Abstract of “Pulse: Database Support for Efficient Query Processing Of Temporal Polynomial Models” by Yanif Ahmad, Ph.D., Brown University, May 2009.

This thesis investigates the practicality and utility of mathematical models to represent continuous and occasionally unavailable data stream attributes, and processing relational-style queries in a stream processing engine directly on these models. We present Pulse, a framework for processing continuous queries over stream attributes modeled as piecewise polynomial functions. We use piecewise polynomials to provide a compact, approximate representation of the input dataset and provide query language extensions for users to specify precision bounds to control this approximation. Pulse represents queries as simultaneous equation systems for a variety of relational operators including filters, joins and standard aggregates. In the stream context, we continually solve these equation systems as new data arrives into the system. We have implemented Pulse on top of the Borealis stream processing engine and evaluated it on two real-world datasets from financial and moving object applications. Pulse is able to achieve significant performance improvements by processing queries directly on the mathematical representation of these polynomials, in comparison to standard tuple-based stream processing, thereby demonstrating the viability of our system in the face of having to meet precision requirements.

In addition to our primary contribution of describing the core design and architecture of Pulse, this thesis presents a selectivity estimator and a multi-query optimizer to scale query processing capabilities. Our selectivity estimator uses histograms defined on a parameter space of polynomial coefficients for estimation, passing selectivities to our multi-query optimizer which may then determine how to construct a global query plan that shares work across individual queries. We evaluate these components on both a synthetic dataset and a financial dataset. Our experiments show that our optimization mechanisms provide significant reductions in processing overhead, and that our estimation algorithm provides an accurate and low overhead estimator for selective operators, that can be enhanced by sampling, while also being a general technique that can handle operators such as min and max aggregates, where sampling is known to be inaccurate.

Pulse: Database Support for Efficient Query Processing Of Temporal Polynomial Models

by
Yanif Ahmad
B. Eng., Imperial College of Science, Technology and Medicine, UK, 2001.
Sc. M., Brown University, 2004.

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 2009

© Copyright 2009 by Yanif Ahmad

This dissertation by Yanif Ahmad 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 ______________ ________________________________
Uğur Çetintemel, Director

Recommended to the Graduate Council

Date ______________ ________________________________
Stanley B. Zdonik, Reader

Date ______________ ________________________________
John Jannotti, Reader

Date ______________ ________________________________
Samuel Madden, Reader
CSAIL, MIT

Approved by the Graduate Council

Date ______________ ________________________________
Sheila Bonde
Dean of the Graduate School

Acknowledgments

Looking out of my office window, the snow-covered streets reminds me of an optimistic feeling I had prior to starting graduate school – the promise of fun should I choose to dive into it. My time at Brown has fulfilled that promise in so many more ways than I imagined, and this is due in no uncertain terms to the wonderful colleagues, friends and family that I have had the privilege of spending time with in these last years.

I would first and foremost like to thank my mentor and advisor, Uğur Çetintemel. Uğur has been a source of unending support, motivation, inspiration and patience over the years, in my eyes the model advisor. His amicable nature and sense of humor will always be a constant reminder to take pleasure and enjoy research in light of its ups and downs. He conveyed an invaluable lesson in the need to strike a balance between building future-facing systems while grounding long term potential impact with the needs of today's applications, helping me realize the importance of taste and diversity as key elements of research. I truly feel lucky to have had the opportunity to work with Uğur and look forward to collaborating and continuing to learn from him in the future. Finally, no acknowledgment of Uğur could be complete without reference to his football skills, and I am grateful for his push to start the Brown CS Soccer Team whose intramural successes were one of the highlights of my time at Brown. Thank you Uğur for all of your time and energy.

