Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019

Scheduling Strategies for Processing Continuous Queries Over Streams

Qingchun Jiang, Sharma Chakravarthy

Technical Report CSE-2003-30

Scheduling Strategies for Processing Continuous Queries Over Streams

Qingchun Jiang*
Computer Science & Engineering
The University of Texas at Arlington
jiang@cse.uta.edu

Sharma Chakravarthy*
Computer Science & Engineering
The University of Texas at Arlington
sharma@cse.uta.edu

Abstract

Stream data processing poses many challenges. Two characteristics of stream data processing - bursty arrival rates and the need for near real-time performance requirement - challenge the allocation of limited resources in the system. Several scheduling algorithms have been proposed for minimizing the maximal memory requirements in the literature.

In this paper, we propose novel scheduling strategies to minimize the tuple latency as well as the total memory requirement. We first introduce a path capacity strategy with the goal of minimization of the tuple latency, which is a counterpart of the Chain scheduling strategy proposed in the literature. We then analyze these two strategies to identify their limitations and propose additional scheduling strategies that improve upon them. Specifically, we introduce a segment scheduling strategy with the goal of minimization of the memory requirement and its simplified version. In addition, we introduce a threshold strategy which addresses both tuple latency and memory requirement. This hybrid strategy - a combination of two strategies provides a more practical solution for stream data processing. Finally, we present the results of a wide range of experiments conducted to evaluate the efficiency and the effectiveness of the proposed scheduling strategies.

1. Introduction

The continuous query processing system over streaming data has gained increasing attention in the database area because of its distinguishing characteristics and growing demands from a wide range of applications. It differs from a traditional query processing system in the following ways: (*i*) data on which it operates are high volume streaming data, with intermittent bursty mode input, (*ii*) queries processed are long-running, continuous queries, and (*iii*) these applications have to make near real-time decisions . These characteristics, which are unique to data stream management systems (DSMSs) and are different from traditional database management systems (DBMSs), motivate us to develop newer techniques for DSMS in order to process streaming data efficiently and effectively. Representative applications of DSMS include network traffic engineering, network fault management, network intrusion detection, fraud detection, financial monitoring, and sensor data processing, to name a few. These applications have motivated a considerable body of research ranging from algorithms and architectures for stream processing to a full-fledged data stream processing system such as Telegraph [8], STREAM[13], Aurora [5], Niagara[7], Tapestry [16], and Tribeca[14]. In this paper, we focus on the operator scheduling problem for stream data processing, which determines the order in which operators are scheduled at each time slot in a multiple, continuous query processing system.

The scheduling problem in a DSMS is an important and complicated one. First, it has a significant impact on the performance metrics of the system, such as tuple latency and system throughput, among others.

*This work was supported, in part, by NSF grants IIS-0123730 and ITR-0121297

Second, although the resources such as memory size and CPU speed are fixed, scheduling jobs can be highly dynamic. Third, various predefined Quality of Service (QoS) requirements for a query add more constraints to an already complex problem. On the other hand, a desirable scheduling strategy in a DSMS should be able to: (*1*) achieve the maximal performance within the fixed amount of resources; (*2*) be aware of the unexpected overload situations, and take corresponding actions in a timely manner; (*3*) guarantee user- or application-specified QoS requirements for a query, if any; and (*4*) be implemented easily, and run efficiently with a low overhead.

A single scheduling strategy may not be able to satisfy all of the above properties, as there are trade offs among these performance metrics and usages of the limited resources. In this paper, we introduce several scheduling strategies:*1*) the path capacity strategy to achieve the best tuple latency; *2*) the segment scheduling strategy to achieve the minimal memory requirement; *3*) the simplified segment strategy, which requires a lightly more memory but much better tuple latency than the segment scheduling strategy; *4*) the threshold strategy, which is a hybrid of the path capacity strategy and the simplified segment scheduling strategy. These strategies provide a reasonable overall performance although they do not meet all the desirable properties. The predefined QoS requirements of a query have not been incorporated into these strategies, which is part of our future work.

The rest of the paper is organized as follows. Section 2 provides an overview of query processing over streaming data. In section 3, we first propose the path capacity strategy, and then compare the path capacity strategy with the Chain strategy[3]. We further discuss The segment scheduling strategy, its variant (the simplified segment strategy) and the threshold strategy. Section 4 provides discussions related to these strategies. Section 5 further presents our quantitative experimental results. The related work and conclusion are presented in section 6 and 7 respectively.

2. Stream query processing strategies

In a DSMS, a continuous query plan is decomposed into a set of operators (as in a traditional DBMS) such as `project, select, join`, and other aggregate operators. The difference lies in that each operator in a DSMS has to work in a non-blocking mode. As a consequence, some operators have to work in a window-based manner such as a window-based `symmetric hash join`[20]. For a multiple query processing system, the output of one operator is buffered in a queue, which consequently acts as the input source to another operator, if it is not the final operator. Therefore, from a scheduling point of view, the query processing system over streaming data consists of a set of basic operators and queues. The system can be conceptualized as a directed acyclic graph[1](DAG). In such a DAG, a node represents a pipelined operator that processes the tuples; while the directed edge between two nodes represents the queue connecting those two operators, it also determines the input and output relationship between those two operators. For example, the edge from node `A` to node `B` indicates that the outputs of the operator `A` are buffered into the queue `AB` that is the input source of the operator `B`. Each input stream is represented as a special source node, while all the upper level applications are presented as a root node. Therefore, the edges originating from source nodes represent the earliest input queues that buffer the external inputs, while the edges terminating at the root node represent the final output queues. In this DAG, each tuple originates from a source node, and then passes a series of operators until it reaches the root node or is consumed by an intermediate operator. We refer to the path from a source node to the root node excluding the source node and the root node as an `operator path`, and the bottom node of an operator path as a `leaf node`.

[1]This is an assumption currently used widely in the literature; we are aware that real systems have shared buffers but showing optimality for them is a hard problem (next step). Our intuition is that the techniques proposed will work well for shared buffers by multiple operators as well and the deviation from optimal value is likely to be small.

2.1. Assumptions and notations

In this paper, we make the following assumptions:

1. The root node consumes its inputs immediately after they enter queues. Therefore, there is no tuple waiting in the input queues of the root node, and the root node is simply treated as sink in this paper.

2. The selectivity and the service time[2] of an operator are known. Actually, they can be learned easily by collecting statistics information during the execution period of an operator.

To facilitate our analysis, we use the following notations.

- *Operator processing capacity* $C_{O_i}^P$: the number of tuples that can be processed within one time unit at operator O_i. Inversely, the operator service time is the number of time units needed to process one tuple at this operator.

- *Operator selectivity* σ_i: it is the same as in a DBMS except that the selectivity of a join operator is considered as two semi-join selectiveness.

