An Adaptive Multi-Objective Scheduling Selection Framework for Continuous Query Processing[*]

Timothy M. Sutherland, Bradford Pielech, Yali Zhu, Luping Ding and Elke A. Rundensteiner
Computer Science Department, Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609
{tims, winners, yaliz, lisading, rundenst}@cs.wpi.edu

Abstract

Adaptive operator scheduling algorithms for continuous query processing are usually designed to serve a single performance objective, such as minimizing memory usage or maximizing query throughput. We observe that different performance objectives may sometimes conflict with each other. Also due to the dynamic nature of streaming environments, the performance objective may need to change dynamically. Furthermore, the performance specification defined by users may itself be multi-dimensional. Therefore, utilizing a single scheduling algorithm optimized for a single objective is no longer sufficient. In this paper, we propose a novel adaptive scheduling algorithm selection framework named AMoS. It is able to leverage the strengths of existing scheduling algorithms to meet multiple performance objectives. AMoS employs a lightweight learning mechanism to assess the effectiveness of each algorithm. The learned knowledge can be used to select the algorithm that probabilistically has the best chance of improving the performance. In addition, AMoS has the flexibility to add and adapt to new scheduling algorithms, query plans and data sets during execution. Our experimental results show that AMoS significantly outperforms the existing scheduling algorithms with regard to satisfying both uni-objective and multi-objective performance requirements.

1 Introduction

An increasing number of modern applications need to process streaming data, including sensor network monitoring [14], online transaction processing [22] and network intrusion detection [21]. Correspondingly, stream processing systems, such as Aurora [1], CAPE [18], NiagaraCQ [7], STREAM [17], TelegraphCQ [6], Raindrop [20] to just name a few, have been designed and developed to process long-running continuous queries on potentially infinite data streams and produce query results in a real-time fashion.

A continuous query system may execute a large number of concurrent continuous queries. This raises the problem of allocating processing resources to query operators, i.e., how to schedule the execution of the operators. To have fine-grained control over the query processing, rather than randomly selecting operators to execute or leaving the scheduling task to the underlying operating system, continuous query systems employ scheduling algorithms to determine the order in which the operators will be executed.

1.1 Issues with Operator Scheduling in Continuous Query Processing

Due to the dynamic nature of the streaming environments, it is imperative to employ adaptivity in operator scheduling. The stream characteristics may change significantly in terms of both arrival rates and value distributions. Also queries can be dynamically inserted to or deleted from the system [7]. As a consequence, the resources available for executing each individual operator may differ greatly over time. The traditional scheduling algorithms borrowed directly from the operating system realm [8, 24], such as FIFO and Round Robin, do not adjust their behavior according to these runtime variations. Hence they may not perform well in a volatile streaming environment.

Several newly proposed operator scheduling algorithms for continuous query processing, such as Chain [3] and Train [5], can adapt their runtime behavior due to changes in the streaming environments. These adaptive scheduling algorithms are *uni-objective*, meaning each algorithm aims to satisfy a single performance objective. For example, Chain aims to minimize the intermediate queues. The Train algorithm has three variations and each aims at improving just one of the following performance requirements: average tuple delay, query throughput or total queue size.

[*]The research was partly supported by the RDC grant 2003-04 on "Online Stream Monitoring Systems: Untethered Healthcare, Intrusion Detection, and Beyond." Thanks to TotalETL for providing equipments.

The uni-objective scheduling algorithms can be insufficient for continuous query processing due to the following reasons. First, different performance objectives may be correlated with each other, and this correlation can be either positive or negative. By targeting just one performance metric, the gains in this metric may have the side effect of degradating another metric. This may inversely have a negative effect on the targeted metric. For example, an algorithm aiming to minimize the total queue size may tend to process the part of the query that generates the most tuples. This may possibly cause an increase in average tuple delay, which may impair the effect of minimizing queue sizes. Therefore, even though only a single performance objective is explicitly specified, multiple objectives may need to be considered by a scheduling algorithm.

Secondly, the performance objective of query processing may also vary due to the change of application requirements and system resource availability. For example, the workload in a query system may fluctuate greatly over time. Under a light workload and with abundant memory resources, maximizing the throughput may be the most desirable optimization objective. However, under heavy query workload, minimizing memory overhead may become the most critical optimization task in order to prevent memory overflow. As a result, a scheduling algorithm needs to have the ability to adapt its optimization objective dynamically. This, however, is not supported by the existing scheduling algorithms.

Lastly, in many cases the desired performance of a query may already contain multiple objectives. For example, to run several time-critical queries in a memory-limited machine, two optimization goals should be targeted – minimum result latency and minimum memory overhead. These two objectives are conflicting to some degree. A good balance must be achieved between them by setting priorities to these objectives. Another example of needing *prioritized multiple objectives* is the processing of network intrusion detection queries, which may focus on getting results in real time with relatively relaxed requirements on memory usage, assuming the processing is done at a high-end machine.

In summary, scheduling algorithms need to support the following features: (1) adapting to changing stream characteristics, (2) supporting multiple *prioritized* optimization goals, and (3) being able to dynamically change the optimization objective. The current uni-objective scheduling algorithms lack features (2) and (3). This now is the focus of our work presented in this paper.

