Efficient Scheduling of Heterogeneous Continuous Queries *

Mohamed A. Sharaf   Panos K. Chrysanthis   Alexandros Labrinidis   Kirk Pruhs

Advanced Data Management Technologies Laboratory
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA

{msharaf, panos, labrinid, kirk}@cs.pitt.edu

ABSTRACT

Data Stream Management Systems (DSMS) typically host multiple Continuous Queries (CQ) that process streams of data. In this paper, we examine the problem of how to schedule CQs in a DSMS to optimize for average QoS. We show that unlike standard on-line systems, scheduling policies in DSMSs that optimize for average response time will be different than policies that optimize for average slowdown which is more appropriate metric to use in the presence of a heterogeneous workload. We also propose a hybrid scheduling policy based on slowdown that strikes a fine balance between performance and fairness. We further discuss how our policies can be efficiently implemented and extended to exploit sharing in optimized multi-query plans and multi-stream CQs. Finally, we experimentally show using real data that our policies outperform currently used ones.

1. INTRODUCTION

The growing need for *monitoring applications* has led to a new data processing paradigm and created a new generation of data processing systems, called *Data Stream Management Systems* (DSMSs) that can support *continuous queries* (CQ). In such systems, each monitoring application registers a set of CQs that continuously process continuous data streams looking for data that represent events of interest to the end-user.

Currently, we are developing a DSMS, called *AQSIOS*, that can help support monitoring applications such as the real-time detection of disease outbreaks, tracking the stock market, environmental monitoring via sensor networks, and personalized and customized Web pages. One of the main goals in the design of *AQSIOS* is the development of a scheduling policy that optimizes *Quality of Service* (QoS).

This goal is complicated by the fact that the scheduling policy must take into account that the CQs are heterogeneous, i.e., they may have different time complexities (the amount of processing required to find if input data represents an event), and different productivity or selectivity (the number of events detected by the CQ). For example, consider two CQs, `GOOGLE` and `ANALYSIS` on streams of stock market data. `GOOGLE` is a simple query that asks the DSMS to be notified when there is a stock quote for *GOOGLE*. `ANALYSIS` is a complex query that asks the application to provide some specific technical analysis for any new stock price. Obviously, `GOOGLE` has low cost and it detects less events, whereas `ANALYSIS` has high cost and it detects more events.

The mostly commonly used QoS metric in the literature is *average response time*. In [19], we showed that if the objective is to optimize the response time, then the "right" strategy is to schedule CQs according to their *output rate*. Specifically, in [19] we presented a new scheduling policy called *Highest Rate (HR)*. *HR* generalizes the *Rate-based policy (RB)* [23] for scheduling operators in multiple CQs as opposed to *RB* that has been proposed for scheduling operators within a single query. Under *HR*, the priority of a query is set to its output rate where the output rate of the query is the ratio between its expected selectivity and its expected cost.

However, there are some well known disadvantages to the average response time metric when the workload is heterogeneous. In the above example, the user that issued the `ANALYSIS` query likely knows that it is a complex query, and is expecting a higher response time than the user that issued the `GOOGLE` query. A metric that captures this phenomenon is *average slowdown*. The slowdown of a job is the response time of the job to the ideal processing time of the job [17]. So, for example, if each job had slowdown 1.1, then each user would experience a 10% delay due to queuing (although the responses could be very different).

Interestingly, in most on-line systems (e.g., Web servers), *Shortest-Remaining-Processing-Time (SRPT)* is one policy that is optimal for average response time and near optimal for average slowdown [17]. A surprising discovery of this paper is that this is not the case with the *HR* policy that optimizes average response time of CQs. In general, *HR* will not optimize average slowdown because of the "probabilistic" nature of CQs where the selectivity might not equal to 1. In this paper, we argue that if the objective is to optimize average slowdown then the "right" scheduling strategy is to set the priority of a query to the ratio of its selectivity over the product of its expected cost and its ideal total processing cost.

