope to be before a segment with a higher slope, which would only increase the total area.
- Suppose we do not switch full segments of tuples, but only switch partial segments. For example, in Figure 11(a), instead of switching full segments, we may only switch tuples A_2 and B_1. We can prove that doing this will always switch a partial segment with a lower slope to be in front of a partial segment with a higher slope. To see why this is the case, notice that tuple B_1 can never have a higher slope than tuple A_2. By definition of the algorithm, we have $\bar{\tau_A} < \bar{\tau_B}$, i.e., tuple A_2 should belong to a segment of tuples with a higher average slope. Furthermore, by the way the segments are constructed, we always have $\tau_{A2} < \bar{\tau_A}$ and $\tau_{B1} > \bar{\tau_B}$. I.e., in order to include A_2 into segment A, since A_2 is at the end of segment A, it has to have a slope higher than the average slope of the segment. If A_2 had had a lower slope than the average, then it would not have been included into the segment. Similarly, tuple B_1 is at the front of segment B, it has to have a slope lower than the average of segment B. Therefore, we have $\tau_{A2} < \bar{\tau_A} < \bar{\tau_B} < \tau_{B1}$, and thus the switch of A_2 and B_1 will only increase the total area. These conclusions apply to any other sizes of partial segments too—any front partial segment would have a slope smaller than the overall slope of the segment, and any end partial segment would have a larger slope than average. By the ordering constraint, we can only switch the end part of an earlier segment with the front part of a later segment, (e.g., switch A_1 and B_2 would violate the timing constraint among A and B tuples). Therefore any switch of partial segments would always place a smaller slope to be before a larger slope, and thus increase the total area.

Since any possible re-arrangement of the schedule would lead to increased total area under the curve, it follows that under the tuple-ordering constraint the MSS algorithm gives the minimum area under the curve.

Note that the MSS algorithm can be slightly modified to apply on memory optimization for paths of operators, as we will briefly discuss in the next section.

6. EXPERIMENTS AND RESULTS

In this set of experiments, we evaluate how closely the average-behavior based component scheduling algorithm ap-

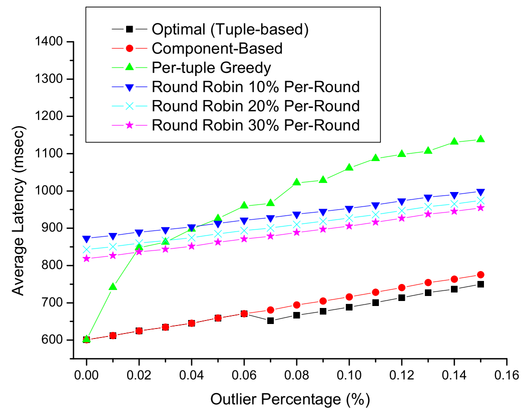


Figure 12: Average Latency by Outlier Percentage

proximate the tuple-based optimal algorithm. As the experiment results demonstrate, component-based algorithms generally results in performances very close to that of the optimal algorithms.

Experiment Setup To ensure prior knowledge of processing costs of individual tuples, the tuple costs are pre-generated based on different normal distributions, as stated in each experiment below. The performance measurements are done through simulated time calculations, thus excluding all actual scheduling overheads[5].

Output Response Time (ORT) Optimization In this experiment, we compare various algorithms for output response time (ORT) optimization. We assume that there are two independent operators A and B (i.e., two simple components), each with 300 waiting tuples on the input buffer. The selectivity of the operators are both set to 0.95. The tuples on operator A have pre-determined processing costs based on a normal distribution $\tau_A \sim N(1.0, 0.2^2)$ (i.e., a mean of 1.0 msec and standard deviation of 0.2 msec). The tuples on operator B have processing costs following $\tau_B \sim N(5.0, 1.0^2)$. Therefore, the average processing cost of tuples on A is much smaller than those on B. However, operator A also has a small amount of outlier tuples having a larger processing cost $\tau'_A \sim N(6.0, 1.0^2)$.