1.2 Our Approach: Adaptive Scheduling Selection Framework

We propose AMoS, an **A**daptive **M**ulti-**o**bjective **S**cheduling selection framework. AMoS is able to leverage the strengths of each scheduling algorithm and avoid its weaknesses, especially in the presence of multiple prioritized performance objectives. AMoS can be viewed as a "meta-scheduler". Given several scheduling algorithms, AMoS employs a lightweight learning mechanism to empirically learn the behavior of the scheduling algorithms over time. It then uses the learned knowledge to continuously select the algorithm that has statistically performed the best. Our work contributes to continuous query processing in the following ways:

- To the best of our knowledge, we are the first to consider a general scheduling framework for continuous query processing with several prioritized performance objectives. We propose a measure to quantify such comprehensive objectives so to effectively assess how well each scheduling algorithm meets these objectives.
- We experimentally study the performance of a variety of state-of-the-art scheduling algorithms in an actual continuous query system named CAPE [18] to examine their strengths and weaknesses under varying performance objectives and query workloads.
- The framework is designed to learn the behavior of the scheduling algorithms with very little processing overhead. No apriori information of the scheduling algorithms, query plans or data sets is needed.
- We build AMoS in the CAPE system to make decisions on operator scheduling. We also equip AMoS with a library of scheduling algorithms for it to utilize.
- We conduct an experimental study that supports our claim that AMoS can in fact leverage the strengths of several existing scheduling algorithms to improve the overall performance of the query execution given a set of (possibly changing) performance objectives.

1.3 Roadmap

This paper is organized as follows. In Section 2, we describe a motivating example and present results of experimental evaluation of existing scheduling algorithms. Section 3 describes the architecture and the key components of AMoS. In Section 4, we present an experimental study that confirms the effectiveness of AMoS. We review related work in Section 5 and conclude our paper in Section 6.

2 Analysis of Existing Scheduling Algorithms

To further illustrate the scheduling problem that motivates our work on AMoS, we now analyze commonly adopted scheduling algorithms in continuous query systems to show that there is indeed no "one size fits all" algorithm. Note that this knowledge about the scheduling algorithms, however, is not required by AMoS because AMoS is able to empirically learn this knowledge during the execution.

2.1 Scheduling Algorithms

Round Robin (RR). Round Robin is perhaps the most basic scheduling algorithm. It is used as a default scheduler by many continuous query systems such as [7]. It places all runnable operators in a circular queue and allocates a fixed time slice to each. Round Robin's most desirable quality is the avoidance of starvation. However, Round Robin does not adapt to changing stream conditions.

First In First Out (FIFO). FIFO operates on the oldest tuples first to push them through the query plan. FIFO reduces the result latency because the older tuples are given a higher priority over newer ones. But it has the same drawbacks as Round Robin - no adaptability and no consideration of operator properties.

Most Tuples in Queue (MTIQ). MTIQ assigns a priority to each operator equivalent to the number of the tuples in its input queue(s). It is a simplified batch scheduler, similar to Train [5]. Operators typically have a start-up cost associated with their execution. The batch scheduler can amortize this cost over a larger group of tuples. The most obvious advantage of MTIQ is the minimization on total queue size.

Chain. Chain [3] is a recently proposed variation of greedy scheduling. Each operator is assigned a priority that is based on its selectivity, processing cost, and the priorities of neighboring operators. By analyzing the priorities of neighboring operators, "Chains" of operators can be scheduled to run together. Chain is shown in [3] to excel in keeping total queue size to a minimum. It may however suffer from poor response time during times of bursty arrivals.

2.2 A Running Example

To gain some intuition on how different scheduling algorithms impose different effects on query processing performance, let us consider the query plan in Figure 1. The plan contains three filter operators Op_1 through Op_3. The input stream has an average arrival rate of 1 tuple per time unit. However, to simulate the bursty arrival pattern that is commonly observed in a streaming system, we assume the tuple arrival rate is 2 tuples per time unit for the first 3 time units, and no tuple arrives for the next 3 time units. So the average tuple arrival rate in the first 6 time units is (3 $\times$ 2 + 3 $\times$ 0) / 6 = 1. At time unit 0, all queues are assumed to be empty. Each operator has two parameters: σ and t. σ represents the selectivity of operator. t refers to the average time the operator takes to process one tuple.

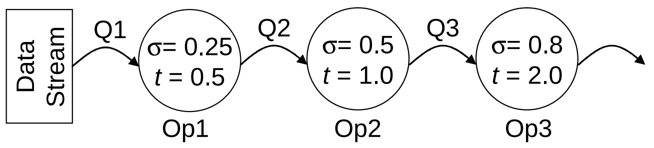


Figure 1. Selectivity σ and Average Tuple Processing Time t for the example query plan.

