Adaptive Scheduling Algorithm Selection in a Streaming Query System

by

Bradford Pielech

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

January 2004

APPROVED:

Professor Elke Rundensteiner, Thesis Advisor

Professor Robert Kinicki, Thesis Reader

Professor Michael Gennert, Head of Department

Abstract

Many modern applications process queries over unbounded streams of data. These applications include tracking financial data from international markets, intrusion detection in networks, monitoring remote sensors, and monitoring patients vital signs. These data streams arrive in real time, are unbounded in length and have unpredictable arrival patterns due to external uncontrollable factors such as network congestion or weather in the case of remote sensors.

This thesis presents a novel technique for adapting the execution of stream queries that, to my knowledge, is not present in any other continuous query system to date. This thesis hypothesizes that utilizing a single scheduling algorithm to execute a continuous query, as is employed in other state-of-the-art continuous query systems, is not sufficient because existing scheduling algorithms all have inherent flaws or tradeoffs. Thus, one scheduling algorithm cannot optimally meet an arbitrary set of Quality of Service (QoS) requirements. Therefore, to meet unique features of specific monitoring applications, an adaptive strategy selector guidable by QoS requirements was developed. The adaptive strategy selector monitors the effects of its behavior on its environment through a feedback mechanism, with the aim of exploiting previously beneficial behavior and exploring alternative behavior. The feedback mechanism is guided by qualitatively comparing how well each algorithm has met the QoS requirements. Then the next scheduling algorithm is chosen by spinning a roulette wheel where each candidate is chosen with a probability equal to its performance score.

The adaptive algorithm is general, being able to employ any candidate scheduling algorithm and to react to any combination of quality of service preferences. As part of this thesis, the Raindrop system was developed as exploratory test bed in which to conduct an experimental study. In that experimental study, the adaptive algorithm was shown to be effective in outperforming single scheduling algorithms for many QoS combinations and data arrival patterns.

Acknowledgements

First, I would like to express my sincere appreciation and gratitude to my advisor Prof. Elke Rundensteiner for her help, guidance, support and encouragement. Without her feedback, ideas, suggestions, incredible responsiveness and the time she always had for me, this thesis would not have been achieved. I would also like to thank her for guiding me throughout my graduate studies.

I thank my reader, Prof. Robert Kinicki for his valuable feedback.

I also would like to thank the entire Raindrop team for the close collaboration in developing the base Raindrop system and invaluable help in achieving this thesis. Special thanks to Tim Sutherland for his help in experiments and analysis.

Also, I would like to thank Xin Zhang and the other members of the Rainbow team for all of their work on the base Rainbow system that served as the stepping stone to Raindrop. I'm thankful to all DSRG members.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Many modern applications process queries over unbounded streams of data. These applications include tracking stock and other financial data from various international markets, intrusion detection in networks [17], monitoring remote sensors, and monitoring patients vital signs at a hospital. These data streams arrive in real time, are unbounded in length and have unpredictable arrival patterns due to external, uncontrollable factors such as network congestion, weather (in the case of remote sensors), or objects moving in and out of sensor range. The data streams also can have high volumes of incoming data. The queries in such environments typically are long running, continuous (always running) and thus must be answered incrementally to avoid possibly infinite delay for any result.

At first, we shall consider if existing database management systems (DBMS) could be applied for the processing of such continuous queries. DBMSs have mature query optimization and indexing techniques and are used to store large volumes of business data. This would potentially be a good foundation for continuous query processing. However, because the applications described above work with continuously arriving data streams rather than data that had previously been stored, they have unique features that render traditional database management systems ineffective. These features include:

1. The database must be continuously updated in the form of INSERT queries and would require unbounded storage to handle the possibly never ending stream.

2. The only means DBMSs have to process rapidly arriving data is through the use of triggers. Triggers are query plans that are stored within the database and are executed on some event, such as a tuple being inserted or deleted. The downside to triggers is that they typically do not scale well beyond four or five simultaneous executions [5]. Thus, the monitoring application's scalability would be severely limited. An alternative approach to using database triggers would be to encode the queries in a middleware application built on top of the DBMS. However, this approach may not scale well because the middleware application would need to either continuously poll the database or rely on triggers to alert it of newly arriving data. The middleware application would have little control over query execution and planning and thus could not reoptimize a poorly performing plan [9] or efficiently share computation among multiple queries [6].

3. Some of these applications require real-time results and thus have clear result delay deadlines. The application needs to rely on partial or approximate results because not all of the data is available at any given time. However, the traditional DBMS was designed to answers queries in full, regardless of the time needed to produce them. Thus they do not meet this requirement. For example, a system monitoring valves at a chemical plant wants to have an alert sound when the average pressure over the last 5 minutes exceeds a given threshold. The traditional DBMS is not capable of producing any partial result because the "average" operator will block until it has seen all data and thus never produce a result.

4. Traditional DBMSs assume that all data is present when the query is issued and that the data will not change during execution (transaction processing). Thus DBMSs typically use a static query evaluation strategy where the optimization is performed before query is actually executed and no run time adaption is performed. In a streaming environment where many external factors affect data arrival patterns, a static optimization or execution strategy would likely not perform well. This is because a plan that initially performed well may have its performance deteriorate when the data arrives much quicker. In the case of the chemical plant monitoring system, if one sensor temporarily went off-line, query execution in the DBMS may need to likely wait until the sensor came back. A better solution would be to recognize that one data source is arriving slowly and thus perform other work instead to keep the system busy.

5. Because of the nature of continuous queries, certain applications may have Quality of Service (QoS) requirements based on domain-specific needs. For example, stock market system users are interested in receiving results as soon as possible (data rate), a content server wants maximum throughput, whereas a sensor network has strict memory requirements. Furthermore, a single application may contain several administrative-specified goals relating to how the server should process the data and each contains weights as a relative priority rating. The DBMS may not capable of explicitly addressing the server-specific QoS requirements, such as memory usage, or client-side QoS requirements, such as delay and output rate because it lacks the ability to alter execution to meet the goals (see item 4) if its initial execution strategy was not sufficient. The DBMS would not

Due to these and other limitations, several general purpose continuous query systems such as [16][5][15] are being developed. Such systems generally work as follows. First, the system subscribes to any number of data streams (like a stock market ticker) and makes these streams available for end user applications. Next those user applications issue queries against the streams that will run for long periods of time. The system processes these queries and provides the results to the applications as a stream of data. If the performance of the system begins to degrade, the system takes several possible measures to ensure an acceptable performance level.

Continuous queries typically have some measure of adaptivity built-in to cope with unexpected changes in the data. [16] uses an adaptive scheduling algorithm called Chain [3] that helps to keep memory usage down during periods of bursty arrival at the expense of throughput. This system also monitors resource allocation making use of load shedding (tuple dropping) when allocation grows too large. [15] optimizes the global query plan such that multiple user queries will efficiently share the computation. [5] allows an administrator to input QoS specifications and the system monitors execution performance based on these QoS metrics. If the QoS drops below an acceptable level, the system will shed load until the performance increases. This strategy however will produce results that are not necessarily representative of the data that was meant to be processed by the query plan. We believe that a framework must be developed such that the system can recognize performance degradation and adapt accordingly without dropping any data that is to be processed.

This thesis presents a novel technique to adapt the execution that, to my knowledge, is not present in any other continuous query system to date. This thesis hypothesizes that utilizing a single scheduling algorithm to execute a continuous query (as is currently employed in [16][5][17][20]) is not sufficient because all scheduling algorithms have inherent flaws or tradeoffs. For example, Round Robin does not consider the cost of executing an operator and thus may under-utilize inexpensive operators. Chain, employed in [16], works well at minimizing memory requirements, but this comes at the expense of an increased throughput. [20] focuses on maximizing output rates of operators, but does not consider the cost of executing an operator. Thus, we hypothesize one scheduling algorithm cannot meet an arbitrary set of QoS requirements. Therefore, to meet the unique features of query applications that were listed above, we propose an adaptive strategy selector that will be guided by some provided QoS requirements.

1.2 Motivating Scheduling Example

The scheduling policy chosen by the system can have a dramatic effect on the characteristics of the system, including the throughput (i.e., the number of result tuples produced), memory requirements, and delay (i.e., how long does data stay in the system before it is processed and sent to the end-application). The following scheduling example will illustrate this point and motivate the need for the adaptive strategy selector.

Consider the query plan in Figure 1.1 that contains three consecutive filter operators O_1 through O_3. O_1 is connected directly to the input stream and its output is placed into the input queue for O_2. O_2's results serve as input to O_3 and O_3 outputs its results to the end user application. Furthermore, because all operators have selectivity (σ, see Equation 3.2 in Section 3.3.1) less than one, the number of the tuples will decrease as they "move" through the system. Note that when we refer to a tuple, we are really referring to a group of tuples that are organized in some logical way, like on a disk page. Thus, it is possible to have fractional tuples.

Let us assume that the input stream will place one tuple in the input buffer of O_1 every time unit, starting at time t_o. Assume that context switches take zero time and we are running the filter operators on a single processor sharing CPU power and memory. When told to run, an operator will consume at most one tuple from its input queue, process the tuple for a fixed amount of time, C, and then output a fixed percentage, σ, of tuples. For instance, if an operator consumes 0.5 tuples from its input queue and its σ is 0.9, it will output 0.45 tuples and the operation will take C time units. Figure 1.1 shows the σ and C values for the operators in our example plan.

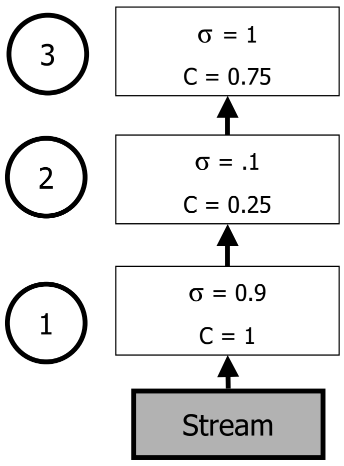


Figure 1.1: Selectivity σ and Average Tuples Processing Time C values for the example query plan.