The average slowdown provided by the DSMS captures the average-case performance of the system. However, improving the average-case performance usually comes at the expense of unfairness toward certain classes of queries that might experience *starvation*. Starvation is typically captured by measuring the *maximum slow-*

*This work was partially supported by NSF IIS-0534531. The first author was also supported in part by the Andrew Mellon Predoctoral Fellowship.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

down of the system [8]. That is, the perceived worst-case performance.

Starvation is an unacceptable behavior in a DSMS that supports monitoring applications where all kinds of events are equally important. Hence, it is important to balance the trade-off between the average-case and worst-case performances of the DSMS. Toward this, we propose a hybrid scheduling policy that optimizes the ℓ_2 norm of slowdowns [7]. As such, it is able to strike a fine balance between the average- and worst-case performances and hence it avoids starvation and exhibits higher degree of *fairness*.

In addition to new scheduling policies, we consider two special features that are unique to CQs and should be exploited by the query scheduler. First, we address the scheduling of *multi-stream queries with time-based sliding window join operators*. We formulate the definition of slowdown for composite tuples produced by join operators and extend our proposed scheduling policies to handle such multi-stream queries. Second, we consider the scheduling of *multiple queries with shared operators* where we show that a proper setting of the priority of shared operators significantly improves the system performance.

Contributions The contributions of this paper can be summarized as follows:

1. We propose a policy for scheduling multiple CQs that maximizes the average-case performance of a DSMS.

2. We propose a hybrid policy that strikes a fine balance between the average- and worst-case performances.

3. We consider two issues that are very particular to DSMSs. Namely, we propose: (1) extending our proposed policies to handle multi-stream continuous queries; and (2) exploiting sharing in optimizing multi-query plans.

4. To ensure that our proposed hybrid policy can be efficiently realized in AQSIOS, we propose a low-overhead implementation which uses clustering in addition to efficient search pruning techniques from [3, 12].

Our extensive experimental evaluation using real and synthetic data shows the significant gains provided by our proposed policies under different QoS measures compared to existing scheduling policies in DSMSs.

Road Map Section 2 provides the system model. Section 3 and 4 define our QoS metrics and presents our proposed scheduling policies. Section 5 focuses on multi-stream queries. In Section 6 and 7, we discuss implementation details and extend our work to consider queries with shared operators. Sections 8 and 9 discusses our simulation testbed and our experimental results. Section 10 surveys related work.

2. SYSTEM MODEL

In a DSMS, users register continuous queries that are executed as new data arrives. Data arrives in the form of continuous streams from different data sources. where the arrival of new data is similar to an *insertion* operation in traditional database systems. A DSMS is typically connected to different data sources and a single stream might feed more than one query.

A continuous query evaluation plan can be conceptualized as a data flow tree [10, 5], where the nodes are operators that process tuples and edges represent the flow of tuples from one operator to another (Figure 1). An edge from operator O_x to operator O_y means that the output of O_x is an input to O_y. Each operator is associated with a *queue* where input tuples are buffered until they are processed.

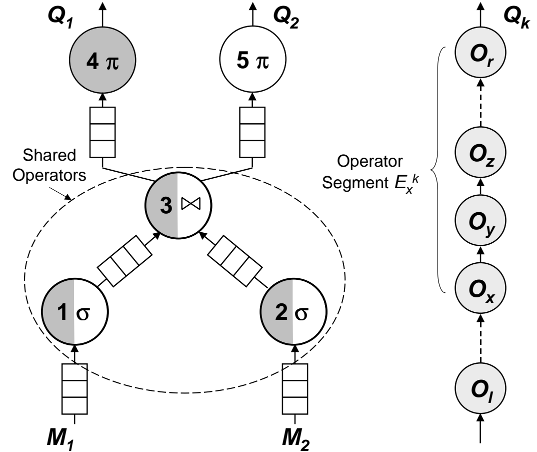

Figure 1: Continuous Queries Plans

Multiple queries with common sub-expressions are usually merged together to eliminate the repetition of similar operations [18]. For example, Figure 1 shows the global plan for two queries Q_1 and Q_2. Both queries operate on data streams M_1 and M_2 and they share the common sub-expression represented by operators O_1, O_2 and O_3.