Time	Queue Size (Q1)	Queue Size (Q2)	Queue Size (Q3)	Queue Size (Total)	Throughput
0	0	0	0	0	0
1	2	0	0	0	0
1.5	1	0.25	0	1.25	0
1.75	1	0	0.125	1.125	0
2	3	0	0	3	0.1
2.5	2	0.25	0	2.25	0.1
3	4	0	0	4	0.2
3.5	3	0.25	0	3.25	0.2
4	3	0	0	3	0.3

Table 1. FIFO

Time	Queue Size (Q1)	Queue Size (Q2)	Queue Size (Q3)	Queue Size (Total)	Throughput
0	0	0	0	0	0
1	2	0	0	0	0
1.5	1	0.25	0	1.25	0
2	2	0.5	0	2.5	0
2.5	1	0.75	0	1.75	0
3	2	1	0	3	0
3.5	1	1.25	0	2.25	0
4	1	1	0.125	2.125	0

Table 2. MTIQ

For simplicity let us assume that switching between operators takes zero time. For the example in Figure 1, the average time the query plan takes to process a newly arrived tuple can be calculated as 1 $\times$ 0.5 + 1 $\times$ 0.25 $\times$ 1 + 1 $\times$ 0.25 $\times$ 0.5 $\times$ 2 = 1. So on average the query system has enough resources to keep up with the incoming data rate.

We now use this example to show how two scheduling algorithms, FIFO and MTIQ, differ in their scheduling choices and the consequent query performance in terms of total tuples in queues and query throughput. FIFO and MTIQ are chosen here as the illustrating scheduling algorithms because they can be easily quantified while exhibiting interesting behaviors.

Table 1 summarizes the number of tuples in queues and the throughput (the number of tuples Op_3 outputs thus far) for FIFO. At time unit 1, two tuples arrive in Q1. FIFO first removes 1 tuple from Q1, processes for 0.5 time unit, and outputs 0.25 tuple. This 0.25 tuple is then processed by Op_2 and 0.125 tuples are output. Finally, Op_3 consumes the 0.125 tuple and outputs 0.01 tuples. While FIFO is focusing on propagating the "older" tuples through the query plan, more tuples would continue to enter the input queue of Op_1.

For MTIQ, as shown in Table 2, at time unit 1, Op_1 takes 1 tuple from Q1 and produces 0.25 tuple. MTIQ will continue to run Op_1 as it has the most tuples in its input queue. At time unit 2, Op_1 would have enqueued totally 0.5 tuples in Q2. Meanwhile, two new tuples arrives in Q_1 and MTIQ will choose to run Op_1 yet again. The process continues until when Op_3 finally would have accumulated the largest input queue. Only at that point Op_3 would be scheduled and some output would finally be produced by the query plan.

In summary, FIFO focuses on providing a steady output, especially a much quicker first output, while MTIQ keeps its total queue size small but does not output any results for a relatively long time. As a consequence MTIQ's result output pattern is more bursty than FIFO's.

2.3 Experimental Evaluation of Scheduling Algorithms

We have experimentally evaluated the four scheduling algorithms described in Section 2.1, namely Round Robin, FIFO, MTIQ and Chain, in our continuous query system called CAPE [18]. Figure 4 (b) shows the query plan used in the experiments. Detailed information on the experimental setup can be found in Section 4.1.

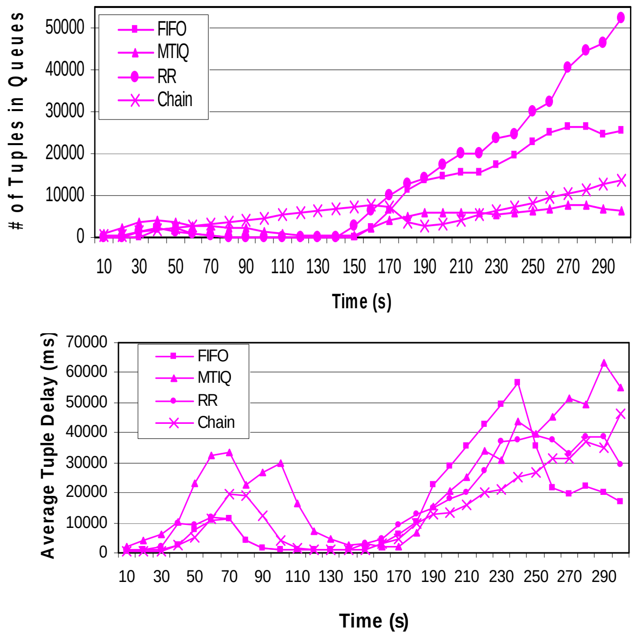


Figure 2. Performance of Scheduling Algorithms with a Two-Stream Query Plan.

Figure 2 shows the performance of these algorithms with regard to the total queue sizes and the result latency. The result latency is the average time that an input tuple takes to be output as a result. As anticipated, Chain and MTIQ perform the best when it comes to minimizing queue sizes. As discussed in Section 2.1, Chain processes operators that can most quickly remove the largest number of tuples. MTIQ processes operators that have the largest queues. Thus these two algorithms are excellent at minimizing queue sizes. However, the results are very different when it comes to the result latency. MTIQ and Chain end up being the two worst ones near the end of the recorded execution. FIFO, which was only mediocre on minimizing the queue size, actually does quite well minimizing the result latency.

Overall we observe from Figure 2 that no one algorithm is outstanding regarding both performance metrics. The key idea that can be deduced from this is that we need to use the best algorithm available at a given time to optimize the performance requirements. This is the fundamental principle exploited by our solution outlined in the next section.

3 Adaptive Scheduling Selection Framework

3.1 Overview

Given a set of state-of-the-art scheduling algorithms, AMoS employs a learning mechanism to empirically learn the behavior of these algorithms. Based on the learned knowledge, it adaptively selects the algorithm that has been statistically shown to best meet the optimization objectives. The scheduling selection process is executed periodically and consists of three steps listed below.