Now consider two different scheduling strategies. Strategy one is a FIFO scheduler that will take tuples from the input queue of O_1 and process them until completion. Strategy two is a variation of a Greedy algorithm, called Most Tuples in Queue (MTIQ). MTIQ always runs the operator with the most tuples in its input queue(s). It is important to note that any scheduling algorithm will eventually produce the exact same query result, otherwise the algorithm is not correct. The difference in scheduling policies becomes apparent when looking at resource allocation, output rate, operator utilization, and freshness of results (how long did it take for the query to produce the result). This example will focus on the throughput and total queue sizes.

Table 1.1 summarizes the number of tuples in all queues and the throughput (number of tuples O_3 outputs) for each strategy as execution progresses. As you can see, the queue sizes for the FIFO scheduler will continue to grow at its present rate. The execution happened as follows: first one tuple is removed from the input queue of O_1 and after processing for 1 unit, 0.9 tuples are outputted (0.9= 1 x O_1's σ). Then 0.9 tuples are processed by O_2 and 0.09 are outputted (0.09 = 0.9 x O_2's σ). Finally, 0.09 are consumed by O_3 and 0.09 are outputted to the end user because the σ for O_3 is 1.

Time	FIFO Queue Size	MTIQ Queue Size	FIFO Throughput	MTIQ Throughput
0	1.0	1.0	0.0	0
1	1.9	1.9	0.0	0
2	2.0	2.8	0.09	0
3	2.9	1.9	0.09	0
4	3.0	1.9	0.18	0
5	3.9	1.9	0.18	0

Table 1.1: Queue Sizes and Throughput for Example Query plan after each Time Unit.

The queue sizes grow as they do because at t_0 there is 1 tuple in the queue. Then, during t_1, 0.9 tuples are in the queue for O_2 and one more tuple arrived into the system from the input stream (1.9 tuples total). Next, after t_2 the 0.9 tuples that were outputted by O_1 were processed by the remaining operators (hence the throughput increased) and removed from the system and one more tuple arrived for O_1 to process, so 1.9 - 0.9 + 1 = 2.0 tuples. This cycle is every two time units.

It is a little tougher to intuitively follow what happens in the MTIQ example because after t_2, operators run over the course of 2 time intervals. MTIQ behaves the same as FIFO during t_0, but differs starting with t_1. At t_1, there is one tuple queued for O_1 and 0.9 for O_2. MTIQ chooses to run O_1 again. At t_2, there is 1 tuple for O_1 and 1.8 for O_2 (0.9 + 0.9), so MTIQ runs O_2. O_2 finishes at time $t_{2.25}$ (because it started at t_2 and processed for 0.25 time units) and now the queue sizes are 1, 0.8, 0.1 for O_1, O_2, O_3 respectively. MTIQ runs O_1 again and at t_3, there is one new tuple for O_1 and still 0.8 and 0.1 at O_2 and O_3, respectively.

The process continues and at about t_{14}, O_3 will have more than one tuple in its input queue and it will finally be run.

The MTIQ strategy keeps its queue sizes smaller than those of FIFO, but it does not output any results for a (relatively) long time. MTIQ's throughput is much burstier than FIFO's. MTIQ will take approximately 14 time units to output its first tuple. The next output will come slightly more quickly, but the output pattern will not be as regular as FIFO. FIFO outputs every 2 time units. Table 1.2 summarizes the positives and negatives of the two scheduling algorithms.

Algorithm	Positive	Negative
FIFO	Schedules operators with the same frequency, outputs tuples sooner and at a constant rate	Queue sizes grow quickly, Output rate is low (0.045 tuples per time unit), Does not utilize operators as fully as MTIQ
MTIQ	Queue sizes are smaller, higher output rate (0.07 tuples per time unit), more fully utilizes operators	Bursty output pattern, tuples spend a long time in the system

Table 1.2: Summarizing the Positives and Negatives of FIFO and MTIQ from Example.

1.3 Adaptive Scheduling Approach and Background

In general, an adaptive system is a system that changes its behavior in response to a changing environment with the goal of improving performance [4]. The improved performance may be quantified as absolute or relative to some predetermined goal. The adaptive system monitors the effects of its behavior on its environment through a feedback mechanism, with the aim of exploiting previously beneficial behavior and exploring alternative behavior [14].

In the context of query execution, the adaptive scheduler selector will periodically evaluate the current scheduling algorithm's performance for the administration-specified QoS requirements and compare this with the other candidate algorithms' performance. This qualitative comparison is based upon assigning a fitness score [13] to each algorithm that captures how well it performed in several metrics, such as throughput, memory size, and output rate. The next algorithm is then chosen based on one or more heuristics, such as "always pick highest score" or "pick next algorithm with a probability equal to its score." This process is repeated continuously during the lifetime of the query.

The example from Section 1.2 has shown the relative strengths and weaknesses of two scheduling algorithms during a period of constant arrival rates. Expanding upon the example, say that the user's QoS requirement specifies that "40% weight be given to maximizing throughput and 60% to minimizing queue sizes." See Section 5.1 for a full discussion on quality of service requirements. During execution, while the stream was producing at the rate of 1 per time unit, assume the Greedy strategy adequately met this requirement (ignore the calculations of how well a strategy meets a requirement for now, they are discussed in depth in Section 5.2) and FIFO did not. Thus, the system utilized the Greedy algorithm.

Next, say that tuples began to arrive from the stream with an average rate of 2 per time unit instead of 1. Thus, we are not dealing with constant bit rate (CBR) streams, but rather variable bit rate (VBR) streams. Let's compare how each algorithm would perform. FIFO would behave the same, although its queue sizes would grow even more quickly than before. If we were using the Greedy algorithm, it may never produce any results, but the queue sizes would grow much more slowly than FIFO. To see the intuition of this, realize that O_1 will now have 2 tuples placed into its input queue at each time interval and O_1 can only process one tuple per time unit. After each interval, the queue size of O_1 will grow by 1 while the queue size of O_2 will grow by 0.9. Therefore, Greedy will never choose to run any operator other than O_1 (because O_1 will always have a larger queue size than O_2)!

Once the system has recognized that Greedy's queues are growing quickly, but no result is ever being produced, it will switch to FIFO because FIFO better meets the throughput requirement. Because this switch comes at the expense of queue sizes, the system may switch between the two algorithms such that it can leverage the throughput from FIFO and the queue size control of Greedy. If the system did not possess these adaptive qualities, either the memory usage would grow very rapidly and the output rate would be constant or memory usage would grow less quickly, but there would not be any throughput. Either way, QoS would not be met.

The goal of the adaptive strategy selection is to leverage the relative strengths and weaknesses of the various scheduling algorithms in order to guide the behavior of execution, such that it will meet the given QoS requirement. We assume ahead of time that we know all scheduling algorithms that are available for use in our system, but do not know anything about their relative strengths and weaknesses. This is important to keep in mind because if we are deficient in one metric, we cannot clairvoyantly find the algorithm that is best suited for improving that metric. Also performance of any one of the algorithms can fluctuate wildly as the data arrival characteristics change. For example, one algorithm may perform very well when the streams arrive at constant intervals, but break down precipitously during periods of bursty arrival.

This thesis will make use of a diverse set of greedy and fair use algorithms including Chain [3], a variation of Batch scheduling [5], Round Robin, FIFO, and Greedy. The system needs to determine which scheduling algorithms to consider as candidates to help answer each query. If too many are chosen the system will spend all of its time exploring strategies and not enough time running the best strategy. Too few will limit the adaptive abilities of the system.

1.4 Research Challenges

There are several challenges associated with creating an adaptive execution engine that can meet user's QoS requirements. First, a continuous query system and data stream generator must be built before any adaptive scheduling hypothesis can be tested. Second a metric needs to be developed that can quantify how well an algorithm performs relative to the arbitrary QoS requirements. The scoring function needs to 1) allow the individual goals to be weighed for relative importance and 2) normalize the collected statistics for those metrics such that one algorithm can be ranked against another (i.e., scheduler A meets the QoS requirements better than B does).

Next, the adaptive strategy selector needs to be able to intelligently choose the next scheduling algorithm to use. It must be able to weigh the benefits of choosing another algorithm vs. staying with the existing algorithm. No strategy can ever be completely eliminated from the selection set because there is no means to gauge the effect the stream arrival patterns had on the strategy's performance. Therefore, the adaptive strategy needs to be carefully chosen such that it favors the well-performing (relative to QoS requirements) strategies, but still allows the other strategies to be periodically explored. However, exploring too much will degrade performance because there is a non-zero overhead cost associated with switching the scheduler. The adaptive strategy selector needs to be made as generic as possible such that it may be applied to as many different applications as possible. Therefore, the selector will not make any assumptions about query languages, incoming data values, or query plans.

Perhaps the toughest challenges associated with creating the scoring function and choosing the adaptive strategy selector is ensuring the system will be able to meet the various QoS requirements better than a single algorithm. For example, if rotating between three algorithms in order to maximize throughput and minimize queue sizes did not yield higher throughput and smaller queue sizes than running just one algorithm, then the proposed adaptive techniques are not useful. The adaptor needs to be able to be self-monitoring in order to observe then assess this behavior. Therefore, an experimental framework needs to be developed that will be capable of evaluating the usefulness of the adaptive strategy for a variety of query plans and data arrival patterns.

It is important to notice that the goal of the adaptive strategy selector is not to “beat” any one algorithm at any one metric. Theoretically if there exists an algorithm that is optimal for one particular goal, then it will not be outperformed by a combination of itself and other sub-optimal strategies. Hence our adaptive framework should be able to recognize this situation and to indeed end up picking this winner most of the times. Rather, the goal of this work is to leverage the strengths of various scheduling strategies for a set of performance goals. This works under the assumption that no scheduling algorithm can meet every possible goal. This assumption indeed proved to be the case as our base experiments have confirmed.

