

NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

Application-Aware Resource Scheduling

by

Donald P. Carney

B. S., Rensselaer Polytechnic Institute, 1990

M. S., Polytechnic University, 1996

Sc. M., Brown University, 2001

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2005

UMI Number: 3174582

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 3174582

Copyright 2005 by ProQuest Information and Learning Company.

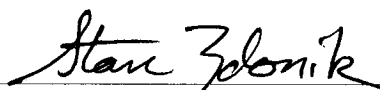
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© Copyright 2002-2005 by Donald P. Carney

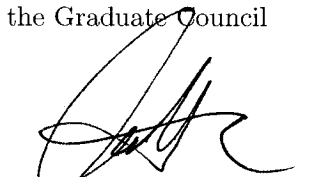
This dissertation by Donald P. Carney is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date 5/6/05

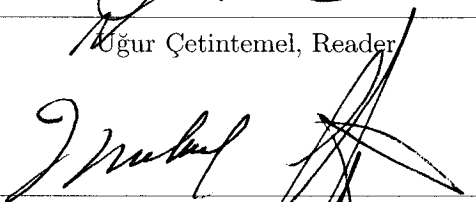

Stan Zdonik, Director

Recommended to the Graduate Council

Date 5/6/05



Uğur Çetintemel, Reader

Date 5/15/05


Michael Stonebraker, Reader
Massachusetts Institute of Technology

Approved by the Graduate Council

Date 5/20/05


Karen Newman
Dean of the Graduate School

Abstract of “Application-Aware Resource Scheduling” by Donald P. Carney, Ph.D., Brown University, May 2005.

The proliferation of the Internet and World Wide Web, the development of wireless networks, the availability of asymmetric high-bandwidth links to the home, and the recent emergence of small-scale computing devices have fueled the development of applications with wide and varied data needs. The data-intensive environments in which these applications exist are characterized by very high data rates, the scale and variety of autonomous data sources, and the diverse needs of users and applications. Often, the resources (CPU, memory, bandwidth, etc.) in these environments are constrained and data management plays an important role in providing fast responsive systems through the effective scheduling of resources.

Databases have long benefited from the use of application-level knowledge to make decisions. Often, this application-level awareness comes from *a priori* knowledge of user interests. In these new networked environments there is evidence that more responsive systems can be delivered if users provide either some general indication of their interest or even better, a detailed query. We call this specification of user interest a “profile”.

This dissertation focuses on two applications domains. In the first application domain, the process of synchronizing a database to improve the freshness of copies is studied. In the second application domain, we study the timely processing of large volumes of continuous high-rate data streams in the context of the Aurora stream processing engine. In both cases, profiles show promise in improving resource scheduling to provide effective data management.

Vita

Donald Carney was born on January 7th, 1968 in Valley Stream, New York. He completed secondary school at Chaminade High School in Mineola, New York and attended Rensselaer Polytechnic Institute in Troy, New York where he received his Bachelor of Science in Interdisciplinary Science. He worked as a software engineer for Porta Systems Corporation in Syosset, New York while working part time to complete a Master of Science in Computer Science at Polytechnic University, Brooklyn, New York. He joined the doctoral program in the Computer Science Department at Brown University in Providence, Rhode Island and he received an Sc.M. degree in Computer Science in May, 2001.

Refereed Articles

- *Retrospective on Aurora*, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts and S. Zdonik. Submitted to VLDB Journal Special Issue on Data Stream Processing, 2004.
- *Operator Scheduling in a Data Stream Environment*, D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack and M. Stonebraker, Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), September, 2003.
- *Aurora: A New Model and Architecture for Data Stream Management*, D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, VLDB Journal, Vol. 12, No. 2, August, 2003.

- *Reducing Execution Overhead in a Data Stream Manager*, D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack and M. Stonebraker, Proceedings of the ACM Workshop on Management and Processing of Data Streams (MPDS 03), June, 2003.
- *Aurora: A Data Stream Management System (Demonstration)* D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan and S. Zdonik, Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, CA, June, 2003.
- *Scalable Application Aware-Data Freshening*, D. Carney, S. Lee, S. Zdonik. IEEE International Conference on Data Engineering (ICDE), March 2003
- *Index Hint for On-Demand Broadcasting (Poster)* S. Lee, D. Carney, S. Zdonik. IEEE International Conference on Data Engineering (ICDE), March 2003
- *Scalable Distributed Stream Processing*. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, S. Zdonik. First Semiannual Conference on Innovative Data Systems Research (CIDR), Asilomar, CA. Jan. 2003
- *Monitoring Streams: A New Class of Data Management Applications* D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. In proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02), August 20-23, Hong Kong, China.

Acknowledgements

I owe a debt of gratitude to my colleagues, friends and family for their support during my doctoral work at Brown. I would first like to thank my advisor, Stan Zdonik, who has been a great mentor and a good friend during my growth as a database researcher. A special thanks to Ugur Cetintemel for his contributions and advice on Aurora scheduling. To Mike Stonebraker for his advice and very practical perspective on research. To the faculty of the Computer Science department for providing a wonderfully collegial environment, especially Tom Doeppner, Steve Reiss, and David Laidlaw

I am especially grateful for Professor Sangdon Lee's contributions to our work on Data Freshening described in Chapter 2.

I would also like to thank my fellow students, past and present, in the database research group at Brown, including Alex Rasin, Nesime Tatbul, Jeong-Hyon Hwang, Ying Xing, Narayanan Ramachandran, Mitch Cherniack, Kerry Kurian, and Rahul Bose.

The Brown Computer Science Department is a great place to work. I would like to thank the Technical and Administrative staff of the department including John Bazik, Tom Heft, Max Salvas, Jeff Coady, Lori Agresti, and Fran Palazzo. To my many colleagues within the Computer Science department who provided me with not only academic and research assistance, but also many social outlets.

Also a special thanks to my many buddies with whom I enjoyed the occasional beer, ice cream, and continual involvement in intramural sports. Special thanks to Joe Laviola, David Gondek, Steven Dollins, Keith Hall, Bob Zeleznik, Dom Bhuphaibool, Markus Meister, Sidhartha Bhatia, Harsh Kumar, Didier Lucor, and Adriana Meszaros.

This work would not have been possible without the unyielding support and encouragement of my family. Thanks Mom, Dad, Bill, Mary, and Tara.

Contents

List of Tables	xii
List of Figures	xiii
1 Introduction	1
1.1 A Brave New World	1
1.2 The Problem: Application-Aware Resource Scheduling	2
1.2.1 Data Freshening	2
1.2.2 Data Stream Management Systems	4
1.3 Thesis organization	6
2 Resource Scheduling for Data Refreshing	7
2.1 Introduction	7
2.2 Perceived Freshness	8
2.2.1 Problem Definition	10
2.2.1.0.1 Core Problem	11
2.2.2 The Optimal Solution	11
2.2.2.1 Some Intuition	13
2.2.2.1.1 Example	13
2.2.2.2 Solving Optimally for Small Cases	14
2.3 Scalability and Heuristic Approaches	15
2.3.1 Partitioning Alternatives	17

2.3.1.0.1	<i>p</i> -Partitioning	17
2.3.1.0.2	λ -Partitioning	18
2.3.1.0.3	$\frac{p}{\lambda}$ -Partitioning	18
2.3.1.0.4	<i>PF</i> -Partitioning	18
2.3.2	Optimization	18
2.4	Experiments	20
2.4.1	Heuristic Case - Comparing the Techniques	21
2.4.1.1	Quality of the Result	21
2.4.1.2	Scalability of the Techniques	23
2.4.1.3	An Additional Improvement	24
2.5	Object Size Considerations	26
2.5.1	Problem Extension	26
2.5.2	Freshening Approaches	26
2.5.3	Results and Further Analysis	27
2.6	Time average of perceived freshness	30
2.7	Solution for the Core Problem	31
2.8	Summary	32
3	Resource Scheduling for the Management of Data Streams	33
3.1	Introduction	33
3.2	Aurora Overview	34
3.2.1	Aurora System Model	34
3.2.2	Architecture	35
3.2.3	Basic Execution Model	36
3.3	Two-Level Scheduling	38
3.3.1	Operator Batching - Superbox Processing	39
3.3.1.1	Superbox Selection	40
3.3.1.2	Superbox Traversal	41
3.3.2	Tuple Batching - Train Processing	46

3.4	Experimental Evaluation	47
3.4.1	Experimental Testbed	47
3.4.2	Operator Batching - Superbox Scheduling	48
3.4.3	Superbox Traversal	49
3.4.4	Tuple Batching - Train Scheduling	51
3.4.5	Overhead Distribution	53
3.5	Summary	54
4	Scheduling for QoS in Aurora	55
4.1	Introduction	55
4.2	QoS-Aware Scheduling	56
4.2.1	Static Schedules	56
4.2.2	Dynamic Schedules	58
4.3	Comparing Fixed-Priority and SlopeSlack	60
4.3.1	A Look at the Run-time Schedules	61
4.3.2	Meeting QoS Deadlines as Load Increases	63
4.4	Overhead Reduction	66
4.4.1	Schedule Size Matters	66
4.4.2	Approximation for Scalability	68
4.4.2.1	Time Complexity	70
4.4.3	Bucketing Results	72
4.5	Superboxes: Immediately Pushing Tuples to Outputs	73
4.5.1	Very Large Query Networks with Superboxes and Approximation	78
4.6	Overload	80
4.6.1	Sustained Overload	81
4.6.2	Fluctuating Workload	84
4.7	Sensitivity to Network Topology	89
4.8	Summary of QoS Scheduling for Aurora	92
4.9	Conclusion	93

5	Related Work	95
5.1	Related Work - Freshness	95
5.2	Related Work - Aurora Scheduling	97
6	Future Work	100
6.1	Introduction	100
6.2	Future research directions for Data Freshening	100
6.2.1	Expected Freshness	100
6.2.2	Is bandwidth free?	101
6.2.3	Gathering Statistics	102
6.2.4	Split Sync Bandwidth for a Dynamic Synchronization Policy	102
6.2.5	Pushing to reduce staleness in conjunction with synchronization of cache	104
6.2.6	Page Replacement	105
6.2.7	Multiple levels of cache	105
6.3	Future research directions for Aurora Scheduling	106
6.3.1	Selecting Superboxes	106
6.3.2	Superbox Priority Assignment	108
6.3.3	Superbox Compositing	108
6.3.4	Actual Priority Assignment	108
6.3.5	More interesting queries	108
6.3.6	Proposed Scheduling Algorithms	109
6.3.6.1	Heuristics for choosing superboxes	109
6.3.6.1.1	Static Heuristics	110
6.3.6.1.2	Dynamic Heuristics	111
6.3.7	Dealing with high cost boxes	113
6.3.8	Superbox Priority Assignment	114
	Bibliography	115

List of Tables

2.1	Optimal sync frequencies for example	14
2.2	Setup for Ideal Experiments	16
2.3	Setup for Partitioning Experiments	23
3.1	Stepping Through Min-Cost Traversal	42
3.2	Stepping Through Min-Latency Traversal	43
3.3	Costs, Selectivities, and <i>mem_rr</i> for Min-Memory Traversal Example	46
4.1	Box Priorities for Fixed-Priority Scheduling Example	58
4.2	Experimental Setup	64
4.3	Bucketing Examples	70
4.4	Experimental Setup for Large Network	78
4.5	Scheduling overhead as a percentage of run-time	79
4.6	This table shows the QoS results for Figures 4.16 and 4.17	83
4.7	Summary of Missed Deadlines	89

List of Figures

2.1	Relationship among f, λ, p	12
2.2	Alignment options	15
2.3	The Ideal Case for Perceived Freshness and General Freshness	16
2.4	Simulation Model	20
2.5	Comparing the Techniques	22
2.6	Sensitivity of Partitioning to Zipf Skew (θ) in Shuffle-Change Alignment . .	23
2.7	Big Case	24
2.8	Improvement in Perceived Freshness after Clustering	25
2.9	Improvement in Perceived Freshness after Clustering	26
2.10	Optimal Sync Resource Distribution	28
2.11	Sync Allocation	29
3.1	Aurora System Model	34
3.2	Aurora Run-Time	36
3.3	High-level comparison of stream execution models	37
3.4	Sample Query Graph	40
3.5	Sample Query Tree	40
3.6	Box vs. Application Scheduling	49
3.7	Box vs. Application Scheduling	50
3.8	Memory requirements of traversal algorithms	51
3.9	Train scheduling effects	52

3.10	Distribution of execution overheads	53
4.1	Critical points and expected output delay	59
4.2	Tuple Latencies for Fixed Scheduler	63
4.3	Tuple Latencies for SlopeSlack Scheduler	63
4.4	Round Robin Scheduler loses to Slope-Slack Scheduler, but, Fixed beats Slope-Slack	65
4.5	Time Spent Scheduling	67
4.6	Schedule Size has an impact on the performance of scheduling algorithms	68
4.7	Illustrating (a) bucket traversal, (b)gradient-latency graphs, and (c) slack-latency graphs	69
4.8	QoS graphs for Bucketing Example	71
4.9	Possible Slack Bucket Assignments for Bucketing Example	71
4.10	Possible Slope Bucket Assignments for Bucketing Example	71
4.11	Buckets for Bucketing Example	72
4.12	Bucketing effects on QoS	73
4.13	Bucketing overheads	74
4.14	Scheduling superboxes	77
4.15	Bucketing makes best scheduling decisions for large query networks	79
4.16	FixedPushThrough	83
4.17	SlopeSlackPushThrough	83
4.18	Fluctuating Workload: Running Average QoS	87
4.19	Fluctuating Workload: Running Recent Average QoS	87
4.20	Fluctuating Workload: All missed deadlines	88
4.21	Fluctuating Workload: FixedPriorityPushThrough	88
4.22	Fluctuating Workload: SlopeSlackPushThrough	89
4.23	Fanout Query	90
4.24	Straight Query	90
4.25	QoS results for Fanout Query	91

4.26	QoS results for Straight Query	92
6.1	Bandwidth/Perceived Freshness Tradeoff	102
6.2	Bandwidth/Perceived Freshness Tradeoff	104
6.3	Levels of Cache	105
6.4	Example Aurora Network with overlapping queries	106
6.5	High Cost Windowed Operator Example	107
6.6	Example Aurora SubNetwork	112

Chapter 1

Introduction

1.1 A Brave New World

The proliferation of the Internet and World Wide Web, the development of wireless networks, the availability of asymmetric high-bandwidth links to the home, and the recent emergence of small-scale computing devices have fueled the development of applications with wide and varied data needs. The data-intensive environments in which these applications exist are characterized by the amount and rate of data flowing, the scale and variety of data sources, and the specific needs of the applications. Data Management has played an important role in dealing with these new environments. Recent research has included work in data-dissemination technology, continuous query systems, and data stream management systems.

Managing resources has always been an issue as Data Management adapts to changing technology. Generally, the environments in which resources are constrained provide the most interesting problems. The limited resources can be in the machines which are making the decisions, for example, CPU, disk, or memory. Or, the resource can be the bandwidth available between machines.

Whenever resources are constrained, scheduling plays an important role in providing fast responsive systems through the effective use of resources. Multiprogrammed operating

systems use process scheduling to maximize CPU utilization. Disk controllers re-order read requests to improve performance. Real-time systems schedule tasks in an attempt to satisfy deadlines. Databases schedule transactions to run in parallel using both the CPU and I/O system to improve throughput and resource utilization.

1.2 The Problem: Application-Aware Resource Scheduling

Databases have long benefited from the use of application-level knowledge to make decisions. For example, database indices are built on attributes that are known to be commonly used. Often, this application-level awareness comes from *a priori* knowledge of user interests. In modern networked environments such as the internet there is evidence that more responsive systems can be delivered if users provide either some general indication of their interest or even better, a detailed query.

Many research and commercial systems rely on user input to make data management decisions. Tivo users specify their viewing interests and the local Tivo box manages a cache of television programs. MyCNN and MyYahoo allow users to personalize their web interfaces to news and information. Several financial data sources allow users subscribe to stock feeds and are promised a certain level of "freshness" of the data (e.g. quotes that are no older than 20 minutes, and, in some cases users can receive "real-time" quotes) These systems provide a higher level of satisfaction to users by allowing them to specify interests.

This thesis focuses on two environments which can benefit from the use of application-level knowledge to make decisions about scheduling resources.

1.2.1 Data Freshening

Replication is a well-known data management technique for improving the performance and availability of a distributed computing system. Internet mirror sites geographically distribute the load for popular information sources. Data warehouses replicate data from the transactional database in order to provide a platform for decision support queries that does not interfere with the originating source.

Of course, as updates occur at the underlying data sources, the replicas can drift out of synch. In tightly coupled distributed systems, algorithms exist for atomically installing an update to all replicas. In more loosely coupled systems like the Internet, heavy-weight synchronization schemes are not feasible. Instead methods that respect the independence of the participants are much more common. For example, a site that stores copies may itself be responsible for detecting changes in the underlying source and for updating its local copy accordingly.

While a good solution might involve an active server that pushes updates to the mirror, most existing data providers do not support this. Without cooperation from the source, the site with a local copy must poll the source in order to detect changes. If this is done too often with respect to the update frequency, resources are wasted. If it does not occur often enough, data freshness suffers. The goal, then, is to pick an appropriate polling frequency for each local copy that balances these two needs.

Data management has always benefited when we can use application-level knowledge to make decisions. For example, database indices are built on attributes that are known to be commonly used. In modern networked environments such as the Internet, there is evidence that more responsive systems can be delivered if users supply a description of their interests (a.k.a. a *profile*). Consider MyYahoo and Tivo, the personal video recording system. In these systems, users supply profiles to create personalized content; however, personalization is only one application of profiles. In the case of Tivo, the profile is also used as a way to manage the local cache in the Tivo box. This notion can be extended to a wide range of data management situations. In particular, we show that profiles can be used effectively to control replica management policies.

Others [CGM00b] have studied techniques for managing the process of updating local copies. In these studies, the goal has been to maximize the average freshness of a collection of copies. This is accomplished by producing a schedule for polling master copies to see if they have changed. For our contribution, we will show that there is much to be gained from taking user interests into account when producing a schedule for freshening. Intuitively, it

makes little sense to spend freshening resources on items that no one wants.

1.2.2 Data Stream Management Systems

Applications that deal with potentially unbounded, continuous streams of data are becoming increasingly popular due to a confluence of advances in real-time, wide-area data dissemination technologies and the emergence of small-scale computing devices (such as GPSs and micro-sensors) that continually emit data obtained from their physical environment. Example applications include sensor networks, position tracking, fabrication line management, network management, and financial portfolio management. All these applications require timely processing of large volumes of continuous, potentially rapid and asynchronous data streams.

We have designed a system called Aurora [CCC⁺02], a data stream manager that addresses the performance and processing requirements of stream-based applications. Aurora supports multiple concurrent continuous queries, each of which produces results to one or more stream-based applications. Each continuous query consists of a directed acyclic graph of a well-defined set of operators (or boxes in Aurora terminology). Applications define their service expectations using Quality-of-Service (QoS) specifications, which guide Aurora's resource allocation decisions.

A key component of Aurora, or any data stream management architecture for that matter, is the operator scheduler that decides which operators to execute and how long to execute them. A naïve approach to scheduling would pick a tuple to run through an operator, schedule that operator, then loop back to pick another tuple. For operators with very low execution costs (e.g. filter or union), this tuple-at-a-time approach would experience serious performance problems because it does not take various execution and scheduling overheads into account. When operator costs are low, such overheads become dominant and scheduling without considering overhead leads to system degradation.

The traditional model for structuring database servers is thread-based execution, which is supported widely by traditional programming languages and environments. The basic

approach is to assign a thread to each query or operator. The operating system (OS) is responsible for providing a virtual machine for each thread and overlapping computation and I/O by switching among the threads. The primary advantage of this model is that it is very easy to program, as the OS does most of the job. On the other hand, especially when the number of threads is large, the thread-based execution model incurs significant overhead due to cache misses, lock contention, and switching. More importantly for our purposes, the OS handles the scheduling and does not allow the overlaying software to have fine-grained control over resource management.

We suggest using a state-based execution model. In this model, there is a single scheduler thread that tracks system state and maintains the execution queue. The execution queue is shared among a small number of worker threads responsible for executing the queue entries, where each entry is a sequence of operators belonging to a (sub-)query. This state-based model avoids the mentioned limitations of the thread-based model, enabling fine-grained allocation of resources according to application-specific targets (such as QoS). Furthermore, this model also enables effective batching of operators and tuples, which we expect to have a significant effect on system performance as it cuts down the scheduling and box execution overheads.

An important challenge with the state-based model is that of designing an intelligent but low-overhead scheduler. In this model, the scheduler becomes solely responsible for keeping track of system context and deciding when and for how long to execute each operator. In order to meet application-specific QoS requirements, the scheduler should carefully multiplex the processing of multiple continuous queries. At the same time, the scheduler should try to minimize the system overheads, time not spent doing useful work (i.e., processing), with no or acceptable degradation in its effectiveness.

1.3 Thesis organization

- Chapter 2 : Resource Scheduling for Data Refreshing
- Chapter 3 : Resource Scheduling for the Management of Data Streams
- Chapter 4 : QoS-Aware Scheduling in Aurora
- Chapter 5 : Related work
- Chapter 6 : Future work
- Bibliography

Chapter 2

Resource Scheduling for Data Refreshing

2.1 Introduction

Distributed databases and other networked information systems use copies or mirrors to reduce latency and to increase availability. Copies need to be refreshed. In a loosely coupled system, the copy sites are typically responsible for synchronizing their own copies. This involves polling and can be quite expensive if not done in a disciplined way. This chapter explores the topic of how to determine a refresh schedule given knowledge of the update frequencies and limited bandwidth. The emphasis here is on how to use additional information about the aggregate interest of the user community in each of the copies in order to maximize the perceived freshness of the copies. A model and optimal solution are developed for small cases. Next, several heuristic algorithms that work for large cases are presented. Then, the impact of object size on the refresh schedule is explored. This chapter also presents experimental evidence that our algorithms perform quite well.

2.2 Perceived Freshness

We are concerned with the problem of keeping a set of copies as up to date as possible with respect to a master data source. We will speak of the set of copies as the *mirror* and the master data source as the *source*. A single member of the mirror will be referred to as a *local copy*. The computer that is responsible for maintaining the currency of the mirror will be called the *mirror site*. Intuitively, the *freshness* of a mirror is the fraction of its local copies whose value reflects the current value in the source. It is important to note that copies are not added or deleted at the mirror.

The mirror site is connected to the source with a fixed and limited bandwidth. The limitation in bandwidth will determine how many refresh operations can be performed per unit time. The work in this chapter is concerned with planning the best use of this bandwidth by determining a refresh schedule that will maximize the freshness of the mirror.

Our work follows closely from the work presented in [CGM00b]. Thus, we briefly summarize their notion of freshness for a local database S consisting of N elements ($S = e_1, \dots, e_N$). It should be noted that as in [CGM00b], we initially assume all objects have the same fixed size and, therefore, take up the same amount of space in the mirror. In Section 2.5, we relax this assumption to determine the impact of object size.

Definition 1 *The freshness of a local element e_i at time t is*

$$F(e_i; t) = \begin{cases} 1 & ; \text{if } e_i \text{ is up-to-date at time } t \\ 0 & ; \text{otherwise} \end{cases}$$

Definition 2 *The freshness of a local database S at time t*

$$F(S; t) = \frac{1}{N} \sum_{i=1}^N F(e_i; t)$$

Freshness of an element is binary. At any instant, the element at the mirror is either up-to-date (1) or not (0). The freshness of a local database at any instant is the average freshness of all elements at the local database at that instant.

The freshness metrics given above make perfect sense for the case in which no knowledge of the access patterns exists. After all, in the absence of profiles, it is reasonable to assume

that all accesses to local copies are equally likely. However, if all accesses are not equally likely, we can define a freshness metric that weights the contribution of each local copy by its popularity (expected number of accesses per unit time). We call our metric *perceived freshness*. Perceived freshness captures what users see as they use the mirror. *Average Perceived Freshness* measures the freshness that users observe as they access various portions of the mirror. If a given item is never accessed, it does not contribute to the average perceived freshness regardless of how stale its value is.

We assume that many users frequently access the mirror site. In order to improve their chances of getting fresh data, they each submit a *profile*. Simply stated, a profile is a declarative specification of the relative importance of each copy in the mirror. This could be written in a sophisticated profile language [CFZ01]; however, for our purposes, we assume that the profile is a distribution of access frequencies. If the access frequencies are scaled by the total number of accesses per unit time, the profile can be viewed as an access probability distribution. The assumption is that a user will access important items more often than unimportant items. This is, of course, not in general required, but it simplifies our discussion.

The mirror collects all the user profiles and aggregates them into one master profile that is a combined frequency distribution for all users. Our algorithms make decisions based on the master profile. Thus, we use the term *profile* in what follows, to refer to the master profile.

It is important to note that our profile model is a very simple one. We assume that user interest in an element x is directly proportional to the frequency with which that user accesses x . This is clearly not always the case. We can imagine more sophisticated profiles [CFZ01], but find it useful for the purposes of this study to restrict our attention to a limited, but plausible model. It should also be noted that it is not difficult to modify our profile model to be a density function over any measurable attribute of our objects. For example, in a stock market application, a profile might be cast as a plot of importance vs. ticker symbol. Also, individual profiles can be weighted before they are aggregated into the

master profile, so as to give higher priority to more important users (e.g., generals or higher paying customers).

2.2.1 Problem Definition

Suppose that over some period of time the mirror experiences a set of M accesses called the *access set* that we denote as $A = a_1, \dots, a_M$. A given a_j names an element e_i in the mirror. The access set A can contain more than one access to a given element. If we denote the number of accesses for an element e_i as M_i , then $M = \sum_{i=1}^N M_i$. It follows that each element e_i has an access probability, $p_i = \frac{M_i}{M}$ (note that $\sum_{i=1}^N p_i = 1$).

While average overall freshness is an interesting system metric, it is not that relevant to individual users. A user might rate the quality of the mirror by “keeping score” at each access. If the user gets a fresh copy, score a point. If the user gets a stale copy, score nothing. In this way, the score would reflect how well the system is freshening those items that user actually references. *Perceived freshness* captures this idea.

Let $F(a_i; t)$ be the freshness of the element referenced by access a_i . We define *perceived freshness* as follows.

Definition 3 *The perceived freshness of a set of accesses A at time t is*

$$PF(A; t) = \frac{1}{M} \sum_{i=1}^M F(a_i; t)$$

Intuitively, $PF(A; t)$ is the fraction of accesses that see an up-to-date copy.