- *Operator memory release capacity* $C_{O_i}^M$: the number of memory units such as bytes, page, can be released within one time unit by operator O_i.

$$C_{O_i}^M = C_{O_i}^P(InputTupleSize - T_i\sigma_i) \tag{2.1}$$

  where T_i is the size of the tuple outputted from operator O_i; the input tuple size of an operator with multiple inputs such as join, is the weighted mean of the input tuple sizes of its inputs.

- *Operator path processing capacity* $C_{P_i}^P$: the number of tuples that can be consumed within one time unit by the operator path P_i. Therefore, the operator path processing capacity depends not only on the processing capacity of an individual operator, but also on the selectivity of these operators and the number of operators in the path. For an operator path P_i with k operators, its processing capacity can be derived from the processing capacities of the operators that are along its path, as follows:

$$C_{P_i}^P = \frac{1}{\frac{1}{C_{O_1}^P} + \frac{\sigma_1}{C_{O_2}^P} + \cdots + \frac{\prod_{j=1}^{k-1}\sigma_j}{C_{O_k}^P}} \tag{2.2}$$

  where $O_l, 1 \leq l \leq k$ is the l_{th} operator along P_i starting from the leaf node. The denominator in (2.2) is the total service time for the path P_i to serve one tuple; and the general item $(\prod_{j=1}^{h}\sigma_j)/C_{O_k}^P$ is the service time at the $(h+1)^{th}$ operator to serve the output part of the tuple from the h^{th} operator along the path, where $1 \leq h \leq k-1$.

- *Path memory release capacity* $C_{P_i}^M$: the number of memory units can be released within one time unit by the path P_i. Again, in this paper, we assume that all the output tuples from a query are consumed immediately by its applications. Therefore, no memory is required to buffer the *final* output results.

$$C_{P_i}^M = C_{P_i}^P InputTupleSize \tag{2.3}$$

[2]The CPU time that an operator needs to process one tuple.

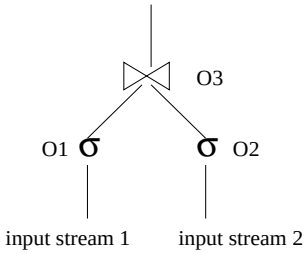

Figure 1. A query execution plan

From equation (2.3), we know that the processing capacity and the memory release capacity of an operator path differs with only a constant factor. Therefore, we assume that the partial order between the processing capacities of two paths is the same as the partial order between their memory release capacities. We believe that this assumption is reasonable under a data stream processing environment. For instance, a query processing system that is used to analyze Internet traffic has the same size input tuple, where the tuples are the header of the Internet IP packets. Although the sizes of all input tuples from some applications may not be exactly the same, their differences are not large enough to change the relative partial orders of their operator paths. Hereafter, we use the path capacity to refer to both the processing capacity and the memory release capacity.

2.2. Preliminary scheduling strategies

A DSMS has multiple input streams and if the input rate of each stream is trackable, which means it is known that how many tuples will be arriving in the future time slots, we can find an optimal scheduling strategy that can achieve the best performance[3] by using the minimized resources. However, in most cases, the input of a data stream is unpredictable, and highly bursty, which makes it harder, or even impossible to find such a feasible, optimal scheduling strategy. In practice, a heuristic-based or near-optimal strategies are usually used. And these strategies have different impact on the performance and the usages of the system resources. The Chain scheduling strategy[3] is a near optimal scheduling strategy in terms of total internal queue size. However, the tuple latency and the memory requirement are equally important for a query processing system. And they are especially important for a DSMS where its applications have to respond to an event in a near real-time manner. In the rest of this section, we use the First-In-First-Out (FIFO) strategy and the Chain strategy to show how a strategy impacts the queue size, the tuple latency, and the throughput of a query processing system.

FIFO Strategy: *Tuples are processed in the order of their arrival. Once a tuple is scheduled, it is processed by the operators along its operator path until it is consumed by an intermediate operator or output to the root node. Then the next oldest tuple is scheduled.*

Chain Strategy: *At any time, consider all tuples that are currently in the system; of these, schedule a single time unit for the tuple that lies on the segment with the steepest slope in its lowest envelope simulation. If there are multiple such tuples, select the tuple which has the earliest arrival time.*

The slope of an operator in the Chain strategy is the ratio of the time it spends to process one tuple to the change of the tuple space. For those who are not familiar with the Chain strategy, you can simply treat the steepest slope as the biggest operator memory release capacity. The tuple in the Chain strategy refers to a batch of tuples, instead of an individual tuple. For further details, refer to [3].

Let us consider the simple query plan illustrated in Figure 1, which is a common query plan that contains

[3]We mean tuple latency, throughput, etc in this paper.

Table 1. Operator properties

Operator Id	1	2	3
Selectivity	0.2	0.2	0.8
Processing capacity	1	1	0.2

Table 2. Performance(F:FIFO, C:Chain)

	Input		Queue Size		Tuple Latency		Throughput	
Time	1	2	F	C	F	C	F	C
1	1	0	1.0	1.0	-	-	0	0
2	0	1	1.2	1.2	-	-	0.0	0
3	1	1	3.0	2.4	2	-	0.16	0
4	0	0	2.2	1.6	-	-	0	0
5	0	0	2.0	0.8	3	-	0.16	0
6	0	0	1.2	0.6	-	5	0	0.16
7	0	0	1.0	0.4	4	5	0.16	0.16
8	0	0	0.2	0.2	-	5	0	0.16
9	0	0	0	0	6	6	0.16	0.16
10	1	0	1.0	1.0	-	-	0	0

both `select` and `join` operators. The processing capacity and selectivity of the operators are listed in Table 1. The input streams are highly bursty, which is the case in most stream query processing systems. Table 2 shows the total internal queue size, tuple latency, and throughput of the query plan under both the FIFO scheduling strategy and the Chain scheduling strategy for the given input patterns.

The results clearly show that the Chain strategy performs much better than the FIFO strategy for the total internal queue size. However, it performs much worse than the FIFO strategy in terms of tuple latency and throughput. Clearly, The FIFO strategy maintains its entire backlog of unprocessed tuples at the beginning of each operator path. It does not take the inherent properties of an operator into consideration such as selectivity, processing rate, which causes the total internal queue size to be bigger under the FIFO strategy than under the Chain strategy. In contrast, as the Chain strategy pushes the tuples from the bottom, it inherits the bursty property of the input streams. If the input streams are highly bursty in nature, the output of the query processing system under the Chain strategy demonstrates highly bursty property too, but the bursty input property in its output is partially determined by the selectivity of the operators and the system load.

It is worth noting that the total internal queue size is much less than the amount of memory needed to maintain the status information at some operators in order to compute results correctly. For example, a `join` operator has to maintain a number of tuples in its window(s) in order to compute the correct answers. Considering an input stream with a bursty period of 1M tuples and its input rate faster than the processing rate, how long do we have to wait to get the first result in a query processing system which has the maximal ability to completely process[4] 10000 input tuples per second? Under the Chain strategy, it is almost 100 seconds! Of course, the total internal queue size is much less than 1M tuples, which is the difference between the input rate and processing rate times the length of the bursty period without considering the changes of tuple size in the system. Based on this observation, we develop the path capacity strategy, which takes the tuple latency and the throughput as their primary priorities, the total internal queue size as its secondary priority.

