

Abstract of “Load Shedding Techniques for Data Stream Management Systems” by Emine Nesime Tatbul, Ph.D., Brown University, May 2007.

In recent years, we have witnessed the emergence of a new class of applications that must deal with large volumes of streaming data. Examples include financial data analysis on feeds of stock tickers, sensor-based environmental monitoring, and network traffic monitoring. Traditional database management systems (DBMS) which are very good at managing large volumes of stored data, fall short in serving this new class of applications, which require low-latency processing on live data from push-based sources. Aurora is a data stream management system (DSMS) that has been developed to meet these needs.

A DSMS such as Aurora may be subject to higher input rates than its resources can handle. When input rates exceed system capacity, the system will become overloaded and Quality of Service (QoS) at system outputs will fall below acceptable levels. Under these conditions, the system will shed load by selectively dropping tuples, thus degrading the answer, in order to improve the observed latency of the results.

In this dissertation, we first define the load shedding problem in data stream management systems and provide a general solution framework which handles the overload problem in a light-weight manner, while minimizing the loss in result accuracy. Then we present additional techniques on top of this basic framework to handle windowed aggregation queries in a way that preserves the subset result guarantee.

Due to the distributed nature of stream-based data sources as well as the need for better scalability and fault tolerance, we have recently extended Aurora into Borealis - a larger-scale system that can operate in distributed environments. In such an environment, the load shedding problem involves simultaneously removing excess load from multiple overloaded nodes in a scalable fashion. In the final part of this thesis, we investigate this distributed load shedding problem, and provide several alternative solutions to extend our earlier framework in Aurora to the distributed setting of the Borealis system.

Load Shedding Techniques for Data Stream Management Systems

by

Emine Nesime Tatbul

B. S., Middle East Technical University, Ankara, Turkey, 1996

Sc. M., Middle East Technical University, Ankara, Turkey, 1998

Sc. M., Brown University, Providence, RI, USA, 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 2007

© Copyright 2003, 2005, 2006, 2007 by Emine Nesime Tatbul

This dissertation by Emine Nesime Tatbul 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 _____
Stan Zdonik, Director

Recommended to the Graduate Council

Date _____
Uğur Çetintemel, Reader

Date _____
Mitch Cherniack, Reader
Brandeis University

Date _____
John Jannotti, Reader

Approved by the Graduate Council

Date _____
Sheila Bonde
Dean of the Graduate School

Vita

Emine Nesime Tatbul was born on December 15th, 1974 in Devrek, Turkey. She completed İstiklâl Elementary School in Devrek and TED Zonguldak College Foundation Private High School in Zonguldak. She attended the Middle East Technical University in Ankara, where she received her Bachelor of Science degree in 1996, and her Master of Science degree in 1998, both in Computer Engineering. She then joined the Ph.D. program at Brown University Computer Science Department in the fall of 1999, where she received a Masters degree in 2001. During her graduate school years at Brown, she also worked as a research intern at the IBM Almaden Research Center, and as a consultant for the U.S. Army Research Institute of Environmental Medicine.

Acknowledgements

This thesis work could not have been completed without the help and support of my colleagues, friends, and family.

First and foremost, I am in deep gratitude to my Ph.D. supervisor Stan Zdonik. Stan has been the perfect advisor. Starting from the first day I met him, he has been a constant source of moral support and encouragement. He has done so much to help me establish confidence in my research work. There have been times when I felt like he believed in me even more than I believed in myself. He provided invaluable resources and opportunities for me to accumulate academic experience in various areas including writing research papers and grant proposals, giving clear and convincing presentations, advising junior students, organizing conferences, and technical consulting to external institutions. Stan, I will forever be grateful for the time and effort that you invested in me. I hope I can be worthy of your unending trust.

I would like to also thank Uğur Çetintemel, whose role in my graduate school years has gone far beyond being a member of my thesis committee. The first and the most influential thing I learned from Uğur was to keep trying your very best, even when things seem to be hopeless, or impossible at times. In addition to being a great source of motivation and energy, Uğur has had significant technical contributions in my thesis research. This dissertation would not have been the same without him. I have also been in close interaction with Uğur on planning my career path. He not only shared his extensive experience with me, but also served and will continue to serve as an excellent role model.

When I joined Brown, Mitch Cherniack had just finished his Ph.D. work with Stan. I have always seen Mitch as my older academic brother. In my initial years at Brown, Stan and I would drive to Brandeis University to have research meetings with Mitch. We had a lot of fun in those meetings and I learned so much. In particular, observing Mitch come up with elegant ways to model and express his research ideas, I realized the importance of having good taste in research. Mitch has also served in my research comps and Ph.D. thesis committees, providing me great feedback. I am also thankful for his support during my job search.

I would like to thank John Jannotti who served in my Ph.D. thesis committee. Being from a closely related but a different subfield of computer systems, John provided the essential networking perspective that was needed on my research work. He has always raised interesting questions and given complementary feedback. His detailed comments on my thesis proposal document was very

helpful to improve the clarity of my writing.

The Aurora/Borealis projects, which form the basis of this dissertation work, involved tremendous group effort. I would like to thank all the team members from Brandeis University, Brown University and MIT: Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Bradley Berg, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Christina Erwin, Eddie Galvez, Matt Hatoun, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Sangdon Lee, Wolfgang Lindner, Sam Madden, Anurag Maskey, Olga Papaemmanouil, Alex Rasin, Esther Ryvkina, Jon Salz, Adam Singer, Mike Stonebraker, Richard Tibbetts, Robin Yan, Wenjuan Xing, Ying Xing, and Stan Zdonik. It was a great opportunity for me to work with Mike Stonebraker. I learned a lot from his extensive experience in designing and building large-scale database systems. Christian Convey, who worked as a software engineer for the Aurora project, deserves special thanks for his dedicated support in preparing our system demo for the SIGMOD Conference [7]. Mark Humphrey, Jeong-Hyon Hwang, Robin Yan, Wenjuan Xing worked on the graphical interfaces of the projects, which made our demos look much more sensible and nicer.

I want to also thank the members of the Brown Database Group. It was great fun working with you folks. We shared the stress of the conference deadlines, and the fun of partying after work, celebrating our success. In particular, I will always remember the sleepless nights that we spent together, debugging the Borealis code to get it working for the SIGMOD demo [12]. Yanif, Jeong-Hyon, Mark, Olga, Alex, Vivian, Ying: We deserved the best demo award after all, didn't we?

I am also thankful to the administrative and technical staffs of the Brown Computer Science Department. They were always available to answer questions and solve problems. I would like to especially acknowledge the help from Lori Agresti, Katrina Avery, Eugenia deGouveia, Jennet Kirschenbaum, Dorinda Moulton, Max Salvas, and Jeff Coady.

I had the opportunity to spend the summer of 2002 at the IBM Almaden Research Center as a research intern. Although the work I did at Almaden did not become a part of this dissertation, I learned various skills during my internship which I believe made me a better researcher in general. I would like to thank my project manager C. Mohan and my mentor Mehmet Altınel for providing me the opportunity. Special thanks are also due to my long-time friend Fatma Özcan from IBM Almaden, whose career steps I closely followed over the years. Fatma has given me great advice and support at various periods of my graduate school life from research-related issues to career planning.

I will always be grateful to Asuman Doğaç, my Masters advisor at the Middle East Technical University in Turkey. She first introduced me to the database field and she encouraged me to pursue my Ph.D. study at Brown in the first place. I will continue to admire and get motivation from her enthusiasm, hard work, and energy for database research.

I thank all my friends at Brown Computer Science for their companionship and support. Çağatay Demiralp was one of the first people that I met when I came to Providence. With Çağatay, we have shared many interesting conversations in research, politics, and miscellaneous gossip over coffee. I joined Brown at the same time with Olga Karpenko and Prabhat. We have been very good friends

ever since. Thanks to you folks for your help and support in everything. I would like to also thank my colleague, exercise buddy, hotel roommate, and chicken-thrower Olga Papaemmanouil, with whom we both worked and shared laughs together on various occasions.

I also thank all my friends in the Brown Turkish community for their friendship, help, and support over the years. They always made me feel at home.

I also send my thanks to my close friends in Turkey including Sibel Aktaş-Şen, Nuray Aydın, Özlem Demir-Kaya, and Burçak Ekinci for giving me continuous moral support and for cheering me up from 5000 miles away.

My family has always been a wonderful source of support and encouragement. I would like to thank my parents Emin Tatbul and Şaziye Tatbul, and my brother Mustafa Tatbul for their unconditional love and support. They have always had everlasting faith in me which gave me strength to overcome the obstacles that I encountered during my Ph.D. years. Thanks are due to my parents-in-law Müberra and Ferhat Bitim as well as my brother-in-law Semih Bitim for their continuous support and prayers.

Finally, I would like to express my deepest gratitude to my husband Melih Bitim. Melih is responsible for initially motivating me to pursue an academic career in the United States. I wouldn't be where I am today if he didn't encourage me in the first place. He had to endure the ups and downs of my long graduate school life. There have been many times that he woke up in the middle of the night to pick me up from school. He had to spend many boring Sundays without me. Despite all this suffering, he never stopped supporting me and believing in that I will become a doctor one day. Melih, I thank you for your love and support, and I dedicate this dissertation to you.

Credits

This thesis is based on several papers that we have jointly written with Stan Zdonik, Uğur Çetintemel, Mitch Cherniack, Michael Stonebraker, and some other members of the Aurora/Borealis Projects. Chapter 2 is based on our system design papers for Aurora [8] and Borealis [6]. Chapter 3 is based on our VLDB 2003 paper [93]. Chapter 4 is based on our VLDB 2006 paper [95]. Finally, Chapter 5 is based on our NetDB 2006 paper [94] and a recent conference submission [92].

This thesis research has been supported by several grants, including NSF grants IIS-0086057 and IIS-0325838, and Army contract DAMD-17-02-2-0048, for which we are grateful.

*Uzun ince bir yoldayım.
Gidiyorum gündüz gece.
Yetişmek için menzile.
Gidiyorum gündüz gece,
Gündüz gece.*

*I am on a long, narrow path.
I am walking on, day and night.
On the road to reach my destination.
I am walking on, day and night,
Day and night.*

Âşık Veysel (1894-1973)

To my dear husband, Melih Bitim.

Contents

List of Tables	xxi
List of Figures	xxiii
1 Introduction	1
1.1 Data Stream Processing	1
1.2 The Overload Challenge	3
1.3 Thesis Contributions	4
1.4 Thesis Outline	6
2 Background	7
2.1 The Aurora System	7
2.1.1 System Architecture	7
2.1.2 Data Model	8
2.1.3 Query Model	9
2.1.4 Quality of Service Model	10
2.2 The Borealis System	11
2.2.1 System Architecture	11
2.2.2 Optimizer Hierarchy	12
3 Load Shedding in a Data Stream Manager	15
3.1 The Load Shedding Problem	15
3.2 General Solution Framework	17
3.3 Overload Detection	18
3.3.1 The Theory	19
3.3.2 The Practice	20
3.4 A Greedy Approach	21
3.4.1 Drop Locations	24
3.4.2 Loss/Gain Ratio	27
3.5 The Load Shedding Road Map	28
3.6 Drop Operators	30

3.7	Semantic Load Shedding	32
3.7.1	Translation between QoS Functions	32
3.7.2	The Drop Predicate	34
3.7.3	Semantic Drop on Joins	34
3.8	Simulation-based Performance Evaluation	35
3.8.1	Experimental Setup	35
3.8.2	Comparing Load Shedding to Simple Admission Control	36
3.8.3	Comparing Random and Semantic Load Shedding	38
3.8.4	Evaluating the Effect of Operator Sharing	39
3.9	Case Study: Battalion Monitoring	40
3.10	Chapter Summary	42
4	Window-aware Load Shedding	47
4.1	Overview	47
4.2	Aggregation Queries	48
4.2.1	The Window Model	49
4.2.2	The Aggregate Function	49
4.2.3	The Aggregate Operator	49
4.3	The Subset-based Approximation Model	50
4.4	Window-aware Load Shedding with Window Drop	51
4.5	Handling Multiple Aggregates	53
4.5.1	Pipeline Arrangement of Aggregates	53
4.5.2	Fan-out Arrangement of Aggregates	55
4.5.3	Composite Arrangements	56
4.6	Decoding Window Specifications	56
4.7	Early Drops	58
4.8	Window Drop Placement	59
4.9	Analysis	59
4.9.1	Correctness	59
4.9.2	Performance	60
4.10	Extensions	61
4.10.1	Multiple Groups	61
4.10.2	Count-based Windows	62
4.11	Performance Evaluation on Borealis	62
4.11.1	Experimental Setup	62
4.11.2	Basic Performance	63
4.11.3	Effect of Window Parameters	65
4.11.4	Processing Overhead	68
4.12	Chapter Summary	68

5	Distributed Load Shedding	71
5.1	Overview	71
5.1.1	Motivating Example	72
5.1.2	Design Goals	73
5.1.3	Our Solution Approach	74
5.1.4	Assumptions	74
5.1.5	Chapter Outline	75
5.2	The Distributed Load Shedding Problem	75
5.2.1	Basic Formulation	75
5.2.2	Operator Splits and Merges	76
5.3	Architectural Overview	78
5.4	Advance Planning with a Solver	80
5.4.1	Region-Quadtree-based Division and Indexing of the Input Rate Space	81
5.4.2	Exploiting Workload Information	83
5.5	Advance Planning with FIT	84
5.5.1	Feasible Input Table (FIT)	84
5.5.2	FIT Generation	85
5.5.3	FIT Merge and Propagation	88
5.5.4	Point-Quadtree-based Division and Indexing of the Input Rate Space	89
5.6	Putting it All Together	90
5.7	Performance Evaluation on Borealis	91
5.7.1	Experimental Setup	91
5.7.2	Experimental Results	91
5.8	Chapter Summary	99
6	Related Work	103
6.1	Computer Networking	103
6.2	Network Services	104
6.3	Multimedia Streaming	105
6.4	Real-Time Databases	106
6.5	Approximate Query Processing	107
6.6	Parametric Query Optimization	108
6.7	Load Shedding in Data Stream Management Systems	109
6.7.1	STREAM	109
6.7.2	TelegraphCQ	110
6.7.3	NiagaraCQ	111
6.7.4	The Cornell Knowledge Broker	111
6.7.5	Model- and Control-based Approaches	112

7	Conclusions and Future Work	113
7.1	Future Directions	114
7.1.1	Managing Other Resources	114
7.1.2	Other Forms of Load Reduction	114
7.1.3	Other Latency Problems	115
7.1.4	Window-awareness on Joins	115
7.1.5	Prediction-based Load Shedding	115
7.1.6	Load Management on Update Streams	116
	Bibliography	117

List of Tables

3.1	Drop operators (* indicates limited use)	31
3.2	Notation for translation between QoS functions	33
3.3	Example value intervals	33
3.4	Value-based QoS and other metadata for the join of Figure 3.10	35
3.5	Notation for the utility formulas	36
4.1	Window specification attribute	52
4.2	Rules for setting window drop parameters	53
4.3	Decoding window specifications	57
4.4	Throughput ratio (WinDrop($p = 0$)/NoDrop)	68
5.1	Alternate load shedding plans for node A of Figure 5.1	72
5.2	Linear query diagram notation	76
5.3	Four phases of distributed load shedding	81
5.4	FITs for example of Figure 5.1 (spread = 0.2)	85
5.5	Effect of dimensionality ($\epsilon_{max} = 10\%$)	99

List of Figures

1.1	DBMS vs. DSMS: The paradigm shift	2
2.1	Aurora system architecture	8
2.2	Aurora query network	9
2.3	Quality of Service (QoS)	10
2.4	Borealis system architecture	12
2.5	Borealis optimizer hierarchy	13
3.1	Calculating load coefficients	19
3.2	Query network with load coefficients	20
3.3	Candidate drop locations	24
3.4	Query plan with no fan-out	25
3.5	Query plan with fan-out	25
3.6	Drop insertion	27
3.7	Load Shedding Road Map (LSRM)	29
3.8	LSRM construction	30
3.9	Derivation of the loss-tolerance QoS	33
3.10	Join	35
3.11	Load shedding vs. admission control variants (% Tuple utility loss)	37
3.12	Load shedding vs. admission control variants (% Value utility loss)	38
3.13	Value utility loss ratio for Random-LS/Semantic-LS vs. skew in utility	39
3.14	Tuple utility loss ratio for Input-Uniform/Random-LS vs. % excess load	40
3.15	Battalion Monitoring	41
3.16	Battalion Monitoring Queries	43
3.17	Aurora Performance Monitoring GUI for Battalion Monitoring Queries	44
3.18	Load Shedding Results for the Battalion Monitoring Application	45
4.1	An example nested aggregation query	47
4.2	Drop alternatives for an aggregate	51
4.3	Pipeline example	54
4.4	Fan-out example	55

4.5	Inserting drops into an aggregation query	60
4.6	Drop-batch (when $\mathcal{B} \geq \lfloor \frac{w}{\delta} \rfloor$)	61
4.7	Drop insertion plans for the pipeline arrangement (RDrop, Nested1, and Nested2)	63
4.8	Comparing alternatives (pipeline)	64
4.9	Drop insertion plans for the fan-out arrangement	65
4.10	Comparing alternatives (fan-out)	65
4.11	Effect of window size	66
4.12	Effect of window slide	67
4.13	Filtered aggregation query	68
5.1	Two continuous queries distributed onto two servers	72
5.2	Linear query diagram	76
5.3	Two levels of operator splits	77
5.4	Merging two streams via Union	78
5.5	Centralized approach	79
5.6	Distributed approach	80
5.7	Region-Quadtree-based space division and index for Solver	83
5.8	Choosing feasible points	86
5.9	Splits to be supported by complementary plans	88
5.10	Point-Quadtree-based space division and index for FIT	89
5.11	Query networks with different query load distributions and feasibility boundaries	92
5.12	Effect of query load imbalance	93
5.13	Effect of workload distribution and provision level on Solver-W plan generation	95
5.14	Exponential workload distribution for different λ values	96
5.15	Effect of operator fan-out ($\epsilon_{max} = 1\%$)	97
5.16	Effect of input dimensionality	98
5.17	D-FIT Overhead	100

Chapter 1

Introduction

New applications that must deal with vast numbers of input streams are becoming more common. These include applications that process data from small embedded sensors, applications that must correlate financial data feeds, and applications that must manage input from a very large number of geo-positioning devices. A new class of data management systems has emerged in response to these applications [24, 28, 31, 70]. These systems are commonly known as Data Stream Management Systems (DSMS). A DSMS aims at providing the same kind of infrastructure to stream-based applications that Database Management Systems (DBMS) have provided for data processing applications.

In this chapter, we first introduce the data stream processing concept. We then focus on some of the important challenges of data stream processing that motivated this thesis work, and highlight our main contributions in addressing those challenges. Finally, we conclude the chapter with a brief outline of the thesis.

1.1 Data Stream Processing

Data streams are possibly unbounded sequences of data elements that are typically generated rapidly one after the other. Common sources of this kind of data include sensors that make physical measurements about their environments and emit these as value readings (e.g. a temperature sensor), and software that continuously report the occurrence of certain events of interest (e.g., a program reporting the trades of stock shares). An important property of these data sources is that they are *push-based*, i.e., they are not programmed to store and provide data on demand, but to release it as soon as new data becomes available. There are an increasing number of applications that require continuous, real-time monitoring and processing on data streams. Well-known examples include sensor-based monitoring (e.g. habitat monitoring [89], biomedical monitoring [82, 91], road traffic monitoring [15]), RFID-based asset tracking [42], financial analysis applications [105, 107], GPS-based location tracking [66], and network traffic monitoring [18].

Data streams have caused an important paradigm shift in data management. In traditional

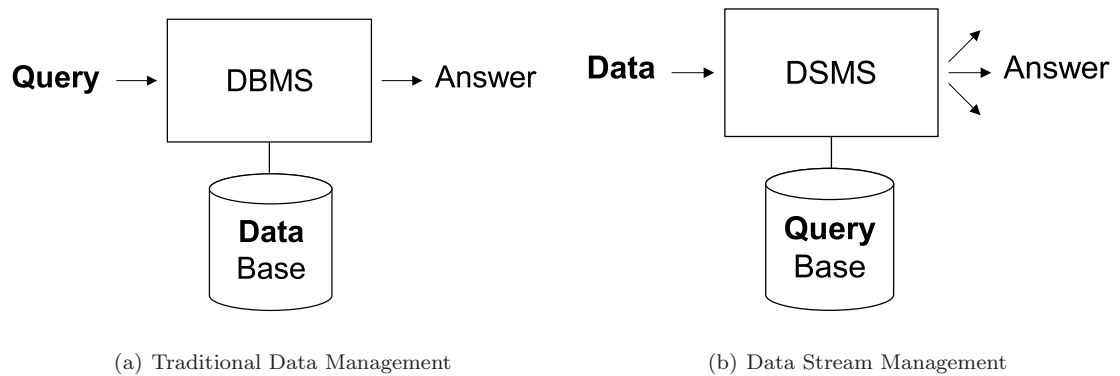


Figure 1.1: DBMS vs. DSMS: The paradigm shift

data processing systems, large volumes of datasets are persistently stored on disk, and one-time queries are executed on them (see Figure 1.1(a)). In this model, data is *pulled* from the disk as it is demanded by the queries. In case of data streams, the two basic elements of data processing, data and queries, have their roles completely reversed (see Figure 1.1(b)). A large number of long-lived queries called *continuous queries* are defined in advance of the data, and are persistently stored in the system. Data streams, as they arrive from their push-based sources, are continuously evaluated against these standing queries. In this model, data is *pushed* from the sources and needs to be processed through the queries¹. This radical reversal of roles for data and queries has not only required that the database community rethink most of the earlier data processing problems and their respective solutions, but also introduced a number of brand new challenges:

- **Continuous Queries.** Stream data typically arrives in certain order and may have no end. Queries have to be continuously executed, taking the order of data arrival into account. This is radically new since the traditional relational operations assume set-based (i.e., unordered) semantics on finite data. Furthermore, since data streams can be unbounded, some of these operations, which were designed to process their inputs as a whole may be blocked (e.g., aggregate operations) or may have to keep infinitely growing internal state (e.g., join operation). To address these issues, such operations need to be redefined to perform “window-based” continuous query processing.
- **Low-latency Processing.** Stream data usually represents real-time events and therefore has close association with the time of its generation. In most cases, data will quickly lose its value to the application as it gets older. Therefore, low-latency processing of data streams is highly important.
- **High and Variable Input Rates.** Stream data arrival rates can be very high. For example,

¹In some cases, data can also be optionally archived at the receiver for post-processing.

in the financial domain, applications have to deal with 100,000+ messages per second [3] (although not all of these messages may be equally important or interesting to the application). Furthermore, input load often fluctuates in an unpredictable fashion due to bursty nature of data arrival. This may further cause unexpected resource bottlenecks and performance degradation in the system. Therefore, continuous performance monitoring and adaptive resource management are essential for maintaining system scalability and good quality of service, including low latency.

The database community has worked on many problems in the past that closely relate to data stream processing. These include temporal and real-time databases focusing on representing time and satisfying timing constraints [73]; active databases with database events and trigger processing [75]; publish/subscribe systems focusing on push-based data processing [40]; approximate query processing techniques as a trade-off for improved performance [44]; and sequence databases where order of processing is important [83]. Data streams embody the challenges of all of these problem areas and more.

Consequently, data stream processing has emerged as a new subfield of research in database systems in the past few years². Researchers have investigated various aspects of data stream processing, including data models and query algebras, basic query processing and optimization algorithms, approximation and adaptivity for scalable performance, XML streams, and stream mining. Several large-scale research prototypes have been built, some of which have already successfully turned into promising start-up companies (e.g., StreamBase [87], Coral8 [2], and Amalgamated Insight [1]). This thesis research has been conducted in the context of two such prototype systems, Aurora and Borealis, which we will describe in detail in the next chapter.

1.2 The Overload Challenge

The high-level focus of this thesis is the resource overload problem in data stream management systems. As also mentioned earlier, streaming applications are characterized by a large number of push-based data sources in which the data arrival rates can be high and unpredictable. Each of these applications is responsible for monitoring data to detect critical situations, during which the data rates can significantly increase. During bursty data arrival, the demand on system resources (such as CPU) may exceed the available capacity. In this case, tuple queues will build up, thereby seriously increasing the latency of the query results. This is a major problem since low-latency processing is an important requirement of real-time stream processing.

Providing meaningful service even under system overload is one of the key challenges for stream processing systems. Since such overload situations are usually unforeseen and immediate attention is vital, adapting the system capacity to the increased load by adding more resources may not be

²To give an idea about the growing interest, starting in 2002, the two leading conferences in databases, SIGMOD and VLDB, have included increasingly more number of research sessions on data streams, reaching up to 4 sessions each in 2005 (roughly, 18% and 16% of the total research sessions, respectively) [38].

feasible or economically meaningful. In order to meet the low-latency requirements, there may be no alternative but to *shed* some of the load.

In general terms, we define *load shedding* as the process of dropping excess load from the system such that latency bounds are preserved. While dropping tuples will certainly reduce the processing requirements on the system, and thus, reduce the effective load, it will also have a detrimental effect on the accuracy of the answer. Said another way, load reduction to improve latency and accuracy are fundamentally at odds. When we improve utility by shedding load and reducing latency, we necessarily lose utility by producing an *approximate answer*. The technical challenge in this problem is to improve latency with minimal loss in answer accuracy.

Another important challenge with load shedding is to be able to produce approximate answers for queries with different levels of semantic complexity. As we mentioned earlier, in order to deal with ordered and unbounded data arrival, continuous queries may involve operators that operate on “windows” of tuples. These are also known as “stateful” operators. One such common operator is windowed aggregation. These operators can also have customizable components such as user-defined functions, and multiple of them can occur in the same query plan in a nested or shared fashion. Load shedding for such complex queries brings additional semantic challenges.

On the other hand, due to the distributed nature of stream-based data sources as well as the need for better scalability and fault tolerance, data stream management systems have recently been extended to operate in distributed environments. Although distributed stream processing systems are potentially equipped with more powerful computational resources where query workload can be carefully distributed onto multiple machines to avoid overload situations as much as possible, the overload problem can still arise since data is likely to arrive in unexpected bursts, and the system usually cannot be provisioned based on a bursty workload. In a distributed environment, the load shedding problem involves simultaneously removing excess load from a multitude of overloaded nodes. Note that presence of even a single overloaded node is sufficient to cause latency increase. Therefore, it is important to make sure that all nodes operate below their processing capacity. Furthermore, in a distributed setting, there is a load dependency between nodes that are assigned to run pieces of the same query. Shedding load at an upstream node affects the load levels at its downstream nodes, and the load shedding actions at all nodes along a query chain will collectively determine the quality degradation at the query end-points. It is essential that nodes shed load in a coordinated manner in order not only to relieve all of the overloaded nodes, but also to minimize the total quality loss at query end-points.

1.3 Thesis Contributions

This thesis defines the resource overload problem in data stream management systems and provides scalable and efficient load shedding techniques in order to solve it. In particular, we model load shedding as automatic insertion of load reducing drop operators into running query plans, where a drop operator essentially provides a simple and convenient abstraction for load reduction. We

provide a solution framework which addresses the following four key questions of load shedding:

- When load shedding is needed?
- Where in the query plan to insert drops?
- How much of the load should be shed at that point in the plan?
- Which tuples should be dropped?

Our load shedding framework is shaped by several important principles. First of all, any practical load shedding algorithm must be very light-weight, as the system is already under duress whenever load shedding is needed. To minimize run-time overhead, our approach relies on pre-computing and materializing load shedding plans in advance, which can then be efficiently used at run time whenever an overload is detected. Secondly, load shedding is essentially an approximate query answering technique which trades result accuracy for low-latency processing. It is important that the loss in accuracy is kept minimal. To achieve this, we try to balance two conflicting requirements:

- The earlier the load is reduced in a query plan, the larger is the savings in resources, and hence, the smaller is the needed data reduction.
- Shedding load early in a shared query plan may hurt the accuracy for multiple end-point applications.

To address this conflict in a way to minimize the total accuracy loss at the query end-points, we discovered that load reduction should be applied either on the input dataflows, or on dataflows that split onto multiple subqueries. Furthermore, we developed a metric called “loss/gain ratio”, which ranks these alternative dataflows. The load shedding plans are then generated based on the order determined by this ranking [93].

Our work further adopts a subset-based approximation model in which all of the delivered output tuples are guaranteed to be part of the original query answer, i.e. no incorrect values are generated. It is particularly challenging to provide this guarantee for windowed aggregation queries. Our window-aware load shedding approach achieves this by applying load reduction in units of windows [95].

The final piece of this thesis investigates the distributed load shedding problem [94]. We developed two alternative solution approaches to this problem:

- a centralized approach, where a coordinator node produces globally optimal plans, and the rest of the nodes adopt their share of these global plans;
- a distributed approach, where nodes exchange metadata information with their neighbors, and each node produces its own plan based on the available metadata.

All of these techniques have been implemented both as simulations and as part of the Aurora/Borealis prototype systems. As we will show, their performances have been extensively studied through theoretical analysis, simulation- and prototype-based experimentation, and through applying them on several realistic case studies.

1.4 Thesis Outline

This thesis is outlined as follows. In Chapter 2, we provide detailed overviews of the Aurora/Borealis systems which the research described in this thesis builds upon. Chapter 3 presents the core load shedding techniques that we developed for Aurora together with their performance evaluation results and a detailed look at a case study. We then extend this work in Chapter 4 to handle an important class of queries for data stream applications, namely, *windowed aggregation queries*, and present experimental results on Borealis. Chapter 5 presents our work on handling the load shedding problem in distributed stream processing environments such as that of the Borealis system. A detailed review of the related work is presented in Chapter 6. Finally, we conclude in Chapter 7, discussing several interesting directions for future research.

Chapter 2

Background

In this chapter, we present a comprehensive overview of two stream processing prototype systems, Aurora and Borealis¹. The research conducted in the scope of this thesis is an integral part of these two systems.

2.1 The Aurora System

Aurora is a data stream management system for processing continuous queries over data streams [7, 8, 20, 24]. It has been developed by a group of researchers from Brandeis University, Brown University, and MIT. The project started in early 2001, and in 2003 the built research prototype has been commercialized into a start-up company named StreamBase Systems [87]. In what follows, we provide an overview of the basic Aurora architecture and summarize the models underlying its design.

2.1.1 System Architecture

Figure 2.1 illustrates the basic system architecture of Aurora. We briefly describe the functionality of each component and the flow of data and control among them.

The *catalogs* store metadata information regarding the query network topology, QoS functions, and run-time statistics (e.g., selectivity and average processing cost for an operator). This information is fundamental for the operation of all run-time components, and hence, the catalogs are accessed by all components as required.

The *router* is responsible for forwarding data streams between run-time components. It receives input tuples from data sources as well as from box processors. If query processing on a stream tuple has been completed, the router outputs this tuple from Aurora to feed external applications.

¹“*Aurora* is a glow in a planet’s ionosphere caused by the interaction between the planet’s magnetic field and charged particles from the Sun. This phenomenon is known as the *Aurora Borealis* in the Earth’s northern hemisphere and the *Aurora Australis* in the Earth’s southern hemisphere.” [47].

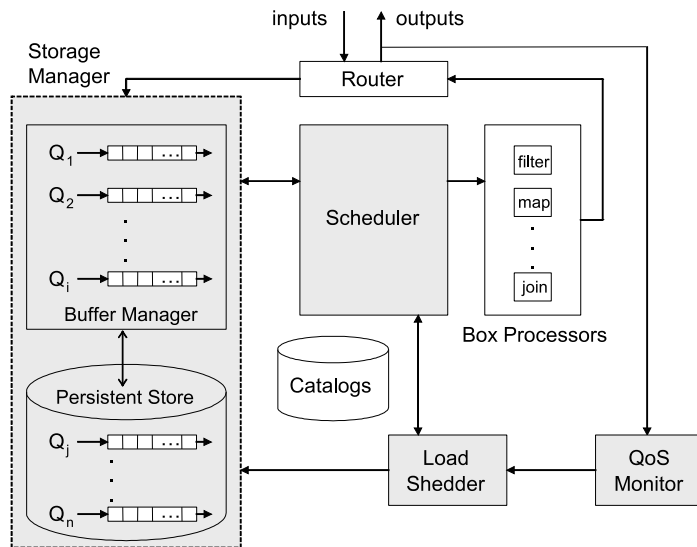


Figure 2.1: Aurora system architecture

Otherwise, the router forwards the tuple to the storage manager to be placed on proper queues for further processing.

The *storage manager* is responsible for efficient storage and retrieval of data queues on arcs between query operators. It manages the in-memory buffer pool that stores stream tuples for immediate use by box processors as well as the persistent store that keeps history for processing potential ad hoc queries.

The *scheduler* is the core component that makes decisions about operator execution order. It selects an operator with waiting tuples in its queues and executes that operator on one or more of the input tuples [25]. There is one processor per box type that implements the functionality for the corresponding query operator. When invoked by the scheduler, the *box processor* executes the appropriate operation and then forwards the output tuples to the router. The scheduler then ascertains the next processing step and the cycle repeats.

The *QoS monitor* continually monitors system performance and triggers the load shedder if it detects a decrease in QoS. The *load shedder* is responsible for handling overload due to high input rates [93]. It reads in system statistics and query network description from the catalogs, and makes certain modifications on the running query plans to bring the demand on CPU down to the available capacity level.

2.1.2 Data Model

Aurora models a stream as an append-only sequence of tuples with a uniform schema. In addition to application-specific data fields, each tuple in a stream also carries a header with system-assigned fields. These fields are hidden from the application and are used internally by the system for QoS

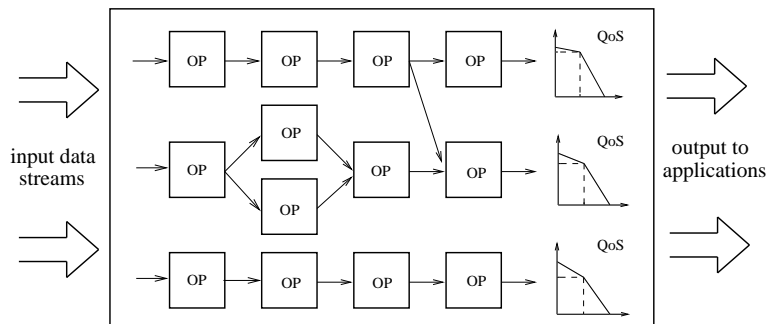


Figure 2.2: Aurora query network

management purposes. For example, every input tuple to Aurora is tagged with a timestamp upon system entry to indicate its arrival time, and every tuple generated by an Aurora operator is tagged with the timestamp of the oldest tuple that was used in generating it. This timestamp is further used to measure the processing latency of a tuple at any time point during its processing.

2.1.3 Query Model

Aurora queries are defined through a boxes-and-arrows-based dataflow diagram. Each box represents a query operator and each arc represents a data flow or a queue between the operators. Aurora is expected to process a large number of queries that are built out of a set of operators. Each such query may take an arbitrary number of input streams and always ends at a single output. An operator may be connected to multiple downstream operators. All such splits carry identical tuples and enable sharing of computation among different queries. Multiple streams can also be merged by the operators that accept more than one input. A *query network* is a collection of such queries. Figure 2.2 illustrates an Aurora query network.

Queries are composed using the operators defined by the Aurora Stream Query Algebra (SQuAl) [8]. SQuAl has nine primitive operators: Filter, Map, Union, Aggregate, Join, BSort, Resample, Read, and Update. *Filter* applies a predicate to tuples and retains those for which the predicate is true. *Map* applies a function to each stream element. *Union* merges two input streams into one. *Aggregate* applies a function on a *window* of tuples. *Join* correlates tuples from two streams that are within the same time band. *BSort* is an approximate sort operator that sorts tuples on an order attribute using a bounded-pass bubble sort algorithm. *Resample* is used to align pairs of streams by interpolating missing values. Finally, *Read* and *Update* are used to query and update database tables respectively, for each input tuple received.

Aurora also has a set of system-level drop operators. These operators are not used to build queries by the applications. Rather, the load shedder uses these operators to modify running query networks to deal with system overload. We will describe drop operators in Chapter 3. It is sufficient to note at this point that a drop operator is a load reducing operator that eliminates some fraction of its input.

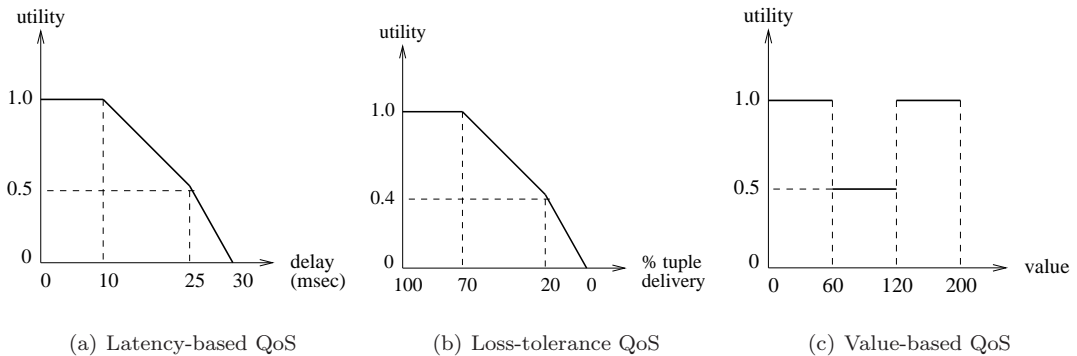


Figure 2.3: Quality of Service (QoS)

We further group our primitive operators into two as, *order-agnostic* and *order-sensitive*. Order-agnostic operators process input tuples one at a time and regardless of their tuple arrival order. Filter, Map, Union, Read, and Update have this property. On the other hand, Aggregate, Join, BSort, and Resample are *order-sensitive*. These operators require order on their input data streams and can only be guaranteed to execute with finite buffer space and in finite time if they can assume this ordering. Aurora does not restrict input streams to arrive in a specific order. Instead, order-sensitive operators are allowed to require order specifications on their inputs which may allow some bounded disorder [8]. Allowance for disorder complicates operators' basic functionality. In this thesis, we simply assume strict order specification for order-sensitive operators when order is relevant. We also consider query networks with a subset of SQuAl operators. More specifically, we do not consider BSort, Resample, Read, and Update. These operators, although they make SQuAl more expressive and may be useful for some specific applications, are not used as commonly as the other five operators.

2.1.4 Quality of Service Model

Most of Aurora optimizations are driven by a Quality of Service (QoS) model. Query results are sent to external applications for which a number of QoS functions are defined (see Figure 2.2). For each application, these functions relate a characteristic of the output to its utility (i.e., its usefulness to the receiving application).