My thesis committee consisted of Stan Zdonik, John Jannotti and Sam Madden, each of whom in their own ways have greatly enriched my experience at Brown, and have been fantastic role models as researchers and provided critical insights over the years. I thank them all for helping me grow as a researcher. Of those researchers with whom I have had the opportunity to work closely, Stan has set the high bar for the kinds of goals any young database researcher should seek to accomplish in a research career. I hope to be able to reproduce the elegance and simplicity in designing database systems that I have witnessed as part of Stan's work during the Aurora/Borealis research meetings. Chatting in the hallways with Stan frequently brought up a valuable reminder, that there are pursuits outside the

academic confines in putting research ideas through its paces in industry.

I thank John for all of his efforts in our collaboration on the topics we pursued earlier in my time as a PhD student on the SAND project. John continually amazed me with his ability to grasp and find the holes in the details of my work, providing me with greater appreciation of doing research online at the whiteboard and of having an example or two in the back of my pocket. I am grateful John provided me the opportunity to be a Teaching Assistant for his CS161 course on building high-performance systems, and for his continual support over the years through numerous references for internships. Working with John has provided me confidence in my abilities as a systems researcher, and I hope to continue a side line of research collaborations in networks and distributed systems.

While he may not remember, I first met Sam during his job interview at Brown and proceeded to ask him his secret to completing the PhD in four years. Now I do not remember his answer (perhaps I would have graduated sooner if I had), but I recall being impressed with Sam's accomplishments on the TinyDB system and from that day on have often found myself inspired by Sam's research. I would like to thank Sam for his unsapping willingness to attend meetings, as well as my thesis proposal and defense, as well as providing detailed feedback on both this thesis research and document. I hope to continue to bounce ideas off Sam and look forward to future research chats.

My collaborators in the New England area truly created an enriching academic environment, and in particular I would like to thank the members of the Brown Database Group, and the Aurora/Borealis projects: Olga Papaemmanouil, Jeong-Hyon Hwang, Nesime Tatbul, Magda Balazinska, Bradley Berg, Jennie Rogers, Alex Rasin, Anurag Maskey, Mitch Cherniack, and Mike Stonebraker. I would also like to thank Suman Nath and Sandeep Tata for providing me the opportunity to intern at Microsoft Research and IBM Research respectively, for their mentoring services and all of their efforts in our collaborations. Brown CS TStaff and AStaff, especially Lori Agresti, Kathy Kirman, Jeff Coady and Max Salvas, were extremely patient with all of my requests, I am extremely grateful for their responsiveness and willingness to provide administrative support.

The PhD experience is so much more than the research and collaborations making up the academics, it is an opportunity to meet many gifted and unique individuals who I have the pleasure of naming as my friends. My time at Brown was made immeasurably richer by Eric Koskinen and Vanessa Vigna, Peter Sibley, Gregoire Dooms and Julie Cailleau, Manuel Cebrian, Luc Mercier, and Mert Akdere. In addition I would like to thank Daniel Acevedo, Tudor Antoniu, Russell Bent, and Frank Wood for being wonderful officemates, and the following for their companionship: Aris Anagnostopoulos, Sebastien Chan-Tin,

Jason Mallios, Liz Marai, Tomer Moscovich, Manos Renieris, Stefan Roth, Alex Zgolinski.

Finally and most importantly, I would like to thank my parents, Ali and Tahmina, and my extended family, especially Tawhid, Tawfique and Taimor, for all of the help and encouragement they have provided through the years. Without the doors they opened, I could not have pursued this path so freely, and with such single-mindedness. Thank you.