[4]We mean the computation from reading in the input tuples to output the final results.

3. Proposed Scheduling Strategies

In this section, we first present the path capacity strategy, and then provide a thorough comparison with the Chain strategy. To overcome the large memory requirement of path capacity strategy, we further propose the segment strategy and its variant – the simplified segment strategy. Finally, we discuss the threshold strategy, which is a hybrid of the path capacity strategy and the simplified segment strategy.

3.1. Path capacity strategy

From above results, we observe that the FIFO strategy has two promising properties: reasonable tuple latency and throughput. But it does not consider the characteristics of an operator path such as processing capacity and memory release capacity. Also as it schedules one tuple each time, the scheduling overhead is considerably high. This motivates us to develop the path capacity strategy which improves upon the high memory requirement of the FIFO strategy, and has even better tuple latency and throughput than the FIFO strategy.

Path Capacity Strategy: *At any time instant, consider all the operator paths that have input tuples waiting in their queues in the system, schedule a single time unit for the operator path with a maximal processing capacity to serve until its input queue is empty or there exists an operator path which has non-null input queue and a bigger processing capacity than the currently scheduled one. If there are multiple such paths, select the one which has the oldest tuple in its input queue. The following internal scheduling strategy is used to schedule the operators of the chosen path.*

Internal scheduling strategy: Once an operator path is chosen, a bottom up approach [5] is employed to schedule all the operators along its path. The maximal number of tuples served during one round is exactly its processing capacity.

The path capacity strategy is a static priority scheduling strategy. The priority of an operator path is its processing capacity, which is completely determined by the number of operators along its path, and the selectivity and the processing capacity of each individual operator. Therefore, the priority of an operator path does not change over time until we revise selectivity of operators. The scheduling cost is minimized and can be negligible. Most importantly, the path capacity strategy has the following two optimal properties that are critical for a multiple continuous query processing system.

Theorem 1. *The path capacity strategy is an optimal one in terms of the total tuple latency or the average tuple latency among all the scheduling strategies.*

Proof. First, operator path-based scheduling strategies have a better tuple latency than those which do not use an operator path as a scheduling unit. In [10], we showed that if the operators of two query plans or operator paths are scheduled in an interleaving manner, the overall tuple latency becomes worse. The operators of two operator paths under the path capacity strategy are not scheduled as an interleaving manner and therefore, the path capacity strategy has a better tuple latency than any non-path based scheduling strategy. Second, the path capacity strategy has a minimized tuple latency among all path based scheduling strategies. At any time instant, consider k operator paths $p_1, p_2, \cdots, p_k$, which have $N_i \geq 1, i = 1, 2, 3, \cdots, k$ tuples in their input queue in the system, with their capacities $C_1, C_2, \cdots, C_k$ respectively. Without loss of generality, we assume that $C_1 \geq C_2 \geq \cdots \geq C_k$. The path capacity strategy has a schedule of $p_1, p_2, \cdots, p_k$, that is to serve the N_1 tuple of operator path p_1, following the N_2 tuple of operator path p_2 , and so on. In the simplest case where $N_i = 1, i = 1, 2, 3, \cdots, k$, the total tuple latency $T = k\frac{1}{C_1} + (k-1)\frac{1}{C_2} + \cdots + (k-g+1)\frac{1}{C_g} + \cdots + (k-h+1)\frac{1}{C_h} + \cdots + \frac{1}{C_k}$, where $(k-i)\frac{1}{C_i}, i = 0, 1, \cdots, k-1$ is the total waiting time of all the tuples in the system due to processing the tuple at operator O_i. If we switch any

[5] For lack of space this paper only discusses bottom-up scheduling strategies. Work on pipelines strategies are being investigated.

two tuples (two paths), say g, h, where $g < h$, in the path capacity strategy, then the total tuple latency $T' = k\frac{1}{C_1} + (k-1)\frac{1}{C_2} + \cdots + (k-g+1)\frac{1}{C_h} + \cdots + (k-h+1)\frac{1}{C_g} + \cdots + \frac{1}{C_k}$. The difference of two tuple latency$\Delta = T - T' = (h-g)\left(\frac{1}{C_g} - \frac{1}{C_h}\right) \leq 0$ because of $g < h$ and $C_g \geq C_h$. Similarly, for the general case, by switching any two tuples in two input queues of these k operator paths, we still have $\Delta \leq 0$. Therefore, any other scheduling strategy causes at least the same total tuple latency or mean delay that the path capacity strategy causes. □

Theorem 2. *Any other path-based scheduling strategy requires at least as much memory as that required by the path capacity strategy at any time instant in the system.*

Proof. At any time instant, the path capacity strategy schedules the tuples waiting in the input queue of the operator path which has the biggest capacity among all the paths with non-empty input queue in the system. Within one time unit, the path scheduled by the path capacity strategy consumes the maximal number of tuples because it has the biggest capacity. Any other path based scheduling strategies which do not schedule the tuples waiting in the input queue of the operator path with the biggest capacity at that time instant consume less number of tuples. Therefore, any other path based scheduling strategy requires at least the same amount of memory required by the path capacity strategy. □

Theorem 1 clearly shows that the path capacity strategy performs much better than the FIFO strategy and the Chain strategy for tuple latency. Theorem 2 shows that the path capacity strategy performs better than the FIFO strategy, but not as well as the Chain strategy in terms of the memory requirement although it requires the least memory among all the path-based scheduling strategies.

3.2. Critique of the two scheduling strategies

Both the path capacity strategy and the Chain scheduling strategy have their optimal properties and shortcomings. In this section, we will do a comprehensive comparison and show how these two scheduling strategies impact the various performance metrics of a query processing system. A quantitative experimental study for these two scheduling strategies will be presented in section 5.

Tuple latency & throughput: The path capacity strategy can achieve the optimal tuple latency comparing with any other scheduling strategy. It also has a much smoother output rate than the Chain scheduling strategy. The main reason for the large tuple latency in Chain scheduling strategy is that the leaf nodes usually have a much bigger capacity than the other nodes of a query plan, which causes the Chain scheduling strategy gradually to push all the tuple from the leaf nodes toward the root node, and a large amount of tuples are buffered in the middle of an operator path. This situation becomes even worse during a temporary overload period in which the input rates are temporarily over the processing capacity of the system. All the computational resource is allocated to these operators at the bottom of the query plans during the highly bursty input period[6] and there is almost no throughput from the system. On the other hand, the throughput is surprisingly high immediately after the highly bursty period because there are not too many tuples waiting at leaf nodes, and most of the computation resource is left for the operators in the upper side of the query plans where a large amount of partially processed tuples wait. As a result, the Chain scheduling strategy has not only a bad tuple latency, but also a highly bursty output rate if the input streams are highly bursty ones. Its bursty output rates may cancel out part of its saved memory because the consumed rates of applications cannot keep up with the bursty output rates, which causes portion of the results to backlog in the query processing system for a while.