QoS in Aurora is captured by three piece-wise linear functions, along three different dimensions (shown in Figure 2.3):

- Latency-based QoS: This function maps tuple latencies to utility values such that as tuples get delayed, their utility degrades. Latency of a tuple is defined as the time difference between when a tuple arrives at Aurora and when it is output from the system.
- Loss-tolerance QoS: This function maps the percent tuple delivery to a utility value and indicates how averse an application is to approximate answers. The larger the percentage of output

tuples delivered, the higher its utility to the receiving application. As an additional metric, we assume that each output application also specifies a threshold for its tolerance to gaps or “lack of responsiveness”. We call the maximum gap that an application can tolerate the *batch size*. The system must guarantee that the number of consecutive output tuples missed never exceeds this value. Note that batch size puts a lower bound on loss. Given a batch size \mathcal{B} , the query must at least deliver 1 tuple out of every $\mathcal{B} + 1$ tuples. Therefore, the percentage of tuples delivered should not be below $1/(\mathcal{B} + 1)$.

- Value-based QoS: The value-based QoS function shows which values of the output tuple space are most important. For example, in a medical application that monitors patient heartbeats, extreme values are certainly more interesting than normal ones, hence, corresponding value ranges must have higher utility.

Latency-based QoS function drives the scheduling policies, whereas loss-tolerance and value-based QoS functions drive the load shedding decisions in Aurora.

An important assumption we make about latency-based and loss-tolerance QoS functions is that they have concave shapes, i.e., the negative slope of the function is monotonically increasing.

2.2 The Borealis System

After Aurora was commercialized, its research prototype has evolved into Borealis. Borealis is a distributed stream processing engine which inherits its core stream processing functionality from Aurora [6]. Based on the needs of recently emerging, second-generation of stream processing applications, Borealis extends Aurora with the ability to:

- operate in a distributed fashion,
- dynamically modify various data and query properties without disrupting the system’s runtime operation,
- dynamically optimize processing to scale with changing load and resource availability in a heterogeneous environment,
- tolerate node and network failures for high availability.

In this section, we provide a brief overview of the Borealis System architecture, with a special focus on its optimizer structure.

2.2.1 System Architecture

Borealis accepts a collection of continuous queries, represents them as one large network of query operators (also known as a query diagram), and distributes the processing of these queries across multiple server nodes. Sensor networks can also participate in query processing behind a sensor proxy interface which acts as another Borealis node.

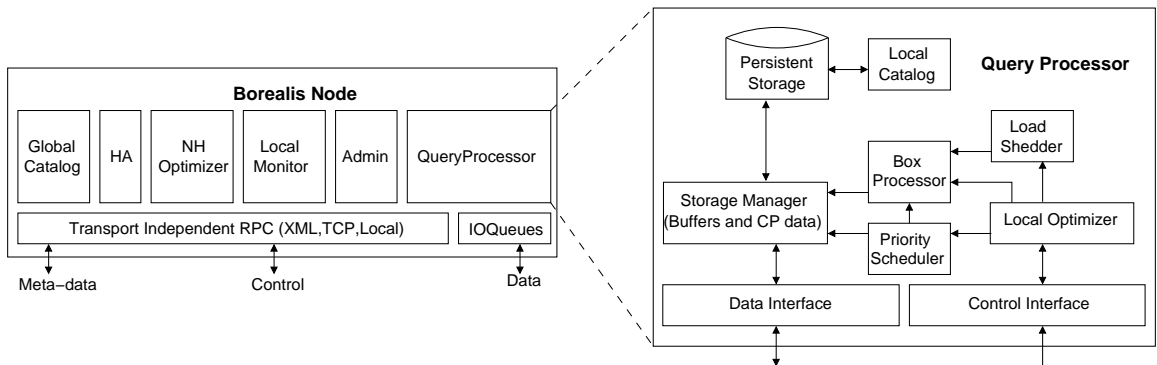


Figure 2.4: Borealis system architecture

Each node runs a Borealis server whose major components are shown in Figure 2.4. The *query processor (QP)* forms the essential piece where local query execution takes place. Most of the core QP functionality is provided by parts inherited from Aurora. *I/O queues* feed input streams into the QP and route tuples between remote Borealis nodes and clients.

The *admin* module is responsible for controlling the local QP, performing tasks such as setting up queries and migrating query diagram fragments. This module also coordinates with the *local optimizer* to provide performance improvements on a running diagram. The local optimizer employs various tactics including, changing local scheduling policies, modifying operator behavior on the fly via special control messages, and locally discarding low-utility tuples via load shedding when the node is overloaded.

The QP also contains the *storage manager*, which is responsible for storage and retrieval of data that flows through the arcs of the local query diagram, including memory buffers and *connection point (CP)* data views. Lastly, the *local catalog* stores query diagram description and metadata, and is accessible by all the local components.

Other than the QP, a Borealis node has modules which communicate with their respective peers on other Borealis nodes to take collaborative actions. The *neighborhood optimizer* uses local load information as well as information from other neighborhood optimizers to improve load balance between nodes. The *high availability (HA)* modules on different nodes monitor each other and take over processing for one another in case of failures. The *local monitor* collects performance-related statistics as the local system runs to report to local and neighborhood optimizer modules. The *global catalog* provides access to a single logical representation of the complete query diagram.

2.2.2 Optimizer Hierarchy

For scalable operation, Borealis uses a multi-tier optimizer structure with a hierarchy of monitors and optimizers, as seen in Figure 2.5.

Each Borealis node runs a local monitor that collects local statistics. These statistics include

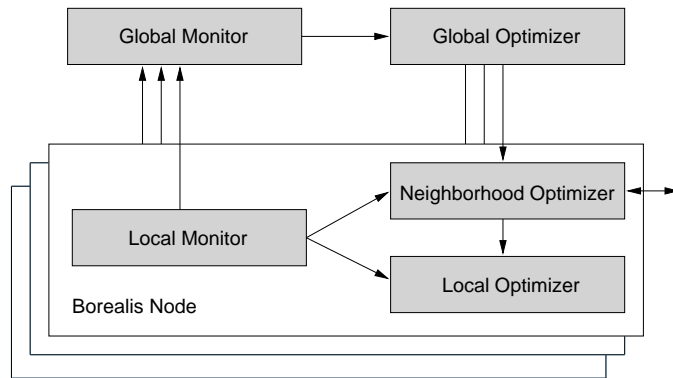


Figure 2.5: Borealis optimizer hierarchy

various operator- and node-level statistics regarding utilization and queuing delays for various resources such as CPU, bandwidth, and power (only relevant to sensor proxies). Local statistics are periodically forwarded to a global monitor.

Monitors trigger optimizers. There are three levels of collaborating optimizers: At the lowest level, the local optimizer runs at every node and is responsible for optimized execution of local query plans. In the middle tier, the neighborhood optimizer runs at every node and is responsible for communicating with immediate neighbors of this node for collaborative optimization. At the highest level, a global optimizer is responsible for making global optimization decisions on the whole query network, by identifying bottleneck nodes or neighborhoods based on global statistics provided by the global monitor.

Monitoring components run continuously and trigger optimizer(s) when they detect problems (e.g., resource overload) or optimization opportunities (e.g., neighbor with significantly lower load). The local monitor triggers the local optimizer or the neighborhood optimizer, while the global monitor triggers the global optimizer.

As we go up in the optimizer hierarchy, optimizers provide better and more effective solutions, but they also incur higher overhead, and can be more disruptive to system's operation. Hence, our general approach is to activate the optimizers in a bottom-up fashion. Each optimizer first tries to resolve the situation itself. If it can not achieve this within a pre-defined time period, monitors trigger the optimizer at the next level of the hierarchy. A higher level optimizer, when activated, reflects its decisions to lower level ones, instructing them to apply certain modifications on local query fragments. This approach strives to handle problems locally when possible because in general, local decisions are cheaper to make and realize, and are less disruptive. Another implication is that transient problems are dealt with locally, whereas more persistent problems potentially require global intervention.

Chapter 3

Load Shedding in a Data Stream Manager

In this chapter, we provide a detailed definition of the load shedding problem for data stream management systems. We present our solutions to various important subproblems of load shedding from overload detection to load shedding plan generation. Our two main forms of load shedding, random and semantic load shedding, are also introduced in this chapter. Finally, we present performance results for these approaches on an Aurora simulation as well as on a battlefield monitoring application.

3.1 The Load Shedding Problem

Data stream management systems operate on real-time data whose utility decreases as it gets older. Therefore, latency is an important performance factor. Furthermore, since data is being pushed from autonomous sources into the system, the system may not have direct control over the data arrival rates. Data rates can change in unpredictable ways, getting bursty at times. This requires that data processing has to keep up with data arrival; otherwise queues may build up, thereby seriously increasing the latency of the query results. In other words, the data arrival rates can get so high that the demand on CPU may exceed the available capacity. In this case, the CPU will be overloaded and will not be able to process input tuples as fast as they are received. Unless the overload problem is resolved, tuples will continue accumulating in queues, latencies will continuously grow, and latency-based QoS will degrade.

There are two high-level solutions to the CPU overload problem: (i) increase the CPU capacity to a level that is higher than the demand, or (ii) decrease the demand below the available CPU capacity. The former option may not be a feasible solution due to several reasons. First, the overload problem requires an immediate solution, therefore it may be slow and impractical to add new hardware. Second, getting new hardware may be expensive. Third, rates can change unpredictably, therefore

the problem may repeat itself even after the capacity is increased to solve the current problem. The last but not the least, bursts that we consider are usually short durations of very intense activity, and the CPU demand during bursts may be orders of magnitude larger than the CPU demand during regular workload. It may not be worth provisioning the system based on temporary overload scenarios, in which case the CPU will stay idle for most of the time. For these reasons, we explore the latter approach in this research.

To decrease demand on CPU below available capacity requires that we reduce the workload. We call this process of dropping excess load from the system *load shedding*¹. More precisely, load shedding is an optimization problem and can be formally stated as follows. We are given a query network N , a set of input streams I with certain data arrival rates, and a processing capacity C for the system that runs N . Let $N(I)$ indicate the network N operating on inputs I , and $Load(N(I))$ represent the load as a fraction of the total capacity C that network $N(I)$ presents. Load shedding is typically invoked when $Load(N(I)) > H \times C$. The constant H is the *headroom factor* that is a conservative estimate of the percentage of processing resources required by the system at steady state. The headroom is reserved to guard the system against thrashing. The problem is to find a new network N' that is derived from network N such that $Load(N'(I)) < H \times C$ and $Utility(N(I)) - Utility(N'(I))$ is minimized. *Utility* is the aggregate utility that is measured from the loss-tolerance QoS graphs of the application set. $Utility(N(I))$ represents the measured utility when there is no load shedding.

To put it another way, we model the load shedding problem as automatic modification of a running query network into a new network with reduced load. This modification can be in various different forms. In this thesis, we consider one specific type of modification where load reducing drop operators are inserted into the query plans. The load shedding problem can be broken into the following fundamental subproblems:

1. **Determining when to shed load.** The processing load of the query network needs to be continuously monitored. If there is overload, it should be detected quickly.
2. **Determining where to shed load.** Tuples can be dropped at any point in the processing network. Dropping them early avoids wasting work; however, because a stream can fan out to multiple streams, an early drop might adversely effect too many output applications.
3. **Determining how much load to shed.** Once we have determined where to insert a drop operator, we must decide the magnitude of that drop.
4. **Determining which tuples to shed.** Depending on the semantics of a particular drop operator used, we must figure out which tuples must be dropped.

¹Load shedding is a term that originally comes from electric power management, where it refers to the process of intentionally cutting off the electric current on certain lines when the demand for electricity exceeds the available supply to save the electric grid from collapsing. The same term has also been used in computer networking to refer to a certain form of congestion control approach, where a router simply drops packets when its buffers fill up.

It must be emphasized that any practical load shedding algorithm must be very light-weight. The system is by definition under duress whenever load shedding is needed. Therefore, an important part of the overload problem is to provide an efficient mechanism with minimal run-time overhead.

Note that CPU overload is one of many potential problems that can be caused by fast data arrival. High data rates can overload any of the system resources, including memory and bandwidth. For example, given a query network with a large number of stateful operators (e.g., aggregates, joins), memory overflow may become an issue. Similarly, if the system is running in a distributed environment, then the bandwidth between nodes may become insufficient. Each of these resource problems may manifest itself in a different way (e.g., CPU and bandwidth overload lead to latency increase, memory overflow may crash the system). However, they all fundamentally relate to increase in data arrival rates. Thus, although the load shedding approach we investigate in this thesis focuses on CPU as the primary scarce resource, it can be extended to handle other resource problems as well.

3.2 General Solution Framework

Our objective in this thesis is to develop a suite of load shedding techniques with the following principle features:

- **Light-weight overload handling.** The main objective of load shedding is to handle temporary bursts in input rates in a light-weight manner. Our techniques must react to overload immediately and they must be low-overhead not to make things worse in an already-overloaded system.
- **Minimizing QoS degradation.** An overloaded system experiences constant growth in its queues which consequently leads to ever increasing latencies at outputs. In a system where there is any excess load, sooner or later, latency utilities of all outputs will eventually drop to zero. The actual amount of overload only affects how fast this drop will take place. It is essential that load shedding controls queue growth in the system. To achieve this, it trades answer quality for improved latency. In this case, it must be guaranteed that the degradation in answer quality is minimal.
- **Delivering subset results.** Load shedding can approximate the query result in many different ways. Two alternative approximation models exist. One model produces approximate answers by omitting tuples from the correct answer. In this model, all delivered tuples are guaranteed to be a subset of the “exact” answer. In other words, we never produce incorrect answers. This property is important since it allows the application to rely on the tuples that it receives to be correct. An alternate approach to degrading the result is to emit nearly the same number of values, each of which might be inaccurate. The goal is to ensure that the errors are bounded by some amount. This model is reasonable as well. It is an application-level decision as to whether it is better to have all values, some of which may be inaccurate, or fewer values,

all of which are accurate. Our work is based on the subset model. The load shedder must guarantee to deliver results that are subsets of original query answers.

Our general operational model can be abstracted as follows. The load shedder operates in a loop, detecting important changes to the load status and reacting accordingly. Periodically, current load of the system is measured and compared against the known system capacity. If the load is above the available capacity, then load must be shed by inserting drops into the query network. On the other hand, if load is below the available capacity, then load can be returned by removing drops that might have been inserted in previous iterations of the loop. Hence, load shedding involves automatic insertion and removal of load reducing operators into query plans as required by the current load due to changing data arrival rates.

A load shedding approach must be accompanied by an underlying approximation model. In our case, the QoS model described in Section 2.1 drives load shedding. More specifically, load shedder uses two of the QoS functions, loss-tolerance and value-based QoS. Although the ultimate goal is to control output latencies, the latency-based QoS function need not be directly considered. This is due to an Aurora design decision which decouples the scheduler and the load shedder. We assume that any processor cycles that are recovered by the load shedder will be used sensibly by the scheduler to improve the overload situation thereby best improving the latency. We can, therefore, simply figure out how much capacity we need to recover and then produce a plan to do so. The scheduler will do the rest.

In the rest of this chapter, we first describe common techniques that apply to any load shedding algorithm, assuming a generic drop operator (which is reducing its input by a certain fraction), and abstracting other operators as operations with known costs and selectivities. Later, we will present additional mechanisms specifically required by each load shedding algorithm and for specific operator types.

3.3 Overload Detection

We provide a mathematical description of how to determine when a system is overloaded. This approach may have limited practical use in a real system due to a couple of reasons. First, it assumes the presence of reliable and stable statistics. Second, it assumes that a running query network is the only source of load in the system. This is not true even if Aurora were running on a dedicated machine with no other processes. Aurora run-time components themselves also have contribution to load. In fact, we use a headroom factor H to account for the portion of resources reserved for running the query network. However, it may sometimes be unrealistic to assume that there exists such a constant factor, or that it can be precisely measured. For these reasons, we also discuss several practical techniques that help overload detection.

3.3.1 The Theory

To detect an overload, we must first have a way to measure load. For this purpose, we use *load coefficients*. Each input to the query network has an associated load coefficient. A load coefficient represents the number of processor cycles required to process a single input tuple through the rest of the network to the outputs. Consider an input stream I running through a query of n operators and producing an output stream O as shown in Figure 3.1.

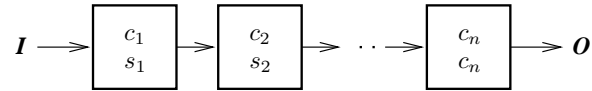


Figure 3.1: Calculating load coefficients

Assume that each operator i has an associated cost c_i (cycles per tuple) and selectivity s_i . We assume that such statistics can be gathered over some characteristic run period. The load coefficient for input I is computed as

$$L = \sum_{i=1}^n \left(\prod_{j=1}^{i-1} s_j \right) \times c_i \quad (3.1)$$

If an input has load coefficient L (in processor cycles per tuple) and input rate r (in tuples per time unit), the actual run-time load for that input is $L \times r$ (cycles per time unit). If there are m inputs, we can compute the total load as

$$\sum_{i=1}^m L_i \times r_i \quad (3.2)$$

The load coefficient formulation given in Equation 3.1 is based on the flat query shown in Figure 3.1. We now generalize this equation to query networks with fan-outs and n-ary operators (e.g., Union, Join). Again consider an input stream I , running through a query network N with p distinct *query paths*. We define a query path as a flat sequence of operators connecting I to any Aurora output. The load coefficient for input I in this case is

$$L = \sum_{i=1}^p \left(\sum_{j=1}^{n_i} \left(\prod_{k=1}^{j-1} s_{i,k} \right) \times c_{i,j} \right) \quad (3.3)$$

where n_i denotes the number of operators on path i , $s_{i,k}$ denotes the selectivity of operator k on path i , and $c_{i,j}$ denotes the cost of operator j on path i .

To illustrate, in Figure 3.2, we provide an example query network with two continuous queries and two input streams. Input I_1 has three distinct query paths and input I_2 has one query path. We present load coefficients for each of the input streams as well as load coefficients for each intermediate stream in the network. When there is fan-out, each different path's load is added to an input's load coefficient. When there is a binary operator, input rate flowing into the operator is a sum of both inputs (i.e., summed per query path).

One important point to note is that load coefficients can be computed statically when cost and selectivity statistics are known. Later, the actual run-time load can easily be calculated using these pre-computed coefficients. For example, if it turns out that input streams I_1 and I_2 have rates of 10 and 20 tuples per time unit, respectively, then the load of this network becomes $26.5 \times 10 + 18.75 \times 20 = 640$ cycles per time unit.

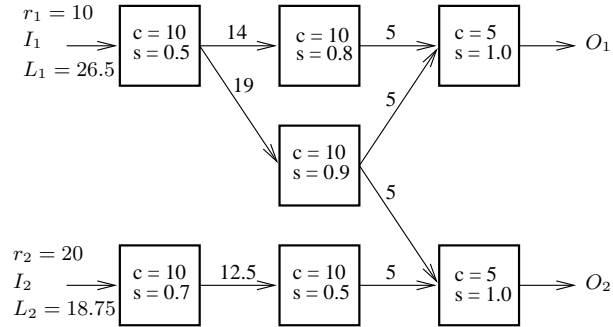


Figure 3.2: Query network with load coefficients

An overload is detected when the network load computed as above exceeds the allowed capacity of the system.

Note that the actual load of a query network is a combination of current input rates and any queues that may have built up since the last load evaluation step. With the headroom constant $H > 0$, queues will eventually disappear as long as the load shedder maintains the headroom space. Alternatively, load shedder may exploit the queue length information to speed up this process through *overshedding*. To achieve this, we compute load coefficients not only for input arcs, but for intermediate arcs as well, which represents the number of processor cycles required to push a single tuple from a queue on that arc to the outputs. We use this coefficient to calculate the contribution of queue on an arc to the total queue load Q . First, we define a system parameter called *MELT_RATE*. This parameter is a lower bound on how fast we want to melt (i.e., shrink) the queues. It represents the queue length reduction per time unit as a fraction of the current queue length. An arc i with a load coefficient L_i and queue of length q_i contributes to the total queue load by $MELT_RATE * L_i * q_i$. The queue load of all arcs are summed to calculate Q . Essentially, by having $Q > 0$, we cause an overshedding in the network to melt the queues. After all the queues are melted, the load shedder will discover that some of the drops are redundant and have to be removed (or their drop probabilities must be decreased). Note that under normal circumstances, dealing with queues is scheduler's responsibility. However, tuning the *MELT_RATE* parameter, we can let the load shedder speed up the process of shrinking the queues for a faster improvement on latency QoS.

3.3.2 The Practice

In practice, the first thing to examine in order to detect CPU overload is CPU utilization. If this quantity is measured to be below 100%, then there is no CPU overload. Otherwise, the system may

be running just at capacity or may be overloaded. Next, we could check two things: queue lengths on arcs of the Aurora query network or tuple latencies at Aurora outputs. In most cases, the latter is a direct reflection of the former.

If there is a monotonic increase in total length of queues on arcs, then CPU is overloaded. One special case to consider is multi-input operators that can potentially block, such as Join. Join may block if data on one of its input streams is not arriving as fast as the other. In this case, the queue on the fast arc may be growing even though there are enough CPU cycles, which further leads to a misjudgement that CPU is overloaded (due to a constant increase in total queue length in the system). To avoid this problem, we must further check that join operators have growing queues on both input arcs.

Monotonic increase in tuple latencies at outputs could also be an indication of CPU overload. However, this information might be misleading when there are blocking operators in the query network. For example, windowed aggregates only produce output tuples when they receive enough tuples to close a window. If data does not arrive fast enough, output tuples may end up having large latency values (remember that output tuple gets the smallest timestamp in the window). The problem is similar with Join. Therefore, queue lengths might be a more reliable indication of CPU overload than output latencies alone.

3.4 A Greedy Approach

We cast our basic problem of “which of the possible arcs should have drops placed on them and how much of the data should be dropped on each such arc” as one of the classical optimization problems, namely the *Fractional Knapsack Problem*. We first define this problem and discuss its optimal greedy solution. Then we define our own problem in terms of this problem and formulate a similar greedy solution. Finally, we provide a proof that this greedy algorithm indeed produces the optimal solution.

Definition 1 (Fractional Knapsack Problem). *Given n items, each with weight w_i and value v_i , and a knapsack that can hold items up to a total weight limit of W , find the most valuable mix of items which will fit in the knapsack. If a fraction x_i , $0 \leq x_i \leq 1$ of item i is placed into the knapsack then a value of $v_i * x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total value earned. In other words, we want to find fractions x_i , $0 \leq x_i \leq 1$, for each item i , $1 \leq i \leq n$, that will maximize $\sum v_i * x_i$ subject to the constraint that $\sum w_i * x_i \leq W$.*

The Fractional Knapsack Problem has an optimal greedy solution [34]. The idea is to take as much of the item with the maximum value per unit weight (i.e., v_i/w_i) as possible. If that item is fully in the knapsack and if there is still room available in the knapsack, then the item with the next largest value per unit weight is taken. This procedure continues until the knapsack is full.

Our load shedding problem can be modeled as a Fractional Knapsack Problem. The processing system with a fixed CPU capacity of C represents the knapsack of a fixed capacity. The query

network with a certain input load represents the materials we want to fit into the knapsack. We must make sure that the materials do not take up capacity more than C . If that happens, the materials with smallest unit value must be removed from the knapsack. Materials in our problem correspond to data flowing through the arcs of the query network. The unit value of a material represents the total QoS utility a unit (i.e., 1%) of data on the corresponding arc provides to the end-point application(s) in exchange for the total processing power needed before that data unit reaches to the end-point(s). Thus, the least valuable data unit is the one whose removal from the query network causes the least utility loss for the amount of processing power it returns back to the system.

Note that the original knapsack problem tries to fill in the knapsack until the capacity limit is hit whereas our problem tries to empty an overfull knapsack until it is no more full beyond the capacity limit. Although the two tasks are completely opposite of each other, the final objective is the same: to leave the knapsack filled with the highest total value.

We will now define our version of the Fractional Knapsack Problem, which we call the *Overloaded CPU Problem*; reformulate the original greedy algorithm for our version; and show that it provides the optimal solution as in the original case. For easy exposition, we assume that each loss tolerance QoS function defined at the query end-points is a linear function (i.e., piece-wise linear with a single piece). This assumption implies that we completely drop from one output before we start dropping from the next output ².

Definition 2 (Overloaded CPU Problem). *We are given a computer system with CPU capacity C that runs a query network N with a set of input streams, each arriving at a certain rate. There are n independent arcs (a.k.a. drop locations) on N where drop operators can be inserted. Each such operator i ($1 \leq i \leq n$) can have a drop fraction of x_i ($0 \leq x_i \leq 1$). Furthermore, each unit of data dropped by i causes a total output QoS utility loss of l_i , while providing a processing power gain of g_i . Find drop fractions $X = (x_1, \dots, x_n)$ that will minimize the total QoS utility loss $\sum l_i * x_i$ subject to the constraint that $\sum g_i * (1 - x_i) \leq C$.*

The greedy algorithm for the Overloaded CPU Problem can be stated as follows: Find loss/gain ratios (l_i/g_i) for each drop arc and sort them in ascending order. If the current system load is above C , then remove data from the drop arc with the smallest loss/gain ratio. If data on that arc is completely dropped and the system is still overloaded, then remove data from the drop arc with the next smallest loss/gain ratio. Continue removing data until the CPU load drops below the capacity.

Theorem 1. *The greedy algorithm that always selects to remove data on the drop arc with the smallest loss/gain ratio finds an optimal solution to the Overloaded CPU Problem.*

Proof. Assume the drop locations $\{1, \dots, n\}$ and that

$$\frac{l_1}{g_1} \leq \frac{l_2}{g_2} \leq \dots \leq \frac{l_n}{g_n}$$

²We remove this assumption later in Section 3.4.2.

Let $X = (x_1, \dots, x_n)$ be the solution computed by the greedy algorithm. If $x_i = 0$ for all i , then the solution is optimal (i.e., we do not drop any data since the query network is already operating below the CPU capacity). Note that there is no possibility that $x_i = 1$ for all i (assuming that $C > 0$) since the system should be able to process some fraction of the inputs to the extent allowed by its capacity C . Thus, some entries of X can be 1, some other entries can be 0, and there may be one entry with $x_i < 1$. Let j be the smallest value for which $x_j < 1$. According to the greedy algorithm, if $i < j$, then $x_i = 1$, and if $i > j$, then $x_i = 0$. Furthermore, $\sum_{i=1}^n (1 - x_i) * g_i = C$.

Let $Y = (y_1, \dots, y_n)$ be any feasible solution. Then

$$\begin{aligned} \sum_{i=1}^n (1 - y_i) * g_i &\leq C = \sum_{i=1}^n (1 - x_i) * g_i \\ \sum_{i=1}^n g_i - y_i * g_i &\leq \sum_{i=1}^n g_i - x_i * g_i \\ \sum_{i=1}^n (y_i - x_i) * g_i &\geq 0 \end{aligned}$$

Let $L(Z)$ denote the total QoS utility loss of a feasible solution Z .

$$L(Y) - L(X) = \sum_{i=1}^n (y_i - x_i) * l_i = \sum_{i=1}^n (y_i - x_i) * g_i * \frac{l_i}{g_i}$$

If we can show that $L(Y) - L(X) \geq 0$, then we prove that the greedy solution X minimizes the total QoS utility loss and therefore is the optimal solution.

As mentioned earlier, if $i < j$, then $x_i = 1$, and $y_i - x_i \leq 0$, and $\frac{l_i}{g_i} \leq \frac{l_j}{g_j}$, and therefore we have

$$(y_i - x_i) * \frac{l_i}{g_i} \geq (y_i - x_i) * \frac{l_j}{g_j}$$

If $i > j$, then $x_i = 0$, and $y_i - x_i \geq 0$, and $\frac{l_i}{g_i} \geq \frac{l_j}{g_j}$, and therefore we again have

$$(y_i - x_i) * \frac{l_i}{g_i} \geq (y_i - x_i) * \frac{l_j}{g_j}$$

If we plug this inequality into $L(Y) - L(X)$, we obtain

$$L(Y) - L(X) = \sum_{i=1}^n (y_i - x_i) * g_i * \frac{l_i}{g_i} \geq \sum_{i=1}^n (y_i - x_i) * g_i * \frac{l_j}{g_j} \geq \frac{l_j}{g_j} * \sum_{i=1}^n (y_i - x_i) * g_i \geq 0$$

Hence, any given feasible solution Y causes at least as much QoS utility loss as the solution X computed by the greedy algorithm. Therefore, X is an optimal solution. \square

Next we discuss drop locations and loss/gain ratios in detail.

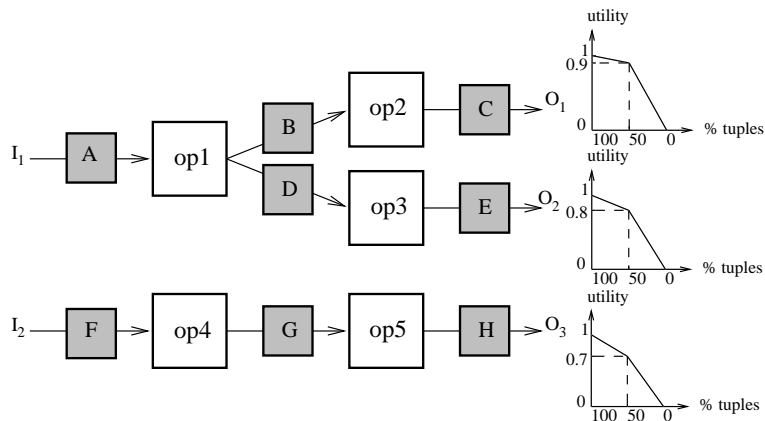


Figure 3.3: Candidate drop locations

3.4.1 Drop Locations

Technically, all arcs on the query network can have drop operators placed on them. However, to maximize the amount of processing power gained, some arcs are better candidates than others. In this section, we will show how we determine the set of arcs to be considered for drop insertion.

Consider the query network shown in Figure 3.3. This network consists of two input streams and three queries. Each query has a loss-tolerance QoS attached to its output. Smaller gray boxes marked with letters indicate candidate arcs for drops, which we call the *drop locations*. We will use this example query network to illustrate two important ideas.

First, consider a query plan that has no sharing with others. In a network that contains such a plan, a drop insertion at any location on that query does not affect any of the other queries. Hence, the utility loss is only observed at the output of that particular query plan. For example, in Figure 3.3, the bottom-most query has no sharing with the top two. In general, it is best to insert drops as early in the query plan as possible since it minimizes wasted work³. Therefore, for query plans with no sharing, the best location for a drop is always at its input. Hence, the small box marked with *F* is the best location for a drop in the bottom query.

Second, consider a query network with sharing. Sharing occurs when the output of an operator fans out to more than one downstream operator leading to different query outputs. This is observed at the output of *op1* in Figure 3.3. Any tuple coming out of *op1* would be routed to both outputs O_1 and O_2 . Hence, a drop operator inserted at *A* would affect both of these outputs. Inserting a drop at *B* or *D* affects only one output, and is thus a way to isolate the effect to a single output. If both *B* and *D* were assigned drops, then the drop should be placed at *A*, thereby saving the work of *op1*. Note that if *A* were preceded by a linear stretch of boxes, then by our previous reasoning, the drops should be pushed further upstream towards the inputs. As a result, output of operators with split points (*B*, *D*) and network inputs (*A*, *F*) are the only locations that need to be considered for drop insertion. We formally state these ideas with the following theorems.

³This is reminiscent of the standard selection-pushdown heuristic used in relational query optimization.

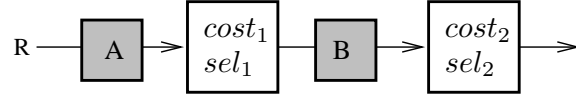


Figure 3.4: Query plan with no fan-out

Theorem 2. *For query plans with no fan-out, placing drops at inputs minimizes the output data loss.*

Proof. Consider the query plan in Figure 3.4. Assume that the system has an excess CPU load of Δ . For simplicity, assume also that drop operator has no CPU cost. If we place a drop at location A, then the magnitude of that drop (and hence, the fraction of data lost at the query output) must be:

$$p * R * (cost_1 + sel_1 * cost_2) = \Delta$$

$$p = \frac{\Delta}{R * (cost_1 + sel_1 * cost_2)}$$

If instead, we place a drop at location B, then the magnitude of that drop must be:

$$p' * R * sel_1 * cost_2 = \Delta$$

$$p' = \frac{\Delta}{R * sel_1 * cost_2}$$

It is easy to see that $p < p'$. Thus, placing the drop on the input arc minimizes the output data loss. \square

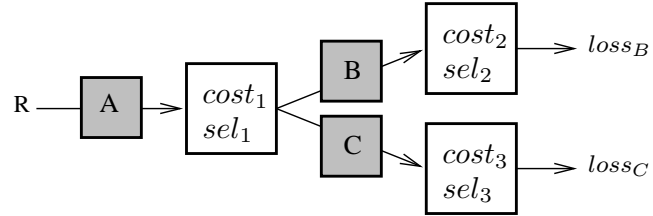


Figure 3.5: Query plan with fan-out

Theorem 3. *For query plans with fan-out, placing drops at inputs not necessarily minimizes the total output data loss.*

Proof. Consider the query plan in Figure 3.5. Without loss of generality, assume that the system has an excess CPU load of $R * sel_1 * cost_2$. If we place a drop at location A, then the magnitude of that drop must be:

$$p * R * (cost_1 + sel_1 * cost_2 + sel_1 * cost_3) = R * sel_1 * cost_2$$

$$p = \frac{sel_1 * cost_2}{cost_1 + sel_1 * cost_2 + sel_1 * cost_3}$$

In this case, both outputs of the query network will observe this fraction of data loss, adding up to a total output data loss of $2 * p = 2 * \frac{sel_1 * cost_2}{cost_1 + sel_1 * cost_2 + sel_1 * cost_3}$.

If instead, we place a drop at location B , then the magnitude of that drop must be:

$$p' * R * sel_1 * cost_2 = R * sel_1 * cost_2$$

$$p' = 1$$

In this case, only output of B will observe data loss. Drop location A provides smaller total loss if $2 * p < p'$. This holds only if:

$$2 * \frac{sel_1 * cost_2}{cost_1 + sel_1 * cost_2 + sel_1 * cost_3} < 1$$

$$\frac{sel_1 * cost_2}{cost_1 + sel_1 * cost_2 + sel_1 * cost_3} < \frac{1}{2}$$

$$sel_1 * cost_2 < cost_1 + sel_1 * cost_3$$

Only under this condition, drop location A would provide smaller total data loss than drop location B . Therefore, placing drops at inputs not necessarily minimizes the total output data loss. \square

Theorem 4. *Dropping upstream from a fan-out point has more value than dropping collectively on all fan-out arcs.*

Proof. Consider the query plan in Figure 3.5 where $loss_B$ denotes the unit QoS utility loss at the output of B and $loss_C$ denotes the unit QoS utility loss at the output of C . Assume further that dropping a unit fraction of data at A provides a processing power gain of $gain_A$, while dropping that same fraction at B and C provides gain of $gain_B$ and $gain_C$ respectively. We know that $gain_A > gain_B + gain_C$. Then

$$\frac{loss_B + loss_C}{gain_A} < \frac{loss_B + loss_C}{gain_B + gain_C}$$

$$\frac{loss_B + loss_C}{gain_A} < \frac{loss_B}{gain_B + gain_C} + \frac{loss_C}{gain_B + gain_C}$$

Since $\frac{loss_B}{gain_B + gain_C} < \frac{loss_B}{gain_B}$ and $\frac{loss_C}{gain_B + gain_C} < \frac{loss_C}{gain_C}$, we have

$$\frac{loss_B + loss_C}{gain_A} < \frac{loss_B}{gain_B} + \frac{loss_C}{gain_C}$$

In other words, the loss/gain ratio for A is smaller than the sum of loss/gain ratios of B and C . Therefore, it has more value to drop at A than to drop at both B and C . \square

3.4.2 Loss/Gain Ratio

As mentioned earlier, loss/gain ratios indicate the value of a data unit on a given arc of the query network. In this ratio, loss refers to the total loss-tolerance QoS utility loss at query outputs for each percentage of tuples dropped on the given arc; and gain refers to total processor cycles gained as a result of this drop. We will now show how we compute this ratio for each candidate drop location.

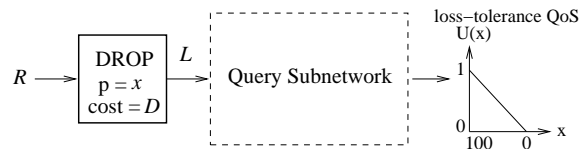


Figure 3.6: Drop insertion

Consider Figure 3.6, where a drop operator has been inserted upstream from a query subnetwork. R is the input rate flowing into the drop, x is the drop amount, D is the cost of the drop operator itself, and L is the load coefficient on the input arc of the downstream subnetwork. The gain from inserting this drop is:

$$G(x) = \begin{cases} R \times (x \times L - D) & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3.4)$$

Assume that the loss-tolerance QoS function of the affected output is given as $U(x)$. Then the loss/gain ratio can be computed as follows:

$$\frac{-dU(x)/dx}{dG(x)/dx} = \frac{\text{negative slope of } U(x)}{R \times L} \quad (3.5)$$

An important point to note is that since the order of the loss/gain ratios (not their absolute values) is all we need for our greedy algorithm, absolute values of input data rates are in fact not needed. Rather, it would be sufficient to know relative proportions of the input rates. Hence, if statistics on rate proportions are available, the gain values, and therefore the loss/gain ratios can be computed statically.

In our previous example shown in Figure 3.3, we identified a set of four candidate drop locations: $\{A, B, D, F\}$. The QoS graphs to be used for drop locations B , D , and F are the same as those of O_1 , O_2 , and O_3 , respectively. However, the QoS graph to be used for location A is the cumulative QoS graph from O_1 and O_2 . One way to obtain this cumulative graph is to sum the QoS functions of both of these outputs⁴. Furthermore, since loss is a function of the slope of the utility function, each drop location may have as many different loss/gain ratios as there are different function pieces in its (cumulative) QoS graph. For example, the percent loss for location F is $(1 - 0.7)/50$ for the first piece of the QoS function, whereas it is $(0.7 - 0)/50$ for the second piece. Furthermore, it is

⁴Note that we are not considering the semantics of the operators in this analysis. Depending on the types of the operators we have in the query network, the cumulative QoS may have been obtained in different ways.

always guaranteed that loss/gain ratio increases as we move from 100% to 0% on a given graph. This is a result of our concaveness assumption mentioned earlier in Section 2.1.4.

3.5 The Load Shedding Road Map

For light-weight load adaptivity, our approach utilizes a tabular data structure called the Load Shedding Road Map (LSRM). LSRM materializes a sequence of drop insertion plans that can be pre-computed in advance of a system overload based on statistics on operator costs, selectivities, output value histograms and an estimation about relative proportions of input data rates. At run-time, when an overload is detected, we use the LSRM to find a proper plan for recovering the excess load by simply performing a table lookup and adapting the matching drop insertion plan. Materializing load shedding plans in advance using the LSRM can significantly reduce the run-time overhead of making load shedding decisions.