To my parents, Ali and Tahmina

Contents

List of Figures xi

1 Introduction 1

1.1 Motivating Examples and Challenges . . . 4

1.2 Thesis Contributions and Outline . . . 6

2 Polynomial-Driven Query Processing 7

2.1 System Model . . . 7

2.1.1 Predictive Processing. . . . 7

2.1.2 Historical Processing. . . . 8

2.2 Data Stream Model . . . 8

2.2.1 Modeled Attributes. . . . 8

2.2.2 Key attributes. . . . 9

2.3 Selective Operators . . . 10

2.4 Aggregate Operators . . . 12

2.4.1 Min and Max Window Functions . . . 12

2.4.2 Sum and Average Window Functions . . . 12

2.5 Continuous Segment Processing . . . 14

2.6 Enforcing Precision Bounds . . . 15

2.6.1 Query Output Semantics . . . 16

2.6.2 Query Inversion . . . 17

2.6.3 Split Functions . . . 18

2.7 Experimental Evaluation . . . 20

2.7.1 Operator Benchmarks . . . 20

2.7.2 Query Processing Performance . . . 23

2.7.3 NYSE and AIS Processing Evaluation . . . 24

2.7.4 Precision Bound Sensitivity Evaluation 27

3 Selectivity Estimation for Continuous Function Processing 29

3.1 Cost-Based Query Optimization 29

3.1.1 Risk-Based Cost Models 30

3.1.2 Multi-Query Optimizer Overview 31

3.2 Segmentation-Oriented Selectivity Estimation 32

3.2.1 Dual Space Representation of Polynomials 33

3.2.2 Selectivity Estimation in the Dual Space 35

3.2.3 Query-Based Histogram Derivation 37

3.3 Histogram-Based Estimation of Intermediate Result Distributions 38

3.3.1 Filters 41

3.3.2 Joins 43

3.3.3 Aggregates 45

3.3.4 Histogram-Based Selectivity Bounds 49

3.4 Experimental Evaluation 50

3.4.1 Dataset Analysis 51

3.4.2 Synthetic Dataset 59

4 Adaptive Multi-Query Optimization with Segment Selectivities 62

4.1 Covered Sharing 63

4.2 Union Sharing 65

4.3 Ordering 69

4.4 Processing Cost Model 70

4.5 Detecting Optimization Opportunities 72

4.5.1 Shared Operator Algorithm and Expression Adaptation Conditions . 72

4.5.2 Shared Instance Creation and Adaption Conditions 75

4.5.3 Reordering Conditions for Shared Operators 77

4.6 Collecting Query Statistics 79

4.6.1 Cost model and collection mechanism 79

4.6.2 Adaptation conditions 81

4.7 Condition Evaluation 82

4.8 Experimental Evaluation 83

5 Related Work **89**
5.1 Data Stream Management Systems 89
5.2 Databases for Sensor Data Processing 91
5.2.1 Model-Based Query Processing 91
5.2.2 Querying Regression Functions 92
5.2.3 Model-Driven Query Processing in Probabilistic Databases 92
5.2.4 Querying Time-Series Models 93
5.3 Moving-object Databases 93
5.3.1 Moving-object Indexes 94
5.4 Constraint Databases 94
5.4.1 Dedale 94
5.4.2 Optimization queries 95
5.5 Approximate Query Processing 95
5.5.1 Stream Filtering 95
5.5.2 Approximation in Stream Processing Engines 97
5.6 Selectivity Estimation 98
5.6.1 Selectivity Estimation for Intermediate Result Cardinalities 98
5.6.2 Sampling-Based Selectivity Estimation 98
5.6.3 Handling Correlations and Functional Dependencies 99
5.7 Multi-Query Optimization 100
5.7.1 Multi-Plan Optimization 100
5.7.2 Sharing in Stream Processing Engines 101
5.7.3 Staged Databases 101
5.8 Mathematical Software and Computer Algebra Systems 102
5.8.1 Mathematica and Maxima 102
5.8.2 Matlab and Octave 103

6 Conclusions **104**

Bibliography **111**

List of Figures

1.1 Query processing on continuous functions 4

2.1 Pulse transforms predicates in selective operators to determine a system of equations whose solution yields the time range containing the query result. 8

2.2 A geometric interpretation of the continuous transform, illustrating predicate relationships between models for selective operators, and piecewise composition of individual models representing the continuous internal state of a `max` aggregate. 11

2.3 High level overview of Pulse's internal dataflow. Segments are either given as inputs to the system or determined internally, and processed as first-class elements. 16