Finally, an accurate statistics engine is also needed so the system can correctly and continuously assess how it is performing. There are two primary issues related to the design of a statistics engine: which statistics to collect during execution and the trade off between freshness of statistics and the overhead associated with the gathering of those statistics.

1.5 Outline

The remainder of this paper is structured as follows. Chapter 2 briefly reviews the related research. Chapter 3 describes the architecture of the system including the underlying query model, queue and operator structure, and the major modules. Several key concepts and terms necessary to understand the adaptive techniques are also defined in this section. Statistical calculations and metrics used are explained in Chapter 3.3. Chapter 4 describes the scheduling algorithms that were chosen and explains their advantages and disadvantages. Chapter 5 describes quality of service requirements in detail, including how they are structured and used. The section also details the algorithm scoring functions and strategy selection heuristics. Finally, Chapter 6 decribes the experiments used that will validate this work and contributions and conclusions are in Chapter 7.1.

Chapter 2

Related Work

There is a recent surge of ongoing research in the field of executing queries over streaming data. A comprehensive overview of the challenges of executing queries in a stream environment can be found in [4]. Most closely related to this work is that of STREAM [16] and Aurora [5].

2.1 Stream Query Systems

Several data stream processing systems have been proposed in the current database research. The STREAM [16] project's goal is to "manage resources carefully, and to perform approximation in the face of resource limitations in a flexible, usable, and principled manner." STREAM focuses on efficiently allocating memory to queues, synopses, and operators by making use of stream constraints and the Chain [3] scheduling algorithm. STREAM also provides techniques to best approximate the query result using various static and dynamic techniques such as dropping unimportant tuples and reducing the time that historical data is joined with current data.

STREAM differs from this thesis in the following ways. First, STREAM's Chain scheduler does not consider other heuristics such as maximizing tuple throughput or minimizing overall response time. Second, STREAM only supports one scheduling algorithm, namely Chain. While Chain works well in certain situations, Chain can fail in others (i.e., if high priority Chains are higher in the query plan, those operators will starve for input). In these cases, STREAM does not have any means to recover. In this work, if a scheduling algorithm starves or is ill-performing, the adaptive algorithm is able to choose an alternative strategy that will perform better. One of the primary contributions of this approach is the ability to adapt to any changing conditions in the data stream and thus, STREAM's performance should lag behind the adaptive strategy more if the arrival patterns change frequently. STREAM also does not allow for the system administrator to specify their own quality of service requirements.

Aurora [5] aims to reduce tuple execution costs while maximizing overall QoS. They accomplish this by having operators queue as many tuples as possible without processing and then the operator processes all tuples at once generating a train. The benefit is that tuples passed to subsequent operators do not have to go to disk and thus they incur less I/O time. Aurora allows the administrator to input a graph that defines what a "good" QoS means. Aurora takes into account many different QoS metrics such as response times, tuple drops, and importance of values. It allows for arbitrary compositions to be created. When the performance deteriorates (as detected by the QoS monitor), the load shedder is activated to bring the QoS to an acceptable level [5].

Aurora contrasts from our work in that Aurora makes use of one dynamic scheduling algorithm as opposed to adapting the scheduling algorithm depending on the circumstances. Aurora focuses on maintaining administrator-specified QoS requirements, as in this work, but the key difference is how the systems behave when poor performance is detected. Aurora assigns a priority to each tuple based on several heuristics and drops the unimportant tuples to improve performance. This strategy can be effective in some situations because it reduces the load on the system, however it leads to an approximated result. This work will always keep data that has arrived to ensure the accuracy of the result, but will alter the scheduling strategy in hopes of achieving better performance.

Tribeca [17] is a stream oriented system that was designed to analyze network traffic. Tribeca's goals are similar to this work, but does not support ad hoc queries, adaptive scheduling, and does not allow for administrator-defined QoS requirements.

2.2 Operator Scheduling

Several works focus on operator scheduling. The Chain algorithm [3] is a modified greedy algorithm that takes into account the importance of an operator relative to those around it in the query plan. Rate-based stream scheduling in [20] deals with ordering the execution of input streams so that the stream with the highest output rate will have a higher priority, and thus will be executed more often. The goal is to produce tuples as quickly as possible and to maximize throughput. They also take into account the relative importance of tuples, based on how well the system believes the tuple will contribute to the query answer, and strives to output the more important tuples quickly. Telegraph [2], [12],[11] is another adaptive query system that makes use of Eddies [2] to adapt the execution for each tuple. Eddies uses a lottery-type scheduler to decide which tuple should go to which Join operator. Eddies will dynamically route tuples to any available operator that will need to eventually process the tuple. The goal is to prevent tuples from waiting in input queues for a slow operator to be ready to process them. [12] extended the previous Eddies work by providing support for queries over stream. This level of adaption is much finer than compared to what is used in this thesis, although the idea of varying the tuple scheduling served as inspiration for this work. Eddies does not consider administrator-specified QoS metrics and thus, while the system is adaptive, it does not allow the administrator to customize the behavior of the system.

2.3 Query Plan Adaption

Various works [10][21] focus on adapting the query plan to better meet performance goals. This adaption can either happen during execution [9] by reorganizing poor performing query plans or before execution begins [22] by generating a better plan based on statistics from similar plans.

The Tukwila project [9] proposed the use of synchronization packets to "tell" each operator to complete the processing of the tuples in its input buffers so that the query plan may be reorganized. [21] proposed cost-based heuristics to dynamically scramble partial and complete query plans. NiagaraCQ [6][15] is a continuous query system that uses XML as data format. Niagara focuses on efficiently sharing processing between large amounts of continuous queries. In [22], NiagaraCQ was augmented with two rate based heuristics to consider when processing a user's long running or continuous query. The first heuristic optimizes for a specific time point in the execution process which answers "which plan will produce the most results by time t_0." The second heuristic optimizes for output production size, answering "which plan is the first one to reach N results." However, Niagara primarily focuses on generating an efficient query plan and does not focus on any execution-time scheduling issues, thus these heuristics do not map directly to our work. Niagara only considers those two rate heuristics and does not account for queue sizes.

The XJoin [18] operator was created in order to reduce the initial delay needed to produce results, efficiently create output tuples by breaking the Join into three stages, and keep the system occupied during periods of slow arrival rates. As future work, this thesis could incorporate an implementation of an XJoin operator to assist in meeting a “maximize throughput” QoS.

Chapter 3

Background

3.1 Architecture

A primary part of this thesis entailed developing the core continuous query architecture for the Raindrop system. Raindrop is made up of five primary components as shown in Figure 3.1. It acts as a middleware application between end user applications and the raw streaming data. End user applications submit queries to Raindrop and have results returned to them when they are available. The *Stream Receiver* is responsible for receiving the streaming data from various *Stream Sources* across the Internet and submitting the data to the *Storage Manager*. The Operator Scheduler orders the execution of operators according to a given scheduling algorithm. The Execution Engine (EE) actually runs the operators and the Statistics Gatherer (SG) manages statistics about the current system performance. This thesis deals with the EE and SG in depth and both of these modules are discussed in detail below. An overview of the Storage Manager and Operator Scheduler is provided at the end of the section.

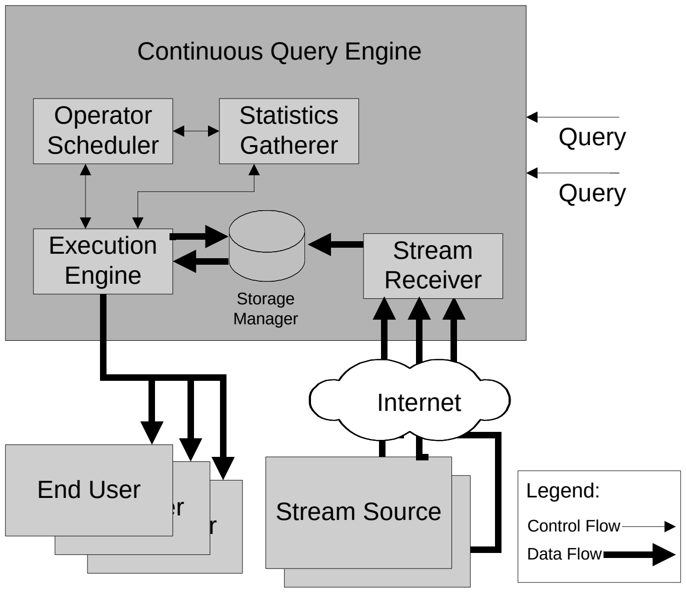


Figure 3.1: Raindrop System Architecture

3.2 Query Model

3.2.1 Query Plans

Raindrop executes a query plan over streaming sources. The query plan can be thought of as a directed acyclic graph (DAG) where the nodes represent query operators (Section 3.2.2) and the edges represent queues (Section 3.2.3) and an example can be found in Figure 1.1. The streams are connected at the bottom of the plan and the end user application resides at the very top. The operator(s) that connect directly to the end user application(s) are called the *roots* and those that connect to the streams are called *leaves*.

Assume that each user query in Raindrop is able to be maximally shared with the other query plans. That is, Raindrop is able to combine all similar operators and functions from one query plan with another, thus saving execution time. We also assume that the query plan does not change during the course of execution and that all user queries are specified ahead of time.

3.2.2 Operators

All query operators in Raindrop have been implemented in a pipelined, non-blocking manner. That is, every operator is capable of producing results after seeing only a partial data set and the operator will not block waiting for more input. Some relational operators, such as Select and Project can easily be implemented in this manner, while others, like Join, need a new implementation strategy [19].

