METRICS AND ALGORITHMS FOR PROCESSING MULTIPLE CONTINUOUS QUERIES

by

Mohamed A. Sharaf

M.Sc. in Computer Science, University of Pittsburgh, 2004

M.Sc. in Computer Engineering, Cairo University, 2000

B.Sc. in Computer Engineering, Cairo University, 1997

Submitted to the Graduate Faculty of

the Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2007

UNIVERSITY OF PITTSBURGH

ARTS AND SCIENCES

This dissertation was presented

by

Mohamed A. Sharaf

It was defended on

June 22nd 2007

and approved by

Panos K. Chrysanthis, PhD, Professor

Alexandros Labrinidis, PhD, Assistant Professor

Walid Aref, PhD, Professor

Christos Faloutsos, PhD, Professor

Kirk Pruhs, PhD, Professor

Dissertation Advisors: Panos K. Chrysanthis, PhD, Professor,

Alexandros Labrinidis, PhD, Assistant Professor

METRICS AND ALGORITHMS FOR PROCESSING MULTIPLE CONTINUOUS QUERIES

Mohamed A. Sharaf, PhD

University of Pittsburgh, 2007

Data streams processing is an emerging research area that is driven by the growing need for *monitoring applications*. A monitoring application continuously processes streams of data for interesting, significant, or anomalous events. Such applications include tracking the stock market, real-time detection of disease outbreaks, and environmental monitoring via sensor networks.

Efficient employment of those monitoring applications requires advanced data processing techniques that can support the continuous processing of unbounded rapid data streams. Such techniques go beyond the capabilities of the traditional *store-then-query* Data Base Management Systems. This need has led to a new data processing paradigm and created a new generation of data processing systems, supporting *continuous queries* (CQ) on data streams.

Primary emphasis in the development of first generation *Data Stream Management Systems* (DSMSs) was given to basic functionality. However, in order to support large-scale heterogeneous applications that are envisioned for subsequent generations of DSMSs, greater attention will have to be paid to performance issues. Towards this, this thesis introduces new algorithms and metrics to the current design of DSMSs.

This thesis identifies a collection of *quality of service (QoS)* and *quality of data (QoD)* metrics that are suitable for a wide range of monitoring applications. The establishment of well-defined metrics aids in the development of novel algorithms that are optimal with respect to a particular metric.

Our proposed algorithms exploit the valuable chances for optimization that arise in the presence of multiple applications. Additionally, they aim to balance the trade-off between the DSMS's overall performance and the performance perceived by individual applications. Furthermore, we provide efficient implementations of the proposed algorithms and we also extend them to exploit sharing in optimized multi-query plans and multi-stream CQs. Finally, we experimentally show that our algorithms consistently outperform the current state of the art.

TABLE OF CONTENTS

PREFACE . . . xii
1.0 INTRODUCTION . . . 1
2.0 SYSTEM MODEL . . . 4
2.1 Continuous Queries . . . 4
2.2 Continuous Query Processing . . . 7
2.2.1 Processing Sliding Window Joins . . . 8
2.2.2 Processing Sliding Window Aggregates . . . 9
3.0 QOS METRICS AND ALGORITHMS . . . 11
3.1 Average-case Performance . . . 13
3.1.1 Response Time Metric . . . 13
3.1.1.1 Highest Rate Policy (HR) . . . 13
3.1.2 Slowdown Metric . . . 14
3.1.3 Highest Normalized Rate Policy (HNR) . . . 15
3.1.4 HNR vs. HR . . . 17
3.1.5 HNR vs. HR vs. SRPT . . . 19
3.2 Average-case vs. Worst-case Performance . . . 20
3.2.1 Worst-case Performance . . . 20
3.2.2 Balancing the Trade-off between Average-case and Worst-case Performance . . . 21
3.2.2.1 The Second Norm Metric . . . 21
3.2.2.2 Balancing the Trade-off for Slowdown . . . 21
3.2.3 Balancing the Trade-off for Response Time . . . 23

3.3 Implementation Issues 24
3.3.1 Priority Dynamics under HNR 24
3.3.2 Priority Dynamics under BSD 24
3.3.2.1 Search Space Reduction 25
3.3.2.2 Search Space Pruning 26
3.3.2.3 Clustered Processing 27
3.3.3 Adaptive Scheduling 28
3.4 Multi-Stream Queries 31
3.4.1 Metrics For Joins 31
3.4.1.1 Response Time of Joined Tuples 31
3.4.1.2 Slowdown of Joined Tuples 32
3.4.2 Scheduling Multi-stream Queries 34
3.5 Aggregate Continuous Queries 36
3.5.1 EDF vs. HR for Scheduling Aggregate CQs 36
3.5.2 Hybrid Policy for Scheduling Aggregate CQs 37
3.6 Operator Sharing 39
3.6.1 HNR with Operator Sharing 39
3.6.2 Priority-Defining Tree (PDT) 41
3.7 Evaluation Testbed 42
3.8 Experiments 44
3.8.1 Performance under Different Metrics 44
3.8.1.1 Average Slowdown 44
3.8.1.2 Average Response Time 45
3.8.1.3 Maximum Response Time 45
3.8.1.4 Maximum Slowdown 45
3.8.1.5 Trade-off in Slowdown 47
3.8.1.6 Second Norm of Slowdowns 47
3.8.1.7 Second Norm of Response Times 47
3.8.1.8 Slowdown per Class 49
3.8.1.9 Impact of Selectivity 49

3.8.1.10 An Oracle Scheduling Policy . . . 50
3.8.1.11 Performance over Time . . . 52
3.8.1.12 Second Norm for Multi-stream Queries . . . 52
3.8.1.13 Tardiness of Aggregate CQs . . . 54
3.8.2 Memory Usage . . . 56
3.8.3 Comparison of Implementation Techniques . . . 58
3.8.4 Operator Sharing . . . 59
3.8.5 Adaptive Scheduling . . . 62
4.0 QOD METRICS AND ALGORITHMS . . . 65
4.1 Freshness of Data Streams . . . 67
4.1.1 Average Freshness for Single Streams . . . 67
4.1.2 Average Freshness for Multiple Streams . . . 69
4.2 Freshness-Aware Scheduling of Multiple Continuous Queries . . . 70
4.2.1 Scheduling without Selectivity . . . 70
4.2.2 Scheduling with Selectivity . . . 72
4.2.3 The FAS-MCQ Policy . . . 73
4.2.4 Weighted Freshness . . . 73
4.3 Scheduling for QoD vs. Scheduling for QoS . . . 76
4.3.1 Scheduling for QoS . . . 76
4.3.2 Balancing the Trade-off between QoD and QoS . . . 77
4.4 Evaluation Testbed . . . 78
4.4.1 Implementing the FAS-MCQ Scheduler . . . 78
4.4.2 Simulation Parameters . . . 78
4.5 Experiments . . . 81
4.5.1 Impact of Utilization . . . 81
4.5.2 Staleness vs. Response Time . . . 82
4.5.3 Impact of Selectivity . . . 84
4.5.4 Impact of Bursts . . . 85
4.5.5 Real Data . . . 86
5.0 RELATED WORK . . . 88

6.0 SUMMARY AND FUTURE WORK 91
6.1 Summary 91
6.2 Future Work 94
6.2.1 Integrated Processing and Dissemination Schedulers 94
6.2.2 Integrated Load Shedding 94
BIBLIOGRAPHY 96

LIST OF TABLES

1 Table of Symbols 15
2 Results for Example 1 19
3 Simulation Parameters for QoS Experiments 43
4 Priority functions for scheduling QoS vs. QoD 77
5 Simulation Parameters for QoD Experiments 80
6 Classification of priority-based scheduling policies for CQs 93

LIST OF FIGURES

1 Core components of a DSMS . . . 5
2 Continuous Queries Plans . . . 6
3 Output of Example 1 . . . 18
4 An example that illustrates the different implementation techniques . . . 28
5 An example of a multi-stream query plan . . . 32
6 Multiple CQs plans sharing operator O_x . . . 39
7 [§3.8.1.1] Avg. slowdown vs. system load . . . 44
8 [§3.8.1.2] Avg. response vs. system load . . . 45
9 [§3.8.1.3] Max. response time vs. system load . . . 46
10 [§3.8.1.4] Max. slowdown vs. system load . . . 46
11 [§3.8.1.5] Max. vs Avg. Slowdown for HNR, LSF, and BSD . . . 47
12 [§3.8.1.6] ℓ_2 of slowdowns vs. system load . . . 48
13 [§3.8.1.7] ℓ_2 of response times vs. system load . . . 48
14 [§3.8.1.8] Slowdown per class for low-cost queries . . . 49
15 [§3.8.1.9] ℓ_2 of slowdown vs. maximum operator selectivity . . . 50
16 [§3.8.1.10] Performance of an oracle scheduling policy . . . 51
17 [§3.8.1.11] Response time over time . . . 53
18 [§3.8.1.11] Slowdown over time . . . 53
19 [§3.8.1.11] ℓ_2 of response times over time . . . 53
20 [§3.8.1.11] ℓ_2 of slowdowns over time . . . 53
21 [§3.8.1.12] ℓ_2 of slowdown for multi-stream queries . . . 54
22 [§3.8.1.13] Tardiness of aggregate CQs . . . 54

23 [§3.8.1.13] Tardiness of aggregate CQs at low utilization 55
24 [§3.8.1.13] Tardiness of aggregate CQs at high utilization 55
25 [§3.8.2] Memory usage vs. system load 56
26 [§3.8.2] Performance of Chain under QoS metrics 57
27 [§3.8.3] ℓ_2 of slowdown vs. number of clusters 58
28 [§3.8.3] Efficient implementation of BSD 59
29 [§3.8.4] Response time for grouped queries 60
30 [§3.8.4] Slowdown for grouped queries 61
31 [§3.8.4] ℓ_2 of slowdowns for grouped queries 61
32 [§3.8.5] Ratio of adaptive scheduler performance vs. static 62
33 [§3.8.5] Impact of monitoring window length on adaptive scheduling 63
34 [§3.8.5] Impact of α value on adaptive scheduling 64
35 An example on measuring the freshness of a data stream 68
36 [§4.5.1] Average staleness vs. system utilization 81
37 [§4.5.1] Average response time vs. system utilization 82
38 [§4.5.2] Response time for different βs 83
39 [§4.5.2] Staleness for different βs 83
40 [§4.5.2] Trade-off between staleness and response time at utilization 0.95 . . . 84
41 [§4.5.3] Staleness vs. skewness in selectivity (using Zipf parameter) 85
42 [§4.5.4] Staleness vs. number of bursty streams (out of 10) 86
43 [§4.5.5] Staleness vs. system utilization (real data traces) 87

PREFACE

I would like to thank my advisors, Panos K. Chrysanthis and Alexandros Labrinidis, for their guidance and support throughout my PhD studies. Their knowledge, intelligence, dedication, and personal integrity have helped me set higher standards for myself on the academic, professional, and personal levels. Moreover, they are two of the most thoughtful and considerate people that I have ever met in my life.

Panos has provided me with all the time and support I have needed. He has always presented me with observations and challenges that have helped me to continuously improve my work. I also appreciate his insightful vision that made my research more comprehensive and mature.

Alex has taught me how to formulate research problems and how to clearly present my solutions. He has always been able to identify new problems that added depth and strength to my research. I also appreciate his constructive critiques and his careful review of my work.

I am also grateful to Kirk Pruhs for spending long hours with me working on several theoretical problems that I have faced during my research. I have been very fortunate to work with him.

I would also like to extend my gratitude to my external committee members, Christos Faloutsos and Walid Aref, for their support, thoughtfulness, and the invaluable reviews and suggestions.

Acknowledgment is also due to the NSF for supporting my research through grants: ANI-0123705, ANI-0325353, and IIS-0534531.

I also appreciate the collaborative work I have had with several members of the ADMT Lab. at Pitt, especially, my work with Jonathan Beaver and Shenoda Guirguis.

I have also enjoyed the support of many friends during my stay in Pittsburgh. I would

especially mention Soudi Abdesalam, Christina Adamou, Polina Kats, Yiannis Koutis, and Panickos Neophytou.

Finally, the most appreciation is due to my family: my mother Shadya, my father Abdel-Kader, and my brother Ahmed. With gratitude, I dedicate this work to them.

1.0 INTRODUCTION

Mission-critical systems incorporate a set of sub-systems and personnel that work together for accomplishing a certain set of critical tasks. Succeeding in accomplishing a critical task relies on different factors including the early detection of events, and the timely availability of the right information at the right place and to the right people. To fulfill these requirements, *monitoring applications* are employed as a core component in any mission-critical system. That is, sub-systems will install a set of monitoring applications that continuously report the information required for these sub-systems to accomplish their tasks. Typically, events arrive at the different components in the form of *data streams* which a monitoring application continuously scans, searching for significant or anomalous events, and reports its findings in near real-time.

Efficient employment of monitoring applications requires using advanced data processing techniques that can support the continuous processing of continuous rapid data streams. Such techniques go beyond the capabilities of the traditional *store-then-query* Data Base Management Systems and have to be implemented in an ad-hoc manner by a combination of stored procedures, triggers, and external database applications. This need has led to a new data processing paradigm and created a new generation of data processing systems, called *Data Stream Management Systems* (DSMSs), that support *Continuous Queries* (CQs) on data streams. In such systems, each monitoring application registers a set of CQ, where a CQ is continuously executed with the arrival of new relevant data [63]. Tribeca [60], Niagara [20], Aurora [15], STREAM [44], TelegraphCQ [17], Gigascope [23], Niagara [20] and Nile [31] are examples of current prototype DSMSs.

Primary emphasis in the development of first generation DSMSs was given to basic functionality of query processing and workload shaping. In order to support the large scale

applications that are envisioned for subsequent generations of DSMSs, greater attention will have to be paid to performance issues.

Such performance can be captured by means of non-functional, *Quality of Service (QoS)* measures as well as functional, *Quality of Data (QoD)* measures. Optimizing performance will require the development of integrated policies and new execution algorithms of query operators that exploit the specific properties of queries and state of the DSMS components. It will also result in new designs for DSMSs.

In this thesis, we address the following algorithmic and design issues:

1. **Collection of QoS and QoD metrics:** We identify a collection of *quality of service (QoS)* and *quality of data (QoD)* metrics that are suitable for a wide range of monitoring applications. For instance, *fairness* is one metric that has been overlooked in current DSMSs prototypes.
2. **Scheduling Policies:** The establishment of well-defined metrics will aid us in the development of novel algorithms for query scheduling policies, as we seek policies that are optimal with respect to a particular metric or algorithms that can strike a fine balance between different metrics.

The contribution of this dissertation is a set of multiple query scheduling policies whose effectiveness is empirically demonstrated [55, 57, 54, 56]. Further, our analysis and comparison of our proposed algorithms with the various algorithms in the literature is expected provides an insight to the inherent performance trade-offs for the metrics that we identify.

Specifically, we examine the problem of how to schedule multiple Continuous Queries (CQs) in a DSMS to optimize different Quality of Service (QoS) metrics. We show that, unlike traditional on-line systems, scheduling policies in DSMSs that optimize for average response time will be different from policies that optimize for average slowdown, which is a more appropriate metric to use in the presence of a heterogeneous workload. Towards this, we propose policies to optimize for the average-case performance for both metrics.

Additionally, we propose a hybrid scheduling policy that strikes a fine balance between performance and fairness, by looking at both the average- and worst-case performance, for both metrics. We also show how our policies can be adaptive enough to handle the inherent

dynamic nature of monitoring applications. Furthermore, we discuss how our policies can be efficiently implemented and extended to exploit sharing in optimized multi-query plans and multi-stream CQs.

Finally, we propose to exploit query scheduling to improve QoD in DSMSs. Specifically, we are presenting a new policy for scheduling multiple continuous queries with the objective of maximizing the freshness of the output data streams and hence the QoD of such outputs. The proposed Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ) policy decides the execution order of continuous queries based on each query's properties (i.e., cost and selectivity) as well the properties of the input update streams (i.e., variability of updates). We also propose and evaluate a parameterized version of our FAS-MCQ scheduler that is able to balance the trade-off between freshness and response time according to the application's requirements.

The rest of the thesis is organized as follows: Chapter 2 provides the system model. In Chapter 3, we define our QoS metrics and present our proposed scheduling policies for improving QoS for single-stream and multi-stream queries. It also includes implementation details, scheduling queries with shared operators, and experimental evaluation and results for QoS metrics and policies. In Chapter 4, we present our freshness-based QoD metric along with scheduling polices and experimental results. Chapter 5 describes the related work and in Chapter 6, we summarize the thesis and discuss potential future work.

2.0 SYSTEM MODEL

We assume a DSMS whose four core components are shown in Figure 1. Specifically, the figure shows the following components: 1) the query optimizer, 2) the query scheduler, 3) the load shedder and 4) the memory manager. The role of each these components will be illustrated through the following discussion.

2.1 CONTINUOUS QUERIES

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.

For the purpose of this work, we assume a SQL-like continuous query language (e.g., CQL [4]) which is used in the STREAM system. CQL (Continuous Query Language) provides the typical semantics of SQL in addition to stream window semantics. A continuous query evaluation plan can be conceptualized as a data flow tree [15, 6], where the nodes are operators that process tuples and edges represent the flow of tuples from one operator to another (Figure 2). 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.

Multiple queries with common sub-expressions are usually merged together to eliminate the repetition of similar operations [48]. For example, Figure 2 shows the global plan for

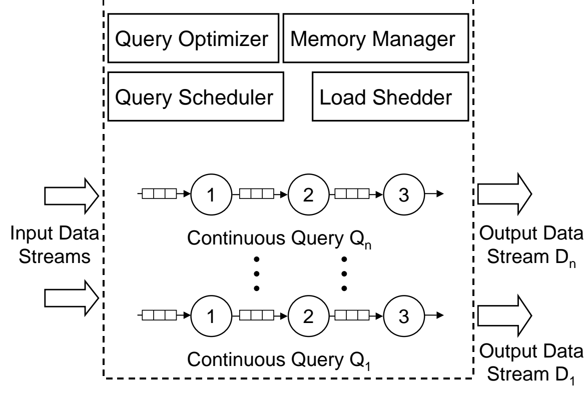

Figure 1: Core components of a DSMS

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, as illustrated by the half and half pattern for these operators.

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 2, $E_1^1 = <O_1, O_3, O_4>$, 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 2), we define the following characterizing parameters for

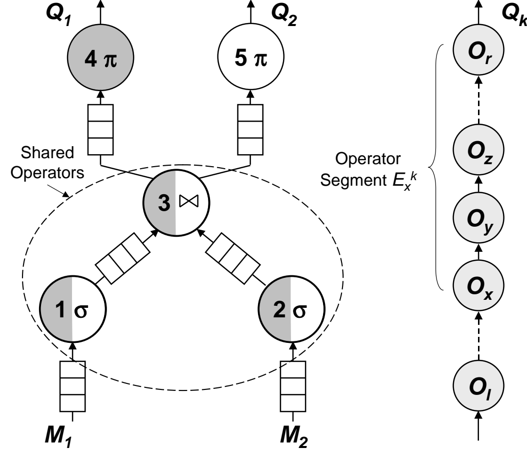

Figure 2: Continuous Queries Plans

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 3.4.

2.2 CONTINUOUS QUERY PROCESSING

The DSMS is responsible for processing the multiple CQs registered by different monitoring applications. In order to accomplish that, the DSMS uses a *query optimizer* that decides the query plans and the execution algorithm used by each query operator.

The DSMS also employs a *query scheduler* that decides the local execution order of operators within each CQ as well as the global execution order of multiple CQs. The algorithms and policies employed by the DSMS directly affects its overall performance.

Improving the response time of a single query over data streams has been the focus of many research efforts (e.g., [16, 66, 64]). These efforts are the successors of past efforts on improving the response time of interactive queries over Web databases (e.g., [65, 20]). The work in [66] proposed a *rate-based* query optimization technique for CQs to replace the traditional *cost-based* query optimization in DBMSs. However, in both optimization strategies, the way operators are scheduled can lead to significantly different kinds of output behavior for the same generated query plan [65]. The work in [65] focused on the problem of operator scheduling. It proposed the dynamic rate-based pipeline scheduling policy. Aurora [15, 16] also uses a policy similar to the rate-based pipeline scheduling to minimize the average tuple latency.

For multiple queries, multi-query optimization has been exploited by [19] to improve system throughput in an Internet environment and by [42, 18] for improving the throughput of DSMSs. Multi-query scheduling is exploited by Aurora to achieve application-specified QoS requirements [16]. In Aurora, each query is associated with a QoS graph which defines the utility of stale output. That is, the QoS decreases with the increase in tuple's response time. The QoS graph is user-specified and it assumes that the specification is feasible.

In the next section, we will describe QoS requirements based on the *stretch* metric [43, 12, 45, 2]. We believe that the stretch of the output provides a natural way to define the QoS requested by a query for the following reasons: (1) it does not require the user to have any prior knowledge about the query processing requirements or to guess the appropriate QoS graph; and (2) it prevents the user from specifying unrealistic QoS requirements for the submitted query.

Aside from QoS, QoD is another important requirement in a DSMS. QoD typically measures the *freshness* of data, that is the deviation between the latest information provided to an application and the latest available update for that information. That deviation could be time-based, value-based or both time-based and value-based. For a comprehensive overview of QoD metrics can be found in our work in [35].

Current DSMSs prototypes do not provide any mechanisms to control the provided QoD. However, QoD has been extensively studied in the context of online and Web databases. For example, the work in [21, 22] provides policies for crawling the Web in order to improve the freshness of a local database. More mechanisms have been exploited in [47] for refreshing distributed caches and in [36] for multi-casting updates over the Web.