Memory requirement: Both strategies have an optimal property in terms of the memory requirement. But the optimal property of the path capacity strategy is a relative one among all path-based scheduling

[6]Especially, the bursty period is relatively long if the input streams have self similar property.

algorithms, while the optimal property of the Chain scheduling strategy is a global optimal property. Under the non-overload conditions, the amount of memory required by these two strategies is similar, and there are not too many tuples buffered in the queues. However, during the highly bursty input periods, the Chain scheduling strategy performs better than the path capacity strategy because the path capacity strategy buffers the unprocessed tuples at the beginning of an operator path.

Starvation problem: Both strategies have the starvation problem in which some operators or operator paths may never be served because both of them depend on a set of static priorities. Under heavy load situations, the Chain scheduling strategy spends all its computation resource on these operators with a bigger capacity, most of operators in the upper side of an operator path (closer to the root) have less capacity and are under starvation; while the path capacity strategy spends all its computation resource on these operator paths with a bigger path processing capacity, the operator paths with the less capacity are under starvation. The difference is that the Chain scheduling strategy has very small or even no throughput at all during the heavy load situations, whereas the path capacity strategy still has reasonable throughput during the heavy loaded situations.

Scheduling overhead: Clearly, both strategies have very small scheduling overhead because they are static priority strategies. But the scheduling overhead incurred by the path capacity strategy is less than that incurred by the Chain scheduling strategy because the number of operator paths in a system is less than the number of operators. Although the cost of scheduling one operator or operator path is very small, the cost to process one tuple is even smaller than that. Therefore, the number of tuples served by each schedule has significant impact on the performance of a query processing system. In a practical system, we employ either an exhaustive service discipline or a limited service discipline in order to further decrease the overall scheduling overhead. When an exhaustive service discipline is used, all the tuples waiting in its queue plus those that arrived during the current service period will be served once an operator is scheduled. When an limited-service discipline is employed, only a limited number of tuples waiting in the queue are served, or only limited amount of CPU time is granted to the scheduled operator or operator path. The limited number of tuples or the limited amount of CPU time should be big enough in order to avoid the high ratio of the scheduling cost to the processing cost of one tuple. On the other hand, the very large or disproportionate values of these limited parameters have a negative impact on the tuple latency as well. The system load and the predefined QoS of a query mainly determine how to choose an effective values of these limited parameters.

Context switching overhead: When each operator is implemented as a single thread, the context switching cost incurred by a scheduling algorithm is considerably high. The performance of the system will degrade dramatically as the number of operators increases. However, we have implemented the whole query processing system as a single thread, and hence the cost of switching from one operator path to another is just a function call, which is of low cost in a modern processor architecture. Especially, a batch of tuples rather than one tuple, is scheduled for each scheduling round. Therefore, the cost of making function call is negligible as compared with the cost of processing a batch of tuples.

3.3. Segment Scheduling strategy and its variant

Although the path capacity strategy has the optimal memory requirement among all the path scheduling strategies, it still buffers all the unprocessed tuples at the beginning of an operator path. In a query processing system with a shortage of main memory, a trade off exits between the tuple latency and the total internal queue size. Therefore, we develop the segment scheduling strategy which has a better total internal queue size requirement than the path scheduling strategy, and a smaller tuple latency than the Chain strategy. Furthermore, we introduce the simplified segment strategy which further improves its tuple latency with a little bigger memory requirement than the segment strategy.

The segment scheduling strategy employs an idea that allows us to improve upon the path scheduling strategy. Operator scheduling and path scheduling can be seen as two ends of the spectrum, whereas segment

scheduling covers the points in between. Instead of buffering the unprocessed tuples at the beginning of an operator path, we partition an operator path into a few segments, so that the partially processed tuples can be buffered at the beginning of a segment. This allows the system to take advantage of the lower selectivity and fast service rate of the bottom side operators of a query execution plan. Before we discuss the strategy, we define the processing capacity and the memory release capacity for a segment, and then find an efficient algorithm to construct segments from an operate path.

The processing capacity $C_{S_i}^P$ of the segment S_i that consists of k operators $(O_1, O_2, \cdots, O_k)$ in the bottom to top (i.e., leaf to root) is defined as:

$$C_{S_i}^P = \frac{1}{\frac{1}{C_{O_1}^P} + \frac{\sigma_1}{C_{O_2}^P} + \cdots + \frac{\prod_{j=1}^{k} \sigma_j}{C_{O_k}^P}} \tag{3.1}$$

Similarly, its memory release capacity $C_{S_i}^M$ is defined as:

$$C_{S_i}^M = C_{S_i}^P(InputTupleSize - S_o * \prod_{i=1}^{k} \sigma_i) \tag{3.2}$$

where S_o is the size of the output tuple from segment S_i. For the last segment, the size of the output tuple is zero because of the assumption that the output can be consumed by its application immediately whenever it is available.

The segment construction algorithm: The construction algorithm consists of two main steps. First, it partitions an operator path into a few segments. Second, it prunes the current system segment list, which is initially empty, and then adds the new segments into the list. For each operator path in the system, we repeat the following procedure: Consider an operator path with m operators $O_1, O_2, \cdots, O_m$ from bottom to top. Starting from O_1, a segment of the operator path is defined as a set of consecutive operators $\{O_k, O_{k+1}, \cdots, O_{k+i}\}$ where $k \geq 1$, such that $\forall j, k \leq j < k+i, C_{O_j}^M \leq C_{O_{j+1}}^M$. Once such a segment is constructed, we start the construction procedure again from O_{k+i+1} until all the operators along the operator path have been processed. In the pruning procedure, a new segment is added to the segment link list only if: (*i*) any of its subset has already been in the list, we have to remove all its subsets from the segment list, and then add the new segment into the list; (*ii*) none of its supersets has been in the list, we add it to the list; otherwise, the new segment is discarded.

The order in which we partition an operator path does not matter because the final segment list is the same given the query plans, and the order of a segment in the segment list does not affect its priority. We only need to run the above algorithm once in order to construct the segment list. Later on, when a new query plan is registered into the system, we need to run the algorithm for the operator paths of the newly registered query plan. When a query plan is unregistered from the system, we have to delete all the segments belongs to the query plan. In a multiple query processing system, one segment may be shared by two or more query plans, therefore, we have to add a `count` field to each operator of a segment to indicate how many query plans are using it. Once a segment is deleted from the system, we decrease the value in the `count` field by one for each operator that belongs to the segment. When the `count` value of an operator reaches zero, it is deleted from the segment.

As all the continuous query plans in a stream processing system are long-running queries, the number of queries that are registered into a system or unregistered from a system is not likely to be too large (typically no more than a few per hour). Therefore, the cost of the algorithm has very little impact on system performance.