The adaptive scheduling techniques will work over a generic set of algebra operators that do not depend on a specific query language or data format. This allows us to focus on the issues related to adaptive scheduling. In this paper, the query plans and operators are generalized. Two representative operators will be used, one for single streams and one for multi-streams. Both operators can be assigned a chosen selectivity and per-tuple processing time. To avoid an unbounded memory requirement, we implemented the multi-stream operator as a windowed-hash join.

3.2.3 Queues

Intra-operator data results are stored in main memory queues as tuples. Queues serve as the connections between operators and define the routes that tuples take during execution. Each queue maintains one pointer to the last tuple that has been processed by each consumer. Periodically the queue cleans up these pointers and removes filtered tuples in order to minimize memory consumption.

3.2.4 Execution Engine

The Execution Engine (EE) lies at the heart of Raindrop. It is responsible for asking the scheduler to determine the next operator to run, running that operator, updating statistics through the Statistics Gatherer (see Section 3.3), and then determining if the scheduling algorithm should be changed. Here is a walkthrough of the EE's tasks during execution:

1. Ask the Operator Scheduler to choose the next operator, Op, to run.

2. If the workload, the number of tuples available for an operator to process, for $Op > 0$, then update the statistics for Op's input and output queues and pass the workload to the operator. If the workload $= 0$, then there is starvation and the algorithm will pick another operator.

3. Run the operator. When the operator has processed its assigned work, control is returned to EE.

4. Update statistics for Op including: run_count, outputed_tuples, and time_run. Periodically update Op's selectivity, average tuple processing time, output rate, and priority, all of which can change depending on the characteristics of the data that arrives. See Section 3.3.1 for equations to calculate these statistics.

5. Determine if the scheduling algorithm should be switched out for an algorithm that better meets the user's QoS requirements. This check is done periodically, on the order of seconds, based on some administrator-defined frequency.

6. Repeat steps 1-6 for the duration of the query.

The EE needs to ensure that the system will not deadlock when an algorithm starves because some scheduling algorithms are inherently prone to starvation (i.e. any Greedy algorithm). Therefore, if an algorithm chooses an operator to run and the operator has no work to do, the EE will note the starvation and ask the algorithm to choose another operator. This process will continue until either the algorithm either chooses an operator that has work to do or a starvation threshold is reached. If the threshold is reached, the EE will change the scheduling algorithm.

The Execution Engine's behavior is strongly influenced by the system administrator's parameter setting. The system administrator is responsible for setting the frequency that certain statistics are updated and how often the EE checks the performance of the current scheduling algorithm relative to the quality of service requirements. A poor choice of parameters can cause a large increase in the system overhead due to updating statistics too frequently. It can also cause the EE to stick with a poor algorithm too long. Therefore, careful tuning is needed to ensure the EE performs optimally.

In our current setup, the EE only asks for the scheduling algorithm to choose one operator at a time. This simplifying assumption was made because the overhead to choose the next operator was experimentally shown to be negligible. With some scheduling algorithms, such as round robin, the algorithm can determine the complete running order for operators. On the other hand, greedy algorithms such as most tuples in queue, cannot predict which operator will be run next because the system state is constantly changing.

Unlike Eddies [2], the execution runs in a single thread to be able to assess the effectiveness of adaption. Future work can investigate the benefits of running multiple operators at the same time, but for now, only one operator executes at a time.

3.2.5 Statistics Gatherer

In order for the query plan to be executed according to user-defined priorities, cost formulas must continuously be evaluated to measure how well execution is performing relative to the desired behavior. To aid in the evaluation of these formulas, a comprehensive statistics engine was created to store, calculate, and sort statistics related to various operators and queues in the system. All statistics related to query objects are stored in one place to allow for efficient analysis of system state.

The Statistics Gatherer's (SG) foremost requirement is efficiency both through storage and in retrieval abilities. If the bulk of the execution time were spent calculating statistics, then the system would not be able to adapt very well. It also needs to be compact and efficiently support queries. Therefore, the statistics gatherer is analogous to a database that only stores its data in main memory. Like a database, the statistics gatherer can also be tuned by the system administrator. The administrator has control over parameters such as the size of historical data kept, weight of old data compared to new values, and frequency that data is sorted and indexed.

Retrieving and Updating Statistics.

Traditional (relational) cost models primarily gather statistics on selectivity of predicates and estimated sizes of relations [8]. The optimizer then uses these statistics, which can be updated at various time intervals, to determine how to organize the query plan and how to schedule execution. In comparison, SG stores statistics to try to quantify the unpredictable nature of a streaming execution.

During execution, both the scheduling algorithms, execution controller, and operators access the statistics gatherer to either update or retrieve (query) statistics about other operators or queues. After an operator has executed, the execution controller updates statistics related to how long an operator ran, how many tuples were produced, how many tuples were consumed, and how many times an operator ran. Periodically, the controller updates aggregate statistics for operators including selectivity, output rate, average tuple processing cost, and priority. The various scheduling algorithms query the SG to find properties about operators such as which operator has the highest priority, lowest processing cost, or highest output rate. The SG also keeps track of statistics regarding queue sizes.

Because we assume that all of the statistics will fit in main memory, we can make use of a nested-hashtable that will keep the cost of retrieving and updating values nearly constant. This is important because often updating one statistic requires querying values of several other statistics and we want to minimize the time needed to update a statistic.

For example, to update the average tuple processing time, the calculator needs to retrieve the number of tuples the operator has inputted and the time the operator has run. SG also exposes an interface to allow for the query objects to be returned based on defined filters, such as min, max, highest, or lowest. This allows scheduling algorithms to retrieve the operator that has the highest priority or most tuples in input queues.

Statistics Organization.

The individual statistics for the operators are organized in views in the SG where each scheduling algorithm has a view created that represents the operator’s statistics while that algorithm was selected by the EE. There are two additional views of the statistics created, one containing the overall statistics for the operators and one containing generic operator statistics. The overall or historic view provides a means to see how each operator has performed regardless of the scheduling algorithms employed thus far. The generic view is used to store statistics about the current state of an operator, such as the number of tuples enqueued in the operator’s input or output queue(s).

The scheduling algorithm views allow for the SG to provide some way for the execution controller to determine how well the current scheduling algorithm is performing compared with the others. Using the views, the EE can make queries to the SG to compare the value of statistic 1 for operator A for the Round Robin and Greedy algorithms. Furthermore, using the overall view, the EE could compare the performance for statistic 1 for Round Robin to the overall performance, which includes an aggregate of Round Robin and the other possible algorithms. Note that if only one algorithm has been employed thus far in execution, then the historical view will be equivalent to the view for that algorithm.

3.2.6 Storage Manager and Operator Scheduler

The Storage Manager is responsible for storing the data that has either arrived from the data streams or been generated during the execution of the query. The Storage Manager utilizes efficient algorithms for writing and reading the data from disk and memory and is essential for the system to process queries quickly. The Query Processor provides an interface to the external user applications to submit queries over the data streams. The Query Processor will generate query plans that will then be sent to the EE.

3.3 Statistics

3.3.1 Supported Statistics

This section defines the statistics that are calculated by the statistics gatherer and the formulas used to calculate them. Statistics are calculated for query objects and a query object is a generic base class for all operators, tuples, and entire query plans. Statistics can also be an aggregate of any number of query objects. Table 3.3.1 describes the notations and variables that will be used in defining these statistics. In general, the value of a statistic will be represented as $A(B)$. This is interpreted as "retrieve the value of statistic B for the query object A". The possible query objects are O, T, and Q for operator, tuple, and query plan, respectively.

Note:

1. $O(n^p_{total})$ is equivalent to the number of output tuples that the operator O could have produced during *all* of the total time that the operator has been run, $O(t'_{total})$. That is, if an operator has one input queue of size x, then $O(n^p_{total})$ = x. If an operator

var name	description
t'	Unit of time
$O(n^i)$	Number of tuples in all of the input queues for operator O
$O(n^p_{total})$	Total number of tuples processed by operator O during all t'
$O(n^o_{total})$	Total number of tuples outputted by operator O during all t'
$O(t'_{total})$	Total time units that operator O has run
$T(t'_a)$	Total time tuple T has spent in the system
$Q(n)$	The number of operators in query plan Q
$Q(n^r)$	The number of root operators in query plan Q

Table 3.1: Variables and notations used in the forthcoming equations

has i input queues with sizes x_1 to x_i, $O(N^p_{total})$ is equivalent to the product of those sizes.

2. $O(n^o_{total})$ represents the total number of tuples that operator O has outputted during all of its time slices.

3. Frequently, the weighed average of statistics is used to calculate other statistics. This average is calculated by the Equation 3.1, where w is the weight given to the older value. Setting w higher will force the system to "remember" the older values for longer. Thus over time the value is less likely to fluctuate. On the other hand, lower w values will cause the average to fluctuate more.

   In Equation 3.1, $A_{average}(B)$ represents the new average value of statistic B for query object A, $A(B_{new\_value})$ represents the most recent value of statistic B for query object A, and $A_{old\_avg}(B)$ represents the prior calculated average.

$$A_{average}(B) = (A_{old\_avg}(B) * w) + (A_{new\_value}(B) * (1 - w)) \tag{3.1}$$

Statistics apply to operators. All statistics are computed using the data from the beginning of execution. This does not create unbounded storage because most of the statistics are averages and thus their storage space does not grow over time.

Selectivity:

$$O(\sigma) = \frac{O(n^o_{total})}{O(n^p_{total})} \tag{3.2}$$

Output Rate:

$$O(\tau_{op}) = \frac{O(n^o_{total})}{O(t'_{total})} \tag{3.3}$$

Average Tuple Processing Time:

$$O(t^C) = \frac{O(n^p_{total})}{O(t'_{total})} \tag{3.4}$$

Operator Throughput:

$$O(t_{op}) = O(n^o_{total}) \tag{3.5}$$

The following statistics apply to entire query plans.