Definition 4 *The time averaged perceived freshness of a set of local database accesses A is*

$$\overline{PF}(A) = \lim_{t \rightarrow \infty} \frac{1}{t} \int_0^t PF(A; t) dt$$

The time averaged perceived freshness for a mirror S that experiences an access set A is $\overline{PF}(S)$. It is this last quantity, $\overline{PF}(S)$, that is important to our algorithms. From the definitions, we can prove that $\overline{PF}(S) = \overline{PF}(A) = \sum_{i=1}^N p_i \overline{F}(e_i)$ where $\overline{F}(e_i)$ is the time-averaged freshness of element e_i . (For a detailed proof, refer to Section 2.6 and [CLZ02]).

We assume that it is possible to obtain the number of updates to an element over some time period. Prior work has shown how the source can use estimation [CGM00a] and sampling [CN02] techniques to obtain a good estimate of these update frequencies. These frequency estimates would be periodically communicated to the mirror.

Given these definitions and assumptions, we can now formulate our problem more precisely.

2.2.1.0.1 Core Problem Given an update frequency λ_i for each element at the source and an access probability p_i for each of the local copies in the mirror ($i = 1, 2, \dots, N$), find the refresh frequencies f_i ($i = 1, 2, \dots, N$) which maximize

$$\overline{PF}(S) = \sum_{i=1}^N p_i \overline{F}(e_i) = \sum_{i=1}^N p_i \overline{F}(f_i, \lambda_i) \quad (2.1)$$

satisfying the following constraint

$$\sum_{i=1}^N f_i = TotalBandwidth \quad \text{and} \quad f_i \geq 0 (i = 1, 2, \dots, N) \quad (2.2)$$

In equation 2.1, we write $\overline{F}(f_i, \lambda_i)$ to represent the time averaged freshness for e_i given its synchronization frequency f_i and its change rate λ_i .

Note that the bandwidth is given in terms of the number of refreshes that are allowed over some time period. The refresh frequencies f_i are computed as the number of refreshes of local copy i that should be scheduled over the same time period.

2.2.2 The Optimal Solution

In this section, we examine the optimal solution to the core problem.

Solving the core problem produces a bandwidth allocation for each local copy. Given such an allocation, we must further determine how those allocations should be ordered in time. In [CGM00b], a Fixed Order synchronization-order policy is shown to give the best performance. In this policy, all objects are synchronized in the same order repeatedly, each at a fixed interval. Since we can determine a closed form of $p_i \overline{F}(\lambda_i, f_i)$, it is possible to solve

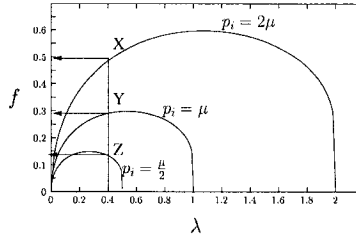


Figure 2.1: Relationship among f , λ , p

the core problem by the method of Lagrange multipliers. We show the general solution of the problem in Section 2.7.

To solve the problem we must solve an optimization problem with a non-linear objective function and with a linear constraint on the overall available bandwidth. This problem is solved using non-linear programming to produce a schedule with a fixed synchronization rate for each local copy. Such a solution will optimally produce a schedule that maximizes the freshness of the mirror given the constraint on bandwidth. Since the complexity of non-linear programming is bounded by a high-order polynomial, this approach is feasible for small sets of objects only.

From the solution, we can produce the graph in Figure 2.1. We can understand the relationships among f_i , λ_i and p_i better by examining the graphs for various p_i 's. Figure 2.1 shows three curves on which solutions are located respectively for objects with access probabilities $p_i = \{\frac{\mu}{2}, \mu, 2\mu\}$. Again, we assume the Fixed Order refresh policy. It clearly shows that for any given change rate (λ) an element e_i needs to get more bandwidth as its access probability p_i increases. For example, a point Y on the graph for $p_i = \mu$ moves to point X as p_i is increased to $p_i = 2\mu$. Again, starting from point Y , as p_i is decreased to half its original value, we move to point Z . However, note that increasing p_i means that some other $p_j (i \neq j)$ needs to be decreased since $\sum p_i = 1$. Elements with large p_i will get higher f_i than elements with lower p_i and the same λ . In Figure 2.1, an element with $\lambda = 1.2$ does not get any bandwidth when $p_i = \mu$. However, it will require significant bandwidth as its p_i becomes 2μ . This suggests the general principle that elements with large λ_i need increasing bandwidth as their p_i 's increase.

2.2.2.1 Some Intuition

Here we illustrate the characteristics of the solution for perceived freshness versus the solution given in [CGM00b] for average freshness which ignores user interest. We will base these observations on a toy example that was used in [CGM00b].

2.2.2.1.1 Example The database consists of five equal-sized elements, which change at the frequencies of $1, 2, \dots, 5$ (times/day). Let's assume that there are 3 sets of access probabilities for these five elements as follows. $P1 = (1/5, 1/5, 1/5, 1/5, 1/5)$, $P2 = (1/15, 2/15, 3/15, 4/15, 5/15)$ and $P3 = (5/15, 4/15, 3/15, 2/15, 1/15)$. $P1$ represents a uniform access distribution for the five elements, while $P2$ and $P3$ represent skewed access distributions. $P2$ represents the case for which the hottest elements change most frequently. $P3$ represents the opposite case. We assume that freshening the local database can only occur at a rate of 5 elements/day (the bandwidth constraint). We can solve this example problem numerically to determine the best freshening schedule.

Table 2.1 shows the results. Row (a) in table 2.1 shows change frequencies for each element. Row (b) shows the optimal synchronization frequencies for the uniform distribution case ($P1$) which coincide with the results in [CGM00b]. However, as rows (c) and (d) show, optimal synchronization frequencies for non-uniform access distributions are much different from the result for the uniform distribution case. Especially in row (c), for which the most frequently changing element (e_5) now needs to get the highest synchronization frequency (instead of 0 as in row (b)). This implies that frequently changing elements may need to be synchronized much more than less frequently changing elements if they are more likely to be accessed. This case, where users are more interested in items which change frequently, is not uncommon in practice (for example, volatile stocks might be more interesting to day-traders purely due to their volatility). In the case of less frequently changing elements, row (d) shows that they should receive more freshening bandwidth if the access frequency for them is high. □

Table 2.1: Optimal sync frequencies for example

	e_1	e_2	e_3	e_4	e_5
(a)change freq	1	2	3	4	5
(b)sync freq (P1)	1.15	1.36	1.35	1.14	0.00
(c)sync freq (P2)	0.33	0.67	1.00	1.33	1.67
(d)sync freq (P3)	1.68	1.83	1.49	0.00	0.00

We will show how differences in optimal synchronization frequencies such as those illustrated above will affect the perceived freshness in Section 2.2.2.2.

2.2.2.2 Solving Optimally for Small Cases

We can solve the synchronization problem optimally for a small number of objects. Table 2.2 shows the setup we use. As in [CGM00b] we assume a gamma distribution for the distribution of change frequencies of database objects. We use a Zipfian ¹ distribution to model user profiles from which we generate a master profile of user interest. It has been shown in practice that θ can be as high as 1.6 [PQ00]. The example we use is a database with 500 objects and each object is updated on average twice during a synchronization period (the per interval update rate of individual objects is determined by the gamma distribution - the mean of the gamma is 2). We provide a sensitivity analysis study of these parameters in [CLZ02]. We assume no relationship between these parameters and object size, and, for now, assume that all objects have the same size (we relax this restriction later).

In this section the GF technique refers to General Freshening (proposed in [CGM00b]) which optimizes average freshness and the PF technique refers to Perceived Freshening (our approach). In both cases, we measure the average perceived freshness achieved with the resulting schedule. While it is not too surprising that the Perceived Freshening technique does better when you measure perceived freshness, it is interesting to note exactly where the benefits lie.

¹A Zipfian distribution is commonly used to model non-uniform data access. As its parameter θ increases, the access pattern becomes increasingly skewed. The probability of accessing any object i is proportional to $(1/i)^\theta$.

We chose three possible alignments of change and access frequency. Figure 2.2 shows two of these configurations. Note that the shapes of the curves in the figure are for demonstration purposes and that their actual shapes would be determined by their distributions. In the *aligned* case, objects that change frequently are of highest interest and in the *reverse* case, volatile objects are less interesting to users. A third case shuffles the change rates from the gamma distribution randomly across the elements so that no relationship exists between interest and updates. We call this *shuffled-change*.

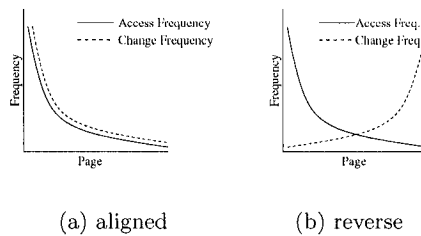


Figure 2.2: Alignment options

Figures 2.3(a)-2.3(c) shows the perceived freshness results. The graphs all share the same characteristic: when the users' interests are considered in the synchronization algorithm, the users' perceived freshness increases as users' interests become more skewed. Note that in all cases, when $\theta = 0$, perceived freshness equals general freshness. This is because $\theta = 0$ corresponds to a uniform distribution. Both synchronization techniques produce the exact same schedule because all data items have the same access probability.

As expected, the PF technique outperforms the GF technique as the interest skew increases. The most significant difference is the aligned case in which perceived freshness approaches 0 for high interest skew when user interest is ignored.

2.3 Scalability and Heuristic Approaches

While the optimization of *freshness* and *perceived freshness* presented in section 2.2 are interesting for studying the effectiveness of refresh algorithms, they are not accurately computable in practice for large problems.

Table 2.2: Setup for Ideal Experiments

NumObjects	500
NumUpdatesPerPeriod	1000
NumSyncsPerPeriod	250
Theta (θ)	0.0 - 1.6
UpdateStdDev(δ)	1.0

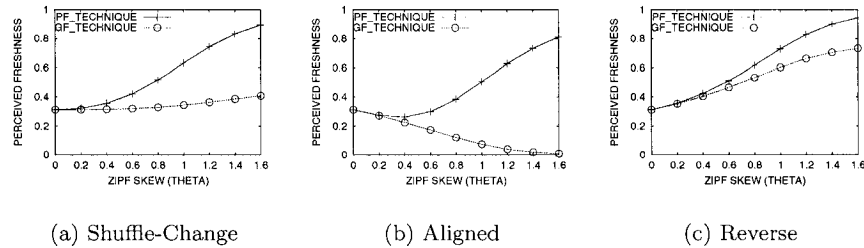


Figure 2.3: The Ideal Case for Perceived Freshness and General Freshness

The problem is classified as a convex optimization problem with a non-linear objective function². It is known that convex optimization problems have polynomial complexity. However, it is still not scalable for a huge number of variables. Even for a convex optimization problem with a linear objective function, the worst case complexity is $O(n^{3.5})$ where n is the number of variables [Kar84]. For non-linear objective functions the complexity is worse. Empirically, we have observed that running our non-linear programming package is manageable for problems with fewer than a thousand items; however, as we increase the size of the problem to hundreds of thousands of items, the package runs for days without terminating.

Managing mirrors with millions of elements is common in many real-world applications such as web search engines or data warehouses. For large real-world problems for which the contents of the mirror or the user interests might change, we would need to periodically solve the Core Problem to ensure that the freshening schedule still produces good results. The

²We can verify that $p_i \bar{F}(\lambda_i, f_i)$ is convex over f_i by computing second order derivatives of the analytic form for the Fixed Order policy.

time needed to solve for the optimal result will be intolerable. Previous work [CGM00b] does not consider this. In this section, we will explain several heuristic algorithms that reduce the problem size. We will also show that, for the most part, the accuracy of these algorithms compares favorably to the optimal solution.

The heuristic approaches we present in this section consist of two steps. The first step divides elements in a database into G partitions such that the number of elements in each of G partitions is T_1, T_2, \dots, T_G and $\sum_{i=1}^G T_i = N$.

In the second step, we pick a representative element from each of the partitions and solve the optimization problem for each of these G elements. This solution allocates bandwidth to each partition. This bandwidth is divided evenly among the elements of that partition.

The quality of the solution will depend on how the partitions are constructed and how the original optimization problem is transformed to a smaller more manageable problem. In the following, we describe our set of alternatives for partitioning and for constructing the smaller optimization problem.

2.3.1 Partitioning Alternatives

We define four partitioning techniques. Each of our techniques creates partitions in the same way. All N elements are sorted. Then N/G successive elements are assigned to a partition. When N is divisible by G , the number of elements in each partition will be equal. If N is not divisible by G , some of the partitions will have fewer elements. The effect on the performance from this difference will be negligible for very large N and relatively small G , which will be the usual case.

Clearly, p and λ will have an effect on the result, thus, we consider each of them as well as two ways of combining them as our sorting criteria.

2.3.1.0.1 p -Partitioning In this approach, all N elements in the database are sorted by access frequency (p) and therefore, elements in each partition will have similar access probabilities.

2.3.1.0.2 λ -Partitioning This approach is similar to P-Partitioning except that the elements in the database are sorted by change frequency (λ). This technique is included for completeness.

2.3.1.0.3 $\frac{p}{\lambda}$ -Partitioning This partitioning alternative orders elements by (p/λ) . The intuition behind this approach comes from the fundamental results found in [CGM00b] and in section 2.2. As change rate increases, synchronization frequency allocation should decrease and second, as access probability increases, synchronization frequency allocation should increase. Quantity $\frac{p}{\lambda}$ reflects these properties.

2.3.1.0.4 PF -Partitioning Though the partitioning techniques described above can identify similarities between elements based on their characteristics (p and λ) they do not reflect the relative importance of access probability and change frequency on perceived freshness. The PF -Partitioning approach orders elements by their perceived freshness (PF) given a fixed synchronization frequency³. The exact synchronization frequency used in our calculations is not important. We use a synchronization frequency of 1.0.

2.3.2 Optimization

After the partitioning phase, we need to solve an optimization problem to determine the best synchronization frequencies. We first treat all elements in a partition as identical. For identical elements, synchronization frequencies will be the same. The optimization problem on groups of identical elements is transformed to an equivalent problem in which the same objective function (but over fewer elements) and bandwidth constraint operate on a data set that includes one representative from each partition. This yields a bandwidth allocation for each partition which, then, must be allocated proportionately to each element in that partition. In this approach, optimization can be done in one step.

A second approach would transform the original problem into a number of smaller

³ When we use the Fixed-order synchronization policy, $PF(e_i) = p_i \times \frac{1-e^{-\lambda_i/c}}{\lambda_i/c}$ where c is the fixed synchronization frequency.

problems, in which only a small number of elements participate. Then G optimization problems are solved, each with a small number of elements. This approach requires multiple steps. We considered this approach but it does not make sense for large problems because it is still very costly to run. Whereas solving each individual subproblem is feasible, the sheer number of subproblems is too large. For example, if it is tolerable (solvable within an hour) to solve the optimization problem over 1000 elements, you would have to solve 1000 such problems for a database with 1,000,000 elements. Thus we did not report the results of these multi-stage algorithms.

To be clear, we formalize our optimization step. For G partitions, partition i ($i \leq G$) is regarded as having T_i identical elements. A representative must be chosen that reflects the similarity among these elements. For each partition C_i with T_i elements, a representative element is determined by averaging the access probabilities and the change frequencies of all elements in the partition. Thus, a representative element v_i for partition i has access probability(p_i) and change frequency(λ_i) as follows.

$$p_i = \sum_{e_j \in \text{partition } i} p_j / T_i, \quad \lambda_i = \sum_{e_j \in \text{partition } i} \lambda_j / T_i$$

The original optimization problem is converted to a smaller problem with G representative elements as shown below. Note that the objective function and constraint for the original problem have been changed by scaling the terms by the number of elements in each partition.

- **Transformed Problem** Given λ_i 's and p_i 's ($i = 1, 2, \dots, G$) for each representative element, find the values of f_i 's ($i = 1, 2, \dots, G$) which maximize

$$\overline{PF}(S) = \sum_{i=1}^G p_i \overline{F}(f_i, \lambda_i) T_i$$

when f_i 's satisfy the constraint

$$\sum_{i=1}^G T_i f_i = \text{TotalBandwidth} \quad \text{and} \quad f_i \geq 0$$

Solving for f_i gives an amount of bandwidth that is assigned to each of the elements of partition i .

2.4 Experiments

In this section we describe and analyze performance results for large numbers of elements for which we use our approximation techniques.

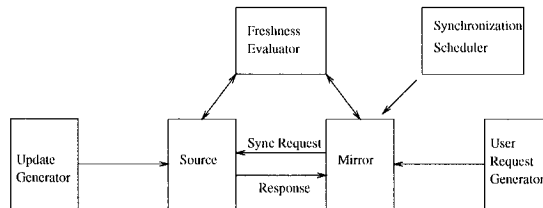


Figure 2.4: Simulation Model

A detailed simulation was built to demonstrate various synchronization algorithms. Figure 2.4 shows the model for the simulation. The parameters for the simulator are described below.

- **NumObjects:** number of objects in the database. In the current simulation, all objects have the same size.
- **PeriodLength:** length of a sync period in simulation clock time
- **NumSyncsPerPeriod:** number of syncs per period (can be characterized as the sync bandwidth)
- **UpdateStdDev(δ):** standard deviation of update distribution (gamma)
- **Theta(θ):** skew of the access distribution (zipf)

The *Synchronization Scheduler* has knowledge of the master profile and object change frequencies. From these it creates the schedule of synchronization requests for the *Mirror*. We use IMSL [IMS98] libraries to solve the non-linear objective function.

The *Freshness Evaluator* operates in two modes. It can analytically calculate freshness metrics based on simulated access and update workloads, or it can track system activity by monitoring updates and user requests and computing the metrics. The results shown in the next section have been verified using both modes.

2.4.1 Heuristic Case - Comparing the Techniques

In this section we show the results from our simulator for the approximation techniques described in section 2.3. We first compare the techniques to each other and to the optimal case for relatively small problem sizes. Next we demonstrate the differences between the techniques for much larger cases for which solving optimally is infeasible. We also show empirically that our solutions scale.

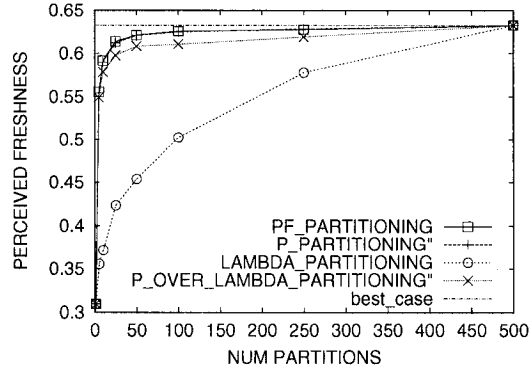
We compare techniques based on the quality of the result and the scalability of the technique.

2.4.1.1 Quality of the Result

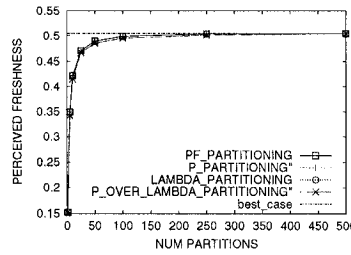
We use the set of parameters specified in Table 2.2 to compare the heuristic approaches. Recall from section 2.2.2.2 we study three alignment cases: *aligned*, *reverse*, and *shuffled-change*. Figure 2.5 shows the relative performance of our four partitioning techniques for these cases. With any of these techniques, as the number of partitions increases, the number of items in each partition decreases, and the approximate solution naturally approaches the ideal solution. The plots verify this and show that as the number of partitions increases the solution grows closer to the ideal case. The astute reader will notice that there is little difference between the techniques in Figures 2.5(b) and 2.5(c). This is because the partitioning techniques for the *aligned* and *reverse* alignments produce nearly identical results. In the *aligned* case, it is obvious that the partitions would be roughly the same for p -Partitioning and λ -Partitioning (both distributions have similar shape). Since access probability dominates the Perceived Freshness equation, PF -Partitioning and p -Partitioning generate very similar partitions for the case where the access is skewed. In both alignment cases, $\frac{p}{\lambda}$ -Partitioning will order based on whichever parameter is more highly skewed, therefore it follows that $\frac{p}{\lambda}$ -Partitioning will be nearly the same as the other three.

Hereafter we will only consider the *shuffled-change* alignment for comparing partitioning techniques. We do this because the independence of p and λ presents more of an average case rather than the extrema represented by *aligned* and *reverse* alignments and the average

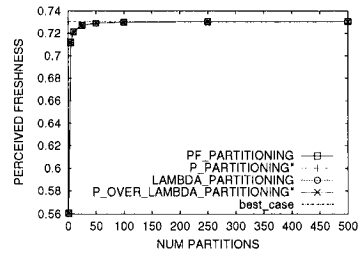
case demonstrates the differences between the partitioning techniques more plainly.



(a) Shuffle Change



(b) Aligned



(c) Reverse

Figure 2.5: Comparing the Techniques

Examination of the graph in Figure 2.5(a) reveals that some techniques approach the optimal solution more quickly (with a smaller number of partitions) than others under *shuffled-change* alignment. Note that p , PF , and $\frac{p}{\lambda}$ Partitioning approach the optimal solution more quickly than λ Partitioning. As we had seen from the optimal case in section 2.2.2.2, Perceived Freshness is dominated by access frequency. Figure 2.6 shows this more clearly. In the figure, the λ Partitioning technique can not achieve the same level of perceived freshness as θ is increased. Also notice that the plots have the same general shape as those for Perceived Freshness in Figure 2.3. The plot shows that as θ increases, perceived freshness increases for all of the techniques. The reason for this is that for high θ a small number of the

elements that have a high access probability receive the bulk of synchronization bandwidth. For low θ , elements with high change frequencies only have modest interest, therefore, λ dominates the impact on Perceived Freshness.

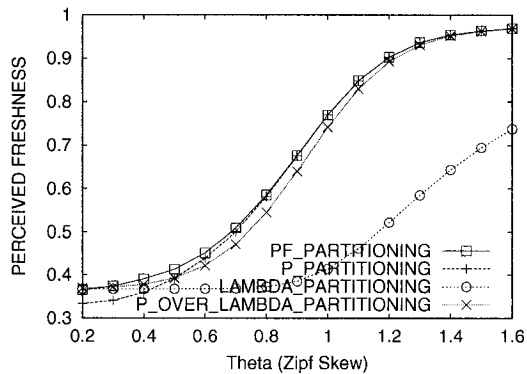


Figure 2.6: Sensitivity of Partitioning to Zipf Skew (θ) in Shuffle-Change Alignment

2.4.1.2 Scalability of the Techniques

All of the approximation techniques we have studied share the same basic principle: divide the search space into pieces and solve. The time required to solve the problem is based on the technique used to divide the search space and the resultant optimization problem(s). For large cases, the goal is to use the smallest number of partitions to achieve a good approximate answer.

Table 2.3: Setup for Partitioning Experiments

NumObjects	500000
NumUpdatesPerPeriod	1000000
NumSyncsPerPeriod	250000
Theta (θ)	1.0
UpdateStdDev(δ)	2.00

Table 2.3 shows the parameter setup used to demonstrate the performance of the two techniques. Figure 2.7 shows plots similar to those in figure 2.5, however, the size of the problem is much larger. Though we cannot verify the ideal solutions for the cases shown

here, the shape of the graph suggests the results from previous runs.

Notice that *PF*-PARTITIONING is still the clear winner and that the solutions using more than 100 partitions do not appreciably improve the answer.

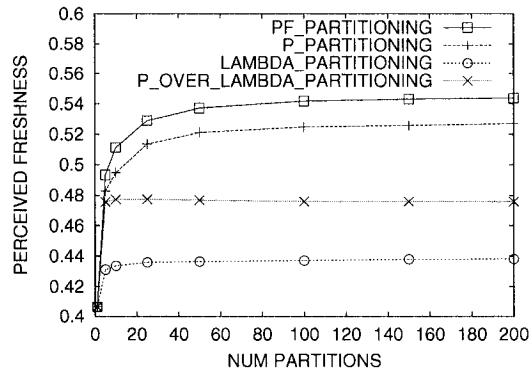


Figure 2.7: Big Case

2.4.1.3 An Additional Improvement

Empirically we have seen that the partitions can be improved by running several iterations of a k -Means clustering algorithm. In this technique, we start with partitions (clusters) as before, and we use k -Means clustering to clean up any clustering problems in our initial partitions. The distance metric we use for clustering is Euclidean Distance as described below:

- **Euclidean Distance** The Euclidean distance between two elements, $e_1(p_1, \lambda_1)$ and $e_2(p_2, \lambda_2)$, is defined as follows. Note that λ_i 's are normalized into $\check{\lambda}_i$'s so that $\sum_{i=1}^N \check{\lambda}_i = 1$ and $0 \leq \check{\lambda}_i \leq 1$.

$$E_{dist}(e_1, e_2) = \sqrt{(p_1 - p_2)^2 + (\check{\lambda}_1 - \check{\lambda}_2)^2} \quad (2.3)$$

During each iteration of k -Means clustering, each element is compared with the mean (average p and λ) of each partition (or cluster) and moved to the closest one. With each successive iteration, a better clustering is obtained. See Figure 2.8. In each iteration of the

clustering, there are $N \times k$ comparisons for N elements and k clusters. Thus, it is possible to improve the result in two different ways. We can increase the number of partitions, or we can run more iterations of k -Means clustering to improve the quality of the partitions. Figure 2.9 demonstrates this tradeoff. In the figure, the x-axis is time, and the thick line in the figure labeled "CLUSTER LINE" contains the points for the "0 iterations" case in Figure 2.8. That is, if you changed the x-axis of Figure 2.8 to time, the "0 iterations" plot would correspond to the "CLUSTER LINE" plot in figure 2.9. The other plots in Figure 2.9 show how Perceived Freshness improves using the k -Means clustering step. The first point on each of those plots (the point on "CLUSTER LINE") is 0 iterations, the next successive for each plot is 1 iteration followed by 3, 5, 7, 10, 15, and 25 iterations (when they fall within range on the graph). The very interesting phenomena is that with very few iterations, significant gains are seen in Perceived Freshness when clustering is used to improve the partitions created by PF -Partitioning. For this particular setup, a good solution is found earliest with 50 partitions and 10 iterations which finishes in 62 seconds.