As illustrated in Figure 3.7, LSRM is a table in which each subsequent row represents a load shedding plan that sheds more load than its predecessor. A load shedding plan consists of an indication of the expected cycle savings (c_i indicating the cycle savings coefficient for input i), the network arcs to place the drop operators, the corresponding drop amounts, and the effect of the drops on QoS graphs (p_i indicating the QoS cursor for output i). More specifically, each LSRM entry is a triple of the form:

<Cycle Savings Coefficients, Drop Insertion Plan, Percent Delivery Cursors>

The Drop Insertion Plan (DIP) is a set of drops that will be inserted at specific points in the network. Cycle Savings Coefficients (CSC) is a list of input streams that will be affected by the plan along with their associated savings coefficients. CSCs are used to determine how many cycles will be saved along the path of each input stream if the corresponding DIP is adopted. Percent Delivery Cursors (PDC) is a list of cursors for the loss-tolerance QoS graphs, one for each output. They indicate where the system will be running (in terms of percent of tuple delivery) if the corresponding DIP is adopted.

At run-time, when an overload is detected, we simply search the LSRM to find a plan for recovering the required number of processor cycles and adapt the corresponding plan by updating the query network accordingly. As load levels change, the load shedder adaptively switches from one plan to another. As shown in Figure 3.7, we keep a cursor that indicates which row in the LSRM was used last. Later on, if additional load needs to be shed, the search on the LSRM can begin at the cursor. On the other hand, if the load is ever determined to be within the capacity bound and there are drops in the network (i.e., cursor is non-zero), then the search can be done in the reverse direction.

Figure 3.8 shows the steps followed in the construction of the LSRM. We first identify the candidate drop locations in a query network (as discussed in Section 3.4.1). Next, for each location, we compute a loss/gain ratio, indicating the unit QoS utility loss per CPU cycles gained if a drop

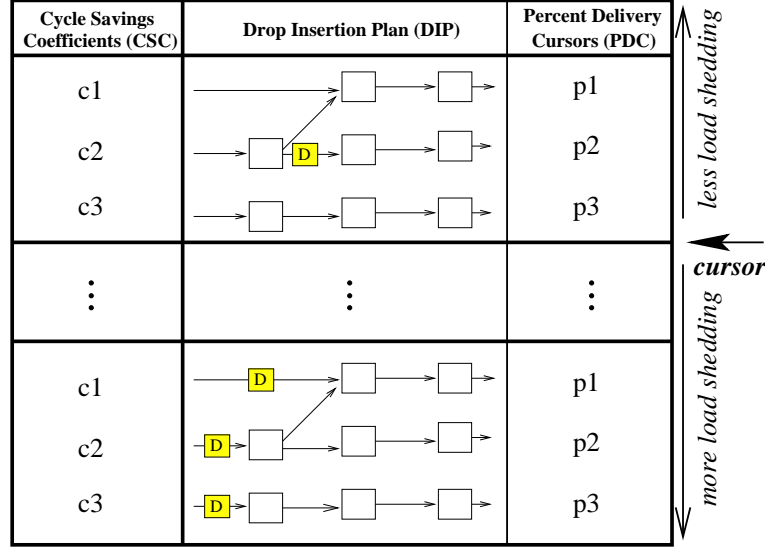


Figure 3.7: Load Shedding Road Map (LSRM)

were inserted at that location (as discussed in Section 3.4.2). These locations are sorted in increasing order of their ratios. Finally, drop insertion plans are generated by processing each location in this sort order. As discussed earlier in Section 3.4, this algorithm guarantees minimal QoS utility loss; i.e., for any overload level, drops are inserted such that utility loss for gained cycles in return is minimized.

We will now describe how we create the drop insertion plans in detail, i.e., the bottom loop at LSRM construction chart of Figure 3.8. We process the sorted drop locations one by one, starting with the one with the smallest ratio. Each time, we insert an additional amount of drop and record the resulting plan and its effects on cycle savings and QoS as a new LSRM entry. Each entry builds on the plan of the previous plan.

To create a new LSRM entry, first the drop parameter p , which denotes the fraction of tuples to be dropped, has to be determined. Drops are applied in increments of a constant called $STEP_SIZE$. To explain how we set this parameter, we will turn back to Equation 3.4 presented in Section 3.4.2. To guarantee $G(x) > 0$, i.e., that the gain from drop insertion is more than its cost, we must ensure that $x > \frac{D}{L}$. For this purpose, we use a $STEP_SIZE$ parameter such that $STEP_SIZE > \frac{D}{\min\{L\}}$, where $\min\{L\}$ is the minimum load coefficient in the network (over all drop arcs). We use this minimum value so that the chosen $STEP_SIZE$ will work for all drop locations in the network. The value for this parameter also affects the granularity of the entries in LSRM. Thus, the granularity can be adjusted using the $STEP_SIZE$ parameter provided that $STEP_SIZE > \frac{D}{\min\{L\}}$.

Assume that the loss-tolerance QoS for the output that correspond to the current drop location has a percent delivery cursor (PDC) value of x (as stored in the previous LSRM entry). Then we choose the new drop amount such that the cursor moves to $x - 100 \times STEP_SIZE$. For this, the new drop to be inserted must have a drop rate of $p = 1 - (x - 100 \times STEP_SIZE)/x$.

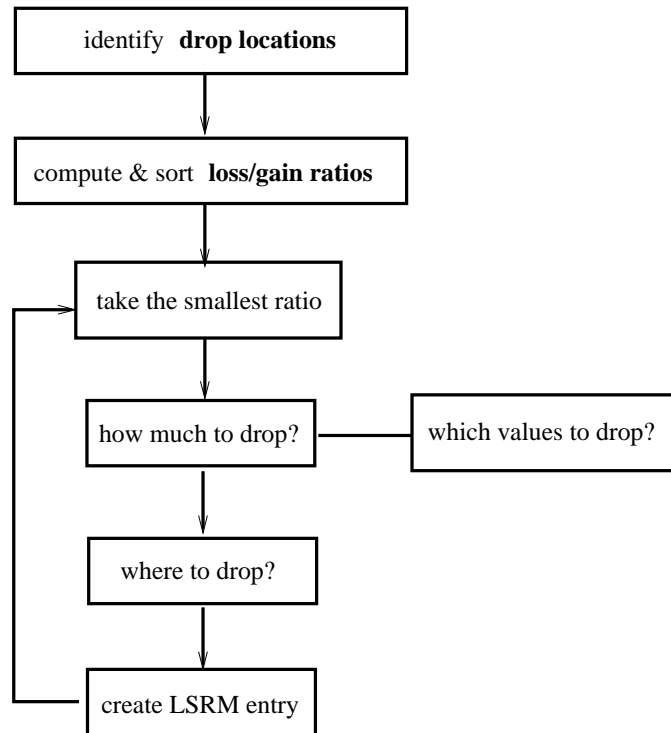


Figure 3.8: LSRM construction

After p is determined, we create a new drop operator to be placed at the designated drop location. Since every LSRM entry builds on a previous one, we must maintain the drop insertion decisions of earlier LSRM entries while placing the new one into the plan. Our algorithm achieves this by inserting the new drop at the output first, and then pushing it towards the designated drop location. Necessary adjustments are made on drop parameters of the existing drops that are encountered along the way. For example, if we were to insert a drop with $p = p_A$ at A in Figure 3.3, and if B already had a drop with $p = p_B$ where $p_B \leq p_A$, then the drop at B becomes redundant because the drop at A subsumes the amount that B is supposed to drop. On the other hand, if $p_B > p_A$, then B 's percentage must be reduced to produce a total percentage of p_B at the output.

New drops combined with the drops of the previous LSRM entry form the DIP of our new LSRM entry. We update the CSCs for each stream that sends tuples to the drops in the DIP. Finally, QoS cursors of the affected outputs are advanced and stored in PDCs. The next iteration of the loop takes the current entry and builds upon it. This way, each LSRM entry has more savings than the previous one.

3.6 Drop Operators

A drop operator forms the essence of our approach. There may be many instances of such an operator. We first define a generic drop operator. Then, we define several instances of it that we

Drop Type	Parameters	Metadata	Prime Target	Across
Random Drop	p : drop probability	Loss-tolerance QoS	order-agnostic operators	Filter Map Union
Window Drop	\mathcal{T} : windowing attribute \mathcal{G} : group-by attribute(s) ω : window size δ : window slide $p[\text{domain}(\mathcal{G})]$: drop probability array $\mathcal{B}[\text{domain}(\mathcal{G})]$: drop batch size array	Loss-tolerance QoS Batch size (\mathcal{B})	sliding window aggregates	Filter Map Union* Join* Aggregate
Semantic Drop	\mathcal{P} : drop predicate	Value-based QoS Histograms	value-preserving operators	Filter Union Join Aggregate*

Table 3.1: Drop operators (* indicates limited use)

specifically used in our approach.

Definition 3 (Drop Operator). *Given a data stream \mathcal{S}_{in} , with rate \mathcal{R}_{in} tuples per time unit, a drop operator is an operation that maps \mathcal{S}_{in} into \mathcal{S}_{out} with rate \mathcal{R}_{out} tuples per time unit, such that $\mathcal{S}_{out} \subseteq \mathcal{S}_{in}$ and $\mathcal{R}_{out} \leq \mathcal{R}_{in}$.*

Definition 4 (Random Drop). *Random drop is a drop operator which takes a parameter p that represents the probability that an input tuple $t \in \mathcal{S}_{in}$ will not appear in \mathcal{S}_{out} . For each input tuple t , random drop makes an independent decision using a Bernoulli distribution with drop probability p . The expected value for \mathcal{R}_{out} then becomes $E[\mathcal{R}_{out}] = (1 - p) * \mathcal{R}_{in}$.*

Definition 5 (Window Drop). *Window drop is a drop operator which takes six parameters \mathcal{T} , \mathcal{G} , ω , δ , $p[|\text{domain}(\mathcal{G})|]$, and $\mathcal{B}[|\text{domain}(\mathcal{G})|]$. \mathcal{T} denotes the windowing attribute and \mathcal{G} denotes the group-by attributes(s), where $|\text{domain}(\mathcal{G})|$ represents the number of distinct groups in the domain of \mathcal{G} . For each group $g \in \mathcal{G}$, $p[g]$ represents the probability that a consecutive batch of $\mathcal{B}[g]$ windows, each with size ω and slide δ will not appear in the final query result. For each batch of input tuples t in group g that is a window starter (as defined by ω and δ), window drop makes an independent decision using a Bernoulli distribution with drop probability $p[g]$. Window drop further marks this decision into t . Certain tuples succeeding t , if t is marked with a drop decision, are dropped by the window drop under some conditions. Details of this operator will be discussed in Chapter 4.*

Definition 6 (Semantic Drop). *Semantic drop is a drop operator which takes a drop predicate \mathcal{P} as parameter such that an input tuple $t \in \mathcal{S}_{in}$ will not appear in \mathcal{S}_{out} if $\mathcal{P}(t)$ evaluates to true. If selectivity of \mathcal{P} is s , then the expected value for \mathcal{R}_{out} becomes $E[\mathcal{R}_{out}] = (1 - s) * \mathcal{R}_{in}$.*

Table 3.1 provides a summary of our drop operators. The Metadata column denotes the QoS specifications that must be provided by output applications and the statistics that must be collected

by Aurora as required for using each type of drop. The Prime Target column indicates the kind of operators or query plans that each drop type is primarily designed for (or has the best fit). The last column of the table refers to the list of operators across which a particular drop operator can be placed. Window drop is an improved version of random drop which should be used to shed load across aggregation queries. Semantic drop should only be used across queries with operators that do not generate tuples with new attribute values (i.e., value-preserving operators). Query plans with the *-marked operators have limited use of the corresponding drop type (e.g., a semantic drop should not be placed upstream from an aggregate unless the drop predicate is defined on aggregate’s group-by attribute). These limitations mainly arise due to the subset guarantee requirement.

We call a load shedding algorithm *random load shedding* if it only uses random drop operators. Note that random drop is the simplest operator of our drop operator set. It requires a single parameter which indicates the probability that a given tuple should be discarded. Therefore, all the load shedding techniques that we described in the previous sections of this chapter essentially constitute our random load shedding algorithm. If the algorithm additionally uses window drops, we call it *window-aware*. These two algorithms drop tuples or tuple groups probabilistically, without paying attention to actual tuple values. If semantic drops are used instead, we call such an algorithm *semantic load shedding*, where drop decisions are taken based on tuple values. Next we discuss additional techniques that we specifically developed for the semantic and the window-aware load shedding algorithms.

3.7 Semantic Load Shedding

LSRM entries for semantic load shedding are created in almost the same way as for the random one. The major difference is that we need to create semantic drops with proper predicates which will provide us the desired level of load shedding, while filtering out the data values with the lowest utility. Therefore, in this section, we mainly discuss additional techniques that the semantic load shedding algorithm uses to construct the LSRM entries.

We developed a technique that enables us to follow the same method to determine the drop amount and location, as the one described for the random load shedder. Our approach essentially derives the loss-tolerance QoS graph from a given value-based QoS graph. In this way, the derived loss-tolerance QoS graph captures the value utility information and can be directly used for deciding loss/gain ratios.

3.7.1 Translation between QoS Functions

For the purposes of this study, we restrict value-based QoS graphs to be piece-wise linear functions. For simplicity, assume that values in each value interval have a constant utility⁵. Assume further

⁵For non-constant linear utility functions, we simply define a *chunk_size* and assume that each load shedding step will drop value ranges in multiples of a chunk. Hence, the values in the same chunk can be assumed to have the average utility of that piece of the utility function.

that the output data value histograms are available for each of the intervals specified in the value-based QoS graph. Such histograms are commonly created in conventional DBMS's to assist in query processing. A histogram shows the relative frequency of each value interval. Using a histogram and a value-based QoS graph, we can produce a loss-tolerance QoS graph as described below. We use the notation shown in Table 3.2.

symbol	description
u_i	utility of values in interval i
f_i	relative frequency of values in interval i , $\sum f_i = 1$
w_i	weighted utility of values in interval i , $w_i = u_i \times f_i$
n_i	normalized utility of values in interval i , ($n_i = \frac{w_i}{\sum w_i}$)

Table 3.2: Notation for translation between QoS functions

We order the value intervals based on their u values in ascending order and store them in a table together with their u , f , w , and n values. Table 3.3 illustrates such a table with two value intervals.

interval	u	f	w	n
0-50	0.2	0.4	0.08	$0.08/0.68 = 0.12$
51-100	1.0	0.6	0.6	$0.6/0.68 = 0.88$

Table 3.3: Example value intervals

The derivation relies on the fact that given a value-based QoS, if we needed to drop some tuples, we would always start dropping from the lowest utility interval (hence, the table is ordered on u). When we drop all values in an interval i with normalized utility n_i , then the utility for that output drops to $1 - n_i$. Based on the corresponding relative frequency f_i , we can infer that dropping values of interval i will lead us to drop about $f_i \times 100$ percent of the tuples. Therefore, while utility of 100% is 1, the utility of $(100 - f_i \times 100)\%$ drops to $1 - n_i$, and the utility values for the range in-between, $(100, 100 - f_i \times 100)$, decrease linearly.

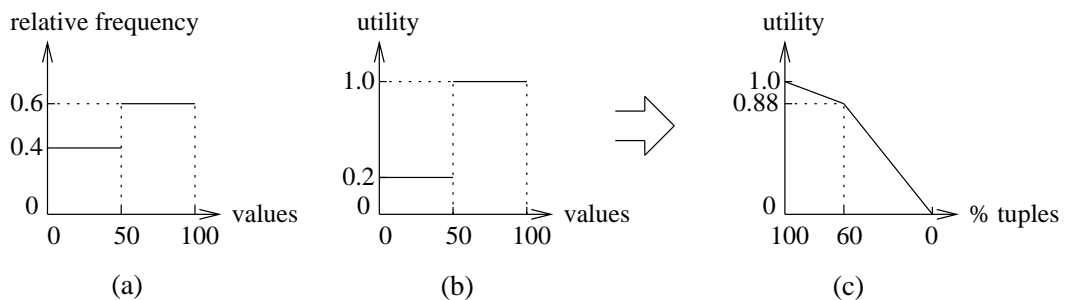


Figure 3.9: Derivation of the loss-tolerance QoS

Consider the simple example in Figure 3.9. A histogram for two value intervals (Figure 3.9(a)) and a value-based QoS (Figure 3.9(b)) are provided. Our goal is to generate a loss-tolerance QoS from these two. Notice that the first value interval makes up 40% of the values and has a normalized

utility of 0.12 (see Table 3.3). This means that when we drop 40% of the tuples, our utility drops from 1 to $1 - 0.12 = 0.88$. Therefore, the point (60, 0.88) is the inflection point at which the utility function and hence the slope changes. This leads us to the loss-tolerance QoS graph in Figure 3.9(c).

3.7.2 The Drop Predicate

There is a pre-determined order for dropping value intervals imposed by their utilities. We capture this by keeping a sorted list of intervals in ascending order of their utilities. The cursor on the loss-tolerance QoS graph, say x , indicates how much of the data we already dropped. Each time we need to drop an additional k percent of the tuples, we locate the right entry in the interval table based on relative frequency of the intervals.

Consider the example we presented in Figure 3.9. Assume that x is 100, i.e., we have not dropped any tuples yet. Assume also that the algorithm has decided to drop 20% of the tuples. Interval $[0, 50]$ has a percentage of 40% and it is the smallest utility interval that we have not dropped from before. We should drop the 20% from interval $[0, 50]$, i.e., half the data in this interval⁶. The interval to be dropped will be $[0, 25)$ and the predicate for the semantic drop operator to be inserted will be $value \geq 25$. If we needed to drop 70%, then interval $[0, 50]$ would not be sufficient. We would have to drop all of interval $[0, 50]$ plus half of the interval $[51, 100]$. Hence, the required predicate would be $value \geq 75$.

3.7.3 Semantic Drop on Joins

The main challenge of load shedding on joins is to be able to control the size of the resulting subset. Random sampling from join inputs does not work as the result can be reduced in arbitrary ways [30]. Load shedding on joins has to take either tuple contents into account, or assume some kind of stream arrival model [85]. Thus, our random load shedding approach by itself does not work well with join queries.

Our semantic load shedding approach, on the other hand, can easily be used with joins. Assume that we are given a value-based QoS function for the join output, defined on the join attribute. Additionally, we keep track of histograms for the join attribute, at both inputs as well as at the output of the join. Based on this metadata, we can infer loss-tolerance QoS functions for the join inputs, which we can then use to drop lowest utility values using a semantic drop with the appropriate predicate.

For example, assume the equi-join operator shown in Figure 3.10 with inputs S_1 and S_2 , and join attribute x . We are given histograms $H(x)$ for input and output streams. We are also given the value-based QoS function $U_v^o(x)$ for the join output. Based on these, we can derive the loss-tolerance QoS function $U_l^o()$ for the join output (see Table 3.4). Furthermore, we can infer loss-tolerance QoS functions $U_l^1()$ and $U_l^2()$ for the inputs by directly copying the normalized output utilities to the

⁶Any 50% of this interval could be dropped. However, we restrict our drop predicates to be range predicates. Therefore, we drop contiguous data values from beginning of the interval towards its end.

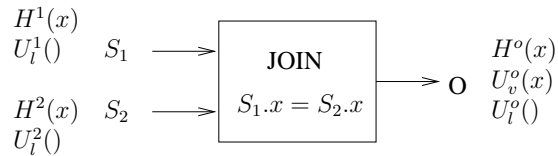


Figure 3.10: Join

$O.x$	utility	frequency	weighted utility	normalized utility
0-50	0.25	0.1	0.025	$0.025/0.925 = 0.027$
51-100	1.0	0.9	0.9	$0.9/0.925 = 0.972$

$S_1.x$	frequency	normalized utility
0-50	0.4	0.027
51-100	0.6	0.972

$S_2.x$	frequency	normalized utility
0-50	0.2	0.027
51-100	0.8	0.972

Table 3.4: Value-based QoS and other metadata for the join of Figure 3.10

corresponding value ranges. As discussed in Section 3.7.2, we can then generate an appropriate drop predicate for each join input depending on how much load needs to be shed. Note that we save the most processing cycles if we use the same drop predicate on both inputs of the join. The reason for this is that dropping a value range from one input, say S_1 , implies that this range will never appear in the join result. Therefore, there is no need to keep that range in S_2 . This would only waste cycles, trying to find matching tuples on S_1 that do not exist.

3.8 Simulation-based Performance Evaluation

We conducted various experiments to evaluate the performance of our approach. Our initial set of experiments have been performed on a simulation of the Aurora System implemented using the CSIM18 Simulation Engine [69].

3.8.1 Experimental Setup

Input Streams. Input streams have uniformly distributed integer values from the range $[0, 100]$. They arrive Aurora with a constant mean inter-arrival time.

Queries. For our experiments on the simulator, we used queries with a mix of filter and union operators. Filter predicates are simple comparisons of the form $value > constant$. Filter selectivities can be easily estimated since input values have a uniform distribution. Each operator is also assumed

Variable	Description
n	number of value intervals in value-based QoS graph
k	number of epochs, where an epoch is the time period during which the same percentage of tuples are being received.
n_i	number of tuples seen in epoch i
u_l^i	loss-tolerance utility of each tuple during epoch i
f_i	relative frequency of tuples for value interval i without drops
f'_i	relative frequency of tuples for value interval i with drops
u_v^i	average value utility for value interval i

Table 3.5: Notation for the utility formulas

to have a fixed known average time cost per tuple.

Quality of Service. We use value-based QoS graph as our primary QoS function. Given the value-based QoS for an output and its value histogram which we collect in a test run, we generate a loss-tolerance QoS graph for that output. We use two value intervals in the value-based graphs. The utility of the first interval is selected from a $[0, 1.0]$ range with a Zipf distribution, while the second interval has a utility of 1.0. Using Zipf distribution, we can control the skewedness of utility for the first value interval on QoS graphs of multiple output applications.

Evaluation Metrics. We use two metrics to evaluate the utility of a query output. *Tuple Utility* refers to the utility based on the loss-tolerance QoS graph. *Value Utility* refers to the utility based on the value-based QoS graph. The following formulas are used to compute these utilities:

$$\text{Tuple Utility} = \frac{\sum_{i=1}^k u_l^i \times n_i}{\sum_{i=1}^k n_i}$$

$$\text{Value Utility} = \frac{\sum_{i=1}^n f'_i \times u_v^i}{\sum_{i=1}^n f_i \times u_v^i}$$

The overall tuple (value) utility of a query network in the presence of multiple queries is computed by taking a sum of individual tuple (value) utilities for each query. Variables used in these formulas are described in Table 3.5.

3.8.2 Comparing Load Shedding to Simple Admission Control

Admission Control Algorithms

Input-Random. When an excess of load, ΔL , is detected, this algorithm randomly selects one input stream and sheds sufficient load on that stream to compensate for ΔL . If shedding all the data from the chosen input does not suffice, we select another input stream at random and repeat the same step until all of the remaining excess load is shed.

Input-Top-Cost. This is a variant of the Input-Random algorithm. Rather than selecting random inputs for load shedding, input stream with the highest load share is selected.

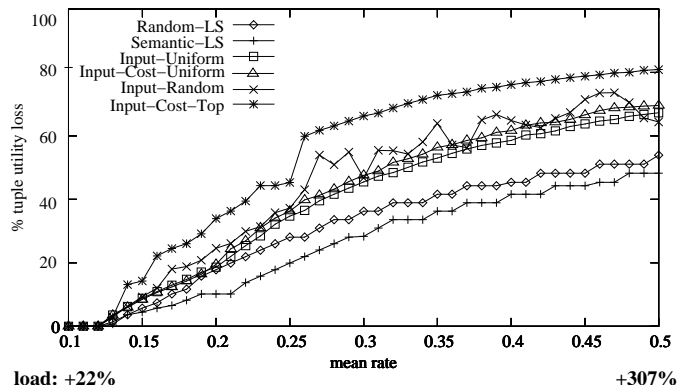


Figure 3.11: Load shedding vs. admission control variants (% Tuple utility loss)

Input-Uniform. Rather than choosing streams one at a time, this algorithm distributes the excess load evenly across all input streams, attempting to shed the same amount of load from each. If an input stream cannot provide its share of cycle gains, then the extra amount is distributed evenly to the other inputs until all of the excess load is shed.

Input-Uniform-Cost. This is a variant of the Input-Uniform algorithm that distributes excess load across all input streams weighted by their costs instead of an even distribution.

Random Load Shedding

Our first experiment quantifies the loss in tuple utility for different load shedding schemes and for varying overload levels. Different load levels are characterized by different mean (arrival) rates for the input streams. The mean arrival rate is defined as the mean number of tuple arrivals per time unit at each input stream in the network.

Figure 3.11 shows that all algorithms are clearly negatively affected by increasing input rates. Because the system has fixed capacity, the percentage of tuples that need to be dropped increases with increasing input rates, thereby, decreasing the loss-tolerance utility of the system.

As expected, we observe that the two QoS-driven algorithms perform much better than the admission control algorithms. They follow a similar pattern, which is not surprising as they make their decisions based on the same loss-tolerance QoS graph (generated from the same value-QoS graph). Utility loss for Semantic-LS is observed to be less than that of Random-LS by a constant amount. This is a result of information in the output value histograms that Semantic-LS can exploit whereas Random-LS cannot. Since some of the input tuples are filtered out by the filter operators before they reach the outputs, they show up in the input streams but not in the output. Those are the tuples that should be dropped from the input in the first place. They provide cycle gain without causing any utility loss at the output. Semantic-LS can capture this with a predicate, but Random-LS is only allowed to drop random tuples. The constant utility difference between the two curves amounts to this free “cycle gain” in Semantic-LS at no utility loss.

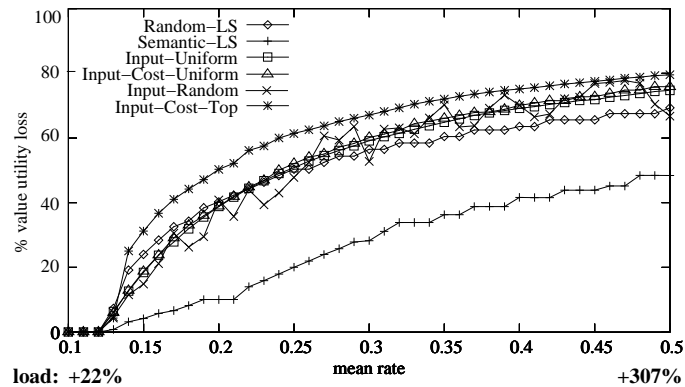


Figure 3.12: Load shedding vs. admission control variants (% Value utility loss)

The Input-Random and Input-Top-Cost algorithms perform poorly compared to others, incurring relatively higher utility losses for all input rates.

Because Input-Uniform spreads tuple drops uniformly across the applications, for low excess loads, all applications can manage to remain at the top, relatively flat portions of their loss-tolerance QoS graphs. With increased load, as shown in Figure 3.11, this situation changes and we start observing the benefits of the QoS-driven algorithms over the Input-Uniform algorithm. Weighting drops from input streams based on their costs does not help much and almost performs exactly the same as Input-Uniform for low excess loads.

Semantic Load Shedding

We now investigate the loss in the value utility for different algorithms and input rates. Our goal is to quantify the semantic utility gains we can achieve by exploiting information present in the value-QoS graph. We compare our value-based algorithm against others that do not utilize such semantic information.

Figure 3.12 clearly demonstrates that the semantic drop algorithm significantly outperforms the other approaches in terms of the value utility metric. Note that comparing the other approaches among each other based on the outcome of this experiment would not be fair. Since those approaches drop in a randomized fashion, they must be compared on the basis of the tuple utility metric, as we presented in the previous experiment.

3.8.3 Comparing Random and Semantic Load Shedding

In the scenarios we considered so far, the utility values for the first data intervals of different outputs were uniformly distributed. (Remember that the utility value of the second interval is taken as 1.0 for all the outputs.)

In order to characterize the impact of skewed utility distributions for different outputs, we devise a scenario where we inject skew to the utility values for the first data interval. We use a Zipf

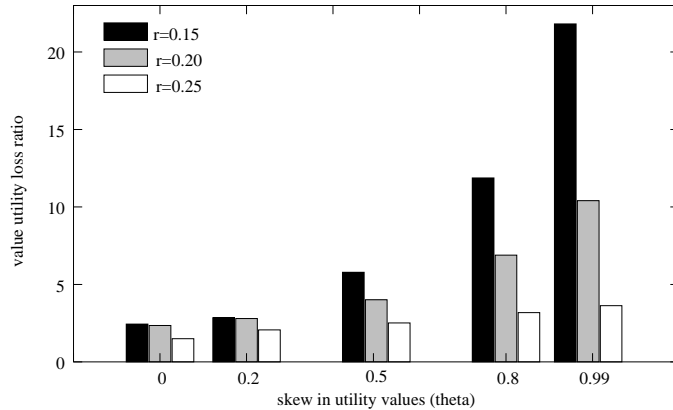


Figure 3.13: Value utility loss ratio for Random-LS/Semantic-LS vs. skew in utility

distribution to generate the utility values and use the Zipf parameter θ to control the skew. For low skew values, the utility values are more evenly distributed. For higher values, low utilities have higher probability of occurrence. Hence, we expect that with high skew, value-based dropping will perform much better than the randomized dropping approach. The rationale is that the latter will tend to drop high utility tuples, whereas the former will be able to fine-select the lower utility tuples for dropping.

We now demonstrate the effect of skew on utility loss for drops and filters for different values of mean input rates. Figure 3.13 illustrates our results. On the y-axis, we show the ratio of the value utility loss coming from the random load shedding algorithm to that coming from the semantic one. As we hypothesized, as the skew gets larger, the Semantic-LS algorithm gets increasingly more effective compared to the Random-LS algorithm. Interestingly, as the input rates increase, this effect tends to diminish. The reason is that when the rates are sufficiently high, the Semantic-LS algorithm also starts to drop tuples from the higher utility value intervals.

3.8.4 Evaluating the Effect of Operator Sharing

For our last experiment, we used a network with 20 identical queries. Queries receive input from a single input stream and have one common operator. This operator's output is split into 20 arcs and routed to the query-specific operators on separate arcs to create a scenario of full sharing among the queries.

This experiment investigates the behavior of our algorithms against the admission control algorithms in the presence of shared operators (i.e., splits) in the query network. In this experiment, we compare the tuple utility loss of two algorithms: Input-Uniform, the best of the admission control algorithms as observed in the earlier experiments; and our Random-LS algorithm. Our comparison here is based on the loss-tolerance QoS rather than the value-QoS, to factor out the advantage of our algorithms due to using semantic information.

The bar chart in Figure 3.14 shows how tuple utility loss ratio between the two algorithms change

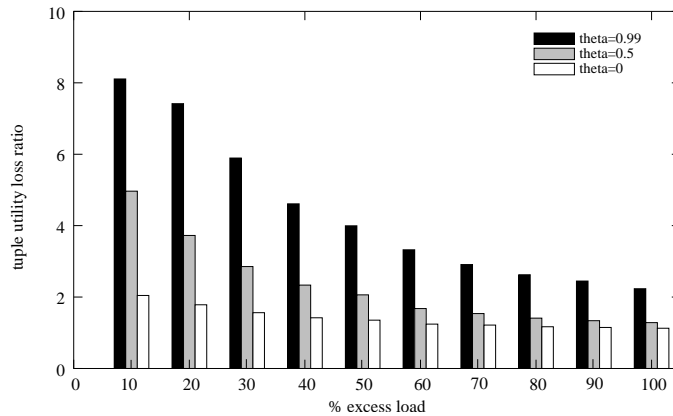


Figure 3.14: Tuple utility loss ratio for Input-Uniform/Random-LS vs. % excess load

as the amount of excess load in the system is increased. At each point of excess load, we present three different results. Each result is obtained using a different set of QoS graphs for the 20 queries. The loss-tolerance QoS graphs are generated from value-QoS graphs for which two data intervals are used: $[0, 75]$ with a utility chosen from a Zipf distribution with skew parameter θ , and $[76, 100]$ with utility 1.0. Hence, as the skew parameter theta increases, the uniformity of the QoS graphs decreases.

We observe that, as QoS graphs get more skewed, Random-LS performs better than the Input-Uniform algorithm. The reason is that our algorithm takes slopes of the QoS graphs into account while deciding where to shed load; whereas Input-Uniform always uniformly drops from the inputs. We further observe that the success of Random-LS against Input-Uniform starts to diminish as the amount of excess load gets to extremely high levels. This is because of the fact that, as the load increases to extreme levels, dropping from the inner arcs of the network does not suffice to recover all extra the cycles. Our algorithm is forced to adopt the plans down in the LSRM, which eventually correspond to dropping at input points of the query network.

3.9 Case Study: Battalion Monitoring

While developing Aurora, we have worked closely with a major defense contractor on a battlefield monitoring application. In this application, an advanced aircraft gathers reconnaissance data and sends it to monitoring stations on the ground. This data includes positions and images of friendly and enemy units. Commanders in the ground stations monitor this data for analysis and tactical decision making. Each ground station is interested in particular subsets of the data, each with differing priorities. In a potential battle scenario, the enemy units cross a given demarcation line on the field, and move toward the friendly units thereby signaling an attack (see Figure 3.15). When such an attack is initiated, the priorities for the data classes change. More data becomes critical, and the bandwidth likely saturates. In this case, selective dropping of data is allowed in order to service the more important classes.

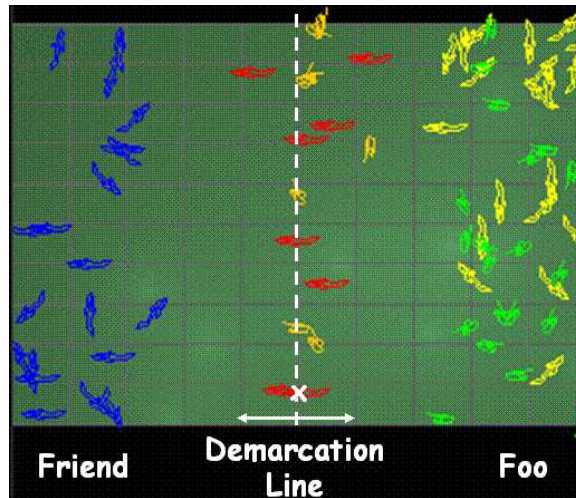


Figure 3.15: Battalion Monitoring

In the real application, the limiting resource is the bandwidth between the aircraft and the ground. For our purposes, we built a simplified version of this application to test our load shedding techniques. Instead of modeling bandwidth, we assume that the limited resource is the CPU. We introduce load shedding as a way to save cycles.

One of the query networks that we used in this case study is shown in Figure 3.16(a). There are four queries in this network. The *Analysis* query merges all tuples about positions of all units for analysis and archiving. The next two queries labeled *Enemy Tanks* and *Enemy Aircraft* select enemy tank and enemy aircraft tuples using predicates on their ids. The last query, *Across The Line*, selects all the objects that have crossed the demarcation line towards the friendly side.

Each query further has a value-based QoS function attached to its output. A value-based QoS function maps the tuple values observed at an output to utility values that express the importance of a given result tuple. In this example, the functions are defined on the *x-coordinate* attribute of the output tuple which indicates where an object is positioned horizontally. The functions take values in the range $[0, 500]$, of which 350 corresponds to the position of the vertical demarcation line. Initially all friendly units are on $[0, 350]$ side of this line whereas enemy units are on the $[350, 500]$ side. The QoS functions are specified by an application administrator and reflect the basic fact that tuples for enemy objects that have crossed the demarcation line are more important than others.

Query results are also displayed on a visualizer as shown in Figure 3.16(b). In the screenshot of Figure 3.16(b), there is one screen per query and an additional screen for the video results.

Figure 3.17 illustrates the performance monitoring GUI that we implemented to watch the changes in system load as the queries are running. There are two main screenshots shown in this Figure: (i) Figure 3.17(a) is showing the queue lengths on the arcs between the query operators (red and thick arcs indicating there is high tuple accumulation in the corresponding queues), as well as drop operators that are dynamically inserted at run-time (small gray boxes); (ii) Figure 3.17(b) is

showing a manual interface through which we can change the load shedding level in the system to see its effect on QoS values as presented on the small graphs on the righthand side.

We ran this query network with tuples generated by the Aurora workload generator based on a battle scenario that we got from the defense contractor. We fed the input tuples at different rates to create specific levels of overload in the network; then we let the load shedding algorithm remove the excess load by inserting drops to the network. Figure 3.18 shows the result. We compare the performance of three different load shedding algorithms in terms of their value utility loss (i.e., the average degradation in the QoS provided by the system) across all outputs at increasing levels of load.

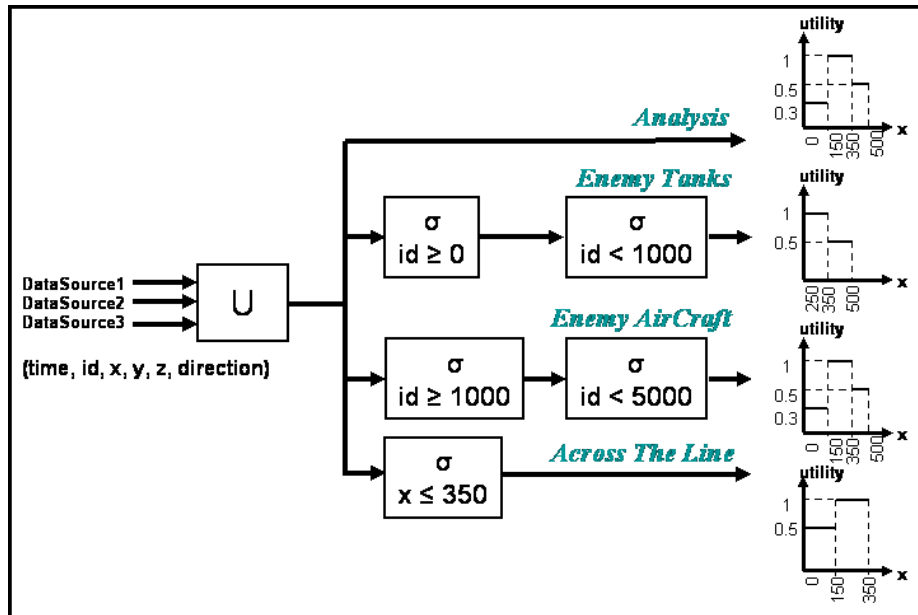
We make the following important observations: First, our semantic load shedding algorithm, which drops tuples based on attribute values, achieves the least value utility loss at all load levels. Second, our random load shedding algorithm inserts drops of the same amounts at the same network locations as the semantic load shedder. Since tuples are dropped randomly, however, loss in value utility is higher compared to the semantic load shedder. As excess load increases the performance of the two algorithms becomes similar. The reason is that at high load levels, our semantic load shedder also drops tuples from the high utility value ranges. Lastly, we compare both of our algorithms against a simple admission control algorithm which sheds random tuples at the network inputs. Both our algorithms achieve lower utility loss compared to this algorithm. Our load shedding algorithms may sometimes decide to insert drops on inner arcs of the query network. On networks with operator sharing among queries (e.g., the union operator is shared among all four queries in Figure 3.16(a)), inner arcs may be preferable to avoid utility loss at multiple query outputs. On the other hand, at very high load levels, since drops at inner arcs become insufficient to save the needed CPU cycles, our algorithms also insert drops close to the network inputs. Hence, all algorithms tend to converge to the same utility loss levels at very high loads.