Query plan's throughput:

$$Q(t_q) = \sum_{j=1}^{Q(n^r)} O_j(t_{op}) \tag{3.6}$$

The throughput of a query plan is the sum of the throughput of each root operator.

Output rate:

$$Q(\tau_q) = Q(t_q)/t' \tag{3.7}$$

The output rate of the query plan, $Q(\tau_q)$, can be thought of as a normalized throughput for the query plan for an average time unit. It is equal to the query plan's throughput divided by a unit of time. Typically the denominator used is the overall length that the query has been running, but sometimes we are interested in the output rate over the last x time units.

Total number of tuples in queues:

$$Q(q_{total}) = \sum_{j=0}^{Q}(n)O_j(n^i) \tag{3.8}$$

Average age of tuples:

$$T(t_{age}) = \sum_{j=0}^{T} T_j(t'_a) \tag{3.9}$$

3.3.2 Adding New statistics

The statistics gatherer is flexible and supports the addition of user-defined statistics about operators, tuples, and query plans. The statistics can be defined before the system has started (during linking) if the new statistic requires a special computation that is not presently supported by the system. If the statistics can reuse existing calculations, then the user may define statistics during the initialization phase. The user needs to specify where the calculation is performed, which type of query object(s) support this statistic (or all of them), and the frequency the statistics should be calculated - either every time statistics are updated or along with the rest of the periodically updated statistics.

Chapter 4

Scheduling

4.1 General Issues

Raindrop uses several scheduling algorithms for execution scheduling of query operators. The Execution Engine will ask a scheduler to choose the next operator to run and to determine its workload. After the operator is run, the controller may decide to choose another scheduling algorithm if it deems the current algorithm is not meeting the user's QoS requirements for execution behavior. The heuristics for deciding which algorithm to choose are presented in Section 5.3.

A scheduling algorithm is responsible for two tasks: choosing the operator to run next and assigning a workload to that operator. The next operator decision depends on the algorithm itself while the workload assignment is often fixed regardless of the scheduler. In Raindrop, the workload assignment is controlled by two administrator-defined parameters. The first parameter, $RATIO$, is the ratio of tuples that an operator should dequeue relative to the total number tuples available. Currently this ratio is fixed for each strategy, but future work could adapt this depending on statistics. The second parameter, $THRESHOLD$, aids in calculating how much work to assign to an operator. It aims to reduce the chances that an operator is underutilized by setting a limit for when to use the $RATIO$ and when to use the total number of tuples available. Figure 4.1 illustrates the intuition of this parameter.

```
N = the number of tuples in operator O’s input queue
A = N x RATIO
if A > THRESHOLD
 then O dequeues A tuples.
else
 then O dequeues N tuples
```

Figure 4.1: Pseudo code for determining operator workload

Without this threshold, if Op has 50 tuples in its input queues and the ratio is .1, Op first runs for 5 tuples, then 5 (45 x .1), then 4, and so forth. Op will have to be run many times over to work with all 50 tuples. If the $THRESHOLD$ is set to 50, which experimentally was shown to yield good performance for each operator, O_p only has to run once. Setting this too high could decrease performance because an operator may be overwhelmed with tuples, but setting it too low could result in a lower performance as well because an operator may not be fully utilized. In our experiments (Section 6), we found that setting the RATIO to 30% yielded the best performance.

Every scheduling algorithm has its advantages and its flaws. Certain algorithms are particularly good at keeping memory utilization to a minimum [3]. Other algorithms are excellent at quickly producing some result set for the end application user [22]. The adaptive technique utilized in this paper focuses on selecting a particular scheduling algorithm when its advantages can be exploited. There are times when one single algorithm is the best to use and is more effective than switching between possibly several algorithms. Our aim is that in this situation the adaptive technique would select this algorithm as often as possible. However, supported by our experimental results, we will show that many queries based on varying QoS requirements do not have one particular scheduling strategy that works the best. In fact in such cases, it is better to utilize the strengths of several algorithms to produce the best overall quality of service possible. For review purposes, we now describe several scheduling strategies employed by our adaptive scheduling framework, and explain their advantages and disadvantages.

4.2 Round Robin

Round Robin (RR) is perhaps the most basic scheduling algorithm. It works by placing all runnable operators in a circular queue and allocating a fixed time slice to each. Round Robin's best quality is the avoidance of starvation. An operator is guaranteed to be scheduled within a fixed period of time. In fact, as long as an operator always has work to do, no operator will be run more times than any other. However, Round Robin does not adapt at all to changing stream conditions. It also does not consider many possibly important factors, such as an operator's performance relative to other operators, size of the input queues, or the selectivity. Therefore, the intermediate queue sizes can grow rapidly because RR may spend its time running other operators that have less work to do or are less favorable for other reasons.

4.3 FIFO

FIFO (first in first out) chooses a leaf operator to execute and attempts to push its tuples through the system as far as possible. FIFO typically yields a consistent throughput, because it tries to execute older tuples until completion before it considers newly arrived tuples. But it has the same drawbacks as Round Robin - no adaptiveness and no consideration of operator properties.

4.4 Greedy

Greedy scheduling assigns a priority to each operator and always tries to run the operator with the highest priority. If the operator with the highest priority has no work to do (i.e. empty input queues), Greedy will choose the next highest priority. The priority, calculated dynamically, is shown in Equation 4.1 and was originally shown in [3]. The operator's priority and corresponding selectivity and tuple cost are recalculated periodically during execution to insure that the information is not stale.

Greedy eliminates some of the drawbacks of Round Robin and PTT because it considers the cost of each operator before choosing which operator to run. However, it is prone to starvation. If the high priority operator, O, is proceeded by lower priority operators, O will eventually starve for input because its children operators may not be run as often. On the other hand, if that same operator O were connected to the streams instead, it will almost always have work and thus the other operators in the system would never be run often. The output queues of O would grow indefinitely.

Throughput and average delay may suffer with Greedy because the strategy does not take into account where in the execution plan the operator lies. Thus, it does not give higher priority to those operators that will output results to the end user. These metrics will suffer more if the higher priority operators are lower in the plan and less if those operators are near the top of the plan.

Greedy will slowly adapt to bursty streams. To see this , first assume that an operator's selectivity is relatively constant over time. Some operators may be able to work more efficiently with a larger amount of tuples (i.e some hash-based joins because the fixed cost of hashing each tuple can be spread out over more possible matches) and thus their average tuple processing time would increase or decrease during periods of burstiness. Therefore, the priority would fluctuate accordingly. However, it is unclear if this fluctuation would be immediate or if there would be some lag time before these changes are propagated throughout the system and the scheduler adapts. The speed of the propagation depends on how often the priorities are updated. Updating too frequently will keep the priorities current, but at the expense of system overhead, especially with operators whose priorities are unlikely to change much over time. For example, the cost of performing a traditional RDB Project operator will likely remain constant.

The Greedy priority is calculated in Equation 4.1. Equation 4.1 will produce higher (better) values when the operator has a low selectivity (1 - $\sigma \rightarrow$ 1) and / or has a low cost to process 1 tuple.

$$O(\rho) = \frac{1 - O(\sigma)}{O(t^C)} \tag{4.1}$$

4.5 Most Tuples in Queue

The Most Tuples in Queue (MTIQ) scheduler is a greedy algorithm that assigns a priority to each operator equivalent to the number of the tuples in its input queues. MTIQ is a simplified batch scheduler similar to [5]. Batch schedulers work under the assumption that the average tuple processing cost can be reduced if an operator works on more tuples at a time. Operators typically have a start-up cost associated with their execution and the batch scheduler can amortize this cost over a larger group of tuples. Round Robin and FIFO do not have this property and thus those algorithms tend to under-utilize operators.

Second, MTIQ tends to have a bursty output pattern. Typically it takes a relatively long period of time for enough tuples to make it through the system such that the root operator has more work to do than the operators below it. However, when the root operator runs, it then will output a large block of tuples. Some tuples will experience little delay while others will be enqueued for long periods of time, but on average, the mean delay will not be much worse than the other algorithms.

The most obvious advantage is that MTIQ works well at minimizing memory consumption. By running the operator with the most tuples enqueued, the algorithm will have a better chance than the previous algorithms at ensuring that no queue will grow unbounded. If the data arrives faster than MTIQ can process it, then that queue will grow infinite in size.

4.6 Chain

Chain [3] is a recently proposed variation of Greedy Scheduling. Conceptually, each operator is assigned a priority that is based on selectivity, tuple processing cost, and the priorities of each operator preceeding it in the query plan. This inductive priority is calculated by plotting the query plan on what is called a progress chart. The horizontal axis of the chart represents time and the vertical axis represents the number of tuples in the system at each operator at the given time. Each point on the chart corresponds to the time that an operator takes to process its tuples. The points are connected and the priority is calculated based on the slope of the line between points. Chain schedules the operator who has tuples that lie on the steepest slope of the progress chart. The intuition here is that several operators will be ”chained” together in such a manner that when the operators are selected they will remove the largest number of tuples from the query plan in the shortest amount of time.

Chain may suffer from starvation and poor response time during times of burst [3], but was shown, using experiment results, to be a near optimal strategy for keeping queue sizes to a minimum.

The relative strengths and weaknesses of the algorithms described above can be found in Table 4.6.

Table 4.1: Comparison of Scheduling algorithms.

Algorithm	Advantages	Disadvantages
Round Robin	- Guarantees that every operator is scheduled.	- Does not select an operator because it is ”best”, but because it is ”next”. - Over time, poor output rate and memory utilization.
FIFO	- Schedules operators with the same frequency. - Outputs tuples sooner and at a constant rate.	- Queue sizes grow quickly. - Output rate is low. - Does not utilize operators as fully as Greedy.
MTIQ	- Queue sizes are smaller. - Higher output rate. - More fully utilizes operators.	- Bursty output pattern. - Tuples spend a long time in the system.
Chain	- Keeps queue sizes small. - Chains logically scheduled operators together.	- Chains need to be periodically recalculated with latest statistical data. - Suffers from poor average processing time and can hinder output rate.