1. *Scoring statistics*: update the statistics of overall historical scheduling behaviors using the statistics of the scheduling algorithm that was just used, denoted as $A_{current}$. Then the historical statistics and the statistics for $A_{current}$ are scored.

2. *Ranking scheduling algorithms*: use scored statistics to evaluate $A_{current}$ against all the other scheduling algorithms that have been used.

3. *Selecting next candidate algorithm*: select the scheduling algorithm to use next based on algorithm ranking.

Figure 3 shows the architecture of AMoS. All candidate scheduling algorithms are maintained in the *scheduling algorithm library*. Newly developed algorithms can also be easily added into the library. The *algorithm evaluator* takes statistics collected by the system statistics collector to evaluate the performance of each algorithm. The changing performance requirements are also input to the algorithm evaluator. The *algorithm selector* then employs the *learning strategy* to select the next algorithm to use.

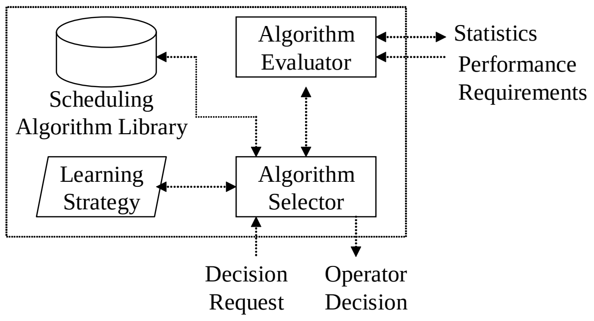


Figure 3. Architecture of AMoS.

Several issues have to be tackled in the design of AMoS. First, a scoring function needs to be developed to quantify how well a scheduling algorithm is performing regarding certain performance metrics. This function should allow the statistics of individual performance metric to be weighed for relative importance and be normalized such that one algorithm can be ranked against others. Second, AMoS should be able to judiciously choose the next scheduling algorithm. It must be able to weigh the benefits of choosing an alternate algorithm versus staying with the current one. Hence the learning strategy needs to be carefully chosen such that it favors the well-performing ones, but still allows the other algorithms to be periodically explored. In Sections 3.3 – 3.5, we will provide details on how we address these issues in AMoS. Before that, we first introduce in Section 3.2 how our system enables users to specify multi-dimensional performance requirements. Such performance specification is used by AMoS as a benchmark to determine how well or poorly each scheduling strategy is performing.

3.2 Specifying Performance Requirements

Our continuous query system provides a panel for the system administrator to input their performance requirements on query execution, which we call *performance specification*. The performance specification is composed of requirements on one or more performance metrics. Each requirement consists of three parts: (1) the *performance metric* that is to be controlled, (2) the *quantifier*, either *maximize* or *minimize*, that specifies how the administrator wants to manage this performance, and (3) the *weight*, which is the relative priority of each requirement, with the sum of all weights equals to 1. Using this panel, the administrator can also revise the performance specification during query execution by adding or removing some performance metrics or adjusting the weight of any of the performance metrics.

We choose to quantify the performance requirements using relative priorities instead of absolute numbers such as "output 10 results per second" or "keep no more than 5000 tuples in queues" due to the following reasons. First, the absolute performance requirements are often impractical to achieve because they are dependent on many uncontrollable factors such as data value distributions or stream arrival rates. Second, in most cases we only care about the relative importance of each performance. For example, we may be primarily interested in minimizing the total queue size, say 75%, while caring to a less degree about minimizing result latency, say 25%, as shown in Table 3.

Statistic	Quantifier	Weight
Total queue size	minimize	75%
Result latency	minimize	25%

Table 3. Example Performance Specification

Throughout this paper, we will work with the following performance metrics:

- *Result output rate*: the number of tuples output by the query per time unit.
- *Total queue size*: the total number of tuples stored in all the queues within a query plan.
- *Result latency (or tuple delay)*: the average delay from the time a tuple enters the query system until it is output as a result.

3.3 Scoring Statistics

During query execution, the statistics collector in the system will use the new statistics collected from the query plan executor to update the statistics that are related to the targeted performance metrics. These updated statistics serve as feedback for assessing how well the scheduling algorithms that was just used, i.e., $A_{current}$, has performed compared to other algorithms that was used before. Since the performance specification can be multi-dimensional, the evaluation of an algorithm should consider all related statistics. So, as the first step, we evaluate each statistic *i* in a normalized way.

$$z_{i\_new} = \frac{(\mu_i^C - \mu_i^H)}{max_i^H - min_i^H} \cdot (1 - decay) + z_{i\_old} \cdot decay \quad (1)$$

For each candidate scheduling algorithm, we keep an array of normalized statistics scores calculated by Equation 1 for the latest time that this algorithm is run. For currently running algorithm $A_{current}$, we use Equation 1 to update score z_i for each statistic i of $A_{current}$. The new score z_{i_new} is computed based on the historical statistics (with superscript H), the statistics from executing $A_{current}$ (with superscript C), the old score z_{i_old} and a *decay factor*.