3.10 Chapter Summary

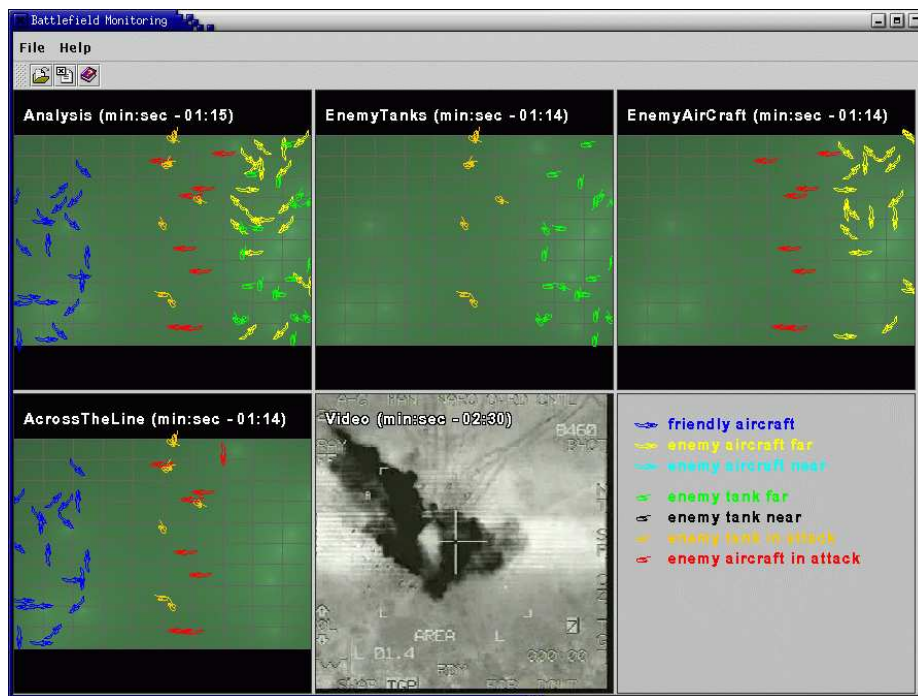
In this chapter, we have described the general problem of shedding load in a data stream management system by discarding tuples that have the least impact on QoS. We discussed the way in which we detect an overload, our mechanism for discarding tuples (i.e., inserting drops), and a technique for determining the proper location and the right magnitude of the drops. The key feature of our solution is that most of the analysis concerning dropping strategies can be done statically and captured in a simple data structure. The dynamic load shedding process involves a very cheap use of the static information. This technique makes our solution practical and scalable.

Also, our solution does not depend on the details of the scheduling algorithm. Instead it assumes that any cycles that are recovered as a result of load shedding are used sensibly by the scheduler to relieve the congestion. This makes our solution much more general in that it works equally well with any good scheduler.

We have shown some experimental evidence that our load shedding techniques outperform basic

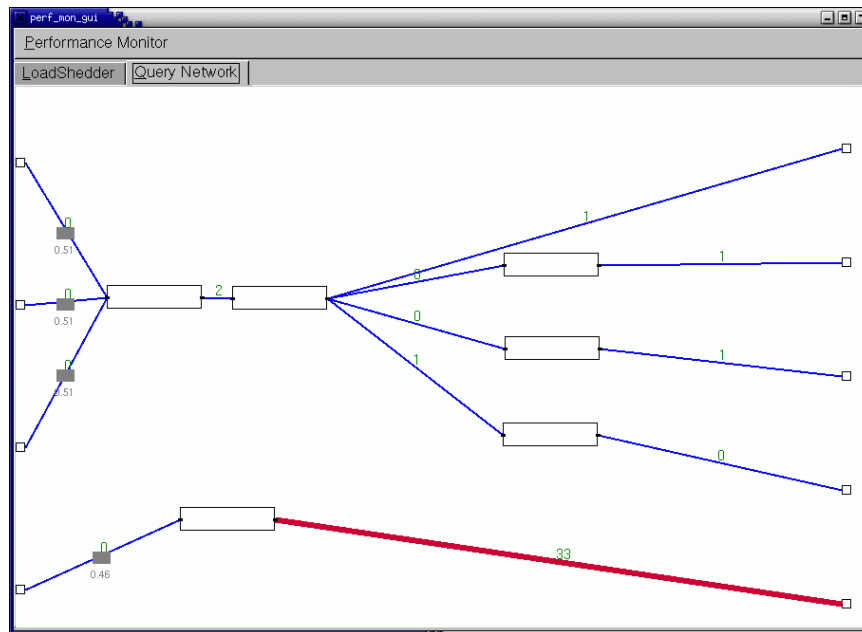


(a) Aurora Query Network for Battalion Monitoring Queries

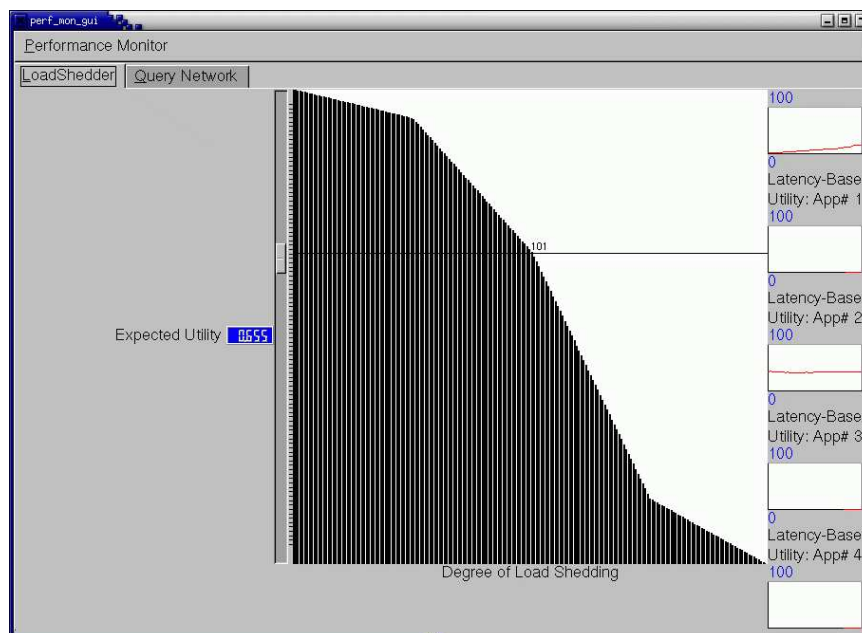


(b) Aurora Visualizer for the Battalion Monitoring Queries

Figure 3.16: Battalion Monitoring Queries



(a) Load Monitor



(b) Load Shedding Controller

Figure 3.17: Aurora Performance Monitoring GUI for Battalion Monitoring Queries

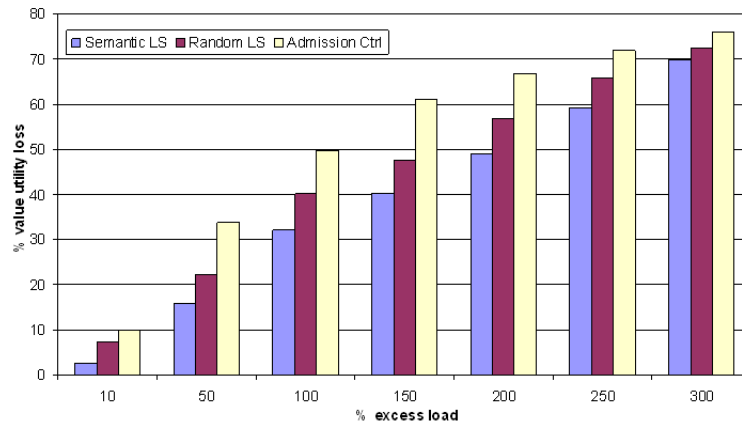


Figure 3.18: Load Shedding Results for the Battalion Monitoring Application

admission control and its variants. We have also shown that while our probabilistic dropping technique can do fairly well, the method that takes tuple semantics into account can do even better. Our experiments also clearly show that as we increase the difference in importance between the most valuable tuples and the least valuable tuples, semantic load shedding produces more striking benefits. All of these results verify our intuitions. The most crucial observation of this chapter is that it is possible to design a low-overhead mechanism for putting these concepts into practice in the context of a stream data manager.

Chapter 4

Window-aware Load Shedding

4.1 Overview

Window-based computation is an essential component of stream processing systems. As streams are potentially unbounded tuple sequences, often times query results are obtained by grouping tuples into finite windows of consecutive tuples. A key operation on windows is aggregation. In this case, an aggregate function such as sum, average, or count is applied on a window of tuples and an output tuple for that window is produced with the value of the function result. This function can also be an arbitrary user-defined function. Most stream processing systems provide full support for user-defined aggregates (e.g., [8, 65]). With such a capability, it is highly likely to use aggregates at arbitrary places in a query plan. Thus, *nested user-defined aggregates* have proven to be essential in various applications, ranging from habitat monitoring with sensors to online auctions [86], and highway traffic monitoring [15].

As a concrete example, consider the query plan in Figure 4.1, that computes the number of times in an hour that IBM's high price and low price in a 5-minute window differ by more than 5. Box 2 is a user-defined aggregate that collects all prices for a symbol in a 5-minute window, and then emits a tuple that contains the difference between the high and the low price. This stream is then filtered to retain differences that are larger than a given threshold, in this case, 5. A downstream aggregate then counts these extreme price differences. This kind of behavior can nest to an arbitrary depth.

Load shedding techniques devised so far, including our own approach described in the previous

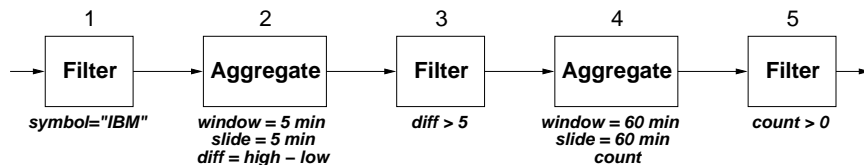


Figure 4.1: An example nested aggregation query

chapter, have applied drops in units of individual tuples. Drops are implemented as a specific operator, and this operator is pushed toward the inputs to avoid wasted work. A major limitation of this approach is that windowed operators such as aggregates, either block the motion of such drops (e.g., drop must be placed between box 4 and box 5 in Figure 4.1), or result in non-subset answers if drops are pushed across them. Furthermore, if inexact answers are produced through load shedding in the middle of the query plan, it is difficult to understand how this error will propagate through subsequent downstream operators. This further limits the query topologies across which such drops can be placed. For example, some solutions allow at most one aggregate operator in the query, and only to appear at the leaf of the query tree [17].

In this chapter, we introduce a new approach which applies drops in units of windows. This approach never produces wrong values and does not suffer from any of the problems mentioned above. It further enables the placement of drops at early points in a query plan (e.g., before box 1 in Figure 4.1), maximizing the amount of processing saved while keeping the error under control. More specifically, in this chapter, we study the problem of load shedding for aggregation queries over data streams, building upon our general framework that we described in the previous chapter. The main contributions of our work can be summarized as follows:

- We propose a novel load shedding approach for windowed aggregation queries which guarantees to deliver subset results.
- Our technique is general enough to handle arbitrary (user-defined) aggregate functions, multiple levels of aggregate nesting, and shared query plans.
- Regardless of where the aggregates appear in a query plan, our approach enables pushing drops across them, to early points in the plan, maximizing the amount of processing saved.
- We mathematically analyze the correctness and performance of our approach.
- We experimentally evaluate the performance of our approach on a stream processing system prototype.

The rest of this chapter is organized as follows: We first introduce important models and assumptions underlying our work. In particular, we introduce windowed aggregation queries in Section 4.2 and the subset-based approximation model in Section 4.3. Our subset-based, window-aware load shedding approach is presented in detail in Sections 4.4, 4.5, 4.6, 4.7, and 4.8. We provide a mathematical analysis of this approach in Section 4.9. Extensions to multiple aggregate groups and count-based windows are briefly described in Section 4.10. We present an experimental evaluation of our approach in Section 4.11. Finally, we conclude in Section 4.12.

4.2 Aggregation Queries

An *aggregation query* is composed of one or more aggregate operators along with other operators. Aggregate operators act on windows of tuples. Before we define the aggregate operator, we describe

how we model its two important building blocks: windows and aggregate functions.

4.2.1 The Window Model

Data streams are continuous sequences of data records that may have no end. Traditional set operations like join or aggregate may block or may require unbounded memory if data arrival is unbounded. Most applications, however, require processing on finite portions of a stream rather than the whole. Each such excerpt is called a *window*. Windows can be modeled in various ways [48]. In our system, there are two ways to physically build windows: (i) attribute-based windows, and (ii) count-based windows. In the first case, an attribute is designated as the windowing attribute (usually time), and consecutive tuples for which this attribute is within a certain interval constitute a window (e.g., stock reports over the last 10 minutes). Here, tuples are assumed to arrive in increasing order of their windowing attributes. In the second case, a certain number of consecutive tuples constitute a window (e.g., the last 10 readings from the temperature sensor). Our system also uses a sliding window model in which a window’s endpoints move by a given amount to produce the next window.

4.2.2 The Aggregate Function

An aggregate function \mathcal{F} takes in a window of values and performs a computation on them. \mathcal{F} can be a standard SQL-style aggregate function (sum, count, average, min, max) or a user-defined function. Aggregate functions in our system have the form $F(\text{init}, \text{incr}, \text{final})$, such that the `init` function is called to initialize a state when a window is opened; `incr` is called to update that state whenever a tuple that belongs to that window arrives; and `final` is called to convert the state to a final result when the window closes. Note that, as will soon become apparent, our approach is in fact independent of the particular aggregate functions used in a query.

4.2.3 The Aggregate Operator

An aggregate operator $Aggregate(\mathcal{S}, \mathcal{T}, \mathcal{G}, \mathcal{F}, \omega, \delta)$ has the following semantics. It takes an input stream \mathcal{S} , which is ordered in increasing order on one of its attributes denoted by \mathcal{T} , which we call the *windowing attribute*. If \mathcal{T} is not specified, $Aggregate$ requires no order on its input stream. In practice, \mathcal{T} usually corresponds to tuple timestamps which can either be embedded in the tuple during its generation at the source (e.g., temperature readings from a sensor, recorded with the time they were measured), or can be assigned by the stream processing system at arrival time. From here on, we will use the terms “timestamp” and “windowing attribute” interchangeably.

\mathcal{S} is divided into substreams based on optional group-by attribute(s) \mathcal{G} , if specified. Each substream is further divided into a sequence of windows on which the aggregate function \mathcal{F} is applied. $Aggregate$ ’s window properties are defined by two important parameters: *window size* ω and *window slide* δ . These parameters can be defined in two alternative ways: (i) in units of the windowing attribute \mathcal{T} (e.g., time-based window), (ii) in terms of number of tuples (i.e., count-based window).

According to the time-based windowing scheme, a window W consists of tuples whose timestamp values are less than ω apart. When *Aggregate* receives a tuple whose timestamp is equal to or greater than the smallest timestamp in $W + \omega$, W has to be closed. While ω denotes how large a window is and thus when it should be closed, δ denotes when new windows should be opened. Every δ time units, *Aggregate* has to open a new window. We assume that $0 < \delta \leq \omega$. When $\delta = \omega$, we say that we have a *tumbling window*. Otherwise, we say that we have a *sliding window*. Tumbling windows constitute an interesting case because they partition a stream into non-overlapping consecutive windows.

Aggregate outputs a stream of tuples of the form (t, g, v) , one for each window W processed. t is the smallest timestamp of the tuples in W , g is the value of the group-by attribute(s) (skipped when \mathcal{G} is not specified), and v is the final aggregate value returned by the `final` function of \mathcal{F} . In this work, we will initially assume that \mathcal{S} consists of a single group and windows are time-based. Extensions to more general forms of aggregates are provided in 4.10.

4.3 The Subset-based Approximation Model

Approximate answers result from dropping tuples. In our system, we shed load such that the total utility loss measured at the output is minimized. As explained in Chapter 2, a loss-tolerance QoS function maps the percent tuple delivery to utility values (see Figure 2.3(b)). The larger the percentage of output tuples delivered, the higher its utility to the receiving application. In the case of a single query, minimizing utility loss corresponds to providing the largest possible subset of the actual query result. In the case of multiple queries, the load to shed from each is based on their (possibly different) tolerance to loss. If no QoS functions are specified, then we assume that all applications have the same tolerance and the goal is to maximize the amount of total percent tuple delivery. It is important to note that in all of these cases we shed load such that the delivered answer is a subset of the original answer.

In this chapter, additionally, we assume that each output application can also specify a threshold for its tolerance to *gaps*. Gap represents a succession of tuples missing from the result. For example, we have worked on a sensor network application [91], in which a person’s physiological measurements must be delivered at least once per minute, i.e., losing or choosing not to deliver results observed more frequently is acceptable. We call the maximum output gap to which an application can tolerate the *batch size*. The system must guarantee that the amount of consecutive output tuples missed due to load shedding never exceeds this value. Note that batch size can be defined in terms of tuple counts or time units. In this work, we assume the former.

Note that batch size puts a lower bound on loss. Given a batch size \mathcal{B} , the query must at least deliver 1 tuple out of every $\mathcal{B} + 1$ tuples. Therefore, the fraction of tuples delivered can never be below $1/(\mathcal{B} + 1)$. Under heavy workload, it may not be possible to remove excess load while still meeting all applications’ bounds on \mathcal{B} . In this case, we apply “admission control” on queries, where the most costly queries whose bounds can not be met have to be completely shut down (by inserting

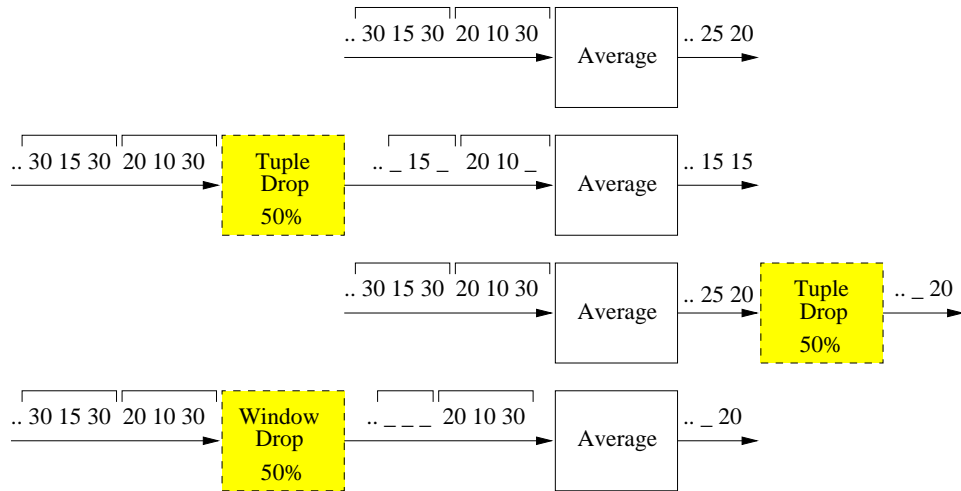


Figure 4.2: Drop alternatives for an aggregate

drops at their inputs with drop probability $p = 1$).

4.4 Window-aware Load Shedding with Window Drop

In our subset-based load shedding framework, queries deliver values all of which would also occur in the exact answer; no new values are generated. As such, this framework has to address an important challenge when windowed aggregates are involved: dropping individual tuples from streams does not guarantee subset results when such streams are to be processed by windowed aggregates. Therefore, if the goal is to deliver subset results as approximate answers, the windowing mechanisms of a query must be taken into account when discarding tuples. Thus, load shedding on windowed queries must be *window-aware*.

Let us illustrate our point with a simple example. Consider the aggregate operator in Figure 4.2, which computes 3-minute averages on its input in a tumbling window fashion. If we place a tuple-based random drop before the Average which cuts the load down by 50%, then we obtain a non-subset result of nearly the same size as the original. In this case, the load between the Tuple Drop and the Average is reduced by a factor of 50%, but the load is the same downstream from the Average. Alternatively, we can place the Tuple Drop after the Average, which drops tuples after the average has been computed. In this case, we produce a subset result of smaller size. However, load reduction has been achieved too late in the query plan, and we do not save from the computation of the aggregate. As a result, there is a tradeoff between achieving subset results and reducing load early in a query plan. We need a mechanism which would drop load before the Average, but would still produce a subset result. *Windowed aggregates deliver subset results if and only if they operate on original windows as indivisible units.* This observation led us to invent a new type of drop operator, called a *Window Drop*. As shown in Figure 4.2, the Window Drop can be placed before the Average, and applies drops in units of windows. As a result, it can achieve early load

Window Specification	Description
-1	don't care
0	window disallowed
τ	window allowed; must preserve tuples with $\mathcal{T} < \tau$

Table 4.1: Window specification attribute

reduction without sacrificing the subset guarantee.

A window drop operator $WinDrop(\mathcal{S}, \mathcal{T}, \mathcal{G}, \omega, \delta, p, \mathcal{B})$ takes six parameters in addition to an input stream \mathcal{S} . \mathcal{T} denotes the windowing attribute, \mathcal{G} denotes the group-by attribute(s), ω denotes the window size, δ denotes the window slide, p denotes the drop probability, and \mathcal{B} denotes the drop batch size. The $\mathcal{T}, \mathcal{G}, \omega, \delta$ parameters of $WinDrop$ are derived from the properties of the downstream aggregate operators. p is determined by the load shedder according to the amount of load to be shed. Finally, \mathcal{B} is derived based on the requirements of the output applications.

The basic functionality of $WinDrop$ is to encode window keep/ drop decisions into stream tuples to be later decoded by downstream aggregate operators. $WinDrop$ logically divides its input stream \mathcal{S} into time windows of size ω , noting the start of a new window every δ time units. For every group of \mathcal{B} consecutive windows, $WinDrop$ makes a probabilistic keep/drop decision. Each decision is an independent Bernoulli trial with drop probability p . The drop decision for a window is encoded into the tuple which is supposed to be the window's first (or starter) element, by annotating this tuple with a *window specification* value.

Each tuple has a window specification attribute as part of its system-assigned tuple header, with a default value of -1. To allow a downstream aggregate to open a window upon seeing a tuple t , $WinDrop$ sets the window specification attribute of t to a positive value for the windowing attribute \mathcal{T} . This value not only indicates that a window can start at this tuple, but also indicates until which \mathcal{T} value the succeeding tuples should be retained in the stream to ensure the integrity of the opened window. To disallow a downstream aggregate from opening a window upon seeing a tuple t , $WinDrop$ sets the window specification attribute of t to 0. Table 4.1 summarizes the semantics for the window specification attribute.

Consider an aggregate operator $Aggregate(\mathcal{F}, \omega, \delta)$ ¹ and assume that we would like to place a window drop before $Aggregate$. In order to drop p fraction from the output of $Aggregate$, we insert $WinDrop(\omega, \delta, p, \mathcal{B})$ at $Aggregate$'s input. Note that the first two parameters of $WinDrop$ are directly inherited from $Aggregate$ so that $WinDrop$ can divide the input stream into windows in exactly the same way as $Aggregate$ would. Then it decides which of those windows should be dropped and marks their starter elements. Finally, when a tuple t is received by $Aggregate$, $Aggregate$ examines t 's window specification attribute and skips windows that are disallowed. As a result of this, we save system resources at multiple levels. First, when a window is skipped,

¹We do not show \mathcal{S} when the input stream is clear from the context. From here on, we also drop the \mathcal{T} and \mathcal{G} parameters from both aggregate and window drop, simply assuming that windows are commonly defined on time, and \mathcal{S} consists of a single group.

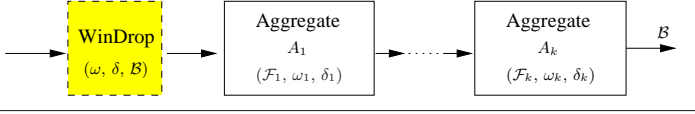
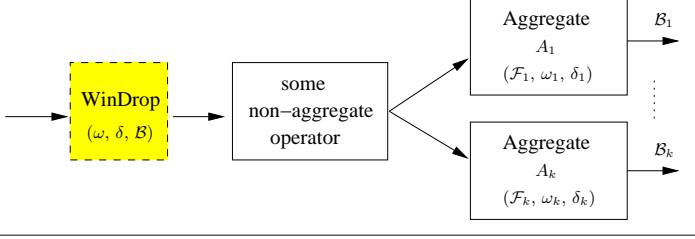
Aggregate Arrangement	Parameters for WinDrop
<p><i>Pipeline:</i></p> 	$\omega = \sum_{i=1}^k \omega_i - (k - 1)$ $\delta = \delta_k$ \mathcal{B}
<p><i>Fan-out:</i></p> 	$\omega = \text{lcm}(\delta_1, \dots, \delta_k)$ $+ \max_{i=1}^k \{ \text{extent}(A_i) \}$ <p>where $\text{extent}(A_i) = \omega_i - \delta_i$</p> $\delta = \text{lcm}(\delta_1, \dots, \delta_k)$ $\mathcal{B} = \min_{i=1}^k \left\{ \frac{\mathcal{B}_i}{\text{lcm}(\delta_1, \dots, \delta_k) / \delta_i} \right\}$

Table 4.2: Rules for setting window drop parameters

Aggregate need not open and maintain state for that window. In other words, *Aggregate* does less work upon seeing tuples that arrive immediately after t because there is one fewer open window that those tuples can contribute to. Second, when *Aggregate* skips a window, it produces no output for that window, thereby reducing data rate and saving from processing in the downstream subnetwork. Third, *WinDrop* not only encodes window specifications into tuples, but it is also capable of actually dropping tuples under certain conditions, which we call an *early drop*. More specifically, tuples that are marked with a negative window specification value and that are beyond the \mathcal{T} range imposed by the most recently seen positive window specification value can be dropped right away, without waiting to be seen by a downstream aggregate. Early drops are discussed in detail in Section 4.7. It should be emphasized here that the ability to move a drop upstream from an aggregate enables us to continue pushing it toward the inputs. This is important as it can save computation for the complete downstream subquery from where it ends up.

4.5 Handling Multiple Aggregates

There are two basic arrangements of aggregates in a query network: (1) a pipeline arrangement, (2) a fan-out arrangement. Table 4.2 summarizes the rules for setting window drop parameters for these two arrangements. Any query network can be handled using a composition of these two rules. We now discuss these rules and how we derived them in detail.

4.5.1 Pipeline Arrangement of Aggregates

A query arrangement with a sequence of operators where each operator's output is fed into another one is called a *pipeline arrangement*.

Assume that we have k aggregates, $A_i(\mathcal{F}_i, \omega_i, \delta_i)$, $0 < i \leq k$, pipelined in ascending order of i ,

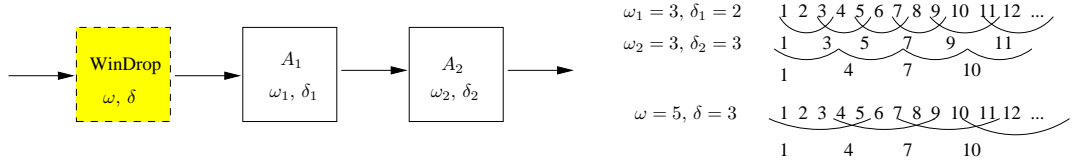


Figure 4.3: Pipeline example

as shown in the first row of Table 4.2. We would like to drop p fraction from the output of A_k by placing $WinDrop(\omega, \delta, p, \mathcal{B})$ before the leftmost operator in the pipeline (A_1). $WinDrop$ must have a slide δ that is equal to the slide of the last aggregate A_k in the pipeline. The reason for this is that A_k is the last operator that divides the stream into windows and produces one output every δ_k time units. Dropping p fraction from A_k 's output requires that we encode a drop decision once every δ_k time units. Furthermore, $WinDrop$ must have a window size which will guarantee the preservation of all tuples of a window W when W is kept. If we only had A_k , the window size would simply be ω_k . However, there are $k - 1$ aggregates preceding A_k , each with its own corresponding window of tuples to be preserved. To be on the safe side, we consider the following worst case scenario: To produce an output tuple t_m with time m , A_k needs outputs of A_{k-1} in the range $[t_m, t_{m+\omega_k})$; A_{k-1} in turn needs outputs of A_{k-2} in the range $[t_m, t_{m+\omega_k+\omega_{k-1}})$; and so on. Finally, A_2 needs outputs of A_1 in the range $[t_m, t_{m+\omega_k+\dots+\omega_2-(k-2)})$ and A_1 needs stream inputs $[t_m, t_{m+\omega_k+\dots+\omega_1-(k-1)})$ in order to guarantee the desired range. Therefore, $WinDrop$ has to preserve a window of size $\omega_1 + \dots + \omega_k - (k - 1)$ whenever it decides to retain a window, which forms its effective window size. Note that this is a conservative formulation, based on the worst case scenario when each aggregate's window slide is such that the last time value in a window opens up a new window. As such, it is an upper bound on the required window size for $WinDrop$. Finally, the batch size parameter \mathcal{B} of $WinDrop$ is assigned as specified by the output application at the end of the pipeline.

The simple example in Figure 4.3 illustrates the pipeline arrangement rule. We show a query that consists of two aggregates. A_1 has a window size and slide of 3 and 2 respectively, followed by A_2 with window size and slide of 3 each. We first show how an input stream with the indicated time values is divided into windows by these aggregates consecutively. Then we show the corresponding $WinDrop$ to be placed before this arrangement. According to our pipeline arrangement rule, $WinDrop$ must have a window size and slide of 5 and 3 respectively. Hence, it divides the input stream as shown, marking the tuples that correspond to window starts. Notice how $WinDrop$ considers input tuples with time values in the range $[1, 6)$ as an indivisible window unit to produce a result tuple with time value of 1. The original query uses exactly the same time range to produce its result with time value of 1.

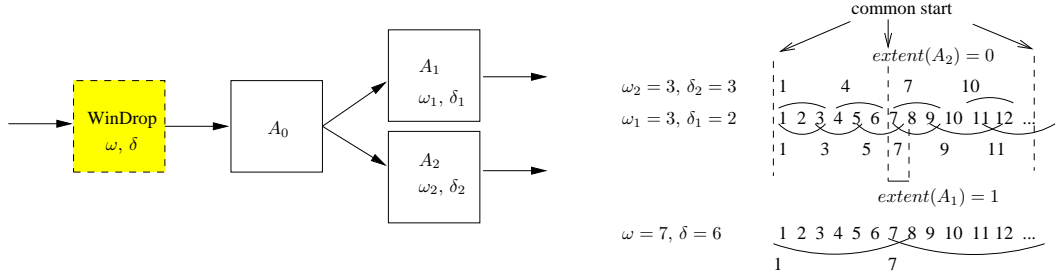


Figure 4.4: Fan-out example

4.5.2 Fan-out Arrangement of Aggregates

A query arrangement with an operator whose output is shared by multiple downstream branches is called a *fan-out arrangement*.

When there are aggregates at child branches of a fan-out, we need a *WinDrop* which makes window keep/drop decisions that are common to all of these aggregates. Assume k sibling aggregates, $A_1(\mathcal{F}_1, \omega_1, \delta_1), \dots, A_k(\mathcal{F}_k, \omega_k, \delta_k)$, as in the second row of Table 4.2. A common *WinDrop* for all aggregates would have a drop probability of p , a window slide of $\text{lcm}(\delta_1, \dots, \delta_k)$ and a window size of $\text{lcm}(\delta_1, \dots, \delta_k) + \max(\text{extent}(A_1), \dots, \text{extent}(A_k))$, where $\text{extent}(A_i) = \omega_i - \delta_i$. $\delta = \text{lcm}(\delta_1, \dots, \delta_k)$ represents the lowest common multiple of slides of all sibling aggregates, i.e., every δ time units, all aggregates start a new window at the same time point. Assume T to be such a time point where all aggregates meet to start a new window. $\text{extent}(A_i)$ represents the number of time units that A_i needs beyond T in order to cleanly close its most recently opened window. A_i must have opened a window at $T - \delta_i$, because its next window will be starting at T . Therefore, its extent beyond T is $\omega_i - \delta_i$. We take the maximum of all the aggregates' extents so that all aggregates can cleanly close their open windows. As a result, the logical window that encloses all aggregate siblings must have a window size of $\omega = \text{lcm}(\delta_1, \dots, \delta_k) + \max(\text{extent}(A_1), \dots, \text{extent}(A_k))$. In other words, window slide δ is formulated such that each time *WinDrop* slides, it positions itself to where all of the aggregates, A_1 through A_k , would attempt to start new windows. Window size ω is formulated such that when a keep decision is made, enough of the range is kept to preserve integrity of all of the aggregates' windows. Finally, the batch size of *WinDrop* is the minimum allowed by all sibling aggregates. Note that we need to scale each aggregate's batch size \mathcal{B}_i before computing the minimum. This scaling is required because, when *WinDrop* slides once, A_i slides $\text{lcm}(\delta_1, \dots, \delta_k)/\delta_i$ times. Hence, $\text{lcm}(\delta_1, \dots, \delta_k)/\delta_i$ consecutive windows for A_i correspond to 1 window for *WinDrop*.

The example in Figure 4.4 illustrates the fan-out arrangement rule. We show a query that consists of two sibling aggregates. Window sizes and slides of these aggregates are the same as in the pipeline example of Figure 4.3. Both aggregates receive a copy of the stream emanating from their parent, but they divide it in different ways based on their window parameters. We first show how this is done together with the extents and common window start positions for the aggregates. Both aggregates start new windows at time values 1 and 7. A_1 has an extent of 1 (i.e., its last window before a new

window starts at 7 extends until 8). A_2 has an extent of 0 (i.e., its last window completely closes before a new window opens at 7). Based on these, we show the corresponding *WinDrop* that must be placed before this aggregate arrangement. *WinDrop* must have a window size and slide of 7 and 6 respectively. This way, it makes window keep/drop decisions at time values where both A_1 and A_2 expect to open new windows. Furthermore, in case of a keep decision, *WinDrop* retains all tuples required to cleanly close open windows of both of the aggregates.

4.5.3 Composite Arrangements

We will now briefly illustrate the composition of the rules in Table 4.2. Assume that A_0 in Figure 4.4 is an aggregate with $\omega_0 = 4$ and $\delta_0 = 1$ (i.e., $A_0(4, 1)$). Thus, we have a combined arrangement with two pipelines and a fan-out. There are two alternative ways to construct *WinDrop* for this arrangement:

1. We first apply the fan-out rule on A_1 and A_2 , which gives us *WinDrop*(7, 6) as illustrated in Figure 4.4. Then we apply the pipeline rule on $A_0(4, 1)$ and *WinDrop*(7, 6), which gives us *WinDrop*(10, 6).
2. We first apply the pipeline rule on paths $[A_0(4, 1), A_1(3, 2)]$ and $[A_0(4, 1), A_2(3, 3)]$, which gives us *WinDrop*(6, 2) and *WinDrop*(6, 3), respectively. We then apply the fan-out rule on these, which gives us *WinDrop*(10, 6).

4.6 Decoding Window Specifications

As mentioned earlier in Section 4.4, window drop attaches window specifications to tuples that are potential window starters. These specifications further indicate the fate of those windows and need to be decoded by downstream aggregates in order for them to take the right action. In this section, we describe how this decoding mechanism works.

Table 4.3 summarizes how an aggregate *Aggregate* with window size ω decodes the window specifications coded by a preceding *WinDrop*. First assume that *Aggregate* receives a tuple with time value t and according to the slide parameter of *Aggregate*, a new window has to start at t (upper half of Table 4.3). If the tuple has a positive window specification τ , then *Aggregate* opens a new window with a window specification attribute of $\tau - (\omega - 1)$ (i.e., when this window closes and produces an output tuple, the window specification of this output tuple will be $\tau - (\omega - 1)$). *Aggregate* also has to make sure that all successive tuples with time values up to $\tau - (\omega - 1)$ are retained in the stream (i.e., *Aggregate* sets its keep_until variable to $\tau - (\omega - 1)$). If the tuple has a non-positive (0 or -1) window specification, then *Aggregate* checks if t is within the time range that it must retain (i.e., if $t < \text{keep_until}$). If so, a new window is opened with the given window specification and the keep range is set to $\max(\text{keep_until}, t + \omega)$. If not, *Aggregate* skips this window.

Now assume that *Aggregate* receives a tuple with time value t where *Aggregate* does not expect to open a new window (lower half of Table 4.3). *Aggregate* will not open any new window. However, it

win_start?	win_spec	keep_until	relevant action
yes	τ	within	open window keep_until = $\tau - (\omega - 1)$ win_spec = $\tau - (\omega - 1)$
yes	τ	beyond	open window keep_until = $\tau - (\omega - 1)$ win_spec = $\tau - (\omega - 1)$
yes	0	within	open window keep_until = $t + \omega$, (if >)
yes	0	beyond	skip window
yes	-1	within	open window keep_until = $t + \omega$, (if >)
yes	-1	beyond	skip window
no	τ	within	keep_until = $\tau - (\omega - 1)$ win_spec = $\tau - (\omega - 1)$ mark as fake tuple
no	τ	beyond	keep_until = $\tau - (\omega - 1)$ win_spec = $\tau - (\omega - 1)$ mark as fake tuple
no	0	within	mark as fake tuple
no	0	beyond	mark as fake tuple
no	-1	within	ignore
no	-1	beyond	ignore

Table 4.3: Decoding window specifications

has to still maintain the window specification attribute in the tuple for other downstream aggregates' disposal (if any). The two important specifications are τ and 0, the former indicating the opening of a window and the latter indicating the skipping of a window. If the specification is -1, *Aggregate* does not need to do anything. If the tuple has a positive window specification τ , *Aggregate* updates its time range as well as the window specification of the tuple. In both of the non-negative cases, *Aggregate* marks this tuple as a *fake tuple*. A fake tuple is one which has no real content but only carries a window specification value that may be significant to some downstream aggregates. Such tuples should not participate in query computations and should be solely used for decoding purposes.