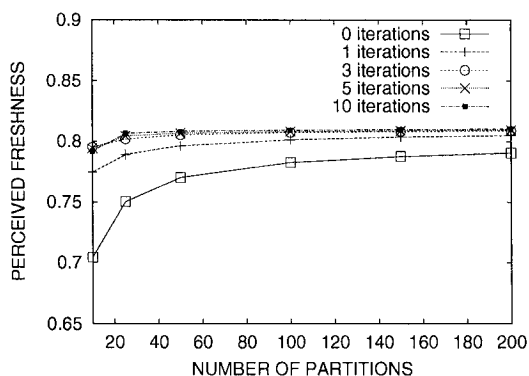


Figure 2.8: Improvement in Perceived Freshness after Clustering

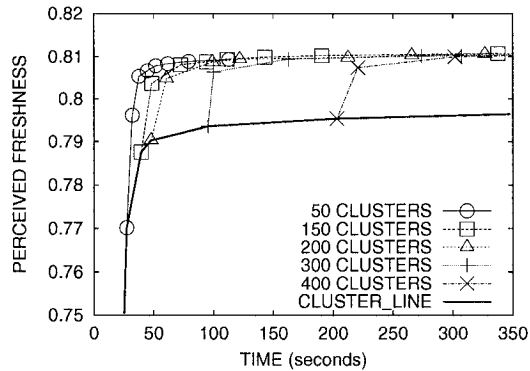


Figure 2.9: Improvement in Perceived Freshness after Clustering

2.5 Object Size Considerations

2.5.1 Problem Extension

We now relax the fixed object size assumption. Clearly, freshening a large object will take more bandwidth than a small object. For example, if an object whose size is 3 units of bandwidth is synchronized 2 times during a period, it will now consume 6 units of bandwidth. Recall from the earlier discussion of the Core Problem that for fixed size objects the linear constraint requires the sum of the refresh frequencies to be equal to the total bandwidth available (equation 2.2). For variable sized objects, this constraint becomes the sum of the products of sync frequencies (f_i) and object sizes (s_i) must be less than or equal to the total bandwidth (equation 2.4).

$$\sum_{i=1}^N s_i f_i \leq TotalBandwidth \quad \text{and} \quad s_i \geq 0, f_i \geq 0 \quad (2.4)$$

2.5.2 Freshening Approaches

Since the extended problem is different from the core problem only in its constraint specification, the optimal freshening problem can be solved with little difficulty.

For the freshening of a large number of variable-sized elements where heuristics are needed, we take the same steps of partitioning and optimization as described in section

2.3. In the partitioning step, we have to change the calculation of Perceived Freshness for *PF*-Partitioning. When we do not consider object size, we assume a constant amount of bandwidth for each element (see footnote 3). When considering object size, it is not fair to assume a constant bandwidth, because one sync of a big page costs more bandwidth than one sync of a small page. Therefore, we divide the constant bandwidth by the object size. We call this *PFS*-Partitioning.

The optimization alternatives are basically the same except that the constraint now becomes $\sum_{i=1}^G T_i \bar{s}_i f_i = TotalBandwidth$. The average of the size of elements in the same partition is represented as \bar{s}_i .

2.5.3 Results and Further Analysis

Other studies [KR01] have shown that object size on the web tends to follow a Pareto⁴ distribution. Figure 2.10 shows the optimal allocation of synchronization resources for two different page size distributions (uniform, where each object has the same size (1.0) and Pareto, where the object sizes follow a Pareto and have a mean of 1.0). For simplicity, access is uniform ($\theta = 0.0$) and change rate is aligned (object 1 has the highest change rate). Page size is also aligned (object 1 is largest). Figure 2.10(a) shows the number of synchronizations given to each item. Figure 2.10(b) shows the amount of synchronization bandwidth allocated to each item. Because the Pareto case has a large number of small objects, the total number of syncs is larger while the total amount of synchronization bandwidth is the same. It is important to note that when object size is variable, a smaller object can be given more synchronizations while consuming less bandwidth. For both cases, notice that all of the sync resources are given to the pages with the lowest change rates.

For large cases, there are important considerations to take into account concerning bandwidth allocation. There are two possible approaches for bandwidth allocation to the elements in the same partition:

⁴A *Pareto* distribution typically captures the highly variable size of objects, where $F(x) = (k/x)^a, x \geq k$ a is the *shape* and k is the *scale*. The distribution has a mean of $ka/(a - 1)$.

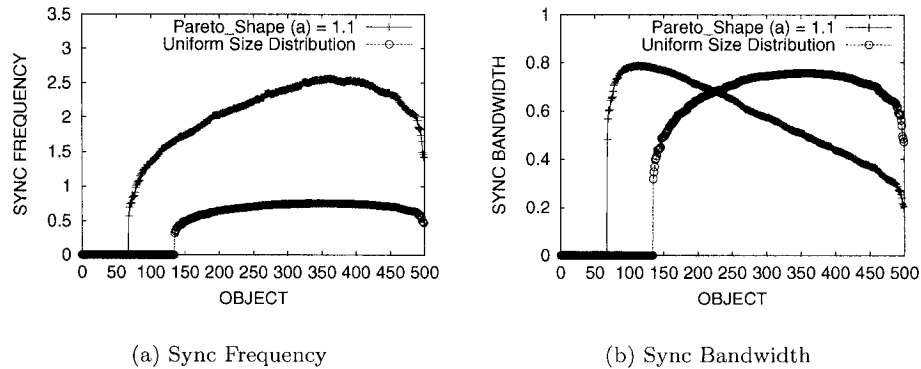


Figure 2.10: Optimal Sync Resource Distribution

1. **Fixed Refresh Frequency Allocation(FFA)** : All the elements get the same frequency of refresh. This technique was appropriate when all elements were of the same size. However, now this leads to disproportional bandwidth allocation. To address this we introduce the following technique.
2. **Fixed Bandwidth Allocation(FBA)** : The same amount of bandwidth is allocated for every element in the same partition. They will have different freshening frequencies which are determined by (assigned bandwidth⁵) / s_i . As a result, smaller objects will get higher number of refreshes than larger objects although they are in the same partition.

Figure 2.11 shows the relative performance of the allocation approaches described for *PFS*-Partitioning. For the experiment, the alignments of change rate and object size are reversed, and access is shuffled. (object 1 has a high change rate and a low size). This scenario seems to make sense for the real world. For instance, on the web, large objects like images and movies rarely change, whereas small objects like stock quotes and weather reports change quite often.

Notice that Fixed Bandwidth Allocation (FBA) approaches a better solution earlier (with fewer partitions) than with the Fixed Frequency Allocation (FFA). We have seen

⁵Note that the bandwidth for a representative element for a partition i from the optimization step will be $\bar{s}_i \times f_i$.

through empirical study of these allocation policies for small and large setups that FBA always outperforms FFA.

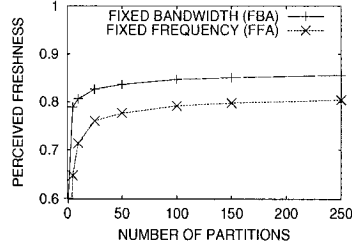


Figure 2.11: Sync Allocation

We have also applied the clustering improvement described in section 2.4.1.2 to the object size case and have found that it again improves the solutions within a small number of iterations ⁶. We also implemented *SIZE*-Partitioning which orders elements by size. Like p and λ -Partitioning, ordering by size only does not capture the relationship between elements so as to improve Perceived Freshness as much as *PFS*-Partitioning.

To summarize, in the experiments of (fig. 2.10(a)), the synchronization frequencies that are produced when we ignore object size (uniform distribution) yield a perceived freshness of 0.312. Whereas the synchronization frequencies that are produced when we take the size distribution into account (Pareto) yield a perceived freshness of 0.586. As shown in fig. 2.10(b) when we ignore object size, too much bandwidth is given to large objects which robs bandwidth from smaller objects, thereby, making suboptimal use of bandwidth. Furthermore, in our heuristic algorithms, the bandwidth allocation policy is effected by the variation in object size. Objects should be given a fixed bandwidth allotment.

⁶Euclidean Distance between two elements is $E_{distSize}(e_1, e_2) = \sqrt{(p_1 - p_2)^2 + (\lambda_1 - \lambda_2)^2 + (s_1 - s_2)^2}$. The s_i 's are normalized so that $\sum_{i=1}^N s_i = 1$.

2.6 Time average of perceived freshness

The perceived freshness of a mirror S is the sum of the freshness of its local copies observed by every access on a local copy in the mirror. The perceived freshness of S averaged over time should be the sum of the perceived freshness of its local copies averaged over time. The proof of the following theorem demonstrates this.

We assume that an *access set*, $A = a_1, \dots, a_M$, represents all the accesses over some period for mirror S with N local copies, e_1, \dots, e_N . When we denote the number of accesses for a local copy e_i as M_i , then $M = \sum_{i=1}^N M_i$ holds. Note that the access probability for a local copy e_i becomes $p_i = \frac{M_i}{M}$ and $\sum_{i=1}^N p_i = 1$. When a local copy, e_k , is not included in A , its M_k and p_k are 0.

Theorem 1

$$\overline{PF}(A) = \overline{PF}(S) = \sum_{i=1}^N p_i \overline{F}(e_i)$$

Proof 1

$$\begin{aligned} \overline{PF}(A) &= \lim_{t \rightarrow \infty} \frac{1}{t} \int_0^t PF(A; t) dt && \text{(from Definition 4)} \\ &= \lim_{t \rightarrow \infty} \frac{1}{t} \int_0^t \left(\frac{1}{M} \sum_{i=1}^M PF(a_i; t) \right) dt && \text{(from Definition 3)} \\ &= \frac{1}{M} \sum_{j=1}^M \lim_{t \rightarrow \infty} \frac{1}{t} \int_0^t PF(a_j; t) dt \\ &= \frac{1}{M} \sum_{i=1}^M \overline{PF}(a_i) \\ &= \frac{1}{M} \sum_{j=1}^N (\overline{PF}(e_j) \times M_j), \text{ (for } a_i = e_j) \end{aligned}$$

We can reformulate the above equation in the context of a database S as follows.

$$\begin{aligned} \overline{PF}(S) &= \sum_{j=1}^N (\overline{PF}(e_j) \times \frac{M_j}{M}) \\ &= \sum_{j=1}^N (\overline{F}(e_j) \times p_j) \end{aligned}$$

2.7 Solution for the Core Problem

Here we address the solution of the Core Problem as expressed by equations 2.1 and 2.2.

The perceived freshness of database S

$$\overline{PF}(S) = \sum_{i=1}^N p_i \overline{F}(e_i) = \sum_{i=1}^N p_i \overline{F}(f_i, \lambda_i)$$

takes its maximum, when all f_i 's satisfy the equations

$$\frac{\partial p_i \overline{F}(f_i, \lambda_i)}{\partial f_i} = \mu \quad \text{and} \quad \sum_{i=1}^N f_i = TotalBandwidth$$

The solution consists of $(N + 1)$ equations (N equations of $\partial p_i \overline{F}(f_i, \lambda_i) / \partial f_i = \mu$ and one equation of $\sum_{i=1}^N f_i = TotalBandwidth$) with $(N+1)$ unknown variables $(f_1, f_2, \dots, f_N, \mu)$ by introducing another variable μ , which is a typical way to apply method of Lagrange multipliers. We can solve these $(N + 1)$ equations for the f_i 's, since we know the closed form of $p_i \overline{F}(f_i, \lambda_i)$. From the solution, we can see that all optimal f_i 's satisfy $\partial p_i \overline{F}(f_i, \lambda_i) / \partial f_i = \mu$.

To compute the optimal f_i for each λ_i and p_i , we need to solve the equation

$$\partial p_i \overline{F}(f_i, \lambda_i) / \partial f_i = \mu \tag{2.5}$$

for f_i . When the Fixed-order policy is applied, $\overline{F}(f_i, \lambda_i)$ is derived as $\frac{1 - e^{-\lambda_i/f_i}}{\lambda_i/f_i}$. Unfortunately, Equation 2.5 is a non-linear equation which does not have a closed-form solution. Moreover, the equation has a parameter μ , whose value depends on the distribution of λ_i 's and p_i 's. Therefore, we may get totally different solution f for different distributions of λ_i 's and p_i 's. Cho et al.[CGM00b] proved that the solutions of Equation 2.5 are essentially the same, independently of μ when p_i 's are equal.

Equation 2.5 can be rewritten as

$$\partial \overline{F}(f_i, \lambda_i) / \partial f_i = \frac{\mu}{p_i} \tag{2.6}$$

Equation 2.6⁷ implies that a solution for an element $e_i(p_i, \lambda_i)$ which satisfy $\partial p_i \overline{F}(f_i, \lambda_i) / \partial f_i =$

⁷For the Fixed-order policy, this becomes $\frac{1}{f} e^{-\lambda_i/f_i} + \frac{1}{\lambda_i} (1 - e^{-\lambda_i/f_i}) = \frac{\mu}{p_i}$ after applying partial derivation.

μ is on the graph of $\partial\bar{F}(f_i, \lambda_i)/\partial f_i = \frac{\mu}{p_i}$. When such $p_i = \mu$, the solution will be on the graph of $\partial\bar{F}(f_i, \lambda_i)/\partial f_i = 1$.

2.8 Summary

This chapter presented a new criterion called *perceived freshness* for optimizing the freshness of a mirror given limited bandwidth. Unlike previous studies, we take into account the access frequencies for each copy.

We analyzed the optimal solution to the problem of maximizing perceived freshness, but observed that the solution for large problems takes far too much time. For cases in which we need to revisit the refresh schedule often (e.g., frequent updates to the mirror), the high cost of the optimal solution is unacceptable. Thus, several heuristic partitioning techniques that produce approximate solutions at drastically reduced costs were studied. The *PF-Partitioning* technique is shown to perform best and is shown to track the ideal solution for modest problem sizes. We further show that for very large problems, these techniques can produce good results in reasonable solution times.

The most surprising result was that reclustering partitions before running the optimization algorithm had dramatic effects on the results at a small cost in running time. Thus, using a small number of simple partitions as a starting point for reclustering was shown to be the best approach.

Next we analyze the impact of object size on the problem and the approaches. It was shown that there is an important distinction between synchronization frequency and synchronization bandwidth. We also present how to proportionally allocate synchronization bandwidth across different sized objects.

These results are relevant to many Internet-scale applications. They depend on the ability to gather information on user access-patterns, but this is feasible through direct feedback from users or from a simple learning algorithm that monitors the system request log.

Chapter 3

Resource Scheduling for the Management of Data Streams

3.1 Introduction

Many stream-based applications have sophisticated data processing requirements and real-time performance expectations that need to be met under high-volume, time-varying data streams. In order to address these challenges, novel operator scheduling approaches are introduced that specify (1) which operators to schedule (2) in which order to schedule the operators, and (3) how many tuples to process at each execution step. We study our approaches in the context of the Aurora data stream manager.

We show that a fine-grained scheduling approach in combination with various scheduling techniques (such as batching of operators and tuples) can significantly improve system efficiency by reducing various system overheads. Prototype-based experimental results are shown that characterize the efficiency and effectiveness of our approaches under various stream workloads and processing scenarios.

3.2 Aurora Overview

3.2.1 Aurora System Model

Aurora data is assumed to come from a variety of data sources such as computer programs that generate values (at regular or irregular intervals) or hardware sensors. We will use the term data source for either case. In addition, a data stream is the term we will use for the collection of data values that are presented by a data source. Each data source is assumed to have a unique source identifier and Aurora timestamps every incoming tuple to monitor the prevailing QoS.

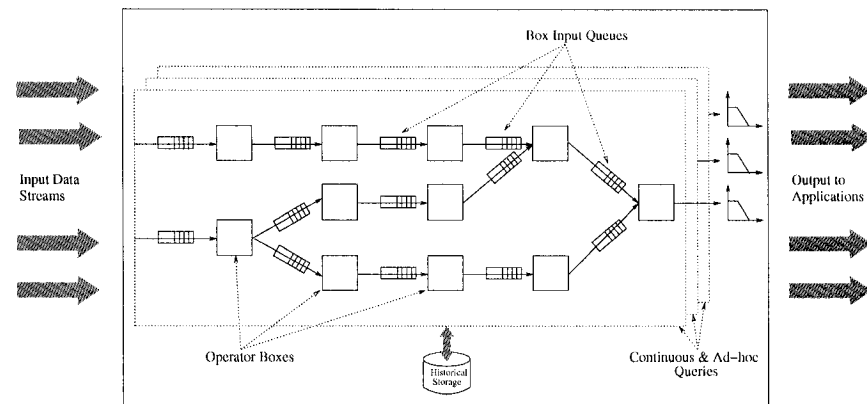


Figure 3.1: Aurora System Model

The basic job of Aurora is to process incoming data streams in the way defined by an application administrator. Figure 3.1 illustrates Aurora's high-level system model. Aurora is fundamentally a data-flow system and uses the popular boxes and arrows paradigm found in most process flow and workflow systems. Hence, tuples flow through a loop-free, directed graph of processing operations (a.k.a. boxes). Ultimately, output streams are presented to applications, which must be programmed to deal with output tuples that are generated asynchronously. Aurora can also maintain historical storage, primarily in order to support ad-hoc queries.

Tuples generated by data sources arrive at the input and are queued for processing.

Boxes with waiting tuples are selected and execute on one or more of their input tuples. The output tuples of a box are queued at the input of the next box in sequence. In this way, tuples make their way from the inputs to the outputs. Each output is associated with one or more Quality of Service (QoS) specifications, which define the utility of stale or imprecise results to the corresponding application.

The primary performance-related QoS is based on the notion of the latency (i.e., delay) of output tuples. Output tuples should be produced in a timely fashion, otherwise, QoS will degrade as latencies get longer. We will only deal with latency-based QoS graphs; for a discussion of other types of QoS graphs and how they are utilized, please refer to [ACC⁺03, CCC⁺02]. Aurora assumes that all QoS graphs are normalized, and are thus quantitatively comparable. Aurora further assumes that the QoS requirements are feasible; i.e., under normal operation (i.e., no peak overload), Aurora will be able to deliver maximum possible QoS for each individual output.

Aurora contains built-in support for a set of primitive operations for expressing its stream processing requirements. Some operators manipulate the items in the stream, others transform individual items in the stream to other items, while other operators, such as the aggregates (e.g., moving average), apply a function across a window of values in a stream. A description of the operators can be found in [ACC⁺03, CCC⁺02].

3.2.2 Architecture

Figure 3.2 illustrates the architecture of the basic Aurora run-time engine. The *Storage Manager* is responsible for maintaining the box queues and managing memory buffers used to store the queues. Boxes are selected for execution by the *Scheduler* which ascertains how many tuples to process from corresponding input queues, and pushes a pointer to box and queue descriptions (together with pointers to box state) onto an *execution queue*. A pool of *Worker Threads* individually pop *execution queue* elements and execute appropriate operations. Output tuples are forwarded to downstream queues. The scheduler then ascertains the next processing step and the cycle repeats. A *Catalog* contains information

regarding the network topology, inputs, outputs, and QoS information. Relevant statistics (e.g., selectivity, average box processing costs) are gathered and system performance (e.g. QoS) is continually monitored by a *Run-time Statistics Monitor*.

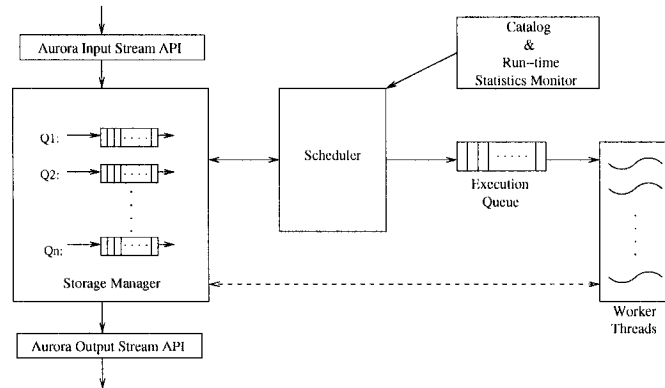


Figure 3.2: Aurora Run-Time

Scheduling in Aurora is a complex problem due to the need to simultaneously address several issues including large system scale, real-time performance requirements, and dependencies between box executions. Furthermore, tuple processing in Aurora spans many scheduling and execution steps (i.e., an input tuple typically needs to go through many boxes before potentially contributing to an output stream) and may involve multiple accesses to secondary storage. Basing scheduling decisions solely on QoS requirements, thereby failing to address end-to-end tuple processing costs, might lead to drastic performance degradation especially under resource constraints. To this end, Aurora not only aims to maximize overall QoS but also makes an explicit attempt to reduce overall tuple execution costs.

3.2.3 Basic Execution Model

The traditional model for structuring database servers is thread-based execution, which is supported widely by traditional programming languages and environments. The basic approach is to assign a thread to each query or operator. The operating system (OS) is responsible for providing a virtual machine for each thread and overlapping computation

and I/O by switching among the threads. The primary advantage of this model is that it is very easy to program, as the OS does most of the job. On the other hand, especially when the number of threads is large, the thread-based execution model incurs significant overhead due to cache misses, lock contention, and switching. More importantly for our purposes, the OS handles the scheduling and does not allow the overlaying software to have fine-grained control over resource management.

Instead, Aurora uses a state-based execution model. In this model, there is a single scheduler thread that tracks system state and maintains the execution queue. The execution queue is shared among a small number of worker threads responsible for executing the queue entries (as we discuss below, each entry is a sequence of boxes). This state-based model avoids the mentioned limitations of the thread-based model, enabling fine-grained allocation of resources according to application-specific targets (such as QoS). Furthermore, this model also enables effective batching of operators and tuples, which we show has drastic effects on the performance of the system as it cuts down the scheduling and box execution overheads.

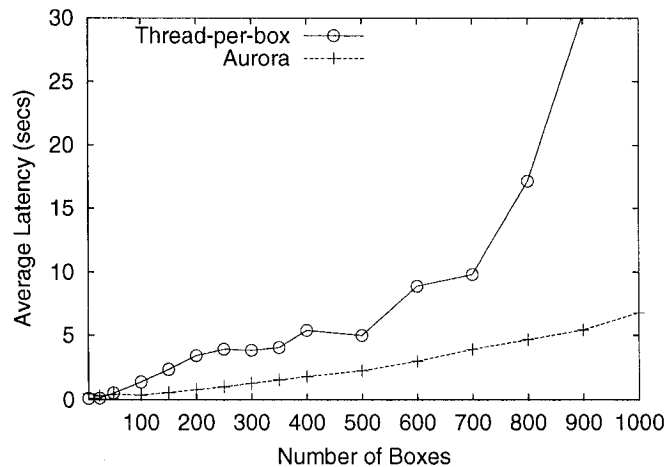


Figure 3.3: High-level comparison of stream execution models

In order to illustrate the basic performance benefits of Aurora’s state-based model over the traditional thread-based model (where each operator is assigned a single thread), we

ran a simple processing network consisting of multiple queries, each consisting of a chain of five filter operators (see Section 3.4.1 for a description of our experimental testbed). Figure 3.3 shows the tuple latencies observed as a function of the total number of operators. As we increase the system workload in terms of number of operators, the performance degrades in both cases, however much less in the Aurora case. In fact, performance degrades almost linearly in Aurora and exponentially in the thread-per-box case, a result that clearly supports the aforementioned scalability arguments.

An important challenge with the state-based model is that of designing an intelligent but low-overhead scheduler. In this model, the scheduler becomes solely responsible for keeping track of system context and deciding when and for how long to execute each operator. In order to meet application-specific QoS requirements, the scheduler should carefully multiplex the processing of multiple continuous queries. At the same time, the scheduler should try to minimize the system overheads, time not spent doing "useful work" (i.e., processing), with no or acceptable degradation in its effectiveness.

3.3 Two-Level Scheduling

Aurora uses a two-level scheduling approach to address the execution of multiple simultaneous queries. The first-level decision involves determining which continuous query to process. This is followed by a second-level decision that then decides how exactly the selected query should be processed. The former decision entails dynamically assigning priorities to operators at run-time, typically according to QoS specifications, whereas the latter decision entails choosing the order in which the operators involved in the query will be executed. The outcome of these decisions is a sequence of operators, referred to as a scheduling plan, to be executed one after another. Scheduling plans are inserted into an execution queue to be picked up and executed by a worker thread.

In order to reduce the scheduling and operator overheads, Aurora heavily relies on batching (i.e., grouping) during scheduling. We developed and implemented algorithms that batch both operators and tuples. In both cases, we observed significant performance

gains over the non-batching cases. We now describe in detail our batching approaches for constructing scheduling plans.

3.3.1 Operator Batching - Superbox Processing

A superbox is a sequence of boxes that is scheduled and executed as an atomic group. Superboxes are useful for decreasing the overall execution costs and improving scalability as (1) they significantly reduce scheduling overheads by scheduling multiple boxes as a single unit, thereby decreasing the number of scheduling steps; (2) they eliminate the need to access the storage manager for each individual box execution by having the storage manager allocate memory and load all the required tuples at once .

Conceptually, a superbox can be an arbitrary connected subset of the Aurora network. However, we do constrain the form of superboxes such that each is always a tree of boxes rooted at an output box (i.e., a box whose output tuples are forwarded to an external application). The reasons that underlie this constraint are twofold. First, only the tuples that are produced by an output box provide any utility value to the system. Second, even though allowing arbitrary superboxes provide the highest flexibility and increase opportunities for optimization, this will also make the search space for superbox selection intractable for large Aurora networks.

A note about query topology: Though we speak of queries as trees, Aurora queries can actually take the form of a graph like that shown in Figure 3.4. The figure shows two applications with overlapping queries (they have operator boxes in common). Also, boxes b_3 , b_4 , and b_5 form a subgraph which could not be contained in a tree. For our purposes, a *query tree* is defined as a digraph where each node (operator) has an out-degree of one. In this dissertation, we assume that all queries are trees. However, in Chapter 6 we discuss future research directions which consider overlapping queries (common subexpressions) and methods for composing superboxes from query graphs.

The following subsections discuss the two key issues to deal with when scheduling superboxes, namely superbox selection and superbox traversal.

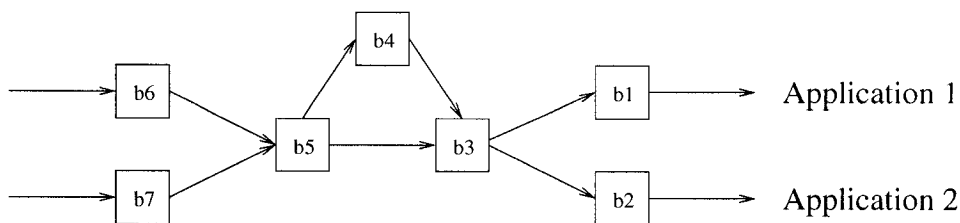


Figure 3.4: Sample Query Graph

3.3.1.1 Superbox Selection

The first-level scheduling issue involves determining which superboxes to schedule. Fundamentally, there are two different approaches to superbox selection: static and dynamic. Static approaches identify potential superboxes statically before run time, whereas the dynamic approaches identify useful superboxes at run time.