A *single-stream query* Q_k has a single *leaf* operator O_l^k and a single *root* operator O_r^k, whereas a *multi-stream* query has a single root operator and more than one leaf operators. In a query plan Q_k, an *operator segment* $E_{x,y}^k$ is the sequence of operators that starts at O_x^k and ends at O_y^k. If the last operator on $E_{x,y}^k$ is the root operator, then we simply denote that operator segment as E_x^k. Additionally, E_l^k represents an operator segment that starts at the leaf operator O_l^k and ends at the root operator O_r^k. For example, in Figure 1, $E_1^1 = <O_1, O_3, O_4>$, whereas $E_1^2 = <O_1, O_3, O_5>$.

In a query, each operator O_x^k (or simply O_x) is associated with two parameters:

1. *Processing cost* or *Processing time* (c_x): is the amount of time needed to process an input tuple.

2. *Selectivity* or *Productivity* (s_x): is the number of tuples produced after processing one tuple for c_x time units. s_x is less than or equal to 1 for a filter operator and it could be greater than 1 for a join operator.

Given a single-stream query Q_k which consists of operators $<O_l^k, ..., O_x^k, O_y^k, ..., O_r^k>$ (Figure 1), we define the following characterizing parameters for any operator O_x^k (or equivalently, for any operator segment E_x^k that starts at operator O_x^k):

- **Operator Global Selectivity (S_x^k):** is the number of tuples produced at the root O_r^k after processing one tuple along operator segment E_x^k.

$$S_x^k = s_x^k \times s_y^k \times ... \times s_r^k$$

- **Operator Global Average Cost ($\overline{C}_x^k$):** is the expected time required to process a tuple along an operator segment E_x^k.

$$\overline{C}_x^k = (c_x^k) + (c_y^k \times s_x^k) + ... + (c_r^k \times s_{r-1}^k \times ... \times s_x^k)$$

If O_x^k is a leaf operator ($x = l$), when a processed tuple actually satisfies all the filters in E_l^k, then $\overline{C}_l^k$ represents the ideal total processing cost or time incurred by any tuple *produced* or *emitted* by query Q_k. In this case, we denote $\overline{C}_l^k$ as T_k:

- **Tuple Processing Time (T_k):** is the ideal total processing cost required to produce a tuple by query Q_k.

$$T_k = c_l^k + ... + c_x^k + c_y^k + ... + c_r^k$$

We extend the above parameters for multi-stream queries in Section 5.

3. AVERAGE-CASE PERFORMANCE

In this section, we focus on QoS for single-stream queries and present our scheduling policies for optimizing these metrics. Multi-stream queries are discussed in Section 5.

3.1 Response Time Metric

In DSMSs, the arrival of a new tuple triggers the execution of one or more CQs. Processing a tuple by a CQ might lead to discarding it (if it does not satisfy some filter) or it might lead to producing one or more tuples at the output which means that the input tuple represents an event of interest to the user who installed the CQ. Clearly, in DSMS, it is more appropriate to define response time from data/event perspective rather than from query perspective as in traditional DBMSs. Hence, we define the *tuple response time* or *tuple latency* as follows:

DEFINITION 1. *Tuple response time, R_i, for tuple t_i is $R_i = D_i - A_i$, where A_i is t_i's arrival time and D_i is t_i's output time. Accordingly, the average response time for N tuples is: $\frac{1}{N}\sum_{i=1}^{N} R_i$.*

Notice that tuples that are filtered out do not contribute to the metric as they do not represent any event [22].

3.2 Slowdown Metric

Average response time is an expressive metric in a homogeneous setting. That is, when all tuples require the same processing time. In a heterogeneous workload, as in our system, the processing requirements for different tuples may vary significantly and average response time is not an appropriate metric since it cannot relate the time spent by a tuple in the system to its processing requirements. Other on-line systems with heterogeneous workloads such as DBMSs, OS, and Web servers have adopted *average slowdown* or *stretch* [17] as another metric. This motivated us to consider stretch as the metric in our system.