We must point out here that fake tuples have one other important use. A query network may have other types of operators lying between a window drop and the downstream aggregates which are supposed to decode window specifications generated by the window drop. We must make sure that window specifications correctly survive through such operators. For example, assume that the filter between the two aggregates in Figure 4.1 (box 3) decides to drop a tuple t from the stream since this tuple does not satisfy its predicate. If t is carrying a non-negative window specification, then we can not simply discard it. Instead, we must mark t as a fake tuple and let it pass through the filter. This is because t is carrying a message for the downstream aggregate (box 4) about whether to open or to skip a window at a particular time point.

Note that it can be argued that fake tuples introduce additional tuples into the query pipeline.

However, since these are not real tuples, operators except aggregates will just pass them along without doing any processing on them, whereas aggregates will check the flag to see if they should open a window. Hence, it is unlikely that fake tuples will drive the system into overload.

4.7 Early Drops

Window drop not only marks tuples, but it can also drop some of them. In this section, we discuss how this early drop mechanism works. We start with a useful definition.

Definition 7 (Window Count Function (WCF)). *Consider a stream \mathcal{S} with tuples partially ordered in increasing order of their time values. Assume that the very first tuple in \mathcal{S} has a time value of θ . Consider an aggregate $\text{Aggregate}(\mathcal{S}, \omega, \delta)$, where $\omega = m * \delta + \phi$, $m \geq 1$, $0 \leq \phi < \delta$. We define a Window Count Function $WCF : \mathbb{Z}^* \rightarrow \mathbb{N}$, that maps time value t to the number of consecutive windows to which tuples with t belong as:*

$$WCF(t) = \begin{cases} i + 1, & \text{if } t \in [\theta + i * \delta, \theta + (i + 1) * \delta - 1], \text{ where } 0 \leq i < m \\ m + 1, & \text{if } t \in [\theta + i * \delta, \theta + (i - m) * \delta + \omega - 1], \text{ where } i \geq m \\ m, & \text{if } t \in [\theta + (i - m) * \delta + \omega, \theta + (i + 1) * \delta - 1], \text{ where } i \geq m \end{cases}$$

Note that the first case only occurs once at the start of the stream. Thereafter, the second and the third cases occur repeatedly one after the other. If the aggregate window is tumbling (i.e., $\omega = \delta$), then the second case has a time range length of 0, i.e., it is skipped. Also, the first case is equivalent to the third case since $m = 1$. As a result, for tumbling window aggregates, $WCF(t) = m = 1$ for all tuples (i.e., each tuple belongs to only 1 window).

Rule 1 (Early Drop Rule). *If a tuple with time value t belongs to k windows (i.e., $WCF(t) = k$), then this tuple can be early-dropped if and only if the window drop operator decides to drop all of these k windows.*

If a window drop operator *WinDrop* flips a coin every time it observes a potential window start and decides to drop that window with probability p , then for an early drop, *WinDrop* has to flip the coin for k consecutive times, which has probability p^k . Unless p is a big number or k is a small number (e.g., in the case of a tumbling window), then the probability of an early drop is very small. Instead, to take advantage of early drops, we use the following (more deterministic) drop mechanism: We mentioned in Section 4.3 that, to indicate its tolerance to gaps in the answer, each query specifies a constant \mathcal{B} for the maximum number of consecutive windows that can be shed. Given a drop probability p , *WinDrop* flips the coin once for every batch of \mathcal{B} windows and drops them *all* with probability p . Based on the window count function *WCF*, dropping \mathcal{B} consecutive windows corresponds to a certain number of early drops. Note that to satisfy \mathcal{B} , at least one window has to be opened after each dropped batch. If the coin yields two consecutive drops, then we allow the first window of the second batch to open and compensate for it later by skipping a window when in fact the coin yields a keep. This ensures that we satisfy both \mathcal{B} and p .

4.8 Window Drop Placement

In general, load reduction should be performed at the earliest point in a query plan to avoid wasted work. However, there may be certain situations where placing drops at inner arcs of the query plan might be more preferable than placing them at the input arcs. We will briefly discuss these situations.

Unless \mathcal{B} is large enough to allow early drops (i.e., $\mathcal{B} \geq \lfloor \frac{\omega}{\delta} \rfloor$), there is no benefit in placing a window drop operator *WinDrop* further upstream than the leftmost aggregate in the pipeline. For the pipeline arrangement, as we place *WinDrop* further upstream, the difference between ω and δ widens (i.e., $m = \lfloor \frac{\omega}{\delta} \rfloor$ in Definition 7 grows). Similarly, for the fan-out arrangement, both ω and δ may get larger across a split while \mathcal{B} may get smaller. *WinDrop* must be placed at the earliest point in the query where it saves processing while also not violating the constraints on \mathcal{B} .

Although not so common, a query plan may have multiple aggregates with different sliding window properties over the same data stream (e.g., a pipeline arrangement with a mix of count-based and time-based windows, and/or with different group-by attributes). In this case, the window drop must be placed at a point where such properties are homogeneous downstream. It requires further investigation to extend our framework to handle the heterogeneous case.

4.9 Analysis

Next we mathematically analyze our approach for correctness and performance.

4.9.1 Correctness

Definition 8 (Correctness). *A drop insertion plan is said to be correct if it produces subset results at query outputs.*

Theorem 5. *WinDrop inserted aggregation queries preserve correctness.*

Proof. The proof for this theorem has two parts, one for each aggregate arrangement. We can prove each by induction. Consider a pipeline \mathcal{P} of N aggregates $A_i(\omega_i, \delta_i)$. Given a finite input stream \mathcal{S} , assume that the result of $\mathcal{P}(\mathcal{S})$ is the set \mathcal{A} , and the result for the window drop inserted version, $\mathcal{P}'(\mathcal{S})$, is the set \mathcal{A}' . For $N = 1$, *WinDrop*(ω, δ) is inserted before A_1 such that $\omega = \omega_1$, $\delta = \delta_1$. Every δ_1 time units, *WinDrop* marks a tuple t as either keep (τ , where $\tau = t.time + \omega$), or drop (0). When A_1 receives t with specification of τ , it opens a new window at $t.time$ and retains all tuples in time range $[t.time, \tau)$. In this case, A_1 delivers an output tuple $o \in \mathcal{A}$. When A_1 receives t with a 0 or -1 specification, it does not open a window. In this case, A_1 adds no output tuple to the result. Therefore, $\mathcal{A}' \subseteq \mathcal{A}$. Next, assume that the theorem holds for $N = n$. We will show that it must also hold for $N = n + 1$. We are given that a *WinDrop*(ω, δ), with $\omega = \sum_{i=1}^n \omega_i - (n - 1)$ and $\delta = \delta_n$, inserted before A_2 preserves correctness. Consider a window W at A_1 with a time range of $[T, T + \omega_1 - 1]$, when processed produces an aggregate output with time value T . Any aggregate

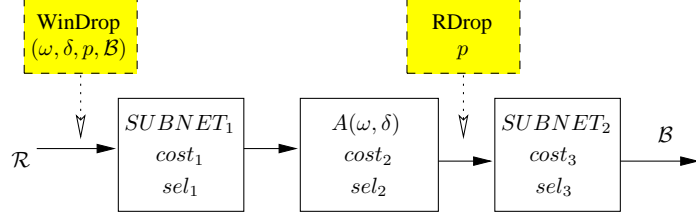


Figure 4.5: Inserting drops into an aggregation query

downstream from A_1 , that includes a tuple with time T in its window effectively incorporates S values with time up to $T + \omega_1 - 1$. Therefore, if $WinDrop'$ is placed before A_1 , its effective window size must include this range to preserve window integrity. As a result, $WinDrop'$ must have a window size of $\omega' = \omega + \omega_1 - 1 = \sum_{i=1}^n \omega_i - (n - 1) + \omega_1 - 1 = \sum_{i=1}^{n+1} \omega_i - (n + 1 - 1)$. This proves our window size formulation for a pipeline of $n + 1$ aggregates. Finally, in order to produce subset results, the $WinDrop'$ must produce results either δ_{n+1} apart or in multiples of this quantity. Therefore, $WinDrop'$ must have a window slide of $\delta' = \delta_{n+1}$. This concludes the first part of our proof. The part for the fan-out case follows a similar inductive reasoning, therefore we do not discuss it here. \square

4.9.2 Performance

We now analyze the effect of window drop on CPU performance. We also compare it against the random drop alternative [93]. Consider a query network as in Figure 4.5, where an aggregate $A(\omega, \delta)$ is present between two subnetworks of other non-aggregate operators, whose total costs and selectivities are as shown. The CPU cycles needed to process one input tuple across this query network can be estimated as $cost_1 + sel_1 * (cost_2 + sel_2 * cost_3)$. If the input stream has a rate of \mathcal{R} tuples per time unit, then the CPU load as processing cycles per time unit is $\mathcal{R} * (cost_1 + sel_1 * (cost_2 + sel_2 * cost_3))$.

If a random drop were inserted downstream from the aggregate operator, the CPU load would become:

$$L_{RDrop} = \mathcal{R} * (cost_1 + sel_1 * (cost_2 + sel_2 * (cost_{RDrop} + (1 - p) * cost_3)))$$

The CPU cycles saved as a result of this would be:

$$S_{RDrop} = \mathcal{R} * (sel_1 * sel_2 * p * cost_3 - sel_1 * sel_2 * cost_{RDrop})$$

Instead, if a $WinDrop$ were inserted at the query input, the CPU load would become:

$$L_{WinDrop} = \mathcal{R} * (cost_{WinDrop} + cost'_1 + cost'_2 + sel_1 * sel_2 * (1 - p) * cost_3)$$

$$cost'_1 = cost_{fcheck} + sel_{w1} * cost_1 + sel_{w2} * cost_{copy}$$

$$cost'_2 = \frac{sel_{w1}}{\delta} * cost_{wcheck} + sel_{w2} * cost_{wcheck} + sel_{w1} * sel_1 * cost_2$$

$SUBNET_1$ first checks if a tuple is fake or not ($cost_{fcheck}$). Assume sel_{w1} of tuples from $WinDrop$ are normal and sel_{w2} of them are fake. Then, former are processed normally ($sel_{w1} * cost_1$), and latter are just copied across to A ($sel_{w2} * cost_{copy}$). A checks window specification attributes for

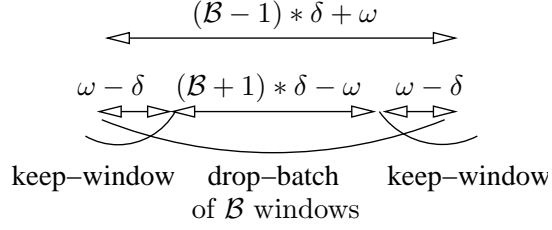


Figure 4.6: Drop-batch (when $\mathcal{B} \geq \lfloor \frac{\omega}{\delta} \rfloor$)

ones to be opened ($\frac{sel_{w1}}{\delta} * cost_{wcheck}$) and for ones to be skipped ($sel_{w2} * cost_{wcheck}$). Then, tuples in the former group go through normal aggregate processing ($sel_{w1} * sel_1 * cost_2$). The CPU cycles saved would be:

$$S_{WinDrop} = \mathcal{R} * (sel_1 * sel_2 * p * cost_3 + (1 - sel_{w1}) * cost_1 + (1 - sel_{w1}) * sel_1 * cost_2 - cost_{WinDrop} - cost_{fcheck} - sel_{w2} * cost_{copy} - (\frac{sel_{w1}}{\delta} + sel_{w2}) * cost_{wcheck})$$

If we compare S_{RDrop} with $S_{WinDrop}$, we see that $S_{WinDrop}$ has two additional savings terms: it saves from the aggregate operator's cost ($cost_2$) as well as from the first subnetwork's cost ($cost_1$) with an amount determined by sel_{w1} (as a result of early drops). On the other hand, there are three additional cost terms for handling the flags introduced by *WinDrop*. We expect these costs to be much smaller than the savings of *WinDrop*. We experimentally show the processing overhead of *WinDrop* in Section 4.11.

Let us now briefly show how sel_{w1} and sel_{w2} can be estimated. For simplicity, we will assume a stream with one tuple per time value. We drop windows in batches of size \mathcal{B} . By definition, each drop-batch must be preceded and followed by at least one keep-window. The total number of tuples in a batch is $(\mathcal{B} - 1) * \delta + \omega$ (see Figure 4.6). Given a drop-batch, $2 * (\omega - \delta)$ of its tuples overlap with the preceding and the following keep-windows, therefore the number of tuples that belong only to the drop-batch is $(\mathcal{B} + 1) * \delta - \omega$. These are the tuples that can be early-dropped (assuming that $\mathcal{B} \geq \lfloor \frac{\omega}{\delta} \rfloor$). This many tuples out of a total of $(\mathcal{B} - 1) * \delta + \omega$ can be early-dropped and this would occur with probability p . Therefore, we end up with $sel_{w1} = 1 - p * \frac{(\mathcal{B} + 1) * \delta - \omega}{(\mathcal{B} - 1) * \delta + \omega}$ of *WinDrop*'s output tuples being kept as normal tuples. Furthermore, one tuple out of every δ tuples may have to be retained as a fake tuple since it carries a 0 window specification. Thus, $\lfloor \frac{(\mathcal{B} + 1) * \delta - \omega}{\delta} \rfloor = (\mathcal{B} + 1) - \lfloor \frac{\omega}{\delta} \rfloor$ out of $(\mathcal{B} + 1) * \delta - \omega$ will be additionally kept with probability p . Therefore, we end up with $sel_{w2} = p * \frac{(\mathcal{B} + 1) - \lfloor \frac{\omega}{\delta} \rfloor}{(\mathcal{B} + 1) * \delta - \omega}$.

4.10 Extensions

4.10.1 Multiple Groups

An aggregate can be defined to operate on groups formed by common values of one or more of its input attributes as specified by its group-by attribute \mathcal{G} . In this case, windows are formed separately

within each group. In this case, *WinDrop* operator must make window keep/drop decisions on a per-group basis. *WinDrop* uses the same ω , δ , \mathcal{T} parameters on all groups. However, we allow p and \mathcal{B} parameters to be assigned differently for each group. First, groups may have different QoS requirements. For example, consider an aggregate which computes the average price for the last 1 hour’s stock prices grouped by company. The loss-tolerance for Company A tuples may be smaller than for Company B tuples. Additionally, an application may not want to miss more than 5 windows in a row for A, whereas it may tolerate missing up to 10 consecutive windows for B. Hence, we handle this kind of cases by using a different \mathcal{B} parameter for each aggregate group (stored as a drop batch size array). Second, groups may be experiencing different arrival rates, thereby contributing differently to the CPU load. Therefore, it may be preferable to shed different amounts of load from each group. Hence, we handle such cases by using a different p parameter for each aggregate group (stored as a drop probability array).

4.10.2 Count-based Windows

Although the fan-out arrangement rule in Table 4.2 directly applies to aggregation queries with count-based windows, a modification is required to the pipeline arrangement rule. The window size parameter for *WinDrop* across a pipeline of aggregates must be modified as $\omega = \omega_k * \prod_{i=1}^{k-1} \delta_i + \sum_{i=1}^{k-1} (extent(A_i) * \prod_{j=1}^{i-1} \delta_j)$, and the window slide parameter for *WinDrop* must be modified as $\delta = \prod_{i=1}^k \delta_i$ (assuming that $\prod_{i=1}^0 \delta_i = 1$).

There is an additional issue with count-based windows when there are certain types of other operators lying between a *WinDrop* and an aggregate. For example, if a Filter in-between these operators removed tuples from the stream, the downstream aggregate would count tuples in a different way than the upstream *WinDrop*. There is a similar counting problem when a Union that merges two streams into one lies between a *WinDrop* and an aggregate. In both cases, window integrity could be lost. As a result, with count-based windows, existence of other operators introduce a limitation on how far a *WinDrop* can be pushed upstream from an aggregate.

4.11 Performance Evaluation on Borealis

4.11.1 Experimental Setup

We implemented the window drop operator as part of the load shedder component of the Aurora/Borealis stream processing prototype system [7, 12]. We conducted our experiments on a single-node Borealis server, running on a Linux PC with an Athlon 64 2GHz processor. We created a basic set of benchmark queries as will be described in the following subsections. We used synthetic data to represent readings from a temperature sensor as (time, value) pairs. For our experiments, the data arrival rates and the query workload were more important than the actual values of the data workload. Thus, for our purposes, using synthetic data was sufficient.

We first compare our approach against the random drop alternative which can only provide the

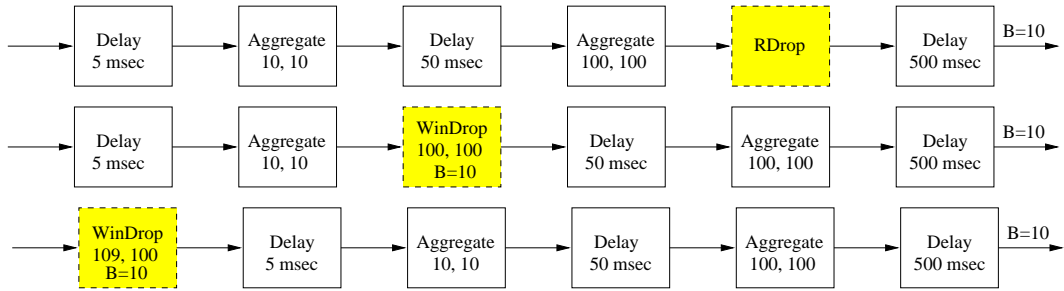


Figure 4.7: Drop insertion plans for the pipeline arrangement (RDrop, Nested1, and Nested2)

subset guarantee by applying tuple-based drops when placed downstream from all the aggregates in a query plan.² As part of this initial set of experiments, we also show the advantage of shedding load early in a query plan, by placing the window drop operator at various locations in a given plan and comparing the results. Then in Section 4.11.3, we examine the effect of window parameters, by varying window size and slide, and measuring the result degradation for various query plans. We also compare these experimental results against the analytical estimates of Section 4.9 to confirm their validity. Finally, in Section 4.11.4, we evaluate the processing overhead of our technique.

4.11.2 Basic Performance

First we will show the basic performance of window drop for both the pipeline and the fan-out (i.e., shared) query arrangements.

Nested Aggregates. For this experiment, we used the nested aggregation query shown in Figure 4.7, which is similar to the stock count example of Figure 4.1. There are two aggregate operators, each with tumbling windows of size 10 and 100 respectively, and both with count functions. We used a batch size of 10. We added delay operators before and after each aggregate to model other operators that may exist upstream and downstream from the aggregates. A delay operator simply withholds its input tuple for a specific amount of time (busy-waiting the CPU) before releasing it to its successor operator. A delay operator is essentially a convenient way to represent a query subplan with a certain CPU cost; its delay parameter provides a knob to easily adjust the query cost. In Figure 4.7, we used appropriate delay values to make different parts of the query equally costly.

The goal of this experiment is twofold. First, we show how much window drop degrades the result for handling a given level of excess load. Second, we compare it against two alternatives: one is a variation of our window drop approach, where a window drop is inserted in the middle of the query network; the other is random drop that is placed downstream from both of the aggregates. Figure 4.7 illustrates these three alternative drop insertion plans.

Figure 4.8 presents our result. The excess rate on the x-axis represents the percentage of the input rate that is over full capacity. The y-axis shows the drop rate (i.e. the fraction of the answer

²Note that we do not compare our approach against the relative error-based approaches (e.g., [17]) since there is no fair way to do this when error models are different.

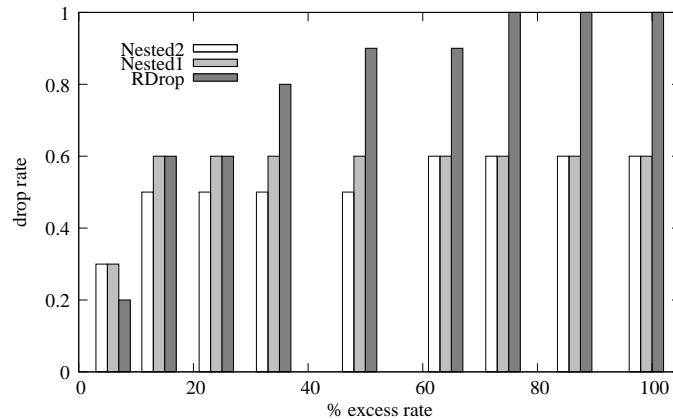


Figure 4.8: Comparing alternatives (pipeline)

that is missing from the result). **RDrop** represents random drop, **Nested1** corresponds to window drop inserted in the middle, and **Nested2** is for window drop placed at the input. At relatively low input rates, RDrop shows comparable performance to window drop approaches. However, as the rate gets higher, both Nested1 and Nested2 scale far better than the RDrop approach. In fact, RDrop stops delivering any results once the excess load gets beyond 65%. Nested2 either results in equal or smaller degradation in the answer compared to Nested1 at all load levels.

As a result, window drop is effective in handling system overload. It scales well with increasing input rate and outperforms the random drop alternative. Note that the RDrop case is all that would have been allowed by our previous work [93], since one could not move drops past aggregates. Placing the window drop further upstream in a nested aggregation query significantly improves the result quality, as more load can be saved earlier in the query, which reduces the total percentage of the data that needs to be shed.

Shared Query Plans. We repeated the previous experiment on a shared query plan. We used a fan-out arrangement with two aggregate queries as shown in Figure 4.9. The figure plots the three alternative load shedding plans that we compared. **Shared_WinDrop** is when window drop is placed at the earliest point in the query plan, **Split_WinDrop** is when each query has a separate window drop placed after the split point, and **Split_RDrop** is when we apply tuple-based random load shedding downstream from the aggregates. The parameters of the window drops are appropriately assigned based on the rules in Table 4.2.

Figure 4.10 presents our result. The y-axis shows the total drop rate for both of the queries, when the system experiences a certain level of excess load. Similar to our earlier result, shedding load at the earliest possible point in the query plan provides the smallest drop rate, and hence, the highest result quality. Again, the window drop operator enables pushing drops beyond aggregates and split points in a query plan, reducing quality degradation without sacrificing the subset guarantee.

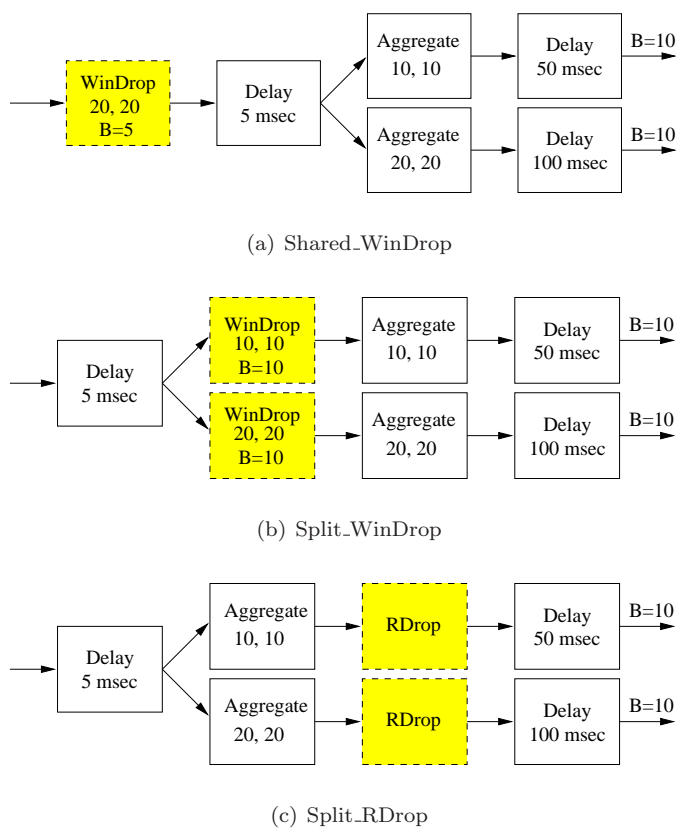


Figure 4.9: Drop insertion plans for the fan-out arrangement

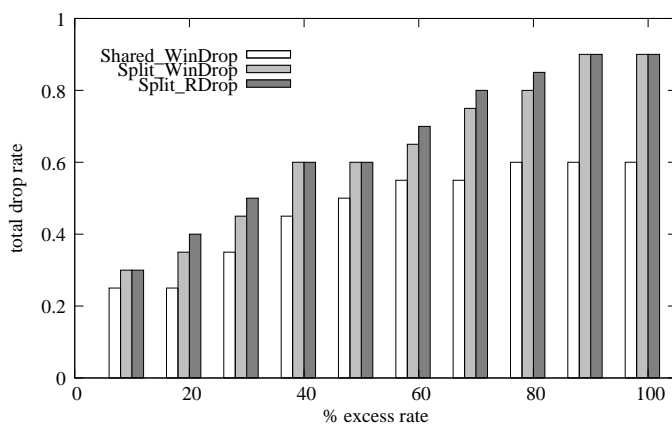


Figure 4.10: Comparing alternatives (fan-out)

4.11.3 Effect of Window Parameters

Next we investigate the effect of window parameters on window drop performance. We used a query with one aggregate operator with a count function as shown in Figure 4.11. We again added delay

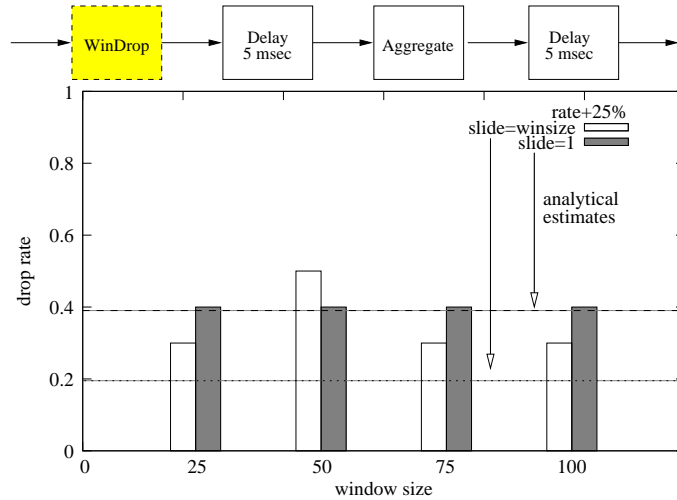
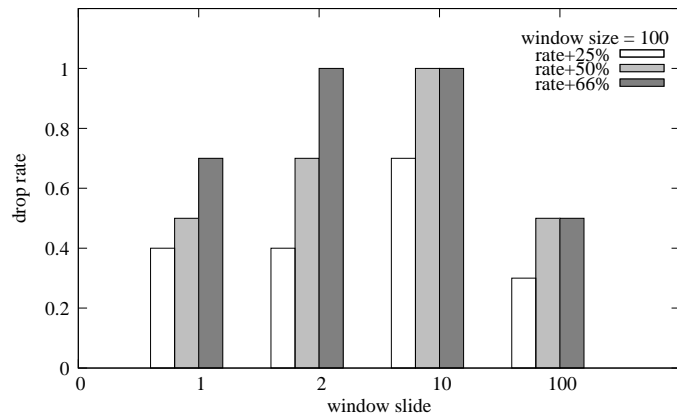


Figure 4.11: Effect of window size

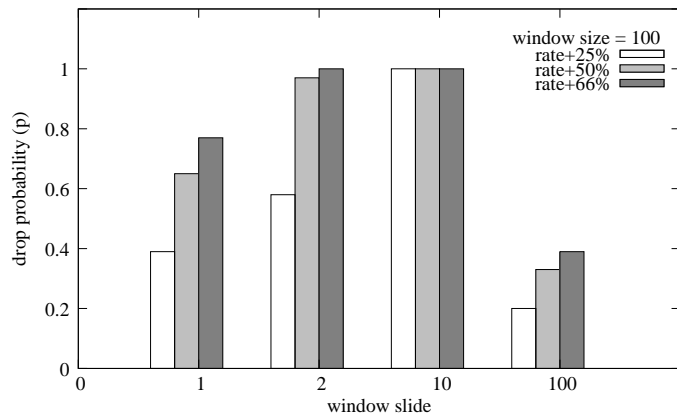
operators of 5 milliseconds each, before and after the aggregate, and set the batch size to 10.

The bar chart in Figure 4.11 shows the effect of window size on drops. An input rate that is 25% faster than the rate the system can handle at full capacity is fed into the query. For each window size, we measure the fraction of tuples that must be dropped to bring the system load below the capacity. We take these measurements for a window that slides by 1 (slowly sliding window) and for a window that slides by the window size (tumbling window). In most cases, the drop rates came out to be lower for the tumbling window case. This is because window drop can achieve early drops in this case. A second observation is that drop rates stay almost fixed as the window size increases. Interestingly, the formulas presented earlier in Section 4.9 also suggest that load should be independent of the aggregate window size when $\delta = 1$ and $\delta = \omega$ (see sel_{w1} and sel_{w2}). We also measured average operator costs and plugged them into our formulas. The formulas estimate drop rates to be 0.2 and 0.39 for the tumbling and the sliding window case, respectively (shown with dotted lines). The latter case is experimentally confirmed in Figure 4.11. However, our formulas underestimate the drop rate for the tumbling window case. In this case, the aggregate operator has a very small selectivity. It closes a window and produces an output once every ω tuples, at which point the downstream delay operator is scheduled. Our analysis models the average case behavior and fails to capture cases where internal load variations may occur due to changes in operator scheduling frequency.

Figure 4.12 details the effect of window slide on window drop performance: Figure 4.12(a) shows our experimental result and Figure 4.12(b) plots the analytical estimates of Section 4.9. A window size of 100 with four different slide values is used. A slide value of 1 corresponds to a large number of simultaneously open windows, therefore, a high degree of window overlap, and high aggregate selectivity. A slide value of 100 corresponds to one open window at a time, zero window overlap, and low aggregate selectivity, providing more opportunity for early drops. As we increase the window



(a) Experimental result



(b) Analytical estimate (from Section 4.9)

Figure 4.12: Effect of window slide

slide, the number of saved CPU cycles upstream from the aggregate increases (due to early drops) while the number to be saved downstream from the aggregate decreases (due to low aggregate selectivity). The required drop amount first increases, but then starts decreasing due to additional savings from early drops. Note that this decrease is observed when slide gets above 10 (i.e., when $\lfloor \frac{w}{\delta} \rfloor \leq \mathcal{B}$). Window drop shows the best advantage as the degree of window overlap decreases. We continue to observe this effect as excess load increases. Our analysis, plotted in Figure 4.12(b), captures the general behavior very well, but as window slide grows, it shows a departure from the measured results, for the same reason as explained in the previous paragraph.

As a brief note, we also compared our window drop with a random drop inserted after the aggregate. For slide=1, the performance is similar (no early drops). For slide=100, random drop fails to remove the overload, even at rate+25%. Thus, in the worst case where there is a very high degree of window overlap and zero opportunity for early drops, window drop behaves similar to the

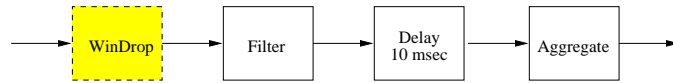


Figure 4.13: Filtered aggregation query

window size	selectivity=1.0	selectivity=0.5
25	0.99	0.96
50	0.99	0.98
75	1.0	0.98
100	1.0	1.0

Table 4.4: Throughput ratio ($\text{WinDrop}(p=0)/\text{NoDrop}$)

random drop. As slide grows, window drop achieves a clear advantage.

4.11.4 Processing Overhead

Next we evaluate the overhead of adding window drop into query plans. This overhead has several potential sources: (1) an additional operator to be scheduled in the query plan, (2) other operators interpreting window specifications, (3) fake tuples.

In this experiment, we used the query layout shown in Figure 4.13. We varied the predicate of the filter to obtain various selectivity values. We used a tumbling window whose window size is also varied. Table 4.4 shows the ratio of throughput values for a query that contains a window drop that does not drop anything ($p=0$) and for the case when no window drop is present. We ran each query for a minute, at a rate that is 50% higher than the system capacity. Since no tuples are dropped in either case, the reduction (if any) in the number of tuples produced with window drop must be due to the additional processing overhead. First, Table 4.4 shows that in general, the overhead is low. Second, as the window size increases, the overhead decreases. This is due to the fact that the window drop marks tuples less frequently. Third, for lower selectivity, the overhead seems to be higher. This result accounts for the effect of handling fake tuples. As we mentioned earlier in Section 4.6, filter generates fake tuples when its predicate evaluates to false but the tuple has to be retained if it is carrying a non-negative window specification. The chance of generating fake tuples increases as the filter selectivity decreases. This may further lead to an increased overhead of processing fake tuples in the downstream query network. As shown in Table 4.4, we see only a slight increase in overhead when the filter selectivity is lowered to 0.5.

4.12 Chapter Summary

In this chapter, we have shown a window-aware load shedding technique that deals with sliding window aggregate operators. Moreover, we have done this in a way that preserves the subset result

guarantee. Our techniques also support load shedding in query networks in which aggregates can be arbitrarily nested. We believe that this is very important since, in our experience with the Aurora/Borealis system, user-defined aggregates have been used extensively in practice for many tasks that involve operating on a subsequence of tuples. Thus, they occur quite frequently in the interior of query networks. Our contribution is the ability to handle aggregates in a very general way that is consistent with a subset-based error model.

We have shown that, as is expected, with the added ability to push drops past aggregates, we can recover more load early; thereby, regaining the required CPU cycles while minimizing the total utility loss. By focusing on dropping windows, we can better control the propagation of error through the downstream network.

Some of the complexity in our solution is a result of the simple flat data model. For example, not being able to denote windows as sets of tuples results in a tuple marking scheme. However, a simple model simplifies implementation and allows for faster execution in the general case.

Chapter 5

Distributed Load Shedding

5.1 Overview

In this chapter, we consider the overload management problem in the context of distributed stream processing systems. In this environment, large numbers of continuous queries are distributed onto multiple servers. Distributed stream processing systems (e.g., [6, 13, 76, 84]) are important because distribution is the principle way that we can scale our systems to cope with very high stream rates. An example of an application in which this is a crucial issue is Internet-scale dissemination in which content from many sources is aggregated and distributed to an audience of many millions of listeners (e.g., [37, 74]). Also, many applications are naturally distributed. An example of this kind of application is distributed sensor networks in which the processing elements are the sensors themselves (e.g., [67]).

Data streams can arrive in bursts which can have a negative effect on result quality (e.g., throughput, latency). Provisioning the system for worst-case load is in general not economically sensible. On the other hand, bursts in data rates may create bottlenecks at some points along the server chain. Bottlenecks may arise due to excessive demand on processing power at the servers or bandwidth shortage at the shared physical network that connects these servers. Bottlenecks slow down processing and network transmission, and cause delayed outputs. Load shedding in this case aims at dropping tuples at certain points along the server chain to reduce load. Unlike TCP congestion control, there are no retransmissions and dropped tuples are lost forever. This will have a negative effect on the quality of the results delivered at the query outputs. The main goal is to minimize the quality degradation.

In the previous chapters of this thesis, we focused on the single-server load shedding problem. This chapter studies distributed load shedding. In distributed stream processing systems, each node acts like a workload generator for its downstream nodes. Therefore, resource management decisions at any node will affect the characteristics of the workload received by its children. Because of this load dependency between nodes, a given node must figure out the effect of its load shedding actions

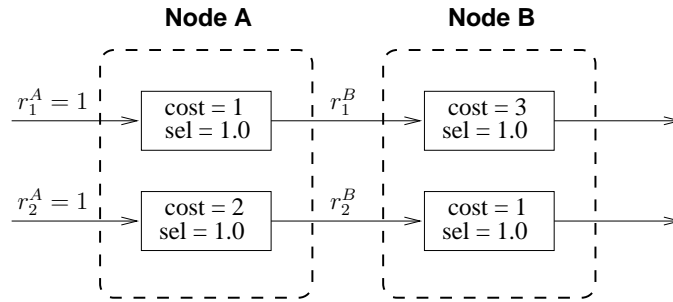


Figure 5.1: Two continuous queries distributed onto two servers

Plan	Reduced rates at A	A.load	A.throughput	B.load	B.throughput	Result
0	1, 1	3	1/3, 1/3	4/3	1/4, 1/4	originally, both nodes are overloaded
1	1/3, 1/3	1	1/3, 1/3	4/3	1/4, 1/4	B is still overloaded
2	1, 0	1	1, 0	3	1/3, 0	optimal plan for A, but increases B.load
3	0, 1/2	1	0, 1/2	1/2	0, 1/2	both nodes ok, but not optimal
4	1/5, 2/5	1	1/5, 2/5	1	1/5, 2/5	optimal

Table 5.1: Alternate load shedding plans for node A of Figure 5.1

on the load levels of its descendant nodes. Load shedding actions at all nodes along the chain will collectively determine the quality degradation at the outputs. This makes the problem more challenging than its centralized counterpart. We will illustrate our point with a simple example.

5.1.1 Motivating Example

Consider the simple query network in Figure 5.1 with two queries that are distributed onto two processing nodes, A and B. Each small box represents a subquery with a certain cost and selectivity. Cost reflects the CPU time that it takes for one tuple to be processed by the subquery, and selectivity represents the ratio of the number of output tuples to the number of input tuples. Both inputs arrive at the rate of 1 tuple per second. Therefore, the total processing load on node A is 3 and the total processing load on node B is 4/3, i.e., both of the nodes are above the capacity limit of 1. Potentially each node can reduce load at its inputs by dropping tuples to avoid overload. Let's consider node A. Table 5.1 shows various ways that A can reduce its input rates and the consequences of this in terms of the load at both A and B, as well as the throughput observed at the query outputs. Note that we are assuming a fair scheduler that allocates CPU cycles among the subqueries in a round-robin

fashion. In all of these plans, A can reduce its load to the capacity limit. However, the effect of each plan on B can be very different. In plan 1, B stays at the same overload level. In plan 2, B's load increases to more than twice its original load. In plan 3, B's overload problem is also resolved, but throughput is low. There is a better plan which removes overload from both A and B, while delivering the highest total throughput (plan 4). However, node A can only implement this plan if it knows about the load constraints of B. From A's point of view, the best local plan is plan 2. This simple example clearly shows that nodes must coordinate in their load shedding decisions to be able to achieve high-quality query results.

5.1.2 Design Goals

An effective load shedding solution for a distributed stream processing system should have the following properties:

- **Fast reactivity to load.** The main goal of load shedding is to maintain low-latency query results even during load spikes. These situations need to be detected and addressed as soon as possible. To achieve this, the load shedding algorithm must be light-weight and the load shedding plans must be efficiently deployed. In a distributed setting, having even a single overloaded node in the query pipeline is sufficient to boost latency at the outputs. Therefore, the solution must be equally attentive to each individual node's problem.
- **Global control on output quality.** While maintaining low-latency results, the algorithm must also ensure that the quality of the query answers does not arbitrarily degrade. As evidenced by our example above, a load shedding plan that can deliver very high-quality results at a node locally, does not necessarily achieve similar quality at the global level. What eventually matters to the applications is the quality of the results that they observe at the query end-points. Therefore, regardless of where load is being shed in the node chain, the resulting effect at the query end-points must be kept under control.
- **Scalability.** The load shedding algorithm should scale well with certain important performance factors. These include the number of server nodes, the number of input streams, and the amount of branching that may exist along the queries. Branching is important since it causes a stream to split and replicate its contents into multiple streams, creating more load in the system. Also, these replica streams may serve multiple different end-point applications. Therefore, shedding load before or after a split point makes a difference in output quality.
- **Adaptivity to dynamic changes.** The distributed stream processing environment can be highly dynamic. The processing requirements can change not only due to changing input rates, but also due to changing operator selectivities, placement, and so on. The load shedding algorithm should efficiently adapt to such changes, possibly making small incremental adjustments.