In Aurora, we implemented a static superbox selection approach, called application-at-a-time (AAAT). AAAT statically defines one superbox for each query tree. As a result, the number of superboxes is always equal to the number of continuous queries (or applications) in the Aurora network. Figure 3.5 illustrates a simple query tree that consists of six boxes (the tree is rooted at box b_1). Once the superboxes are identified, they can be scheduled using various scheduling policies (e.g., round-robin).

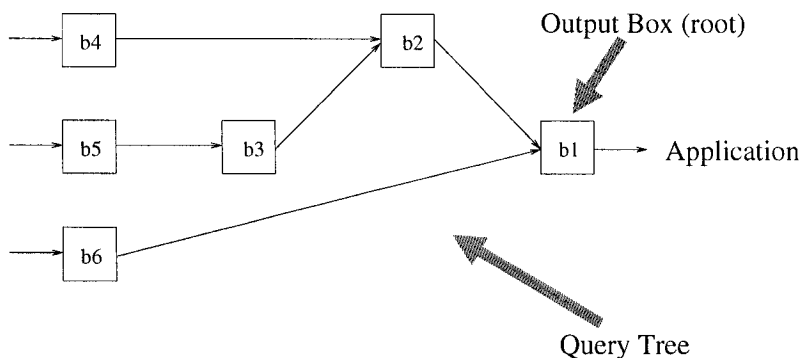


Figure 3.5: Sample Query Tree

3.3.1.2 Superbox Traversal

Once it is determined which superboxes need to be executed, a second-level decision process specifies the ordering of component boxes. This is accomplished by traversing the superbox. The goal of superbox traversal is to process all the tuples that are queued within the superbox (i.e., those tuples that reside on the input queues of all boxes that constitute the superbox).

We investigate three traversal algorithms that primarily differ in the performance-related metric they strive to optimize: *throughput*, *latency*, and *memory consumption*.

Min-Cost (MC): The first traversal technique attempts to optimize per-output-tuple processing costs (or *average throughput*). The goal here is to do the least amount of work per unit time. Given a query and a set of input tuples, the amount of box processing is constant, and the opportunity to reduce the cost of executing the query comes from overhead reduction. We can do this by minimizing the number of box calls per output tuple. One call per box in the query is minimum. Therefore, a traversal which processes all tuples for a query with only one box-call per query operator would achieve this minimum overhead. This can be accomplished by traversing the superbox in *post order*, where a box is scheduled for execution only after all the boxes in its sub-tree are scheduled. Notice that a superbox execution based on an MC traversal consumes all tuples (available at the start of execution) while executing each box only once.

Consider the query tree shown in Figure 3.5 and assume for illustration purposes that a superbox that covers the entire tree is defined. Also, assume that each box has a processing cost per tuple of p , a box call overhead of o , and a selectivity equal to one. Furthermore, assume that each box has exactly one non-empty input queue that contains a single tuple. An MC traversal of the superbox executes each box only once:

$$b_4 \rightarrow b_5 \rightarrow b_3 \rightarrow b_2 \rightarrow b_6 \rightarrow b_1$$

There are 15 tuples which have to be processed and this traversal requires 6 box calls to process them through to the output. Table 3.1 steps through the processing of tuples.

Note that the processing of tuples through box b_1 has been expanded to show individual output latencies. Note that the total execution cost of the superbox (i.e., the time it takes to produce all the output tuples) is $15p + 6o$ and that the minimum call overhead ($6o$) has been achieved. The average output tuple latency for this example is $12.5p + 6o$. We next consider an traversal which reduces average output tuple latency.

Box	Processing (p)	Call Overhead (o)	Output Tuple Latency
b_4	1	1	
b_5	1	1	
b_3	2	1	
b_2	4	1	
b_6	1	1	
b_1	1	1	$10p + 6o$
b_1	1	0	$11p + 6o$
b_1	1	0	$12p + 6o$
b_1	1	0	$13p + 6o$
b_1	1	0	$14p + 6o$
b_1	1	0	$15p + 6o$
	Total Processing: $15p$	Total Call Overhead: $6o$	Average Latency: $12.5p + 6o$

Table 3.1: Stepping Through Min-Cost Traversal

Min-Latency (ML): Average latency of the output tuples can be reduced by producing initial output tuples as fast as possible. In order to accomplish this, we define a cost metric for each box b , referred to as the output cost of b , $output_cost(b)$. This value is an estimate of the latency incurred in producing one output tuple using the tuples at b 's queue and processing them downstream all the way to the corresponding output.

This value can be computed using the following formulas:

$$o_sel(b) = \prod_{k \in D(b)} sel(k)$$

$$output_cost(b) = \sum_{k \in D(b)} cost(k)/o_sel(k)$$

where $D(b)$ is the set of boxes downstream from b and including b , and $sel(b)$ is the estimated selectivity of b . In Figure 3.5, $D(b_3)$ is $b_3 \rightarrow b_2 \rightarrow b_1$, and $D(b_1)$ is b_1 . The output selectivity

of a box b , $o_sel(b)$, estimates how many tuples should be processed from b 's queue to produce one tuple at the output.

We start with an empty traversal sequence. To come up with the traversal order, the boxes are first sorted in increasing order of their *output_cost*. Then, for each box, b , in this sorted list, $D(b)$ is appended to the traversal sequence. This will create a traversal which consumes all tuples in the query. For example, an ML traversal of the superbox of Figure 3.5 is:

$$b_1 \rightarrow b_2 \rightarrow b_1 \rightarrow b_6 \rightarrow b_1 \rightarrow b_4 \rightarrow b_2 \rightarrow b_1 \rightarrow b_3 \rightarrow b_2 \rightarrow b_1 \rightarrow b_5 \rightarrow b_3 \rightarrow b_2 \rightarrow b_1$$

Table 3.2 steps through the traversal. The ML traversal incurs nine extra box calls over an MC traversal (which only incurs six box calls). In this case, the total execution cost is $15p + 15o$, and the average latency is $7.17p + 7.17o$.

Box	Processing (p)	Call Overhead (o)	Output Tuple Latency
b_1	1	1	$1p + 1o$
b_2	1	1	
b_1	1	1	$3p + 3o$
b_6	1	1	
b_1	1	1	$5p + 5o$
b_4	1	1	
b_2	1	1	
b_1	1	1	$8p + 8o$
b_3	1	1	
b_2	1	1	
b_1	1	1	$11p + 11o$
b_5	1	1	
b_3	1	1	
b_2	1	1	
b_1	1	1	$15p + 15o$
	Total Processing: $15p$	Total Call Overhead: $15o$	Average Latency: $7.17p + 7.17o$

Table 3.2: Stepping Through Min-Latency Traversal

In general, MC achieves a lower total execution time than ML. This is an important improvement especially when the system is under CPU stress, as it effectively increases the throughput of the system. There are exceptions where ML may achieve the same execution

time as MC. This is true whenever ML and MC produce the same traversal. For example, consider this query which consists of a chain of boxes:

$$\longrightarrow b_4 \longrightarrow b_3 \longrightarrow b_2 \longrightarrow b_1 \longrightarrow$$

If only b_4 has input tuples, then both MC and ML will execute the same schedule (b_4, b_3, b_2, b_1) . Of course, this negates the overhead that an ML scheduler would incur determining that only b_4 has input tuples.

ML *may* achieve lower latency than MC depending on the ratio of box processing costs to box overheads. In the example traversals shown in Tables 3.1 and 3.2, ML yields lower latency if

$$12.5p + 6o > 7.17p + 7.17o$$

or, $\frac{p}{o} > 0.22$. That is, if processing a box is greater than 22% of the overhead of executing a box. Note that as the box processing cost becomes negligible (or overhead increases), MC produces tuples with lower average latencies than ML.

Min-Memory (MM): This traversal is used to maximize the consumption of data per unit time. In other words, we schedule boxes in an order that yields the maximum increase in available memory (per unit time).

$$mem_r = \frac{tsize(b) \times (1 - selectivity(b))}{cost(b)}$$

The above formula is the expected memory reduction rate for a box b ($tsize(b)$ is the size of a tuple that reside on b 's input queue). Once the expected memory reduction rates are computed for each box, the traversal order is computed. Boxes are placed in the traversal sequence in order of decreasing mem_rr . For each box b placed in the traversal sequence, the boxes in $D(b)$ are tested in sequence for higher mem_rr . As mem_rr increases for the boxes in $D(b)$, those boxes are also placed in the traversal. Pseudocode for this procedure follows:

```

// boxes_sorted_by_mem_rr is a vector of boxes sorted by mem_rr
// traversal[] is the traversal
// D(box) is a function which returns a vector of boxes
//         representing D(b), the downstream boxes from box, including box
// mem_rr(box) is a function which returns the mem_rr of box

index = 0;
for ( i = 0; i < boxes_sorted_by_mem_rr.size(); i++ )
{
    box = boxes_sorted_by_mem_rr[i];
    index = index + 1;
    traversal[index] = box;
    downstream = D(box);
    for ( j = 1; j < downstream.size(); j++ )
    {
        if ( mem_rr(downstream[j]) > mem_rr(box) )
        {
            box = downstream[j];
            index = index + 1;
            traversal[index] = box;
        }
    }
}

```

Let's now consider the MM traversal of the superbox in Figure 3.5, this time with the box selectivities and costs shown in Table 3.3. Assuming that all trains are of size one, the *mem_rr* for all the boxes is also shown in Table 3.3. Therefore, the MM traversal is:

$$b_3 \rightarrow b_2 \rightarrow b_1 \rightarrow b_6 \rightarrow b_1 \rightarrow b_2 \rightarrow b_5 \rightarrow b_3 \rightarrow b_2 \rightarrow b_1 \rightarrow b_4 \rightarrow b_2 \rightarrow b_1$$

Note that this traversal might be shorter at run time: for example, if b_5 consumes all of its input tuples and produces none on the output, the execution of b_3 after b_5 will clearly be unnecessary.

Box	Selectivity	Cost	<i>mem_rr</i>
b_1	0.9	2	0.05
b_2	0.4	2	0.3
b_3	0.5	1	0.5
b_4	1.0	2	0
b_5	0.4	3	0.2
b_6	0.6	1	0.4

Table 3.3: Costs, Selectivities, and *mem_rr* for Min-Memory Traversal Example

3.3.2 Tuple Batching - Train Processing

A tuple train (or simply a train) is a sequence of tuples executed as a batch within a single box call. Tuple train processing reduces overall tuple processing costs. This happens for several reasons: First, given a fixed number of tuples to process, train processing decreases the total number of box executions required to process those tuples, thereby cutting down low-level overheads such as scheduling overhead (including maintenance of the execution queue and memory management), reducing calls to the box code, and decreasing context switches. Second, as in the case of superbox scheduling, train processing also improves memory utilization when the system operates under memory stress (see Section 3.3.1). A third reason, which we have not yet explored, is that some operators may optimize their execution better with larger number of tuples available in their queues. For instance, a box can materialize intermediate results and reuse them in the case of windowed operations, or use merge-join instead of nested loops in the case of joins.

In general, a scheduled box consumes the entire train of tuples in its input queue when it executes. In a later section (section 3.4.4) we explore different train sizes and their impact on system performance.

3.4 Experimental Evaluation

3.4.1 Experimental Testbed

We use the Aurora prototype system to study our operator scheduling techniques. The reference run-time architecture is defined in Section 3.2.2.

The prototype is implemented on top of Debian GNU/Linux using C++. In the experiments, we used a dedicated Linux workstation with a 2.0 GHz Pentium IV processor and 512M of RAM. The machine was isolated from the network to avoid external interference.

Due to the fact that the domain of stream-based applications is still emerging and that there are no established benchmarks, we decided to artificially generate data streams and continuous queries to characterize the performance of our algorithms, as described below.

We generated an artificial Aurora network as a collection of continuous queries, each feeding output tuples to individual applications. We modeled a continuous query as a tree of boxes rooted at an output box (i.e., a box whose outputs are fed to one or more applications). We refer to such a query tree as an application tree. Each query is then specified by two parameters: depth and fan-out. Depth of a query specifies the number of levels in the application tree and fan-out specifies the average number of children for each box.

For ease of experimentation, we implemented a generic, universal box whose per-tuple processing cost and selectivity can be set. Using this box, we can model a variety of stateless stream-based operators such as filter, map, and union. For simplicity, we chose not to model stateful operators as their behavior is highly- dependent on the semantics they implement, which would introduce another dimension to our performance evaluation and restrict the generality of our conclusions. This would complicate the understanding of the results without making a substantial contribution to the understanding of the relative merits of the algorithms.

An Aurora network consists of a given number of query trees. All queries are then associated with latency- based QoS graphs, a piece-wise linear function specified by three

points: (1) maximum utility at time zero, (2) the latest latency value where this maximal utility can be achieved, and (3) the deadline latency point after which output tuples provide zero utility.

To meaningfully compare different queries with different shapes and costs, we use an abstract capacity parameter that specifies the overall load as an estimated fraction of the ideal capacity of the system. For example, a capacity value of .9 implies that 90% of all system cycles are required for processing the input tuples. Once the target capacity value is set, the corresponding input rates (uniformly distributed across all inputs) are determined using an open-loop computation. Because of various system overheads, the CPU will saturate typically much below a capacity of one.

The graphs presented here provide average figures of six independent runs, each processing 100K input tuples. Unless otherwise stated, the fan-out parameter is set to three; the depth is set to five; the selectivities of the boxes are set to one; and the per-tuple processing costs are selected from the range [0.0001 sec/tuple - 0.001sec/tuple].

3.4.2 Operator Batching - Superbox Scheduling

We investigate the benefits of superbox scheduling by comparing the performance of box-at-a-time (BAAT) with application-at-a-time (AAAT). For both, we used a round-robin (RR) scheduler which in the case of AAAT, cycles through applications, running each with an MC traversal. In the case of BAAT, it cycles through runnable¹ boxes in a pre-determined random order.

Figure 3.6 shows the average tuple latencies of these approaches as a function of the input rate (as defined relative to the capacity of the system) for five application trees. As the arrival rate increases, the queues eventually saturate and latency increases arbitrarily. The interesting feature of the curves in the figure is the location of the inflection point. RR-BAAT does particularly badly. In these cases, the scheduling overhead of the box-at-a-time approach is very evident. This overhead effectively steals processing capability from the

¹a *runnable* box is one which has at least one input tuple to process.

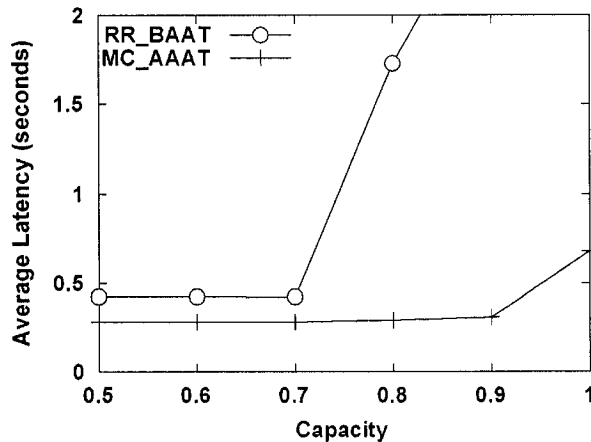


Figure 3.6: Box vs. Application Scheduling

normal network processing, causing saturation at much earlier points. On the other hand, the MC_AAAT algorithm performs quite well in the sense that it is resistant to high load. This technique experiences fewer scheduler calls and, thus, have more processing capacity and is able to hang on at input rates of over 90% of the theoretical capacity.

3.4.3 Superbox Traversal

We first investigate the performance characteristics of the Min-Cost (MC) and Min-Latency (ML) superbox traversal algorithms. In this experiment, we use a single application tree and a capacity of 0.5. Note that as box and application costs change, to maintain a fixed capacity, input rates must change.

Figure 3.7 shows the average output tuple latency as a function of per-tuple box processing cost. As expected, both approaches perform worse with increasing processing demands. For most of the cost value range shown, ML not surprisingly performs better than MC as it is designed to optimize for output latency. Interestingly, we also observe that MC performs better than ML for relatively small processing cost values. The reason is due to the relationship between the box processing cost and box call overhead, which is the operational cost of making a box call. The box call overhead is a measure of how much time is spent outside the box versus inside the box (processing tuples and doing real work).

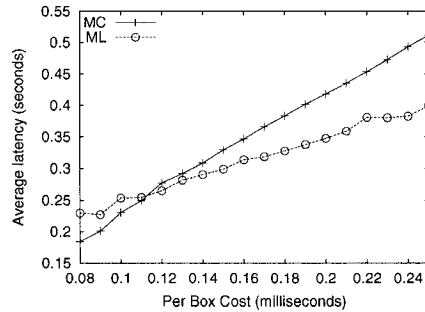


Figure 3.7: Box vs. Application Scheduling

As we decrease the box processing costs, box call overheads become non-negligible and, in fact, they start to dominate the overall costs incurred by the algorithms. As we explained in Section 3.3.1.2, an MC traversal always requires fewer box calls than ML does. We thus see a cross-over effect: for smaller box processing costs, box call overheads dominate overall costs and MC wins. For larger processing costs, ML wins as it optimizes the traversal for minimizing output latency.

MC incurs less overall box overhead as it minimizes the number of box calls. The difference increases as the applications become deeper and increase in the number of boxes. In fact, the overhead difference between the two traversals is proportional to the depth of the traversed tree.

These key results can be utilized for improving the scheduling and overall system performance. It is possible to statically examine an Aurora network, obtain box- processing costs, and then compare them to the (more or less fixed) box processing overheads. Based on the comparison and using the above results, we can then statically determine which traversal algorithm to use. A similar finer-grained approach can be taken dynamically. Using a simple cost model, it is straightforward to compute which traversal algorithm should do better for a particular superbox.

Figure 3.8 demonstrates the amount of memory used over the time of superbox run. The curves are normalized with respect to the MM values. ML is most inefficient in its use of memory with MC performing second. MC minimizes the amount of box overhead. As a

result MC discards more tuples per unit of time than ML.

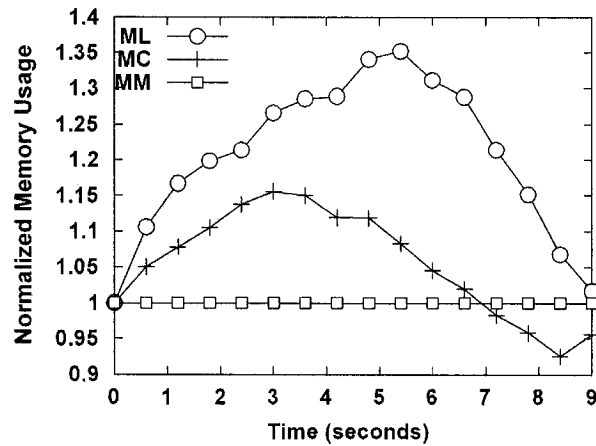


Figure 3.8: Memory requirements of traversal algorithms

MM loses its advantage towards the end since all three traversals are executed on a common query network. Even though each chooses a different execution sequence and incurs different overhead, all three traversals push the same tuples through the same tree of boxes. The crossover towards the end of the time period is a consequence of different traversals taking different amounts of times to finish. In general, MC has the smallest total execution time and as a result catches up with MM towards the end of the shown execution range.

3.4.4 Tuple Batching - Train Scheduling

Train scheduling is only relevant in cases in which multiple tuples are waiting at the inputs to boxes. This does not happen when the system is very lightly loaded. In order to see how train scheduling affects performance, we needed to create queues without saturating the system. We achieved this by creating a bursty (or clustered) workload that simply gathers tuples in our previously studied workloads and delivers them as a group. In other words, if our original workload delivered n tuples evenly spaced in a given time interval T , the bursty version of this delivers n tuples as a group and then delivers nothing more for the next T time units. Thus, the bursty workload is the same in terms of average number of tuples

delivered, but the spacing is different.

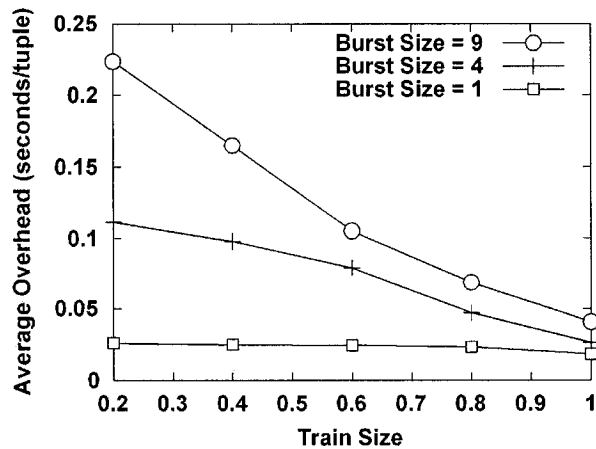


Figure 3.9: Train scheduling effects

The graph in Figure 3.9 shows how the train scheduling algorithm behaves for several bursty workloads. In this experiment, we have a single application tree with a fan-out of two and a depth of five. In order to isolate the effects of operator scheduling, we use round-robin BAAT for this experiment. The train size (x-axis) is given as a percentage of the queue size. As we move to the right, the trains bite off increasingly larger portions of the queues. With a burst size of one, all tuples are evenly spaced. This is equivalent to the normal workload. Notice that the curve for this workload is flat. If there are no bursts, train scheduling has no effect. For the other two curves, however, as the burst size increases, the effect gets more pronounced. With a train size of 0.2, the effect on the overhead (i.e., total execution time less processing time) of increasing the burst size is substantial. For a burst size of 4, we quadruple the average overhead. Now as we increase the train size, we markedly reduce the average overhead for the bursty cases. In fact, when the train size is equal to one (the entire queue), the average overhead approaches the overhead for the non-bursty case. Trains improve the situation because tuples do not wait at the inputs while other tuples are being pushed through the network. It is interesting to note that the bursty loads do not completely converge to the non-bursty case even when the train size is

one (i.e., the whole queue). This is because the tuples still need to be processed in order. Since the bursty workload generation delivers $n-1$ of the tuples early, their latency clock is ticking while the tuples in front of them are being processed. In the non-bursty case, the tuples arrive spaced out in time, and a fair amount of processing can be done on queued tuples before more tuples arrive.

3.4.5 Overhead Distribution

Figure 3.10 shows a comparison of the relative execution overheads and how they are distributed for TAAT (tuple-at-a-time), BAAT (tuple trains), and MC (superbox), for four application trees. Each bar is divided into three fundamental cost components: the worker thread overhead, the storage management overhead, and the scheduler overhead. The number at the top of each bar shows the actual time for processing 100K tuples in the system.

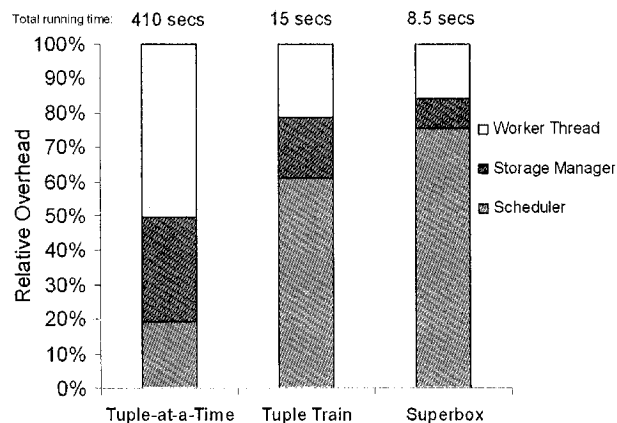


Figure 3.10: Distribution of execution overheads

Looking at the total running times, the first thing to notice is that TAAT is significantly worse than the other two methods, underscoring our conclusion that train and superbox scheduling are important techniques for minimizing various system overheads and improving the overall system throughput. Additionally, this graph shows clearly the benefits of superbox scheduling, which decreased the overall execution time of the system running tuple trains almost by 50%.

Finally, we note the interesting trend in the relative component costs for each approach. While the percentages of the worker thread and storage manager overheads decrease, as we go from the leftmost bar to the right, the percentage of the scheduler overhead increases and starts to dominate the rest. The reason is that, as batching is increased, more tuples are processed at each scheduling step. In other words, the number of scheduling steps to process a specific number of tuples decreases, but the number of box executions decreases more. Because the worker thread and storage management overheads are primarily associated with the number of box executions, their overheads decrease more relatively to that of the scheduler. Another contributing factor is that, again as we go from left to right, the scheduler algorithms become increasingly more intelligent and sophisticated, taking more time to generate the scheduling plans.

3.5 Summary

This chapter presented an experimental investigation of scheduling algorithms for stream data management systems. It demonstrated that the effect of system overheads (e.g., number of scheduler calls) can have a profound impact on real system performance. Experiments were run on the Aurora prototype since simulators do not reveal the intricacies of system implementation penalties.

The naive approach of using a thread per box was shown not to scale. It was further shown that train scheduling and superbox scheduling help a lot to reduce system overheads. Exactly how these overheads are affected in a running stream data manager was also discussed. In particular, these algorithms require tuning parameters like train size and superbox traversal methods.

The overriding message of this chapter is that to build a practical data stream management system, one must ensure that scheduler overhead be small relative to useful work. Some interesting results were provided in this direction by focusing on batching techniques.

Chapter 4

Scheduling for QoS in Aurora

4.1 Introduction

Many of the applications for which a Stream Processing Engine (SPE) is designed fall in the class of "monitoring" applications. Monitoring applications are applications that monitor continuous data streams. Many of these applications have a low tolerance for stale data and a SPE must employ intelligent resource management. As described in [CCC⁺02], many monitoring applications have latency-based QoS expectations. Here, we present new techniques for achieving the best level of QoS through scheduling in the Aurora Stream Processing Engine.

The Aurora architecture was described in Chapter 3. Queries in Aurora are built with operators (*boxes*) which process input tuples and pass resulting tuples to downstream boxes. In a running Aurora system, tuples arrive at the inputs to boxes and a scheduler runs periodically to determine which tuples to process and the order in which the tuples should be processed ¹. This *schedule* is passed to the *box processor* which then executes boxes.

Aurora applications specify latency-based QoS graphs which indicate the utility of a tuple relative to the amount of time it took to process the tuple. Aurora's scheduler determines the order in which to process tuples based on the ability to meet QoS expectations.

¹Tuples that arrive at the input of a box will always be processed by the box in the order they arrive, that is, in FIFO order

Therefore, the scheduler uses QoS expectations and the cost of processing tuples through to an output in its decision.