Chapter 5

Adaptive Scheduling

5.1 Quality of Service Requirements

Raindrop allows for the system administrator to specify the desired execution behavior as a composition of several possible goals. A QoS requirement consists of three parts, the statistic, the quantifier, and the weight. The statistic corresponds to which requirement the user wishes to control. Currently, Raindrop supports the following requirements:

Output Rate: how many tuples does the query plan output to the end user application per time unit.

Intermediate Queue Size: how many tuples are stored in intermediate queues

Tuple Delay: what is the delay from the time tuples enter the system until they are outputted.

The quantifier, either *maximize* or *minimize*, specifies what the system administrator wants to do with this preference. Because the system supports any metric in the QoS specification, Raindrop needs information regarding whether the metric should be maximized or minimized. The weight is the relative importance of each requirement and the sum of all the weights is equivalent to 1. The weights will vary from application to application meaning it is imperative that the administrator assigns weights that reflect the individual application's needs. Table 5.1 shows an example QoS specification. Here, the administrator has specified that the system should give highest priority to the minimizing queue size and the maximizing throughput is assigned priority #2.

Statistic	Quantifier	Weight
Input Queue Size	minimize	0.75
Throughput	maximize	0.25

Table 5.1: An example preference

Here we assume that all application queries share the same quality of service requirements. That is, assume the system administrator will specify a single quality of service requirement that applies to all registered application queries, i.e., for the one global query plan in the system. The administrator has the ability to change the requirements during the course of execution, but that would affect all queries. Using this assumption allows us to ignore issues related to conflicting QoS specifications for multiple application queries.

QoS requirements are a key concept in Raindrop. They will guide the adaptive execution by encoding the goal that the system is supposed to pursue. Without these preferences, the system will not have any benchmarks to determine how well or poorly it is performing. It is important to note that the requirements specify the desired behavior in relative terms. That is, the administrator does not specify an absolute performance goal (i.e., achieve an output rate of X tuples / sec or have no more than Y tuples in the queues at once), but rather specifies that they want the system to maximize output rate or minimize queue size. Absolute requirements are too dependent on data arrival patterns and so on thus, may not be achievable without drastic measures such as dropping tuples from the load and thus affecting the actual answer [16][5].

Aurora's [5] definition of quality of service requirement is similar to what is used here, although the terminology is slightly different. Generally, QoS relates to a desired execution behavior in Aurora and it specifies an absolute requirement. In this work, QoS requirements also relate to the goals specified by the administrator (as in Aurora), but this work does not try to improve the QoS as execution progress. Rather, Raindrop tries to match system performance to what is specified in the QoS requirement.

The system also provides QoS requirement templates. Each template contains one or more requirements and each has been tuned to best achieve a certain goal. This allows the administrator to more easily specify the desired execution behavior without having to worry about the lower level details including making guesses about relative weights. If no service requirements are provided, the system will choose a default suite that will give equal weight to minimizing queue sizes and delay and maximizing output rate.

5.2 Algorithm's Score Computation

During execution, the Execution Engine will update the statistics that are related to the QoS requirements. Once these statistics have been updated, the system needs to decide how well the previous scheduler, S_{old}, has performed, compare this performance to the other scheduling algorithms and then determine how to continue execution. To accomplish this, the system calculates the mean (μ_H) and the spread of the values ($max_H - min_H$) of each of the statistics specified in the service preferences for the historical category, H. Next, using the statistics from S_{old} the mean μ_S of each of the statistics is calculated. Finally, each μ_S is normalized according to the formula in Equation 5.1. This normalizes each value in the -0.5 to 0.5 range. 0.5 is added to the z_i to insure it is always between 0 and 1.

$$z_i = \frac{(\mu_S - \mu_H)}{max_H - min_H} * decay^{time} + 0.5 \tag{5.1}$$

A $decay$ parameter is used to exponentially decay old and out of date data to reflect the unreliability of the score of algorithms that have not run for long periods of time. The decay is calculated by raising the $decay$ parameter $(0 < decay < 1)$ to a given $time$. $time$ can be expressed in units of time since μ_S was updated *or* it can be in expressed as the number of times that other algorithms have been chosen since this strategy was chosen. Both approaches have their merits and the choice depends heavily on several other factors such as the frequency that the scores are computed. If the scores are frequently recalculated, using the time since updated makes more sense than using the number of algorithms since last chosen because the number of times will decay the score too quickly and an algorithm's score will rapidly approach zero.

Next we compute a scheduler's overall score, $scheduler\_score$ for the algorithm we just used, using a weighed sum, using the statistics score from Section 5.1 and the weights of each requirement as given by the user. Equation 5.2 shows how this score is computed. In this equation, each of the normalized values produced by Equation 5.1 are multiplied by the corresponding weight w_i. The quantifier, from the preference, is used to determine if we wish to maximize or minimize z_i. If the quantifier equals maximize, z_i = z_i. If the quantifier is to minimize, then z_i = 1 - z_i.

$$scheduler\_score = \sum_{i=0}^{I} (z_i)(w_i) \tag{5.2}$$

Finally by comparing S_{old}'s $scheduler\_score$ with the scores for all of the other algorithms (that have run so far), the system can decide how well or poorly the previous scheduler performed. The system then determines which algorithm to choose next, Section 5.3 describes this next decision making process.

Analysis of Behavior of Equation 5.2 Equation 5.2 gives a higher score to QoS requirements that have a high weight and a high z value from Equation 5.1. Equation 5.1 maps each scheduler's score for each statistic to a value between 0 and 1 and allows for a comparison among different statistics. The weighed sum from Equation 5.2 will also yield a value between 0 and 1 for each scheduler. In our case, we wanted to map a whole data set (statistics for a scheduler) into a single value that could be compared to another set.

The score assigned to an algorithm is not based solely on the previous time that it was used, but rather is an aggregate over time. While the performance of an algorithm is largely coupled to the behavior of the data, over time the score of the algorithm should reflect its true potential.

There are several important properties to note regarding Equation 5.2. First, the statistics in the H category are a union of the statistics for each individual scheduling algorithm. That is, because H (historical category) contains data from every category, and thus every time that the statistics for S_{old} are updated, those same statistics are updated for H with the same values.

Second at the beginning of execution, Equation 5.2 is prone to outliers and initially will assign misleading scores. Similarly, if S_i has been used more frequently than any other scheduler or if it has been run for a long time, the mean values for statistics for S_i will be similar to the mean in H. This is due to the normalization technique chosen in the equation. The min-max technique captures the behavior of S_i relative to H by noting the difference between their means for the given statistic. Every time an algorithm is run, the mean of H is drawn closer (skewed) towards that algorithm. By continuously running the same algorithm, the mean of S_i will end up on top of the mean of H any relative performance information is lost. Thus Equation 5.1 will produce a value closer to 0, even if S_i is performing well or poorly.

To overcome this problem, after each algorithm has been run, we do not update the score for S_{old}, but rather the z score for every algorithm that did not run is computed. If S_i ran for sufficiently long, then this will effectively compare the other algorithm's previous performance to that of the current algorithm's. Thus we can directly compare how well each algorithm is performing. The score used for the current algorithm, S_{old}, comes from the last time S_{old} was run. To account for stale data for S_{old}, we decay that score every time using the $decay$ parameter that was previously discussed.

Several items that must be considered 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 did not run. Therefore, at the beginning of execution, we want to allow some degree of exploration on the part of the adapter. However, if the query is relatively short-lived, i.e. the application only issues the query for a short amount of time, allowing too much exploration will not allow the adapter to do its job.

2. Not switching algorithms periodically during execution (i.e., greedily choosing the next algorithm to run) could result in a poorly performing algorithm being run more often than a potentially better performing one. Hence, we have to periodically explore other strategies.

3. Switching algorithms too frequently could cause one algorithm to impact the next and skew the latter's results. For example, using Chain as described in Section 4 could cause a glut of tuples at the input queues of lower priority operators. If MTIQ were to be run, its throughput would initially be artificially inflated because of the way Chain operated on the tuples. If we switched to another algorithm soon after, the z-score from Equation 5.1 for throughput would be skewed.

More generally, when a new algorithm is chosen, it should be used for enough time such that its behavior is not significantly controlled by the previous algorithm.

5.3 Adapting Scheduling Strategy

After the above computation has been completed, the system needs to decide if the current scheduling algorithm performed well enough that it should be used again or if better performance could be achieved through changing algorithms. Considering the two points above, initially running each algorithm in a round robin fashion is the fairest way to start the adaptive scheduling.

In an effort to consider all scheduling strategies while still probabilistically choosing the best fit we adopted the Roulette Wheel strategy [13] from Genetic Algorithms research. This strategy assigns to each algorithm a slice of a circular "roulette wheel" with the size of the slice being proportional to the individual's score that was calculated by Equation 5.2. This strategy is also referred to as "fitness proportion selection". Then the wheel will be spun once and the algorithm under the wheel's marker is selected to run next. This strategy may initially choose bad scheduling algorithms, but over time, should fairly choose the correct algorithm. The strategy also allows for a fair amount of exploration and it prevents one algorithm from dominating.

The adaptive strategy will first run each algorithm once, for approximately one second, in a round robin fashion. The first algorithm run will be run once more at the end to account of the initial start up time for the query. Once this process has completed, the roulette wheel will be used for the duration of the query execution.

Chapter 6

Experiments

6.1 Experiment Setup

This section will describe the experiments conducted in order to compare the performance of a single scheduling algorithm to our adaptive solution exploiting multiple algorithms. The first phase of experiments establishes a performance baseline for a single algorithm during bursty streams, where bursty streams are defined as streams whose arrival rate spikes to an order of magnitude above the average arrival rate. The second phase will then compare the performance of the adaptive strategy to the single algorithms during periods of burst. Four scheduling algorithms described in Section 4 were used- Round Robin, Chain, FIFO, and MTIQ.