2.2.1 Processing Sliding Window Joins

Understanding the semantics of sliding window joins is essential for extending the QoS and QoD metrics for multi-stream queries as well as designing multi-stream query schedulers as proposed in Section 3.4. To simplify the discussion, we assume time-based sliding window equi-joins that use *Symmetric Hash Join* (SHJ) [68, 33] which is a non-blocking, in-memory join processing algorithm.

To illustrate the semantics of a time-based sliding window join, let us assume a sliding window continuous query Q that performs a join between two streams M_L and M_R with a window interval V_Q. Each tuple that arrives at the system has a *timestamp* which is either assigned by the data source or the DSMS. For such a query Q, when a tuple t arrives at stream M_L, it will be compared against the tuples from M_R that are within V_Q time units from t's timestamp [23, 15]. That range defines the set of tuples from M_R that are compared against the newly arriving tuple at M_L. Out of those tuples, the ones that satisfy the join predicate are streamed up the query plan.

To use SHJ for performing the join operation in the query described above, hash tables H_L and H_R are defined over streams M_L and M_R respectively. As a tuple t with timestamp $t.ts$ arrives at one of the streams (say M_L), it is first hashed and inserted into H_L, then the hash value is used to probe H_R for tuples with matching key. Out of those matching

tuples, each tuple that satisfies the window predicate is concatenated to the input tuple and a new composite tuple is generated. Additionally, all tuples in H_R with a timestamp less than $(t.ts - V_Q)$ are pruned from the hash table since they are not expected to match any of the tuples that will arrive at M_L in the future.

2.2.2 Processing Sliding Window Aggregates

Aggregate queries over a data stream typically use windows to divide the unbounded data stream into subsets of tuples. A *Sliding Window*, which is the most widely used type of windows, is defined by means of two attributes: 1) *RANGE*; and 2) *SLIDE*. *RANGE* defines the length of the window which is either time-based or tuple-based, whereas *SLIDE* defines how the window boundaries move over the data stream. Processing each subset of tuples produces new values for the aggregate function used in the aggregate query. We will call each produced aggregate result an *aggregate instant*.

When the SLIDE is less then the RANGE, different consecutive windows overlap and a single tuple will belong to more than one window, hence, it is involved in the computation of different aggregate instants. For example, for an aggregate query with a RANGE of 100 tuples and a SLIDE of 100 tuples, there is no overlapping between consecutive windows and a new aggregate value should be produced for every 100 tuples entering the system. However, if the SLIDE is 5 tuples, then the boundary line is reached after the arrival of each 5th tuple and each aggregation is performed over the last 100 tuples.

In a straight forward implementation of aggregates, input tuples are buffered and once a boundary line is reached, the aggregate function is evaluated using the buffered tuples that are within the window boundaries. After evaluating the aggregate, the window boundaries are shifted and all the buffered tuples that fall outside the new boundaries are expired since they cannot be involved in any future computation.

An alternative implementation that uses *Window-Id (WID)* appeared in [38]. In that implementation, each input tuple is mapped to one or more WID. The mapping is done using the tuple's timestamp for time-based windows, or using a tuple's serial number for tuple-based windows. For instance, in our example above where the RANGE is 100 and

the SLIDE is 5, each arriving tuple (after the first 100) is mapped to the 20 WIDs that correspond to all the windows where the input tuple is used for generating 20 aggregate instants.

Additionally, a hash table is used to incrementally maintain the aggregate values for each of the currently active windows. Specifically, after mapping a newly input tuple to its set of corresponding WIDs, for each WID, if it exists in the hash table, then the aggregate value is updated, whereas if it does not exist in the hash table, then a new entry is inserted. When a new tuple signals that the boundary of some window has been reached, then the aggregate instant for that window is retrieved from the hash table and the corresponding entry is purged.

It should be clear that the WID implementation leads to significant saving in buffer space. However, the amount of savings depends on the ratio between the window size and the hop size. Additionally, when augmenting the WID approach with *panes* [37], significant saving in processing time is achieved.

The techniques mentioned above fall under the *Input Triggered* execution model [29]. In the Input Triggered execution model the operators are activated with the arrival of new input tuples. Unfortunately, that execution model will not always provide correct answers for time-based windowed aggregates. The reason is that if the input rate of tuples is relatively low, then using the timestamps associated with the input tuples to detect which tuples to expire and which is the current window instant is insufficient. For example, if the RANGE and SLIDE are 100 and 10 time units respectively and the tuples' input rate is uniform and it is 1 tuple every 20 time units, then at time 110, the window should slide (i.e., the first tuple should expire) and a new aggregate should be reported. However, since the next tuple will not arrive until time 120, then the system will only update the aggregate at time 120 and report the aggregate value for the window instant 20-120 and it will miss reporting the aggregate for the window 10-110.

The above anomaly motivated the proposal of what is called the *Clock Triggered* execution model where windowed query processing is based on time [29]. Currently, time probing [29], negative tuples [29] and direct approach [28] are examples of mechanisms for implementing the Clock Triggered execution model.

3.0 QOS METRICS AND ALGORITHMS

One of the main goals in the design of a data stream management system is the development of scheduling policies that optimize *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 most commonly used QoS metric in the literature is *average response time*. In this work, we show that if the objective is to optimize the response time, then the "right" strategy is to schedule CQs according to their *output rate*. Specifically, we present a new scheduling policy called *Highest Rate (HR)*. *HR* generalizes the *Rate-based policy (RB)* [65] 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.

Although scheduling to minimize average response time works well for homogeneous workloads, there are some well known disadvantages to using average response time as the metric to optimize when the workload is heterogeneous. In the above example, the user who 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 [45]. 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 [45]. A surprising discovery of this work 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 work, 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. We call this policy the *Highest Normalized Rate (HNR)* policy.

The average slowdown provided by the DSMS captures the system's average-case performance. 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 slowdown* of the system [13], i.e., 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 crucial 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 [9]. 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*.

3.1 AVERAGE-CASE PERFORMANCE

In this section, we focus on QoS metrics for single-stream queries and present our scheduling policies for optimizing these metrics. Multi-stream queries are discussed in Section 3.4.

3.1.1 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 predicate) 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 registered the CQ. Clearly, in DSMSs, it is more appropriate to define response time from a data/event perspective rather than from a 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 [64].

3.1.1.1 Highest Rate Policy (HR)

The *Rate-based* policy (*RB*) has been shown to improve the average response time of a single query [65]. In Aurora [16], *RB* was used for scheduling operators within a query, after the query had been selected by *Round Robin (RR)*. Below, we present a policy that extends *RB* for scheduling both queries and operators.

In the basic *RB* policy, each operator path within a query is assigned a priority that is equal to its output rate. The path with the highest priority is the one scheduled for execution. In our proposed *Highest Rate* policy (*HR*), we simply view the network of multiple queries as a set of operators and at each scheduling point we select for execution the operator with the highest priority (i.e., output rate).

Specifically, under *HR*, each operator O_x^k is assigned a value called *global output rate* (GR_x^k). The output rate of an operator is basically the expected number of tuples produced per time unit due to processing one tuple by the operators along the operator segment starting at O_x^k all the way to the root O_r^k. Formally, the output rate of operator O_x^k is defined as follows:

$$GR_x^k = \frac{S_x^k}{\overline{C}_x^k} \tag{3.1}$$

where S_x^k and $\overline{C}_x^k$ are the operator's global selectivity and global average cost as defined in Section 2. The intuition underlying *HR* is to give higher priority to operator paths that are both productive and inexpensive. In other words, the highest priority is given to the operator paths with the minimum latency for producing one tuple.

The priority of each operator O_x^k is set to its global output rate GR_x^k, or equivalently, the output rate of the operator segment E_x^k starting at O_x^k. Hence, the priority of E_x^k is basically equal to the priority of O_x^k and 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 (or operator segment) as we will describe in Section 3.3.

3.1.2 Slowdown Metric

Average response time is an expressive metric in a homogeneous setting, i.e., when all tuples require the same processing time. However, 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. Given this realization, other on-line systems with heterogeneous workloads such as DBMSs, OSs, and Web servers have adopted *average slowdown* or *stretch* [45] 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 [43] 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 [45]. In DSMS, we define the slowdown of a tuple as follows:

Table 1: Table of Symbols

Symbol	Description
O_x^i	Operator x in query i
$E_{x,y}^i$	Segment of operators that starts at O_x^i and ends at O_y^i
E_x^i	Segment of operators that starts at O_x^i and ends at the root O_r^i
c_x^i	Processing time/cost of operator O_x^i
s_x^i	Selectivity of operator O_x^i
$\overline{C}_x^i$	Expected processing time/cost of operator segment E_x^i
S_x^i	Selectivity of operator segment E_x^i
W_x^i	Wait time for tuple at the head of O_x^i's input queue
T_i	Ideal processing time/cost of a tuple produced by query Q_i
V_x	Window interval for join operator O_x
τ_l	Mean inter-arrival time of data stream M_l
SE_x	Set of operator segments starting at shared operator O_x
$\overline{SC}_x$	Expected processing time/cost of set of segments in SE_x

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.

3.1.3 Highest Normalized Rate Policy (HNR)

Based on the above definitions, we developed the *Highest Normalized Rate (HNR)* policy for minimizing average slowdown. Table 1 summarizes the parameters used for describing the *HNR* policy for single-stream queries as well as the other scheduling policies policies discussed in the next Section. It also includes the parameters used for join operators (Section 3.4) and shared operators (Section 3.6).

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.

We then consider two different scheduling policies:

- **Policy** (A), where E_x^i is executed before E_y^j, and
- **Policy** (B), where E_y^j is executed before E_x^i.

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

$$H_A = S_x^i \times H_{A,i} + S_y^j \times H_{A,j} \tag{3.2}$$

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 (3.2),

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

Similarly, under the 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:

$$S_y^j \times \frac{\overline{C}_x^i}{T_j} < S_x^i \times \frac{\overline{C}_y^j}{T_i} \tag{3.3}$$

The left-hand side of Inequality 3.3 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.

In order to select the segment with the smallest negative impact, 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:

$$V_x^k = \frac{1}{T_k} \times \frac{S_x^k}{\overline{C}_x^k} \tag{3.4}$$

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 [65]. 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 3.3.

3.1.4 HNR vs. HR

It is interesting to notice that if the objective is optimizing the response time, then the ideal total processing cost T should be eliminated from the denominators of all the above equations resulting in setting the priority V_x^k of operator O_x^k to:

$$V_x^k = \frac{S_x^k}{\overline{C}_x^k} = GR_x^k \tag{3.5}$$

In fact, this is the prioritizing function we use in our *Highest Rate (HR)* policy for optimizing the response time presented in Section 3.1.1.1. The *HR* policy, schedules jobs in descending order of output rate which might result in a high average slowdown because a low-cost query can be assigned a low priority since it is not productive enough. Those few tuples produced by this query will all experience a high slowdown, with a corresponding increase in the average slowdown of the DSMS.

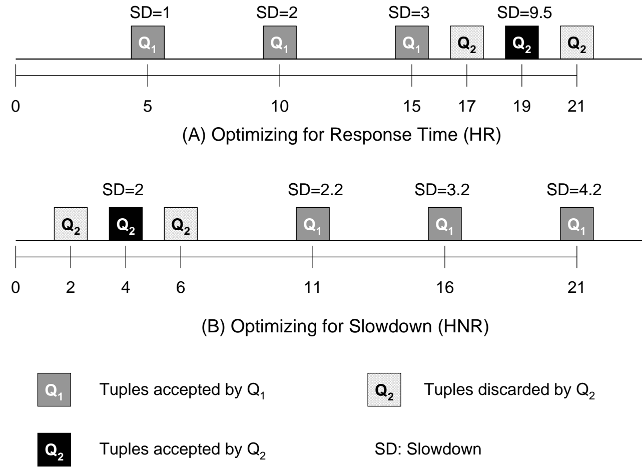

Figure 3: Output of Example 1

Our policy *HNR*, like *HR*, is based on output rate, however, it also emphasizes the ideal tuple processing time in assigning priorities. As such, an inexpensive operator segment with low productivity will get a higher priority under *HNR* than under *HR*.

Example 1 To further illustrate the difference between the *HR* and the *HNR* policies, let us consider an example where we have two queries Q_1 and Q_2. Each query consists of a single operator. For Q_1, the cost of the operator is 5 ms and its selectivity is 1.0. For Q_2, the cost of the operator is 2 ms and its selectivity is 0.33. Further, assume that there are 3 pending tuples to be processed by the 2 queries and that all 3 tuples have arrived at time 0.

Under the *HR* policy, Q_1's priority is $\frac{1.0}{5.0} = 0.2$, whereas Q_2's priority is $\frac{0.33}{2.0} = 0.1667$ (which is the output rate of each query). Figure 3(A) shows the queries' output under the *HR* policy where Q_1 is executed first and it accepted/emitted all the pending 3 tuples, then Q_2 is executed and it only accepted one of the 3 pending tuples (since its selectivity is 0.33); we assume it was the middle one in this example.

Under the *HNR* policy, Q_1's priority is $\frac{1.0}{5.0 \times 5.0} = 0.04$, whereas Q_2's priority is $\frac{0.33}{2.0 \times 2.0} = 0.08$. Hence, under *HNR*, Q_2 is scheduled before Q_1 resulting in the output shown in Figure 3(B).

Table 2: Results for Example 1

	Response Time	Slowdown
HR	12.25	3.875
HNR	13.0	2.9

Table 2 summarizes the results of the two different policies and shows that *HNR* provides the lower average slowdown compared to *HR*. The reason is that the one tuple accepted by Q_2 experienced a slowdown of $\frac{4}{2} = 2.0$ under *HNR* while its slowdown under *HR* is $\frac{19}{2} = 9.5$. This unfairness of *HR* toward Q_2 resulted in a higher overall average slowdown compared to *HNR*.

3.1.5 HNR vs. HR vs. SRPT

It should be clear that under *HR*, if all the operators' selectivities are equal to one, then Equation 3.5 is simply the inverse of the processing time. Hence, in this case, *HR* is equivalent to *SRPT*. Similarly, if all the operators' selectivities are equal to one, then in Equation 3.4, $\overline{C}_x^k$ is equal to T_k and O_x^i is executed before O_y^j if $1/(T_i)^2 > 1/(T_j)^2$. By taking the square root of both sides, then *HNR* is also equivalent to *SRPT*.

The above observation shows the effect of the selectivity parameter on this problem. That is, under a probabilistic workload, *HR* reduces the response time, whereas, *HNR* reduces the slowdown. However, as the workload becomes deterministic, both *HR* and *HNR* converge to a single policy which is the *SRPT* policy, which has been shown to be optimal for task scheduling when looking at response time and near optimal when looking at slowdown.

3.2 AVERAGE-CASE VS. WORST-CASE PERFORMANCE

Here, we first define the worst-case performance and a policy that minimizes it. Then, we introduce our scheduling policy for balancing the trade-off between the average- and worst-case performance.

3.2.1 Worst-case Performance

It is expected that a scheduling policy that strives to minimize the average-case performance might lead to a poor worst-case performance under a relatively high load. That is, some queries (or tuples) might starve under such a policy. The worst-case performance is typically measured using *maximum response time* or *maximum slowdown* [13].

Definition 3 *The maximum response for N tuples is max$(R_1, R_2, ..., R_N)$.*

Definition 4 *The maximum slowdown for N tuples is max$(H_1, H_2, ..., H_N)$.*

Intuitively, a policy that optimizes for the worst-case performance should be pessimistic. That is, it assumes the worst-case scenario where each processed tuple will satisfy all the filters in the corresponding query. An example of such a policy is the traditional *First-Come-First-Serve (FCFS)* that has been shown to optimize the maximum response time metric in [13]. Similarly, the traditional *Longest Stretch First (LSF)* [2] has been shown to optimize the maximum slowdown. Under *LSF*, each operator O_x^k is assigned a priority V_x^k which is computed as:

$$V_x^k = \frac{W_x^k}{T_k} \tag{3.6}$$

where W_x^k is the wait time of the tuple at the head of O_x^k's input queue and T_k is the ideal processing cost for that tuple.

LSF is a greedy policy under which the priority assigned to an operator O_x^k is basically the current slowdown of the tuple at the top of O_x^k's input queue; the current slowdown of a tuple is the ratio of the time the tuple has been in the system thus far to its processing time.

3.2.2 Balancing the Trade-off between Average-case and Worst-case Performance

A policy that strikes a fine balance between the average-case and worst-case performance needs a metric that is able to capture this trade-off. In this section, we first present such a metric, and then describe our proposed scheduling policy which optimizes that metric.

3.2.2.1 The Second Norm Metric On one hand, the average value for a QoS metric provided by the system represents the expected QoS experienced by any tuple in the system (i.e., the average-case performance). On the other hand, the maximum value measures the worst QoS experienced by some tuple in the system (i.e., the worst-case performance). It is known that each of these metrics by itself is not enough to fully characterize system performance.

To get a better understanding of system performance, we need to look at both metrics together or, alternatively, we can use a single metric that captures both of these metrics. The most common way to capture the trade-off between the average-case and the worst-case performance is to measure the ℓ_2 *norm* [9]. Specifically, the ℓ_2 norm of response times, R_i, is defined as:

Definition 5 *The ℓ_2 norm of response times for N tuples is equal to $\sqrt{\sum_1^N R_i^2}$.*

The definition shows how the ℓ_2 norm considers the average in the sense that it takes into account all values, yet, by considering the second norm of each value instead of the first norm, it penalizes more severely outliers compared to the average slowdown metric.

Similarly, the ℓ_2 norm of slowdowns, H_i, is defined as:

Definition 6 *The ℓ_2 norm of slowdowns for N tuples is equal to $\sqrt{\sum_1^N H_i^2}$.*

In the following sections, we present our policies for balancing the trade-off between the average and worst cases.

3.2.2.2 Balancing the Trade-off for Slowdown Our proposed *HNR* policy is still biased toward certain classes of queries. These classes are:

1. Queries with high productivity; and/or
2. Queries with low processing cost.

For example, under *HNR*, a query with high cost and low productivity comes at the bottom of the priority list. When the system is overloaded, such low priority query will starve waiting for execution. This behavior may be viewed as being unfair as it yields a system with a high value for the maximum slowdown metric. The *LSF* policy, on the other hand, avoids the starvation of tuples yet yields a poor average-case performance.

In order to balance the trade-off between the average- and worst-case performance, we are proposing a new scheduling policy that minimizes the ℓ_2 norm of slowdowns. We will call this new policy *Balance Slowdown (BSD)*. To understand the intuition underlying *BSD*, we will use the same technique from the previous section but with the objective of minimizing the ℓ_2 norm of slowdowns.

Specifically, consider a policy A where operator segment E_x^i is executed before operator segment E_y^j. The ℓ_2 norm of slowdowns of tuples produced under this policy is:

$$L_A = \sqrt{S_x^i \times (H_{A,i})^2 + S_y^j \times (H_{A,j})^2}$$

where S_x^i, $H_{A,i}$, S_y^j, and $H_{A,j}$ are calculated as in Section 3.1. Similarly, we can compute L_B which is the ℓ_2 norm of slowdowns of tuples produced under policy B. In order for L_A to be less than L_B, then the following inequality must be satisfied:

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

As an approximation, we drop $(2T_j + \overline{C}_x^i)$ and $(2T_i + \overline{C}_y^j)$ from the above inequality which yields to:

$$\frac{S_y^j}{\overline{C}_y^j T_j} \times \frac{W_y^j}{T_j} < \frac{S_x^i}{\overline{C}_x^i T_i} \times \frac{W_x^i}{T_i}$$

Hence, under our proposed policy *BSD*, each operator O_x^k is assigned a priority value V_x^k which is the product of the operator's normalized rate and the current highest slowdown of its pending tuples. That is:

$$V_x^k = \left(\frac{S_x^k}{\overline{C}_x^k T_k}\right)\left(\frac{W_x^k}{T_k}\right) \tag{3.7}$$

Notice that the term $S_x^k/\overline{C}_x^k T_k$ is the normalized output rate of operator O_x^k as defined in (3.4), whereas the term W_x^k/T_k is the current highest slowdown experienced by a tuple in O_x^k's input queue. As such, under *BSD*, an operator is selected either because it has a high weighted rate or because its pending tuples have acquired a high slowdown. This makes our proposed heuristic a hybrid between our previous policy for reducing the average slowdown (i.e., *HNR*) and the greedy heuristic to optimize maximum slowdown (i.e., *LSF*). Comparing the priority used in *BSD* to that used by *HNR*, we find that *BSD* considers the waiting time of tuples, and gives greater emphasis to the cost.

3.2.3 Balancing the Trade-off for Response Time

We use the same observations from above to devise a policy that balances the trade-off between average response time and maximum response time. Specifically, our proposed heuristic for balancing the trade-off under the response time metric is a hybrid of our proposed *HR* policy (that optimizes average response time) and the *FCFS* policy (that optimizes maximum response time). As such, under our proposed *Balance Response Time (BRT)* policy, each operator O_x is assigned a priority value V_x which is defined as:

$$V_x^k = (\frac{S_x^k}{\overline{C}_x^k})\left(W_x^k\right) \tag{3.8}$$

3.3 IMPLEMENTATION ISSUES

At each *scheduling point*, our scheduler is invoked to decide which operator to execute next. The definition of a scheduling point depends on the scheduling level as follows:

- **Query-level Scheduling,** where the scheduling point is reached when a *query* finishes processing a tuple (i.e., non-preemptive)
- **Operator-level Scheduling,** where the scheduling point is reached when an *operator* finishes processing a tuple (i.e., preemptive).

3.3.1 Priority Dynamics under HNR

Under *HNR*, the priority given to each operator is static over time. Thus, the scheduler simply keeps a sorted list of pointers to operators. At each scheduling point, the scheduler traverses the list in order and selects for execution the first operator with pending tuples.