In this chapter, we explore scheduling techniques to achieve the best level of QoS. In particular, we focus on the creation of static and dynamic schedules. A scheduler which uses a *static* schedule always schedules boxes in the same order. A scheduler which uses a *dynamic* schedule determines at run-time the order in which boxes should be executed.

This chapter presents our static and dynamic scheduling techniques and provides experimental results.

4.2 QoS-Aware Scheduling

This section describes our static and dynamic approaches to QoS-aware scheduling. The goal here is to explain the details of both Fixed-Priority (static) and SlopeSlack (dynamic). A discussion of their relative performance is presented in section 4.3

4.2.1 Static Schedules

In Aurora, a static schedule consists of an ordering of boxes which remains fixed for a particular Aurora query network. The system determines *a priori* a schedule for the execution of boxes. Each time the scheduler is invoked, the runnable boxes² are sorted in that pre-determined order and run. We consider two types of static schedules:

- *Round-Robin*: When the scheduler runs, all runnable boxes are scheduled in a pre-determined random order.
- *Fixed-Priority*: Boxes are prioritized a priori based on their QoS constraints. Each time the scheduler runs, it schedules the top N priority runnable boxes.

The Round-Robin schedule is not created with any application specific QoS information, and, since we are concerned with measuring the QoS of output tuples, it should not perform

²As in the previous chapter, a *runnable* box is one which has at least one input tuple to process.

well. It is used as a baseline from which to compare our QoS-aware schedulers. Without the use of a QoS-aware scheduler, Aurora would rely on the operating system scheduler which would essentially schedule boxes in round-robin order.

The Fixed-Priority scheduler uses application QoS constraints to determine per box priority. A box's priority is based on the amount of *slack* left in the QoS graph after accounting for the latency an input tuple would incur due to downstream box costs. *Slack* is a measure of how close an output tuple's latency value is to a *critical point* on a QoS graph; i.e. a point where the QoS drops sharply. The Fixed-Priority schedule is an ordering on these slack-based box priorities. For example, consider the following applications:

$$\longrightarrow b_3 \longrightarrow b_2 \longrightarrow b_1 \longrightarrow \textit{Application}_1$$

$$\longrightarrow b_6 \longrightarrow b_5 \longrightarrow b_4 \longrightarrow \textit{Application}_2$$

For simplicity, consider that each box costs 1 time unit to process a single tuple. Also, since QoS graphs essentially specify soft deadlines for applications to meet, let's say that their QoS graphs specify the following deadlines after which output tuples produce reduced utility:

$$\textit{Application}_1 \textit{ deadline} : 4 \textit{ time units}$$

$$\textit{Application}_2 \textit{ deadline} : 8 \textit{ time units}$$

A tuple queued at the input to b_3 would incur 3 time units of latency due to required processing through boxes b_3 , b_2 , and b_1 . This tuple has a slack value of 1 since its application deadline is 4 time units. Table 4.1 gives the slack values for all boxes in the above example. Note that the slack values are also the priorities of the boxes. The Fixed-Priority schedule for this example is: $b_3, b_2, b_1, b_6, b_5, b_4$

The Fixed-Priority scheduler is adapted from real-time systems *deadline-monotonic* scheduling [LW82] where tasks are ordered by deadline (shortest deadline first).

Box	Slack (Priority)
b_1	3
b_2	2
b_3	1
b_4	7
b_5	6
b_6	5

Table 4.1: Box Priorities for Fixed-Priority Scheduling Example

4.2.2 Dynamic Schedules

A dynamic schedule is created at each scheduling event. When the scheduler runs, it prioritizes a box based on the *expected* latency of the tuples in its input queues. The basic approach is to keep track of the latency of tuples that reside at the queues and pick for processing the tuples whose execution will provide the highest expected increase in the aggregate QoS delivered to the applications. Taking this approach per tuple is not scalable. We therefore maintain latency information at the granularity of individual boxes and define the latency of a box as the averaged latencies of the tuples in its input queue(s).

Our priority assignment approach is to order the boxes in terms of their *utility* and *urgency*. We define the utility of a box b in terms of its *expected slope value*, $slope(b)$, and define its urgency in terms of its *expected slack time*, $slack(b)$.

We compute the utility of b as follows:

$$utility(b) = gradient(eol(b)) \quad (4.1)$$

This value is the gradient, or *slope*, of the QoS-latency curve for b 's corresponding output at the latency value $eol(b)$, where $eol(b)$ is the *expected output latency* of b . This value is an estimation of where b 's tuples currently are on the QoS-latency curve at the corresponding output. In other words, this value provides a lower bound on the expected latency of the corresponding tuples at the output (assuming that the tuples are pushed all the way to the output without further delay). $eol(b)$ is computed by adding the current latency value to

the expected computation time for a given output as follows:

$$eol(b) = latency(b) + ect(D(b))$$

where $D(b)$ is the set of boxes including b and those downstream from b and $ect(D(b))$ is an estimate of how long it will take to process the tuples downstream from b . This *expected computation time*, $ect(b)$, of the tuples in the input queues of box b can be calculated using this formula:

$$ect(b) = \sum_{k \in D(b)} (train(k) \times (cost(k)/o_sel(k)))$$

The output selectivity of a box b , $o_sel(b)$, estimates how many tuples should be processed from b 's queue to produce one tuple at the output. $train(k)$ represents the number of tuples in the input queues of box k (i.e. the train size). $train(k)$ is included in the equation because all tuples preceding the tuple who's expected output latency is being calculated must first be processed through to the output. The utility function shown in equation 4.1 is a measure of the expected QoS (per unit time) that will be lost if the box is not chosen for execution.

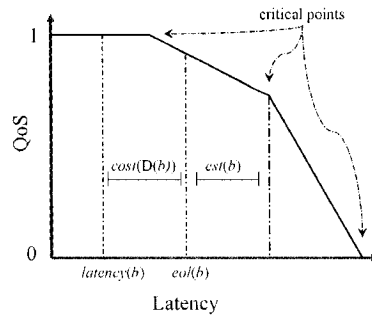


Figure 4.1: Critical points and expected output delay

The *expected slack time*, $est(b)$, is an indication of how close a box is to a critical point; i.e., a point where the QoS changes sharply. Urgency can be trivially computed by subtracting the expected output latency from the latency value that corresponds to the critical point. If there are multiple critical points, $est(b)$ always corresponds to the distance

to the closest critical point. These concepts are illustrated in Figure 4.1, where the QoS is specified as a piece-wise linear function of latency with three critical points.

The expected latency value maps to a slope and slack value on the QoS graph and based on those values, the priority of a box can be determined. To be more precise, at each scheduling point in time, we can order the boxes with respect to their *priority tuple*, or *p-tuple* :

$$priority(b) = (utility(b), -est(b))$$

In other words, we first choose for execution those boxes that have the highest utility, and then choose from among those that have the same utility, the ones that have the minimum (i.e., least) slack time. We call this scheduling technique *p-tuple* scheduling or *SlopeSlack* scheduling.

SlopeSlack scheduling is adapted from *Earliest Deadline First* scheduling [LL73] in real-time systems where tasks are ordered at run-time based on closest deadlines.

4.3 Comparing Fixed-Priority and SlopeSlack

This section first provides an intuition for how scheduling decisions are made at run-time by both Fixed-Priority and Slope-Slack. Then, we compare the performance of both schedulers. Two issues are recognized which can lead to improvements in our techniques:

- Overhead is an issue for these box-at-a-time approaches. The size of the schedule trades off scheduling accuracy with reduction in overhead.
- Boxes are scheduled based on how urgently the tuples in their input queues must get to outputs. Scheduling boxes other than those immediately downstream from an urgent tuple causes the tuple to incur latency which may cause it to miss its deadline.

4.3.1 A Look at the Run-time Schedules

To fully understand the scheduling decisions made at run-time, we can look at individual tuple latencies at the output. This will allow us to infer the scheduling order selected by the schedulers. For simplicity, we consider a simple case with 10 applications and only one box per application. We use a box cost of 0.0001 for each box. Applications are assigned one of 2 QoS graphs: the tighter QoS graph has the following coordinates: $\{(0,1),(0.01,1),(0.0101,0)\}$ and the looser QoS graph has these coordinates: $\{(0,1),(0.02,1),(0.0201,0)\}$. The QoS graphs are assigned alternately to each application (odd numbered applications have loose QoS constraints and even numbered applications have tight constraints). Specific QoS graph coordinates were chosen carefully for this example after multiple test runs. The goal of the test runs was to properly *provision* the system for this network. Provisioning the system entails verifying that for a particular workload, a set of QoS graphs are feasible. That is, under a particular workload, it is possible to achieve a very high level of QoS.

We ran an experiment which passes 100,000 tuples through the query network at a rate of 8000 tuples per second. Figure 4.2 shows the output latencies for all tuples resulting from a run which used the Fixed-Priority scheduler. Each individual point plotted in the figure represents the latency for one tuple. The points in the figure are grouped by application. Values for tuples associated with a particular application are labeled (for example, all tuples output from application 7 are above "A7"). There are 100,000 points plotted and notice that there are 10 clusters of points at various latency values. These clusters represent the latencies experienced by the vast majority of tuples for each application. In fact, the execution order of the boxes (or applications since there is one box per application) can be determined by associating latency with order. We can see that A2's tuples are always output with the lowest latency. This is due to the schedule determined by the Fixed-Priority Scheduler. The actual schedule chosen by the Fixed-Priority scheduler is:

A2, A0, A8, A6, A4, A1, A9, A7, A5, A3

Note from the figure that A0's tuples generally have the second lowest latency, A8's tuples

have the third, and so forth. Also note that the tuples from even numbered applications have lower latencies than those from odd numbered applications. This is because the fixed schedule always orders tighter constrained (even numbered applications) before looser constrained applications (odd numbered applications). Since all even numbered applications have the same QoS graph, an arbitrary order is selected among them a priori. Finally, it is important to note that nearly all tuples arrive before their respective application's QoS deadline. Most even numbered application's tuples arrive before 0.01 seconds have elapsed. And, most tuples for odd numbered applications arrive before 0.02 seconds have elapsed. The average QoS across applications for this experiment was 0.99 out of a possible 1.0.

Figure 4.3 shows the resulting tuple latencies when the SlopeSlack scheduler is used for the same network and workload described above. There is a striking difference between the figures. This actually demonstrates the key difference between *Slope-Slack* and *Fixed-Priority*. Slope-Slack selects the execution order of boxes at run-time. In the figure, there appear to be 5 clusters for the latencies of the tuples for each application. In Figure 4.2 we saw that there were a total of 10 clusters (5 clusters for the even numbered applications and 5 clusters for the odd numbered applications). The difference between the plots is due to the fact that the SlopeSlack scheduler determines box priority at run-time based on tuple urgency. Therefore, any one of the even numbered applications can be scheduled before another, and likewise for the odd numbered applications. The average QoS for this experiment was also 0.99.

In both experiments, a high level of average QoS was achieved. However, the run-time decisions made by each scheduling technique were different. SlopeSlack dynamically adapts to tuples which need to be processed more urgently whereas the Fixed scheduler remains very rigid in its scheduling decisions. Whereas, these experiments provide some intuition for the run-time decisions made by each of our schedulers, they do not give any insight into their relative performance. Next we consider how the techniques compare as we stress the system with increasingly higher load.

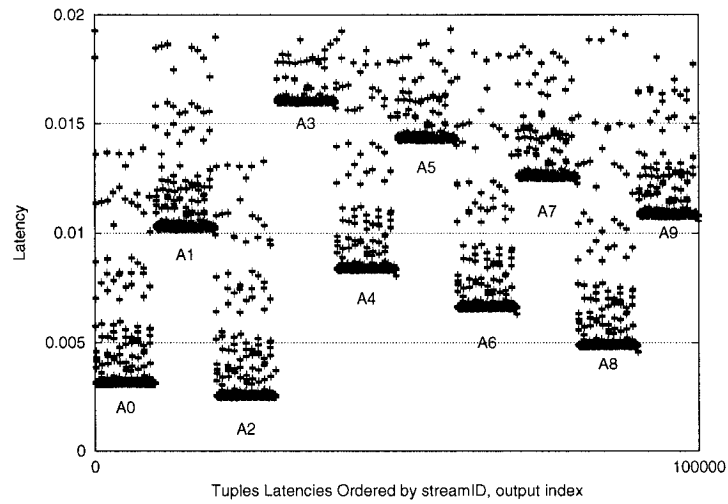


Figure 4.2: Tuple Latencies for Fixed Scheduler

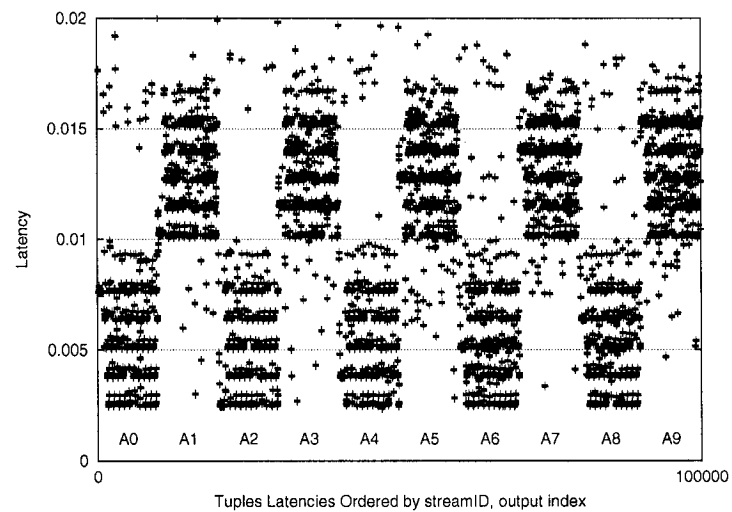


Figure 4.3: Tuple Latencies for SlopeSlack Scheduler

4.3.2 Meeting QoS Deadlines as Load Increases

We now look at a set of experiments where each application consists of more than one box. The network setup is shown in Table 4.2.

Num Apps	20
App Depth	5
Fanout	1
Total Num Boxes	100
Box Cost	0.0001 seconds
Tight QoS Deadline	0.01 seconds
Loose QoS Deadline	2.00 seconds

Table 4.2: Experimental Setup

A note on the experimental setup and QoS deadlines. We use a piece-wise linear latency-based QoS graph as described earlier. The network consists of 20 applications and 2 classes of QoS graphs: 10 with the same tight QoS deadline, and 10 with the same loose QoS deadline. The QoS deadlines listed in Table 4.2 represent the first critical point of their respective QoS graphs and the second critical point is 0.0001 seconds further for each (there is a steep decline in QoS after missing the first critical point).

This simple, relatively small network is used here to aid the discussion comparing various scheduling techniques. In a later section we provide experimental evidence that our results apply to many network configurations of varying scale.

Figure 4.4 shows the QoS performance among the three schedulers we have discussed so far. The *Average QoS* shown in the results refers to the average QoS across all applications. This is computed by first averaging the QoS of all tuples output for each application, then averaging across applications. Since we are measuring Average QoS, it is not surprising to see the Round-Robin Scheduler not performing well. Though it is interesting to see that the two QoS-aware schedulers are able to sustain a high level of QoS for a significantly higher load.

There is also a difference in performance between the dynamic scheduler (SlopeSlack) and the static scheduler (Fixed-Priority). It is not surprising to see that the static scheduler outperforms the dynamic scheduler. Although SlopeSlack dynamically adapts to urgent tuples, each scheduling decision has a higher cost than the simple scheduling decisions of

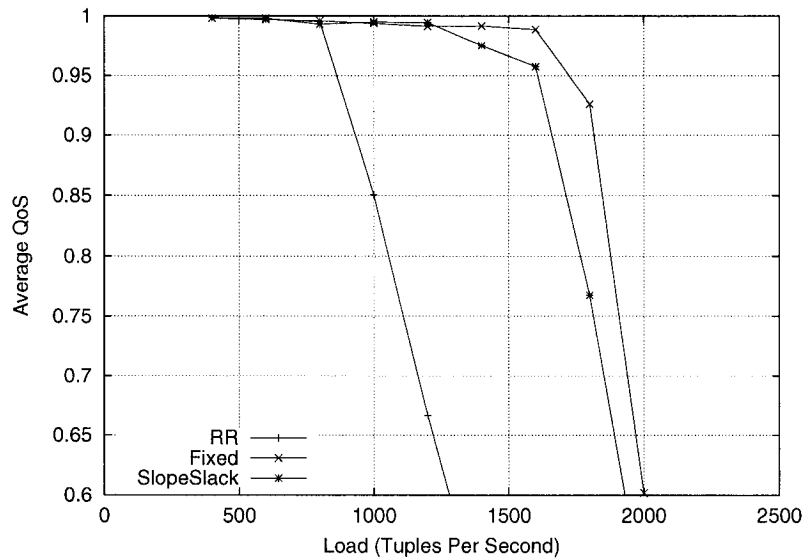


Figure 4.4: Round Robin Scheduler loses to Slope-Slack Scheduler, but, Fixed beats Slope-Slack

Fixed-Priority. The remainder of this chapter focuses on techniques for improving the performance of our scheduling approaches. In particular, the techniques address the following combination of issues which have a significant impact on performance:

- Overhead is an issue for these box-at-a-time approaches. Overhead manifests itself in two ways: the amount of work required by the scheduler to make scheduling decisions, and the frequency with which the scheduler has to make its decision. We address the issues associated with overhead reduction in section 4.4.
- Boxes are scheduled based on how urgently the tuples in their input queues must get to outputs. Scheduling boxes other than those immediately downstream from an urgent tuple causes the tuple to incur latency which may cause it to miss its deadline. We address this issue in section 4.5.

4.4 Overhead Reduction

As mentioned in the previous section, the overhead experienced due to scheduling has two forms. The first overhead results from the frequency with which scheduling decisions are made. Scheduling is considered overhead (the actual useful work is done by the boxes). Each time the scheduler is invoked, it must at least determine which boxes are runnable and sort them. Though invoking the scheduler may not be extremely costly, if done with high frequency, the cumulative cost can grow significantly. So, reducing the number of times the scheduler is invoked can reduce overall overhead. The frequency with which the scheduler must run is a function of the number of tuples processed per scheduling decision. The more tuples processed per scheduling decision, the less frequently the scheduler must run, thus, the lower the overhead. Therefore, the more boxes scheduled per scheduling decision, the lower the overhead. This translates to choosing the correct schedule size so as to trade off the benefit of reducing overhead with the ability to meet QoS deadlines.

The second overhead comes from the amount of processing (CPU time) it takes to make a single scheduling decision. The Fixed-Priority scheduler simply determines which boxes are runnable and sorts them based on an a priori ordering. The Slope-Slack scheduler is a bit more involved. It first determines which boxes are runnable, then scans the box's input queues for tuple latency values, then boxes are sorted as described in Section 4.2.2.

In this section, we first show the impact of schedule size on overhead and QoS, then we present a method to reduce the overhead of the dynamic scheduler.

4.4.1 Schedule Size Matters

Figures 4.5 and 4.6 demonstrate the tradeoffs in selecting various schedule sizes. The figures correspond to the experiment which uses an input rate of 1200 tuples per second from Figure 4.4. Figure 4.5 shows the cumulative overhead (CPU time) incurred for schedule sizes ranging from 1 to 100 (note that there are 100 boxes in the query network). As with any schedule size, the Aurora scheduler will always sort the set of runnable boxes, then select the highest N priority boxes to run. With a schedule size of 1, the scheduler incurs

the significant penalty of sorting all boxes just to run one box. Even for small networks like the one used in this experiment (100 boxes) the scheduling overhead is much too significant to schedule one box at a time. Note that the SlopeSlack and Fixed schedules used in Figure 4.4 used a schedule size of 10. The plots shown in Figure 4.6 show the tradeoff between reducing overhead by using a large schedule size and being able to achieve a high level of QoS. Figures 4.5 and 4.6 show this relationship between overhead and QoS. Notice that QoS is very low when overhead is high (this corresponds to a schedule size of 1). Too much time is spent making scheduling decisions and not doing the useful work of processing tuples. Also notice that when overhead is low (schedule size 20-100) not enough time is spent scheduling urgent tuples and QoS suffers. This demonstrates the tradeoff between reducing overhead and achieving better QoS.

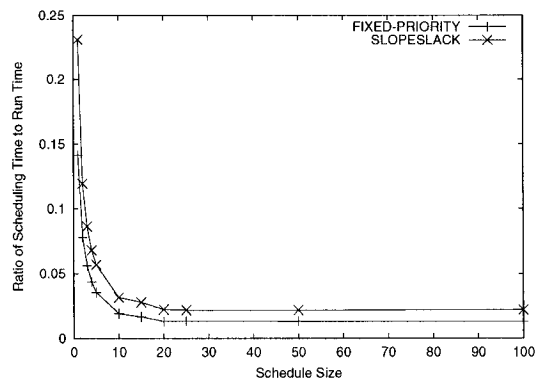


Figure 4.5: Time Spent Scheduling

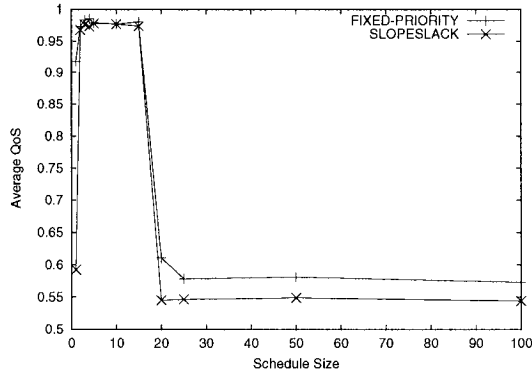


Figure 4.6: Schedule Size has an impact on the performance of scheduling algorithms

4.4.2 Approximation for Scalability

As shown in Sections 4.3.2 and 4.4.1, the Fixed-Priority scheduler is able to sustain a higher level of QoS under heavier load. Here, we will show that by using an approximation technique we can reduce the overhead of the SlopeSlack scheduler and thereby improve its performance.

A straightforward implementation of the SlopeSlack scheduling approach requires, at each scheduling point, computing the p-tuple for each box and then sorting the boxes with respect to their p-tuples. This is an $O(n \times \log n)$ operation, where n is the number of boxes. In order to scale to very large networks, we use a combination of (1) approximation (via bucketing) and (2) pre-computation. The approach is to partition the utility-urgency space into discrete buckets, and efficiently assign boxes to individual buckets based on their p-tuple values at run time. During scheduling, buckets can be traversed in the order of decreasing p-tuples (illustrated in Figure 4.7(a)), and the corresponding boxes are placed in the execution queue. Given a latency value, our first goal is to compute the corresponding bucket assignment in $O(1)$. To do this, we make use of two auxiliary graphs, gradient- and slack-latency graphs.

We divide the range of the gradient (i.e., utility) values into g buckets (Figure 4.7(b) shows an example with four buckets; the cross symbols highlight the latency values where

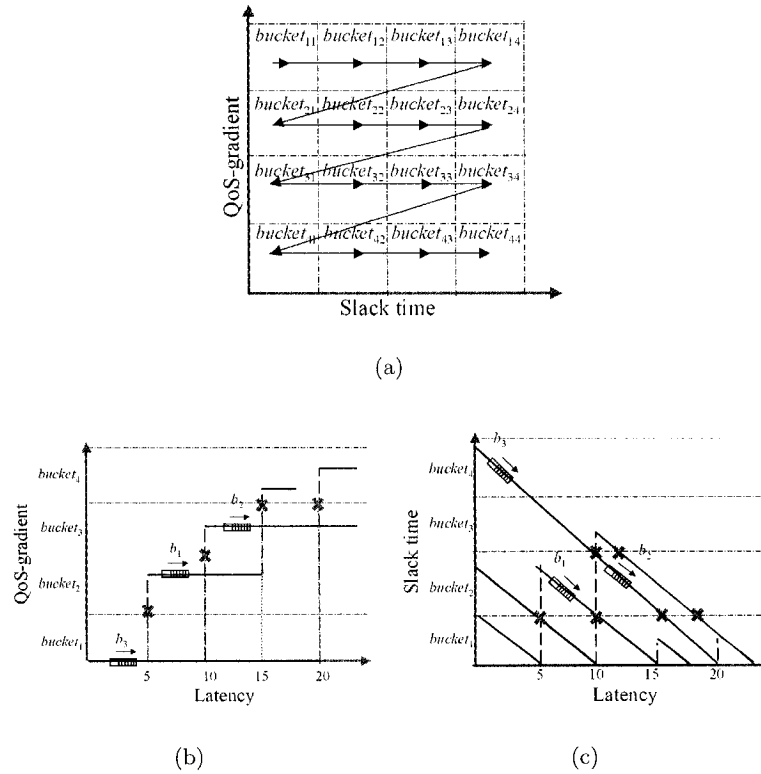


Figure 4.7: Illustrating (a) bucket traversal, (b) gradient-latency graphs, and (c) slack-latency graphs

bucket transitions take place). All gradient values in the same bucket are treated as the same. The width of each bucket, thus, defines a bound on the inaccuracy (or variance) that we are willing to tolerate in terms of the potential deviation from the highest possible gradient value. In other words, the width of a bucket is a measure of the bound on the quantitative deviation from the optimal (with respect to the heuristic) scheduling decision.

Similarly, we divide the slack values into s buckets (Figure 4.7(c)) and treat all the slack values within a single bucket as equal. Again, the width of a bucket is an indication of the level of approximation we make with regards to the slack values.

Given pre-computed gradient-latency graphs, it is possible to pre-compute the application-specific latency ranges that correspond to each bucket. For example, b_1 will be in $bucket_2$

beyond latency = 5 and in $bucket_3$ beyond latency = 15; whereas b_3 will be in $bucket_1$ till latency = 12 and in $bucket_4$ afterwards. Slack-latency graphs can be interpreted in a similar fashion as illustrated in the figure: b_1 falls in $bucket_2$ when latency is between 5 and 10, and in $bucket_1$ for other latency values.