5.1.3 Our Solution Approach

A load shedder will insert *drop operators* on selected arcs in order to reduce the load to a manageable level. We call a set of drop operators with given drop levels at specific arcs a *load shedding plan*. In practice, a load shedder cannot spend large amounts of time determining the best plan at runtime, when the system is already under duress. Instead, in this and in our previous work, we run an off-line algorithm first to build a set of plans that can be quickly invoked for different combinations of input load.

For the distributed case, the simplest way to run the off-line algorithm is to have each node send its requirements to a central site at which the coordinated load shedding plans are built. As we shall see, this allows us to formalize distributed load shedding as a linear programming problem which can be solved with a standard optimization algorithm.

Unfortunately, the centralized case does not scale as the number of nodes grows very large. Moreover, since the solver can take a long time to run, it is not very useful as a tool for replanning when the environment is highly dynamic. A dynamic environment is one in which the selectivities, processing costs, and network topology are likely to change often. In these cases, the previously computed load shedding plans will likely not be desirable, therefore, a new set of plans must be constructed.

For these very large and potentially dynamic environments, we describe a distributed algorithm that does not require the high-level of communication that the centralized case demands. We also show how this distributed algorithm can incrementally compute changes to the previous plan in response to local changes in the environment, thereby making it more responsive than a centralized version.

5.1.4 Assumptions

We study the distributed load shedding problem in the context of our Borealis distributed stream processing prototype system. As also described earlier in Chapter 2, Borealis accepts a collection of continuous queries, represents them as one large network of query operators, and distributes the processing of these queries across multiple server nodes. Each server runs Aurora as its underlying query processing engine that is responsible for executing its share of the global query network. In addition to this basic distributed stream processing functionality, the system also provides various optimization, high availability, and approximation capabilities, including load shedding.

For the purposes of this chapter, we treat the query operators as black boxes with certain cost and selectivity properties. The costs and selectivities are obtained by observing the running system over time. As such, they are statistical in nature and combining them produces approximations. Our operator-specific load shedding techniques that are presented in the previous chapters are complementary to this work.

Bottlenecks in a distributed setting may arise both due to the lack of required processing power and also due to bandwidth limitations. In line with the previous chapters of this thesis, we will

continue to limit our scope to the CPU problem.

There can be various quality metrics defined for the query outputs. The QoS model we used in Chapter 3 is an example. In this chapter, we will focus on total query throughput as the main quality metric (also referred to as quality score) to maximize.

Finally, in this work, we focus on tree-based server topologies, while the query network itself can have operator splits and merges.

5.1.5 Chapter Outline

The rest of this chapter is outlined as follows. In Section 5.2, we present a precise formulation of the distributed load shedding problem. In Section 5.3, we discuss the architectural aspects of our solution to this problem. Our solver-based centralized approach is detailed in Section 5.4, while Section 5.5 provides the details for our distributed alternative. In Section 5.7, we show experimental evidence that our techniques are viable. Finally, we conclude the chapter with Section 5.8.

5.2 The Distributed Load Shedding Problem

5.2.1 Basic Formulation

We define the distributed load shedding problem as a linear optimization problem as follows. Consider a query diagram as shown in Figure 5.2, that spans N nodes, each with a fixed dedicated CPU capacity ζ_i , $0 < i \leq N$. Assume that we designate D arcs on this diagram as drop locations where drop boxes can be inserted. As we showed earlier in Chapter 3, drop locations are the input arcs of the query diagram, and the output arcs of splitting operators. In Figure 5.2, they appear only at the input arcs since there are no operator splits. For a drop location d_j on arc j , $0 < j \leq D$, let $c_{i,j}$ represent the total CPU time required at node i , to process one tuple that is coming from arc j , and similarly, let $s_{i,j}$ represent the overall selectivity of the processing that is performed at node i on tuples that are coming from arc j . Assume that r_j represents the data rate on arc j , and s_j represents the overall selectivity of the query from arc j all the way down to the query outputs (i.e., $s_j = \prod_{i=1}^N s_{i,j}$). Lastly, we denote the partial selectivity from arc j down to the inputs of node i by s_j^i (i.e., for $1 < n \leq N$, $s_j^n = \prod_{i=1}^{n-1} s_{i,j}$, and for $n = 1$, $s_j^1 = 1.0$).

Our goal is to find x_j , i.e., the fraction of tuples to be kept at drop location d_j (or drop selectivity at d_j), such that for all nodes i , $0 < i \leq N$:

$$\sum_{j=1}^D r_j \times x_j \times s_j^i \times c_{i,j} \leq \zeta_i \quad (5.1)$$

$$0 \leq x_j \leq 1 \quad (5.2)$$

$$\sum_{j=1}^D r_j \times x_j \times s_j \quad \text{is maximized.} \quad (5.3)$$

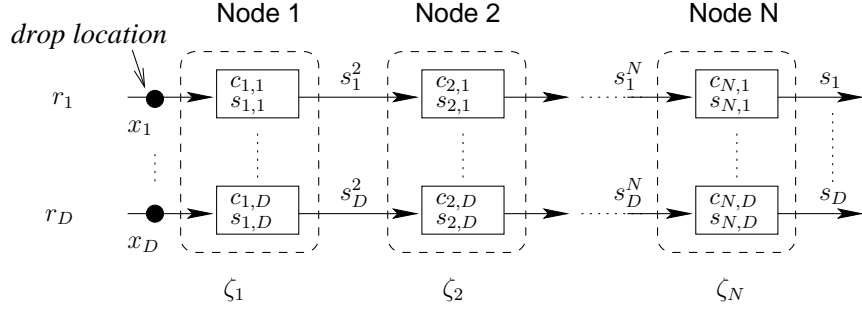


Figure 5.2: Linear query diagram

r_j	rate on arc j
x_j	drop selectivity on arc j
$c_{i,j}$	cost of processing tuples at node i coming from arc j
$s_{i,j}$	selectivity of processing tuples at node i coming from arc j
s_j^i	partial selectivity of processing tuples coming from arc j down to node i
s_j	total selectivity of processing tuples coming from arc j down to the outputs
ζ_i	fraction of the dedicated CPU capacity at node i

Table 5.2: Linear query diagram notation

This optimization problem can be expressed as a linear program as follows. We have a set of N linear constraints on processing load of the nodes, which we call *load constraints*, as given by (5.1). We have a set of D variables x_i on *drop selectivities*, which can range in $0 \leq x_i \leq 1$, as given by (5.2). Our goal is to maximize a linear objective function that represents the *total throughput* as given by (5.3), subject to the set of constraints (i.e., (1) and (2)). In other words, we want to make assignments to variables that represent the fraction of tuples to be kept at the drop locations such that load constraints for all nodes are satisfied and the total throughput at the query end-points is maximized.

5.2.2 Operator Splits and Merges

In this section, we show how we can extend the basic problem formulation of Section 5.2.1 to query diagrams with operator splits and operator merges.

Operator Splits

We have operator splits in a query network when output from an operator fans out to multiple downstream operators which further lead to separate query outputs. Note that if split branches merge downstream in the diagram, we do not consider this as a split case. Split is an interesting case because shedding load upstream or downstream from a split may result in different quality degradation at the outputs due to sharing. Therefore, all output arcs of a split constitute potential drop locations [93].

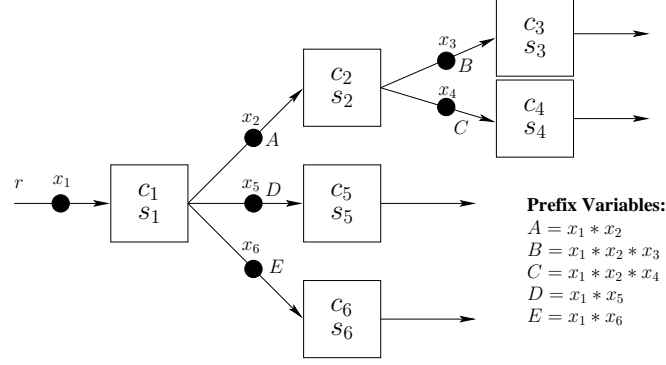


Figure 5.3: Two levels of operator splits

We illustrate the problem formulation for operator splits on a simple single-node example with two levels of splits shown in Figure 5.3. Let x_i denote the drop selectivity on a particular drop location, and c_i and s_i denote the processing cost and selectivity of a given operator, respectively. The variables shown with capital letters denote the total drop selectivity at various points in the query network. We can formulate the optimization problem for our example as follows:

$$r(c_1x_1 + s_1c_2A + s_1s_2c_3B + s_1s_2c_4C + s_1c_5D + s_1c_6E) \leq \zeta \quad (5.4)$$

$$0 \leq x_1 \leq 1 \quad (5.5)$$

$$0 \leq A, D, E \leq x_1 \quad (5.6)$$

$$0 \leq B, C \leq A \quad (5.7)$$

$$\{r(s_1s_2s_3B + s_1s_2s_4C + s_1s_5D + s_1s_6E)\} \text{ is maximized.} \quad (5.8)$$

We create a variable for each path prefix (e.g., x_1 , $A = x_1x_2$, and $B = x_1x_2x_3$), which we call *prefix variables*. We express our load constraints in terms of the prefix variables, as in (5.4). We define constraints on each prefix variable of length k such that its value is constrained between 0 and the values of its matching prefix variables of length $k - 1$ (e.g., $0 \leq x_1x_2x_3 \leq x_1x_2$). We express our objective function in terms of the longest prefix variables on each path (e.g., $x_1x_2x_3$, $x_1x_2x_4$, x_1x_5 , and x_1x_6). Then we solve our problem for the prefix variables. Finally, we plug in the values of the prefix variables to obtain values of the original variables. In our example, we would solve for the prefix variables $\{x_1, A, B, C, D, E\}$ to obtain the original variables $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ as follows: $x_1 = x_1, x_2 = \frac{A}{x_1}, x_3 = \frac{B}{x_1x_2}, x_4 = \frac{C}{x_1x_2}, x_5 = \frac{D}{x_1}, x_6 = \frac{E}{x_1}$.

Operator Merges

Two streams can merge on a query diagram via the Union operator. The output rate of a Union operator will be the result of the relative rates into the Union. Consider the example query segment in Figure 5.4. Let c_i denote the box costs and s_i denote the box selectivities. $\frac{r_1s_1}{r_1s_1+r_2s_2}$ of the

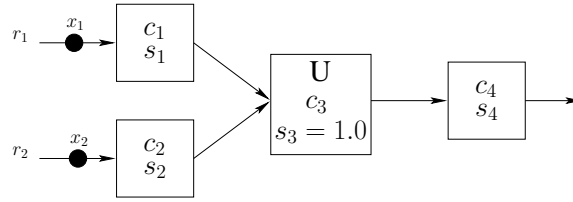


Figure 5.4: Merging two streams via Union

query output comes from stream 1 and $\frac{r_2 s_2}{r_1 s_1 + r_2 s_2}$ of it comes from stream 2. Therefore, we may want to shed different amounts from these streams (e.g., shedding more from the stream whose contribution to the query output is smaller would not affect the total throughput as much as the other stream). Furthermore, top and bottom paths of the query may have different load factors, resulting in different amount of gain from these paths. Thus, it may again be more advantageous to shed more from one of the streams. We can formulate the optimization problem for this merge example as follows:

$$r_1(c_1 + s_1 c_3 + s_1 c_4)x_1 + r_2(c_2 + s_2 c_3 + s_2 c_4)x_2 \leq \zeta \quad (5.9)$$

$$0 \leq x_1, x_2 \leq 1 \quad (5.10)$$

$$\{r_1(s_1 s_4)x_1 + r_2(s_2 s_4)x_2\} \text{ is maximized.} \quad (5.11)$$

5.3 Architectural Overview

In the previous section, we showed how to formulate one instance of the distributed load shedding problem for a specific observation of the input rates. When we solve such a problem instance, we obtain a *load shedding plan*. This plan essentially shows where drop operators should be inserted in the query network, and what the drop selectivity should be for each of them. We will describe how we generate a load shedding plan in Sections 5.4 and 5.5. In this section, we will discuss the architectural aspects of our solution.

During the course of system execution, input rates and hence the load levels on the servers will vary. Therefore, there is a need to continuously monitor the load in the system and react to it using the appropriate load shedding plan. We identified four fundamental phases in the distributed load shedding process:

1. **Advance Planning:** This is the precomputation phase. In this phase, the system prepares itself for potential overload conditions based on available metadata about the system. The idea is to do as much of the work as possible in advance so that the system can react to overload fast and in a light-weight manner. More specifically, in this phase, we generate a series of load shedding plans together with an indication of the conditions under which each of these plans should be used.

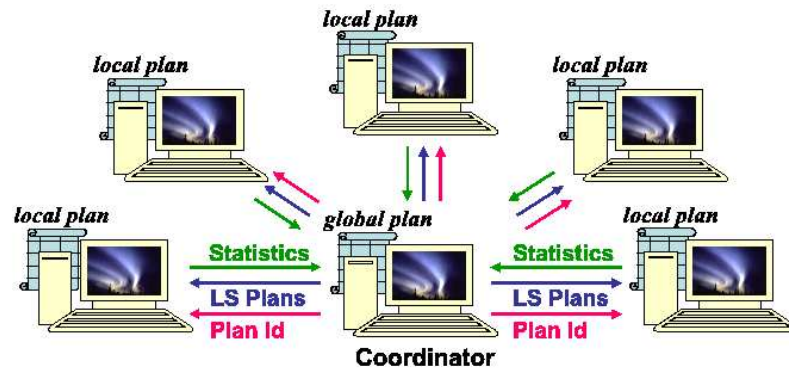


Figure 5.5: Centralized approach

2. **Load Monitoring:** As the system runs, we continuously watch the system load by measuring the input rates and estimating the load level on each server accordingly.
3. **Plan Selection:** This is the decision making phase. If an important change in system load is detected during the monitoring phase, then we decide what action to take. This is achieved by selecting the right load shedding plan from the many computed during Advance Planning.
4. **Plan Implementation:** In this final phase, the selected plan is put into effect by changing the query network by inserting drops to overcome the current overload.

In a distributed stream processing environment, the Plan Implementation phase will always be performed at multiple servers in a distributed fashion. The first three phases however, can be performed in various ways. In this work, we study two general approaches, based on an architectural distinction regarding where these three phases should be performed:

1. **Centralized Approach.** In the centralized approach, Advance Planning, Load Monitoring, and Plan Selection are all performed at one central server, whereas Plan Implementation is performed at all servers in a distributed fashion. One of the servers in the system is designated as the “coordinator node” (see Figure 5.5). It contacts all the other participant nodes in order to collect their local system catalogs and statistics. By doing so, it obtains the global query network topology and the global statistics about various run-time elements in the system (e.g., operator cost and selectivity). Based on the collected global metadata, the coordinator generates a series of load shedding plans for other servers to apply under certain overload conditions. These plans are then uploaded onto the associated servers together with their plan id’s. Once the plans are precomputed and uploaded onto the nodes, the coordinator starts monitoring the input load. If an overload situation is detected, the coordinator selects the best plan to apply and sends the corresponding plan id to the other servers in order to trigger the distributed implementation of the selected plan.

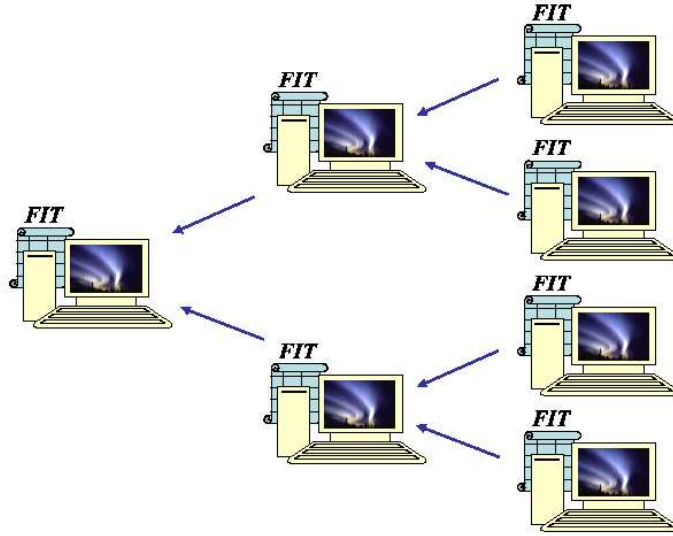


Figure 5.6: Distributed approach

2. **Distributed Approach.** In the distributed approach, all four phases of distributed load shedding are performed at all of the participating nodes in a cooperative fashion. There is no single point of control. Instead, the collective actions of all the servers result in a globally effective load shedding plan. In this chapter, we propose a distributed approach in which the needed coordination is achieved through metadata aggregation and propagation between neighboring nodes. As a result of this communication, each node identifies what makes a feasible input load for itself and its server subtree, and represents this information in a table that we call the *Feasible Input Table (FIT)* (see Figure 5.6). Then using its FIT, a node can shed load for itself and for its descendant nodes.

Table 5.3 summarizes the four phases of the distributed load shedding process and where each phase takes place for the two general classes of approaches. In this chapter, we focus on two specific approaches that fall under these two classes: (i) a solver-based approach, and (ii) a FIT-based approach. The solver-based approach is fundamentally a centralized approach that requires a coordinator, while the FIT-based approach is inherently designed as a distributed approach, but its centralized implementation is also available. Next, we present these two alternative approaches in detail.

5.4 Advance Planning with a Solver

As shown in Section 5.2, the distributed load shedding problem can be formulated as a linear optimization problem. Our solver-based advance planning technique is based on solving it using the

Phases	Centralized	Distributed
Advance Planning	coordinator	all
Load Monitoring	coordinator	all
Plan Selection	coordinator	all
Plan Implementation	all	all

Table 5.3: Four phases of distributed load shedding

Simplex Method of the GNU Linear Programming Kit (GLPK) [4].

Given the global query network topology, global statistics on operator costs and selectivities, and a specific observation of the input rates, the coordinator node first derives the necessary metadata. These include drop locations, path prefixes, partial and total path selectivities, load and rate factors. Basically, all variables described in Section 5.2 are computed to formulate a standard linear program (LP) with an objective function to maximize total system throughput, load constraints one for each node, and bounds and additional constraints on the variables representing drop selectivities for the drop locations. The examples in Section 5.2 illustrate this process in a clear way.

After the LP is fully constructed, we then call the Simplex Method of GLPK. The solution produced by GLPK consists of value assignments to all the prefix variables and the value of the objective function. From the prefix variables, we can obtain value assignments to all original variables, each representing the drop selectivity on a particular drop location on the query network.

The final step is to prepare the local load shedding plans. We go through the list of all drop locations. Each such location belongs to a specific server node. For each drop location d on node i , we create a drop operator with its drop rate determined by the drop selectivity assignment from the LP solution, and add that operator to the load shedding plan of node i . As a result, we obtain one load shedding plan for each node.

5.4.1 Region-Quadtrees-based Division and Indexing of the Input Rate Space

As part of the Advance Planning phase, we need to generate not one, but a series of load shedding plans to be able to handle any potential overload condition. In other words, we must map each infeasible (i.e. overloaded) point in the multi-dimensional input rate space to a load shedding plan that will make that point feasible for all the servers. Since a large query network may have many input streams, this space can be very large. It is not practical to exhaustively consider each possible infeasible rate combination in such a high-dimensional space. Instead, we need to carefully pick a sample of infeasible points that will represent the whole space in a controlled fashion, and only for that sample of points, will we generate load shedding plans. Then for the rest of the infeasible points that are not in the sample, we will use one of those generated load shedding plans. Note that for points not in the sample, we can not guarantee optimal plans. Instead, we relax the optimality requirement such that we allow plans whose output score must not deviate from the score of an

optimal plan by more than a specific percent error value. For example, given a percent error of $\epsilon = 10\%$, it would be acceptable to use an existing load shedding plan with a score ≥ 45 for an infeasible point p whose actual optimal score would be 50.

Our solution is based on dividing the multi-dimensional input rate space into a small number of subspaces such that all infeasible points in a given subspace can be handled using the same load shedding plan. This kind of an approach has two main benefits: (i) it makes the Advance Planning phase more efficient since it reduces the number of times that we need to construct and solve LP's; (ii) it makes the Plan Selection phase more efficient since it reduces the search space.

To divide our space into subspaces, we exploit an interesting property of the throughput metric. *The throughput score of an infeasible point p is always greater than or equal to the throughput score of another infeasible point q , when p is larger than or equal to q along all of the dimensions.* This is because for p , the LP solver gets to choose from a larger range of rate values and therefore has more degrees of freedom to find a solution with higher objective value. For example, given two infeasible points $p(x_1, y_1)$ and $q(x_2, y_2)$ in a 2-dimensional input rate space, if $x_1 \geq x_2$ and $y_1 \geq y_2$, then $p.score \geq q.score$. Based on this fact, given percent error ϵ , if $p.score - q.score \leq \frac{\epsilon}{100} * p.score$, then we can say that all infeasible points captured between p and q can use the same plan as the plan for point q with the guarantee that the score will never deviate from the optimal by more than ϵ percent.

In order to tackle the problem in a systematic way, we use a region-quadtrees-based approach to subdivide the space [81]. This approach also gives us the opportunity to build a quadtree-based index on top of our final subspaces which will make the Plan Selection phase much more efficient.

We assume that the maximum rate along each input dimension is given so that we know the bounds of the space that we are dealing with (e.g., (100, 100) in Figure 5.7(a)). We start by generating load shedding plans for the two extreme points of our input rate space and comparing their scores. Thus, we compare the score of the bottom-most point of this space (e.g., p in Figure 5.7(a)) with the score of its top-most point (e.g., q in Figure 5.7(a)). If the percent difference is above the given ϵ value, then we must further divide each dimension of this space into 2 (e.g., giving us 4 subspaces B, C, D, E in Figure 5.7(a)). Then we repeat the same procedure for each of these 4 subspaces. When we find that the score difference between two extreme points of a subspace is below the ϵ threshold, then we stop dividing that subspace any further. All infeasible points in a given rectangle must get assigned to the load shedding plan that corresponds to the bottom-most point of that rectangle. For example, in Figure 5.7(a), assume that the score of point r is within ϵ distance from the score of point q . Then, all infeasible points in the subspace between these two extreme points (i.e., the subspace E) can safely use the load shedding plan generated for point r . Assume s is such a point. In order for s to use the plan at r , s must additionally be mapped to r . This is accomplished by an additional load shedding step. In this case, we reduce the 60 to 50 by adding a drop of $\frac{50}{60}$, and similarly, we reduce the 75 to 50 with a drop of $\frac{50}{75}$. We can now use the plan for $r = (50, 50)$.

Note that, during the space division process, as we get closer to the origin, we may come across some feasible points. If we ever find that the top-most point of a subspace is already a feasible

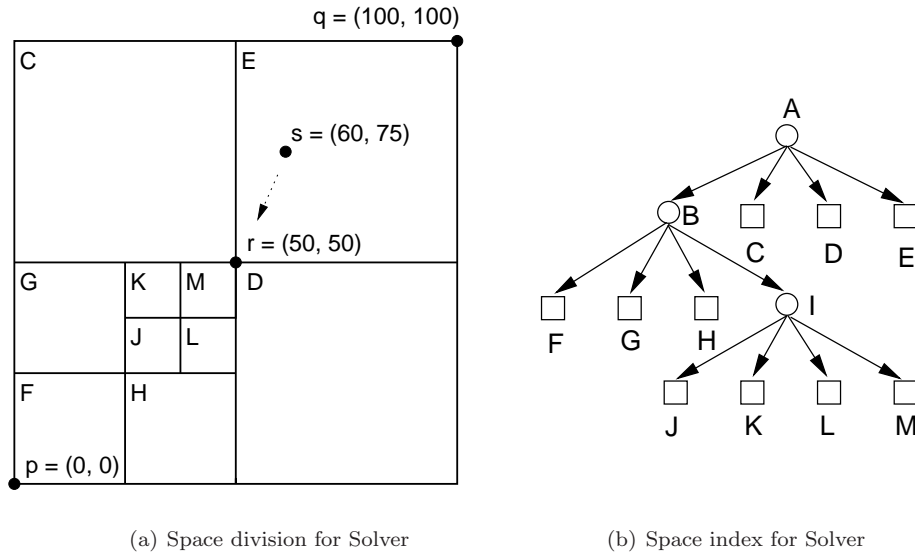


Figure 5.7: Region-Quadtree-based space division and index for Solver

point, it means that all points in that subspace must also be feasible. Therefore, there is no need to generate any load shedding plans for that subspace. Hence, we can stop dividing that subspace any further (e.g., subspace F).

At each iteration of the space division process, we produce a number of new subspaces. As an important implementation detail, we place these subspaces into a priority queue based on their percent error. Then at each step, we pick the subspace at the top of the queue with the highest error value to divide next. This allows us to stop the space division any time a given error threshold is met.

At the end of the space division process, we obtain a set of disjoint subspaces, each of which is mapped to a load shedding plan at a certain infeasible point in the space. During the Plan Selection phase, we will have to search through these subspaces in order to locate the one that contains a particular infeasible point. In order to make this search process more efficient, we further organize our subspaces into an index. The subspaces can be very conveniently placed into a quadtree-based index during the space division process described above. Figure 5.7(b) shows the index that corresponds to the space division of Figure 5.7(a).

5.4.2 Exploiting Workload Information

We have so far assumed that all of the input rate values in the multi-dimensional space have an even chance of occurrence. In this case, we must guarantee the same ϵ threshold for all of the input rate subspaces. However, if the input rate values are expected to follow an uneven distribution and if this distribution is known in advance, then we could exploit this information to make the Advance Planning phase much more efficient. More specifically, given an input rate subspace with

probability p and percent error of e , the expected error for this subspace would be $p * e$. We must then subdivide the input rate space until the sum of expected errors over all disjoint subspaces meets the ϵ threshold. Thus, instead of strictly satisfying the ϵ threshold for all subspaces, we ensure that on the average the expected maximum error will be below some threshold. Again in this case, we store the subspaces to be divided in a priority queue, but this time we rank them based on their expected errors.

5.5 Advance Planning with FIT

Our distributed approach to Advance Planning is based on informing upstream nodes about their children’s load constraints so that they can take load shedding actions that will influence their downstream nodes in the best possible way. To achieve this, each node periodically sends metadata in the form of a Feasible Input Table (FIT) to its parent nodes [94]. FIT is a summary of what a node expects in terms of its input rates and how that translates into a quality score at the downstream query end-points. When a node receives FITs from its children, it merges them into a single table. Furthermore, the parent maps the merged table from its outputs to its inputs and removes the entries that may be infeasible for itself. Finally, the parent propagates the resulting FIT to its own parents. This propagation continues until the input nodes receive the FIT for all their downstream nodes. Using its FIT, a node can shed load for itself and on behalf of its descendant nodes.

This approach follows a bottom-up traversal of the server tree starting from the leaf nodes. For each leaf node, a FIT indicating the feasible input rate combinations for this node is created. Feasible input rate combinations are the ones that either cause no overload in the system, or the ones that can be best handled using a local load shedding plan. Then FITs from leaf nodes with a common parent node are merged at the parent, while also eliminating the entries that are infeasible for the parent. This FIT merging and propagation continues until we arrive at the root nodes. The final outcome is one FIT for each server node, containing all feasible input rate combinations for all the nodes in its subtree, together with their quality scores.

5.5.1 Feasible Input Table (FIT)

For each node, we maintain a Feasible Input Table (FIT). Given a node with m inputs, the FIT for this node is a table with $m + 2$ columns. The first m columns represent the rates for the m inputs; the $(m + 1)$ th column represents the corresponding output quality score; and the last column represents the *complementary local load shedding plan* (if necessary) that must be used together with that input entry. FIT essentially represents the feasible input space for a node that will keep this node’s and all of its descendants’ CPU loads below their available capacities. It is a way for nodes to express their load expectations from their parents. For example, Table 5.4(a) illustrates the local FIT for Node A of Figure 5.1, Table 5.4(b) illustrates the local FIT for Node B, and Table 5.4(c) shows the merged FIT for both of the nodes that A should maintain. In this example, no local plans were necessary.

r_1^A	r_2^A	throughput
0	0	0
0	0.2	0.2
0	0.4	0.4
0.2	0	0.2
0.2	0.2	0.4
0.2	0.4	0.6
0.4	0	0.4
0.4	0.2	0.6
0.6	0	0.6
0.6	0.2	0.8
0.8	0	0.8
1.0	0	1.0

(a) Local FIT for Node A

r_1^B	r_2^B	throughput
0	0	0
0	0.2	0.2
0	0.4	0.4
0	0.6	0.6
0	0.8	0.8
0	1.0	1.0
0.2	0	0.2
0.2	0.2	0.4
0.2	0.4	0.6

(b) Local FIT for Node B

r_1^A	r_2^A	throughput
0	0	0
0	0.2	0.2
0	0.4	0.4
0.2	0	0.2
0.2	0.2	0.4
0.2	0.4	0.6

(c) Merged FIT for Node A and B

Table 5.4: FITs for example of Figure 5.1 (spread = 0.2)

As we will explain in more detail shortly, in some cases, a node may choose to handle some of its excess load using a local plan which is completely transparent to its parents. In such a case, a FIT entry may look like a feasible entry from the parent’s perspective, but is actually only feasible with a complementary local load shedding plan (i.e., the plan stored in the $(m + 2)$ th column). We refer such FIT entries as “feasible with a plan”. We will show another example in Section 5.5.2 where such complementary local plans are needed in FIT entries.

For each leaf node, we generate a FIT from scratch. For all other nodes, FIT is generated as a result of merge and propagation of FITs from downstream nodes. Next, we will describe the FIT generation algorithm for a leaf node.

5.5.2 FIT Generation

Leaf nodes generate their FIT based on the costs of the queries that they are assigned to execute. Consider such a node with a single query of cost c time units per tuple, which is fed by a single input i with rate r tuples per time unit. The node is overloaded unless $r * c \leq 1$. Thus, it can handle an input rate of up to $1/c$ before it becomes overloaded. Let’s call this the “rate limit” for i . Any input whose rate is smaller than its corresponding rate limit constitutes a feasible input for this node. Next, consider a node with multiple query paths from its inputs to its outputs. Assume that there are m inputs, input i having a rate limit of R_i . This rate limit is determined considering the total cost of the query paths fed by i in isolation from other paths (i.e., assuming that all other inputs have 0 rate). For each input i , we can then select a set of values between 0 and R_i . Each different combination of the selected rate values for the input dimensions can constitute a row in FIT if this combination is feasible. Furthermore, each such combination would lead to a certain total query throughput. This total will be the score for that feasible input combination.

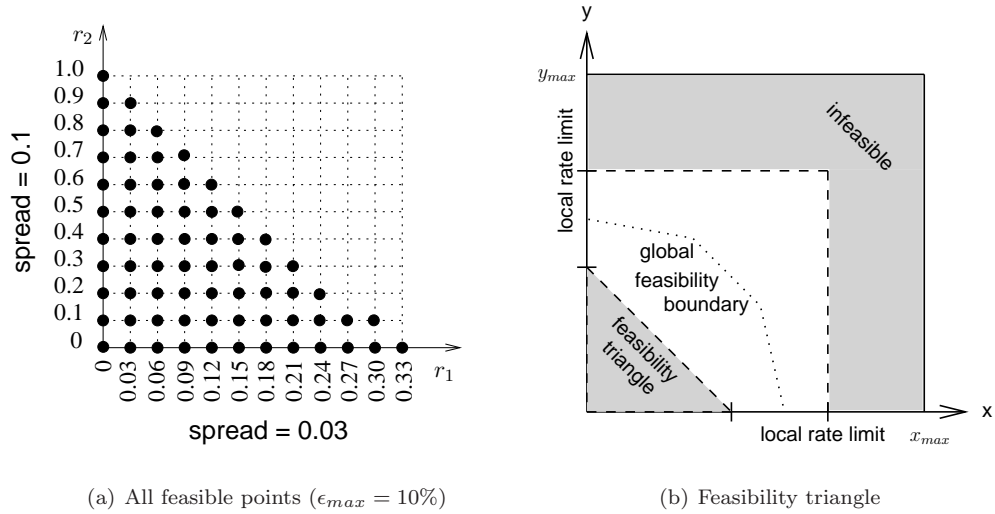


Figure 5.8: Choosing feasible points

Choosing FIT Points

An important issue in generating the FIT is the selection of the points to include in the table. After we determine the rate limit along each input dimension, we can include all possible feasible input combinations given some granularity. This granularity is modeled using *spread*. Spread for an input dimension determines how far apart the rate values should be selected for that dimension. The smaller the spread, the closer we can get to the optimal feasible point. On the other hand, smaller spread requires that we consider a larger number of feasible points, which is more costly to store and process.

As in the solver-based approach, we can exploit the ϵ error tolerance to determine the spread along each dimension. For example, consider node B in Figure 5.1. The rate upper bounds for this node are $r_1^B = 1/3$ and $r_2^B = 1$. Assume that ϵ_{max} is given to be 10%. Then spreads can be chosen as (approximately) $s_1^B = 10/100 * 1/3 = 0.03$, and $s_2^B = 10/100 * 1 = 0.1$. The set of feasible points is shown in Figure 5.8(a). Thus, the FIT for this node has 68 entries. The entry (0, 1.0) provides the highest total throughput score of 1.0.

In most cases, only a certain subset of the combinations are really interesting for us, as we will argue next. The main purpose of generating feasible points is to identify which rate points are acceptable for a node so that if an infeasible point is observed, we can map that infeasible point to an appropriate feasible point by load shedding. The goal should be to map to the point with the highest output score. As also explained in Section 5.4.1, given two feasible points p and q in the multi-dimensional input rate space, if p is greater than or equal to q along all of the dimensions, then $p.score \geq q.score$. Therefore, for an infeasible point r that is greater than or equal to both p and q along all of the dimensions, p should always be preferred over q . This means that, ideally, we are only looking for the feasible points which are on the outer boundary of the global feasibility

space. However, the FIT, which is initially generated at the leaf nodes, will change as it is merged and propagated upstream towards the root nodes. Thus, we will only be able to know the exact global feasibility boundary when the FIT reaches the root nodes. Therefore, at the leaf level, we must generate other feasible points than just the boundary points. Furthermore, we must make sure that we keep a representative sample so that by the time FIT reaches the root nodes, it can still accurately represent a good sample of the feasible points.

We first identify a subset of globally feasible points which will certainly not be needed in the FIT. We call such a subset the “feasibility triangle”. This triangle is computed by making a bottom-up pass from our leaf node towards the root nodes, at each node keeping track of the minimum rate limit. Next, we generate points that are locally feasible at our leaf node, excluding the ones that are in the triangle. To clarify, in Figure 5.8(b), we illustrate a 2-dimensional input rate space. The bottom triangle is the feasibility triangle whereas the top L-shaped region is the infeasible region. The area between these two (shown as the white region) is guaranteed to include the global feasibility boundary (shown with dotted lines) that we are eventually looking for. Hence, to be on the safe side, we initially generate FIT entries for all the points in this area (with some given granularity). Note that when a node has multiple children nodes, then children FITs must be merged at the parent node. In this case, the feasibility triangle must not be extracted from the FITs of its children since these points could still potentially combine with each other during the merge process.

In order to exploit the ϵ error tolerance, we select sample points along each input dimension with a certain spread value. Given ϵ , the spread for an input dimension i with rate limit R_i is determined as $R_i * \frac{\epsilon}{100}$.

Complementary Local Plan

As also mentioned earlier, the query network may include operators whose output may split to multiple paths. Each such path may have a different cost and a different path selectivity. Due to this difference, dropping from some branches may be more desirable. However, this is completely a local issue and need not be exposed to the parent. Instead, we allow a node to create FIT entries which are in fact not feasible and support these entries with complementary local load shedding plans that drop tuples at split branches. By doing this, the node can achieve higher output scores.

We will illustrate this idea with a simple example. Consider the query network in Figure 5.9. The input splits into two branches. The top branch saves 2 units per dropped tuple at the expense of 1 output tuple, whereas the bottom branch saves 5 units per dropped output tuple. Also, dropping from the input saves 8 units while losing 2 output tuples. Dropping from the bottom branch is clearly the most beneficial. Therefore, the node should drop completely from the bottom branch before dropping from its inputs. If the input rate is r , then $5 * r$ out of the total load of $8 * r$ should be handled locally. In other words, the node will be able to handle an additional load of $5 * r$ beyond its capacity based on local drops. Hence, the node must make sure that $8 * r \leq 1 + 5 * r$. Therefore, $3 * r \leq 1$. Note that now the upper value bound for r must be $R = 1/3$. We will generate so-called *feasible* points based on R . Also, for each feasible point satisfying $3 * r \leq 1$, if it is also satisfying

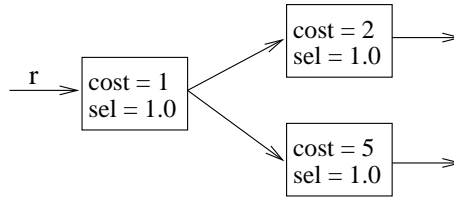


Figure 5.9: Splits to be supported by complementary plans

$8 * r \leq 1$, then no additional local drops are needed. Otherwise, we will create additional plans where some portion of the data on the bottom branch must be dropped locally. For example, given $r = 0.2$, it satisfies the extended load constraint ($3 * r \leq 1$), but it does not satisfy the original load constraint ($8 * r \leq 1$). We must additionally shed 60% of the load on the bottom branch. In this case, the total throughput score becomes 0.28. (The corresponding FIT entry would look like $\{0.2, 0.28, DropBox\{drop1, arc3, random_drop, 0.60\}\}$.) If we instead used the original constraint, then we would have to shed load to reduce to the nearest feasible point $r = 0.125$, which would give us the highest possible score of 0.25. In other words, by using the additional local plan, we can shed sufficient load and provide higher-quality output.

Given a point that is feasible with a local plan, we generate the plan as follows. We go through the sorted list of drop locations, each time picking the drop location with the smallest quality/load ratio. This resembles the “loss/gain ratio” idea that we introduced in Chapter 3. Each drop location can save up to a certain amount of load based on its load factor. Depending on our excess load amount, we should pick enough drop locations from the list that will in total save us the required amount of load. The plan will then consist of drop operators to be placed at the selected drop locations. The drop selectivity for these operators will all be 0 except the last one. The last one will have a drop selectivity which equals $1 - (\text{total excess load} - \text{total load saved so far}) / (\text{max load that can be saved by the last drop})$.