In query-level scheduling, it is sufficient to only keep a list of the priorities of leaf operators where the priority of a leaf operator O_l is basically the normalized output rate of segment E_l.

In operator-level scheduling, the scheduler might decide to proceed with the next operator O_x on the currently executing query or to execute a leaf operator in another query for which new tuples have arrived. As such, it is required to keep a list of the priorities of all operators, where the priority of operator O_x is computed as the normalized output rate of the segment of operators starting at O_x and ending at the root as shown in Section 3.1.

3.3.2 Priority Dynamics under BSD

Recall, the priority of an operator O_x under *BSD* depends on its static normalized output rate and the current slowdown of its pending tuple where the latter increases with time. The increase in the current slowdown for different tuples happens at different rates according to each tuple's current wait time (W) and ideal processing cost (T). As such, the priority of each operator under *BSD* should be re-computed at any instant of time. However, such an implementation renders *BSD* very impractical. An obvious way to reduce such an overhead is

to implement *BSD* using a query-level scheduler; this approximation will reduce the frequency of scheduling points, however it is not enough. For instance, if there are q installed CQs, then at each scheduling point the scheduler will have to compute the priorities for q leaf operators. Next, we describe techniques for an efficient implementation of *BSD*.

3.3.2.1 Search Space Reduction Notice that the priority of an operator under the non-preemptive implementation of *BSD* can be expressed by the product of two components: W_x^k and $S_x^k/(\overline{C}_x^k \times T_k^2)$ where the former is dynamic, while the latter is static. We will denote that static component $S_x^k/(\overline{C}_x^k \times T_k^2)$ as Φ_x.

To reduce the search space, we divide the domain of priorities into *clusters* where each cluster covers a certain range in the priority spectrum. An operator belongs to a cluster if its priority falls within the range covered by the cluster. Then each cluster is assigned a new priority and all operators within a cluster inherit that priority.

Using clustering is a well know technique to reduce the search space for dynamic schedulers. In the particular context of DSMSs, Aurora uses a *uniform* clustering method for its QoS-aware scheduler. However, uniform clustering has the drawback of grouping together operators with large differences in their priorities. For example, if the priority domain is $[1, 100]$ and we want to divide it into 2 clusters, then we will end up with clusters covering the ranges [1,50] and [50,100]. Notice how the ratio between the highest and lowest priority in the second cluster is only 2, whereas that ratio in the first cluster goes up to 50.

In this thesis, we propose to *logarithmically* divide the domain of priorities into clusters, where the priorities of the operators that belong to the same cluster are within a maximum value ϵ from each other. Specifically, the first cluster will cover the priority range $[\epsilon^0, \epsilon^1]$, the second covers $[\epsilon^1, \epsilon^2]$ etc.. In general, a cluster i will cover the priority range $[\epsilon^i, \epsilon^{i+1}]$ where a cluster i is assigned a *pseudo priority* equal to ϵ^i and an operator O_x will belong to cluster i if $\epsilon^i \leq \Phi_x \leq \epsilon^{i+1}$.

The number of resulting clusters depends on ϵ and Δ, where Δ is the ratio between the highest and the lowest priorities in the priority domain. Hence, the number of clusters m is: $m = \frac{log(\Delta)}{log(\epsilon)}$. For example, if the priority domain is $[1, 100]$, then at $\epsilon = 10$, the number of clusters is equal to 2 where the first cluster covers the priorities [1,10] and the second covers

[10,100]. As one can see from this example, the ratio between the highest and lowest priority in each cluster is equal to ϵ (i.e., 10) as opposed to 2 and 50 when using uniform clustering.

Given such a clustering method, when a new tuple arrives, instead of routing it to the input queue of a leaf operator O_l^k, it is routed to the input queue of the cluster that contains O_l^k. Then at each scheduling point, the priority of each cluster is computed using the W of the oldest tuple in the cluster's input queue and the cluster's pseudo priority. Clearly, this "batching" can provide significant savings in computing priorities.

3.3.2.2 Search Space Pruning The clustering method reduces the complexity of the scheduler from $O(q)$ to $O(m)$, however, we can do even better by pruning the search space. Towards this, we use the same method used in the $R\times W$ policy [3] and later generalized by *Fagin's Algorithm (FA)* which quickly finds the exact answer for *top k* queries [26].

FA quickly finds the exact answer for *top k* queries in a database where each object has g grades, one for each of its g attributes, and some aggregation function that combines the grades into an overall grade. *FA* requires that for each attribute there is a sorted list which lists each object and its grade under that attribute in descending order. In this work, we do not present the details of *FA*, but we show how to map our search space to that required by *FA*.

As mentioned above, under *BSD*, our function for computing the priority of an operator cluster is the product of W and its pseudo priority. Hence, the system can keep a list of all clusters sorted in descending order of pseudo priority. Additionally, the system's input queue is already sorted by the tuples' arrival time, which makes it automatically sorted in descending order of wait time with each tuple pointing to its corresponding cluster in the cluster list. At a scheduling point, the two lists are traversed according to *FA* with $k = 1$ (i.e., find the *top 1* answer). The answer returned by *FA* is the cluster with the highest priority which is selected for execution. Note that *FA* will provide the same answer as the one returned by a linear traversal of the list. Hence, the only approximation so far is due to using the clustering method.

3.3.2.3 Clustered Processing Once a cluster is selected for execution, then the tuple at the top of the cluster's input queue is processed by its corresponding query until emitted or discarded (i.e., pipelined and non-preemptive). However, it is often the case that the same tuple is to be processed by more than one query in the system. As such, once a cluster is selected by the scheduler, we execute a complete set of queries Q_c which belong to the selected cluster and they all operate on the head-of-the-queue tuple.

This idea of clustered processing is kind of similar to the *train processing* in Aurora [16] where once a query is selected for execution, it will process a batch of pending tuples. However, each tuple in the same queue will have a different wait time, but in our case, all the queries in the same cluster will have the same pseudo priority which reduces the inaccuracy in the scheduling decision.

Example 2 Figure 4 shows an example that illustrates the three implementation techniques described above. The figure shows two query clusters C_x and C_y together with their pending tuples. It also shows the system's input queue where tuples are sorted according to their wait time W and the clusters list where clusters of queries are sorted according to their static priority Φ. A link between a tuple t and a cluster C means that t is the tuple at the head of C's input queue. Notice that t could be at the head of several input queues at the same time, however, at any point of time, it is only associated with the one cluster that has the highest static priority among these clusters. Finally, the priority of a pair $<t, C>$ is computed using t's wait time W and C's priority Φ as described above.

In this example, we assume that the static priority Φ_x of cluster C_x is higher than the static priority Φ_y of cluster C_y. The figure shows the status of the system's queues right after tuple t_1 has been processed by the queries in cluster C_x. At that moment, tuple t_1 is disassociated from cluster C_x and it is instead, associated with cluster C_y which follows C_x in the priority list. Additionally, tuple t_2 is associated with cluster C_x since it is the tuple currently at the head of C_x's input queue.

Using *FA*, the two lists are searched for the pair that has the highest priority to be executed. If the pair $<t_2, C_x>$ is executed first, then at the next scheduling point, tuple t_3 would be the one associated with C_x. However, if the pair $<t_1, C_y>$ is executed first, then

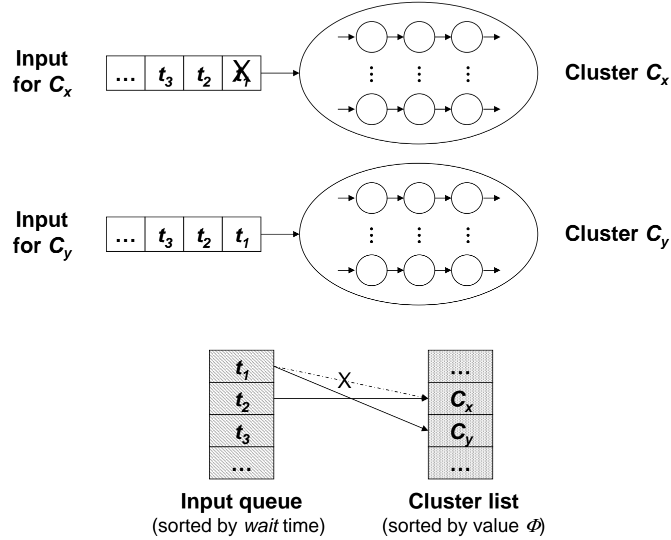

Figure 4: An example that illustrates the different implementation techniques

at the next scheduling point, t_1 would be associated with the next cluster in the cluster list or it would be eliminated from the queue if it has been processed by all clusters.

3.3.3 Adaptive Scheduling

It should be clear that the success of any of our scheduling policies proposed above relies heavily on the DSMS being able to estimate the processing time and the selectivity parameters for each operator. This would enable the scheduler to compute the right priority for each query which in turn would lead to optimizing the desired QoS metric.

The first of these parameters (i.e., the processing time of an operator) is a fairly static parameter which could be estimated once when a CQ is registered and used throughout the lifetime of the CQ. However, the selectivity parameter of an operator could be very dynamic as it depends on the data distribution in the input data stream which may vary significantly over time. For example, in an environment monitoring application, a filter like `where temp < ` $40°F$ will have higher selectivity during the night than during the day. (at least in some parts of the world, including Pittsburgh!).

To circumvent this problem of dynamic selectivity, we propose an *adaptive scheduling* mechanism that enables our proposed policies to deliver the expected performance all the time. Under this mechanism, the DSMS continuously monitors the execution of queries and updates the current priorities of queries based on the new estimations.

Specifically, the DSMS monitors the input and output of query operators over a time window and updates the selectivity of the operator at the end of the window. If the new selectivity is different from the old one, then the operator is assigned a new priority based on the new selectivity. The new selectivity S_{new} assigned to an operator is basically computed as follows:

$$S_{new} = (1 - \alpha) \times S_{old} + \alpha \times \frac{N_O}{N_I}$$

where S_{old} is the current selectivity of the operator and N_O and N_I are respectively, the number of output and input tuples of an operator during the window interval. Finally, α is an aging parameter that determines how much is the weight assigned to the newly observed selectivity as compared to the selectivity currently assigned to an operator.

For instance, if α is set to 0, then the selectivity would never be updated and the system is static. On the contrary, if α is set to 1, then the system always ignores the past and the new selectivity is basically the one that has been observed during the last window. This might lead to a very unstable system especially with a short monitoring window. Hence, a value of α that is greater than 0 and less than one should allow for a stable and and adaptive system. In fact, we found that setting α to 0.125, the same value used in network congestion control mechanisms [32], provides the best performance.

Notice that our mechanism for monitoring and adapting is very similar to the *ticket scheme* used in eddies-based query processing [42]. However, the ticket scheme is basically used for routing tuples between operators rather than scheduling the execution of multiple continuous queries. Specifically, the ticket scheme provides dynamic query plans that can adapt to changes in workload.

Under the ticket scheme, a lottery scheduling mechanism [67] is used where the eddy gives a ticket to an operator whenever it consumes a tuple and takes a ticket away whenever it sends a tuple back to the eddy for further processing. To choose an operator to which a

new tuple should be routed, a lottery is conducted between operators, with the chances of a particular operator winning is proportional to the number of tickets it has acquired. On the one hand, the ticket scheme gives higher priority to operators with low selectivity, which is beneficial for query plan optimization. On the other hand, our proposed policies, generally give higher priority to operators with high selectivity, which is beneficial for multiple query scheduling for improved online performance.

3.4 MULTI-STREAM QUERIES

In this section, we extend our work to handle multi-stream queries which contain *Join* operators and specifically, time-based sliding window joins. To simplify the discussion, we assume *Symmetric Hash Join* (SHJ) [68, 33] which is a non-blocking, in-memory join processing algorithm.

To illustrate the semantics of a time-based sliding window join, let us assume a sliding window continuous query Q that performs a join between two streams M_l and M_r with a window interval V. Each tuple that arrives at the system has a *timestamp* which is either assigned by the data source or the DSMS. For such a query Q, when a tuple t arrives at stream M_l, it will be compared against the tuples from M_r that are within V time units from t's timestamp [6, 15]. Out of those tuples, the ones that satisfy the join predicate are streamed up the query plan.

To use SHJ for performing the join operation in the query described above, hash tables HT_l and HT_r are defined over streams M_l and M_r, respectively. As a tuple t with timestamp $t.ts$ arrives at one of the streams (say M_l), it is first hashed and inserted into HT_l, then the hash value is used to probe HT_r for tuples with matching key. Out of those matching tuples, each tuple that satisfies the window predicate is concatenated to the input tuple t and a new *composite* tuple is generated.

3.4.1 Metrics For Joins

Next, we extend the metrics described in Section 3.1 for composite tuples generated by multi-stream queries.

3.4.1.1 Response Time of Joined Tuples Definition 1 can be used directly to measure the response time of a composite tuple as long as the arrival time is defined. This arrival time is easily defined by considering the dependency between the two joined tuples. That is, the composite tuple cannot be generated until the arrival of the second one (similarly to [6]). In other words, the composite tuple "inherits" the arrival timestamp of the latest of

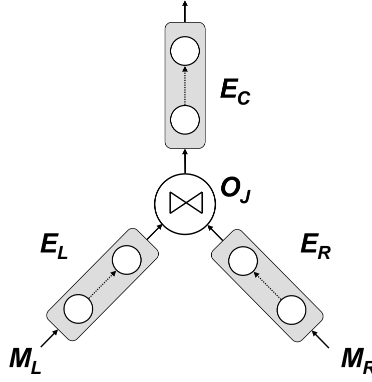

Figure 5: An example of a multi-stream query plan

the tuples used to create it. Hence, the arrival time is defined as follows:

Definition 7 *The arrival time A_i of a composite tuple t_i that is produced from concatenating two tuples t_l and t_r with arrival times A_l and A_r respectively is equal to $max(A_l, A_r)$.*

Thus, the response time R_i for tuple t_i is $R_i = D_i - A_i$, where D_i is the tuple output time and A_i is the arrival time.

3.4.1.2 Slowdown of Joined Tuples In order to measure the slowdown of a composite tuple produced by a multi-stream query Q_k, we first need to identify the ideal processing time T_k incurred by such a tuple. For simplicity, in this section, we drop the query identifier from our notation. To compute T_k, let us consider a query consisting of four components (Figure 5):

1. A join operator (O_J)
2. A left operator segment preceding the join operator (E_L)
3. A right operator segment preceding the join operator (E_R), and
4. A common operator segment following the join operator down to the query root (E_C).

Each of those segments might consist of one or more operators. In the simplest case, when each segment is composed of one operator, the query plan looks like Q_1 or Q_2 in Figure 2.

A tuple that is generated by such a query is the result of concatenating two tuples t_l and t_r received from the left and right inputs, respectively. The tuple t_l is first processed by E_L, then at O_J, the hash, insert, and probe operations are performed on t_l. Similarly, t_r is processed by E_R and O_J. Ultimately, the concatenated tuple generated by the join is processed by E_C. Hence, the ideal processing time of a composite tuple is defined as follows:

Definition 8 *The ideal processing time T_k of a composite tuple processed by a multi-stream query Q_k composed of join operator O_J, a left segment E_L, a right segment E_R, and a common segment E_C is defined as:*

$$T_k = C_L + C_R + (2 \times C_J) + C_C$$

where C_L, C_R, C_J, and C_C are the ideal total processing costs of the operators in E_L, E_R, O_J, and E_C respectively.

To compute the slowdown of a tuple it is important not to penalize the DSMS for the *dependency delay.* That is, the time that the first tuple has to spend waiting for the arrival of its matching tuple. As such, we define the slowdown incurred by a composite tuple t_i produced by a multi-stream query Q_k as follows:

$$H_i = 1 + \frac{D_i^{actual} - D_i^{ideal}}{T_k}$$

where D_i^{actual} is the actual departure time of the composite tuple which includes: 1) processing time; 2) dependency delay; and 3) queuing delay, whereas D_i^{ideal} is the ideal departure time of the composite tuple if it were the only tuple in the system and it includes all the components in D_i^{actual} except for the queuing delay.

3.4.2 Scheduling Multi-stream Queries

In order to solve the problem of scheduling multi-stream queries, we follow the same technique as in [65, 6], where we reduce the problem to that of scheduling individual segments. Specifically, we view a multi-stream query as a set of disjoint virtual single-stream queries and assign a priority value to each operator in these virtual queries.

However, computing such priorities requires global knowledge about the selectivity of the multi-stream query. Specifically, we need to re-define the prioritizing parameters S_x and $\overline{C}_x$ in the presence of windowed-join operators. As such, let us consider a multi-stream query Q which contains a join operator O_J and operator segments E_L, E_R, and E_C as shown in Figure 5. Further, assume that the selectivities of the operators in Q are known, hence, we can compute the segments' global selectivities S_L, S_R, and S_C. Finally, assume that data arrives at the left and right streams with mean inter-arrival times τ_l and τ_r, respectively and that the query performs a time-based windowed join where the window interval is denoted by V time units.

For scheduling, we view the above query as two operator segments E_{LL} and E_{RR} where $E_{LL} =< E_L, O_J, E_C >$ and $E_{RR} =< E_R, O_J, E_C >$. For simplicity, we assume we are implementing a non-preemptive scheduling policy; as such, it is sufficient to compute the priority values for the leaf operators in E_{LL} and E_{RR}. Let O_x be the leaf operator in E_{LL}, then the parameters S_x and $\overline{C}_x$ are defined as follows:

- **Global Selectivity** S_x is the number of tuples produced due to processing one tuple down segment E_{LL} and is defined as follows:

$$S_x = S_L \times S_J \times (S_R \times \frac{V}{\tau_R}) \times S_C$$

  where $(S_R \times \frac{V}{\tau_R})$ estimates the number of tuples present in hash table HT_r at any point of time (as in [33, 6]).

- **Global Average Cost** $\overline{C}_x$ is the expected time required to process an input tuple along segment E_{LL} and is defined as:

$$\overline{C}_x = C_L + (S_L \times C_J) + (S_L \times S_J \times S_R \times \frac{V}{\tau_R} \times C_C)$$

where the first two terms define the cost for processing the input tuple, and the third term is the cost for processing all the tuples generated by concatenating the input tuple with the matching tuples in HT_r.

Using the above parameters as well as the total processing time parameter computed in Definition 8, we set the priority of each operator by substitution in the prioritizing function corresponding to the used scheduling policy (i.e., *HR*, *HNR*, *BSD*, or *BRT*) as defined in Equations 3.1, 3.4, 3.7, and 3.8 respectively. For multi-stream queries with multiple join operators, the above parameters are defined recursively.

3.5 AGGREGATE CONTINUOUS QUERIES

In this section, we propose policies for scheduling multiple time-based sliding window aggregate continuous queries. In such queries, a *Sliding Window*, is defined by means of two attributes: 1) *RANGE*; and 2) *SLIDE*. *RANGE* defines the window length, whereas *SLIDE* defines how the window boundaries move over the data stream. Every *SLIDE* interval, the set of tuples that arrived within the last *RANGE* interval is processed to produce a new aggregate value.

Given the above semantics of aggregate continuous queries, the *SLIDE* attribute acts like a *deadline* at which the new aggregate value should be produced. However, it is not always possible for the DSMS to produce each aggregate query result at its specified deadline, especially when the system is overloaded. This highlights the need for a mechanism to schedule the processing of multiple aggregate queries with the objective of minimizing *tardiness*. Tardiness, or lateness, is basically, the amount of elapsed time between the deadline (i.e., every SLIDE time) and the instant when the result is actually generated.

Towards minimizing tardiness, we studied the performance of two scheduling policies: (1) *Earliest Deadline First (EDF)*; and (2) *Highest Rate (HR)* (Section 3.1.1.1). Our study showed that HR outperforms EDF at higher system utilizations, whereas EDF is a better policy when the system is lightly loaded. This tension between the two policies motivated us to propose a new hybrid policy that integrates the benefits of EDF and HR and it adapts itself automatically to the workload.

In the next sections, we will first explain the advantages and disadvantages of each of the EDF and HR policies when used for scheduling aggregate queries, then we will describe our proposed hybrid policy for scheduling the execution of continuous aggregate queries.

3.5.1 EDF vs. HR for Scheduling Aggregate CQs

Typically, EDF [41] guarantees that all jobs will meet their deadlines if the system utilization is less than or equal to 1.0. In the context of aggregate continuous queries, this means that if the system is under-utilized, then each aggregate value will be generated at the specified

instant of time (i.e., meeting its deadline as specified by the SLIDE attribute). As such, the tardiness of the system is expected to be zero since all the generated results make the deadline.

When the system is over-utilized, it is impossible to generate all the aggregate results at the specified deadlines. That is when some results are delayed beyond their deadlines experiencing tardiness. Using an EDF scheduler in such over-utilization situations will have a substantial negative impact on the overall system tardiness. This negative impact is known as the "domino effect". In such cases, EDF gives a high priority to a query with an early deadline that it has already missed instead of scheduling another query which has a later deadline that could still be met.

In contrast to EDF, HR is the best policy to use when the system is over-utilized, or to be more specific, it is the best policy to use when all queries have already missed their deadlines. The reason is that if all queries have already missed their deadlines, then tardiness and latency become the same metric and we have already shown earlier that HR is the one policy that is capable of minimizing latency. However, generally, at each instant of time, the workload is a mix of queries that have missed their deadlines as well as queries that have not missed their deadlines yet. In that case of mixed queries, HR might run into the problem of giving a high priority to a query with a high output rate though it has missed its deadline as opposed to another query with a low output rare that could still meet its deadline.

3.5.2 Hybrid Policy for Scheduling Aggregate CQs

From the discussion above, it is clear that there is no clear winning policy for scheduling aggregate CQs. Generally speaking, EDF does well at low utilization, whereas at high utilization, HR does better than EDF. In this section, we propose a hybrid policy that combines the advantages of EDF and HR. The proposed hybrid policy is *parameter-free* and it automatically adapts to the system load. This results in a performance that is better than the one exhibited by EDF at low utilization, while at high utilization its performance is better than that of HR.