The historical statistics cover all statistics collected so far[1], including the statistics of $A_{current}$. Each time the statistics for the currently running algorithm are updated, the historical statistics are updated as well with the new statistics. Note that the historical statistics cover the behavior of *all* the scheduling algorithms that have been used, not just the history of $A_{current}$. In Equation 1, μ_i^H, max_i^H and min_i^H respectively denote the mean, maximum and minimum values of statistic i in the historical statistics. μ_i^C denotes the mean value of statistic i from using $A_{current}$.

The *decay factor* is used to exponentially decay out-of-date data to give a higher priority to the data that are collected the most recently, since this data is the most relevant to the current state of the system. The decay factor has a value in the range of (0, 0.5), so more weight is given to current data [2]. We can see that since $min_i^H \leq \mu_i^C \leq max_i^H$ and $min_i^H < \mu_i^H < max_i^H$, both z_{i_old} and z_{i_new} are bounded in the range of (–1, 1).

3.4 Ranking Scheduling Algorithms

As the next step, we use Equation 2 to compute the score for a scheduling algorithm. The score of each algorithm is based on the scores of all the statistics related to the performance requirements. In Equation 2, the total number of related statistics is denoted by symbol *I*. To compute the score for an algorithm, the normalized score for each statistic is multiplied by its corresponding weight w_i specified in the performance specification. The quantifier for each performance requirement in the specification is used to determine the value of q_i. If the quantifier is to maximize the performance metric, q_i is 1. If it is to minimize, q_i is –1.

$$Score_A = \sum_{i=1}^{I} (q_i \cdot z_i \cdot w_i) + 1 \qquad (2)$$

If the performance specification states that a certain statistic should be minimized, then ideally the statistics score generated by Equation 1 should produce a negative value. This implies that $A_{current}$ yields a lower value than the overall performance recorded so far regarding this statistic. Similarly, a statistic to be maximized should ideally yield a positive score. By applying the quantifier q_i, the negative to-be-minimized values (less than overall) will be flipped to a positive value and thus they contribute more to the scheduler's score. Consequently, the negative to-be-maximized values will be subtracted from the score.

Therefore, this equation gives a better score to performance requirements with a high weight and a high z value from Equation 1. The weighed sum will yield a value within (–1, 1) for each scheduling algorithm. We then add 1 to shift the range to (0, 2) so that all algorithms will have a positive score, which will be used by the learning strategy described in Section 3.5. This way we map a complete set of statistics for a scheduling algorithm into a single value that can be compared against the other algorithms.

The score of an algorithm is not based solely on the previous time that it was applied, but rather is an exponentially smoothed average value over time. While the performance of an algorithm is largely coupled to the characteristics of the data, over time the score of the algorithm should reflect its true potential. Therefore, the system is capable of handling reasonable fluctuations in data characteristics.

3.5 Adaptive Scheduling Selection

After each algorithm has been given a score based on its performance, the system is in a position to decide whether $A_{current}$ should be used again or if better performance could potentially be achieved by changing to another algorithm. It does this by comparing the score of $A_{current}$ with the scores for all of the other algorithms (that have run so far). For this, we use a simple Radix sort to rate these algorithms in linear time. Thus this comparison is cheap.

Several issues are considered when using the scores to determine the next scheduling algorithm:

1. Initially, all scheduling algorithms should be given a chance to "prove" themselves. Otherwise the decision would be biased against the algorithms that have not yet run. Therefore, at the beginning of execution, we want to allow some degree of exploration on the part of the adapter. We choose to initially run each algorithm in a Round Robin fashion because this is the fairest way to start the adaptive scheduling selection.

2. If we greedily choose the next algorithm to run instead of switching algorithms periodically during execution, a poor-performing algorithm may consequently run more often than a potentially better one. Hence, we need to periodically explore alternate algorithms.

3. Switching algorithms too frequently could cause one algorithm to skew the results of the next used algorithm. So when a new algorithm is chosen, it should be used for enough time such that its behavior is not significantly over-shadowed by the previously running one. For this purpose, we empirically set the delay threshold before reassessing the potential of a switch.

Once each algorithm has had a chance to run, there are various learning strategies that could be applied to determine if it would be beneficial to change the algorithm. In the current stage, AMoS employs the *Roulette Wheel strategy*

[1]In our implementation, we only store the synopsis of these statistics.

[2]The first time a statistics score of an algorithm is calculated, the decay factor is set to 0.

(also referred to as *fitness proportion selection*) [16]. AMoS is extensible so that other learning strategies, if found to be more effective, can be easily plugged in. The Roulette Wheel strategy assigns each algorithm a slice of a circular "roulette wheel" with the size of the slice being proportional to the individual's score. Then the wheel will be spun once and the algorithm under the wheel's marker is selected to run next. Since the slice of the roulette wheel for each algorithm is proportional to its score, the well-performing algorithm will have higher probability to be selected than other ones. Meanwhile, other algorithms also have chance to be explored. This appropriately solves the second issue described above.

The Roulette Wheel strategy was chosen also because it is lightweight. The lightweightness is of critical importance to AMoS as a meta-scheduling framework because it enables quick response to performance degradation and quick detection of possible performance improvement. We observe close to zero overhead when using this strategy to evaluate the scheduling algorithms in our experimental study, as will be shown in Section 4. In spite of its simplicity, this strategy has been shown by our experimental results (Section 4) to be effective enough to help AMoS outperform all single scheduling algorithms significantly.