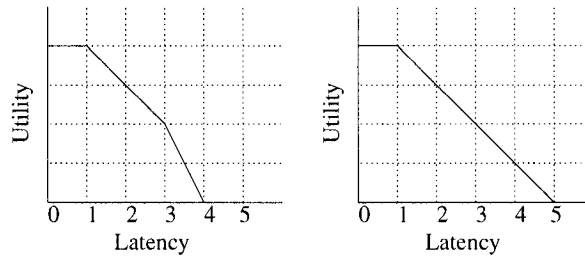
Let's look at a more detailed example. In this example, there are two applications with the QoS graphs shown in Figure 4.8. Figures 4.9 and 4.10 identify the possible slack and slope bucket assignments for tuples in these applications. Consider a tuple queued up in front of application 1. Accounting for downstream box costs, let's say the tuple has a latency of 1.5 time units. When mapped to the QoS graph in Figure 4.8(a) this tuple is at a location which has a gradient of 1 and a slack of 1.5. Recall that gradient is the slope of the QoS curve at a particular latency and slack is the amount of time until the next critical point. Note that the latency value of 1.5 maps directly to *Slack Bucket 2* in Figure 4.9(a). The latency value also maps directly to *Slope Bucket 2* in Figure 4.10(a). These bucket mappings can be directly associated with $bucket_{32}$ in the *bucket traversal* graph shown in Figure 4.11. Table 4.3 gives some examples of bucketing assignments.

Application	Tuple Latency	Slack Bucket	Slope Bucket	Traversal Bucket
1	1.5	2	2	$bucket_{32}$
1	0.5	1	1	$bucket_{41}$
1	3.5	1	4	$bucket_{11}$
2	1.5	4	2	$bucket_{34}$
2	2.5	3	2	$bucket_{33}$

Table 4.3: Bucketing Examples

4.4.2.1 Time Complexity

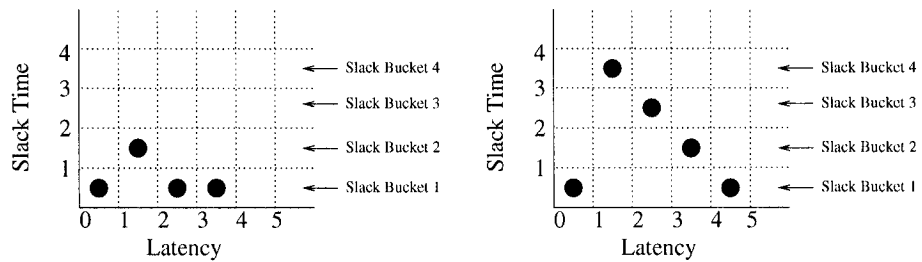
When the execution queue is about to become empty, the scheduler wakes up and performs bucket assignment by going through the boxes and assigning them into their current buckets. A straightforward implementation of bucket assignment takes $O(n)$ time by going through all the boxes, computing the bucket for each box in $O(1)$. This approach has the potential drawback of redundantly reassigning buckets for each box, even if the box's bucket has



(a) QoS Graph for Application 1

(b) QoS Graph for Application 2

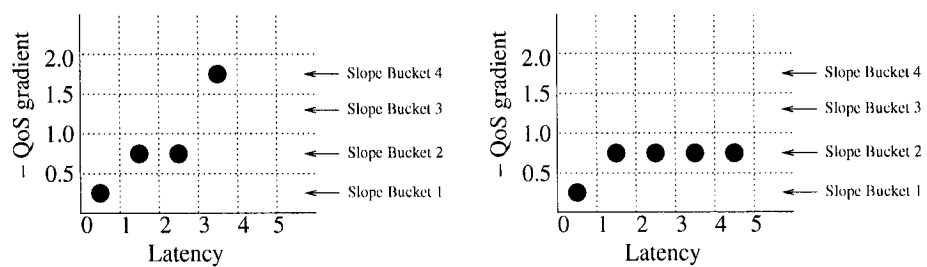
Figure 4.8: QoS graphs for Bucketing Example



(a) Slack Latency Bucket Assignments for Application 1

(b) Slack Latency Bucket Assignments for Application 2

Figure 4.9: Possible Slack Bucket Assignments for Bucketing Example



(a) Slope-Latency Bucket Assignments for Application 1

(b) Slope-Latency Bucket Assignments for Application 2

Figure 4.10: Possible Slope Bucket Assignments for Bucketing Example

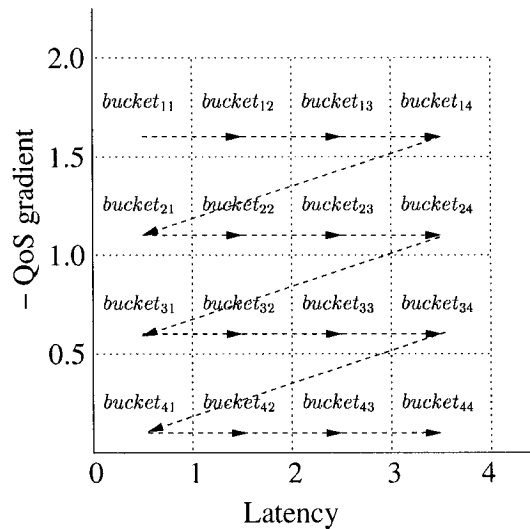


Figure 4.11: Buckets for Bucketing Example

not been changed since the last assignment. In particular, we want the bucket assignment overhead to be proportional to the number of boxes that made a transition to another bucket. In order to accomplish this, we use a calendar queue [Bro88], which is a multi-list priority queue that exhibits $O(1)$ amortized time complexity for the relevant operations (insertion, deletion, and extract-min) under popular event distributions. As a result, we can implement all phases of bucket assignment in constant amortized time.

4.4.3 Bucketing Results

We ran the slope-slack (p-tuple) algorithm and our bucketing algorithm on a network with 200 non-overlapping straight-line applications, each with five boxes. The results are shown in Figure 4.12. The x-axis represents the number of partitions for each of the QoS-gradient and the slack time ranges. We assume that these two dimensions are partitioned equally. Thus, for example, 10 partitions represent 100 buckets.

The slope-slack method produced a measured QoS of 0.796, which is shown for reference on the graph as a horizontal line. When there is only one bucket, the observed QoS is a very poor 0.427. This is because with one bucket all runnable boxes end up in a single grouping

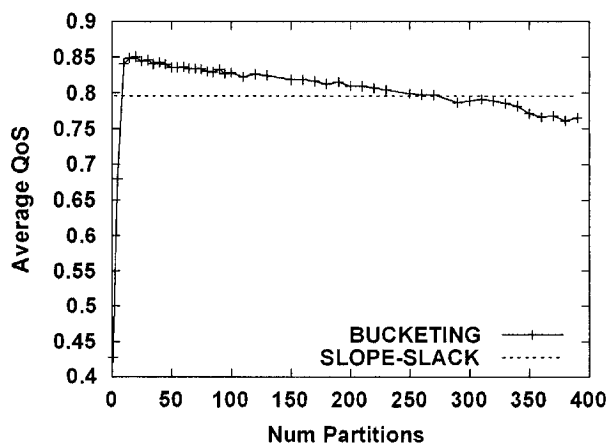


Figure 4.12: Bucketing effects on QoS

which is then equivalent to round-robin scheduling. Notice, however, that as we increase the number of buckets, the QoS rises sharply; until at 20 partitions we reach a maximum QoS value of 0.85. We manage to exceed the slope-slack value (although only by 7%) because the decrease in scheduler overhead dominates the loss of precision in scheduling decisions introduced by bucketing.

Increasing the number of partitions and thus the number of buckets improves the accuracy of scheduling decisions. Working against this effect, though, is the fact that as the number of buckets grows past some moderate level (approximately 30 partitions), the scheduler overhead begins to increase as can be seen in Figure 4.13. Simply having a very large number of buckets becomes a bookkeeping problem. Thus, the scheduler overhead will begin to dominate the incremental gain in accuracy which we see in Figure 4.12 as the QoS curve steadily declines from its maximum and eventually drops below the slope-slack technique at about 260 partitions.

4.5 Superboxes: Immediately Pushing Tuples to Outputs

Until this point, all experiments in this chapter have used *box-at-a-time* scheduling. Box-at-a-time schedulers schedule tuples through individual boxes. This is especially problematic

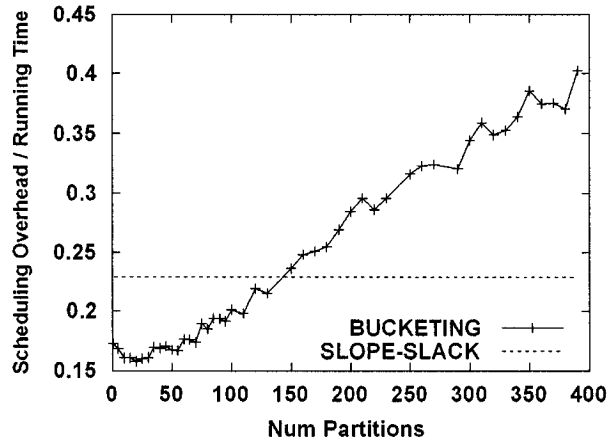
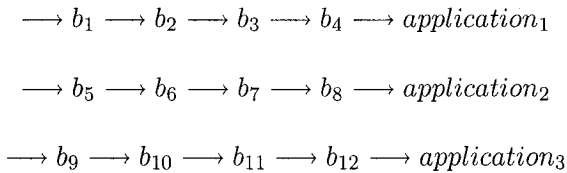


Figure 4.13: Bucketing overheads

for the Slope-Slack scheduler since it orders tuples based on their expected latency if immediately passed through to the output. However, downstream boxes are not scheduled at the time that the tuples were determined to be urgent. Instead, the tuples wait until the scheduler runs again. For example, consider the following network with three applications and 12 boxes:



For simplicity, consider that each box costs 1 time unit to process a single tuple. If we assign the following deadlines (with some slack) for the applications:

*application*₁ deadline : 6 time units

*application*₂ deadline : 10 time units

*application*₃ deadline : 14 time units

Observe that *application*₁ has the tightest deadline and *application*₃ has the loosest. Now, consider that 3 input tuples have arrived at the three network inputs. One possible schedule

to satisfy all deadlines is:

$$b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{12}$$

Ignoring scheduling overhead, *application*₁'s tuple arrives after 4 time units and *application*₃'s tuple arrives after 12 time units. This assumes that the scheduler has recognized that the tuple for *application*₁ should be scheduled first (as would both Slope-Slack and Fixed-Priority). It also assumes that the scheduler scheduled only one box each time a scheduling decision is made. However, as we saw in section 4.4.1 and in figure 4.5, scheduling one-box at a time incurs too much overhead. Therefore, a scheduler must schedule multiple boxes with each scheduling decision. In the example above, if the scheduler were to schedule 2 boxes with each scheduling decision, Fixed-Priority would come up with the following schedule:

$$b_1, b_5, b_2, b_6, b_3, b_7, b_4, b_8, b_9, b_{10}, b_{11}, b_{12}$$

and Slope-Slack would come up with this schedule:

$$b_1, b_5, b_2, b_6, b_3, b_7, b_8, b_9, b_{10}, b_4, b_{11}, b_{12}$$

Both schedules result in *application*₁'s tuple arriving later than its deadline of 6 time units. Note that the Slope-Slack scheduler reduces the priority of a tuple if it cannot meet its deadline given its expected output latency (i.e. the late scheduling of b_4 above). If the scheduler were to schedule 3 boxes at a time, the Slope-Slack scheduler would actually miss two application's deadlines with the following schedule:

$$b_1, b_5, b_9, b_2, b_6, b_{10}, b_7, b_{11}, b_3, b_{12}, b_4, b_8$$

Notice that b_3 was scheduled even though it has already missed its deadline due to latency incurred by executing other boxes. Since the schedule size is 3, b_3 had to be scheduled to fill in the extra schedule slot. The Fixed-Priority scheduler would only miss *application*₁'s deadline for a schedule size of 3.

To solve this problem we next present a technique we call *push-through* which allows for making several scheduling decisions at once while also meeting urgent tuple's deadlines. The

basic idea is simple. When a box is scheduled, all downstream boxes are also scheduled to run in sequence immediately after the first box is run. This creates a *superbox* (see Chapter 3) which is a path from input to output. This push-through technique can be applied to both the Slope-Slack and Fixed-Priority schedulers. In addition to the benefits gained from the scheduling of superboxes (lower overhead), scheduling decisions result in more timely output tuples with respect to their associated QoS graphs. We can use the above example to demonstrate. When the scheduler runs, it determines that the input tuple for b_1 is the most urgent based on $application_1$'s QoS deadline. The scheduler then creates a superbox which includes boxes b_1, b_2, b_3 and b_4 and is ordered such that the tuple is processed through to the output (note that the superbox and its execution order can be determined a priori and stored). The superboxes in the above example are:

$$\{b_1, b_2, b_3, b_4\}, \{b_5, b_6, b_7, b_8\}, \{b_9, b_{10}, b_{11}, b_{12}\}$$

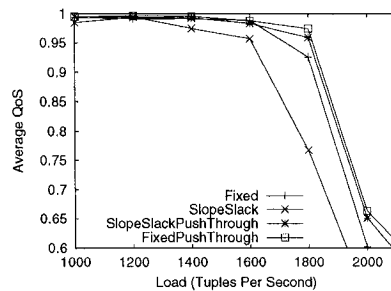
When a superbox is executed, input tuples are processed such that no further scheduling decisions are required in order for them to reach outputs. Scheduling superboxes reduces the total number of scheduling decisions required per tuple.

We explore two push-through schedulers. Both prioritize input boxes and schedule the path of boxes from the input box to the output as a unit. The *FixedPriorityPushThrough* scheduler prioritizes input boxes based on an a priori ordering created in the way described in Section 4.2.1 The *SlopeSlackPushThrough* scheduler prioritizes input boxes dynamically based on the urgency of tuples in the way described in Section 4.2.2.

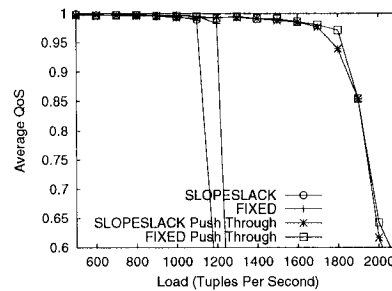
We ran the same experiment as shown in Figure 4.4 using the *FixedPriorityPushThrough* and *SlopeSlackPushThrough* schedulers. The result is shown in Figure 4.14(a). The results for Fixed-Priority and Slope-Slack (box-at-a-time schedulers) are included for comparison. Notice that a significant improvement is seen for *SlopeSlackPushThrough* when compared with its box-at-a-time pre-cursor, *SlopeSlack*. This is due to now pushing urgent tuples to outputs. Also notice that not much of an improvement is gained for Fixed with push-through. This result demonstrates two things. First, *SlopeSlack* without push-through has a fundamental flaw. It determines that tuples are urgent and should be processed

to outputs immediately, however, it does not schedule to process those tuples to outputs. Second, for relatively small, shallow query networks, creating superboxes provides only modest improvements in scheduler performance.

The network used for the experiment in Figure 4.14(a) is very shallow (5 boxes deep). A box-at-a-time scheduler suffers from significant scheduling overhead especially with deep queries. Figure 4.14(b) demonstrates the impact deeper queries have on box-at-a-time schedulers versus the push-through versions. This new network also has 20 applications and the same QoS graph assignments as that used for the experiment in figure 4.14(a). However, each application is 4 times as deep (20 boxes) and each box has $\frac{1}{4}$ the cost (0.000025 seconds). Notice from the plots that the push-through schedulers demonstrate similar performance in both experiments. From a scheduling perspective, the two query networks appear very similar with regard to push-through scheduling. There are four superboxes to schedule in both cases. In contrast, for the box-at-a-time schedulers, the applications are very different. Each application has four times as many boxes which means significantly higher scheduling overhead compared to the more shallow applications.



(a) Pushing tuples through to outputs benefits SlopeSlack most (20 applications, 5 boxes per application)



(b) Pushing tuples through to outputs provides highest benefit for deeper queries (20 applications, 20 boxes per application)

Figure 4.14: Scheduling superboxes

4.5.1 Very Large Query Networks with Superboxes and Approximation

In section 4.4.2 we presented an approximation technique for scaling the SlopeSlack scheduler to larger query networks. In that section, we applied the approximation to the the box-at-a-time SlopeSlack scheduler. Here, we combine the approximation with the PushThrough technique (hereafter called the BucketingPushThrough scheduler). This is done in the same way it is done for SlopeSlack and FixedPriority. Tuples enter the system and boxes nearest the inputs become runnable. Those boxes are prioritized by the BucketingPushThrough scheduler. For each prioritized box, the superbox which forms the path from that input box to its downstream output box is run. The combination of reduced overhead from the use of superboxes, immediately pushing tuples to outputs, and the reduced scheduling overhead of the approximation result in a scheduler which provides the highest level of QoS under heavier load.

Here we show the performance of our scheduling techniques on a large query network. We consider a very large network with 10,000 operator boxes. Table 4.4 shows the experimental setup for this query network. Note that four QoS deadlines are used and are assigned alternately to each of the applications.

Num Apps	1000
App Depth	10
Fanout	1
Box Cost	0.00000500 seconds
QoS Deadline 1	0.075 seconds
QoS Deadline 2	0.150 seconds
QoS Deadline 3	0.225 seconds
QoS Deadline 4	0.300 seconds

Table 4.4: Experimental Setup for Large Network

Figure 4.15 shows the QoS results for this experiment. We tested several schedule sizes for this experiment and found that a schedule size of 300 enabled both the FixedPriorityPushThrough and SlopeSlackPushThrough schedulers to sustain the highest workloads. Also, we tested several bucket values for the BucketingPushThrough scheduler and found

that good performance was achieved with 50 buckets. The BucketingPushThrough scheduler is able to sustain almost perfect QoS until a rate of 4500 tuples per second. This is nearly a 30% improvement over the 3500 tuples per second that both of the other schedulers are able to sustain. Examining the overhead for particular points in the graph reveals the reason for the performance differences. Table 4.5 shows the overhead of each scheduler as a percentage of run-time execution at an input rate of 4000 tuples per second. It is interesting to see that SlopeSlackPushThrough incurs more than twice the overhead of the FixedPriorityPushThrough scheduler yet provides nearly comparable performance. This indicates that it is making good run-time scheduling decisions, however, it is impeded by its own overhead. This is validated in the performance of the BucketingPushThrough scheduler. Decisions made by the BucketingPushThrough scheduler represent approximations of those which would be made by the SlopeSlackPushThrough scheduler but with significantly lower overhead.

BucketingPushThrough	4%
FixedPriorityPushThrough	7.5%
SlopeSlackPushThrough	16%

Table 4.5: Scheduling overhead as a percentage of run-time

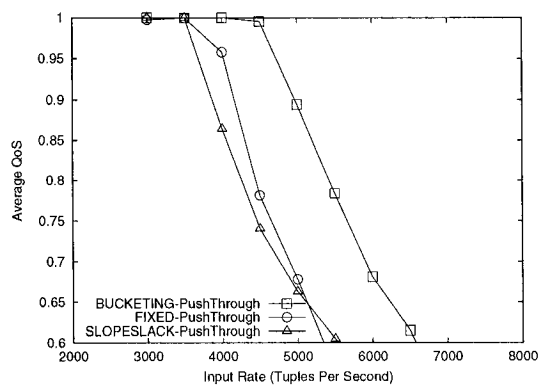


Figure 4.15: Bucketing makes best scheduling decisions for large query networks

4.6 Overload

The workload presented to "monitoring" applications is often inconsistent and bursty. We present two overload conditions and show that the Slope-Slack scheduler adapts dynamically resulting in a higher level of QoS.

There are two types of overload that can occur in Aurora. *Capacity-overload* occurs when there are not enough resources (usually CPU) to handle the workload. For example, if a box can process 1 tuple per time unit and the rate of tuples into the box is greater than 1 tuple per time unit, there will not be enough CPU capacity to handle the input rate. Another type of overload is *QoS-overload*. Consider this example network with 2 boxes:

$$\longrightarrow b_1 \longrightarrow b_2 \longrightarrow \text{application}$$

If the application has a QoS deadline of 2 time units and each box costs 1 time unit, then on average $\frac{1}{2}$ tuple per time unit is the max load that this application can handle. In a real environment, tuples will appear at inputs in batches and queues of tuples will naturally form. So, although this application can handle $\frac{1}{2}$ tuple per time unit, two tuples may appear at the same time. Since Aurora runs tuple trains through boxes, the QoS deadline will not be met. Both tuples will pass through b_1 before either passes through b_2 . This is an example of *QoS-overload*.

Based on queueing theory, Capacity Overload implies QoS Overload (if the rate of tuples entering is greater than the rate at which they can be processed, queues will back up and tuple latency will grow monotonically). However, QoS Overload does not imply Capacity Overload. In general, we expect that Aurora networks and their associated QoS graphs are provisioned so as to not be either QoS-overloaded or Capacity-overloaded. However, since Aurora is a stream processing engine designed to handle monitoring applications, one must expect inconsistent workloads which could lead to brief or sustained periods of overload on either individual inputs or all inputs. Here we will look at two specific overload conditions which have significant impact on our schedulers:

- *Sustained Overload*: Some number of tightly constrained applications experience input

overload for the duration of an experiment.

- *Fluctuating workload*: All inputs experience an overloading burst for some period of time, then return to a normal underload input rate.

4.6.1 Sustained Overload

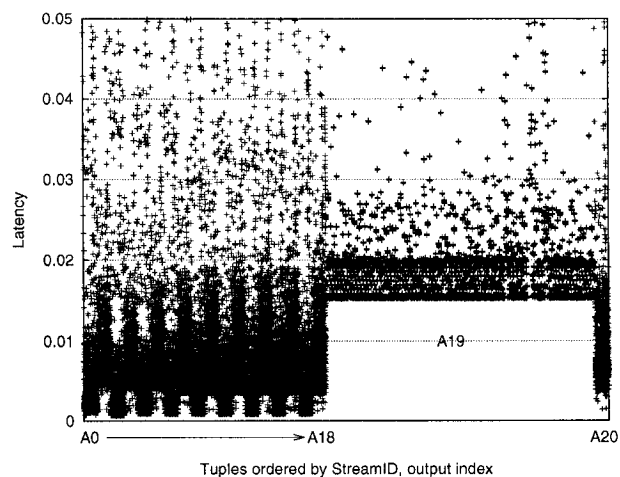
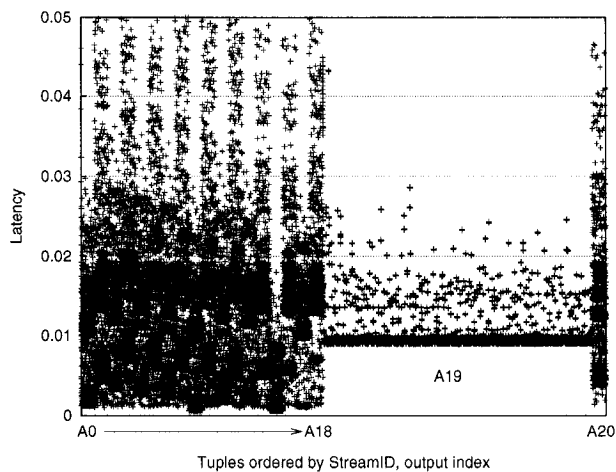
The fundamental difference between SlopeSlack and Fixed-Priority is the ability to adapt to changes in workload. We begin our discussion with a simple experiment. In this experiment, we use the same setup as that presented in earlier sections (see Table 4.2), however, we use a different workload. Here, one input is experiencing a heavy workload for the duration of the experiment. In fact, the workload overloads the application in both capacity and QoS. Figure 4.16 shows the latencies of the tuples. For simplicity, the overloaded application is the most tightly constrained application (A19 in the example). *Fixed* would give this application the highest priority and since this application is receiving an overload, it generally always has tuples to process each time the scheduler makes a decision. Therefore, it is always scheduled. Also, due to the overload, a significant amount of CPU is used to process these tuples. This comes at a cost to other applications which usually have to wait until all of A19's tuples have been processed. This has a significant negative impact on overall QoS, and, as can be seen in Figure 4.16, all tightly constrained applications (even numbered applications) output a significant proportion of tuples which miss their QoS deadline of 0.01 seconds. The average QoS for this experiment was 0.65.

In contrast, the Slope-Slack scheduler adapts to the overload situation. Slope-Slack recognizes when tuples will not meet a QoS deadline. It prioritizes tuples based on their expected utility. Tuples which arrive at outputs beyond a QoS deadline will provide no utility and are therefore given very low priority. Figure 4.17 demonstrates this. The result shown in Figure 4.17 uses the same experimental setup as Figure 4.16, but the SlopeSlackPushThrough scheduler is executed. Notice that the tuples for A19 experience higher latency than other application's tuples. This is due to the SlopeSlack scheduler recognizing that A19's tuples will not meet QoS deadlines and that CPU resources would be wasted

processing A19's tuples. Instead, CPU is allocated for the processing of tuples which will meet QoS deadlines. This results in a higher average QoS of 0.9.

Table 4.6 shows an important difference between measuring for missed deadlines versus measuring for overall QoS. *Average Per Tuple QoS* measures the QoS achieved by each tuple and averages across all tuples. *Average QoS* averages the QoS achieved by each application. The values in the table indicate that FixedPriorityPushThrough misses fewer total deadlines whereas SlopeSlackPushThrough provides higher overall QoS. Average QoS is a measure of the happiness across applications.

This example is presented for illustrative purposes. In this particular example, there is a way to make FixedPriorityPushThrough behave in a manner that provides a higher level of QoS. If the train size (number of tuples processed by a box) is fixed at some small number, then the overload of tuples on the one application will not cause the majority of processing time to be spent on the one application. If the schedule size is larger than the number of inputs which are overloaded, other inputs will receive some processing resource. In the example shown, the schedule size is 10, which does give some processing to other applications. It is the fact that A19 processes all available tuples in its input queue each time it is scheduled which causes significant delay for all other applications. However, decreasing the train size does not entirely solve the problem. Consider the case where the number of applications which are overloaded is equal to the schedule size. In this case, whenever the scheduler runs those overloaded inputs will most certainly have tuples to process and will be scheduled leaving the underloaded inputs unscheduled. The schedule size could be increased, however, as we discussed earlier under normal (non-overloaded) circumstances the best QoS would not be achieved (see Figure 4.6). The point here is not that there is an appropriate schedule or train size to select to best provision for sustained overload conditions. Rather, it is important to note that the SlopeSlack scheduler is able to adapt to changes in workload.



	Average Per Tuple Qos	Average Qos
Fixed Push Through	0.740956	0.653415
Slope-Slack Push Through	0.460723	0.898422

Table 4.6: This table shows the QoS results for Figures 4.16 and 4.17

4.6.2 Fluctuating Workload

The primary goal of the Aurora Stream Processing Engine is to support monitoring applications. The data being analyzed often comes from high-rate feeds which have variable and fluctuating rates. Because of this, the Aurora scheduler must be able to adapt to changes in workload. We now study an experiment where the workload changes over time. In particular we look at the case where overload occurs during bursts of higher-than-normal input rates. In this experiment, the system receives a load which is 60% of its capacity for long durations (40000 tuples). Periodically, the load on the system increases to 130% of capacity for a short duration (15000 tuples).