Under our proposed hybrid policy, the scheduler maintains two priority lists. In the

first list, called *ListEDF*, queries are ordered according to their deadlines as in the EDF scheduling policy. In the second list, called *ListHR*, queries are ordered according to their output rates as in the HR scheduling policy.

The first list, *ListEDF*, contains all queries that can still make their deadlines. Formally, a query Q_i with deadline D_i is included in *ListEDF* if and only if, $t + C_i \leq D_i$, where t is the current time and C_i is the time to process all the tuples that arrived before the deadline D_i.

The second list, *ListHR*, contains all queries that missed their deadlines. Formally, a query Q_i with deadline D_i is included in *ListHR* if and only if, $t + C_i > D_i$, where t and C_i are defined as above. Notice that each query starts in the *ListEDF* then it might move to the *ListHR* if it misses its deadline.

Given the above two lists, at each scheduling point, our hybrid policy selects for execution either the query at the top of *ListEDF* or the one at the top of *ListHR*. For convenience, we will call these two queries, $Q_{1,EDF}$ and $Q_{1,HR}$, respectively.

To decide between $Q_{1,EDF}$ and $Q_{1,HR}$, we use a a simple greedy heuristic under which, $Q_{1,HR}$ is scheduled for execution if $t+C_{1,HR} < D_{1,EDF}$, otherwise, $Q_{1,EDF}$ is the one scheduled for execution, where $C_{1,HR}$ if the processing time of $Q_{1,HR}$ and $D_{1,EDF}$ is the deadline of $Q_{1,EDF}$.

It should be clear that given the above arrangement, at the extreme case if all queries are past their deadlines, then the hybrid policy is basically equivalent to HR. In the other extreme case where all queries can meet their deadlines, then the hybrid policy behaves like EDF. In the general case, where there is a mix of queries that have passed their deadlines and others that can still meet their deadlines, our hybrid policy employs both HR and EDF. This allows our proposed hybrid policy to outperform HR and EDF as it is experimentally shown in Section 3.8.1.13.

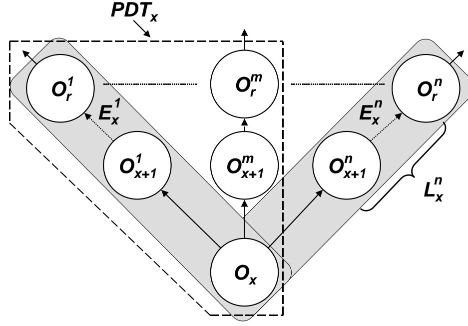

Figure 6: Multiple CQs plans sharing operator O_x

3.6 OPERATOR SHARING

Operator sharing eliminates the repetition of similar operations in different queries. Hence, a multi-query scheduler should exploit those shared operators for further optimization. In this section we show how to set the priority of a shared operator under our proposed policies.

First, let us consider a set of operator segments SE_x in which operator O_x is shared among multiple operator segments $E_x^1, E_x^2, ..., E_x^n$ (Figure 6) where for each segment E_x^i, we can compute the selectivity S_x^i and the average cost $\overline{C}_x^i$.

Further, assume that the cost of the shared operator O_x is c_x and $\overline{SC}_x$ is the average cost of executing the set of segments SE_x. Intuitively, $\overline{SC}_x$ is equal to the total average cost of executing the N segments with the cost of the shared operator O_x counted only once. Formally, the average cost $\overline{SC}_x$ of N paths sharing an operator O_x is:

$$\overline{SC}_x = \sum_{i=1}^{N} \overline{C}_x^i - \sum_{i=1}^{N-1} c_x$$

where $\overline{C}_x^i$ is the average cost of segment E_x^i and c_x is the cost of the shared operator O_x.

3.6.1 HNR with Operator Sharing

In this section, we will describe the general method for setting the priority of a shared operator under *HNR*. In the next section, we will describe the particular details of this

method. Note that the *BSD* policy can also be extended in the same way, however the details are eliminated for brevity.

To set the priority of a shared operator under the *HNR* policy, consider two sets of operator segments SE_p and SE_q, where $SE_p = \{E_p^1, ..., E_p^N\}$ sharing operator O_p and $SE_q = \{E_q^1, ..., E_q^M\}$ sharing operator O_q. For now, assume that if a set of segments is scheduled, then all the segments within that set are executed.

To measure the impact of executing one set on the other, we will use the same concept from the definition of Inequality 3.3. Basically, we will measure the increase in slowdown incurred by the tuples produced from one set if the other set is scheduled for execution first. Hence, if the set of segments SE_p is executed first, then the increase in slowdown incurred by tuples from SE_q is computed as follows:

$$H_q = S_q^1 \frac{\overline{SC}_p}{T_{q,1}} + S_q^2 \frac{\overline{SC}_p}{T_{q,2}} + ... + S_q^M \frac{\overline{SC}_p}{T_{q,M}}$$

where $\overline{SC}_p$ is the amount of time that set SE_q will spend waiting for set SE_p to finish execution and $T_{q,i}$ is the ideal total processing time for the tuples processed by E_q^i.

Similarly, we can compute H_p which is the increase in slowdown incurred by tuples from SE_p. In order for H_q to be less than H_p, then the following inequality must be satisfied:

$$\overline{SC}_p \sum_{i=1}^{M} \frac{S_q^i}{T_{q,i}} < \overline{SC}_q \sum_{i=1}^{N} \frac{S_p^i}{T_{p,i}}$$

Hence, the priority of a set of operator segments SE_x that consists of N segments sharing a common operator O_x is:

$$V_x = \frac{\sum_{i=1}^{N} \frac{S_x^i}{T_{x,i}}}{\overline{SC}_x} \qquad (3.9)$$

3.6.2 Priority-Defining Tree (PDT)

Setting the priority of a shared operator using all the N segments in a set is only beneficial if it maximizes the value of Equation 3.9. However, that is not always the case because Equation 3.9 is non-monotonically increasing. That is, adding a new segment to the equation might increase or decrease its value.

We definitely need to boost the priority of a shared operator, however, we do not want segments with low normalized rate to hurt those with high normalized rate by bringing down the overall priority of the shared operator. As such, we need to select from each set what we call a *Priority-Defining Tree (PDT)* which is the subset of segments that maximizes the aggregated value of the priority function. Hence, the priority of a shared operator is basically the priority of that PDT and once a shared operator is scheduled, the segments in the PDT are executed as one unit (unless it is preempted).

In order to compute the priority value V_x for operator O_x, we sort the segments according to their priority. Then, we visit the segments in descending order of priority, and only add a segment to the priority defining tree of O_x (PDT_x) if it increases the aggregate priority value, otherwise we stop and the shared operator O_x is assigned that aggregate priority value. Hence, for an operator O_x shared between N segments, with a PDT_x that is composed of m segments where $m \leq N$, the priority of O_x under the *HNR* policy is defined as:

$$V_x = \frac{\sum_{i=1}^{m} \frac{S_x^i}{T_{x,i}}}{\sum_{i=1}^{m} \overline{C}_x^i - \sum_{i=1}^{m-1} c_x}$$

If $m = N$, that is, if the PDT consists of all the segments sharing O_x, then V_x is equal to the global normalized rate as defined in Equation 3.9.

For any operator segment E_x^i that does not belong to PDT_x, such segment can be viewed as two components: O_x and L_x^i (as shown in Figure 6). Executing PDT_x will naturally lead to executing the O_x component of E_x^i. Scheduling L_x^i for execution depends on its priority which is computed in the normal way using its normalized rate as in Section 3.1. Hence, for example, in a query-level implementation of the *HNR* scheduler, the priority list will contain all the leaf operators in addition to the first operator in each segment that does not belong to any PDT.

3.7 EVALUATION TESTBED

To evaluate the performance of the proposed algorithms, we created a DSMS simulator with the following properties.

Queries: We simulated a DSMS with 500 registered continuous queries. The structure of the query is the same as in [19, 42] where each query consists of three operators: select, join and project. For the experiments on single-stream queries, we assume a join with a stored relation; for multi-stream queries we use window join between data streams.

Streams: We used the *LBL-PKT-4* trace from the *Internet Traffic Archive*[1] as our input stream. The trace contains an hour's worth of wide-area traffic between the Lawrence Berkeley Laboratory and the rest of the world. This trace gives us a realistic data arrival pattern with On/Off traffic which is typical of many applications.

Selectivities: In order to control the selectivity, we added two extra attributes to each packet in the trace and assigned each attribute a uniform value in the range [1,100]. Then the selectivity of the select and join operators is uniformly assigned in the range [0.1,1.0] by using predicates defined on the introduced attributes. Since the performance of a policy depends on its behavior toward different classes of queries, where a query class is defined by its global selectivity and cost, we chose to use the same selectivity for operators that belong to the same query. This enables us to control the creation of classes in a uniform distribution to better understand the behavior of each policy (e.g., Figure 14).

Costs: Similar to selectivity, operators that belong to the same query have the same cost, which is uniformly selected from five possible classes of costs. The cost of an operator in class i is equal to: $K \times 2^i$ time units, where $i \in [0,4]$ and K is a *scaling factor* that is used to scale the costs of operators to meet the simulated utilization (or load). Specifically, we measure the average inter-arrival time of the data trace, then we set K so that the ratio between the total expected costs of queries and the inter-arrival time is equal to the simulated utilization.

Policies: We compared the performance of our proposed policies to the two-level scheduling

[1] http://ita.ee.lbl.gov/html/contrib/LBL-PKT.html

scheme from Aurora where *Round-Robin (RR)* is used to schedule queries and *Rate-based (RB)* is used to schedule operators within a query. Collectively, we refer to the Aurora scheme in our experiments as *RR*.

We also considered the *SRPT* policy where the priority of an operator segment is inversely proportional to its total ideal processing time, as well as the *Chain* scheduling policy [6] which minimizes memory usage.

Here is a list of the rest of the policies considered in our experiments:

- **FCFS:** First Come First Served policy for minimizing maximum response time (Section 3.2.1).
- **LSF:** Longest Stretch First ploicy for minimizing maximum slowdown (Section 3.2.1).
- **HR:** Highest Rate policy for minimizing average response time (Section 3.1.1.1).
- **HNR:** Highest Normalized Rate policy for minimizing average slowdown (Section 3.1.3).
- **BRT:** Balance Response Time policy for minimizing ℓ_2 norm of response times (Section 3.2.3).
- **BSD:** Balance Slowdown policy for minimizing ℓ_2 norm of slowdowns (Section 3.2.2.2).

Table 3: Simulation Parameters for QoS Experiments

Parameter	Value
Base-case policies	RR, SRPT, Chain
Adopted policies	FCFS, LSF, HR, HNR, BRT, BSD
Queries	500 3-operator queries
Operator cost	$K \times 2^0 - K \times 2^4$ Secs
Operator selectivity	0.1 – 1.0
Window interval	1 – 10 Secs
System Utilization	0.1 – 0.99

Table 3 summarized the simulation parameters described above.

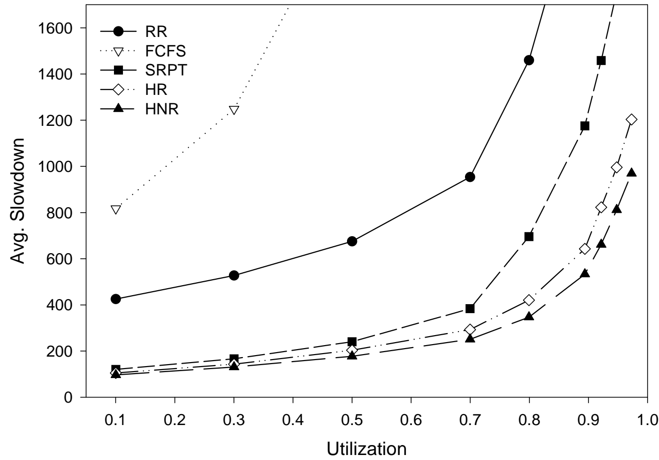

Figure 7: [§3.8.1.1] Avg. slowdown vs. system load

3.8 EXPERIMENTS

In this section, we present the performance of our proposed policies under the different QoS metrics. We also present results on the implementation of the *BSD* policy as well as the performance of the *PDT* strategy for scheduling shared operators.

3.8.1 Performance under Different Metrics

In this section, we present the performance of our proposed policies under the different QoS metrics.

3.8.1.1 Average Slowdown Figure 7 shows how average slowdown increases with utilization. Clearly, *HNR*, our proposed policy, provides the lowest slowdown followed by *HR*. For instance at 0.7 utilization, the slowdown provided by *HNR* is 74% lower than that of *RR*, 51% lower than *SRPT*, and 18% lower than *HR*. At 0.97 utilization, *HNR* is 75% lower than *RR*, 53% lower than *SRPT*, and 20% lower than *HR*.

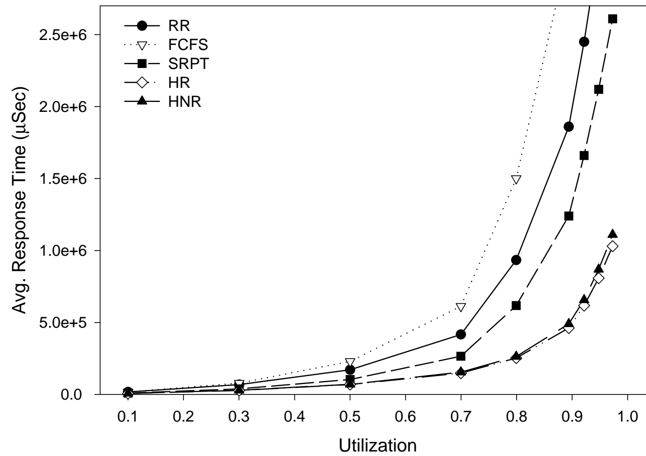

Figure 8: [§3.8.1.2] Avg. response vs. system load

3.8.1.2 Average Response Time As expected, this improvement in slowdown by *HNR* would lead to an increase in response time compared to *HR* as shown in Figure 8. For instance, at 0.7 utilization, *HNR*'s response time is 4% higher than *HR* and it is 7% higher at 0.97 utilization.

3.8.1.3 Maximum Response Time In terms of worst-case performance, Figure 9 shows that FCFS provides the lowest maximum response time which is 75% lower than HR at 0.97 utilization. However, that improvement comes at the expense of poor average-case performance as shown in Figure 8 where the average response time provided by FCFS is 630% that of HR.

3.8.1.4 Maximum Slowdown Similar to FCFS, Figure 10 shows that *LSF* reduces the maximum slowdown by 80% compared to *HNR*. However, that improvement comes at the expense of poor average-case performance (as depicted in Figure 11).

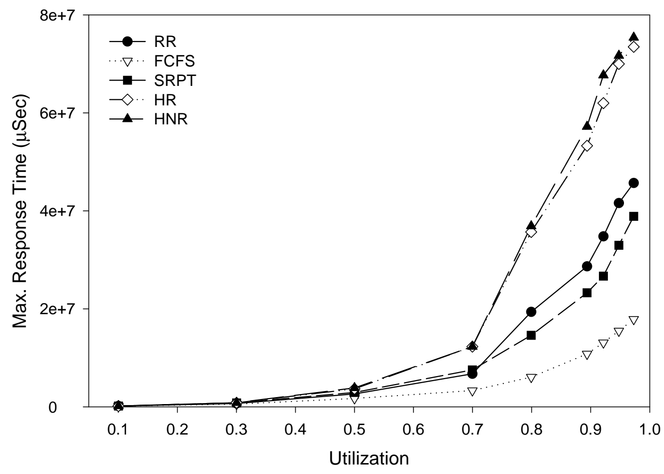

Figure 9: [§3.8.1.3] Max. response time vs. system load

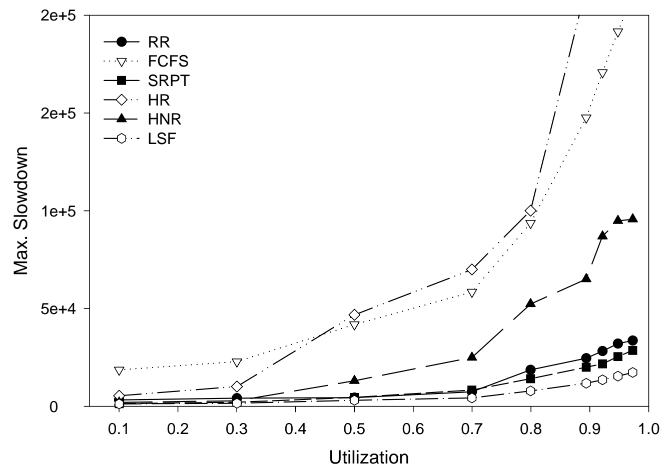

Figure 10: [§3.8.1.4] Max. slowdown vs. system load

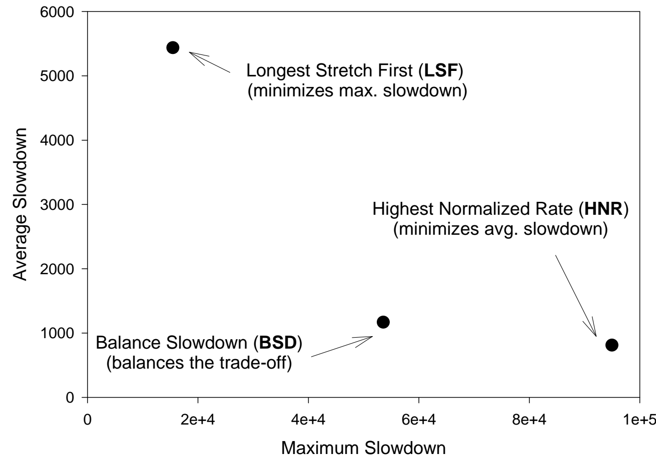

Figure 11: [§3.8.1.5] Max. vs Avg. Slowdown for HNR, LSF, and BSD

3.8.1.5 Trade-off in Slowdown Figure 11 shows that *BSD* can strike the fine balance between average slowdown and maximum slowdown. For instance, as shown in Figure 11, at 0.95 utilization, *BSD* decreases the maximum slowdown by 44% compared to *HNR* while decreasing the average slowdown by 80% compared to *LSF* under the same utilization.

3.8.1.6 Second Norm of Slowdowns As mentioned above, the trade-off between average and maximum slowdowns is easily captured using the ℓ_2 metric. Figure 12 shows the ℓ_2 norm of slowdowns as the utilization of the system increases. The figure shows that *BSD* reduces the ℓ_2 of slowdowns by up to 57% compared to *LSF* and by 24% compared to *HNR*.

3.8.1.7 Second Norm of Response Times Similar to *BSD*, *BRT* reduces the ℓ_2 norm of response times by up to 51% compared to *FCFS* and 23% compared to *HR* as shown in Figure 13.

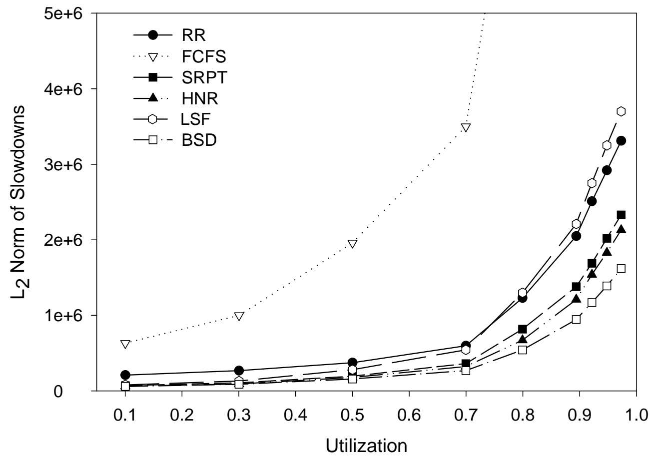

Figure 12: [§3.8.1.6] ℓ_2 of slowdowns vs. system load

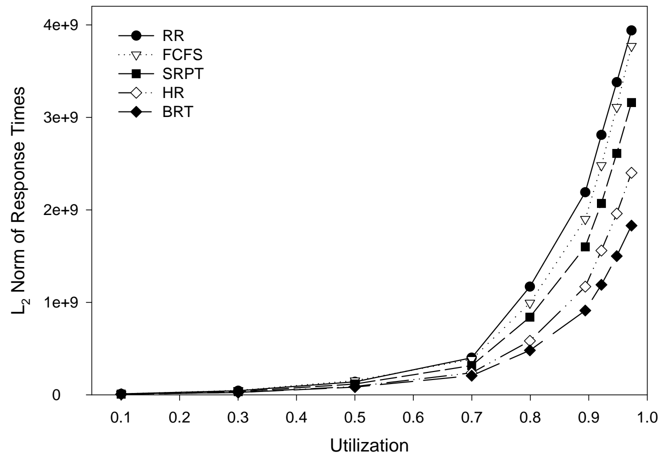

Figure 13: [§3.8.1.7] ℓ_2 of response times vs. system load

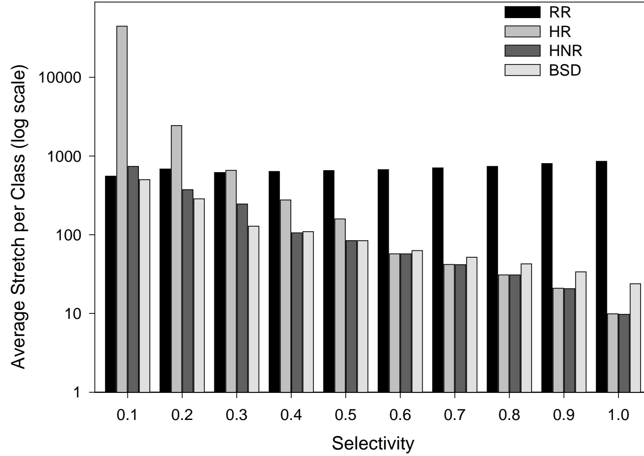

Figure 14: [§3.8.1.8] Slowdown per class for low-cost queries

3.8.1.8 Slowdown per Class To get better insight into the behavior of the different policies toward different classes of queries, we split the workload into distinct classes (as suggested in [2]). Tuples belong to the same class if they were processed by operators with similar costs and selectivities. In Figure 14, we show the slowdown of tuples processed by the class of low-cost queries (i.e., queries where an operator cost is $K \times 2^0$) and different selectivities. The figure shows how *HR* is unfair toward the low-selectivity queries which leads to significant increase in the slowdown of the tuples processed by those queries. *HNR* is still biased toward high-selectivity queries, yet less than *HR*. Similarly, *BSD* is less biased than *HNR*. That balance allowed *BSD* to provide the best ℓ_2 norm of slowdowns as shown in Fig. 12.