4 Experimental Evaluation

4.1 Experimental Setup

We have implemented the AMoS adaptive scheduling selection framework in the CAPE continuous query system [18]. To fully test the capability of AMoS, in our experiments, we use both synthetic data and real-world data from the Internet Traffic Archive [12]. For synthetic data we control the selectivity and the arrival rate of data streams. The tuple inter-arrival time conforms to Poisson distribution. We vary the Poisson mean every 5 seconds to model time-varying data arrivals. The Internet Traffic Archive data is sent based on the timestamp of each data item. They also have bursty arrivals at times. We choose these data sets in order to show that under both steady and bursty data arrivals, the adaptive framework can respond with good query performance. All data streams are sent across a 10 BaseT LAN to achieve a realistic environment in which data streams are from remote sources. The machine that runs the continuous query engine with AMoS has a 1.2 GHz Inter(R) Celeron(TM) CPU and a 512 MB RAM, running Windows XP and Java 1.4.2 SDK.

AMoS has been tested with an extensive array of different query plans ranging from 2 to 90 operators. Here we report our comparative results using existing query plans extracted from a recent scheduling paper [3] to provide the basis for comparing AMoS with published schedul-

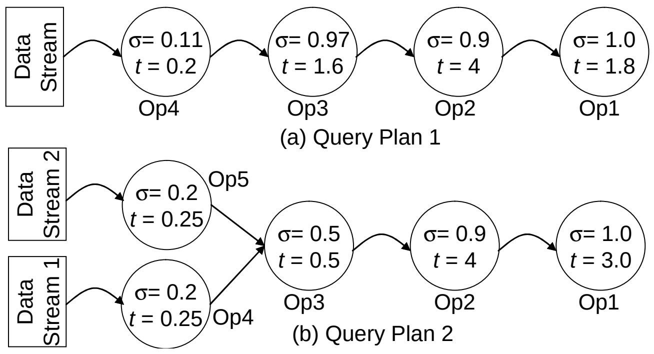


Figure 4. Query Plans used in Experiments.

ing algorithms. The query plans are shown in Figure 4 with selectivity denoted by σ and average tuple processing time by t. The selectivity of the window join operator with a window of 200 milliseconds is defined as $\sigma_{join} = \frac{N_{output}}{N_{leftinput} \cdot N_{rightinput}}$.

4.2 Comparison with Existing Algorithms

The first set of experiments uses a performance specification with only one requirement, i.e., to minimize memory overhead or to minimize tuple delay. These experiments are done to demonstrate that even for only one requirement, the adaptive framework is able to yield better or at least equal performance to individual algorithms most of the time.

In Figure 5 we can see that the adaptive framework outperforms all individual scheduling algorithms with regard to minimizing tuple delay. By tracing the execution history of all these algorithms, we observe that the adaptive framework achieves such performance gains by exploiting *beneficial interactions* between different scheduling algorithms. For example, FIFO excels in minimizing tuple delay. However, it may cause a large number of input tuples to accumulate in the input queues of the query plan after running for a while. On the other hand, MTIQ can take advantage of such queue buildups. When queue buildups arise, MTIQ will be selected (at time t = 7 in Figure 5) and it will progress tuples through the query plan more rapidly. After a while, FIFO will be selected again (at time t = 21) because older tuples still remain in the query plan and need to be processed.

Figure 6 shows that the adaptive framework also often outperforms any single algorithm in terms of reducing memory overhead.

4.3 Reaction to Changing Specifications

Secondly, we evaluate how well the adaptive framework works with two performance requirements, in particular, average tuple delay and result output rate. Most importantly,

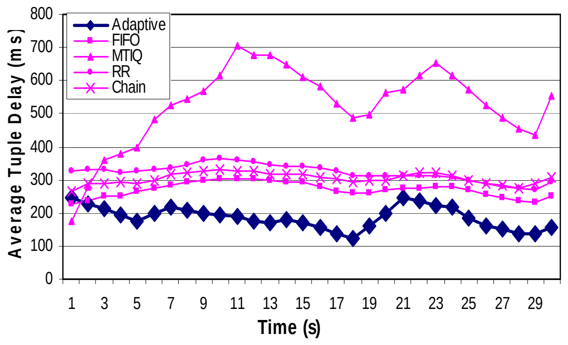


Figure 5. One Performance Requirement: Minimizing Tuple Delay.

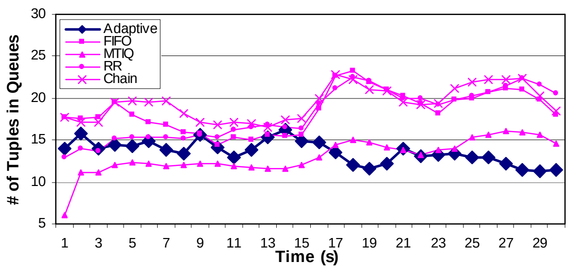
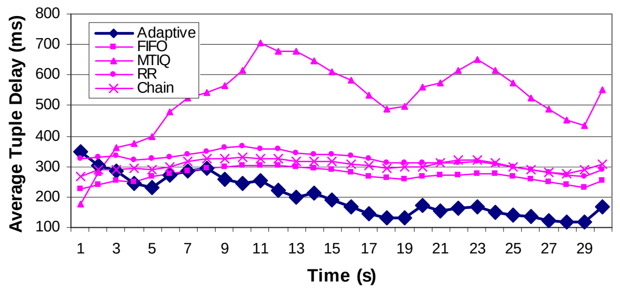


Figure 6. One Performance Requirement: Minimizing Total Queue Size.