The definition of slowdown was initiated by the database community in [16] for measuring the performance of a DBMS executing multi-class workloads. Formally, the slowdown of a job is the ratio between the time a job spends in the system to its processing demands [17]. In DSMS, we define the slowdown of a tuple as follows:

DEFINITION 2. *The slowdown, H_i, for tuple t_i produced by query Q_k is $H_i = \frac{R_i}{T_k}$, where R_i is t_i's response time and T_k is its ideal processing time. Accordingly, the average slowdown for N tuples is: $\frac{1}{N}\sum_{i=1}^{N} H_i$.*

Intuitively, in a general purpose DSMS where all events are of the same importance, a simple event (i.e., event detected by a low-cost CQ) should be detected faster than a complex event (i.e., event detected by a high-cost CQ) since the latter contributes more to the load on the DSMS.

3.3 Highest Normalized Rate Policy (HNR)

Based on the above definitions, we developed the *Highest Normalized Rate (HNR)* policy for minimizing average slowdown.

To illustrate the intuition underlying *HNR*, consider two operator segments E_x^i and E_y^j starting at operators O_x^i and O_y^j respectively. For each of the two operator segments, we compute its global selectivity and global average cost as described above. Further, assume that the current wait time for the tuple at the head of O_x^i's queue is W_x^i and for the tuple at the head of O_y^j's queue is W_y^j.

In a policy A where E_x^i is executed before E_y^j, the total slowdown of tuples produced under this policy is:

$$H_A = S_x^i \times H_{A,i} + S_y^j \times H_{A,j} \quad (1)$$

where S_x^i and S_y^j is the number of tuples produced by E_x^i and E_y^j respectively, and $H_{A,i}$ and $H_{A,j}$ are the slowdowns of the E_x^i tuples and the E_y^j tuples respectively.

Recall that the slowdown of a tuple is the ratio between the time it spent in the system to its ideal processing time. Hence, $H_{A,i}$ and $H_{A,j}$ are computed as follows:

$$H_{A,i} = \frac{T_i + W_x^i}{T_i} \qquad H_{A,j} = \frac{\overline{C}_x^i + T_j + W_y^j}{T_j}$$

where $\overline{C}_x^i$ is the amount of time E_y^j will spend waiting for E_x^i to finish execution. By substitution in (1),

$$H_A = S_x^i \times \frac{T_i + W_x^i}{T_i} + S_y^j \times \frac{\overline{C}_x^i + T_j + W_y^j}{T_j}$$

Similarly, under an alternative policy B, where E_y^j is executed before E_x^i, the total slowdown H_B is:

$$H_B = S_y^j \times \frac{T_j + W_y^j}{T_j} + S_x^i \times \frac{\overline{C}_y^j + T_i + W_x^i}{T_i}$$

In order for H_A to be less than H_B, then the following inequality must be satisfied:

$$S_y^j \times \frac{\overline{C}_x^i}{T_j} < S_x^i \times \frac{\overline{C}_y^j}{T_i} \quad (2)$$

The left-hand side of Inequality 2 shows the *increase* in total slowdown incurred by the tuples produced by E_y^j when E_x^i is executed first. Similarly, the right-hand side shows the increase in total slowdown incurred by the tuples produced by E_x^i when E_y^j is executed first. The inequality implies that between the two alternative execution orders, we should select the one that minimizes the increase in the total slowdown. That is, we should select the segment with the smallest negative impact on the other one.

Thus, in our *HNR* policy, each operator O_x^k is assigned a priority V_x^k which is the *weighted rate* or *normalized rate* of the operator segment E_x^k that starts at operator O_x^k and it is defined as:

$$V_x^k = \frac{1}{T_k} \times \frac{S_x^k}{\overline{C}_x^k} \quad (3)$$

The term $S_x^k/\overline{C}_x^k$ is basically the *global output rate (GR_x^k)* of the operator segment starting at operator O_x^k as defined in [23]. As such, the priority of each operator O_x^k is its normalized output rate, or equivalently, the normalized output rate of the operator segment E_x^k starting at O_x^k. Hence, executing O_x^k implies the pipelined execution of all the operators on E_x^k unless it is interrupted by a higher priority operator as we will describe in Section 6.