3.8.1.9 Impact of Selectivity To further study the impact of selectivity, we conducted an experiment where we assigned the same cost to all operators while varying the maximum value of selectivity assigned to an operator. For instance, if the maximum selectivity is set to 1.0, then the selectivity value assigned to an operator is uniformly distributed in the range [0.1,1.0], whereas if the maximum is 0.5, then the selectivity value assigned to an operator is uniformly distributed in the range [0.1,0.5] etc.

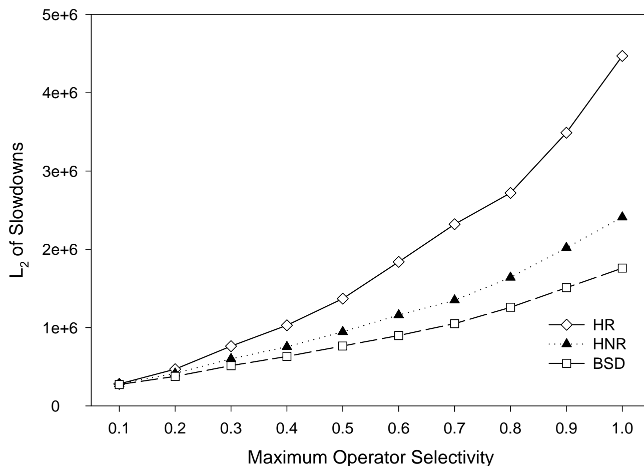

Figure 15: [§3.8.1.9] ℓ_2 of slowdown vs. maximum operator selectivity

Figure 15 shows the ℓ_2 norm of slowdowns for the setting described above. The figure shows that when the maximum selectivity is 0.1, then *HR*, *HNR*, and *BSD* provide almost the same performance since all operators will have the same selectivity of 0.1. As the maximum value of selectivity increases, HR will favor queries with higher selectivity over those with lower selectivity resulting in a high ℓ_2 norm of slowdowns compared to HNR and BSD. The figure shows that BSD always provides the best performance since it considers both the ideal processing time of a query as well as the age of its pending tuples. For instance, when the maximum selectivity is 0.5, BSD reduces the ℓ_2 norm of slowdowns by 44% compared to HR and by 19% compared to HNR; at a maximum of 1.0, the ℓ_2 norm is reduced by 61% compared to HR and by 27% compared to HNR.

3.8.1.10 An Oracle Scheduling Policy In order to validate our general strategy of using output rate (or normalized output rate) for multiple CQ scheduling, we introduce what we call an *oracle* scheduling strategy. The oracle strategy has the ability to "peek" into an input tuple and determine if it will generate an output event or if it will be discarded.

Clearly, the oracle strategy is not implementable in a real system as it requires processing the input stream in the same way continuous queries do. As such, we are introducing this

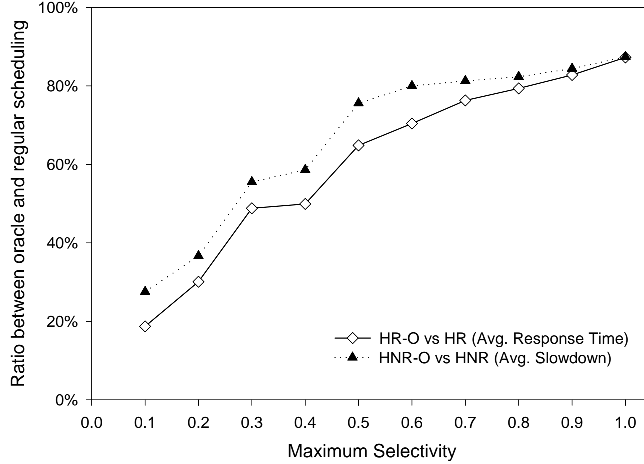

Figure 16: [§3.8.1.10] Performance of an oracle scheduling policy

strategy only for the sake of comparison, since it takes the "guess" out of the scheduling decision. Specifically, at each scheduling point, the oracle strategy is able to compute the exact output rate of a query as opposed to its expected output rate as computed by our proposed policies.

As an example, consider a continuous query Q with 5 pending tuples, where only the 5th one is an event. Under regular scheduling, each of the 5 inputs is an event with probability S (which is the selectivity of the query), whereas under the oracle strategy, only the 5th tuple is an event with probability 1.0 and the other tuples are known to be discarded. Given this information, the oracle can compute the instantaneous output rate of Q as 1.0 (the number of tuples produced) divided by the amount of time needed to process the 5 pending tuples.

Clearly, the oracle strategy has the advantage of not relying on selectivity estimation in making the scheduling decision. This is especially beneficial when the expected selectivity deviates significantly from the exhibited one. This is illustrated in Figure 16, where we plot the ratio in performance between regular policies (*HR* and *HNR*) and oracle policies (*HR-O* and *HNR-O*) while increasing the maximum selectivity in the system (as in the previous experimental setup). The figure shows that at low maximum selectivity (i.e., 0.1), the oracle can improve the performance by up to 80%. As the maximum selectivity increases, the gains

decrease and drop to 12% when the maximum selectivity is 1.0. The reason is that at low selectivity there is a higher chance that an input tuple is not an event. However, only the oracle knows accurately if it is an event or not, which allows it to make a better decision. As the maximum selectivity increases, there are more queries in the system with high selectivity which means that there is a higher chance that a regular policy's guess about a tuple being an event is correct. This brings the performance of regular policies close to the oracle at higher values of maximum selectivity.

Given the above comparisons, it is clear that, in general, using variants of output rate is the right strategy to schedule CQs. However, the exhibited gains in performance depend on the accuracy in computing the rate as illustrated in Figure 16.

3.8.1.11 Performance over Time Figures 17, 18, 19, and 20 show the performance of different scheduling policies over simulation time in the interval from 10^8 to $4 \times 10^8 \mu sec$. The figures show that our proposed policies provide the best performance over time for each of the optimization metrics especially at peak times where traffic is more bursty.

3.8.1.12 Second Norm for Multi-stream Queries *BSD* also provides the lowest ℓ_2 norm of slowdowns for multi-stream queries as shown in Figure 21. In this experimental setting, we generated a workload where queries receive input tuples from 2 data streams, generated following Poisson arrival. In this workload, the costs and selectivities of the operators are assigned uniformly as before and the windows are in the range of 1 to 10 secs. Figure 21 shows that *BSD* improves the ℓ_2 norm by up to 14% compared to *HNR*.

It is also interesting to notice the large improvement offered by *BSD* over policies like *RR* and *FCFS*. For instance, at 0.9 utilization, *BSD* improves the performance 17 times compared to *RR*, and by 15 times compared to *FCFS*. The reason is that *RR* and *FCFS* do not exploit selectivity which plays a more significant role in the case of multi-stream queries where the selectivity of the join operator often exceeds 1.0.

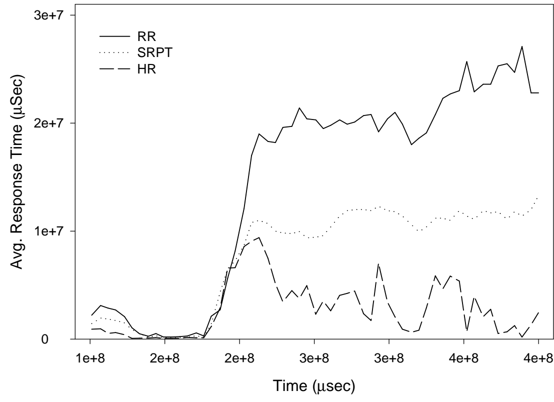

Figure 17: [§3.8.1.11] Response time over time

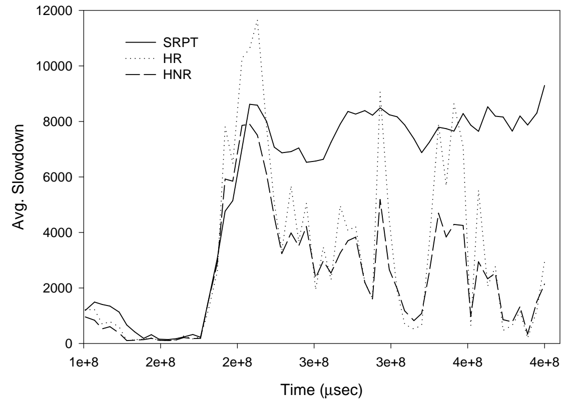

Figure 18: [§3.8.1.11] Slowdown over time

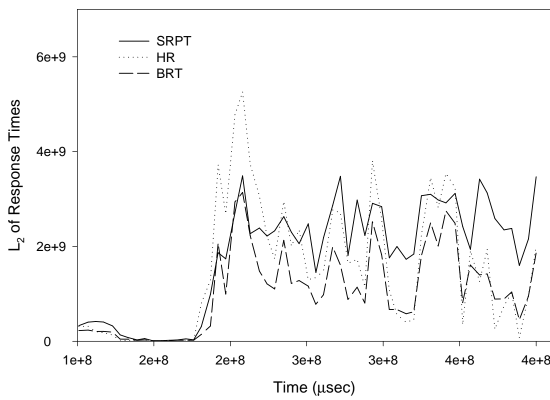

Figure 19: [§3.8.1.11] ℓ_2 of response times over time

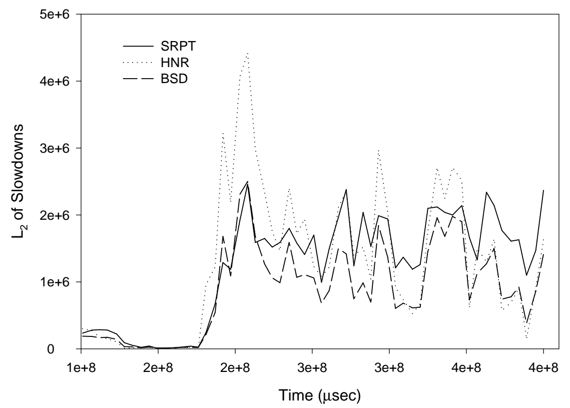

Figure 20: [§3.8.1.11] ℓ_2 of slowdowns over time

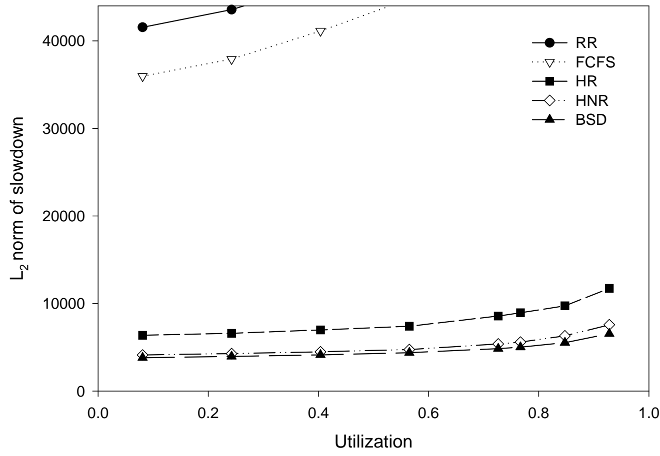

Figure 21: [§3.8.1.12] ℓ_2 of slowdown for multi-stream queries

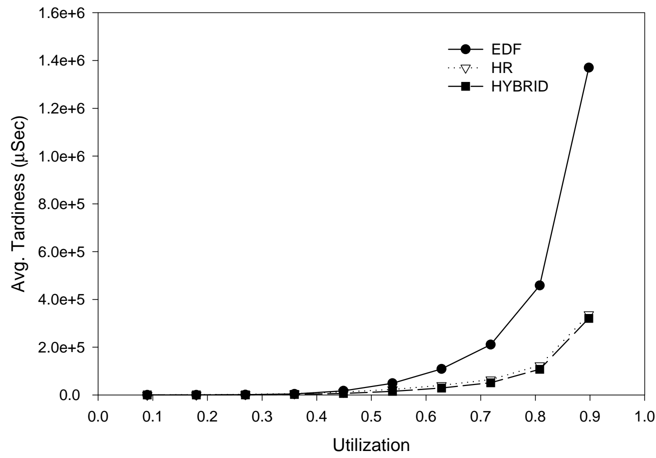

Figure 22: [§3.8.1.13] Tardiness of aggregate CQs

3.8.1.13 Tardiness of Aggregate CQs In this workload, we generated aggregate CQs with random values for the window RANGE and SLIDE parameters. Figure 22 shows that our hybrid policy constantly outperforms the EDF and HR policies. This is further depicted in Figures 23 and 24. Figure 23 shows the improvement in perfomance compared to EDF at low utilization, whereas Figure 24 illustrates the performance at high utilization.

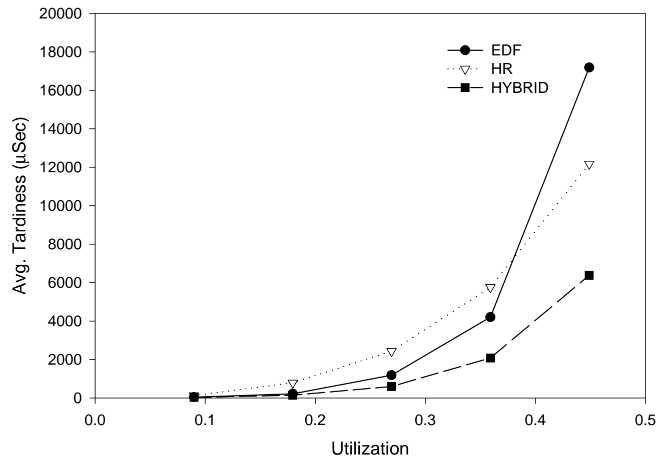

Figure 23: [§3.8.1.13] Tardiness of aggregate CQs at low utilization

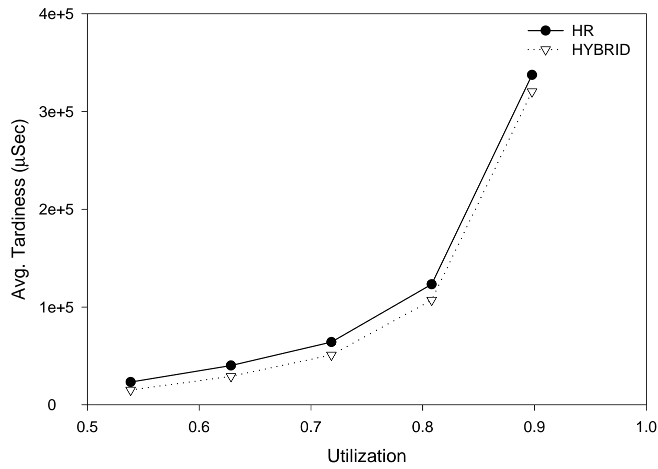

Figure 24: [§3.8.1.13] Tardiness of aggregate CQs at high utilization

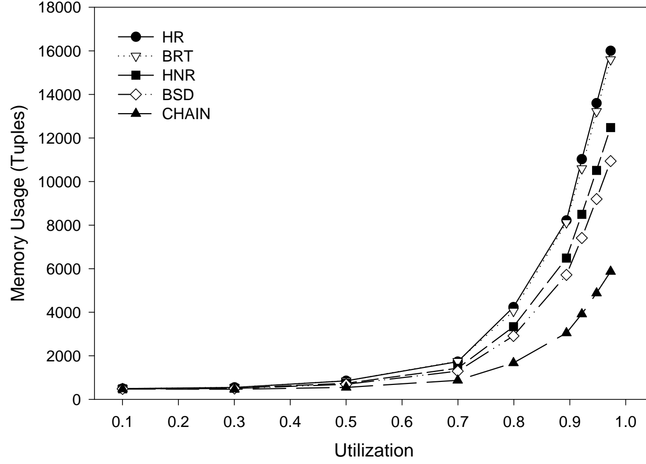

Figure 25: [§3.8.2] Memory usage vs. system load

3.8.2 Memory Usage

Besides CPU, memory is another resource that needs to be considered in a DSMS. For this reason, we also studied the memory requirements of each of the proposed scheduling policies.

Figure 25 shows the average memory usage of our proposed policies, along with that of the *Chain* policy [6] that was designed to minimize memory usage; we are including Chain as a yard-stick for comparison.

Figure 25 shows how policies that optimize for slowdown (i.e., the *HNR and BSD*) reduce the memory usage compared to those that optimize for response time (i.e., the *HR and BRT*). For instance, *HNR* reduces the memory usage by up to 22% compared to *HR*. Both *HNR* and *HR* give higher priorities to queries with low processing cost. Similarly, those queries get higher priorities under policies that optimize for memory usage like *Chain*, since tuples that belong to such queries will spend a short time in memory.

When it comes to selectivity, *Chain* gives higher priorities to queries with low selectivities, or in other words, to queries whose input tuples have a higher chance to be discarded, since executing these queries will consume more input tuples than generating new output tuples. On the contrary, *HR* gives those queries low priority since they will generate very few output events. Meanwhile, since *HNR* emphasizes processing cost, it will boost the priority value

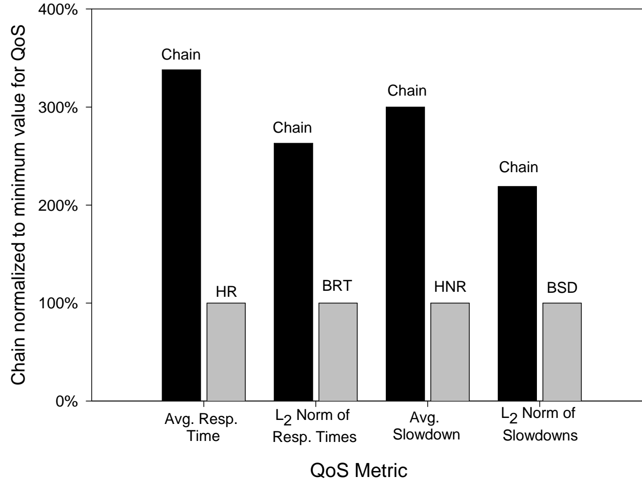

Figure 26: [§3.8.2] Performance of Chain under QoS metrics

of a low selectivity query with low processing cost. Hence, it allows *HNR* to schedule low selectivity queries earlier and save memory space.

Figure 25 also shows how the *BRT* and *BSD* policies provide more savings in memory usage. The reason is that under such policies, if a low selectivity query has been waiting for a long time, its priority increases until it is eventually executed. For instance, *BSD* decreases the memory usage by up to 13% compared to *HNR*.

In order to put these results into the proper perspective, we also compared the performance of *Chain* to our proposed policies under the different QoS metrics that we have studied in previous experiments. Figure 26 shows that *Chain* consistently suffers under all of the QoS metrics studied in this thesis.

For example at utilization 0.97, *Chain* provides 3 times the average slowdown of *HNR* which needs only 2 times the memory of *Chain*. Similarly, at utilization 0.97, *Chain* increases the ℓ_2 norm of slowdowns by 2.2 times compared to *BSD* although *BSD* requires memory space that is only 1.85 times more than that of *Chain*. Thus, *BSD* is able to also strike a fine balance between improving the interactive performance within acceptable memory requirements.

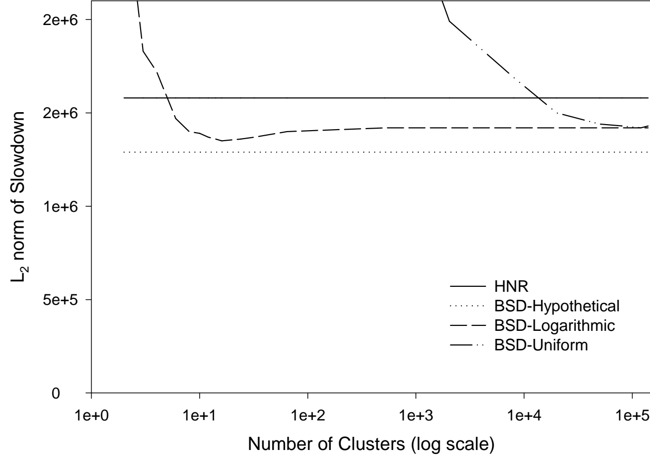

Figure 27: [§3.8.3] ℓ_2 of slowdown vs. number of clusters

3.8.3 Comparison of Implementation Techniques

To evaluate the impact of the implementation techniques proposed in Section 3.3, we compared the performance of four policies: *HNR*, *BSD-Hypothetical*, *BSD-Uniform*, and *BSD-Logarithmic* which are defined as follows:

- *BSD-Hypothetical* is a version of *BSD* where we ignore the scheduling overheads.
- *BSD-Uniform* uses uniform clustering as in [16].
- *BSD-Logarithmic* uses our proposed logarithmic clustering (described in Section 3.3).

In both *BSD-Uniform* and *BSD-Logarithmic*, we set the cost of each of the priority computation and comparison operations to the cost of the cheapest operator in the query plans.