Figures 4.18 and 4.19 demonstrate how FixedPriorityPushThrough and SlopeSlackPushThrough react during this *bursty* workload. Again, we are using the same experimental set up shown earlier (see Table 4.2). Figure 4.18 shows the Running Average QoS as each of the 500,000 tuples are processed. During periods of overload, the system cannot sustain a high level of QoS and during underload, the system is able to recover. Notice the sawtooth shape of each of the plots in the figure. The declining edge corresponds to a period of overload, and likewise, the increasing edge corresponds to a period of underload. The metric *Average QoS* can be slightly misleading. This is the *running average* QoS meaning that it is the average of all tuples which have been output since the experiment started. Notice that the sawtooth becomes less jagged as the number of tuples increases as should be expected when measuring running average. It is important to note that the average QoS for the SlopeSlackPushThrough scheduler is just above 0.81 and the Average QoS for the FixedPriorityPushThrough scheduler is 0.7.

Figure 4.19 provides a better view of what is actually happening as tuples are output. For every 1000 tuples output, the plots show the average QoS for those 1000 tuples. In this way, we can see what the applications would experience as the system runs and bursts occur. The bursts are easy to identify in the figure. The first burst begins at 40,000 and ends at 55,000 tuples. For the duration of a burst, the level of QoS that both schedulers are able to sustain drops dramatically. What is interesting is how quickly each drops. Notice

that the level of QoS for the FixedPriorityPushThrough scheduler case drops immediately to 0.5 whereas the level of QoS for the SlopeSlackPushThrough declines less rapidly. The SlopeSlackPushThrough scheduler adapts to bursts by essentially shutting off applications whose tuples will not satisfy QoS constraints. It does this at run-time as it calculates the expected output latencies of input tuples. Any queues of tuples which will not meet a QoS deadline receive a very low priority in lieu of tuples which will meet deadlines and provide non-zero QoS.

During overload, the SlopeSlackPushThrough scheduler reacts in a non-intuitive way. In fact, it reacts in a way which is exactly opposite that of the FixedPriorityPushThrough scheduler. Whereas the FixedPriorityPushThrough scheduler always gives highest priority to tightly constrained applications, the SlopeSlackPushThrough prioritizes loosely constrained applications during overload. In general, loosely constrained applications have more slack which allow them to tolerate longer trains of tuples which is exactly what happens during an overload situation. The more tightly constrained applications have less slack and cannot tolerate long trains. So, while the FixedPriorityPushThrough scheduler is busy scheduling tuples which will never meet QoS deadlines, the SlopeSlackPushThrough scheduler is focusing on tuples which have a better chance of meeting QoS deadlines. In fact, the FixedPriorityPushThrough scheduler is blind to the fact that QoS deadlines will not be met.

The barcharts shown in figure 4.20 depict the number of tuples which missed deadlines during the execution of this experiment. Each bar represents the number of tuples which missed deadlines for a particular application. The application numbers are listed along the x-axis. Even numbered applications are tightly constrained. Figure 4.20(b) shows all missed deadlines when the SlopeSlackPushThrough scheduler was used. Notice that there is a clear distinction in Figure 4.20(b) between odd and even numbered applications. Each of the even numbered applications had nearly 3000 tuples which missed deadlines. The odd numbered applications missed considerably fewer deadlines. When compared to missed deadlines by the FixedPriorityPushThrough scheduler shown in Figure 4.20(a) we notice

that the number of missed deadlines for even numbered applications is comparable. But, there is a significant difference when comparing the odd numbered applications.

To further understand the run-time decisions made by each of the schedulers and the differences between the barcharts of Figure 4.20, we tagged each tuple generated as either generated during overload or generated during underload. Figures 4.21 and 4.22 show each of these cases. Figure 4.21(a) and Figure 4.22(a) show that both schedulers are not able to meet the deadlines for the tightly constrained applications for the tuples generated during overload. Even though the deadlines cannot be met, the FixedPushThrough scheduler schedules them anyway at the expense of missing deadlines for the looser constrained applications. In contrast, Figure 4.22(a) shows that the tuples generated during overload for the looser constrained applications do meet QoS deadlines. A summary of the missed deadlines is shown in Table 4.7.

So, what happens to those tuples which are not scheduled during the overload? The queues back up for those applications and the system must wait for the underload to catch up. Figure 4.22(b) depicts this "catch-up" time for the SlopeSlackPushThrough scheduler. Any tuples generated at the beginning of the underload period must wait until the backed up queues are emptied. This incurs latency and causes the initial tuples generated during underload to miss QoS deadlines. The figure shows tuples for even numbered applications missing deadlines. Notice that again, no tuples for odd numbered applications miss deadlines. In contrast, Figure 4.21(b) shows that tuples for all applications miss deadlines.

The above results show that the SlopeSlackPushThrough scheduler is more adaptive to overload and during overload provides resources first to tuples which will provide utility to applications.

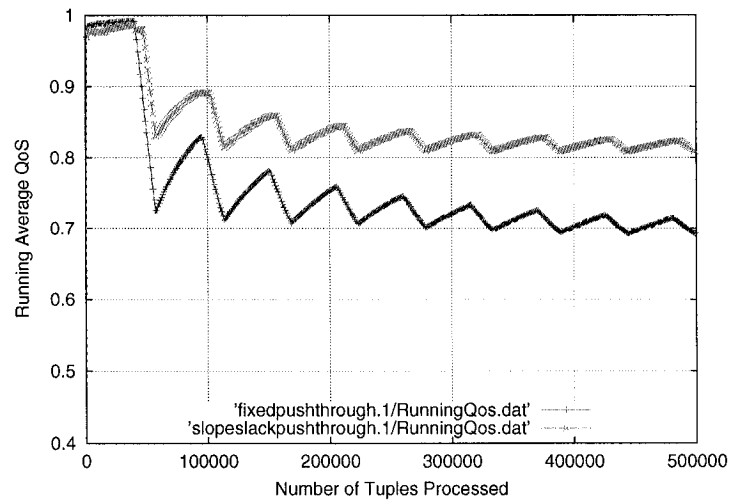


Figure 4.18: Fluctuating Workload: Running Average QoS

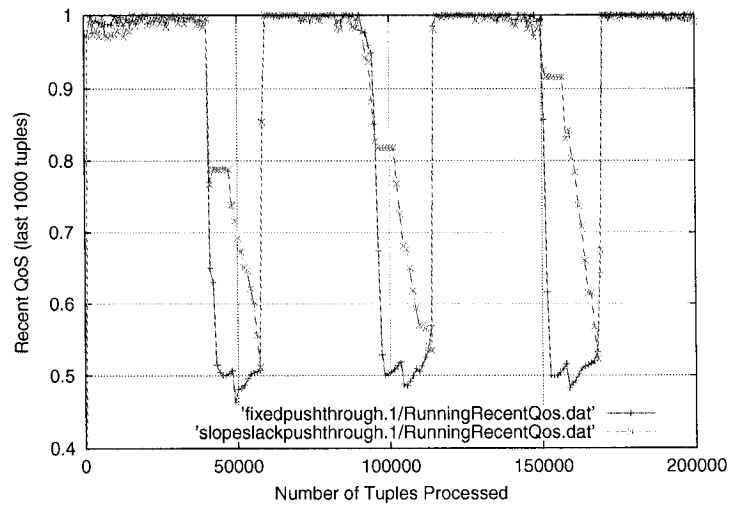
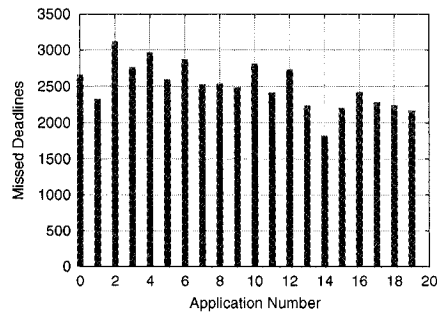
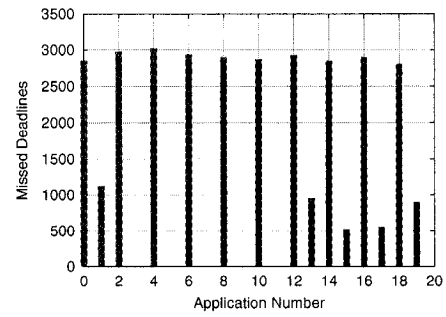


Figure 4.19: Fluctuating Workload: Running Recent Average QoS

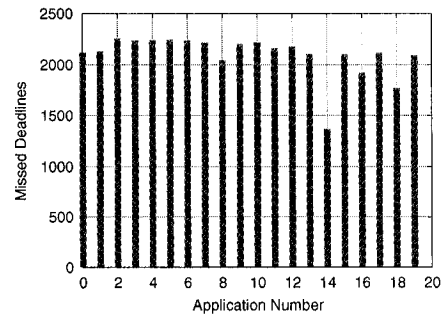


(a) FixedPriorityPushThrough

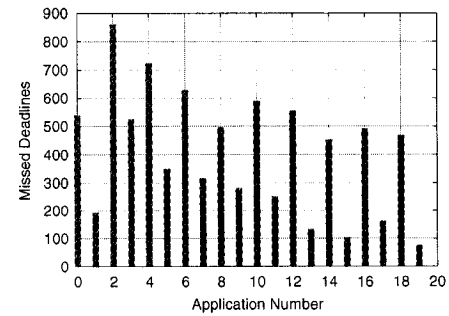


(b) SlopeSlackPushThrough

Figure 4.20: Fluctuating Workload: All missed deadlines



(a) Overload



(b) Underload

Figure 4.21: Fluctuating Workload: FixedPriorityPushThrough

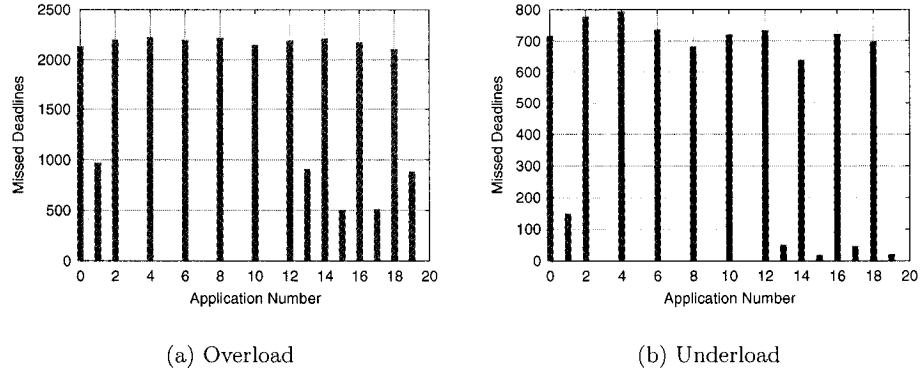


Figure 4.22: Fluctuating Workload: SlopeSlackPushThrough

	Overload Missed	Overload Made
FIXED-PRIORITY	41965	3073
SLOPESLACK	25596	19426
	Underload Missed	Underload Made
FIXED-PRIORITY	8221	146741
SLOPESLACK	7513	147465

Table 4.7: Summary of Missed Deadlines

4.7 Sensitivity to Network Topology

All of the experiments in this chapter have used query networks where each query has a fanout of one (each query is a single path from input to output). Whereas these query topologies aid in the understanding of our scheduling techniques, the results shown also generalize to more complex query topologies (e.g. query trees where the fanout is greater than one)

Since we consider only non-stateful boxes (filter, map, union, etc.) in our work, query trees do not pose any more complex scheduling problems while using a single CPU machine for the techniques discussed in this chapter. Consider the query trees shown in Figures 4.23 and 4.24. For the moment, assume that all outputs from the queries shown in Figure 4.24 have the same QoS graph as the query shown in figure 4.23. Also, assume that all boxes

have the same cost and selectivity.

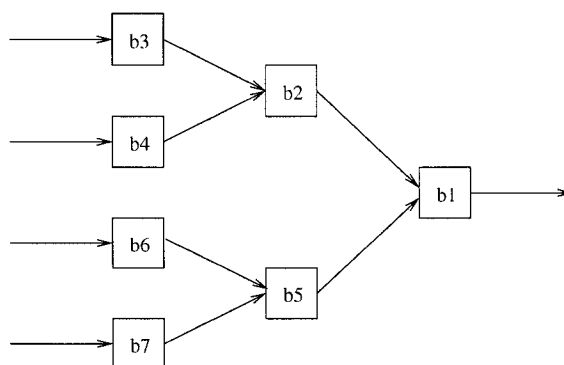


Figure 4.23: Fanout Query

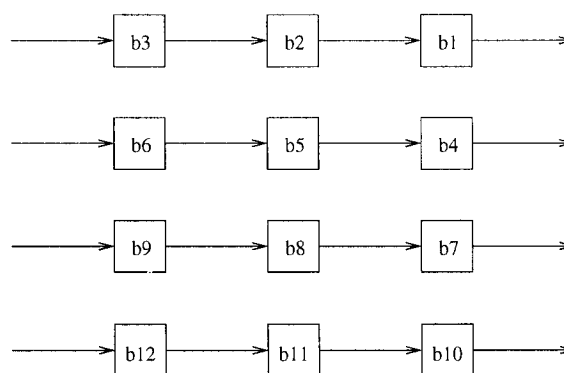


Figure 4.24: Straight Query

To the *PushThrough* techniques described earlier, the query network in figure 4.23 appears very much like the query shown in figure 4.24. That is, there are 4 paths in both which are essentially treated the same by the scheduler. Tuples queued in the input stream of box on the left-hand side are processed through all down-stream boxes as a result of one scheduling decision. Note that we are only considering single CPU machines in our studies, therefore, there is no conflict over processing the same operator box on different CPUs at the same time.

The performance of *box-at-a-time* scheduling techniques is relative to the number of boxes in a query network. Therefore, the same assessment cannot be made of *box-at-a-time*

techniques. Obviously, there are fewer boxes in Figure 4.23 than in Figure 4.24. As a general rule, box-at-a-time scheduling does not provide as high a level of performance as does PushThrough scheduling and we do not consider them here.

We now show a set of experiments which demonstrate the above. In the first experiment, we consider a query network with 10 applications each of which has a fanout of 2 and a depth of 4. Each application has 8 inputs and 1 output and a total of 15 boxes. There are a total of 80 inputs and 10 outputs. Half of the applications have a tight QoS constraint $((0,1), (0.04,1), (0.0401,0))$ and the other half have a loose QoS constraint $((0,1), (3.04,1), (3.0401,0))$. The QoS graphs are alternately assigned. Figure 4.25 shows the relative performance of our three PushThrough schedulers. Note that as in earlier studies, the BucketingPushThrough scheduler is able to sustain a higher load than both FixedPriorityPushThrough and SlopeSlackPushThrough.

To demonstrate the above claim regarding fanout queries being comparable to single path queries, we also consider an experiment with 80 applications each of which has a fanout of 1 and a depth of 4. Like the first experiment, the same QoS graphs are alternately assigned. Based on the above argument, the PushThrough schedulers should provide very similar performance for both experiments. Figure 4.26 show the results of this experiment. Notice that the results in Figures 4.25 and 4.26 are essentially the same.

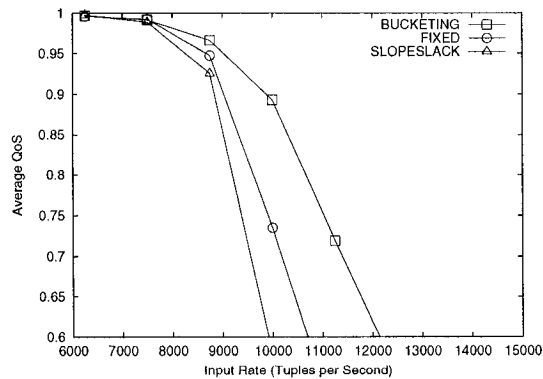


Figure 4.25: QoS results for Fanout Query

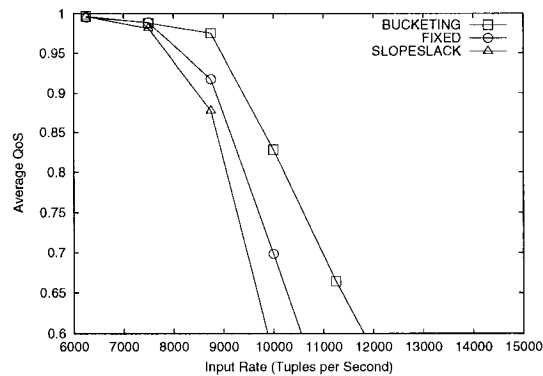


Figure 4.26: QoS results for Straight Query

4.8 Summary of QoS Scheduling for Aurora

Here we summarize our two basic techniques for QoS scheduling in the Aurora Stream Processing Engine.

A Stream Processing Engine (SPE) executes applications which monitor continuous streams of data. These applications have a low tolerance for stale data (tolerance is defined utility specified in a latency-based QoS graph). Therefore, a SPE must employ intelligent resource management. This resource management is performed by a scheduler which must allocate sufficient resources to urgent tuples while not consuming too much of the resource itself while making the scheduling decision. There are two approaches to scheduling in Aurora which take QoS into account. The approaches trade off scheduling overhead with scheduling accuracy.

First, a static approach to scheduling determines *a priori* the scheduling order for operators. The Fixed-Priority scheduler described in this chapter is a static Aurora scheduler which orders operators based on application QoS graphs. The main advantages of this scheduler are that it is very straight-forward, easy to implement, and requires little state. In general, this scheduler provides good performance under high load for applications with very consistent workloads and attainable QoS expectations (i.e. the application has been properly *provisioned*). The Fixed-Priority scheduler does not use the current state of QoS

in scheduling decisions, therefore, QoS suffers when this static scheduler allocates resources to tuples which have no hope of meeting a deadline. This can happen during the overloads caused by bursty or fluctuating workloads (see section 4.6).

A second scheduling approach dynamically determines a schedule at each scheduling event. The SlopeSlack scheduler described in this chapter is a dynamic Aurora scheduler which orders the execution of operators based on the urgency of tuples and their potential contribution to QoS. The implementation of the SlopeSlack scheduler is non-trivial. It requires both maintaining the state of tuple latencies and using an approximation technique (described in section 4.4.2) which reduces its overhead. The benefit of the SlopeSlack scheduler is that it adapts to changes in workload so that tuples can meet QoS deadlines. This ability to adapt is especially important when the application is subject to workloads which temporarily overload the system (see section 4.6).

In summary, the Fixed-Priority scheduler is a simple, low-overhead scheduler which uses little run-time information to make scheduling decisions. The SlopeSlack scheduler is a higher overhead approach which adapts scheduling decisions based on the current state of QoS. When approximation is used to reduce the overhead of the SlopeSlack scheduler, it often outperforms the Fixed-Priority scheduler under consistent workloads and is always the best choice when adapting to bursty workloads is necessary.

4.9 Conclusion

This chapter presented an experimental investigation of QoS-aware schedulers for the Aurora Stream Processing Engine. In particular, two scheduling approaches were introduced. First, a static approach which uses a pre-determined schedule was shown to perform well under consistent load. Second, a dynamic approach which determines its schedule at run-time was shown to perform well when combined with an approximation technique to reduce its overhead. Both techniques perform well because they allocate resources to urgent tuples. A study of the two scheduling approaches under bursty workloads revealed that the dynamic scheduler is better able to adapt to changes in workload which overload

the system temporarily by recognizing which tuples will meet QoS deadlines.

Superboxes (used in "push-through" scheduling) were effectively applied to both scheduling approaches. When applied to these QoS-aware schedulers, the use of superboxes significantly reduces scheduling overhead and as a result better levels of QoS are achieved.

In general, effective QoS-aware schedulers reduce scheduling overhead so that more system resources can be allocated for the useful work of processing tuples while still recognizing which tuples are urgent.

Chapter 5

Related Work

5.1 Related Work - Freshness

There has been a great deal of work on managing replicas in distributed computing systems [Son88, KA00, Her86]. Much of this work requires careful cooperation and standard protocols between the participating sites in order to produce the appearance of perfectly synchronized replicas. Because of the autonomy of sites as well as severe performance requirements, Many networked applications cannot support this level of cooperation. Thus, copies are allowed to drift out of synchronization and a freshening system like that described in Chapter 2 and [CLZ03, CLZ02] will periodically intervene to pull them back into agreement.

In data warehousing, much work has been done on managing *materialized views* efficiently [ZGMHW95]. However, the major focus of most of this work is on issues such as minimizing view size or reducing the view update time.

Data freshening shares some similarities with data caching in that they both must deal with the problem of out of date local copies. The problem of managing the freshness of local copies has been explored in the web caching area[Wan99]. Here, however, the model of synchronization is based on freshening actions that happen on explicit data requests from the clients or by updates pushed from server.

Recent work has addressed the maintenance of local copies for environments where exact consistency is not possible due to the limited resources. For *pull-based* approaches, clients decide on a refresh schedule based on knowledge of the change frequency of documents. This information can be obtained using Time-To-Live(TTL)[DP96] information, application of a probabilistic distributions[CGM00a], or sampling from servers[CN02]. A TTL represents the estimated period a web document will remain fresh and is widely used for web cache consistency. One study [CGM00a] discusses how frequency of change can be modeled by a Poisson process and can be estimated from observed data.

In [CGM00b], a freshness metric and synchronization policies are suggested and discussed. The work reported in Chapter 2 extends their work. Their goal was to maximize the average freshness of a collection of copies. However, their work did not consider the user's access patterns. As a result, their freshening approach may not produce the most productive solution from the point of view of what a user actually sees. Further, their approach is not scalable for a very large number of data items, which could be impractical in many applications.

Sampling-based refresh policies [CN02] initially poll some portion of the documents from each server to determine the relative ratio of the changed documents in the samples. Based on the ranking of this ratio, a greedy method that refreshes documents starting from the highest ranked server produces the best result.

Furthermore, our analysis and simulation results reveal that access probability dominates change frequency when the access distribution is highly skewed which is typical in the Internet. As the access probability can be obtained easily at the mirror site, our approach is applicable even in the case with imperfect knowledge of change frequency.

Profiles can be more specific as in [BR02] where a refresh policy using clients' preferences about the recency and latency is known. This work does not consider limited bandwidth. Refresh is done by explicit client request. Our approach could be combined with theirs to provide a more aggressive refresh approach.

When data sources can notify clients of changes, clients can either determine the exact time to pull the changed documents[LR01], or data sources can *push* the changes to clients[OW02]. Update propagation strategies for the materialized WebView[LR01] have been proposed thereby determining the WebView update schedule based on exact knowledge of when objects are updated at the servers. [OW02] proposes a *push-based* refresh approach in which clients and servers cooperate with each other to determine refresh schedules dynamically based on variable bandwidth. However, there are many environments including the Web where this push-based approach can not be applied because data sources do not inform clients of changes.

In order to gain fine-grained control over the tradeoff between precision and performance, adaptive refresh on replicated data has been studied[OW00, OLW01]. The primary goal of this work is to control the amount of refresh over replicated approximate data to satisfy given data precision constraints. The client informs the server about what level of precision it requires, and the server sends the client a new value when the precision interval is violated.

5.2 Related Work - Aurora Scheduling

There has been extensive research on scheduling tasks under real-time performance expectations both in operating systems [JRR97, Loc88, NL97, RS94] and database systems [AGM92, HCL93, HLC91, OS95, Ram93]. To the best of our knowledge, Aurora's scheduling approach that combines priority assignment and dynamic scheduling plan construction is the first comprehensive proposal for scheduling continuous queries over real-time data streams and QoS expectations. Our solutions no doubt borrow a lot from the myriad of existing work on scheduling.

Scheduling proposals for real-time systems commonly considered the issue of priority assignment and consequent task scheduling based on static (table- or priority-driven) approaches or dynamic (planning or best-effort) approaches [Ram93]. Static approaches are inherently ill suited for the potentially unpredictable, aperiodic workloads we assume, as

they assume a static set of highly periodic tasks. Dynamic planning approaches perform feasibility analysis at run-time to determine the set of tasks that can meet their deadlines, and rejecting the others that cannot [JRR97]. This decision is based on two key observations: First, our priority assignment algorithm is based on a variation of Earliest-Deadline-First (EDF) algorithm [Loc88], which is well known to have optimal behavior as long as no overloads occur. Second, Aurora employs a load shedding mechanism (not described here but can be found in [CCC⁺02]) that is initiated when an overload situation is detected and that selectively sheds load to get rid of excess load in a way that least degrades the QoS. This allows our scheduling algorithm to focus only on underload situations. We note here that Haritsa et al. [HLC91] proposed an extension of EDF that is designed to handle overloads through adaptive admission control.

Real-time database systems [AGM92, HCL93, HLC91, KGM94, OS95, Ram93] attempt to satisfy deadlines associates with each incoming transaction, with the goal of minimizing the number of transactions that miss their deadlines. These systems commonly support short-running, independent transactions, whereas Aurora deals with long-running continuous queries over streaming data. Leaving aside these differences, of particular relevance to Aurora scheduling is the work of Haritsa et al. [HCL93] that studied a model where transactions have non-uniform values (or utilities) that drop to zero immediately after their deadlines. They studied different priority assignment algorithms that combine deadline and value information in various ways, one of which is a bucketing technique. This technique is similar to ours in that it assigns schedulable processing units into buckets based on their utility. The differences are that (1) we use bucketing to trade off scheduling quality for scheduling overhead and, consequently, for scalability; and (2) we also use bucketing for keeping track of slack values.

Also related to Aurora scheduling is the work on adaptive query processing and scheduling techniques [AH00, HFC⁺00, UF01]. These techniques address efficient query execution in unpredictable and dynamic environments by revising the query execution plan as the

characteristics of incoming data changes. Eddies [AH00] tuple-at-a-time scheduling provides extreme adaptability but has limited scalability for the types of applications and workloads we address. Urhan's work [UF01] on rate-based pipeline scheduling prioritizes and schedules the flow of data between pipelined operators so that the resulting output rate is maximized. This work does not address multiple query plans (i.e., multiple outputs) or deal with and support the notion of QoS issues (and neither does Eddies).

Related work on continuous queries by Viglas and Naughton [VN02] discusses rate-based query optimization for streaming wide-area information sources in the context of NiagaraCQ [CDTW00]. Similar to Aurora, the STREAM project [BW01] also attempts to provide comprehensive data stream management and processing functionality. The Chain scheduling algorithm [BBDM03] attempts to minimize intermediate queue sizes, an issue that we do not directly address in this thesis. Neither NiagaraCQ nor STREAM has the notion of QoS.