Segment Scheduling Strategy: *At any time instant, consider all the operator segments that have input tuples waiting in its input queue; schedule a single time unit for the operator segment which has a maximal memory*

release capacity to serve until its input queue(s) is(are) empty or there exists another operator segment which has non-null input queue and a bigger memory release capacity than the current scheduled one. If there are multiple such segments, select the segment which has the oldest tuple in its input queue. The internal scheduling strategy (described earlier) is used to schedule the operators along the chosen segment.

The segment scheduling strategy is also a static priority driven strategy. The segment strategy uses the same operator segment concept used by the Chain strategy. The segment with the biggest memory release capacity has the steepest slope in its lower envelope too. However, it is different from the Chain scheduling strategy in that: (*a*) it uses the memory release capacity of an operator segment as the priority of the segment, while the Chain strategy uses the steepest slope in its lower envelope as the priority of each operator along the segment, (*b*) it clusters a set of operators as a scheduling unit, and hence there are no partially processed tuples buffered in the middle of an operator segment by the end of each time unit as in the Chain strategy, and (*c*) it has a smaller scheduling overhead than the Chain strategy. The Chain strategy is an operator based strategy where all the operators along a segment have a same priority, while the segment strategy is a segment based strategy. In a general query processing system, the number of segments is less than the number of operators, therefore, it decreases the scheduling overhead.

Theorem 3. *In terms of the maximal memory requirement, the segment scheduling strategy requires at most the amount of memory required by the Chain scheduling strategy at any time instant.*

Proof. At any time instant, let S_i denote the segment that has the biggest memory release capacity and has a non-null input queue, and let $\{O_1, O_2, \cdots, O_k\}$ be its k operators ordered from bottom to top. Based on the definition of the Chain strategy, the operators along the segment S_i have the same biggest priority in the system at that time instant. And it schedules one time unit for the tuple waiting at operator O_1, then one time unit for the tuple waiting at operator O_2, and so on. But the maximal memory requirement occurs when it schedules one time unit for the operator O_1 because it has the smallest memory release capacity among all the operators of the segment. At that time instant, memory released by the Chain strategy from (2.1) is $\mathcal{M}_{Chain} = C^P_{O_i}(InputTupleSize - T_i\sigma_i))$, while the memory released by segment strategy (3.2) is $\mathcal{M}_{Segment} = C^P_{S_i}(InputTupleSize - \quad S_o * \prod_{i=1}^{k} \sigma_i)$. However, $\mathcal{M}_{Segment} - \mathcal{M}_{Chain} >= 0$. Therefore, the chain strategy requires at least as much maximal memory as the segment strategy. □

The segment scheduling strategy minimizes the memory requirement of a multiple continuous query processing system, but it also causes longer tuple latency than the path capacity strategy because it separates one operator path into multiple segments. The paper [10] shows that the overall tuple latency of the tuples from an operator path increases significantly if any other operators are scheduled to interleave with the operators along the operator path. In order to decrease the interleaving of the operators of two segments, we propose the following simplified segment strategy.

Simplified Segment Scheduling Strategy: This one differs from the segment scheduling strategy in that it employs a different segment construction algorithm. In a practical multiple query processing system, we observe that: (*a*) the number of segments constructed by the segment construction algorithm is not significantly less than the number of operators presented in the query processing system and (*b*) the leaf nodes are the operators that have faster processing capacities and less selectivity in the system; all the other operators in a query plan have a much slower processing rate than the leaf nodes. Based on these facts, we partition an operator path into at most two segments, rather than a few segments. The first segment includes the leaf node and its consecutive operators such that $\forall i, C^M_{O_{i+1}}/C^M_{O_i} \geq \gamma$, where γ is a similarity factor. In the previous segment construction algorithm, it implicitly states $\gamma = 1$. In the simplified segment strategy, the value of γ used is less than 1 in order to decrease the number of segments. In our system, we have used $\gamma = 3/4$ in our experiments. The remaining operators along that operator path, if any, forms the second segment. The simplified segment strategy has the following advantages: (*i*) Its memory requirement is only slightly larger than

the segment strategy because the first segment of an operator path releases the most of the amount of memory that can be released by the operator path, (*ii*) The tuple latency significantly decreases because the number of times a tuple is buffered along an operator path is at most two, (*iii*) The scheduling overhead significantly decreases as well due to the decrease in the number of segments, and (*iv*) It is less sensitive to the selectivity and service time of an operator due to the similarity factor, which makes it more applicable.

3.4 Threshold strategy

The threshold strategy is a hybrid of the path capacity strategy and the simplified segment strategy. The principle behind it is that the path capacity strategy is used to minimize the tuple latency when the memory is not a bottleneck; otherwise, the simplified segment scheduling strategy is used to minimize the total memory requirement. Therefore, this one combines the properties of these two strategies, which makes it more appropriate for a DSMS.

Threshold Strategy: *Given a query processing system with a maximal available queue memory*[7] $\mathcal{M}$*, the maximal threshold* $\mathcal{T}_{max}$ *and the minimal threshold* $\mathcal{T}_{min}$*, where* $\mathcal{T}_{min} < \mathcal{T}_{max} < \mathcal{M}$*, at any time instant, when the current total queue memory consumed* $\mathcal{M}_c \geq \mathcal{T}_{max}$*, the system enters its memory saving mode in which the simplified segment strategy is employed. The system transits from the saving mode to the normal mode in which the path scheduling strategy is employed when* $\mathcal{M}_c \leq \mathcal{T}_{min}$*.*

The values of the maximal threshold $\mathcal{T}_{max}$ and the minimal threshold $\mathcal{T}_{min}$ mainly depend on the load of the system and the length of the bursty periods; and they can be obtained heuristically or experimentally. Given that the mean total queue memory consumed by a query processing system is $\bar{\mathcal{M}}$ memory units, we define the values of these threshold parameters in our system as

$$\begin{cases} \mathcal{T}_{max} & = min\left(\frac{1+\alpha}{2}\mathcal{M}, \beta\mathcal{M}\right);\ \alpha = \frac{\bar{\mathcal{M}}}{\mathcal{M}} \\ \mathcal{T}_{min} & = min\left(\bar{\mathcal{M}}, \beta\mathcal{T}_{max}\right);\quad 0.5 < \beta < 1 \end{cases} \tag{3.3}$$

In (3.3), β is a safety factor that guarantees a minimal memory buffer zone between the normal mode and the saving mode, which prevents a system from frequently oscillating between the memory saving mode and the normal mode. A smaller value of β causes a longer tuple latency. Therefore, its value need to be in the range of 0.5 to 1.0. α is used to adjust the threshold values as the system load changes. The mean total queue size increases as the system load increases, which causes α increases. When α approaches 1, the $\frac{1+\alpha}{2}\mathcal{M}$ approaches the maximal available queue memory $\mathcal{M}$. That is why we need $\beta\mathcal{M}$ to guarantee that there is a minimal buffer between the $\mathcal{T}_{max}$ and $\mathcal{M}$. We use $\beta = 0.9$ in our system. Our experiments show these parameters work well in general.