Figure 27 shows the ℓ_2 norm of slowdowns provided by the four policies vs. the number of clusters (i.e., m) at 0.95 utilization. The figure shows that for *BSD-Logarithmic*, when m is small (≤ 6), its ℓ_2 might exceed that for *HNR*, because the priority range covered by each cluster is large which decreases the accuracy of the scheduling. However, as we increase m, its performance gets closer to that of *BSD-Hypothetical* such that at 12 clusters, its provided ℓ_2 norm is only 5% higher than *BSD-Hypothetical*. By increasing m beyond 12, its ℓ_2 norm

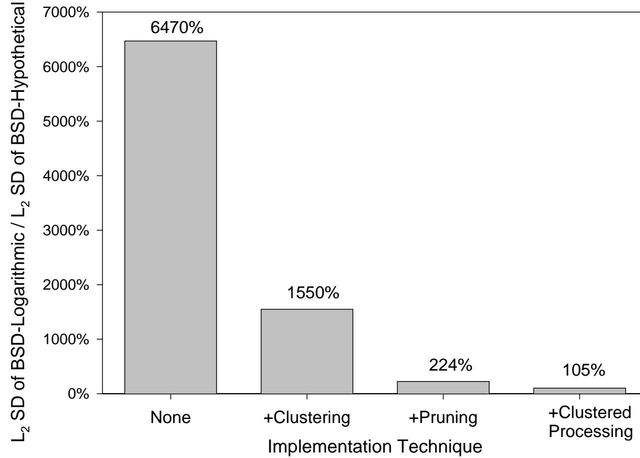

Figure 28: [§3.8.3] Efficient implementation of BSD

starts increasing again due to increasing the search space. On the other hand, *BSD-Uniform* starts at a very high ℓ_2 and it decreases slowly with increasing m. That is, the accuracy of the solution is very poor when the cluster size is large. As such, *BSD-Uniform* starts to provide acceptable performance (10% higher than *BSD-Hypothetical*) when the cluster range is very small (notice that in this setting $\Delta \approx 1.2e + 05$).

Figure 28 shows the incremental gains provided by each of the proposed implementation techniques when using 12 logarithmic clusters. The figure shows that a naive implementation of *BSD* will increase the ℓ_2 norm by 6470% compared to *BSD-Hypothetical*. By incrementally adding each of the implementation techniques, we achieve a performance that is only 5% higher than *BSD-Hypothetical*, i.e., the implementation overhead of the *BSD* policy is only 5%.

3.8.4 Operator Sharing

To measure the performance of the sharing-aware versions of *HNR* and *BSD*, we created a workload in which queries are grouped randomly in sets of 10 queries each where all queries within a set share the same select operator.

Figures 29, 30, and 31 show the performance of different scheduling policies under the

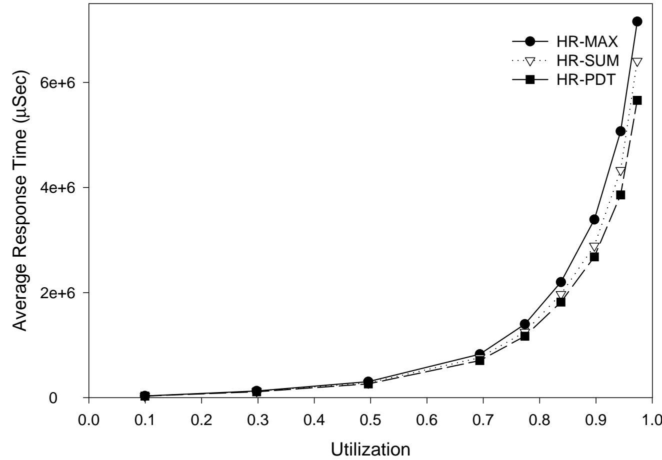

Figure 29: [§3.8.4] Response time for grouped queries

response time, slowdown, and ℓ_2 norm of slowdown metrics respectively. In each figure, we compare the performance of three variants for implementing the same policy that optimizes the metric under investigation. These three variants are: *Max*, *Sum*, and *PDT*, defined as follows:

- *Max:* where the shared operator priority is equal to the priority of that *one* segment within the group that has the maximum priority.
- *Sum:* where the shared operator priority is equal to the aggregation of the priorities of *all* the segments in a group.
- *PDT:* where the shared operator priority is equal to the aggregation of the priorities of the segments in its *priority-defining tree* (as described in Section 3.6).

The figures show that the *PDT* strategy significantly improves the performance of each scheduling policy. For example, Figure 29 shows that, compared to *Max* and *Sum*, *PDT* reduces the response time by 21% and 12% respectively, whereas the reductions in slowdown are 24% and 18% (Figure 30) and finally, the reductions in the ℓ_2 norm of slowdowns are 10% and 8% (Figure 31).

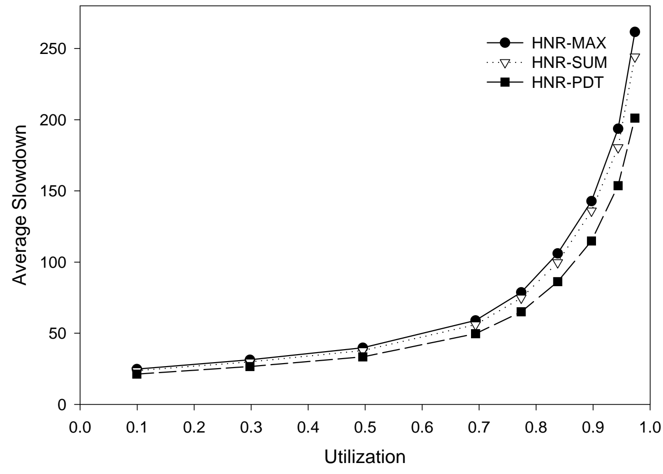

Figure 30: [§3.8.4] Slowdown for grouped queries

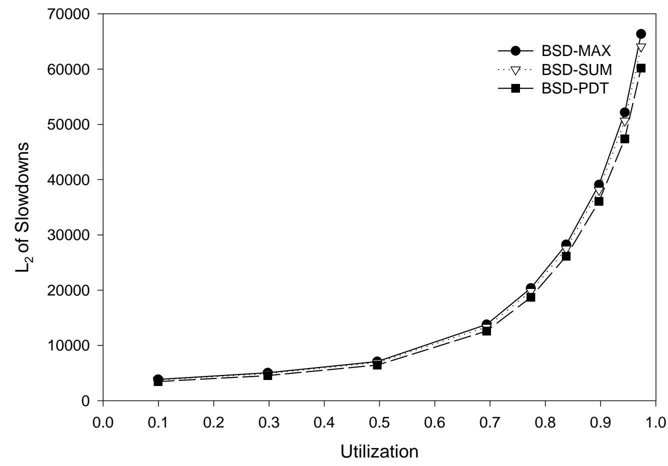

Figure 31: [§3.8.4] ℓ_2 of slowdowns for grouped queries

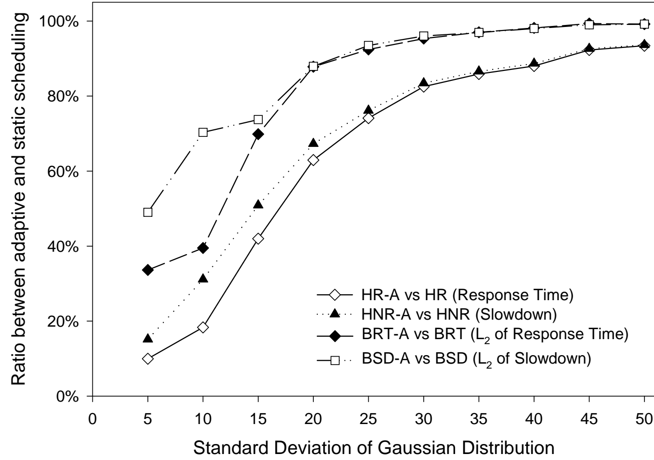

Figure 32: [§3.8.5] Ratio of adaptive scheduler performance vs. static

3.8.5 Adaptive Scheduling

In all the previous experiments, queries operated on data that was generated according to a uniform distribution in the range of [1, 100]. In this experiment, we use a more dynamic setting to study the performance of our adaptive scheduling mechanism. Specifically, we divide the simulation time into 100 intervals, where the data in each interval is generated according to a Gaussian distribution that is specified by a mean and a standard deviation. The mean starts at 50.0 and it is incremented by one with every new interval.

The goal of this set of experiments is to study the behavior of the adaptive variants of our proposed policies; basically, this means that for the adaptive policies, selectivity will be estimated dynamically, as described in Section 3.3.3.

Figure 32 shows the ratio between the performance of the adaptive and the non-adaptive versions of each policy under the metric optimized by that policy. For instance, it compares the performance of the adaptive *HR* (i.e., *HR-A*) to the non-adaptive *HR* under the response time metric. For example, a value of 20% for *HR-A* vs. *HR* means that *HR-A*'s response time is 20% of that of *HR*. The non-adaptive *HR* assumes that data is uniformly distributed, whereas *HR-A* monitors the data distribution and adjusts the operators selectivities and priorities accordingly.

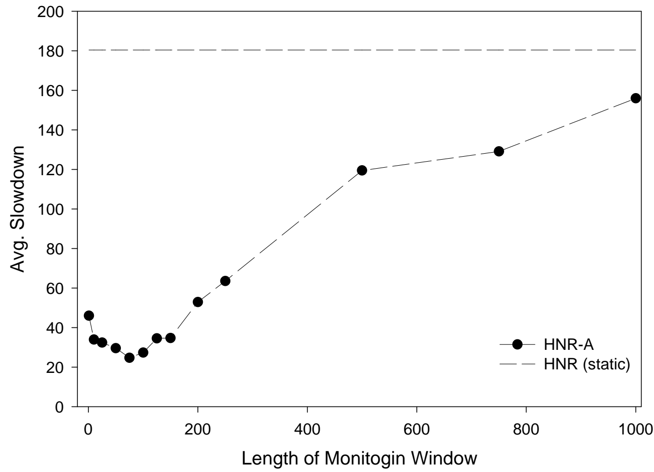

Figure 33: [§3.8.5] Impact of monitoring window length on adaptive scheduling

Figure 32 shows that the adaptive versions of all policies always outperform the non-adaptive ones especially at low values of standard deviation where the distribution is highly skewed within each interval. For instance, at a standard deviation of 25, HR-A's response time is 74% of HR and HNR-A's slowdown is 76% of HNR, whereas at a standard deviation of 5, these values are 10% and 15% respectively.

Figure 32 also shows that the relative gain provided by HNR-A is lower than that provided by HR-A. This is because HNR uses the ideal processing time in its prioritizing function; this makes its non-adaptive version less sensitive to the fluctuations in selectivity. Similarly, the relative gains provided by BRT-A and BSD-A are lower than HR-A, since both BRT and BSD use the wait time in their prioritizing functions.

Obviously, the improvement in performance provided by adaptive scheduling depends on the choice of values for the monitoring window length and the aging parameter α. In the results shown in Figure 32, we selected a window of length 100 input tuples and a value of α equal to 0.175. In order to chose these specific values, we explored the combinatorial search space of the two parameters. We observed that, in general, very low values of α yield a very unstable system as it gives very low weight to the old observations, while high values of α result in an almost static system that cannot adapt fast enough to changes. Similarly, for

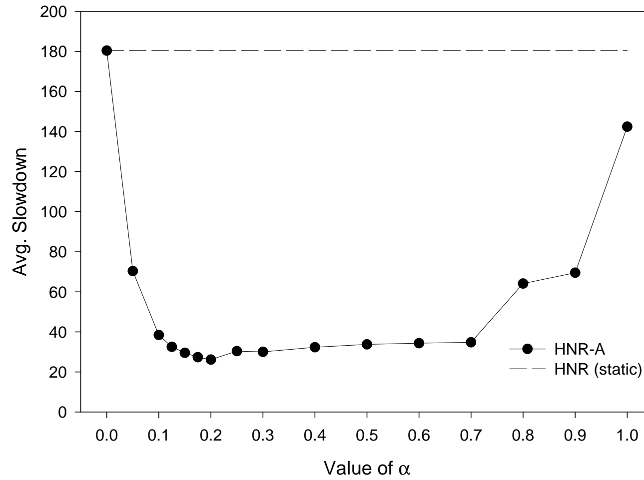

Figure 34: [§3.8.5] Impact of α value on adaptive scheduling

the window length, a short window does not have enough data to provide good estimates of selectivity, while long windows provide outdated statistics.

Samples of the search space are provided in Figures 33 and 34 (at standard deviation 5 as in Figure 32). In particular, in Figure 33, we plot the performance of the adaptive scheduler compared to the static one when α is equal to 0.175 and variable window length. Similarly, in Figure 34, we plot the performance when the window length is 100 and α is variable. The figures show that, in general, windows between 50 and 150 tuples and αs between 0.1 and 0.25 provide the best performance.

4.0 QOD METRICS AND ALGORITHMS

As the amount of updates on the input data streams increases and the number of registered queries becomes large, advanced query processing techniques are needed in order to efficiently synchronize the results of the continuous queries with the available updates. Efficient *scheduling* of updates is one such query processing technique which successfully improves the *Quality of Data (QoD)* provided by interactive systems. QoD can be measured in different ways, one of which is *freshness*. Freshness is especially important, when we are interested in an accurate view of the physical world, be it an outbreak of a disease (as in the RODS system) or the detection of traffic patterns and congestion in an urban setting during a physical disaster. Such accurate views must reflect *all positive event "signals"* (i.e., updates) that satisfy the registered CQs.

Freshness, as well as scheduling policies for improving freshness, has been studied in the context of replicated databases [21, 22], derived views [34], and distributed caches [47]. To the best of our knowledge, our work is the first to study the problem of freshness in the context of data streams. In this respect, our work can be regarded as complementary to the current work on the processing of continuous queries, which considers only Quality of Service metrics like response time and throughput (e.g., [20, 49, 16, 17, 6]) as well as our work presented earlier in Chapter 3.

Our contributions towards improving QoD in data streams are summarized as follows:

1. We propose a policy for *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)*. The proposed policy, FAS-MCQ, has the following salient features:
    - It exploits the variability of the processing costs of different continuous queries registered at the DSMS.

- It utilizes the divergence in the arrival patterns and frequencies of updates streamed from different remote data sources.
- It considers the impact of *selectivity* on the freshness of the output data stream. Reverting back to our RODS/event detection example, our proposed policy will favor queries that lead to positive signals instead of "blindly" processing queries that lead to negative signals.

2. Beyond the basic FAS-MCQ policy, we have also explored a *weighted version* of our FAS-MCQ scheduling policy that supports applications in which queries have different priorities. These priorities could reflect *criticality*, and hence their importance with respect to QoD captured by freshness, or *popularity*, and thus be used to optimize the overall user satisfaction.
3. To be able to study the difference in behavior between scheduling with the goal of improving QoD as opposed to scheduling for improving QoS, we generalized the Rate-based scheduling policy [65] to handle multiple continuous queries. The new generalized version, which we call *Rate-based for Multiple Continuous Queries (RB-MCQ)*, maximizes the QoS defined in terms of response time.
4. Finally, we propose a parameterized version of our FAS-MCQ scheduler that is able to balance the trade-off between freshness and response time according to the application's requirements.

In order to evaluate our proposed scheduling policies, we have implemented a simulator of a DSMS scheduler and ran extensive experiments. As our experimental results have shown, FAS-MCQ can improve QoD by up to 55% compared to existing scheduling policies used in DSMSs. FAS-MCQ achieves this improvement by deciding the execution order of continuous queries based on individual query properties (i.e., cost and selectivity) as well as properties of the update streams (i.e., variability of updates).

4.1 FRESHNESS OF DATA STREAMS

In this section, we describe our proposed metric for measuring the quality of output data streams. Our metric is based on the *freshness* of data and is similar to the ones previously used in [21, 34, 47, 22, 35]. However, it is adapted to consider the nature of continuous queries and input/output data streams.

4.1.1 Average Freshness for Single Streams

In a DSMS, the output of each continuous query Q is a data stream D. The arrival of new updates at the input queue of Q might lead to appending a new tuple to D. Specifically, let us assume that at time t the length of D is $| D_t |$ and there is a single update at the input queue, also with timestamp t. Further, assume that Q finishes processing that update at time t'. At this time we distinguish two cases:

- If the tuple satisfies all the query's predicates, then $| D_{t'} |=| D | +1$. In this case, the output data stream D is considered **stale** during the interval $[t, t']$ as the new update occurred at time t and it took until time t' to append the update to the output data stream.
- If the tuple does not satisfy all the predicates, then $| D_{t'} |=| D |$. In this case, the output data stream D is considered **fresh** during the interval $[t, t']$ because the arrival of a new update has been discarded by Q. Obviously, if there is no pending update at the input queue of D, then D would also be considered fresh.

Equivalently, if we view a tuple that matches all the predicates of a query as a *positive "signal"*, then the current definition of freshness measures the amount of time that passes before the signal becomes "visible" to the end users.

Formally, to define freshness, we consider each output data stream D as an object and $F(D, t)$ is the freshness of object D at time t which is defined as follows:

$$F(D,t) = \begin{cases} 1 & \text{if } \forall u \in I_t, \sigma(u) \text{ is false} \\ 0 & \text{if } \exists u \in I_t, \sigma(u) \text{ is true} \end{cases} \tag{4.1}$$

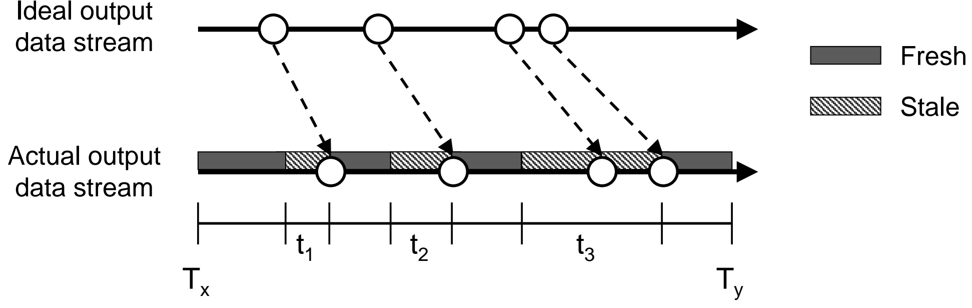

Figure 35: An example on measuring the freshness of a data stream

where I_t is the set of input queues in Q at time t and $\sigma(u)$ is the result of applying Q's predicates on update u. To measure the freshness of a data stream D over an entire discrete observation period from time T_x to time T_y, we have that:

$$F(D) = \frac{1}{T_y - T_x} \sum_{t=T_x}^{T_y} F(D, t) \tag{4.2}$$

Figure 35 shows an example of measuring the freshness of a data stream. Specifically, the figure shows two output data streams; (1) the *ideal* stream, which shows the times instants when updates became available at the DSMS; and (2) the *actual* stream, which shows the time instants when updates became available to the user. The delay between the time an update is available at the system until the time it is propagated to the user is composed of two intervals: (a) the interval where the continuous query is waiting to be scheduled for execution; and (b) the interval where the continuous query is processing the update. The sum of these two intervals represents the overall interval when the output data stream deviates from the ideal one. That is, when the output data stream is stale compared to the physical world. In the example illustrated in Figure 35, the output data stream is stale for the intervals t_1, t_2 and t_3. Hence, the staleness of the data stream is computed as: $(t_1 + t_2 + t_3)/(T_y - T_x)$, equivalently, the freshness of the data stream is computed as: $((T_y - T_x) - (t_1 + t_2 + t_3))/(T_y - T_x)$.

4.1.2 Average Freshness for Multiple Streams

Having measured the average freshness for single streams, we proceed to compute the average freshness over all the M data streams maintained by the DSMS. If the freshness for each stream, D_i, is given by $F(D_i)$ using Equation 4.2, then the average freshness over all data streams will be:

$$F = \frac{1}{M} \sum_{i=1}^{M} F(D_i) \tag{4.3}$$

4.2 FRESHNESS-AWARE SCHEDULING OF MULTIPLE CONTINUOUS QUERIES

In this section we describe our proposed policy for *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)*. Current work on scheduling the execution of multiple continuous queries focuses on QoS metrics (Chapter 3 and [16, 17, 6]) and exploits *selectivity* to improve the provided QoS. Previous work on synchronizing database updates exploited the *amount (frequency)* of updates to improve the provided QoD [21, 47, 22]. In contrast, our proposal, *FAS-MCQ*, exploits both selectivity and the amount of updates to improve the QoD, i.e., freshness, of output data streams.

4.2.1 Scheduling without Selectivity

Assume two queries Q_1 and Q_2, with output data streams D_1 and D_2. Each query is composed of a set of operators, each operator has a certain cost, and the selectivity of each operator is one. Hence, we can calculate for each query Q_i its maximum cost T_i as shown in Section 2. Moreover, assume that there are N_1 and N_2 pending updates for queries Q_1 and Q_2 respectively. Finally, assume that the current wait time for the update at the head of Q_1's queue is W_1, similarly, the current wait time for the update at the head of Q_2's queue is W_2.

In order to determine which of the two queries should be scheduled first for execution, we compare two policies X and Y:

- Under policy X, query Q_1 is executed before query Q_2,
- Under policy Y, query Q_2 is executed before query Q_1.