Under this setting, we compare the following scheduling algorithms:

1. The MSS algorithm, which optimally minimizes the average output response time.
2. The component-based scheduling algorithm that executes first the component with the highest output-generation rate.
3. Per-tuple greedy scheduling algorithm. This algorithm compares the head tuple on each buffer, and executes the tuple with the smallest processing cost.
4. Round Robin (RR) algorithm, which takes turns to visit each operator and process a fix number of tuples. In the experiments, we compare RR algorithms that on each visit process 30 tuples (10% of the total 300 tuples), or 60 tuples (20%), or 90 tuples (30%). In these experiments we always start the schedule by visiting operator A first (which has the higher average slope).

[5]Note that the optimal algorithm has to compute a schedule based on individual tuples, and thus generally has a much higher scheduling overhead than the component-based algorithm.

In Figure 12, we measure the average response time (average latency) of output tuples under increasing percentage of outlier tuples on operator A. The high-cost outlier tuples on A require real-time adaptation in the scheduling algorithm for optimal results. As expected, the average latency per output tuple generally increases with the outlier increasing (since the total time required to process all tuples increases). Furthermore, from Figure 12 we can also draw the following conclusions : i) The per-tuple greedy algorithm, which only compares the cost of the head tuple in each queue, is affected the most by outlier tuples. ii) when the outlier level is under 6% of A tuples, the optimal algorithm MSS and the component-based approximation have almost identical performance. Overall the component-based algorithm approximates the optimal algorithm very well—the average latencies from the two algorithms always stay within 4% of each other. In comparison, the greedy algorithm results in latencies that are up to 50% higher than that of the optimal algorithm. iii) The Round Robin (RR) algorithms confirm that alternating between different buffers generally increases output latency significantly.

As a conclusion, the practical component-based algorithm achieves performances very close to that of the theoretical and unrealistic optimal algorithm.

Memory Optimization

We first describe a tuple-based algorithm for memory optimization on a single operator path. Assume that we have an operator path that consists of two operators O_1 and O_2, where O_2 consumes the output of O_1 (as shown in Figure 1). The scheduling algorithm to optimize for memory on this path is a slight modification of the MSS algorithm for ORT optimization, as follows:

1. The algorithm partitions input tuples into segments exactly as done in MSS, but uses slopes of memory-reduction rates for partitioning.
2. The algorithm recomputes the tuple-segmentation decision on both operators whenever an intermediate tuple arrives at O_2 (i.e., whenever a tuple is produced from O_1).

Thus, this algorithm re-evaluates after each tuple arrival at O_2, whether it is better to push the current intermediate tuple to process on O_2, or to bring down another tuple from O_1, based on the best memory reduction rate achievable over a segment of tuples on both operators.

In the experiment, we will compare the following scheduling algorithms: i) The tuple-based algorithm for memory optimization on a path, as described above. ii) The component-based Chain algorithm [2], which will break the path into two partitions. iii) Per-tuple greedy scheduling algorithm, which compares the first tuple on both buffers only. iv) Round Robin (RR) algorithms.

For the experiments, the selectivity of the two operators are both set to 0.95. Operator O_1 reduces the size of its input tuple by half, and has 300 waiting tuples on its input buffer. Operator O_2 consumes from the output of O_1

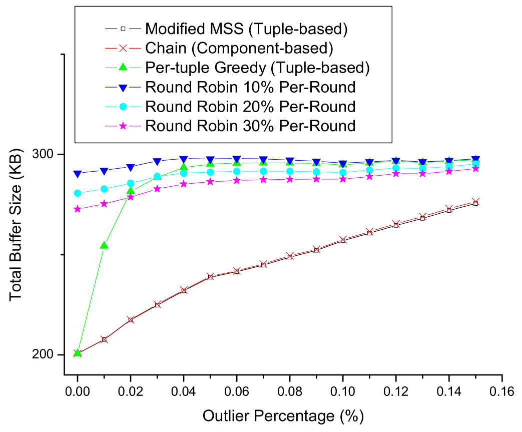


Figure 13: Total Buffer Size by Outlier Percentage

and initially has an empty input. The tuples on operator O_1 have pre-determined processing costs based on a normal distribution $\tau_1 \sim N(1.0, 0.2^2)$ (i.e., a mean of 1 msec and standard deviation of 0.2 msec). The tuples on operator O_2 have processing costs following $\tau_2 \sim N(5.0, 1.0^2)$. Operator O_1 also has a small amount of outlier tuples with a processing costs $\tau_1' \sim N(10.0, 1.0^2)$.