Chapter 6

Future Work

6.1 Introduction

There are many avenues to continue the work presented in this thesis. Here we present several nascent ideas which present opportunities for directions of future study. The future research is conveniently broken up into two subsections. One subsection for future opportunities in the area of data freshening, and one subsection for future directions for Aurora scheduling.

6.2 Future research directions for Data Freshening

6.2.1 Expected Freshness

An interesting problem exists in the current solution where certain elements receive no synchronization bandwidth and therefore never get refreshed. Consider the following scenario. A site might want to ensure that the items it copies maintain some level of freshness (for example, some stock quote sites try to ensure that stock prices are no more than 20 minutes old). To do this a site can simply poll the server for each item at a certain interval (every 20 minutes in the example above). The problem with this simple approach is that bandwidth is wasted on items that change much less frequently than that interval and the

overall freshness of the mirror is not optimized (in the example above, financial instruments that are highly volatile like options should be updated more often than every 20 minutes or else, the data is useless). There can be defined threshold for freshness per data item. (in the example above, stock quotes have a 20 minute threshold, and more volatile instruments might have a 5 minute threshold)

If the site owner is willing to relax the requirement that data items be no more than some amount of time old and instead accept that probabilistically the data will not be more than a certain age, he may be able to maintain a higher overall level of freshness.

The proposed solution is to first require that some level of freshness is maintained for each element. If you know the change rate of an element, you can assign a synchronization frequency to it so as to obtain a given freshness (in addition, if you know the access probability and size of the object you can also find a suitable synchronization frequency). A simple binary search per item can obtain this minimal synchronization frequency. (Doing this on a larger scale might require some heuristics). Thereafter, and with the remaining bandwidth, you assign to items synchronization bandwidth according to our heuristic techniques.

6.2.2 Is bandwidth free?

This problem deals with the development cost model for determining cost/performance tradeoff between bandwidth and perceived freshness. Figure 6.1 shows what might be the relationship between bandwidth and perceived freshness. At some point, there may be the opportunity to significantly reduce bandwidth usage with only a small reduction in perceived freshness. For example, the point *A* located in the figure can be moved in either x direction with little impact on Perceived Freshness. However, a small x move for point *B* would have a significant impact on Perceived Freshness. An obvious opportunity here is to reduced bandwidth cost with little freshness loss.

The goal for this problem might actually be different. If you could reduce bandwidth usage with minimal impact on freshness, you might be able to use the bandwidth to some benefit. The problem described in section 6.2.4 explores one possibility.

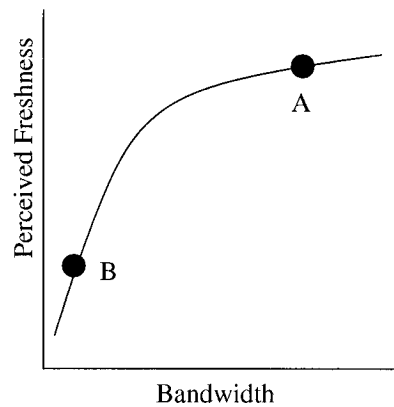


Figure 6.1: Bandwidth/Perceived Freshness Tradeoff

6.2.3 Gathering Statistics

Much of the previous work on Data Freshening with knowledge of change rates assumes perfect knowledge of the change rates of individual data items. In our work to date, we have seen that user interest dominates synchronization bandwidth allocation. Without knowledge of change rates, the only synchronization policies that would make sense would be those which use user interest to allocate synchronization bandwidth. However, since we have seen that change rates play an important role in data freshening, any information about the change rates of data could improve synchronization policies based solely on user interest. The proposal here is to gather statistics about the rates of change of items as they are synchronized. The synchronization server could recognize whether a data item has changed when it is synchronized. If it hasn't, then perhaps its synchronization bandwidth allotment can be reduced while the allotment for another item can be increased.

6.2.4 Split Sync Bandwidth for a Dynamic Synchronization Policy

This problem follows from that described in section 6.2.2. The basic idea is to split the synchronization bandwidth between a fixed synchronization schedule and a dynamic synchronization replacement policy.

In the current model, the synchronization schedule is fixed - this is limiting, as data

changes, and user interests change, the synchronization policy should adapt. As an example, consider the case where a copy of stock prices are being maintained at the mirror site. Enron had been a stock with a fairly stable price, however, on the day the company declared bankruptcy, the stock plummeted, then became very volatile. For our synchronization system, the data item representing Enron's share price would have a new change rate. Synchronization is now in a less than optimal state. The only current solution is to recalculate the entire synchronization schedule. It is likely that for many types of data, this type of occurrence is common. Therefore, an approach that allows for more flexible synchronization is in order.

The proposal here is to develop a dynamic synchronization algorithm which will take advantage of the bandwidth/perceived freshness tradeoff (from section 6.2.2). The dynamic synchronization algorithm can be thought of much like a cache replacement policy, such as LRU. As the rate of changes to a data item increases or the number of requests for the item increases, it moves up the replacement policy chain. Another thought for the policy is to use an on-demand broadcast scheduling policy such as LWF where requests compound the movement along the replacement chain.

To clarify the problem a bit more, Given that a total of B bandwidth is available for synchronization, find a B_s (bandwidth allocated to static synchronization schedule) and B_d (bandwidth allocated to dynamic synchronization) such that $B_s + B_d = B$ and freshness is maximized. Such an approach might maintain a histogram of requests and changes for data items like that shown in fig 6.2. Any page past m would not be eligible for synchronization.

A more complex profile is necessary for this problem. The profile has to associate utility with data values. The solution could include context-dependent profiles which specify how user interests change as the conditions of the data change. The profiles would control the the dynamic caching policy.

Another approach might be to incrementally modify the schedule, where some portion of the schedule is changed (perhaps the schedule of some subset of data objects) This incremental approach tries to reallocate bandwidth (synchronizations) to objects that need

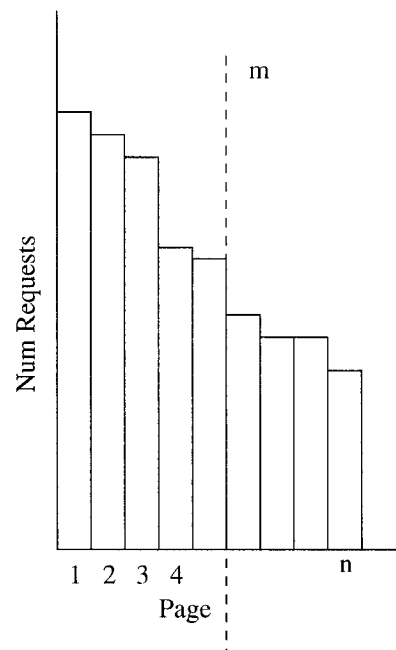


Figure 6.2: Bandwidth/Perceived Freshness Tradeoff

it from objects that require less. From which object(s) can you take X amount of bandwidth away, so as to have the least impact on Perceived Freshness? (this may be related to load shedding)

6.2.5 Pushing to reduce staleness in conjunction with synchronization of cache

An alternative to the use of extra bandwidth proposed in section 6.2.4 is to push changes to users. In the current model, pages are pulled from cache and DB (web). If bandwidth between the user and cache is not free and is not infinite, can you do better by pushing some pages to users (Similar to the schedule that is generated for synchronization, a schedule can be generated for pushing data to users). Question: How much bandwidth is allocated for pushing and how much is allocated for pulling? You have to take synchronization into account (i.e. if a page is going to be synchronized soon, then maybe it should not be pushed).

6.2.6 Page Replacement

Question: How do LRU and profile-driven cache replacement differ? In the current model, the cache is a fixed set of pages, what if you want to synchronize 100,000 pages, but you only have space for 50,000. Is it better to fix the cache at 50,000 with the sync algorithm, or is it better to try to sync all 100,000 and replace pages using a cost function or LRU. And, if the page that you will replace is fresh, should you push it to a user?

6.2.7 Multiple levels of cache

If there are multiple caches (see Figure 6.3) which store objects between the user and the data source, the question of which objects to store in which cache becomes a question. Latency is the issue here - the latency to get the freshest value. If the cache that holds the object that is interesting to the user is far away, then a request for the object incurs a penalty in network latency.

Question: are all of the caches storing the same data? Is it like the memory hierarchy of a computer?

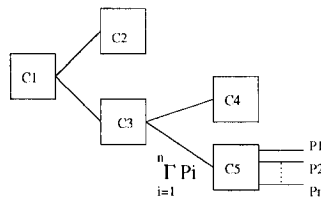


Figure 6.3: Levels of Cache

Multiple levels of cache form a memory hierarchy and objects can be in any level. The users connect to the leaves of the hierarchy. For very large networks, each cache cannot maintain knowledge of the entire network

In Figure 6.3 some function must be devised that determines which objects satisfy profiles and where those objects should be stored. A method must be also devised to identify where objects are located in the network of caches.

6.3 Future research directions for Aurora Scheduling

6.3.1 Selecting Superboxes

A simple approach to scheduling in Aurora would construct superboxes out of applications. That is, the subnetwork that comprises a single application would be the superbox. Also, the execution plan for the superbox would execute each box once over a train of input tuples. However, overlapping queries (common subexpressions, or, common subgraphs in an Aurora network) and more complex operators complicate scheduling.

Dealing with common subexpressions is common in database systems. Similarly, Aurora networks can have common subgraphs. Figure 6.4 shows an Aurora network with three applications which share operators. For example, box 1 is shared by applications A and B, and, box 2 is shared by all three applications.

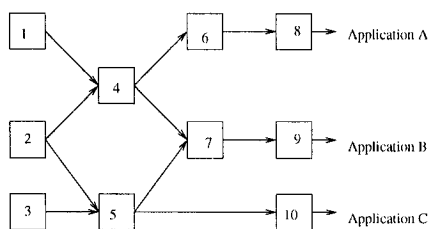


Figure 6.4: Example Aurora Network with overlapping queries

The naive superbox construction and execution technique (constructing superboxes of entire applications) runs into problems when applied to the network in the figure. To start with, only one superbox can be executed at a time since two separate worker threads cannot execute the same box at the same time without a complex synchronization or dependency scheme. Therefore, when a superbox is executed by a workertthread, all boxes contained in the superbox are essentially locked to prevent other threads from executing the boxes.

The Aurora query language consists of operators which vary in complexity, from a simple union which incurs very little overhead to aggregate and SQL operators which potentially incur very high processing or overhead costs. Consider the following portion of a query network:



Figure 6.5: High Cost Windowed Operator Example

And, consider this example scenario. Box 4 is a windowed operator with a cost of 1 time unit for each tuple in its window (its window contains 5 tuples and therefore, it takes 5 time units to process a full window). Box 4 also produces only one output tuple for each full window. Box 6 costs .5 time units and Box 8 costs .5 time units. One tuple arrives at the input to Box 4 per each time unit. So, after 5 units of time, Box 4 can execute over the window and output a tuple. Also consider that a new window is used for every group of 5 tuples. An example execution of this superbox while using tuple trains might work like this: let's say we have a tuple train of size 100. There are 20 windows worth of tuples. The first tuple arrives at time t_1 and the 100th tuple arrives at time t_{100} . Using the superbox and train method described above, the first tuple will be output from box 4 at time t_{105} and the 20th tuple will be output at time t_{200} . Then the tuples pass through box 6 (the first tuple at time $t_{200.5}$ and the last tuple at time t_{210}) then through box 8 (first tuple at time $t_{210.5}$ and the last tuple at t_{220}). The average latency of the tuples output to the application is 210.

If instead, for each tuple output from box b_4 , we then execute box b_6 and then box b_8 , the first tuple will reach the application at time t_{106} and the 20th tuple will reach the application at time t_{220} . The average latency in this scheme for output tuples is 163.

A very high overhead traversal which dynamically considers the tradeoffs of cost and QoS on a per-tuple basis could be considered here. However, simply breaking up the superbox into 2 superboxes might be very effective. We end up with box 4 in its own superbox and boxes 6 and 8 combined as a superbox. Then, if the two superboxes are being executed by two concurrently running threads, one thread could essentially be feeding tuples to the second thread and thereby improving system performance.

There is something to consider here. On a uniprocessor system, the first thread will only yield the processor to the second thread when it is either finished processing or if

it is pre-empted for some reason, maybe for an I/O operation (which may be the reason box 4 has such a high cost). Also, if the first thread does not yield the processor, then a preemptive scheduler may be in order.

6.3.2 Superbox Priority Assignment

Since the quality of Aurora output is measured with QoS graphs, assigning priority to boxes or superboxes becomes more challenging when queries share subexpressions. Defining the QoS graph for a box which is shared by two operators which have different QoS graphs is not straight-forward. Also, assigning priorities to boxes might not be the best way achieve high levels of QoS. Instead, it might be better to assign priority to superboxes. This also is not straight-forward.

6.3.3 Superbox Compositing

Given that subexpressions can be shared, perhaps some sort of Application sharing should be performed. This would entail considering two applications to be the same, combining their QoS specifications and performing their executions together. Perhaps clustering could be used to create superboxes.

6.3.4 Actual Priority Assignment

It may be the case that instead of specifying QoS graphs, applications instead specify priorities. In this case, a priority-based scheduler must be developed which handles application priorities. Operating systems use techniques such as lottery scheduling or stride scheduling [WW95] to slice CPU allocation into discrete time slices (*quanta*) and assign resource rights (*tickets*) to processes. Techniques like these may be applied to Aurora Scheduling.

6.3.5 More interesting queries

A more robust model of experimental query networks must be built. In the model used for the results shown in Chapter 3, query networks are automatically generated using depth

and fan-out (from the application) parameters. All boxes are essentially Union boxes with optional cost and selectivity parameters. While the model is sufficient for studying the scheduling of independent queries, it lacks being able to more realistically model the expected query environment which may include common subexpressions (overlapping query graphs). Therefore, experimental query networks should be generated with a degree of overlap.

It would be interesting to do a performance evaluation which studies the degree of overlap between 2 queries and compares this to overhead of multiple superbox selection techniques.

In [CCR⁺03] experimental studies were performed on query networks with very simple boxes. The boxes were essentially union boxes, however, they were specific to experimentation in that a processing cost and selectivity could be assigned per box. In this way, all possible Aurora network topologies could be studied. However, more complex operators such as *Join*, *Aggregate*, and *SQL Boxes* could not be modeled. To more realistically study scheduling in Aurora we must define a model for such operators and examine their impact on scheduling decisions.

6.3.6 Proposed Scheduling Algorithms

Given that queries will overlap, and aggregates, joins, and SQL boxes may stall superbox execution, our original [CCR⁺03] approaches to superbox construction and priority assignment should be modified appropriately. Below we discuss some possibilities for these modifications.

6.3.6.1 Heuristics for choosing superboxes

In order to address the real-time requirements of an Aurora system, heuristics will be used to select boxes for execution. Here we discuss possible heuristics to use for assembling superboxes. We focus our discussion on static and dynamic heuristics:

- static heuristics: a set of superboxes are selected ahead of time. This would create a

static partitioning of the query network (ad-hoc queries are added as new partitions). If such a heuristic were used, the superbox with the highest accumulated priority is run first. Priority assignment for superboxes is discussed later.

Note: The ordering of superboxes for scheduling might not be straight-forward. If superbox B feeds tuples to superbox A, but A has a higher priority, it might be better to execute B first.

- dynamic heuristics: superboxes are assembled on the fly. That is, once an application or box is selected for execution, other boxes which have a relationship to the box or application are assembled into a superbox. In general, the selection of boxes is based on following paths in either the upstream direction or downstream direction, or both. The characteristics of the queues and boxes along the path are considered as the superbox is built. Further heuristics will include the number of empty queues to skip while moving upstream. Should you stop at an empty queue?

6.3.6.1.1 Static Heuristics

Below we describe 3 different heuristics that could be used to select static superboxes.

1. **Application Trees:** Each application has a sub-network of boxes associated with it. The sub-network is a tree which is rooted at the output port of the application. This is the approach, AAAT, used in Chapter 3.
 - Advantage: easy to generate
 - Disadvantage: Application trees overlap. Running one application tree, and then running another which has significant overlap with the first, runs the same computation twice. This might be wasteful since it incurs significantly higher box call overhead.
2. **Intersections of application trees:** Application trees overlap and the overlap can be clearly determined. Whenever a split is found in the network, the subnetwork rooted at the split is a new superbox.

- Advantage: easy to generate, computation is shared between applications
- Disadvantage: isolating boxes (or superboxes) per application is not as straight forward as for the AAAT approach described above.

3. **Trains of single input boxes:** Boxes can be executed if they have enough data in their respective queues for evaluation. It is difficult to determine if all the boxes of a superbox which contains a complex sub-network will have a sufficient number of tuples so that each box will execute over input tuples. Therefore, one should avoid using complex sub-networks as superboxes because other overlapping application trees might stall, waiting for execution which never happens.

- Advantage: easy to generate
- Disadvantage: superboxes have limited sizes especially in large and complex query networks.

6.3.6.1.2 Dynamic Heuristics

Dynamic scheduling techniques will assign priorities to boxes. From these priorities superboxes must be assembled. The goal for assembling these superboxes is to prepare trains for scheduling. Determining which boxes to include in a superbox is not straight-forward.

The following lists several examples of choices for assembling superboxes, given a prioritized list of boxes.

1. **Tree rooted at an application:** This technique is called *Top-k-spanner*. This algorithm creates, at run-time, a tree that spans the *top k* highest priority boxes for a given application. Note that this algorithm is equivalent to an application tree when *k* is equivalent to the number of boxes in the application tree.

The problem with this approach is that some boxes downstream will have multiple inputs (e.g. Join or Merge). And, those boxes will require the other input queues to contain a sufficient number of tuples to be executed. Those tuples may have to come from their respective upstream boxes. Figure 6.6 shows this more clearly. If B1 is the

box which has been identified as having a high priority, the downstream boxes are B2, B5 and B6. However, B5 has two inputs (from B2 and B4). The superbox would be most effective if all of the tuples from B1 were processed through to the application. If the inputs to B5 depend on each other (in the case of Join or Merge) there should be a balance in the input queues. Therefore, an upstream traversal through B4 is required.

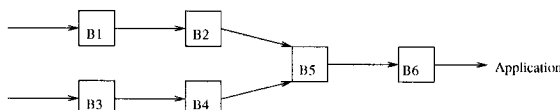


Figure 6.6: Example Aurora SubNetwork

In order for superbox scheduling to be effective, all boxes in the superbox should execute. Any imbalance in the inputs to a multiple-input box could cause the execution of a superbox to stall and result in not all boxes in the superbox being executed.

A simple solution to this problem is to require that superboxes only be composed of boxes with single inputs (except for the first box in the superbox). (i.e. a path of boxes is created towards the application until a multiple input box is found). Of course, the problem here is that the tuples which were identified as critical to the application do not make it to the output.

2. **Center of mass:** Given a box, create a superbox with the closest high priority boxes. Starting at a box, perform a breadth-first search of the network in both the upstream and downstream direction to a certain depth. For each box of a certain priority level, in the first search, perform a new breadth-first search. Continue until a certain number of boxes have been added to the superbox.

This method is application independent. It also tries to attack masses of congestion within an Aurora network. Unfortunately, this approach has two problems. First, it does not necessarily push tuples towards or all the way through to outputs. Second, a lot of overhead might be incurred dynamically creating these superboxes.

3. **Tree rooted at a box:** Given a box, a superbox is assembled with an upstream post-order traversal of the tree rooted at the box containing source-streams (inputs) as leaves.

There is a problem with this approach. The box (root) was identified to have a high priority because its queue has tuples which are nearing a critical point (the tuples are getting old). This approach will only move those tuples one box closer to the application. Subsequent scheduling decisions would be responsible for moving tuples closer to outputs.

Because of the problems associated with items 2 and 3 above, one should not consider these approaches seriously.

6.3.7 Dealing with high cost boxes

High cost boxes can cause the stalling of execution of superboxes. An interesting question is how to determine what constitutes a "high cost" box. The question becomes, how can you determine if a box will cause a superbox to stall? For the purpose of simulating query networks, we will consider two types of high-cost boxes. The first type is boxes which consume the CPU while executing. The second type is boxes which require a long time to execute but may yield the CPU while processing (e.g. SQL boxes which go to disk to find an answer).

After high-cost boxes have been identified, one of two approaches can be used to deal with this problem.

- Superboxes should not include high cost boxes as interior nodes. This will allow other threads to execute over the output of these high cost boxes while the upstream superbox churns away. This would be especially beneficial for non-CPU intensive high-cost boxes. For example, an SQL box might spend much of its time going to disk to retrieve results. As the results are being produced, superboxes executing downstream boxes "eat" the output as it is produced.

- Very high cost boxes can become superboxes on their own (i.e. one box superbox).

Yielding the CPU becomes an important issue for high cost boxes. The example in section 6.3.1 in Figure 6.5 provides an argument for this problem. As shown in that example, a high cost operator can "hog" the CPU and cause average tuple latency to be high. Therefore, high-cost boxes must yield the CPU to lower cost downstream boxes. Of course, this may have an impact on the various traversal techniques being used.

6.3.8 Superbox Priority Assignment

We need a better way to assign priority to superboxes. Currently, we select the superbox with the highest priority single box. This could be changed to be the application with the highest priority n boxes. However, this does not seem to be either efficient, or, the correct way to do things. Perhaps a feedback method could be used at the application or superbox level. If we use *Intersections of Application Trees*, we could focus on feedback from the root node of each superbox (assuming that a QoS graph could be created which represents the QoS for such a superbox).

A simple approach might turn out to be very effective. QoS graphs could be determined for an entire superbox and an approximation for average tuple latency for the superbox could be used to determine superbox priority. This is a very low-overhead approach which approximates the correct priority assignment.

Bibliography

- [ACC⁺03] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, , and S. Zdonik. Aurora: A new model and architecture for data management. In *VLDB Journal*, 2003.
- [AGM92] Robert K. Abbott and Héctor García-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.
- [AH00] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data.*, Dallas, TX, 2000.
- [BBDM03] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in stream systems. In *In Proc. of the International SIGMOD Conference, San Diego, CA*, 2003.
- [BR02] L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the web. In *VLDB*, 2002.
- [Bro88] R. Brown. Calendar queues: A fast $o(1)$ priority queue implementation of the simulation event set problem. In *Communications of the ACM*, 31(10):1220–1227., 1988.
- [BW01] S. Babu and J. Widom. Continuous queries over data streams. In *SIGMOD Record*, 30(3):109–120, 2001.

- [CCC⁺02] 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 *In proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02), Hong Kong, China.*, 2002.
- [CCR⁺03] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *In proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03), Berlin, Germany.*, 2003.
- [CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In Weidong Chen, Jeffery Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data: May 16–18, 2000, Dallas, Texas*, volume 29(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 379–390, New York, NY 10036, USA, 2000. ACM Press.
- [CFZ01] M. Cherniack, M. Franklin, and S. Zdonik. Expressing user profiles for data recharging. *IEEE Personnal Communication Magazine : Special Issue on Pervasive Computing*, August 2001.
- [CGM00a] J. Cho and H. Garcia-Molina. Estimating frequency of change. Technical report, Database Group, Stanford University, November 2000.
- [CGM00b] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the ACM SIGMOD*, pages 117–128, May 2000.
- [CLZ02] D. Carney, S. Lee, and S. Zdonik. Profile-driven data freshening. Technical report, CS Dept., Brown University, 2002.
- [CLZ03] Donald Carney, Sangdon Lee, and Stanley B. Zdonik. Scalable application-aware data freshening. In *ICDE*, pages 481–492, 2003.

- [CN02] Junghoo Cho and Alexandros Ntoulas. Effective change detection using sampling. In *VLDB*, 2002.
- [DP96] Adam Dingle and Thomas Partl. Web cache coherence. In *Proceedings of the 5th International WWW Conference*, May 1996.
- [HCL93] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Value-based scheduling in real-time database systems. *VLDB Journal: Very Large Data Bases*, 2(2):117–152, April 1993. Electronic edition.
- [Her86] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems (TOCS)*, 4(1), February 1986.
- [HFC⁺00] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. In *IEEE Data Engineering Bulletin*, 23(2):7-18,, 2000.
- [HLC91] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In Robert Werner, editor, *Proceedings of the Real-Time Systems Symposium - 1991*, pages 232–243, San Antonio, Texas, USA, December 1991. IEEE Computer Society Press.
- [IMS98] IMSL, Inc. *IMSL C Numerical Libraries, Version 3.01*. Visual Numerics, 1998.
- [JRR97] M. B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Symposium on Operating Systems Principles*,, 1997.
- [KA00] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3), September 2000.

- [Kar84] N. K. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.
- [KGM94] B. Kao and H. Garcia-Molina. An overview of realtime database systems. In *Real Time Computing*, A. D. Stoyenko, Ed.: Springer-Verlag,, 1994.
- [KR01] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, May 2001.
- [LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
- [Loc88] C. D. Locke. Best-effort decision making for real-time scheduling,. In *CMU TR-88-33*,, 1988.
- [LR01] Alexandros Labrinidis and Nick Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *VLDB*, pages 391–400, 2001.
- [LW82] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2:237–250, 1982.
- [NL97] J. Nieh and M. S. Lam. The design, implementation and evaluation of smart: A scheduler for multimedia applications. In *In Proc. 16th ACM Symposium on OS Principles*,, 1997.
- [OLW01] C. Olston, B. Thau Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD*, pages 355–366, May 2001.
- [OS95] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. In *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4):513-532,, 1995.

- [OW00] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the VLDB*, pages 144–155, September 2000.
- [OW02] Chris Olston and Jennifer Widom. Best-effort cache synchronization with source cooperation. In *Proceedings of the ACM SIGMOD*, 2002.
- [PQ00] V. Padmanabhan and L. Qui. The content and access dynamics of a busy web site: findings and implicatins. In *SIGCOMM*, pages 111–123. ACM, 2000.
- [Ram93] Krithi Ramamritham. Real-time databases. *International journal of Distributed and Parallel Databases (Invited Paper)*, 1993.
- [RS94] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems, 1994.
- [Son88] S. H. Son. Replicated data management in distributed database systems. *ACM SIGMOD Record*, 17(4), November 1988.
- [UF01] Tolga Urhan and Michael J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), Rome, Italy,*, 2001.
- [VN02] Stratis D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In Michael Franklin and Bonki Moon and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD international conference on Management of data (SIGMOD-02)*, pages 37–48, New York, June 3–6 2002. ACM Press.
- [Wan99] J. Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 25(9):36–46, October 1999.

- [WW95] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical Report MIT/LCS/TM-528, 1995.
- [ZGMHW95] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD*, pages 316–327, May 1995.