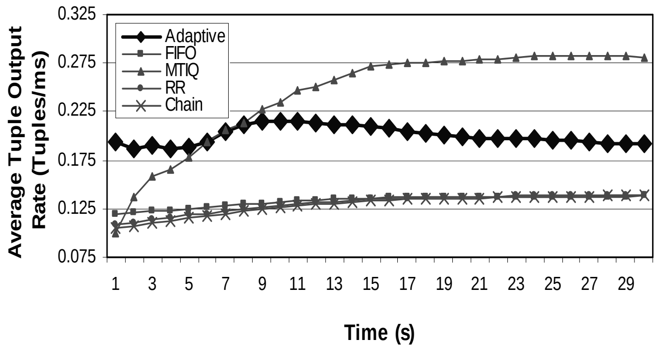


Figure 7. Two Performance Requirements with 70% Focus on Minimizing Tuple Delay, and 30% on Maximizing Output Rate (Plan 1)

we test that as the importance of the performance requirements changes, how efficiently the adaptive framework will acknowledge such change and adapt accordingly.

In Figure 7 we depict the results of an experiment for which we place 70% importance on minimizing tuple delay and a 30% importance on maximizing result output rate. We observe that the adaptive framework outperforms each single algorithm regarding average tuple delay, and performs about average regarding the result output rate. Since average tuple delay is of higher priority, the scheduling algorithms that favor this metric such as FIFO will be used for most of the time. Due to the frequent context switchings between operators caused by FIFO scheduling, the average processing cost for a single input tuple will be increased accordingly. Therefore, another performance requirement, the result output rate, may not be satisfied as well. However, the overall performance of the adaptive framework is better than any of the individual algorithms.

In the experiment shown in Figure 8, we adjust the weights to be 70% on maximizing output rate and 30% on minimizing tuple delay. We observe that with such change in requirements, the adaptive framework still does exceptionally well at minimizing tuple delay, and improves significantly at raising the output rate. This is because the adaptive framework can adaptively adjust the frequency of executing each scheduling algorithm according to their potentials on achieving the desired performance. When more weight is put on maximizing the output rate, the algorithms that are good at this will be used more often than others.

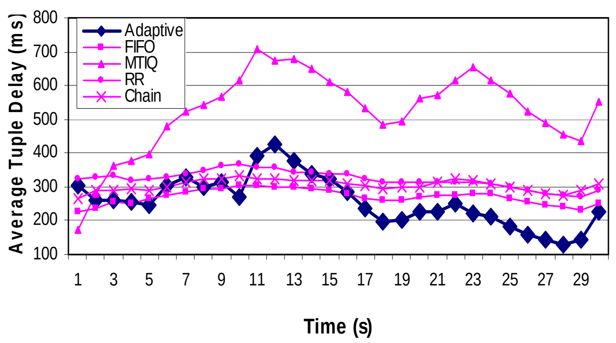
4.4 Handling Multi-Faceted Specifications

Now we compare the performance of the adaptive framework against the individual scheduling algorithms with a performance specification of three requirements – tuple delay, result output rate and memory overhead. Each performance requirement is given equal weight.

In Figure 9 we can see that the adaptive framework again performs well under all three performance requirements, with the biggest improvements on average tuple delay and memory overhead. This is again due to the beneficial interactions between various algorithms.

5 Related Work

There is a recent surge of ongoing research in continuous query processing [1,6,7,11,19]. Resource management has been recognized to be one of the important issues to be addressed [17]. To effectively allocate resources among



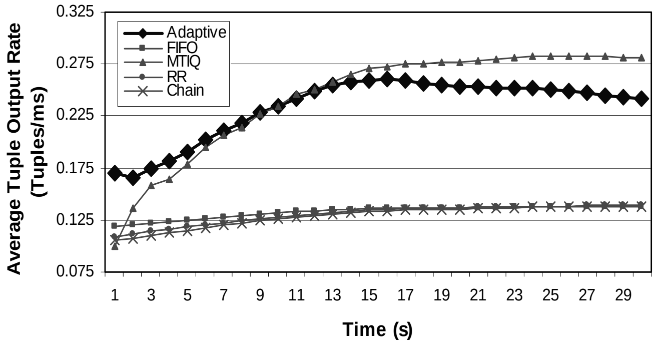
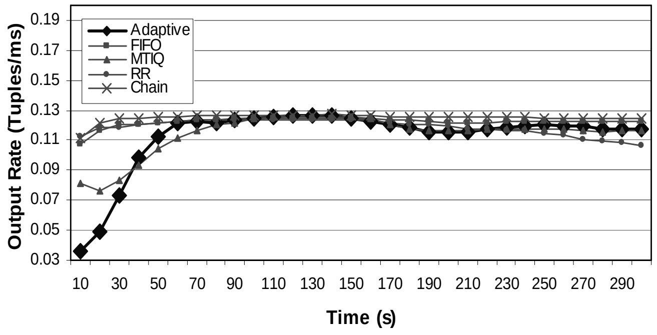
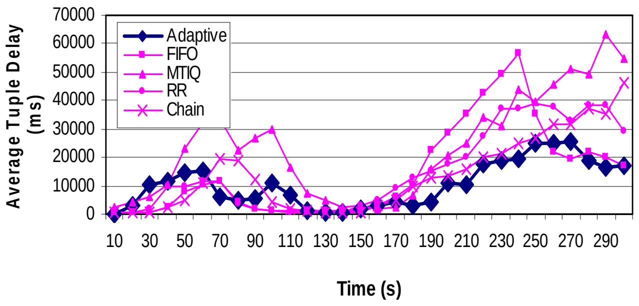