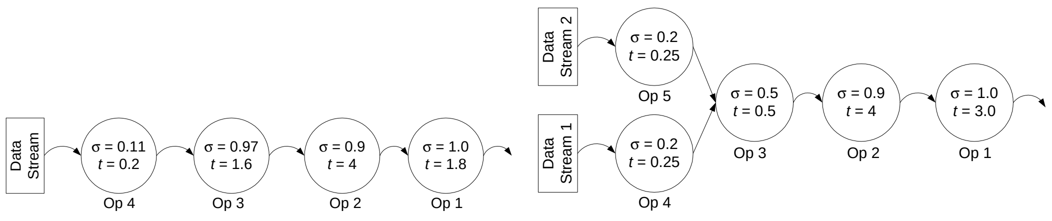
If one algorithm can meet a given service preference on its own then switching between that algorithm and other sub-optimal algorithms will not yield better performance. Therefore, we want to show that the adaptive strategy picks that single algorithm most often and will perform nearly, if not exactly, the same as that one strategy does. The more interesting experimental case is when a clear tradeoff between algorithms exists. We also want to show clearly that, for each quality of service composition, the adaptive strategy selection performs better than any of the candidate algorithms The optimal case would be for the adaptive strategy to meet the maximum value possible for each preference.

We used the three metrics in our experiments- mean output rate, mean delay, and mean memory size that were discussed in Section 5.1. To reiterate, the memory size is defined as the number of tuples in all queues in the system at a given time unit. The delay is equal to the time a tuple spends in the system (time waiting in queues + processing time) and the output rate is the number of tuples a query plan produces per time unit. These requirements were selected for experimentation no one scheduling algorithm can optimize for all of them at the same time.

For each experiment with two preferences, the preferences were assigned weights of 100-0, 70-30, 50-50, 30-70, and 0-100 where the first number is the weight assigned to the first and the second to the second. When all three preferences were used in a single experiment, we used equal weights of 33-33-34.

The single QoS experiments were run for 30 seconds while the multiple QoS experiments ran for 300. The influence of any startup costs was minimized by running the first algorithm for five seconds before beginning the round robin proceedure described in Section 5.3. The charts shown below do not include any statistics gatherered during the startup and exploratory phases. We evaluated the adaptive scoring function every two seconds after running each algorithm for an initial period. The statistics described in Section 3.3.1 were updated every second and 0.875 was used for w in the weighed average equation, Equation 3.1. From Section 4.1, the $THRESHOLD$ is set to 50 and the $RATIO$ is set to 30%.

Two query plans were used in the experiments. The first query plan is a simple query plan with four filter operators. The second query plan utilizes a window join operator [7] with a window of 200ms. That is, any tuples that are received within 200ms of each other are evaluated in the join predicate of the operator. The query plans are listed in



(a) Query Plan 1 (b) Query Plan 2

Figure 6.1: Query Plans used in Experimentation.

Figure 6.1 with selectivity (σ) and average tuple processing time (t). For these particular experiments, the selectivity of O_1 is irrelevant because its output is piped to an end user application and not another operator. Hence we set its selectivity to 1. Setting this value lower would only serve to reduce the query plan's throughput, T_{plan}, by a constant percentage.

The Internet Traffic Archive [1] was used as the data set. This data simulates the contents of real streaming data. The arrival rates of the streams were adjusted to have a random pattern using Poisson distribution. The streams were steady at times, and rather bursty (with a mean arrival time that was approximately two times that of the average rate during non-bursty periods) at other times, due to the unpredictability of users' requests. The stream rates were adjusted using custom built Stream Sources that would generate data with different Poisson means every 5 seconds. This was done to show that under both steady and bursty conditions, the adaptive framework could respond with good experimental results.

6.2 Evaluation of Scheduling

Figure 6.2 shows the performance of the four algorithms while monitoring two different quality of service requirements, the number of tuples in memory, and the average tuple delay. As anticipated, Chain and MTIQ performed best when it comes to minimizing

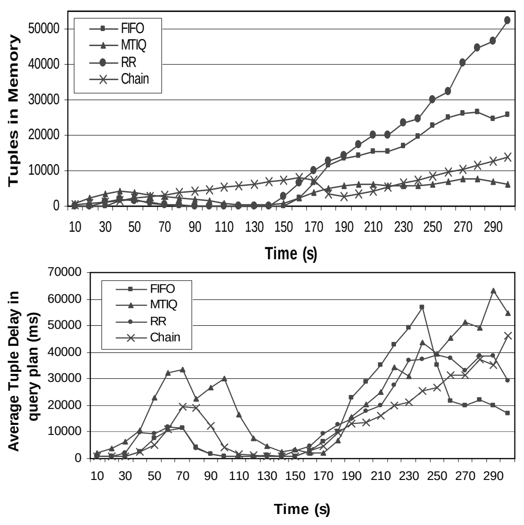


Figure 6.2: Performance of scheduling algorithms with Query Plan 2.

memory use. As discussed in Section 4.1 Chain processes operators that remove the largest number of tuples the most quickly. MTIQ processes operators that have the largest queue in the query plan. Thus it is no surprise that these two algorithms are excellent at reducing memory usage.

However we see very different results when observing how well the algorithms perform when it comes to the average tuple delay. MTIQ and Chain end up being the two worst performers by the end of execution. FIFO, which was only mediocre under the memory requirement, actually does quite well keeping the average tuple delay to a minimum. Overall we observe from Figure 6.2 that no one algorithm has a clear advantage. MTIQ and Chain compete for the best results in memory consumption, while RR, Chain and FIFO compete for the best results for average tuple delay. Therefore, by combining all of the algorithms, we should be able to outperform a single strategy for a given requirement.

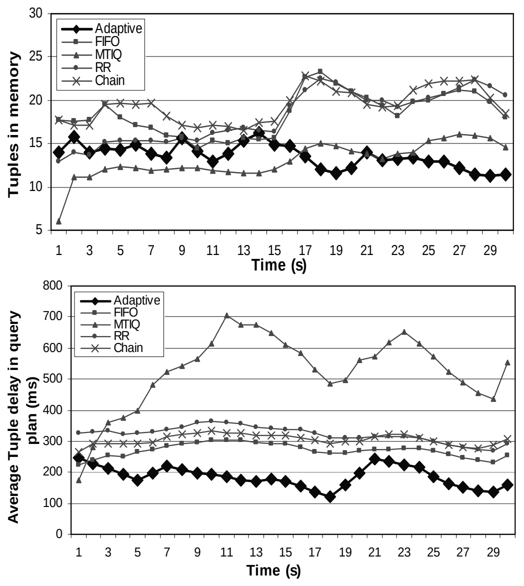


Figure 6.3: Optimizing query execution with one QoS requirement. Figure a, top, memory usage. Figure b, bottom, average delay.

6.3 Direct Competition with Published Scheduling Algorithms

The next experiment used a QoS specification with only one requirement. This was done to demonstrate that the adaptive framework can pick an optimal scheduling algorithm even for only one requirement. Figure 6.3(a) shows that the adaptive framework does exceptionally well at selecting algorithms to keep tuples in memory down. In fact, at many times the framework outperforms every single scheduling algorithm in terms of memory.

In Figure 6.3(b) it can be seen that the adaptive framework outperforms all individual scheduling algorithms. It outperformed the other algorithms by leveraging their relative strengths. It was observed that MTIQ can exploit queue buildups caused by FIFO. As FIFO begins execution, a buildup of tuples is created at the leaf operator. Since there is a buildup in tuples at the leaf operator MTIQ is selected (at time t=7) and progresses the tuples through the query plan. FIFO is then selected again (at time t=21) as older tuples

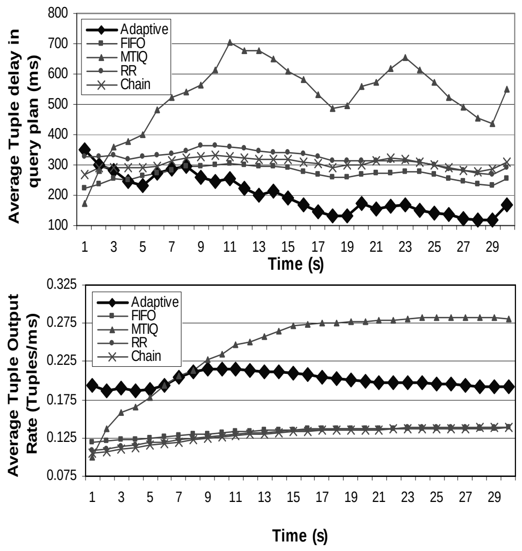


Figure 6.4: Two QoS requirements: 70% minimizing tuple delay, and 30% maximizing output rate (Query Plan 1). Figure a, top, absolute delay. Figure b, bottom, absolute output rate.

were still in the query plan that needed to be processed.

6.4 Reaction to Changing QoS Specifications

For the second set of experiments, the ability of the adaptive framework to react to a QoS specification with two requirements is shown. There are two goals in this set of experiments. First showing that if the importance of a requirement is changed, the framework will acknowledge this and adapt accordingly. Secondly it is important that the framework performs well in both QoS requirements.

Figure 6.4 depicts the results for an experiment for which 70% importance was placed on tuple delay and 30% importance was placed on output rate. Here observe that the adaptive framework outperformed single algorithms with respect to average tuple delay, and performed about average with respect to the average output rate.