5.5.3 FIT Merge and Propagation

Assume two server nodes A and B as in Figure 5.1, where A is upstream from B. After we compute FIT for B, we propagate it upstream to A. The feasible points in B’s FIT are expressed in terms of B’s inputs which correspond to the rates at A’s outputs. To be able to propagate the FIT further upstream, we have to express B’s FIT in terms of A’s inputs.

Each input i of A follows a query path to produce a certain output. Along this path, the rate of i changes by a factor determined by the product of the operator selectivities (say sel_i). Therefore, given an output rate r , the corresponding input rate for i is $\frac{r}{sel_i}$. To obtain A’s FIT, we first apply this reverse-mapping to each row of B’s FIT; the corresponding score for each row stays the same. Then, we eliminate from the resulting FIT the entries which may be violating A’s load constraint.

If there is a split along the path from an input i to multiple outputs, and if all child branches of the split map to the same input rate value, then we just propagate that value as described

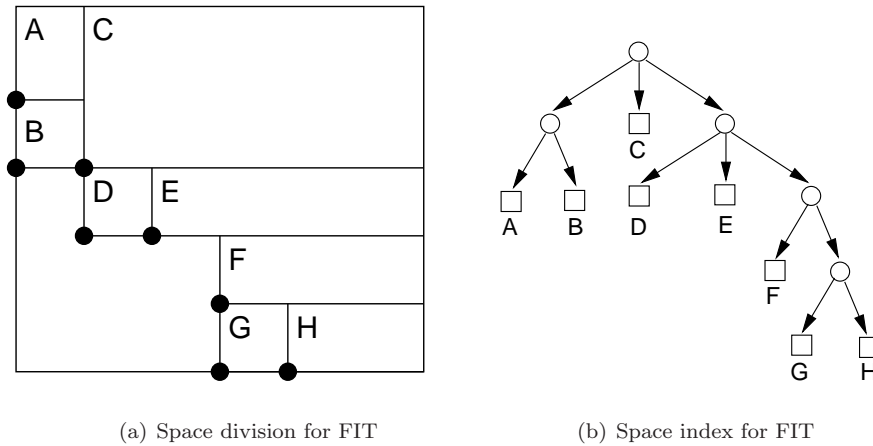


Figure 5.10: Point-Quadtree-based space division and index for FIT

above. Otherwise, we propagate the maximum of all input rates. The assumption here is that any additional reduction will be performed by applying tuple drops at remaining branches of the split. The additional reduction is stored as a complementary local load shedding plan associated with that particular FIT entry, and need not be propagated further upstream.

If node A has multiple child nodes, then the FITs of these children are combined by merging rows from each FIT with the rows from the other FITs. Any new entry violating A's load constraint has to be eliminated. The resulting score is the sum of the children's row scores.

5.5.4 Point-Quadtree-based Division and Indexing of the Input Rate Space

When FITs are propagated all the way from leaves to the roots, we obtain one FIT per node that represents the feasible points for the complete subtree under this node. Next we want to divide the multi-dimensional input rate space for this node into subspaces where each subspace can be mapped to a unique FIT entry. Consider a 2-dimensional space. Let $p(x_1, y_1)$ be a feasible point in this space. Any infeasible point $q(x_2, y_2)$ where $x_2 \geq x_1$ and $y_2 \geq y_1$ could potentially be mapped to p . In fact, if the given bounds of our 2-dimensional space are (x_{max}, y_{max}) , then the complete subspace between p and (x_{max}, y_{max}) could be mapped to p . However, we might like to map some portions of this subspace to other FIT entries that might possibly have higher quality scores. In order to come up with the best mapping, we do the following: Assume that t is the top-most point of our multi-dimensional input rate space. For each FIT point p , we first map the subspace between p and t to p . Then we compare the score of FIT entry p with the scores of the FIT entries for the already existing subspaces. If we find a subspace S whose score is larger than the new subspace N , then we must reduce our new subspace by subtracting S from N . On the other hand, if we find a subspace S whose score is smaller than the new subspace N , then we must reduce S by subtracting N from S . When we do this for each FIT entry, we end up with a disjoint space division where each subspace

is mapped to the FIT entry with the highest possible score. Figure 5.10(a) illustrates how such a division may look like.

As in the solver-based approach, we build a quadtree-based index on top of our space subdivision. However, in the FIT case, instead of dividing the space into regular subspaces and creating plans along the way, we start with a set of plan points and create the subspaces of irregular size based on the existing plan points. Therefore, we end up with a “point-quadtree” rather than a “region-quadtree” [81].

5.6 Putting it All Together

In this section, we briefly summarize how the other three phases of distributed load shedding (i.e., Load Monitoring, Plan Selection, and Plan Implementation) are performed.

- **Centralized Case.** In the centralized approach, the coordinator periodically measures the input rates at the root nodes. Based on the derived input metadata and load factors, the coordinator can estimate the CPU load at each server for the observed input rates. If the CPU load on one or more of the servers is estimated to be above the capacity, then the coordinator searches the quadtree index to locate the load shedding plan to be applied. Otherwise, no server is overloaded and any existing drops in the query plan are removed.

In the case of overload, the coordinator searches the quadtree index to find the load shedding plan that we are looking for. The coordinator sends this plan id to each of the server nodes to trigger the Plan Implementation phase at these servers. Furthermore, inputs may require additional scaling so that the infeasible point exactly matches the plan point. This additional scaling information is also sent to the servers as additional drops to complement the selected plan.

In the Plan Implementation phase, each server node essentially locates the load shedding plan from its precomputed plan table that was uploaded earlier by the coordinator, and changes the query network by removing redundant drops and adding new drops as necessary.

- **Distributed Case.** In the distributed approach, all nodes periodically measure their local input rates and estimate their local CPU load based on these observed input rates. If an overload is detected, a node uses its local quadtree index (built on top of its local FIT) to locate the appropriate local load shedding plan to be applied. Previously inserted drops, if any, must be removed from the local query plan. Note that in all cases, except when local complementary plans are needed due to splits, parent nodes are supposed to ensure that all the nodes in their subtree only get feasible input rates.

The quadtree is used in a similar way as described above, except that instead of sending plan id’s to others, each node directly applies the selected local plan. Again, inputs may require additional scaling and these are applied in the same way as in the centralized case.

Finally, the Plan Implementation phase is the same as described for the centralized case.

5.7 Performance Evaluation on Borealis

5.7.1 Experimental Setup

We implemented our approaches as part of the load shedder component of our Borealis prototype system. We conducted our experiments on a small cluster of Linux servers, each with an Athlon 64 1.8GHz processor. We created a basic set of benchmark query networks which consisted of “delay” operators, each with a certain delay and selectivity value. A delay operator simply withholds its input tuple for a specific amount of time as indicated by its delay parameter, busy-waiting the CPU. The tuple is then either dropped or released to the next operator based on the selectivity value. As such, a delay operator is a convenient way to represent a query piece with a certain CPU cost and selectivity. We used synthetic data to represent readings from a temperature sensor as (time, value) pairs. For our experiments, the data arrival rates and the query workload were more important than the actual tuple contents. We used the following workload distributions for the input rates:

- standard exponential probability distribution with a λ parameter which is commonly used to model packet inter-arrival times in the Internet, and
- real network traffic traces from the Internet Traffic Archive [5].

We present results on the following approaches:

- **Solver.** The centralized solver-based algorithm that is based on the maximum percent error threshold (ϵ_{max}).
- **Solver-W.** A variation of Solver that takes workload information into account and is based on the expected maximum percent error threshold ($E[\epsilon_{max}]$).
- **C-FIT.** The centralized implementation of the FIT-based approach that is based on ϵ_{max} .
- **D-FIT.** The distributed implementation of the FIT-based approach that is based on ϵ_{max} .

5.7.2 Experimental Results

Effect of Query Load Distribution

In this experiment, we investigate the effect of query load imbalance on plan generation time. Figure 5.11 shows the collection of query networks we used for this experiment. Each query network simply consists of two chain queries. Each chain query is composed of two delay operators that are deployed onto two different servers. The numbers inside the operators indicate the processing cost in milliseconds. We apportioned the processing costs such that the total costs of the query networks on each server are the same, while ensuring that there is some load imbalance between two chain queries, (increasing as we go from Figure 5.11(a) to Figure 5.11(e)). In Figure 5.11, we also show the approximate feasibility boundary for the input rates for each query network. Basically, all input

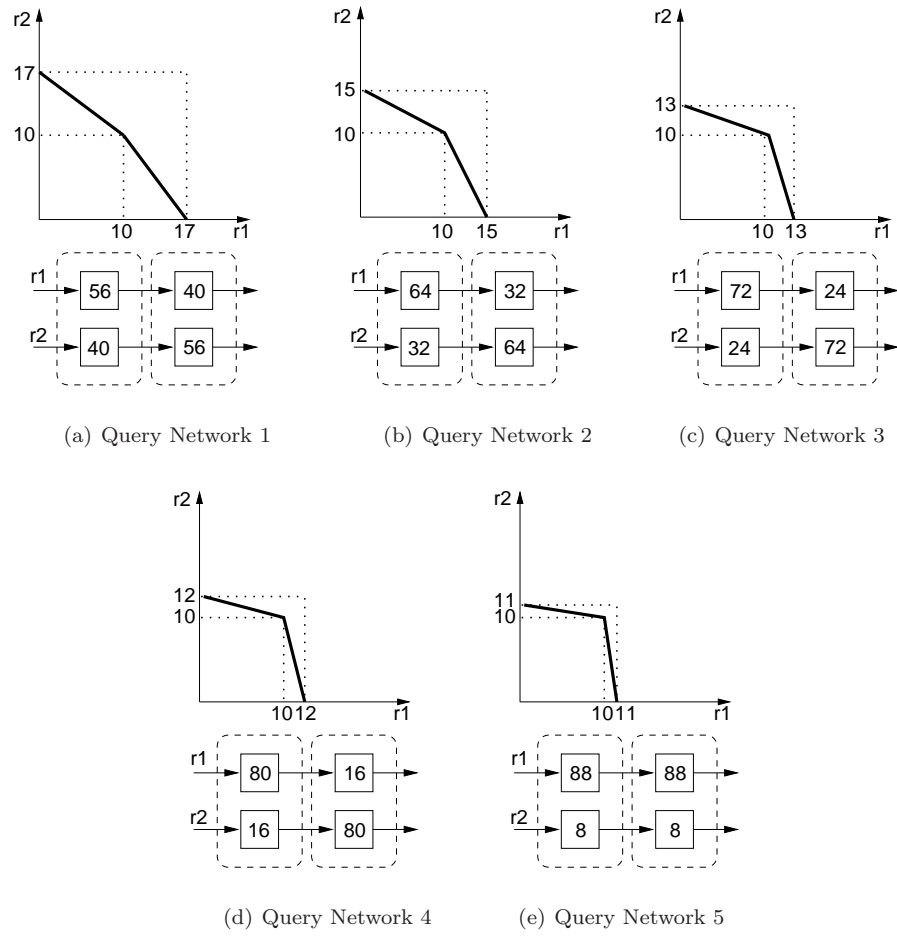


Figure 5.11: Query networks with different query load distributions and feasibility boundaries

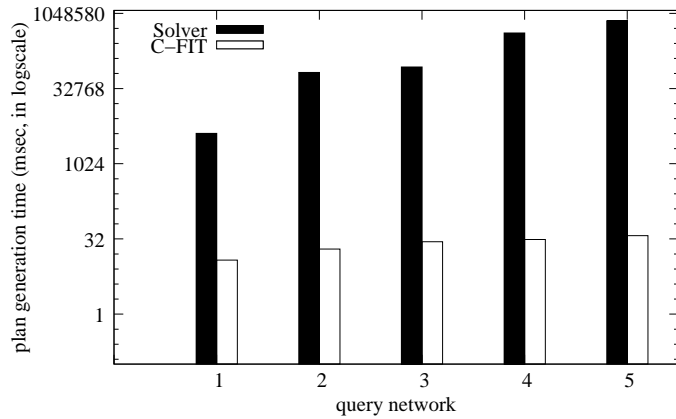
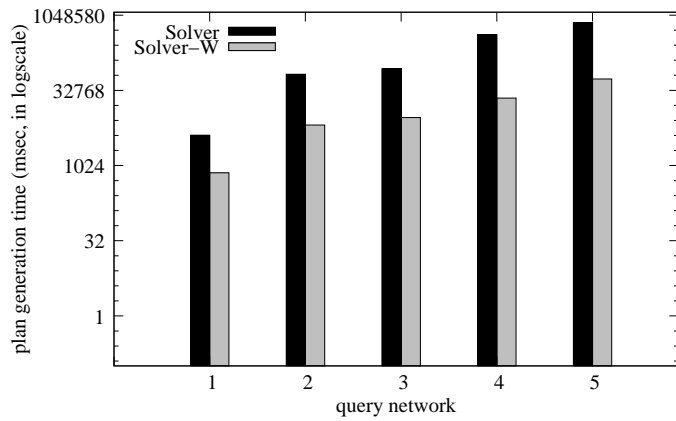
(a) Solver and C-FIT ($\epsilon_{max} = 5\%$)(b) Solver and Solver-W ($\lambda = 5$, exp. $\epsilon_{max} = 1.44\%$)

Figure 5.12: Effect of query load imbalance

rate combinations below this boundary are feasible for both of the servers, while the points on the opposite side represent overload conditions where load shedding would be needed.

In Figure 5.12(a), we compare the plan generation time for Solver and C-FIT, fixing ϵ_{max} at 5%. Both of these approaches guarantee that the maximum error in total quality score will not exceed 5%. C-FIT can generate plans with the same guarantee in significantly shorter time (note the logarithmic y-axis), while Solver turns out to be more sensitive to query load distribution than C-FIT. In Figure 5.12(b), we compare the plan generation time for Solver and Solver-W. In this case, we assumed an exponential distribution for the input rate workload with $\lambda = 5$. When we ran Solver fixing ϵ_{max} at 5%, this produced plans for which $E[\epsilon_{max}]$ turned out to be around 1.44 on the average. Then we used this value as a threshold for Solver-W. As can be seen on our graph, Solver-W takes the workload distribution into account to guarantee the given expected ϵ_{max} value in much shorter time. Both approaches show similar sensitivity to query load distribution.

Effect of System Provision Level

Next, we investigate the effect of expected system provision level on plan generation efficiency for the Solver-W. In order to estimate the provision level, we consider two types of workload: (i) one with a standard exponential workload distribution with parameter λ , and (ii) a real network traffic trace from the Internet Traffic Archive [5]. For the former case, we change the system provision level by varying λ . For the latter case, we use an existing trace from the archive, but we change the query cost in order to create different provision levels.

Figure 5.13(a) shows how plan generation time for the Solver-W increases with increasing λ for the five query networks of Figure 5.11. As λ increases, fewer input rate combinations will fall below the feasibility boundary while more will be above it (i.e., the system will appear as if it is less provisioned). To provide more insight, in Figure 5.14, we provide a color-map of joint exponential probability distribution for two inputs, where axes correspond to input rates and the brightness of an area indicates the expected probability of occurrence. The high probability area shifts up as we increase λ , affecting that area's contribution to expected ϵ_{max} . As we increase λ , we thus expect the plan generation to take more time, as the solver has to be called for more points. Also as in the previous section, as the query load imbalance increases, the plan generation time increases.

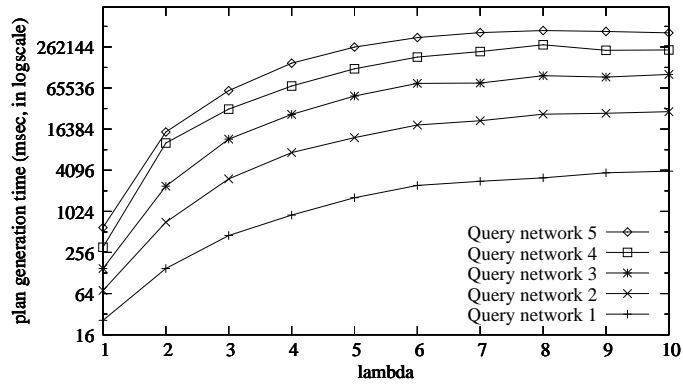
We repeated the same experiment with the TCP traces from the Internet Traffic Archive [5]. Figure 5.13(b) shows this workload distribution, which essentially looks a lot like an exponential distribution. We first used the query network in Figure 5.11(b) (i.e., (64, 32)), which corresponds to a provision level of about 20%. We then reduced the query costs proportionally (e.g., (56, 28), (48, 24), and so on) in order to create increasingly higher provision levels with the same workload distribution. 5.13(c) shows the result, which clearly shows that as the system is provisioned better, the plan generation time decreases.

Effect of Operator Fan-out

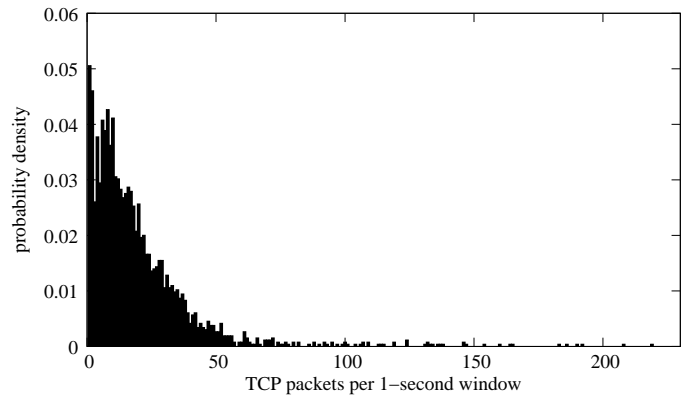
To examine the effect of operator fan-out on Solver and C-FIT, we used a single server deployment of a query tree with 2^k query branches that are fed by a single input stream. These queries share one common operator. Thus, load shedding plans would either place a drop at the input arc or at the split arcs downstream from this common operator. As seen in Figure 5.15, as we increase the degree of sharing in the query network, both approaches spend more time in plan generation. Although Solver can generate plans slightly faster than C-FIT at a fan-out value of 2, C-FIT starts to outperform Solver at higher fan-out values. Thus, C-FIT scales better with increasing operator fan-out.

Effect of Input Dimensionality

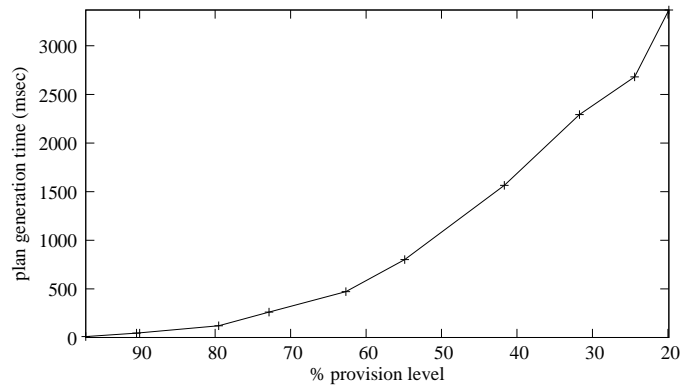
The number of input dimensions is an important factor that can expectedly degrade the performance of plan generation. In this section, we examine how our algorithms are affected from increasing input dimensionality.



(a) Effect of λ (exp. $\epsilon_{max} = 1\%$)



(b) TCP workload from ITA



(c) Effect of provision (exp. $\epsilon_{max} = 1\%$)

Figure 5.13: Effect of workload distribution and provision level on Solver-W plan generation

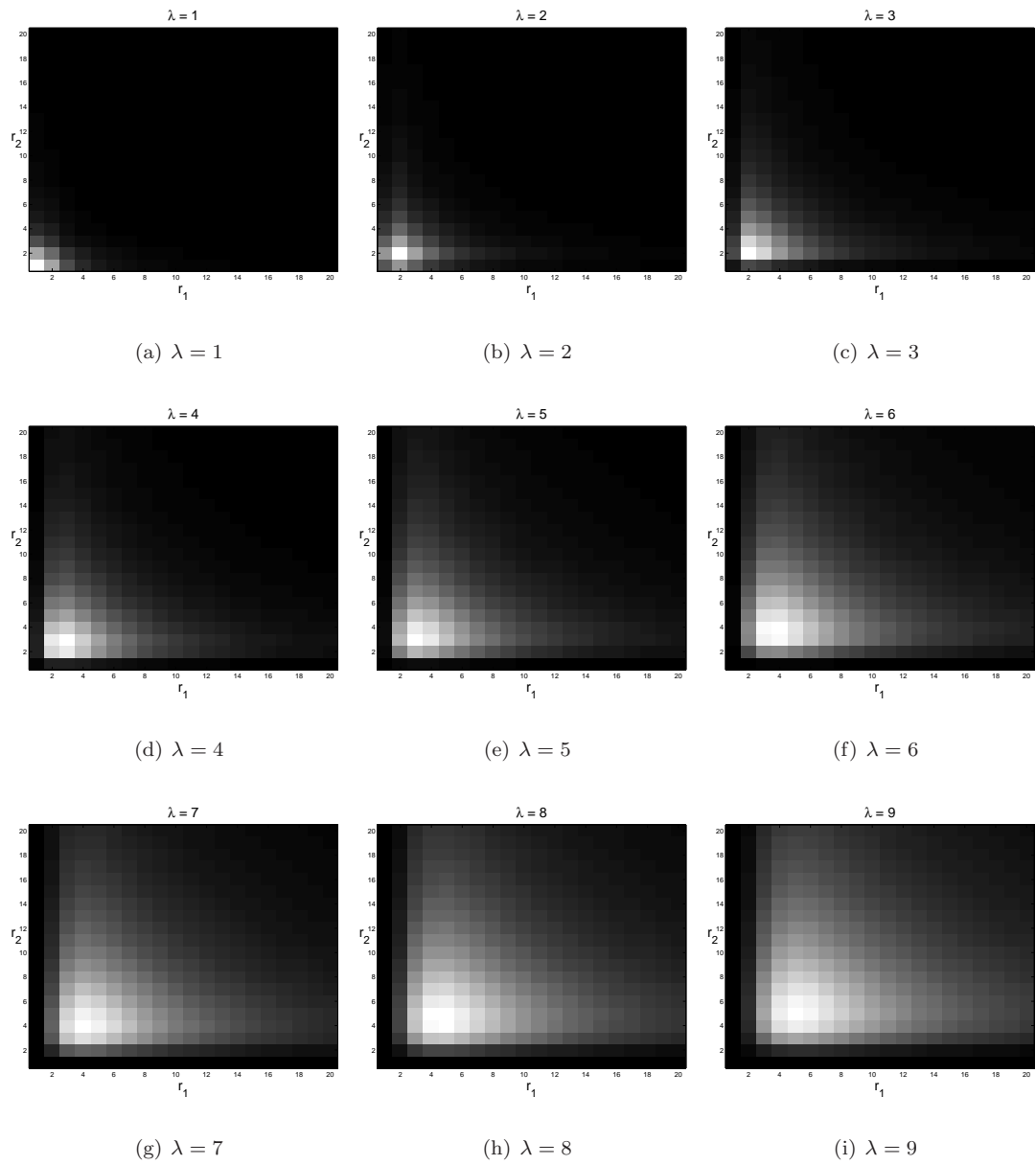


Figure 5.14: Exponential workload distribution for different λ values

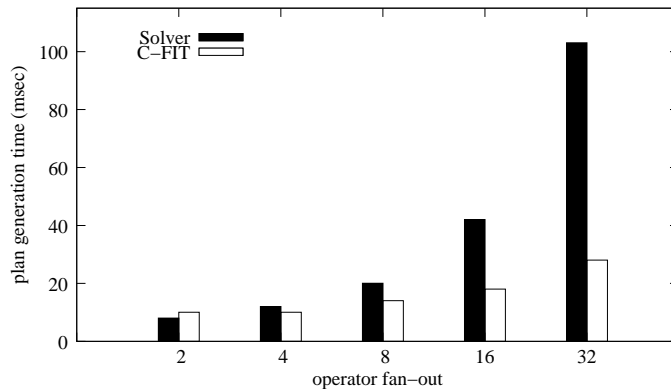


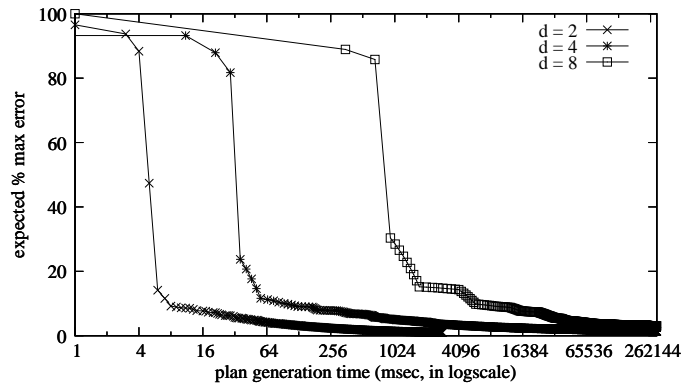
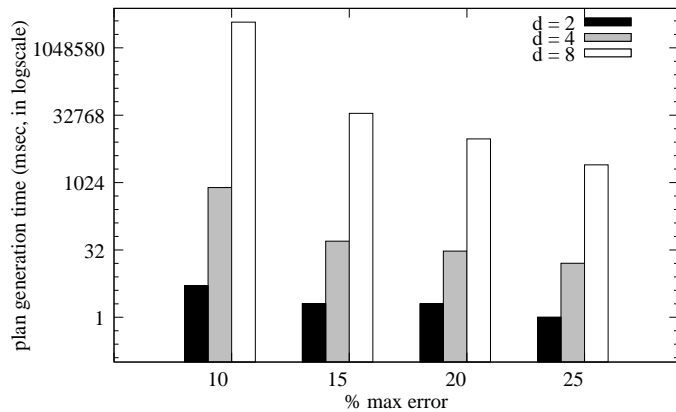
Figure 5.15: Effect of operator fan-out ($\epsilon_{max} = 1\%$)

Figure 5.16(a) shows how the expected % maximum error converges in time for query networks with 2, 4, and 8 input streams. In this experiment, we used the query network of 5.11(b) (i.e., (64, 32)) for the two-dimensional case. Then we increased the number of inputs while proportionally decreasing the query costs for the higher-dimensional cases (i.e., (32, 16, 32, 16) and (16, 8, 16, 8, 16, 8, 16, 8)) and assumed an exponential distribution with $\lambda = 3$. Not surprisingly, as we increase the number of input dimensions, the plan generation time significantly increases. As shown in Figure 5.16(b), the situation is worse for C-FIT (and similarly for the Solver, which is not shown). For example, to ensure a 10% maximum error for 8 inputs, C-FIT needs to run for about an hour. These results demonstrate the "curse of dimensionality". In practice, however, the outlook is better. First, plan generation is an off-line process; it is performed in advance of the overload, potentially using idle cycles. Second, the generated plans are often reusable. Finally, in our experience with stream-oriented queries for the financial services domain, the number of input streams are generally few.

Overhead Analysis for Distributed FIT

FIT is a distributed algorithm by design. As such, it can provide all the features of distributed algorithms such as avoiding hot spots and single point of failures. It is hard to provide quantitative evidence for why D-FIT would beat the coordinator-based approaches. However, as mentioned earlier, it is clear that there are certain settings where D-FIT would be preferable (e.g., multi-hop, resource-limited sensor networks, wide-area data dissemination trees, etc.) because of its ability to scale and to dynamically react to changes. In this section, we analyze some overhead issues that may be associated with D-FIT, that would bear importance in such settings.

Our main criteria in the overhead analysis is how much FIT information needs to be communicated between two neighboring nodes. Since complementary local plans associated with FIT entries are not actually being sent to the parent, FIT entry size is simply proportional to the number of input streams. Therefore, we will focus on the number of FIT entries rather than the total byte size.

(a) Convergence of exp. ϵ_{max} for Solver-W

(b) Effect of dimensionality on C-FIT

Figure 5.16: Effect of input dimensionality

For the same reason, operator fan-out, which affects the size of the complementary plan column of FIT, is not very interesting. As a result, we identified three major sources of overhead for FIT size: (i) the number of input dimensions, (ii) magnitude of the ϵ_{max} threshold, and (iii) query load.

As Table 5.5 clearly shows, for a given ϵ_{max} threshold and a fixed query load, when we have more inputs, we need to represent a higher number of feasible input combinations, which consequently requires a higher number of FIT entries. For 2 inputs, Figure 5.17(a) details the effect of ϵ_{max} and query load. As we reduce the query load, a larger portion of the input rate space becomes feasible and this increases the number of FIT entries. Another interesting point is that if there is operator fan-out in the query plan, where some portion of the query can be handled with complementary local plans, then the query appears less costly to the parent node, and therefore, we would have a higher number of feasible point entries in the FIT. Thus, it is not the actual query load, but the load after local plans are applied that determines the needed FIT size. Figure 5.17(a) also shows that, if the ϵ_{max} threshold is increased, allowing a larger distance from optimal quality plans, then the

# of inputs	# of FIT entries	bytes/entry
2	46	52
4	984	96
8	42472	184

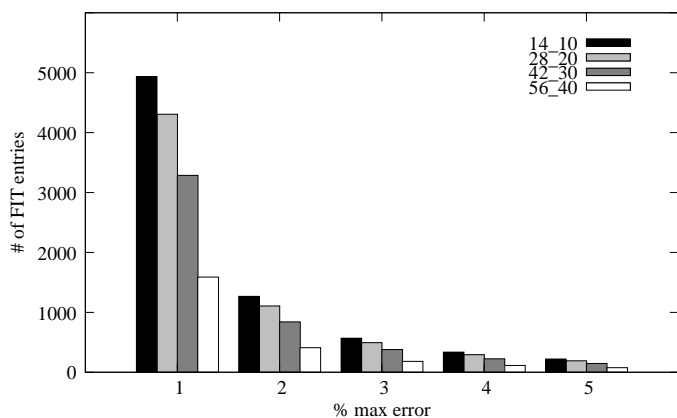
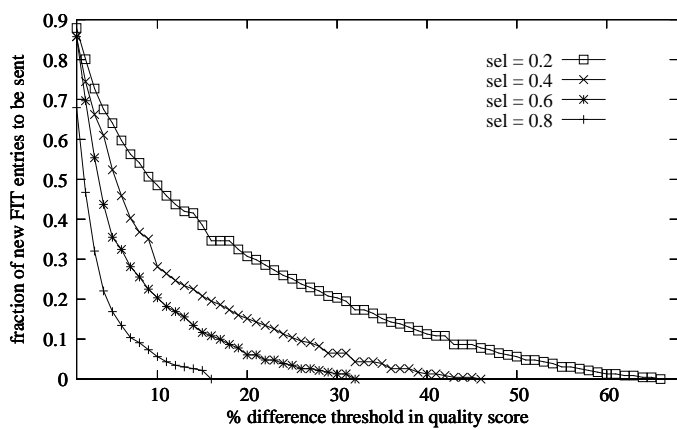
Table 5.5: Effect of dimensionality ($\epsilon_{max} = 10\%$)

number of required FIT entries decreases in a dramatic way. This suggests that ϵ_{max} can actually be adaptively adjusted to trade off plan quality for reduced communication overhead. This could also be used as a remedy to the high dimensionality problem.

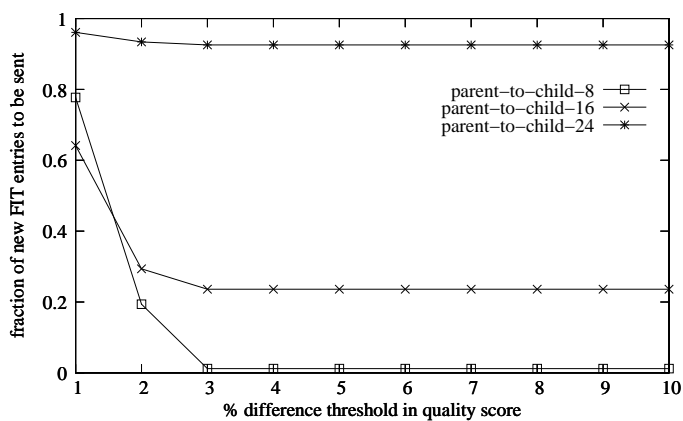
One major advantage of D-FIT over the centralized approaches is that it enables efficient handling of certain dynamic changes in load conditions. Depending on the nature of the change, there may be cases where it would be sufficient to only update the local FIT, or propagate it only to a small number of upstream nodes. Yet in other cases, it would be possible to send deltas to the parent rather than the complete FIT. We identified two important cases to analyze: (i) changes in operator selectivity, and (ii) changes in query load due to operator movement between neighboring nodes. Figure 5.17(b) shows the sensitivity of FIT to operator selectivity change. In this experiment, we used two chain queries. One of the operators in one of these queries initially has a selectivity of 1.0. We compute the FIT to be sent to the parent node. Then we decrease the selectivity and recompute the FIT. We then measure what fraction of the new FIT entries must be communicated to the parent in order to stay within some distance from the actual quality score. We find that as the change in selectivity increases, we need to send more entries in order to achieve a certain difference threshold. Similarly, for a given selectivity change, we need to send more entries if we want to stay within a smaller difference from the actual quality score. Figure 5.17(c) shows the sensitivity of FIT to load movement. In this case, we again use two chain queries similar to the ones in Figure 5.11. Then we reduce the delay parameter of a parent operator, while adding that same amount to the delay parameter of the downstream child operator (e.g., parent-to-child-8 means that we moved 8 units of load from parent to child). We see that as the amount of load moved gets bigger, it requires more FIT entries to be sent to the parent. In the case for parent-to-child-24, the load movement causes a complete reversal in load balance. Therefore, it is not even possible to reduce the communication overhead beyond 8%, no matter how large a difference we allow.

5.8 Chapter Summary

In this chapter, we have introduced the problem of load shedding in distributed stream processing. We have shown how it differs from previous centralized solutions, and we have offered several new practical algorithms for addressing the problem. We presented our main result, a distributed algorithm that we call D-FIT that works by transmitting its load requirements locally to its parents. We also investigated several centralized solutions – a linear programming solution (Solver), a variant on Solver that takes a workload history into account (Solver-W) and a centralized version of our

(a) Effect of ϵ_{max} and query load (52 bytes/entry)

(b) Sensitivity to selectivity change



(c) Sensitivity to load movement

Figure 5.17: D-FIT Overhead

distributed algorithm (C-FIT).

As we have said earlier, the purely distributed version of our algorithm D-FIT is especially useful when the number of nodes is very large, as in an Internet-scale dissemination system or when the underlying query environment is quite dynamic as in a distributed sensor network.

Chapter 6

Related Work

In this chapter, we discuss existing work related to our research. We present past work in increasing order of their relevance to our work. We start with discussing other areas of computer systems, including computer networks, network services, and multimedia systems. Then we discuss various subareas of database systems, including multimedia databases, real-time databases, approximate query processing, and parametric query optimization. Finally, we conclude with discussing load shedding approaches proposed by various research groups working in the area of data stream management systems.

6.1 Computer Networking

The congestion control problem in computer networks has close conceptual relevance to the overload management problem in data stream systems [57, 59]. Both problems essentially arise when load is temporarily greater than the level that available resources can handle. Congestion in computer networks mainly arises when routers run out of buffer space, either because their processors can not keep up with the incoming input traffic, or because the outgoing link has a smaller bandwidth capacity than the incoming link [90]. This situation can be handled either preventively (e.g., traffic shaping, resource reservation), or reactively (e.g., admission control, load shedding).

Load shedding is a network-layer congestion control technique where a router simply drops packets when its buffers fill up. The packets to drop can be chosen randomly, or in an application-dependent way. For example, senders can label their packets with priority levels by marking a special bit in the packet header and the router drops the low priority packets. Dropping can also be age-based. For real-time applications such as multimedia applications, new data is more important than older data, hence older packets are dropped (so-called *milk policy*). For other applications like file transfer, older is better, hence newer packets are dropped (so-called *wine policy*) [90].

Despite conceptual similarity, there are some fundamental differences between network-layer load shedding and load shedding in a single-server data stream management system like Aurora. First, unlike network load shedding, where routers drop packets in an uncoordinated manner, load shedding

decisions in Aurora is made by a load shedder component that has access to the entire system state, and therefore, can potentially provide more intelligent, end-to-end load shedding decisions. Second, Aurora uses QoS information provided by the external applications to trigger and guide load shedding. This may require the load shedder to take tuple contents into account. Third, unlike routers, which simply pass packets between nodes, Aurora query operators perform operations on data tuples. Therefore, the load shedder has to deal with operator semantics.

Distributed load shedding is also relevant to the congestion control problem in computer networks. Various IP-layer architectures have been proposed to maintain Internet QoS including IntServ [23] and DiffServ [22]. OverQoS [88], an overlay-based QoS architecture, uses application-level techniques to prioritize packets that have higher utility for the application. Our upstream metadata propagation technique resembles the pushback mechanism developed for aggregate congestion control [68], where a congested router can request its upstream routers to limit the rate of an aggregate (i.e., a certain collection of packets sharing a common property) primarily to defend against DoS attacks. In our FIT-based approach, nodes also specify their feasible input rates to their parents, but it is the responsibility of the parent to decide how to reduce outgoing stream rates to maximize system throughput.

6.2 Network Services

Overload management techniques for common Internet-based services such as HTTP servers and file sharing services are also relevant to our load shedding approaches. In these applications, increase in the rate of service requests may overload the servers, and as a result, service response times may unpredictably grow. To control response times, servers must provide adaptive service degradation and admission control mechanisms.

Of particular relevance is a recent work proposed by Welsh and Culler [103]. In this work, services are constructed as a network of event-driven stages connected with explicit request queues - an architecture called the Staged Event-Driven Architecture (SEDA) [104]. In SEDA, fine-grained overload control is achieved by applying admission control on each event queue. If a request is rejected at any stage of the service network, then the upstream stage is also informed to take action. The right action can depend on various factors like the service logic, priority of the request, etc. Additionally, depending on the application, different service degradation mechanisms (e.g. lowering the service quality to be able to accept more requests) can be employed at each stage.

Overload management for data stream processing is fundamentally different from that for Internet service processing, since the former has to deal with large volumes of data, and semantics of the query operations on this data, whereas the latter has to deal with service requests. On the other hand, the stages in SEDA highly resemble the query operators in Aurora. Unlike SEDA's fine-grained load control on each stage queue, however, Aurora's load shedder component controls the query network as a whole. This gives Aurora more direct control for optimal resource management.

6.3 Multimedia Streaming

Real-time streaming multimedia applications make up a significant portion of the Internet bandwidth. These applications call for QoS support at multiple system layers including the network, the operating system, and the application layers. Therefore, an important concern in multimedia streaming has been to develop a proper QoS framework that could be supported in an end-to-end fashion across all these layers.