2.4 Microbenchmark for a filter operator with a 1% error threshold. 21

2.5 Microbenchmark an aggregate operator with a 1% error threshold. 22

2.6 Continuous-time and discrete processing overhead comparison for an aggregate operator. 23

2.7 Microbenchmark for a join operator with a 1% error threshold. 24

2.8 Continuous-time and discrete processing overhead comparison for a join operator. 25

2.9 Historical aggregate processing throughput comparison with a 1% error threshold. 25

2.10 Continuous-time processing of the NYSE dataset, with a 1% error threshold. 26

2.11 Continuous-time processing of the AIS dataset, with a 0.05% error 26

2.12 Continuous-time processing of the NYSE data at 3000 tuples/second. . . . 28

3.1 Qualitative comparisons of query plans on a risk-cost space. 31

3.2 Selectivity estimation definition illustrations. 34

3.3 Bounding validity range correction factor. 36

3.4 Parameter spaces of segments in the synthetic i) normal distribution, and ii) uniform distribution 51
3.5 Synthetic dataset generation parameters 52
3.6 NYSE dataset characteristic parameters 53
3.7 Segment parameter spaces for seven stocks from the NYSE dataset, stock ids are: i) 12, ii) 3762 iii) 4878 iv) 6849 v) 6879 vi) 6973 vii) 8239. 54
3.8 Comparison of accuracy vs. performance tradeoff for histogram technique and sampling on the NYSE dataset. 55
3.9 Absolute error differences between sampling at a variety of rates, and histogram-based estimation using a large number of bins. 56
3.10 Estimation evaluation overhead for i) histogram-based and ii) sampling-based techniques on the NYSE dataset. 57
3.11 Comparison of accuracy vs. performance tradeoff for histogram technique and sampling on the synthetic dataset. 58
3.12 Absolute error differences between sampling at a variety of rates, and histogram-based estimation using a large number of bins on the synthetic dataset. . . 59
3.13 Estimation evaluation overhead for i) histogram-based and ii) sampling-based techniques on the synthetic dataset. 60

4.1 Cover-sharing example. 63
4.2 Union-sharing example. 67
4.3 State transition diagram for selecting sharing algorithm type. 73
4.4 Comparison of cover-shared and unshared execution at a variety of upper bounds for query selectivities. 85
4.5 Processing cost differences between unshared and cover-shared execution for query workloads with a variety of covering selectivities. 86
4.6 Processing cost comparison for union-sharing and other mechanisms in a 2-operator chain, with varying selectivity configurations at each operator. Note here high selectivity configuration implies selectivities close to 0.0, spread selectivities indicates randomly chosen selectivities in [0.0, 1.0], and low refers to selectivities near a value of 1.0. Additionally the upstream operator, A, is cover-shared, and for this operator, the covering selectivity is roughly equivalent for high and spread selectivity configurations. 87

6.1 Example polynomial type sampling function. 110

Chapter 1

Introduction

With the emergence of sensor databases that are asked to manage significant volumes of data sampled from physical processes, there has been much recent work in the database community to address a variety of challenges prevalent in, but not limited to, sensor environments. Some of the eminent challenges includes handling noisy and uncertain data [49, 54], handling node and network failures and the subsequent missing inputs in the data stream [84, 86], and exploiting properties of the environment for query and communication optimization [48, 31]. Many of the solutions to these challenges have used some form of mathematical model to represent the values of attributes in database schemas, for example probabilistic models to represent uncertainty and reduce network overhead during query processing, as well as regression models to interpolate missing tuples.

There are numerous other uses for modeled attributes both internally to a database system, and in terms of the functionality they can provide for end users. The database internals application we consider in this thesis is that of using models for query processing, primarily motivated in the context of the physical environments in which sensor networks operate, and the continuous processes they monitor and query. The data attributes present in sensor environments exhibit two key characteristics, they are continuous and they may have strong dependencies on other attributes, for example time and space attributes. Our techniques are generally applicable to any domain which exhibits similar characteristics. There has been at best limited support for continuous attributes in existing database systems. The models we consider for representing our data attributes account for these characteristics, treating modeled attributes as continuous functions of a set of independent attributes.