To measure total memory consumption over time (i.e., the total area under curve in the MR charts), we measure the combined size of both buffers once every msec until all the tuples are finished, and take the sum over time.

In Figure 13, we measure total memory under increasing level of outlier tuples on O_1. Very similar conclusions can be drawn as those for latency optimization. The total memory consumption over time increases with increasing outlier. The per-tuple greedy algorithm is again affected the most by outlier tuples, while the tuple-based MSS algorithm variant and the component-based Chain algorithm stay very close to each other (in fact in Figure 13 they almost appear to be identical). Therefore, we see that for memory optimization, the component-based algorithm also closely approximates the performance of the tuple-based algorithm.

7. CONCLUSIONS

While different approaches had been previously proposed for minimizing response time and memory consumption, in this paper we have shown that the chart partitioning algorithm of chain [2] can be extended to handle both memory and latency minimization. The availability of this unified technique has a significant practical importance since, since, e.g., it simplified the design of the query scheduling optimizer in the Stream Mill system [4].

On the theoretical side, this paper tried to answer basic unanswered questions of optimality for component-based algorithms, which is the category most existing approaches belong to. Since the exact costs of tuples are not known during scheduling time, existing scheduling algorithms are based on average-cost heuristics. In this paper, we have first derived an optimal MSS algorithm that makes scheduling decisions based on true tuple costs. Then, we have shown through simulation that the practical algorithms are quasi-optimal, inasmuch as they closely approximate the theoretical performance of the optimal ones.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF-IIS award 0742267, “SGER: Efficient Support for Mining Queries in Data Stream Management Systems”.

9. REFERENCES

[1] B. Babcock et. al. Models and issues in data stream systems. In *PODS*, 2002.

[2] Brian Babcock et. al. Operator scheduling in data stream systems. *VLDB J.*, 13(4):333–353, 2004.

[3] Yijian Bai, Hetal Thakkar, Chang Luo, Haixun Wang, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, pages 337–346, 2006.

[4] Yijian Bai, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo. Optimizing timestamp management in data stream management systems. In *ICDE*, 2007.

[5] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[6] D. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[7] Don Carney et. al. Operator scheduling in a data stream manager. In *VLDB*, 2003.

[8] Lisa Amini et. al. Adaptive Control of Extreme-Scale Stream Processing Systems. In *ICDCS*, 2006.

[9] MA Sharaf et. al. Preemptive rate-based operator scheduling in a data stream management system. In *AICCSA*, 2005.

[10] Mohamed F. Mokbel et. al. Continuous query processing of spatio-temporal data streams in PLACE. *Geoinformatica*, 9(4), 2005.

[11] N. Tatbul et. al. Load shedding in a data stream manager. In *VLDB*, 2003.

[12] R. Motwani et. al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, Asilomar, CA, 2003.

[13] Sirish Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[14] Theodore Johnson et. al. A heartbeat mechanism and its application in gigascope. In *VLDB*, pages 1079–1088, 2005.

[15] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[16] Qingchun Jiang and Sharma Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *BNCOD*, pages 16–30, 2004.

[17] MA Sharaf, PK Chrysanthis, A Labrinidis, and K Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB*, 2006.

[18] Utkarsh Srivastava et. al. Flexible time management in data stream systems. In *PODS*, 2004.

[19] Y.-C. Tu et. al. Load shedding in stream databases: A control-based approach. In *VLDB*, 2006.

[20] Efstratios Viglasa and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.