A key challenge in sending continuous media over the network is matching the transmission rate to the currently available bandwidth. To achieve this, load shedding at the network layer may be required (as briefly mentioned in Section 6.1). Previous work has taken application performance into account by exploiting layered encoding algorithms for transmitting video data [19]. In this work, packets that carry base layer data are marked as highest priority whereas packets for each successive enhancement layer are marked as successively lower priority. During times of network congestion, low priority packets are dropped. This way, result quality is degraded in a controlled and graceful manner.

The Quasar project developed at OGI has proposed various techniques for building a QoS-adaptive multimedia delivery system [102]. In this work, MPEG frames are partitioned into temporal and spatial resolution layers using a custom scalable format called SPEG. Video packets are marked at the sender with priority levels based on user-specified utility functions along several orthogonal QoS dimensions. Then packets below a certain priority threshold can be dropped at routers (or possibly anywhere in the delivery pipeline) to adapt resource requirements. This priority threshold is adjusted based on feedback from the receiver [64]. A flow and congestion control scheme for media streaming applications, called Streaming Control Protocol (SCP), has also been proposed in scope of the Quasar project [26]. This work identifies various drawbacks of the TCP congestion control model for real-time media applications, and proposes a new scheme that will provide smooth streaming, reduced jitter and predictable latency. Unlike TCP, SCP does not retransmit lost packets; the goal is to improve network utilization and avoid unpredictable latencies. These techniques have recently been merged into a priority drop technique [63].

As in Aurora, these techniques take application-dependent data utility into account while deciding which packets to discard. In fact, Quasar's utility functions look very similar to Aurora's QoS functions. However, these techniques are essentially network-layer techniques and the fundamental differences we mentioned in Section 6.1 still apply.

More recently, the QuaSAQ project from Purdue University has proposed QoS-aware query evaluation and optimization approaches for multimedia databases that are distributed to multiple sites [97]. Alternative query plans are generated and sorted according to their resource consumption level based on a cost model. QoS constraints are used to both prune the plan search space and to pick the right plan to execute. This approach is closer to the application-layer, and follows a similar approach as our idea of materialized load shedding plans.

Lastly, there has been some related work on resource allocation for multiple multimedia applications that share resources of a single machine. In this case, neither networking nor streaming

issues are involved, but load shedding may still be required when multiple applications compete for resources for which the total demand may exceed the available supply. Compton and Tennenhouse explored *collaborative load shedding* techniques for interactive multimedia applications [33]. In this work, applications take user's priorities into account to adjust their demand for system resources like CPU. This work argues that applications can shed load in a much more graceful and intelligent manner than the operating system itself. We also take application-specific QoS functions into account while making load shedding decisions for a multitude of time-critical applications.

6.4 Real-Time Databases

Real-time databases are databases where transactions have deadlines or timing constraints [62, 73]. A transaction scheduler must carefully allocate available CPU cycles to transactions in order to meet these constraints. In most cases, resource requirements may not be known in advance, either because transaction workload dynamically changes, or because transaction execution times can not be precisely estimated due to interactions with the disk-based I/O subsystem. The CPU may become overloaded and some transactions may miss their deadlines. The system must provide effective overload management techniques in such cases, in order to minimize the number of timing constraints violated.

Typical solutions include admission control on newly arriving transactions by simply rejecting them; or termination of already running transactions by aborting them. The transaction manager can also assign priorities to transactions based on the closeness of their deadlines, or other application-specific importance criteria. These priorities are then used to manage and resolve potential scheduling conflicts among transactions.

The STanford Real-time Information Processor (STRIP) is a soft real-time system which uses value functions and maximum age criteria to enforce soft timing constraints [11]. Transactions are worthless if they miss their deadlines and therefore such transactions must be immediately aborted. Additionally, if the system is overloaded, transactions that are less likely to finish on time can also be aborted before their deadline. Transactions are assigned values based on their importance and are prioritized based on value density, i.e., the ratio of transaction value to remaining processing time. This system also studies the tradeoff between scheduling streams of update transactions on materialized views and read transactions on these views [10].

The CASE-DB real-time database system uses a risk control technique on how queries should spend their time quotas [72]. As opposed to a soft real-time system, in this system, timing constraints are always guaranteed to be satisfied. The trick is to provide approximate answers to queries to the extent allowed by their timing constraints. A query is rewritten into a form where it can be evaluated on higher priority fragments of its base relations with zero risk of missing its time constraints. Then this approximate result is iteratively improved by evaluating the query on other less critical fragments of the relation in the remaining time quota (if any). This idea of producing monotonically improving approximate answers to real-time queries has also been studied by the APPROXIMATE

query processing system [100]. This system additionally augments the approximate answer with information on what else might be appearing in the full answer.

Hansson and Son propose an overload resolution algorithm to handle transient overloads in real-time database systems by rejecting non-critical transactions, and selectively replacing critical ones with lower-cost contingency transactions [50]. As in CASE-DB, transactions have hard or firm deadlines. Hence, deadlines for admitted transactions are never missed; but instead, lower quality, approximate results may be produced by the contingency transactions. Similar to our QoS functions, value functions are used to represent the importance and criticality of a transaction as a function of its completion time. Also, as in Aurora, scheduling and overload management policies are handled in an independent fashion.

More recently, Kang et al have proposed a QoS management scheme to provide guarantees on deadline miss ratio and data freshness [61]. As in STRIP [10], the goal is to flexibly balance user and update transactions. Relative frequencies of updates and read-accesses are utilized to prioritize transactions.

Real-time databases deal with *streams* of one-time transactions performed on relatively static, stored datasets; whereas data stream management systems handle standing queries on continuously arriving streams of data. The overload handling approaches for real-time databases summarized above can be regarded as load shedding techniques, where shedding units are individual transactions, rather than data tuples. Aurora’s load shedding approach is finer-grained since we are dealing with individual tuples. In Aurora, dropping can be performed at any place inside the operator network whereas in real-time databases, transactions are rejected or aborted as a complete unit. This is a result of potential operator sharing among multiple continuous queries in Aurora.

6.5 Approximate Query Processing

Approximate query answering, where result accuracy is traded for efficient execution, relates closely to load shedding. Approximate query processing techniques have long been studied for both traditional static data sets and continuous data streams [44].

Data reduction on large, static data sets is a common approach to producing fast approximate answers [21]. These techniques mostly rely on pre-computing a smaller summary of the base data, called a *synopsis*. Queries are then rewritten into a form that can be executed on the synopsis data structures, significantly cutting down from the total query processing time. The synopses can be in the form of samples [9, 30, 71], histograms [56], sketches [41], or wavelets [27, 99], and can represent one- or multi-dimensional data. A major issue with this approach is that the synopses must be efficiently maintained as base data gets updated.

An alternative approach to pre-computing synopses is the *online query processing* approach of the CONTROL project [51]. This approach interleaves sampling with query evaluation. For aggregation queries, running aggregate results on increasingly larger samples of the base data are presented to the user, together with statistical error bounds [52]. The result is continuously refined until the

sample eventually expands to cover the complete data set. For join queries, data from multiple joining relations are sampled at different rates to provide the highest precision confidence intervals using a ripple join operator [49]. Unlike our system, queries are one-time and data sets are large but finite. Moreover, the exact answer is eventually produced. The CONTROL system also reorders data dynamically based on user preferences so that interesting items get processed early on [77]. Rather than changing the data order, we throw away some of the data since we are also concerned with extremely high data rates and real-time answers.

Yet another approach to approximate query answering is to pre-organize data into a semantic hierarchy (rather than a synopsis) which can be used in producing approximations as well as full answers. Read et al propose a multi-resolution data model for producing fast low-resolution answers to queries and then progressively refining these into full answers [78]. The Cooperative Database System (CoBase) proposes a similar query relaxation approach [32]. In this work, data is organized into type abstraction hierarchies and various operators are defined as an extension to standard SQL that can be used to control the degree and form of the required relaxation. For example, similar to our value-based QoS function, the user can specify a preference list which indicates the order of preference between various values of a given attribute. The goal in this work is to improve both the system performance and the answer relevance. The APPROXIMATE query processing system also organizes data into a class hierarchy in order to provide monotonically improving approximate answers [101]. Approximations in this work are supersets of exact answers, composed of a certain and a possible set.

More recent work have also explored synopsis-based approximate query processing techniques for streaming data based on constructing single-pass summaries in different forms. For example, histogram-based techniques are used for maintaining correlated aggregates over data streams [45]; sketches and wavelet transformations are used to approximate aggregate queries [39, 46].

6.6 Parametric Query Optimization

Parametric query optimization addresses the problem of adapting query plans to changing run-time parameters. A set of candidate query plans, each of which is optimal for some region of the parameter space, is pre-computed. Then the appropriate plan is chosen among these candidates at run-time based on actual values of the parameters.

The parametric query optimization concept was first proposed by Ioannidis et al [54]. This work assumed a set of plans that are neighbors of each other in a plan search space. Randomized algorithms are used to locate plans in this space. As the algorithm moves from one plan to another neighbor plan, it recognizes the region changes for certain parameters. Finally, it produces a mapping function from vector of parameters to the plan space. This work studied the optimization of join trees with buffer size as the changing run-time parameter. It does neither provide any guarantees about producing all parametric optimal plans nor any optimality bounds for the resulting plans. This work has later been extended to use dynamic programming algorithms instead of the randomized

ones [55].

Ganguly has observed that the parameter space is partitioned into convex polyhedrons, and each polyhedron is a region of optimality for a parametric optimal plan [43]. This work gives algorithms for linear cost functions with one and two parameters, and then extends them to non-linear cost functions.

A more recent work explores identifying regions of optimality for linear and piecewise linear cost functions [53]. This work focuses on minimizing modifications to the underlying query optimizer as well as minimizing the number of times the optimizer is invoked.

Our off-line load shedding plan generation approach (in particular, our distributed load shedding approaches described in Chapter 5) can be regarded as an instance of parametric query optimization. Unlike previous instances of the problem that studied standard query optimization in single-server environments, our solution addresses throughput optimization under overloads in distributed settings.

6.7 Load Shedding in Data Stream Management Systems

There has been a great deal of recent work in the area of data stream management systems [48]. Several research prototypes have been built, including Aurora [8], STREAM [70], and TelegraphCQ [28]. Efficient resource management, adaptivity, and approximation have been the main points of emphases in these systems. Next we discuss various load shedding approaches developed by several research groups, and provide a detailed comparison of these against our own approaches.

6.7.1 STREAM

The STREAM System built at Stanford University [70] has proposed two major approximation techniques to deal with resource overload problem on data streams: (i) a CPU-limited approximation approach that handles load shedding on a collection of sliding window aggregation queries, and (ii) a memory-limited approximation approach that focuses on discarding operator state for a collection of windowed joins.

First, Babcock et al have proposed a sampling-based load shedding approach for aggregation queries [17]. This work focuses on query trees that consist of a number of filters at higher levels followed by aggregate operators at the leaf level. When the CPU time needed to process tuples that arrive in unit time exceeds 1, random sampling operators are inserted into these query trees. The goal is to minimize the maximum relative error across all queries. Sampling rates are adjusted such that the relative error is the same for all queries. This is achieved using well-known statistical bounds (e.g. Hoeffding inequality), based on mean and standard deviation statistics for windows of tuples that are being aggregated. Our approach produces subset results and tries to minimize QoS utility loss, hence our approximation model is quite different. Moreover, our window-aware approach addresses a more general class of aggregation queries (nested aggregates as well as user-defined ones), while the proposed work applies to query trees, with a single aggregate operator at the leaf level.

Furthermore, the suggested statistical bounds apply to only a limited set of aggregate functions (e.g. sum, average), whereas our approach is independent of the actual aggregate functions.

Second, Srivastava and Widom have proposed a load shedding approach for windowed stream joins in memory-limited environments [85]. This work is based on an age-based data arrival model where it is assumed that the rate at which a tuple produces join results is solely determined by its age, specified as an age curve. To deal with memory shortage, tuples of a certain age are selectively discarded from the join window to make room for others, which have higher expectation of producing matches. The goal here is to maximize the size of the resulting subset. A secondary concern in this work is to be able to produce a random sample from the join in case that the join is followed by an aggregate. In this case, the final output will not be a subset of the exact answer, and the overall goal is to minimize the relative absolute error. The main difference of this work from ours is that it investigates the memory problem rather than the CPU problem.

6.7.2 TelegraphCQ

TelegraphCQ is a continuous query processing system built at University of California, Berkeley [28]. This system's focus has been on adaptive and shared query processing aspects of data stream management. The Berkeley group has worked on two independent approaches that relate to our load shedding techniques.

Reiss and Hellerstein have proposed an adaptive load shedding approach called *data triage* [80]. Data triage is a query processing architecture designed to provide low latency results in the face of bursty data streams. During bursty periods, if streams arrive faster than the rate that the system can process them, excess data is stored in synopsis data structures. At the end of each query window, the synopses are processed through a shadow query plan to compute an approximate result on the summarized portion of the data. Finally, exact and approximate results are merged into one composite result for that query window. The fundamental difference of this approach from ours is that the excess data is not discarded but stored in some reduced form to be processed later when the system has more time to process it ¹. Additionally, data triage does not guarantee subset results; rather uses a different error model based on Minkowski distance [79].

Chandrasekaran and Franklin have investigated a different overload scenario in TelegraphCQ, where large number of hybrid queries are to be processed on a combination of live, real-time data streams and historical data archived on disk [29]. In this case, disk becomes the bottleneck resource. To keep processing of disk data up with processing of live data, a technique called OSCAR (Overload-sensitive Stream Capture and Archive Reduction) is proposed. OSCAR organizes data on disk into multiple resolutions of reduced summaries (such as samples of different sizes). Depending on how fast the live data is arriving, the system picks the right resolution level to use in query processing. This is a form of load shedding that tries to cut down from disk access cost, whereas our focus is on reducing the demand on CPU. Also, in this approach no data is actually discarded; the full version

¹The assumption is that bursts do not last longer than a time window, i.e., the system will have enough time to process synopsis data at the end of each time window.

of the data is always available at the archive.

6.7.3 NiagaraCQ

NiagaraCQ is an XML-based continuous query system that has been developed at University of Wisconsin and Portland State University [31]. Although this project's focus has mostly been on query optimization, a couple of publications provided optimizer extensions for dealing with resource limitations.

Kang et al provided an analytical approach for optimized evaluation of window joins for unbounded streams [60]. A unit-time-based cost model is developed where total cost is broken into two components, one for each join direction. The optimal index and join algorithm combination for each direction is determined. The paper also looks at the resource overload problem for both CPU and memory. The ideal rate for each input is determined and a random drop is inserted for rate adjustment.

The optimizer-unified approach of Ayad and Naughton uses a similar analytical cost model, this time on join trees [16]. It finds that if computational resources are enough, then all join plans have the same throughput, however, they may substantially differ in resource utilization. If all plans are infeasible, then load must be shed via random drops. The focus is on picking the join plan, the locations on the plan to insert the drops and the amount of drops. Interestingly, the optimal join plan (i.e., with the lowest utilization) when resources are sufficient may not be the optimal plan (i.e., with the highest throughput) when resources are insufficient.

Also in the scope of the NiagaraCQ Project, Tucker et al proposed punctuations which are special annotations embedded into data streams to specify end of a subset of data in the stream [98]. Punctuation-based query processing is devised to overcome the blocking and unbounded memory problem in stateful stream operators. As such, punctuations constitute an alternative to windowed processing. Our window-aware load shedding work is relevant to punctuations in the way we attach window indicators into tuples. Although in both cases streams are annotated with information that is important in terms of optimizing query execution, the goals are quite different. In the punctuations case, annotations indicate some property that naturally exists in the stream, whereas in our case, window specifications are artificially injected to cope with overload.

6.7.4 The Cornell Knowledge Broker

The Cornell Knowledge Broker architecture developed at Cornell University supports real-time continuous queries as well as offline data mining and analysis operations on both streaming and archived data [36]. This architecture includes a load smoothing module which is responsible for handling overload situations. During peak load, load shedding is applied, selectively eliminating tuples to be processed later and producing approximate results. During low load, approximate results are reprocessed into exact results, based on the tuples that were previously shed into an archive.

More specifically, this system addresses the memory limitation problem for stream joins, where

the maximum subset measure as the approximation metric [35]. For load shedding, an optimal offline algorithm is developed. This algorithm requires complete future knowledge about tuple arrival and therefore can not be used in practice, but rather theoretically bounds the best approximation that can be achieved. Then based on a frequency-based data arrival model, two practical heuristics are proposed: (i) PROB, which drops tuples from an input stream which had the smallest frequency of occurrence on the opposite stream in the past (assuming that those tuples are the least likely to produce join results also in the future); (ii) LIFE, which drops tuples from an input stream whose product of frequency of occurrence on the opposite stream and remaining window lifetime is the smallest (the goal is to avoid investing on soon to be expired tuples).

6.7.5 Model- and Control-based Approaches

In addition to the systems we discussed above, there have been a few other studies of interest. Jain et al proposed a model-based approach to managing network bandwidth for streaming applications [58]. This approach is based on Kalman Filters that can model data streams as processes with states that evolve over time. As new tuples arrive at the source site, it is checked if the current model installed at the remote server site can still answer the query within given precision bounds. If so, there is no need to send the new tuple to the server, i.e., it can be discarded. Otherwise, the server model has to be updated, hence the new tuple is transmitted. Adaptivity is achieved by adjusting model parameters to changing load characteristics.

In another similar study by Xie et al, each stream is modeled as a stochastic process [106]. At each time point, the probability of observing a certain value is given. Accordingly, expected benefit of keeping a tuple in the join state is computed. Tuples with lowest benefit values are discarded. This work generalizes the model heuristics used in some of the earlier work we discussed above [35, 85].

More recently there has been some related work that applies control-theoretic concepts to adaptive load shedding on data streams [13, 96]. These approaches are based on constructing a feedback loop that continually monitors the high-frequency variations in system parameters and makes the necessary adjustments in the load controllers accordingly. Amini et al. [13] propose a two-tiered approach that combines long-term operator placement and short-term CPU scheduling to maximize throughput in a distributed stream processing system. This is a closed-loop solution that continually adjusts the buffer sizes at each node to achieve high throughput and low latency, while ensuring stability in the presence of varying workload and bursts. The main differences of our approach from these in general are that our solution focuses on relatively longer duration bursts in data rates, and uses an open-loop approach to create parametric off-line load shedding plans that can limit the deviation from the optimal plan while reacting fast in plan selection.

Chapter 7

Conclusions and Future Work

In this thesis, we defined the load shedding problem for data stream management systems and we proposed a general solution framework based on discarding tuples through various types of drop operators. A key feature of our solution is that most of the analysis concerning dropping strategies can be done statically and captured in a simple data structure. The dynamic load shedding process involves a very cheap use of the static information. This technique makes our solution highly efficient. Furthermore, we proposed several different types of drop operators which would guarantee subset results at query outputs, when placed into query plans with operators of different semantics. In particular, our window-aware approach can handle a general class of aggregation queries (with nested as well as user-defined aggregates) while preserving the subset guarantee and achieving early drops.

We have also presented techniques for distributed load shedding. In line with our centralized load shedding approaches developed for Aurora, these techniques also rely heavily on creating advance load shedding plans. Our focus in this case has been on efficient generation and storage of load shedding plans when the search space can be very large due to various factors like large number of processing nodes and input streams, or high degree of operator fan-out or query load imbalance among the data flows.

We evaluated the performance of our algorithms both analytically and experimentally. As our simulation study demonstrates, our core load shedding techniques outperform basic admission control and its variants. We have also shown that while our probabilistic dropping technique can do fairly well, the method that takes tuple semantics into account can do even better. Our experiments also clearly show that as we increase the difference in importance between the most valuable tuples and the least valuable tuples, semantic load shedding produces more striking benefits. We have also conducted experiments on the Borealis prototype. Our prototype study demonstrates that load on aggregation queries can be shed in an effective manner, without sacrificing the subset result property. We have shown that, as is expected, with the added ability to push drops past aggregates, we can recover more load early; thereby, regaining the required CPU cycles while minimizing the total utility loss. By focusing on dropping windows, we can better control the propagation of error

through the downstream network.

7.1 Future Directions

The load shedding problem in data stream management systems is an important and broad problem, and presents future research opportunities in various interesting directions.

7.1.1 Managing Other Resources

In the current study, we have focused on load shedding to reclaim processor cycles. In many stream-oriented applications, cycles are not the limited resource. Often things like bandwidth or battery power will be the resources that must be conserved. Load shedding via dropping of tuples has an important role to play in these environments as well. We intend to investigate ways in which we can design techniques similar to the ones discussed here that can work for other kinds of resource management.

For example, query networks with large number of stateful operators like aggregates may also require load shedding due to insufficient memory. An aggregate, with window size ω and window slide δ has at most $\lceil \frac{\omega}{\delta} \rceil$ open window states per group. However, depending on the form of the aggregate function (i.e., distributive/algebraic vs. holistic), a window state may become unbounded [14]. We can easily adapt our window drop approach to such memory-constrained environments.

In the distributed setting of the Borealis system, bandwidth bottlenecks may also arise. In bandwidth-limited environments, there are several additional issues to consider. First, metadata propagation also takes up bandwidth, and hence metadata exchange has to be kept to a minimum. Second, the feasible rate points in the FIT, which are defined in terms of the inputs of a node, do not directly map to the rates at the outputs of its parent, because rates may also slow down due to network delays. Lastly, we should also consider that under bandwidth limitations, shedding load at the earliest node in the server chain (even though that node itself may not be overloaded) is especially necessary. Bandwidth should not be wasted for tuples that will eventually be dropped down in the chain. Therefore, having parents shed load on behalf of their children is important. We are planning to tackle the CPU and the bandwidth problems under a common framework by treating the network slow-down as just another query operator with a certain time cost (selectivity = 1 since the network can be assumed to be reliable). Of course, this may not be so easy to achieve in shared networks due to frequent variations in cost.

7.1.2 Other Forms of Load Reduction

We can explore other ways of reducing load. Any operator that produces fewer output tuples than it receives as input could potentially be used for load shedding. Our previous load shedding techniques used drop operators which horizontally reduced data streams by discarding tuples. Other alternatives include vertically reducing data streams by using projection operators that discard columns from

tuples, or summarizing or compressing data streams by using aggregate operators. In bandwidth- or memory-limited environments, any of these alternatives would be effective.

7.1.3 Other Latency Problems

In stream processing applications, operators must avoid blocking because blocking can obviously destroy latency. Thus, it is reasonable to have an *operator timeout* [8] before it is certain that a semantic condition is reached (e.g., emit an average before all the stock prices for the 3pm report have been received). Once the timeout has happened, further tuples in the previously terminated category (e.g., 3pm reports) have no benefit and can therefore be discarded. This is a type of pre-planned load shedding that is needed when the data rates are too slow and tuples are delayed. We are interested in exploring the relationship between load shedding for high loads and load shedding (timeouts) for low loads. It would be interesting to see if there is some commonality that could be exploited.

7.1.4 Window-awareness on Joins

Similar to aggregates, joins also operate on windows of tuples. However, the semantics is quite different. A join window involves two input streams, A and B. It defines which tuples from input B are in the range of a given tuple from input A so that the join predicate can be applied on them. Unlike aggregates, where window behavior is crucial in producing subset results, this is not the main issue for joins. Dropping inputs necessarily produces a subset and load shedding on joins is mostly about controlling the size of that subset. Consider a query with an Aggregate followed by a Join. Window Drop placed before the Aggregate causes Aggregate to produce a somewhat random subset per aggregate group. This further affects the overall query result from the Join in different ways, depending on the form of the join predicate as well as tuple values.

We would like to also be able to handle query plans where joins and aggregates coexist. However, some of the techniques we developed for one operator may not apply well on the other one. For example, a semantic drop which works well with a join operator, can only be pushed across an aggregate if its predicate is defined on aggregate's group-by field. This ensures that groups will be dropped as a whole and subset guarantee will be preserved. Similarly, if we would like to push a window drop across a join by replicating it onto two join inputs, we need to make sure that these two window drops work in a coordinated fashion (i.e., decide to drop the same windows). Additionally, if we modify the join code to be aware of the window specifications, join can also perform early drops, possibly bringing additional cycle savings. We are planning to investigate these issues in detail as part of our future work.

7.1.5 Prediction-based Load Shedding

Subset-based load shedding approaches lead to gaps in query results. One way for the output application to interpret these gaps is to predict what might be missing from the result based on

what is delivered (e.g., based on linear interpolation). Using a prediction-based interpretation of the subset result also gives us an opportunity to compare our approach against the relative-error based approaches (e.g., [17]). We have conducted some preliminary experiments in this direction based on linear interpolation. Our results on real data traces show that for small gaps (i.e, low batch size), our approach produces results with lower average error. Additionally, we have also observed that larger slide values result in higher error for both approaches, but our approach seems to scale better with increasing slide. We need to conduct a more comprehensive experimental study to provide a detailed quantitative comparison.

7.1.6 Load Management on Update Streams

Most data stream processing systems model streams as append-only sequences of data elements. In this model, the application expects to receive a query answer on the complete stream. However, there are many situations in which each data element in the stream is in fact an update to a previous one, and therefore, the most recent value is all that really matters to the application. For example, for a sensor-based application that monitors the current room temperature, it is more important to deliver the latest temperature reading with low-latency than deliver the complete temperature history. Furthermore, many applications such as caching and materialized view maintenance, may reuse data until it is refreshed by a newer value. Missing or delaying a recent update would cause such applications to work with stale data, and may drive them into incorrect actions. For all these reasons, we believe that update streams must be processed in a different way. We have recently started working on the problem of efficiently and correctly processing continuous queries under such an *update-based stream data model*. The goal is to provide the most up-to-date answers to the application with the lowest latency possible, while maintaining an acceptable update frequency to minimize staleness. To achieve this, we developed a lossy tuple storage model (called an “update queue”), which under high load, may choose to sacrifice old tuples in favor of newer ones. This technique can correctly process queries with one or more sliding window operations, while efficiently handling large numbers of update keys. We would like to further investigate the update-based stream processing model. For instance, it is an interesting question how append and update streams can be accommodated in the same system, and whether the system can automatically decide which queue type to use for a given dataflow.

Bibliography

- [1] Amalgamated Insight, Inc. <http://www.aminsight.com/>.
- [2] Coral8, Inc. <http://www.coral8.com/>.
- [3] Options Price Reporting Authority (OPRA). <http://www.oprapdata.com/>.
- [4] The GNU Linear Programming Kit (GLPK). <http://www.gnu.org/software/glpk/glpk.html>.
- [5] The Internet Traffic Archive. <http://ita.ee.lbl.gov/>.
- [6] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR Conference*, Asilomar, CA, January 2005.
- [7] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hattoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A Data Stream Management System (demo description). In *ACM SIGMOD Conference*, San Diego, CA, June 2003.
- [8] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), 2003.
- [9] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The AQUA Approximate Query Answering System. In *ACM SIGMOD Conference*, Philadelphia, PA, June 1999.
- [10] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying Update Streams in a Soft Real-Time Database System. In *ACM SIGMOD Conference*, San Jose, CA, May 1995.
- [11] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the Stanford Real-time Information Processor (STRIP). *ACM SIGMOD Record*, 25(1), 1996.
- [12] Y. Ahmad, B. Berg, U. Çetintemel, M. Humphrey, J. Hwang, A. Jhingran, A. Maskey, O. Papammanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik. Distributed Operation

- in the Borealis Stream Processing Engine (demo description). In *ACM SIGMOD Conference*, Baltimore, MD, June 2005.
- [13] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-scale Stream Processing Systems. In *IEEE ICDCS Conference*, Lisboa, Portugal, July 2006.
- [14] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. *ACM Transactions on Database Systems (TODS)*, 29(1), March 2004.
- [15] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryzkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, Toronto, Canada, September 2004.
- [16] A. Ayad and J. F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *ACM SIGMOD Conference*, Paris, France, June 2004.
- [17] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE Conference*, Boston, MA, March 2004.
- [18] S. Babu, L. Subramanian, and J. Widom. A Data Stream Management System for Network Traffic Management. In *ACM Workshop on Network-Related Data Management (NRDM)*, Santa Barbara, CA, May 2001.
- [19] S. Bajaj, L. Breslau, and S. Shenker. Uniform versus Priority Dropping for Layered Video. In *ACM SIGCOMM Conference*, Vancouver, Canada, August 1998.
- [20] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal Special Issue on Data Stream Processing*, 13(4), 2004.
- [21] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20(4), 1997.
- [22] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. IETF RFC 2475, December 1998.
- [23] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: An Overview. IETF RFC 1633, June 1994.
- [24] D. Carney, U. Çetintemel, 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 *VLDB Conference*, Hong Kong, China, August 2002.

- [25] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.
- [26] S. Cen, C. Pu, and J. Walpole. Flow and Congestion Control for Internet Streaming Applications. In *ACM/SPIE Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 1998.
- [27] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing Using Wavelets. In *VLDB Conference*, Cairo, Egypt, September 2000.
- [28] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, Asilomar, CA, January 2003.
- [29] S. Chandrasekaran and M. J. Franklin. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *VLDB Conference*, Toronto, Canada, September 2004.
- [30] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On Random Sampling over Joins. In *ACM SIGMOD Conference*, Philadelphia, PA, June 1999.
- [31] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *ACM SIGMOD Conference*, Dallas, TX, May 2000.
- [32] W. Chu, H. Yang, and G. Chow. A Cooperative Database System (CoBase) for Query Relaxation. In *International Conference on Artificial Intelligence Planning Systems*, Edinburgh, Scotland, UK, May 1996.
- [33] C. L. Compton and D. L. Tennenhouse. Collaborative Load Shedding for Media-Based Applications. In *International Conference on Multimedia Computing and Systems (ICMCS)*, Boston, MA, May 1994.
- [34] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, first edition, 1998.
- [35] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing Over Data Streams. In *ACM SIGMOD Conference*, San Diego, CA, June 2003.
- [36] A. Demers, J. Gehrke, and M. Riedewald. The Architecture of the Cornell Knowledge Broker. In *Symposium on Intelligence and Security Informatics (ISI-2004)*, Tucson, AZ, June 2004.
- [37] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *VLDB Conference*, Toronto, Canada, September 2004.
- [38] Digital Bibliography & Library Project. <http://www.informatik.uni-trier.de/ley/db/>.

- [39] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *ACM SIGMOD Conference*, Madison, Wisconsin, June 2002.
- [40] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2), 2003.
- [41] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Database Applications. *Journal of Computer and System Sciences*, 31(2), September 1985.
- [42] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-In Systems: The HiFi Approach. In *CIDR Conference*, Asilomar, CA, January 2005.
- [43] S. Ganguly. Design and Analysis of Parametric Query Optimization Algorithms. In *VLDB Conference*, New York City, NY, August 1998.
- [44] M. Garofalakis and P. B. Gibbons. Approximate Query Processing: Taming the Megabytes. In *VLDB Conference*, Rome, Italy, September 2001.
- [45] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams Databases. In *ACM SIGMOD Conference*, Santa Barbara, CA, May 2001.
- [46] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-pass Summaries for Approximate Aggregate Queries. In *VLDB Conference*, Roma, Italy, September 2001.
- [47] Glossary of Astronomy Terms. <http://www.seasky.org/astronomy/astronomyglossary.html>.
- [48] L. Golab and M. T. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2), June 2003.
- [49] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *ACM SIGMOD Conference*, Philadelphia, PA, June 1999.
- [50] J. Hansson and S. H. Son. Overload Management in RTDBs. In K. Lam and T. Kuo, editors, *Real-Time Database Systems: Architecture and Techniques*. Kluwer Academic Publishers, 2001.
- [51] J. M. Hellerstein, R. Avnur, A. Chou, C. Olston, V. Raman, T. Roth, C. Hidber, and P. J. Haas. Interactive Data Analysis: The Control Project. *IEEE Computer*, 32(8), August 1999.
- [52] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *ACM SIGMOD Conference*, Tucson, AZ, May 1997.
- [53] A. Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *VLDB Conference*, Hong Kong, China, August 2002.

- [54] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *VLDB Conference*, Vancouver, Canada, August 1992.
- [55] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *VLDB Journal*, 6(2), 1997.
- [56] Y. E. Ioannidis and V. Poosala. Histogram-based Approximation of Set-valued Query-Answers. In *VLDB Conference*, Edinburgh, Scotland, UK, September 1999.
- [57] V. Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4), August 1988.
- [58] A. Jain, E. Y. Chang, and Y. Wang. Adaptive Stream Resource Management using Kalman Filters. In *ACM SIGMOD Conference*, Paris, France, June 2004.
- [59] R. Jain. Congestion Control in Computer Networks: Issues and Trends. *IEEE Network Magazine*, 4(3), 1990.
- [60] J. Kang, J. Naughton, and S. Viglas. Evaluating Window Joins over Unbounded Streams. In *IEEE ICDE Conference*, Bangalore, India, March 2003.
- [61] K. Kang, S. Hyuk Son, J. A. Stankovic, and T. F. Abdelzaher. A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, Vienna, Austria, June 2002.
- [62] B. Kao and H. Garcia-Molina. An Overview of Real-Time Database Systems. In W. A. Halang and A. D. Stoyenko, editors, *NATO Advanced Study Institute on Real-Time Computing*. Springer-Verlag, 1994.
- [63] C. Krasic, J. Walpole, and W. Feng. Quality-Adaptive Media Streaming by Priority Drop. In *ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Monterey, CA, June 2003.
- [64] C. Krasic, J. Walpole, and C. Pu. QoS Scalability for Streamed Media Delivery. Technical Report CSE-99-011, Oregon Graduate Institute School of Science and Engineering, 1999.
- [65] Y. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *VLDB Conference*, Toronto, Canada, September 2004.
- [66] U. Leonhardt and J. Magee. Multi-sensor Location Tracking. In *International Conference on Mobile Computing and Networking (MobiCom)*, Dallas, TX, October 1998.
- [67] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

- [68] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *ACM SIGCOMM Computer Communication Review*, 32(3), July 2002.
- [69] Mesquite Software, Inc. CSIM18 Simulation Engine. <http://www.mesquite.com/>.
- [70] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, Asilomar, CA, January 2003.
- [71] F. Olken and D. Rotem. Random Sampling from Databases: A Survey. *Statistics and Computing*, 5(1), March 1995.
- [72] G. Özsoyoğlu, S. Guruswamy, K. Du, and W. Hou. Time-Constrained Query Processing in CASE-DB. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(6), December 1995.
- [73] G. Özsoyoğlu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4), August 1995.
- [74] O. Papaemmanouil, Y. Ahmad, U. Çetintemel, J. Jannotti, and Y. Yıldırım. Extensible Optimization in Overlay Dissemination Trees. In *ACM SIGMOD Conference*, pages 611–622, Chicago, IL, June 2006.
- [75] N. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1), 1999.
- [76] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *IEEE ICDE Conference*, Atlanta, GA, April 2006.
- [77] V. Raman, B. Raman, and J. M. Hellerstein. Online Dynamic Reordering. *VLDB Journal*, 9(3), 2000.
- [78] R. L. Read, D. S. Fussell, and A. Silberschatz. A Multi-Resolution Relational Data Model. In *VLDB Conference*, Vancouver, Canada, August 1992.
- [79] F. Reiss and J. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. Technical Report IRB-TR-04-004, Intel Research, February 2004.
- [80] F. Reiss and J. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *IEEE ICDE Conference*, Tokyo, Japan, April 2005.
- [81] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

- [82] L. Schwiebert, S. K. S. Gupta, and J. Weinmann. Research Challenges in Wireless Networks of Biomedical Sensors. In *International Conference on Mobile Computing and Networking (MobiCom)*, Rome, Italy, July 2001.
- [83] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *VLDB Conference*, Bombay, India, September 1996.
- [84] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly-Available, Fault-Tolerant, Parallel Dataflows. In *ACM SIGMOD Conference*, Paris, France, June 2004.
- [85] U. Srivastava and J. Widom. Memory Limited Execution of Windowed Stream Joins. In *VLDB Conference*, Toronto, Canada, September 2004.
- [86] Stream Query Repository. <http://www-db.stanford.edu/stream/sqr/>.
- [87] StreamBase Systems, Inc. <http://www.streambase.com/>.
- [88] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *NSDI Conference*, San Francisco, CA, March 2004.
- [89] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a Sensor Network Expedition. In *European Workshop on Wireless Sensor Networks (EWSN)*, Berlin, Germany, January 2004.
- [90] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
- [91] N. Tatbul, M. Buller, R. Hoyt, S. Mullen, and S. Zdonik. Confidence-based Data Management for Personal Area Sensor Networks. In *VLDB Workshop on Data Management for Sensor Networks (DMSN)*, Toronto, Canada, September 2004.
- [92] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: Scalable Load Shedding Techniques for Distributed Stream Processing . Technical Report CS-06-13, Brown University, Computer Science, November 2006.
- [93] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.
- [94] N. Tatbul and S. Zdonik. Dealing with Overload in Distributed Stream Processing Systems. In *IEEE International Workshop on Networking Meets Databases (NetDB'06)*, Atlanta, GA, April 2006.
- [95] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *International Conference on Very Large Data Bases (VLDB'06)*, Seoul, Korea, September 2006.

- [96] Y. Tu, S. Liu, S. Prabhakar, and B. Yao. Load Shedding in Stream Databases: A Control-Based Approach. In *International Conference on Very Large Data Bases (VLDB'06)*, Seoul, Korea, September 2006.
- [97] Y. Tu, S. Prabhakar, A. K. Elmagarmid, and R. Sion. QuaSAQ: An Approach to Enabling End-to-End QoS for Multimedia Databases. In *EDBT Conference*, Crete, Greece, March 2004.
- [98] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3), 2003.
- [99] J. S. Vitter and M. Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *ACM SIGMOD Conference*, Philadelphia, PA, June 1999.
- [100] S. V. Vrbsky. A Data Model for Approximate Query Processing of Real-Time Databases. *Data and Knowledge Engineering*, 21(1), December 1996.
- [101] S. V. Vrbsky and J. W. S. Liu. APPROXIMATE: A Query Processor That Produces Monotonically Improving Approximate Answers. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 5(6), December 1993.
- [102] J. Walpole, L. Liu, D. Maier, C. Pu, and C. Krasic. Quality of Service Semantics for Multimedia Database Systems. In *IFIP 8th Working Conference on Database Semantics (DS-8): Semantic Issues in Multimedia Systems*, Rotorua, New Zealand, January 1999.
- [103] M. Welsh and D. E. Culler. Adaptive Overload Control for Busy Internet Servers. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, March 2003.
- [104] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.
- [105] A. T. Whitney and D. Shasha. Lots o' Ticks: Real-Time High Performance Time Series Queries on Billions of Trades and Quotes. In *ACM SIGMOD Conference*, Santa Barbara, CA, May 2001.
- [106] J. Xie, J. Yang, and Y. Chen. On Joining and Caching Stochastic Streams. In *ACM SIGMOD Conference*, Baltimore, MD, June 2005.
- [107] Y. Zhu and D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *VLDB Conference*, Hong Kong, China, August 2002.