Figure 8. Two Performance Requirements with 30% Focus on Minimizing Tuple Delay, and 70% on Maximizing Output Rate (Plan 1)





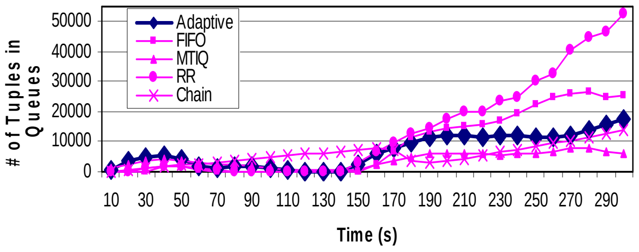


Figure 9. Three Performance Requirements with Same Priority (Plan 2)

possibly a large number of operators in a continuous query plan is critical to achieve desired query performance.

The research effort on resource management that is most related to ours is operator scheduling, i.e., allocating CPU time among query operators. Besides borrowing scheduling strategies from the operating systems realm, such as Round Robin, some continuous query systems also propose more fine-tuned adaptive scheduling algorithms including Chain [3], Train [5] and path capacity [13] strategies. Each of these scheduling strategies aims to optimize one performance objective, such as minimizing total queue size or minimizing result latency. The threshold strategy [13] addresses the combined optimization of both result latency and memory requirement. Our work differs from these prior works in that our system is able to target any number of performance objectives with flexible-configured priorities.

Another line of research on scheduling concentrates on how to schedule data. Rate-based stream scheduling [23] deals with ordering the execution of input streams so to maximize the query output rate. Eddies [2, 15] uses a lottery-type scheduler to route tuples to an available operator. The goal is to prevent tuples from waiting in the input queues of a slow or busy operator.

Besides CPU time, memory is another important resource that should be taken good care of. [4, 9, 10, 22] exploit semantic constraints of streaming data, such as clustered or ordered data arrivals, to discard no-longer-useful data from operator states in a timely manner. [7,15] research on grouping queries or operators to minimize memory overhead. These issues are orthogonal to our work.

6 Conclusion

In this paper, we proposed a novel scheduling selection framework that leverages the strengths of individual scheduling algorithms to meet flexible compositions of performance requirements. Our framework uses the recent performance of each scheduling algorithm in determining how to best adapt operator scheduling given the current status of query execution. Using a lightweight lottery-based heuristic, an algorithm is selected based on its potential to yield better performance.

We have found that not only was our framework able to efficiently meet both single or multiple performance requirements, but it was also able to react to the runtime requirements changes. The framework succeeded where each single scheduling algorithm may fail due to their uni-objective nature. As a result, our technique is shown to aid several highly tuned algorithms, such as Chain [3], in areas where the algorithm would normally not produce satisfactory results. Our framework is general and can be easily plugged into existing continuous query systems.

As future work, we plan to incorporate and experiment with alternate scheduling algorithms, such as the Train strategies [5], path capacity scheduling and hybrid threshold scheduling [13]. We could also explore other adaptive heuristics and learning mechanisms. Another interesting open problem is to investigate the interactions between runtime query restructuring [25] and operator scheduling in order to derive an optimal query execution plan.

References

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

[2] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *ACM SIGMOD*, pages 261–272, June 2000.

[3] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In *ACM SIGMOD*, pages 253–264, 2003.

[4] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 39(3):545–580, Sep 2004.

[5] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 269–280, Jan 2003.

[7] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*, pages 379–390, May 2000.

[8] A. Dan and D. Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *ACM SIGMETRICS*, pages 143–152, 1990.

[9] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, March 2004.

[10] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM*, pages 92–101, Nov. 2004.

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

[12] Internet Traffic Archive. http://www.acm.org/sigcomm/ITA/, 2003.

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

[14] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[15] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD*, pages 49–60, June 2002.

[16] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1999.

[17] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, pages 245–256, Jan 2003.

[18] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB Demo*, pages 1353–1356, Aug/Sep 2004.

[19] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[20] H. Su, J. Jian, and E. Rundensteiner. Raindrop: A uniform algebraic query system for xqueries on xml streams. In *Proceedings of the 12th ACM International Conference on Information and Knowledge Management*, 2003.

[21] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX*, pages 13–24, June 1998.

[22] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 15(3):555–568, May/June 2003.

[23] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, pages 501–510, 2001.

[24] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *ACM SIGMETRICS*, pages 214–225, 1990.

[25] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, pages 431–442, 2004.