Figure 6.5 shows our performance when we adjust the percentage of the weights to 70% focus on maximizing output rate, and 30% focus on minimizing tuple delay. We

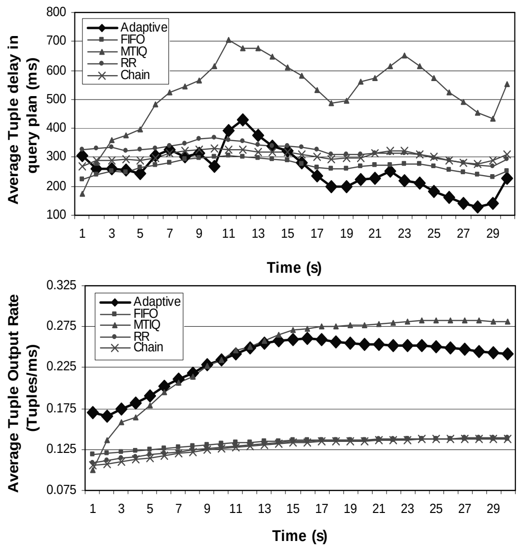


Figure 6.5: Optimizing query execution with two QoS requirements. 30% focus on minimizing tuple delay, and 70% focus on maximizing output rate (Query Plan 1)

can observe that with the change in service requirement, the adaptive framework still does exceptionally well at minimizing tuple delay, but improves significantly at raising the average tuple output rate. This shows that the adaptive framework can adapt accordingly to varying QoS requirements, and also provide significant improvements of single scheduling algorithms.

We will now consider the case of having two equally important QoS requirements. Figure 6.6 shows the performance of the adaptive framework with an equal focus on average output rate and average tuple delay. We make two observations from these charts. First, clearly there is no single optimal scheduling algorithm, as each algorithm has varying performance throughout execution. Second, our adaptive framework is able to outperform all single scheduling algorithms for most of execution. The adaptive algorithm appears to have made better decisions as the execution progressed as evident by the improved memory utilization. The output rate did suffer slightly, however.

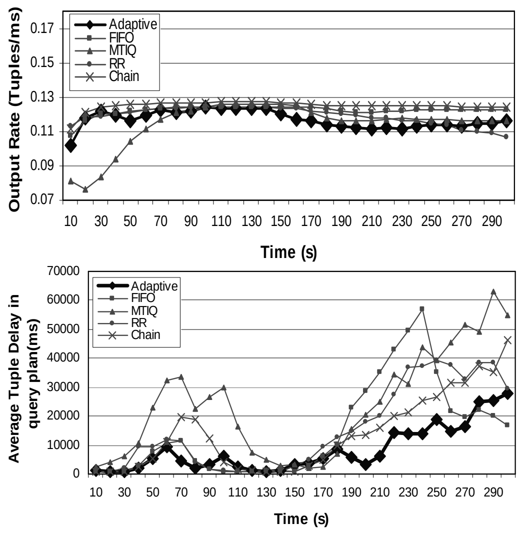


Figure 6.6: Optimizing query execution with two QoS requirements. 50% focus on minimizing tuple delay, and 50% focus on maximizing output rate (Query Plan 2)

6.5 Adaptive Framework with Multi-Facetted QoS Specifications

In our final set of experiments we compared the performance of the adaptive framework against the single scheduling algorithms with a QoS specification of three requirements. In this example each requirement (average tuple delay, average output rate, and average tuples in memory) was each given equal weight.

In Figure 6.7 we can see that the adaptive framework again performs well under all three QoS requirements. The biggest improvements are average tuple delay and the number of tuples in memory, where the adaptive framework significantly improves upon all but the best single scheduling algorithms.

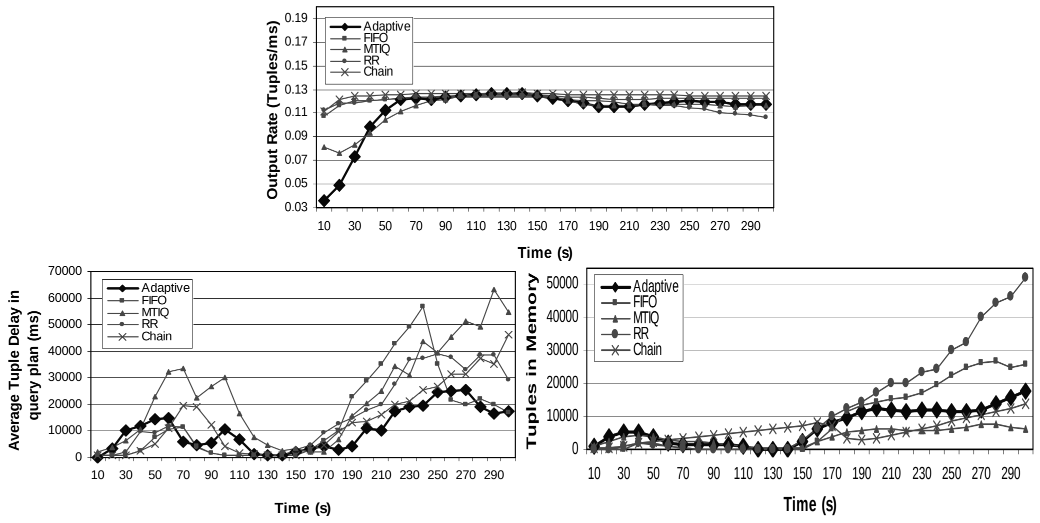


Figure 6.7: Optimizing query execution with three equal QoS requirements (Query Plan 2)

Chapter 7

Conclusions

7.1 Summary

This thesis addressed the issues relating to creating an adaptive execution strategy for the execution of a continuous query over streaming data. The proposed adaptive strategy chooses the next scheduling algorithm to utilize among several candidate algorithms based on their performance thus far relative to the user's quality of service requirements. This leverages prior research in artificial intelligence in the area of multi-agent systems by utilizing ideas in how to combine several candidate solutions into one. This performance is captured by normalizing the statistics for each algorithm and calculating how well each algorithm did compared to the algorithm that was just used. Then, the next algorithm is chosen by spinning a roulette wheel where each candidate is chosen with a probability equal to its performance score. This made use of techniques from genetic algorithms.

Current continuous query systems rely on a single scheduling algorithm. As a consequence, they are restricted in the QoS specs that they may meet by controlling the operator scheduling alone. Thus, the goal of this adaptive algorithm is to leverage the strengths of each of the candidate algorithms against one another to create a solution that outperforms each single strategy for the given QoS.

Our experimental study illustrated that the adaptive algorithm was able to outperform the four candidate algorithms for some, but not all, QoS requirements. The study evaluated the Most Tuples in Queue and Chain batch schedulers and the First in First Out and Round Robin fair-use schedulers against the adaptive strategy for various preferences using a Poisson based arrival patterns. The QoS combination of weights aims for minimizing memory usage and result delay and for maximizing output rate. The algorithm successfully leveraged the consistent-performing nature of the fair use algorithms with the fluctuating behavior of the batch algorithms. The adaptive algorithm was able to successfully identify when one candidate's performance was decreasing (due to the rate of newly arriving tuples) and switched to the other to keep overall performance at an acceptable level. The experimental study also showed that the adaptive algorithm's overhead was comparable to either of the single strategies, even in the case of more complex queries.

We also showed that the user's service preferences do in fact have an effect on the behavior of the adaptive algorithm. In our study, the adaptive algorithm that was optimized for a given metric outperformed the other adaptive algorithm that was optimized for another metric. This is an important conclusion because it shows that the adaptive algorithm behaves intelligently and does not win simply because it combines the other algorithms.

Given the presence of a single algorithm that optimally met the requirement, the adaptive strategy chose that algorithm more than the other. When the adaptive algorithm periodically switched to one of the other candidates for exploratory purposes, the adaptive's overall performance decreased. Thus, the adaptive was never able to outperform that single strategy.

7.2 Contributions

This thesis contributed to Continuous Query Systems, particularly query processing, in the following ways:

- Studied the performance of a variety of scheduling algorithms in a real Continuous Query System, to determine the pros and cons of algorithms under varying QoS requirements, data stream arrival rates, and query plans.
- Designed an adaptive framework that has the ability to observe the behavior of the continuous query system and pick scheduling algorithms that probabilistically have the best chance to fulfill a given set of QoS requirements.
- Built a continuous query system from the ground up, that we used as a test bed to study how our adaptive framework can aid in the processing of a query.
- Performed an experimental study to support our claim that in fact, we can leverage the strengths of several existing scheduling algorithms to improve the overall performance of a continuous query system given a set of QoS requirements.

7.3 Future Work

There are many future topics to investigate based on the preliminary results produced by this thesis. The first direction involves augmenting the experimental study with additional data distributions and more complex query plans. Another direction involves tweaking the various experiment parameters. Further testing to find the optimal values for the weight to give to old values for weighted average, workload ratio, and frequency of updating statistics should result in improved performance. The adaptive strategy can be further tweaked by altering the data decay and algorithm switch parameters or by running multiple operators at the same time.

Another direction involves investigating incorporating alternate adaptive techniques such as those used in [5][9]. Combining these techniques with the adaptive scheduling strategy yields an interesting research question - could we find a formula to weigh the benefits of one technique over the other and always choose the adaptive technique that will meet the user's quality of service best.

Bibliography

[1] I. T. Archive. http://www.acm.org/sigcomm/ita/, 2003.

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

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

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002)*, pages 1–16, 2002.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stone-Braker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, 2002.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *SIGMOD*, pages 379–390, 2000.

[7] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, September 2003.

[8] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[9] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proceedings of SIGMOD*, pages 299–310, 1999.

[10] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 106–117. ACM Press, 1998.

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

[12] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD Conference 2002*, 2002.

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

[14] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[15] J. F. Naughton, D. J. DeWitt, D. Maier, et al. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[16] R. Motwani, J. Widom and A. Arasu et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proceedings of CIDR*, pages 245–256, 2003.

[17] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *I Proceedings of USENIX, 8*, pages 13–24, 1998.

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

[19] T. Urhan and M. Franklin. XJoin: A Reactively Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2), 2000.

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

[21] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proc. of SIGMOD 1998*, pages 130–141, 1998.

[22] S. Viglas and J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Proceedings of SIGMOD*, pages 37–48, 2002.