In a practical system, we have to monitor the current queue memory in order to determine when to switch the mode. But this cost is small because: *1*) each queue in our system maintains its current queue size and the tuple size, and the current queue memory is just the sum of the queue memory occupied by each queue together; *2*) instead of computing the current queue memory by the end of each time, we compute it by the end of each time interval. The length of time interval is dynamically determined based on current queue memory. If the current total queue memory size is far away from the total available memory size, a long time interval is used. otherwise, a shorter interval is used. Therefore, the overall overhead incurred by the threshold scheduling strategy has very little impact on the system performance.

As the mean load of the system increases, the time where the system stays under the saving mode increases and the overall tuple latency becomes worse. When there is no more memory available for the internal queues under the saving mode, the load shedding or sampling techniques have to be used to relieve the system from suffering from a shortage of memory.

[7]The queue memory here refers to the memory available for input queues, not including the memory consumed by maintaining the status information of an operator.

4. Discussion

In this section, we briefly discuss how different execution plans of a continuous query impact the performance of a system under the proposed scheduling strategies, and then discuss how to avoid the starvation problem while using these scheduling strategies.

4.1. Continuous query execution plan

A logical continuous query can be implemented as different physical query execution plans. Based on our analysis of the scheduling strategies, we find that the following points are helpful for choosing the right physical execution plan in order to improve the system performance.

Push the select and the project operators down: Both the path capacity strategy and the segment scheduling strategy can benefit from the lower selectivity of a leaf operator, and from the earlier project operator. From (2.2) and (3.1), we know that: (*a*) the lower selectivity of a leaf operator dramatically increases the processing capacity and the memory release capacity of an operator path or segment; (*b*) the down-side project operators can decrease the output size of the tuples earlier, the released memory can be quickly reused. Therefore, the tuple latency and the memory requirements of an operator path or segment can be optimized through pushing the selection and the project as far down as possible in a physical query plan.

Make the operator path short: The processing capacity of an operator path or segment depends not only on the selectivity of the individual operator, but also on the number of the operators. It may not increase the processing capacity of an operator path or segment to make the operator path short because the service time of an individual operator may increase. For instance, by incorporating a project operator into a select operator, we can shorten the operator path, but it does not increase the processing rate of the path. However, the number of times an output tuple is buffered decreases, which can decrease the tuple latency and the scheduling overhead of a scheduling strategy as well. In addition, less number of operators in a path makes it much easier to control or estimate the tuple latency.

4.2. Starvation free scheduling strategy

All the scheduling strategies discussed so far are static priority driven strategies. Under an overloaded system some paths/segments may have to wait for a long period to be scheduled or even not scheduled at all theoretically. To overcome the starvation problem, we discuses two simple solutions here, and we are still investigating other solutions for this problem.

The solutions we present here are applicable to all strategies discussed in this paper. Furthermore, an operator path and an operator segment are used interchangeably in this subsection.

Periodically schedule the path with the oldest tuple in its input queue: A straight-forward solution to the starvation problem is to periodically schedule the path with the oldest tuple in its input queues in our proposed strategies. The length of the period to schedule the oldest operator path depends on the load of a system and the QoS requirement of its applications.

Dynamic Priority: Another solution is to change the priority of the strategies periodically. The total waiting queue size and the age of the oldest tuple of an operator path characterize its activities, such as the mean input rate, schedule frequency, and so on. In order to avoid the starvation problem, we consider the total waiting queue size and age of the oldest tuple as two additional factors of the priority of an operator path. And we define the priority factor f_i of the operator path i as:

$$f_i = \tau_i Q_i$$

where τ_i is the normalized waiting time of the oldest tuple in the input queue of the path i; Q_i is the normalized total current queue size of that operator path. Therefore, the new capacity of an operator path $\hat{C}_i^P = C_i^P f_i$. As you can see, as the age increases, the queue size increases as well, which makes the priority factor f_i

increase exponentially. During a highly bursty input period, its priority factor increases too. Eventually, the oldest path will be scheduled.

Although the above solutions can solve the starvation or long waiting problem due to the temporary overload (the overall input rate of the system $>$ its overall processing rate), they cannot decrease the system load. Once the load of a system is beyond its maximal capacity it can handle, load shedding [15] or sampling techniques have to be used to relieve the load, which is beyond the capability of a scheduling strategy. However, a scheduling strategy can be aware of when these techniques have to be used. We are currently investigating this problem.

5. Experimental Validation

We have implemented the proposed scheduling strategies as part of the prototype of a QoS aware DSMS. In this section, we discuss the results of various experiments that we have conducted in order to compare the performance of these scheduling strategies.

5.1. Setup

We begin with a brief description of our experimental setup. We are implementing a general purpose DSMS which has its main goal to guarantee predefined QoS requirements of each individual query. Currently, a set of basic operators such as `project, select`, and `join` have been implemented in the system. And the system is capable of processing various Select-Project-Join queries over data streams. The scheduling strategies we have implemented are: the path capacity strategy, the simplified segment strategy, and the Chain strategy.

Due to the fact that the domain of stream-based applications is still emerging and that there are no real multiple data streams available (most available data is just a single data stream which is not adequate to do join), we have synthetically generated data streams and multiple continuous queries for our experiments. We briefly discuss the way in which we generate data streams and the continuous queries used in our experiments.

Input data streams: The input data streams we generate are highly bursty streams that have so-called self-similar property, which we believe resembles the situation in real-life applications. Each input stream is a super imposition of 64 or 128 flows. Each flow alternates ON/OFF periods, and it only sends tuple during its ON periods. The tuple inter-arrival time follows an exponential distribution during its ON periods. The lengths of both the ON and the OFF periods are generated from a Pareto distribution which has a probability massive function $P(x) = ab^a x^{-(a+1)}, x \geq b$. We use $a = 1.4$ for the ON period and $a = 1.2$ for the OFF period. For more detailed information about self-similar traffic, please refer to [12, 4]. In our experiment, we use 5 such self-similar input data streams with different mean input rates.

Experimental query plans: All of our queries are continuous queries that consist of `select, project`, and `symmetric hash join` operators. To be more close to a real application, we run 16 actual continuous queries with 116 operators over 5 different data streams in our system. The whole query set consists of various queries ranging from the simplest select queries to the complicated queries that have 4 join operators over 5 data streams. The selectivity of each operator is widely distributed ranging from 0 to 1, except that few join operators have a selectivity greater than 1. Both the selectivity and the processing capacity of each operator can be determined by collecting the statistics information periodically during run time.

The prototype is implemented in C++, and all the experiments were run on a dedicated dual processor Alpha machine with 2GB of RAM. One of the processors was used to collect experiment results while the other processor was used for query processing.