Under policy X, where query Q_1 is executed before query Q_2, the total loss in freshness, L_X, (i.e., the period of time where Q_1 and Q_2 are *stale*) can be computed as follows:

$$L_X = L_{X,1} + L_{X,2} \tag{4.4}$$

where $L_{X,1}$ and $L_{X,2}$ are the staleness periods experienced by Q_1 and Q_2 respectively. Since Q_1 will remain stale until all its pending updates are processed, $L_{X,1}$ is computed as follows:

$$L_{X,1} = W_1 + (N_1 T_1)$$

where W_1 is the current loss in freshness (i.e., increase in staleness) and $(N_1 T_1)$ is the time required to apply all the pending updates. Similarly, $L_{X,2}$ is computed as follows:

$$L_{X,2} = (W_2 + N_1 T_1) + (N_2 T_2)$$

where W_2 is the current loss in freshness plus the extra amount of time $(N_1 T_1)$ where Q_2 will be waiting for Q_1 to finish execution. By substitution in Equation 4.4, we get

$$L_X = W_1 + (N_1 T_1) + (W_2 + N_1 T_1) + (N_2 T_2) \tag{4.5}$$

Similarly, under policy Y, where Q_2 is scheduled before Q_1, we have that the total loss in freshness, L_Y will be:

$$L_Y = (W_1 + N_2 T_2) + (N_1 T_1) + W_2 + (N_2 T_2) \tag{4.6}$$

In order for L_X to be less than L_Y, the following inequality must be satisfied:

$$N_1 T_1 < N_2 T_2 \tag{4.7}$$

The left-hand side of Inequality 4.7 shows the total staleness incurred by Q_2 when Q_1 is executed first. Similarly, the right-hand side shows the total staleness incurred by Q_1 when Q_2 is executed first. Hence, the inequality implies that between the two queries, we start with the one that has the lower $N_i T_i$ value. Similarly, in the general case, where there are more than 2 queries ready for execution, we start with the one that has the lowest $N_i T_i$ value since it will have the minimum negative impact on the freshness of the other queries in the system. Minimizing the negative impact on the overall freshness was the same general criterion that was used in prior work on scheduling updates over materialized WebViews [34].

4.2.2 Scheduling with Selectivity

Assume the same setting as in the previous section, with the only difference being that the total productivity of each query Q_i is $S_i \in [0, 1]$, which is computed as in Section 2. The objective when scheduling with selectivity is the same as before: we want to minimize the total staleness. Recall from Inequality 4.7 that the objective of minimizing the total loss is equivalent to selecting for execution the query that minimizes the loss in freshness incurred by the other query. In the presence of selectivity, we will apply the same principle.

We first need to compute for each output data stream D_i its *staleness probability* (P_i) given the current status of the input data stream. This is equivalent to computing the probability that at least one of the pending updates will satisfy all of Q_i's predicates. If S_i is the total selectivity of Q_i, then $(1 - S_i)^{N_i}$ is the probability that all pending updates do not satisfy Q_i's predicates, and hence $P_i = 1 - (1 - S_i)^{N_i}$ is the staleness probability for Q_i.

If out of two queries Q_1 and Q_2, Q_2 is executed before Q_1, then the expected loss in freshness incurred by Q_1 due only to the impact of processing Q_2 first will be:

$$L_{Q_1} = P_1 N_2 \overline{C}_2 \tag{4.8}$$

where $N_2\overline{C}_2$ is the expected time that Q_1 will be waiting for Q_2 to finish execution and P_1 is the probability that D_1 is stale in the first place. For example, in the extreme case of $S_1 = 0$, if Q_2 is executed before Q_1, it will not increase the staleness of D_1 since all the updates will not satisfy Q_1. However, at $S_1 = 1$, if Q_2 is executed before Q_1, then the staleness of D_1 will increase by $N_2\overline{C}_2$ with probability one.

Similarly, if Q_1 is executed before Q_2, then the expected loss in freshness incurred by Q_2 only due to processing Q_1 first is computed as:

$$L_{Q_2} = P_2 N_1 \overline{C}_1 \tag{4.9}$$

In order for L_{Q_2} to be less than L_{Q_1}, then the following inequality must be satisfied:

$$\frac{N_1\overline{C}_1}{P_1} < \frac{N_2\overline{C}_2}{P_2} \tag{4.10}$$

Thus, in our proposed policy, each query Q_i is assigned a priority value V_i which is the product of its staleness probability and the reciprocal of the product of its expected cost and the number of its pending updates. Formally,

$$V_i = \frac{1 - (1 - S_i)^{N_i}}{N_i \overline{C}_i} \tag{4.11}$$

4.2.3 The FAS-MCQ Policy

Our proposed policy for *Freshness-Aware Scheduling for Multiple Continuous Queries (FAS-MCQ)* uses the priority function of Equation 4.11 to determine the scheduling order of different queries. Under this priority function *FAS-MCQ* behaves as follows:

1. If all queries have the same number of pending tuples and the same selectivity, then FAS-MCQ selects for execution the query with the lowest cost.
2. If all queries have the same cost and the same selectivity, then FAS-MCQ selects for execution the query with less pending tuples.
3. If all queries have the same cost and the same number of pending tuples, then FAS-MCQ selects for execution the query with high staleness probability.

In case (1), *FAS-MCQ* behaves like the *Shortest Remaining Processing Time* policy. In case (2), *FAS-MCQ* gives lower priority to the query with high frequency of updates. The intuition is that when the frequency of updates is high, it will take a long time to establish the freshness of the output data stream. This will block other queries from executing and will increase the staleness of their output data streams. In case (3), *FAS-MCQ* gives lower priority to queries with low selectivity as there is a low probability that the pending updates will "survive" the filtering of the query operators and thus be appended to the output data stream.

4.2.4 Weighted Freshness

In many monitoring applications, some queries are more important than others. That is especially obvious in emergency systems where a few continuous queries can be more critical than others. For example, under the RODS system that monitors for disease outbreaks, it

is crucial to monitor for signs of waterborne diseases in areas affected by Hurricane Katrina (and thus consider the corresponding query more crucial than the rest), whereas in other areas of the world it may be more important to monitor for signs of the avian flu. In cases like these, when the system is loaded, it is necessary to maximize the freshness of these critical queries.

Towards this, we modify our proposed FAS-MCQ policy to increase the freshness of data streams which have higher levels of importance. Specifically, we assign each continuous query Q_i a *weight* α_i. This assigned weight represents the importance of the query and it takes values in the range $(0.0, 1.0]$ where the weight 1.0 is assigned to the most important query. Hence, the objective of our policy would be to maximize the overall *weighted freshness*. A priority function that allows us to maximize the weighted freshness can be easily deduced from Equations 4.8 and 4.9. Recall that Equation 4.8 measures the expected loss in freshness experienced by Q_1 due to executing Q_2 first, thus, the expected loss in weighted freshness experienced by Q_1 is measured as:

$$WL_{Q_1} = \alpha_1 P_1 N_2 \overline{C}_2$$

Similarly, the expected loss in weighted freshness experienced by Q_2, when Q_1 is executed first, is measured as:

$$WL_{Q_2} = \alpha_2 P_2 N_1 \overline{C}_1$$

In order for WL_{Q_2} to be less than WL_{Q_1}, the following inequality must be satisfied:

$$\frac{N_1 \overline{C}_1}{P_1 \alpha_1} < \frac{N_2 \overline{C}_2}{P_2 \alpha_2}$$

Then, the priority assigned to each query is computed as:

$$V_i = \frac{\alpha_i (1 - (1 - S_i)^{N_i})}{N_i \overline{C}_i} \tag{4.12}$$

The weights of the queries can be explicitly or implicitly defined, depending on the application. For example, in the case of an application that includes queries that are critical, the critical queries can be explicitly assigned higher weights than the rest of the queries. In applications where explicit criticality/importance information is not given, an implicit measure of importance can be derived. For example, the popularity of each query (i.e., the

number of users that registered that query) can be used as the weight. In such an application, the weighted FAS-MCQ policy will provide high levels of overall user satisfaction in terms of QoD (freshness). Finally, it is worth mentioning that the weight given to a query can be dynamic; for example, it can change depending on the time of day or the day of the week (e.g., for traffic management queries).

4.3 SCHEDULING FOR QOD VS. SCHEDULING FOR QOS

In this section we discuss the difference in behavior between scheduling with the goal of improving QoD as opposed to scheduling with the goal of improving QoS (i.e., when the objective is to minimize the *average response time*). We also present a parameterized version of our FAS-MCQ scheduler that balances the trade-off between both the QoD and QoS metrics.

4.3.1 Scheduling for QoS

Recall in Chapter 3, we proposed the *Highest Rate (HR)* policy as a multiple query scheduling policy for minimizing response time. To recap, *HR* generalizes the basic Rate-based strategy [65] for scheduling multiple continuous queries with the objective of minimizing the average response time. That is, multiple continuous queries are scheduled for execution based on their output rates.

In *HR*, we simply view the network of multiple queries as a set of operator paths and at each scheduling point we select for execution the path with the highest priority (i.e., rate). Specifically, under *HR*, each path E_i has a value called the *global output rate* (GR_i) which is defined in terms of the parameters of the path operators. The output rate of a path E_i, composed of the operators $< O_1, O_2, O_3, ..., O_r >$, is basically the expected number of tuples produced per time unit due to processing one tuple by the operators along the path all the way to the root O_r. Formally,

$$GR_i = \frac{S_i}{\overline{C}_i} \tag{4.13}$$

or, equivalently,

$$GR_i = \frac{1 - (1 - S_i)}{\overline{C}_i} \tag{4.14}$$

where S_i and $\overline{C}_i$ are the path's expected productivity and expected cost as defined in Chapter 2.

Table 4: Priority functions for scheduling QoS vs. QoD

Scheduling for QoS	Scheduling for QoD
$\frac{1-(1-S_i)}{\overline{C}_i}$	$\frac{1-(1-S_i)^{N_i}}{N_i\overline{C}_i}$
Equation 4.14	Equation 4.11

4.3.2 Balancing the Trade-off between QoD and QoS

The difference between scheduling for QoD and QoS is easily identified by comparing the priority functions used by FAS-MCQ (Equation 4.11) versus the one used by HR (Equation 4.14), which we replicate in Table 4. That is, FAS-MCQ considers three factors: (1) cost (2), selectivity, and (3) number of pending tuples, whereas HR considers only the first two factors. As a result, FAS-MCQ might favor a query with a relatively expensive cost and very few pending tuples as opposed to HR which might favor an inexpensive query with a large number of pending tuples. In this case, HR may be appending tuples faster to the output data streams, however, the appended tuples would be stale most of the time. On the other hand, FAS-MCQ might be relatively slower in appending tuples to the output data streams yet would maintain most of those output data streams as fresh as possible.

Here, we propose a version of FAS-MCQ that balances the trade-off between QoD and QoS. We refer to this policy as FAS-MCQ(β), where β is a parameter that specifies the weight given to the number of pending tuples N in the priority function. Specifically, under FAS-MCQ(β), query Q_i is assigned a priority value V_i which is computed as follows:

$$V_i = \frac{1-(1-S_i)^{N_i^{\beta}}}{N_i^{\beta}\overline{C}_i} \tag{4.15}$$

The parameter β takes values in the range [0.0,1.0] and it acts as a knob for shaping the system's behavior. For instance, for $\beta = 0.0$, FAS-MCQ(0.0) behaves like the HR policy described above, whereas for $\beta = 1.0$, FAS-MCQ(1.0) reverts to the original FAS-MCQ described in Section 4.2. For settings where $0.0 < \beta < 1.0$, the system achieves the desired balance between QoD and QoS.

4.4 EVALUATION TESTBED

In this section, we first describe our implementation of the FAS-MCQ scheduler then the simulation parameters used for our experimental evaluation.

4.4.1 Implementing the FAS-MCQ Scheduler

The FAS-MCQ Scheduler is invoked at every *scheduling point* and uses the current values for N_i and S_i to compute the priority of each query Q_i, according to Equation 4.11. In our implementation of the FAS-MCQ policy, a *scheduling point* is reached when a query finishes execution. In order to keep the scheduling overhead low when computing priorities, we use a *Calendar Queue* [14] for priority management. Calendar queues have been widely used for implementing priority-based scheduling algorithms in high-speed networks as well as in the Aurora DSMS [16].

A calendar queue is an $O(1)$ priority queue and is based on the idea of *Bucket Sort*. Specifically, the calendar queue is structured as buckets where each bucket corresponds to a class of priorities. To insert an element into the calendar queue, a hash function is used to map its priority to the corresponding bucket. To retrieve elements, buckets are traversed in order. A calendar queue allows us to avoid re-computing the priorities of queries that received no new updates between consecutive scheduling points. Additionally, for queries with new updates, the amortized cost of updating the priority is of $O(1)$.

4.4.2 Simulation Parameters

We have conducted several experiments to compare the performance of our proposed scheduling policy and its sensitivity to different parameters. Specifically, we compared the performance of our proposed *FAS-MCQ* policy to a two-level scheduling scheme from Aurora where Round Robin is used to schedule queries and pipelining is used to process updates within the query. Collectively, we refer to the Aurora scheme in our experiments as *RR*. We also included the *HR* policy described in Section 4.3 as well as a the First-Come-First-Served (FCFS) policy where updates are processed according to their arrival times.

Queries: We simulated a DSMS that hosts 250 registered continuous queries. The structure of the query is adapted from [19, 42] where each query consists of three operators: two predicates and one projection.

Real Data Streams: We use the same *LBL-PKT-4* traces from the *Internet Traffic Archive* that we have used earlier in the experiments described in Chapter 3.8. The traces contain an hour's worth of all wide-area traffic between the Lawrence Berkeley Laboratory and the rest of the world. In our experiments, we use the *TCP* and *UDP* packet traces as two input data streams to the system where the registered queries are uniformly assigned to any of the two data streams.

Synthetic Data Streams: In this setting, we generate 10 input data streams each 10K in length tuples. Initially, we generate updates for each stream according to a Poisson distribution, with its mean inter-arrival time set according to the simulated system utilization (or load). For a utilization of 1.0, the inter-arrival time is equal to the exact time required for executing the queries in the system, whereas for lower utilizations, the mean inter-arrival time is increased proportionally.

To generate a back-log of updates, we traverse the Poisson stream and group every 10 consecutive tuples in a burst where the arrival time of all tuples that belong to the same burst is equal to that of the first tuple in the burst. In the default setting, 5 out of the 10 data streams are bursty.

Selectivities: In any query, the selectivity of the projection is set to 1, while the two predicates have the same value for selectivity, which is selected using a Zipf distribution from the range [0.1, 1.0]. The Zipf distribution is defined using a Zipf parameter which determines the degree of skewness. In our setting, the skewness is toward queries with selectivity equal to 1.0 and in the default setting the Zipf parameter is set to 0.0 (i.e., uniform distribution).

Costs: All operators that belong to the same query have the same cost, which is uniformly selected from three possible classes of costs. The cost of an operator in class i is equal to: $K \times 2^i$ time units, where $i \in [0\text{–}2]$ and K is the *scaling factor* which is used to scale the costs of operators to meet the desired utilization. For synthesized data, K is equal to 1.

For the network traces, we measure the inter-arrival time of the data trace, then we set K so that the ratio between the total expected costs of queries and the inter-arrival time is

Table 5: Simulation Parameters for QoD Experiments

Parameter	Value
Policies	FAS-MCQ, HR, RR, FCFS
Number of Queries	250
Number of Operators per Query	3
Operators' Costs	$1K$, $2K$, $4K$
Operators' Selectivities	0.1–1.0
Utilization	0.1–0.99
Data Streams	real and synthetic
Number of Data Streams	2–10
Number of Bursty Streams	0–10

equal to the simulated utilization. Finally, the cost of each of the calendar queue operations is equal to the cost of the cheapest operator in the system.

Table 5 summarizes our simulation parameters and settings.

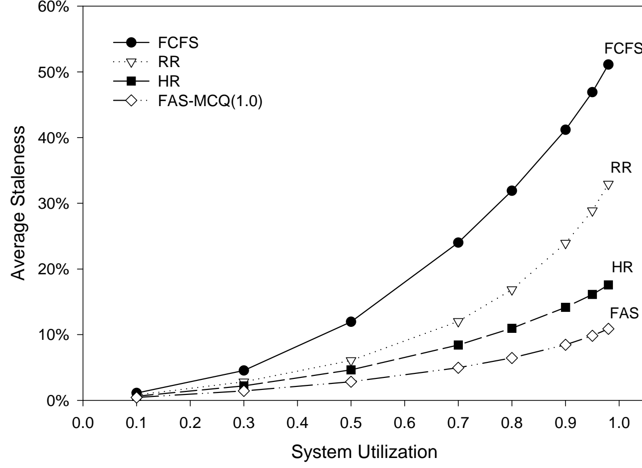

Figure 36: [§4.5.1] Average staleness vs. system utilization

4.5 EXPERIMENTS

4.5.1 Impact of Utilization

Figure 36 shows the average staleness over all output data streams. In this setting, we use the basic version of FAS-MCQ which is equivalent to setting β to 1.0 in FAS-MCQ(β). The figure shows that the staleness of output data streams increases with increasing load. It also shows that the FAS-MCQ policy provides the lowest staleness for all values of utilization with HR being the closest contender. Additionally, the relative improvement provided by FAS-MCQ increases with increasing utilization. For instance, at 0.1 utilization, FAS-MCQ achieves 30% reduction in staleness compared to HR, whereas at 0.95 utilization, HR provides a 16% staleness while FAS-MCQ reduces the staleness to 10% (i.e., a 40% reduction).

As expected, the reduction in staleness provided by FAS-MCQ comes at the expense of an increase in response time which is illustrated in Figure 37. The figure shows how HR reduces the response time compared to FAS-MCQ. For example, at 0.95 utilization, the response time provided by FAS-MCQ is 23% higher than that of HR (at a 40% reduction in staleness compared to HR). The trade-off between freshness and response time is further illustrated using Figures 38 and 39 as explained next.

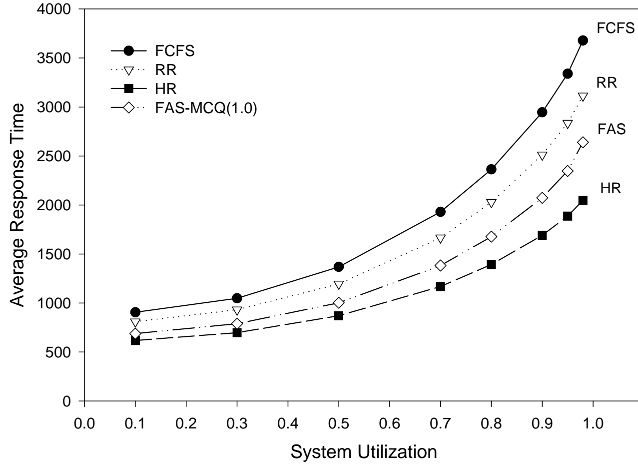

Figure 37: [§4.5.1] Average response time vs. system utilization

4.5.2 Staleness vs. Response Time

Figures 38 and 39 show the average staleness and average response time for the same simulation settings used in the previous experiment. In addition to illustrating the difference in behavior between HR and FAS-MCQ, the figures also show the performance of the parameterized FAS-MCQ(β) policy. For instance, $\beta = 0$ corresponds to HR, whereas $\beta = 1$ corresponds to the pure FAS-MCQ policy.

Figure 38 shows how the response time of FAS-MCQ decreases by decreasing the value of β down to $\beta = 0.0$. At $\beta = 0.0$, the response time of FAS-MCQ(0.0) is slightly higher than HR which is due to the scheduling overheads. On the other hand, Figure 38 shows the reduction in staleness with increasing values of β.

To better assess the magnitude of the trade-off, we plot the performance of the different policies at utilization 0.95 in Figure 40. For instance, the figure shows how FAS-MCQ(1.0) reduces the staleness by 40% while increasing the response time by 23%, whereas FAS-MCQ(0.25) reduces the staleness by 20% and increases the response time by 14%.

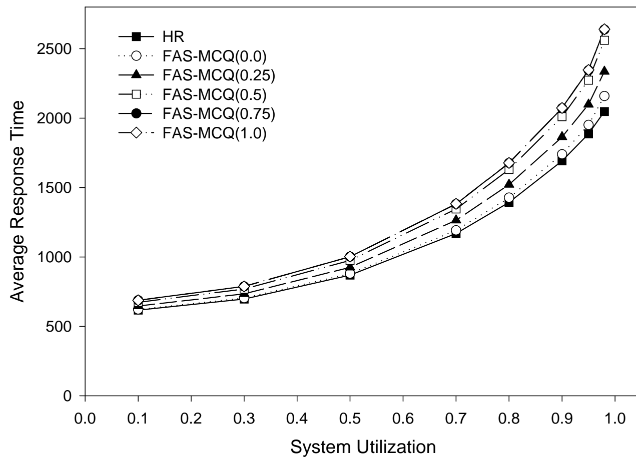

Figure 38: [§4.5.2] Response time for different βs

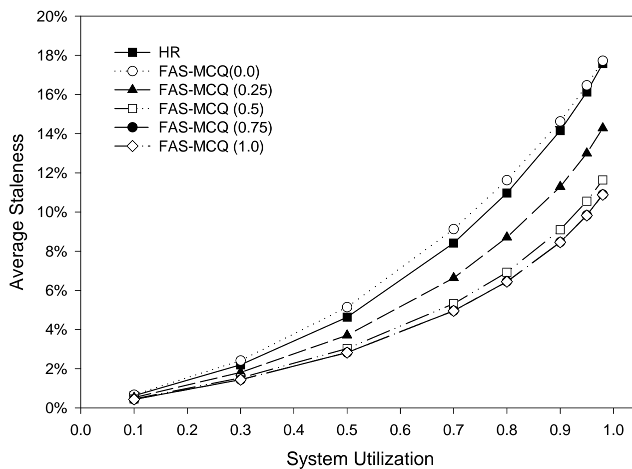

Figure 39: [§4.5.2] Staleness for different βs

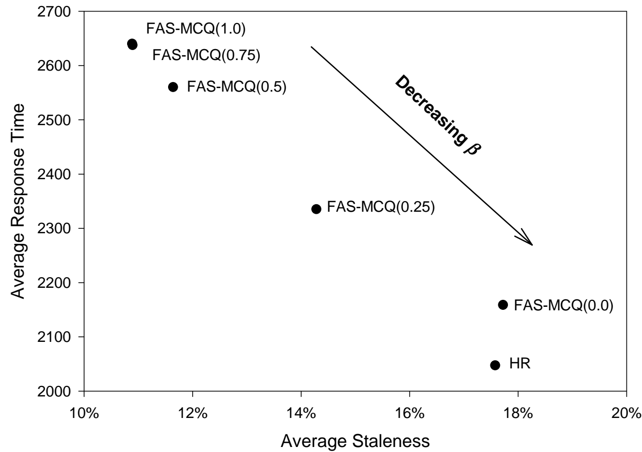

Figure 40: [§4.5.2] Trade-off between staleness and response time at utilization 0.95

4.5.3 Impact of Selectivity

Figure 41 shows the average staleness for an experiment where all operators have the same cost, utilization is set to 95%, and the skewness of selectivity is variable. Recall that we control the degree of skewness using a Zipf parameter. Specifically, setting the Zipf parameter to 0.0 results in a uniform distribution, whereas by the increasing its value the distribution is skewed more towards the 1.0 value for selectivity. That is, most of the registered queries are productive.