Prior to diving into the nature of our models and our use for them, one question that naturally arises is where do these models come from? We perceive two possibilities: the

models can be trained or learned from the raw inputs, or can be provided by domain experts. The former has been the study of much recent work in the artificial intelligence and machine learning communities [34, 12] and the integration of such learning techniques is a burgeoning topic in the database community [51]. In our view the latter option, while this may place a greater burden on the application developer, has been a long-standing limitation of a database's effectiveness as a tool in the domains and applications that heavily use sensor data such as science and engineering.

Using these sensor data monitoring and analysis applications as an example, there is an wealth of literature in the physical sciences developing and experimentally validating mathematical models that represent the behavior of attributes commonly sensed and processed as part of sensor applications. To this day, this domain-specific knowledge has either been used externally to the database system, or has been awkwardly, inefficiently and repeatedly incorporated into the system through the use of stored procedures and other extensible features of database systems. This concept underlies the overarching vision for this thesis. There are large number of data modeling techniques, at a high-level based on calculus and statistics, that have been developed across many disciplines of mathematics, science and engineering. We view that there is a place for such techniques and the properties they entail for attributes inside database systems, side-by-side with the relational algebra used to declaratively manipulate attributes.

In turn database systems can bring several advantages to end users developing sensor data monitoring and analysis applications. These features include providing a declarative interface separating logical and physical concerns to support rapid development and prototyping without sacrificing on efficiency for compute-intensive tasks through under-the-hood query optimization. Furthermore database systems have been strong proponents of shared-nothing architectures for parallel, scalable query processing with the key being to scale out rather than up. Finally database systems have highly tested, robust implementations of key features for multi-user systems, particularly a well-defined concurrency model and concurrency control, eliminating the need to reimplement this hard-to-get-right functionality across multiple applications. These features are integral to an effective tool for data management.

This thesis represents an initial bottom-up approach to tackling our overarching vision, where we develop a query processing framework called Pulse [4, 3] that is capable of handling two specific types of mathematical models, polynomials and differential equations. Due to the attributes' continuous nature, representing each and every attribute value explicitly is not an option as is the case with discretized attributes. Rather our models are

expressed with a symbolic equation representation, and we use the same form internally during query processing. Note this is in contrast to the high-level approach adopted by works incorporating the aforementioned learning techniques and the underlying representation of uncertainty and probabilities required there. In these works the relational model can be use to represent the model with the functionality required of these models implemented using standard database operators on the relational representation. In our approach, we are instead attempting to define and maintain the relational data model on top of a fundamentally different data representation.

Pulse is capable of applying operators based on the relational algebra to the native equation representation of these models, rather than on raw inputs or a discretized form of the models. Modeled attributes provide two interrelated advantages for query processing. Our query processor is capable of using attribute models to compactly and approximately represent data, and can then directly and efficiently perform processing to provide approximate query results with much lower overheads, and thus higher performance, than processing raw input data. Furthermore due to the continuous nature of the data and our models, they eliminate application sensitivities to the discretization of the raw input data.

We have designed Pulse primarily for use in a stream processing context, where the common traits of stream applications include high-volume input data streams which are then processed on-the-fly by continuous queries, to provide low latency and high throughput results. Stream processing engines have become popular tools in several domains including finance, network monitoring, environmental and infrastructure monitoring, real-time inventory tracking and massively multiplayer games amongst many other applications. We refer interested readers to the Aurora/Borealis, STREAM and TelegraphCQ projects [2, 1, 65, 18] for more detailed information of the basic design, architecture and functionality provided by stream processing engines. Several of these domains have immediate need for database support for domain-specific knowledge, for example financial applications operating on stock market models. Our approach can also be applied to problems that naturally fit a streaming model in domains that have thus not been considered by general stream processing engines, for example scientific simulation, experimentation and validation applications, moving object applications, and engineering design and testing applications. In the remainder of this chapter we present two examples capturing the concept of processing relational queries over mathematical models. We use these examples to state the aims of this thesis, before outlining our specific contributions and how they accomplish the goals described.