5.2. Performance evaluation

Due to the fact that the continuous queries are also long running queries and that the scheduling strategies demonstrate different performance during the different system load periods, we run each independent

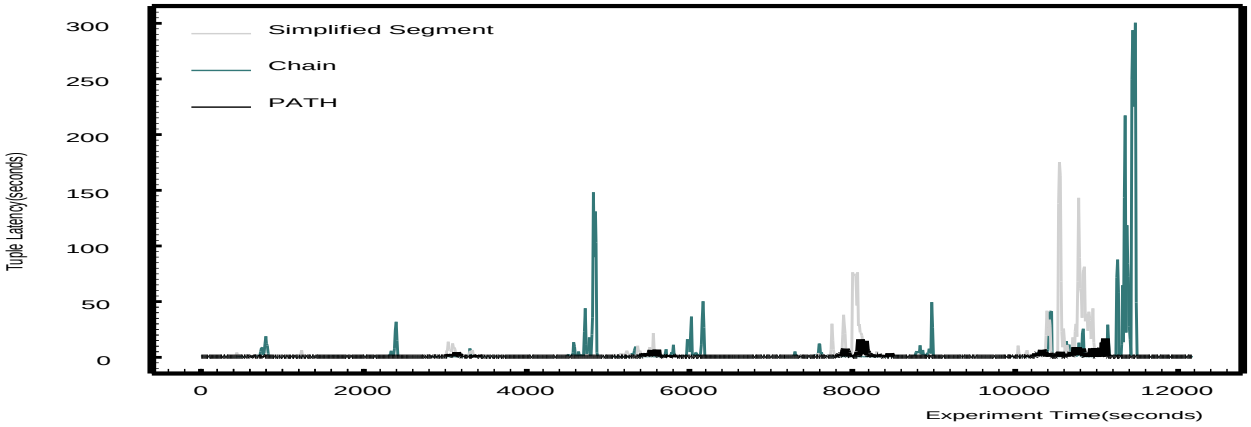

Figure 2. Tuple latency vs. time

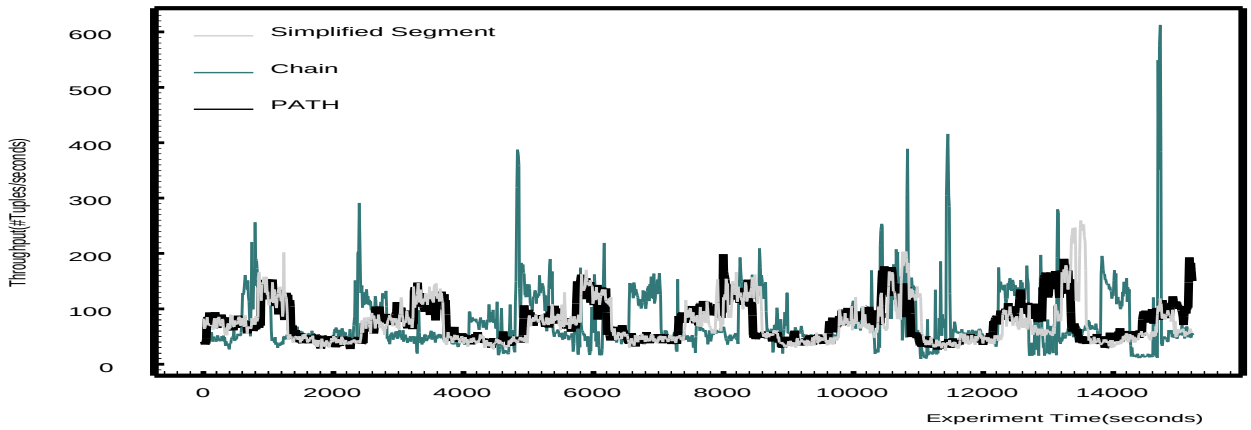

Figure 3. Throughput vs. time

experiment for more than 24 hours including the statistics collection period, and each experiment consists of multiple phases. In each phase, we intentionally increased the average input rates of data streams in order to access the performance characteristics of a scheduling strategy under different system loads. We only present portion of our experiment data (from a few phases), rather than a full range of results due to limited space.

Tuple latency:

The tuple latency of an output tuple is computed by taking the difference of its arrival timestamp and its departure timestamp when it left the query processing system. The tuple latency showed in Figure 2 is the average tuple latency of all output tuples within every 1 second. From the results, we observe that the overall tuple latency is much better under the path capacity strategy than under the simplified segment strategy and the Chain strategy. The Chain strategy performs worst among them. Furthermore, the overall tuple latency increases as the system load increases. but the difference among them becomes much sharper as the system load increases. During the first 4000 seconds, all of them have a reasonable tuple latency except that the Chain has a few spikes. When the system load increases, the tuple latency increases sharply during the highly bursty input periods for both the simplified segment strategy and the Chain strategy. As we explained earlier, the bad tuple latency under the Chain strategy and the simplified segment strategy contributes to their buffered tuples in the middle of an operator path. The simplified segment strategy performs better than the Chain strategy because it buffers less time of a tuple along an operator path than the Chain strategy.

Throughput:

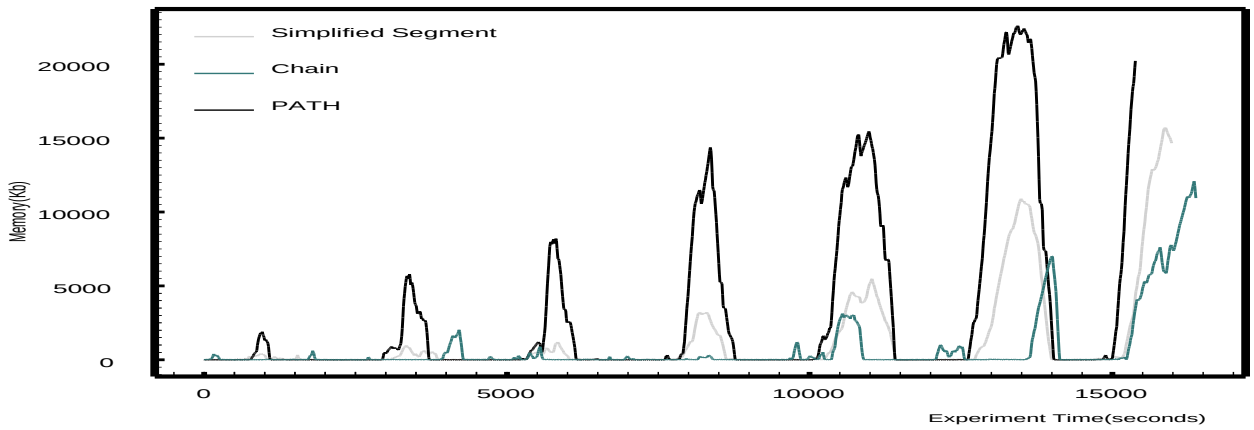

Figure 4. Total memory requirement vs. time

The total throughput of a query processing system under any scheduling strategy should be same because it should output the same number of output tuples no matter whatever scheduling strategy it employs. However, the output patterns are dramatically different under different scheduling strategies. Figure 3 shows the output patterns under three different strategies. The path capacity strategy and the simplified segment strategy have a much smoother output rate than the Chain strategy, and the path capacity strategy performs best among them in terms of the bursty output. The output rate under all three strategies increases as the input rates increases when the system load is moderate, which is the first 4000 seconds, and their output patterns do not differ with each other too much. After the system enters the high load periods, especially the last 2000 seconds in our experiment, the path capacity strategy and the simplified segment strategy have a higher throughput during the high bursty input periods which are the periods of the 10400 second to 11000 second and from the 12400 second to 12800 second, but the Chain strategy has a lower throughput there. In contrast, it has a surprisingly high throughput after the highly bursty input periods. Its highest output rate is almost 5 times its average output rate. The situation becomes worse when the system load or the length of the highly bursty input periods increases. The worst situation is there is no throughput at all under the Chain strategy. This highly bursty output rate is not desirable because of the amount of partial results that have to be buffered in the system temporarily, which consumes unnecessary memory.

Memory requirement:

We study the total memory requirement of a query processing system under the different scheduling strategies given an input pattern. From the results presented in Figure 4, we observe that the Chain strategy performs better than the other two. The Chain strategy and the simplified segment strategy can absorb the extra memory requirement during the bursty inputs periods when system load is not high. Although the path capacity strategy has the capability to absorb the temporary high bursty input, its ability is much less than the other two strategies. The reason, as we mentioned earlier, is that the leaf nodes have a much bigger memory release capacity than the other nodes of a query plan, and that the memory release capacity of an operator path is much less than that of a leaf node. As the system load or the length of an bursty period increases, the path capacity strategy requires much more memory to temporarily buffer the unprocessed tuples than the other two strategies. The simplified segment strategy requires a little bit more memory than the Chain strategy during the highly bust periods because it only takes the benefit of the bigger memory release capacities of the leaf nodes that are major part of the operators with a bigger memory release capacity in a query processing system, but not all of them behave like the Chain strategy.

We did not present the results of the threshold scheduling strategy here because its results are the hybrid of that of the path capacity strategy and that of the simplified segment strategy too. It demonstrates the optimal

tuple latency when system has enough memory to buffer the temporary tuple during a highly bursty input period. Otherwise, it demonstrates the near optimal memory requirement and smooth output rate during its memory saving mode.

6. Related work

Various scheduling problems have been extensively studied in the literature. The work directly related to our work is the scheduling strategies for a stream query processing system. The Chain scheduling strategy[3] has been proposed with a goal of minimization of the total internal queue size. As a complement to that, the path scheduling strategy proposed in this paper minimizes the tuple latency. The Aurora project employs a two-level scheduling approach[6]: the first level handles the scheduling of superboxes which is a set of operators, and the second level decides how to schedule a box within a superbox. They also initially discussed how to incorporate the QoS specification of an application into their scheduling strategy. The rate-based optimization framework proposed by Viglas and Naughton [19] has the goal of maximization of the throughput of a query. However, they do not take the tuple latency and memory requirement into consideration. Earlier work related to improving the response time of a query includes the dynamic query operator scheduling of Amsaleg [1] and the Xjoin operator of Urban and Franklin [17].

Some closely related work to our paper are the adaptive query processing[2, 8, 18, 9], which address the efficient query plan execution in a dynamic environment by revising the query plan. The novel architecture proposed in Eddy[2] can efficiently handle the bursty input data, in which the scheduling work is done through a router that continuously monitors the system status. However, the large amount of the state information associated to a tuple limits its scalability.

Finally, the scheduling work is part of our QoS control and management framework for a DSMS and extends our earlier work of modeling continuous query plan [10, 11], which address the effective estimation of the tuple latency and the internal queue size under a dynamic query processing system. Their results can be used to guide a scheduling strategy to incorporate the pre-defined QoS specification of an application, and to predict the overload situations.

7. Conclusions and future work

In this paper, we have proposed a number of scheduling strategies for a DSMS and investigated them both theoretically and experimentally. We showed how a scheduling strategy impacts/affects the performance metrics such as tuple latency, throughput, and memory requirement of a continuous query processing system. We proved that the path capacity strategy can achieve the overall minimal tuple latency. The simplified segment capacity strategy partially inherits the minimal memory requirement of the segment strategy, and demonstrates a much better tuple latency and throughput than the Chain strategy. Furthermore, the threshold strategy inherits the properties of both the path capacity strategy and the simplified segment capacity strategy, which makes it more applicable to a DSMS.

As part of ongoing work, we are considering to extend both the path capacity strategy and the simplified segment capacity strategy to incorporate the predefined QoS specifications of the applications. Another problem we are investigating is the effective system capacity estimation of a continuous query processing system over data streams, and how to use the estimation to guide a scheduling strategy to be aware of the overload situations, and to further take effective actions such as load shedding and sampling to bring the system back to a normal status.

References

[1] L. Amsaleg, M. Franklin, and A. Tomasic. Dynamic query operator scheduling for wide-area remote access. *Journal of Distributed and Parallel Databases*, 3(6), July 1998.

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

[3] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in stream systems. *In Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.

[4] J. Beran. *Statistics for Long-Memory Processes*. New York: Chapman and Hall, 1994.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. TatbuL, and S. Zdonik. Monitoring streams - a new class of data management applications. *In Proc. Of the 2002 Intl. Conf. On Very Large Data Bases*, 2002.

[6] D. Carney, U. etintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. *In Proc. Of the 2003 Intl. Conf. On Very Large Data Bases*, 2003.

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

[8] J. Hellerstein, M. Franklin, and et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

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

[10] Q. Jiang and S. Chakravarthy. Analysis and validation of continuous queries over data streams. *http://itlab.uta.edu/sharma/Projects/MavHome/fi les/QA-SPJQueries2.pdf*, 2003.

[11] Q. Jiang and S. Chakravarthy. Queueing analysis of relational operators for continuous data streams. *12th International Conference on Information and Knowledge Management (CIKM'03)*, Nov 2003.

[12] W. Leladn, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffi c. *IEEE/ACM transaction on Networking*, Feb 1994.

[13] R. Motwani, J. Widom, and et al. Query processing, approximation, and resource management in a data stream management system. *In Proc. First Biennial Conf. on Innovative Data Systems Research*, Jan 2003.

[14] M. Sullivan. Tribeca: A stream database manager for network traffi c analysis. *In Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, page 594, Sept 1996.

[15] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, September 2003.

[16] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *In Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, June 1992.

[17] T. Urhan and M. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[18] T. Urhan and M. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. *In Proc. Of the 2001 Intl. Conf. On Very Large Data Bases*, Sept 2001.

[19] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. *In Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[20] A. N. Wilschut and P. M. G. Apers. Pipelining in query execution. *In Proc of the Intl. Conf. on Databases, Parallel Architectures and their Applications*, March 1990.