Figure 41 shows that by increasing the skewness, the staleness provided by all policies increases. This is because when most of the queries have high selectivity, then the arrival of new updates will render the output data streams stale most of the time. The figure also shows that the gains provided by FAS-MCQ compared to HR increase with increasing the skewness.

For instance, at 0.0, FAS-MCQ reduces the staleness by 39% compared to HR. This reduction goes up to 55% when the distribution is highly skewed. The reason is that at a highly skewed distribution, all queries will have the same cost and most of them will have the same selectivity, hence, for HR most queries will have the same priority. That is in contrast with FAS-MCQ which will utilize the extra information provided by the number of pending

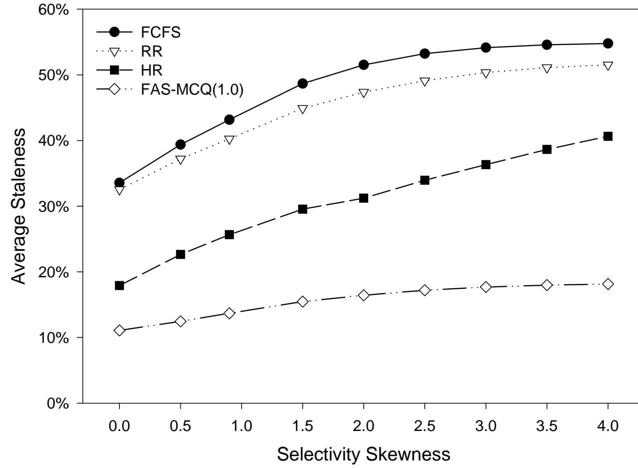

Figure 41: [§4.5.3] Staleness vs. skewness in selectivity (using Zipf parameter)

tuples to differentiate between queries and to assign higher priorities to queries that feed a stale stream or a stream that could be quickly brought to freshness.

4.5.4 Impact of Bursts

The setting for this experiment is the same as the default one. However, the utilization at all points is set to 95%. In Figure 42, we plot the average staleness as the number of input data streams that are bursty increases. At a value of 0, all the arrivals follow a Poisson distribution with no bursts, whereas at 10, all input data streams are bursty. Figure 42 shows the staleness of FAS-MCQ normalized to that of HR. The smaller the value the bigger the reduction.

Figure 42 shows that as the number of bursty streams increases, the reduction in staleness provided by FAS-MCQ compared to HR increases up until there are 5 bursty streams. At that point, FAS-MCQ reduces the staleness by 40%. After that point, the performance of the two policies gets closer. The explanation is that at a lower number of bursty streams, FAS-MCQ has a better chance to find a query with a short queue of pending updates to schedule for execution.

As the number of bursty streams increases, the chance of finding such a query decreases,

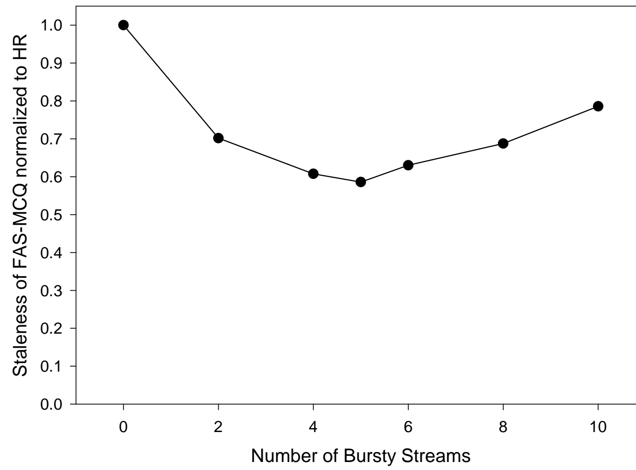

Figure 42: [§4.5.4] Staleness vs. number of bursty streams (out of 10)

and as such, HR is performing reasonably well. For instance, at 10 bursty streams, FAS-MCQ reduces the staleness by only 22% compared to HR.

4.5.5 Real Data

Figure 43 shows the results for our final experiment where we use real network traces. The selectivities and costs of operators are the same as in the first experiment. In this figure, the behavior of the different scheduling algorithms is consistent with the previous experiments, where FAS-MCQ provides the lowest staleness followed by HR, then RR and FCFS. Additionally, it shows the relatively high values of staleness exhibited by all policies, which is explained by the fact that the two traces are highly bursty, reflecting an ON/OFF traffic.

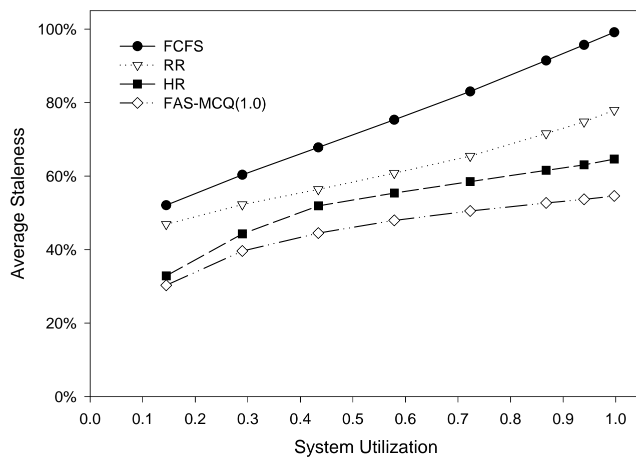

Figure 43: [§4.5.5] Staleness vs. system utilization (real data traces)

5.0 RELATED WORK

The growing need for monitoring applications has led to the development of several prototype DSMSs (e.g., [15, 44, 15, 23, 17, 31]). These prototypes utilize new techniques for the efficient processing of continuous queries over unbounded data streams. For example, [66] proposed rate-based query optimization as a replacement to the traditional cost-based approach. Also, new techniques for processing aggregate CQs appeared in [37], while techniques for processing join CQs appeared in [27].

For multiple queries, multi-query optimization has been exploited by [19] to improve system throughput in the Internet and by [42] for improving throughput in TelegraphCQ. TelegraphCQ uses a query execution model that is based on *eddies* [5]. In that model, the execution order of operators is determined at run-time. This is particularly important when the operators' costs and selectivities change over time. Similar to TelegraphCQ, our policies can work in a dynamic environment with support for monitoring the queries' costs and selectivities, and updating the priorities whenever it is necessary.

Sharing of common work is another important technique for multi-query optimization. That technique has been extended for optimizing multiple CQs by using group filters for processing common predicates [42] and it has also been exploited in processing multiple joins with different window specifications [30].

Load shedding is another mechanism that allows a DSMS to cope with high loads. Specifically, when the input load is beyond the DSMS's capacity, its performance deteriorates significantly as the system becomes unstable. The amount of deterioration depends on the input rate of the arriving streams and the duration of the instability status. As such, a *load shdder* is used to control the degree of degradation in the provided performance under overloaded conditions.

Overloading might occur when the processing requirements are beyond the DSMS's processing capacity (e.g., [62, 8]) or when the memory requirements are beyond the DSMS's memory capacity (e.g., [59]). In the former case, overloading is detected by the query processing engine when the input rate is higher than the output rate, whereas in the latter case, overloading is detected by the memory manager when the intermediate queues start overflowing.

The above techniques are also extended for distributed data stream processings (e.g., [64, 46]), where the main objective is assigning query operators to sites in a way that reduces communications costs which in turn reduces the overall query processing costs. This becomes particularly important in processing sensor data streams where communication between sensors involves significant energy consumption. Reducing that energy has been the focus of several research efforts including our previous work in [50, 11, 51].

Operator scheduling has been addressed in several research efforts (e.g., [65, 16, 6, 30, 61]). The work in [65] proposes the rate-based *(RB)* scheduling policy for scheduling operators within a single query to improve response time. Aurora [16] uses a policy called Min-Latency *(ML)* which is similar to the rate-based one; ML minimizes the average tuple latency in a single query. For multiple queries, Aurora uses a two-level scheduling scheme where Round Robin (*RR*) is used to schedule queries and *ML* (or *RB*) is used to schedule operators within the query.

Aurora also proposes a QoS-aware scheduler which attempts to satisfy application-specified QoS requirements. Specifically, each query is associated with a QoS graph which defines the utility of stale output; the scheduler then tries to maximize the average QoS. In this thesis, we focused on system QoS metrics that do not require the user to have any prior knowledge about the query processing requirements or to predict the appropriate QoS graph. Specifically, we developed policies that minimize the average response time as well as the average slowdown for multiple CQs that include join and shared operators. We also considered balancing the worst- and average-case performance, and presented policies to do so for response time and for slowdown.

Multi-query scheduling has also been exploited to optimize metrics other than QoS. For example, *Chain* is a multi-query scheduling policy that optimizes memory usage [6]. The

work on Chain has also been extended to balance the trade-off between memory usage and response time [7].

To the best of our knowledge, no previous work has proposed multi-query scheduling policies for improving the QoD provided by continuous queries. However, *load shedding* has been devised as a technique to control the degree of degradation in the provided QoD under overloaded conditions. The work in [62] describes a load shedding technique that decides which tuples to drop according to the importance of their content. The work in [8] formalizes the load shedding problem for aggregate queries.

Scheduling policies for improving the QoD has been studied in the context of replicated databases and in Web databases. For example, the work in [21, 22] provides policies for crawling the Web in order to refresh a local database. The authors make the observation that a data item that is updated more often should be synchronized less often. In this work, we utilize the same observation, however, [21, 22] assumes that updates follow a mathematical model, whereas we make our decision based on the current status of the Web server queues (i.e., the number of pending updates). The same observation has been exploited in [47] for refreshing distributed caches and in [36] for multi-casting updates.

The work in [34] studies the problem of propagating the updates to derived views. It proposes a scheduling policy for applying the updates that considers the divergence in the computation costs of different views. Similarly, our proposed *FAS-MCQ* considers the different processing costs of the registered multiple continuous queries. Moreover, *FAS-MCQ* generalizes the work in [34] by considering updates that are streamed from multiple data sources with different traffic patterns as opposed to a single data source.

6.0 SUMMARY AND FUTURE WORK

In this chapter, we first summarize the thesis contributions and then discuss potential avenues for future research.

6.1 SUMMARY

Motivated by the need to support monitoring applications which involve the processing of update streams by continuous queries, in this thesis, we considered the problem of scheduling multiple heterogeneous CQs in a DSMS with the goal of optimizing QoS and QoD for end users and applications.

To quantify such QoS we first used the traditional metric of response time, which we defined over multiple CQs, including CQs that contain joins of multiple data streams. We also considered slowdown as another QoS metric, since we believe it to be a more fair metric for heterogeneous workloads, and, as such, more suitable for a wide range of monitoring applications.

Having defined the QoS metrics to optimize, we developed new scheduling policies that optimize the average-case performance of a DSMS for response time and for slowdown. Additionally, we proposed hybrid policies that strike a fine balance between the average-case performance and the worst-case performance, thus avoiding starvation (which is crucial for event detection CQs).

Further, we have extended the proposed policies to exploit operator sharing in optimized multi-query plans and to handle multi-stream queries. We have also augmented the proposed policies with mechanisms that ensure their adaptivity to changes in workload. Finally, we

have evaluated our proposed policies and their implementation experimentally and showed that our scheduling policies consistently outperform previously proposed policies.

We also studied the different aspects that affect the QoD of monitoring applications. In particular, we focused on the freshness of the output data stream and identified that both the properties of queries, i.e., cost and selectivity, as well as the properties of the input update streams, i.e., variability of updates, have a significant impact on freshness.

Towards this, we proposed a new approach for scheduling multiple queries in Data Stream Management Systems. Our approach exploits both properties of queries and input data streams in order to maximize the freshness of output data streams. In particular, we proposed a new scheduling policy called *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)* and a weighted version of it that supports applications in which queries have different priorities. We also introduced a generalized variant of FAS-MCQ that balances the trade-off between QoD and QoS, according to application requirements.

We have experimentally evaluated our proposed FAS-MCQ policy against scheduling policies used in current DSMS prototypes as well as Web servers.

Table 6 lists the scheduling policies discussed above. For each policy, it states the optimization metric targeted by the policy. It also states if the policy is used in the context of a single query or multiple queries and whether or not the policy handles multi-stream queries that contain join operators.

Table 6: Classification of priority-based scheduling policies for CQs

Policy and Reference		Objective	Supported CQs		
			Single	Multiple	Join
RB	*Rate-based* [65]	Average Response Time	√	×	√
ML	*Min-Latency* [16]	Average Response Time	√	×	×
RR	*Round Robin* [16]	Average Response Time	√	√	×
HR	***Highest Rate*** §3.1.1.1	Average Response Time	√	√	√
HNR	***Highest Normalized Rate*** §3.1.3	Average Slowdown	√	√	√
FCFS	***First Come First Served*** §3.2.1	Maximum Response Time	√	√	√
LSF	***Longest Stretch First*** §3.2.1	Maximum Slowdown	√	√	√
BRT	***Balance Response Time*** §3.2.3	ℓ_2 norm of Response Time	√	√	√
BSD	***Balance Slowdown*** §3.2.2.2	ℓ_2 norm of Slowdown	√	√	√
Chain	Chain [6]	Maximum Memory usage	√	√	√
FAS	***Freshness-Aware Scheduling*** §4.2	Average Freshness	√	√	×

6.2 FUTURE WORK

6.2.1 Integrated Processing and Dissemination Schedulers

Current DSMSs prototypes do not provide an integrated data dissemination component. The assumption is that the underlying network layer is responsible for propagating the output data streams to end-users. However, that decoupling eliminates the chance of exploiting the CQs's characteristics for better bandwidth utilization.

Previous research on Publish/Subscribe information systems shows the importance of considering queries' semantics together with employing advanced data dissemination schemes such as *data multicast* and *data broadcast* (e.g., [3, 2, 1, 24, 25] as well as the work in [40, 39, 10]). In these schemes, data of interest for multiple clients is only disseminated once, thus making an effective use of the available bandwidth and allowing maximum scalability.

The same concept above can be applied in disseminating DSMS output data streams. That is, when multiple clients register the same CQ, the output of that query is multicasted only once. Additionally, results from overlapping CQ's can be efficiently merged to reduce the bandwidth consumption as we previously proposed in [52, 58, 53].

Towards this, we want to build on our own experience in multicast scheduling [52, 58, 53] to design an integrated cost-based stream processing and dissemination scheme that considers both the CQs' properties (i.e., operators' costs and selectivities) together with the output data properties (i.e., size and popularity).

6.2.2 Integrated Load Shedding

Load shedding techniques have been proposed to reduce the amount of work required to process input data streams [62, 8]. This is especially important when the DSMS is overloaded. Current load shedders decide to drop an input tuple based on its CQ's processing cost and selectivity, as well as on its effect on the overall QoS and QoD.

As shown in Chapters 3 and 4, the way queries are scheduled for execution also affects the overall QoD. Thus, we propose to investigate a collaborative scheme that utilizes the synergy between the load shedder and the query processor to improve QoS and QoD when

the server is overloaded. We also plan to consider the communication properties as another parameter in the design of the load shedder, for example, the cost of transmitting the output produced from processing the tuple.

BIBLIOGRAPHY

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1995.

[2] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *ACM/IEEE MobiCom*, 1998.

[3] D. Aksoy and M. Franklin. Rxw: A scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Tran. on Networkin*, 7(6):846–860, 1999.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The International Journal on Very Large Data Bases (VLDB J.)*, 15(2):121–142, 2006.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2000.

[6] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2003.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The International Journal on Very Large Data Bases (VLDB J.)*, 13(4), 2004.

[8] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2004.

[9] N. Bansal and K. Pruhs. Server scheduling in the l_p norm: A rising tide lifts all boats. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, 2003.

[10] J. Beaver, N. Morsillo, K. Pruhs, P. K. Chrysanthis, and V. Liberatore. Scalable dissemination: What's hot and what's not. In *Proc. of the ACM International Workshop on Web and Databases (WebDB)*, 2004.

[11] J. Beaver, M. A. Sharaf, A. Labrinidis, and P. K. Chrysanthis. Location-aware routing for data aggregation for sensor networks. In *Proc. of Geo Sensor Networks Workshop*, 2003.

[12] M. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *Proc. of the ACM Symposium on Discrete Algorithms (SODA)*, 2002.

[13] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proc. of the ACM Symposium on Discrete Algorithms (SODA)*, 1998.

[14] R. Brown. Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[15] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2002.

[16] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2003.

[17] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, V. Raman S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[18] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2002.

[19] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2002.

[20] J. Chen, D. J. DeWitt, F. Tian, and Y .Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2000.

[21] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2000.

[22] J. Cho and H. Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems (TODS)*, 28(4):390–426, 2003.

[23] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2003.

[24] A. Crespo, O. Buyukkokten, and H. G. Molina. Efficient query subscription processing in a multicast environment. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2000.

[25] A. Crespo, O. Buyukkokten, and H. G. Molina. Query merging: Improving query subscription processing in a multicast environment. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(1):174–191, 2003.

[26] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the ACM Symposium on Principles of Database Systems (PODS)*, 2001.

[27] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2003.

[28] L. Golab and M. T. Ozsu. Update-pattern-aware modeling and processing of continuous queries. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[29] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A. K. Elmagarmid. Efficient execution of sliding window queries over data streams. *Technical Report Number CSD TR 03-035, Purdue Univ.*, 2003.

[30] M. Hammad, M. Franklin, W. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2003.

[31] M. A. Hammad, M. F. Mokbel, M. H. Ali, Walid G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2004.

[32] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM International Conference on Communications Architectures and Protocols (SIGCOMM)*, 1988.

[33] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2003.

[34] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2001.

[35] A. Labrinidis and N. Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *The International Journal on Very Large Data Bases (VLDB J.)*, 13(3):240–255, 2004.

[36] W. Lam and H. Garcia-Molina. Multicasting a changing repository. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2003.

[37] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[38] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[39] W. Li, W. Zhang, V. Liberatore, V. Penkrot, J. Beaver, M. A. Sharaf, S. Roychowdhury, P. K. Chrysanthis, and K. Pruhs. An optimized multicast-based data dissemination middleware. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2003.

[40] V. Liberatore, P. K. Chrysanthis, and K. Pruhs. Middleware support for multicast-based data dissemination: A working reality. In *IEEE Workshop on Reliable Dependable Systems*, 2003.

[41] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[42] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[43] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 1993.

[44] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proc. of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[45] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[46] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2003.

[47] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[48] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1), 1988.

[49] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *Proc. of the ACM International Workshop on Web and Databases (WebDB)*, 2002.

[50] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis. Tina: A scheme for temporal coherency-aware in-network aggregation. In *Proc. of the ACM International Workshop on Mobile Data Engineering (MobiDE)*, 2003.

[51] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *The International Journal on Very Large Data Bases (VLDB J.)*, 13(4):384–403, 2004.

[52] M. A. Sharaf and P. K. Chrysanthis. Semantic-based delivery of olap summary tables in wireless environments. In *Proc. of the ACM International Conference on Information and Knowledge Management (CIKM)*, 2002.

[53] M. A. Sharaf and P. K. Chrysanthis. On-demand data broadcasting for mobile decision making. *Journal of ACM MobileNetworking and Applications (MONET)*, 9(6):703–714, 2004.

[54] M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Preemptive rate-based operator scheduling in a data stream management system. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, 2005.

[55] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *The International Journal on Very Large Data Bases (VLDB J.)*, 2006.

[56] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Transactions on Database Systems (TODS)*, to appear.

[57] M. A. Sharaf, A. Labrinidis, P. K. Chrysanthis, and K. Pruhs. Freshness-aware scheduling of continuous queries in the dynamic web. In *Proc. of the ACM International Workshop on Web and Databases (WebDB)*, 2005.

[58] M. A. Sharaf, Y. Sismanis, A. Labrinidis, P. K. Chrysanthis, and N. Roussopoulos. Efficient dissemination of aggregate data over the wireless web. In *Proc. of the ACM International Workshop on Web and Databases (WebDB)*, 2003.

[59] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2004.

[60] M. Sullivan. A stream database manager for network traffic analysis. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 1996.

[61] T. Sutherland, B. Pielech, Y. Zhu, L. Ding, and E. A. Rundensteiner. An adaptive multi-objective scheduling selection framework for continuous query processing. In *Proc. of the International Database Engineering and Applications Symposium (IDEAS)*, 2005.

[62] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2003.

[63] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1992.

[64] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2003.

[65] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2001.

[66] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[67] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of OSDI*, 1994.

[68] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the International Conference on Parallel and Distributed Information Systems (PDIS)*, 1991.