1.1 Motivating Examples and Challenges

We now describe this fundamental concept of processing relational queries on mathematical representations of attributes, before presenting the system design challenges that this form of query processing entails. As an example, consider a financial application driven by stock prices. While we've primarily talked about sensor data thus far, we use this scenario to illustrate that our techniques are applicable beyond the sensor environment. Financial applications pose many different forms of queries on stock data to determine trading strategies, and for illustrative purposes, consider a simple threshold condition on a stock price that indicates whether or not to sell the stock:

```
select * from stocks where symbol = 'IBM' and price < 125
```

In existing stream processing systems, stock feeds indicating trades made on the IBM stock are continuously pushed into the system and processed through a continuous query implementing the above logic. The stock price from IBM can be modeled as a continuous function, for example for forecasting purposes. This is illustrated below for a piecewise polynomial representation of the discrete stock price stream:

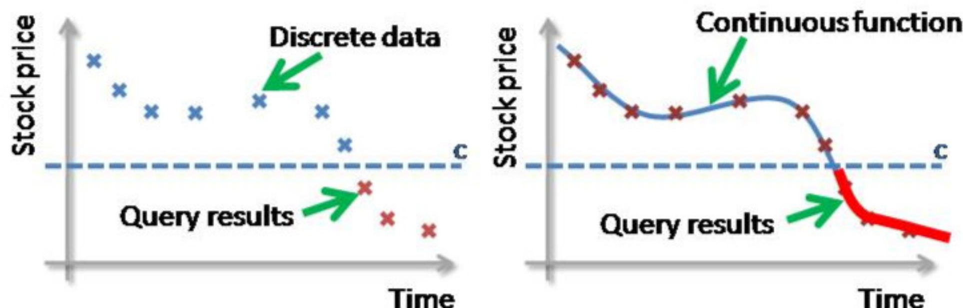

Figure 1.1: Query processing on continuous functions

In this example we would like to apply the threshold condition to the continuous function representation of stock prices rather than the discrete results, and determine when the function crosses the threshold. The same principle applies to join and aggregate operators in addition to the filter operator that would be used to implement this threshold condition. Note that in this diagram we have indicated that the result of such a query is itself a continuous function rather than a discrete tuple. Having a closed-form approach enables operator composition, for example we could apply an aggregate to the continuous function following this filter operation. We return to the type of results produced at the edges of the stream processor for downstream applications and clients in the next chapter when

describing the whole system model.

Given the above example we can begin to see some of the challenges posed for such a query processor. At a high-level we can categorize the challenges into four sub-areas:

1. **semantic issues**, which include what kinds of models should we choose to support, and how do their representations correspond to the set-oriented data model on top of which we've based the relational calculus?

2. **algorithmic issues**, which cover that once we have decided on our model types, how can we then apply relational operators directly to the model's intrinsic representation, treating the model as a first-class entity in the database system, and subsequently how does query processing work?

3. **architectural and optimization issues** ranging from what abstractions can we extract for putting together a general framework for processing models and what novel model properties can we leverage for improving query processing efficiency?

4. **usability issues**, where given there are numerous existing tools for determining models, and clear advantages in declarative style programming as seen in database community experiences, how can we mesh these two factors together?

To emphasize a second critical component in our model-driven query processing, we mentioned in that these models are approximations to the raw data in the above challenges. Thus our query processor's semantics should provide a well-defined approach to handling this approximation both in terms of the processing it performs and the results it produces. Furthermore, as we will see for both polynomial and differential equation models, error handling provides opportunities for trading off the approximation provided in query results and the system overhead in computing these results. The key to our approach lies in assuming users are able to specify the precision they desire in query results, and then using this precision specification to determine the approximation performed by each operator in a query plan. We present a basic approach for allocating precision bounds to each operator in the case of polynomial models, and extend this for numerical methods used to solve differential equations, where the numerical methods themselves produce an approximation to the intended model.

1.2 Thesis Contributions and Outline

This thesis presents the design of a stream processor for processing relational operators on polynomial and differential equation datatypes. While there are many components to a stream processing engine, we focus primarily on the query processor and query optimizer. Chapter 2 presents the Pulse stream processor for attributes represented by a piecewise polynomial datatype. In this system, data streams are made up of multiple piecewise polynomials, one for every key value (for example, a piecewise polynomial for the 'IBM' stock and another for the 'APPL' stock). The first part of this chapter describes the algorithms and implementation of relational processing for polynomials, cover filter, join and aggregate operators. The second part focuses on the bound inversion algorithm to handle precision bounds specified by users at the outputs. The key idea here is to determine a precision bound at the inputs so that the query does not have to be processed on both the discrete and polynomial representation to check the bounds. This system is implemented and evaluated on top of the Borealis stream processing engine. We use two datasets, a NYSE stock feed dataset and a naval vessel location dataset captured by the US Coast Guard with the AIS tracking system.

Given the different nature of the query processing we perform in the above system, we then consider the question of how to perform query optimization in this context. As described in Chapter 3, one of the key challenges we perceive is that of selectivity estimation. Selectivity estimators are used to predict the cost of running an alternative query plan to the currently executing plan, since there are no measured selectivities available for the computing the cost of the alternative plan. We present a selectivity estimator customized to the polynomials we process that is based on maintaining histograms over the coefficients of these polynomials. We then use this selectivity estimator in an actual optimizer, specifically a multi-query optimization technique based on sharing common work. We present a plan-based approach that finds operators that are compatible for sharing, and dynamically selects groups of these operators to actually share based on a cost model. Our cost model represents processing overhead and applies both the commonly used definition of overhead as a product of selectivities and unit processing costs, as well as a notion of risk. This risk metric captures that we are using a selectivity estimator, where our estimates can be erroneous and thus should be considered to lie within a range of values, rather than being a single precise averaged value.

Chapter 2

Polynomial-Driven Query Processing

Continuous-time models provide two distinctive properties for use in query processing: they facilitate random access to data at arbitrary points in time, and enable a compact representation of the data as model parameters. In this section we present an overview of the nature of the application types, data streams and queries we support.

2.1 System Model

In this section, we discuss two novel uses of models in the query execution model of a stream processing engine in terms of both functionality and performance.

2.1.1 Predictive Processing.

In the predictive processing scenario, Pulse uses its modeling component to generate the continuous-time models for unseen data values off into the future, processes these predicted inputs, and generates predicted query results, all before the real input data becomes available from external sources. This style of predictive processing has important uses both from the end-application perspective (e.g., a traffic monitoring application can predict congestions at on road segments and send alerts to drivers) and system optimization perspective (e.g., predictive results can mask I/O latencies, or network latencies in wide-area network games by pre-fetching).

Query: `SELECT * from A MODEL` $A.x = A.x + A.vt$
`JOIN B MODEL` $B.y = B.vt + B.at^2$
`ON(A.x < B.y)`

Transformation	Description
$A.x < B.y$	
$A.x - B.y < 0$	difference equation
$A.x + A.vt - (B.vt + B.at^2) < 0$	substitute models
$A.x + (A.v - B.v)t - B.at^2 < 0$	factor time variable t

Figure 2.1: Pulse transforms predicates in selective operators to determine a system of equations whose solution yields the time range containing the query result.

2.1.2 Historical Processing.

The second scenario is off-line historical data analysis that involves running a large number of "parameter sweeping" or "what-if" queries (common in the financial services domain). Applications replay a historical stream as input to a large number of queries with different user-supplied analytical functions or a range of parameter values. The results are then typically compared against each other and what was obtained in the past, to identify the "best" strategy or parameters to use in the future. In historical processing, Pulse's modeling component is used to generate a continuous-time model of the historical stream that can be stored and used as an input to all historical queries. Thus, the cost of modeling can be amortized across many queries.

2.2 Data Stream Model

Pulse adopts the following assumptions on the uniqueness and temporal properties of data stream attributes.

2.2.1 Modeled Attributes.

For predictive processing, Pulse supports declarative model specification as part of its queries via a `MODEL`-clause, as shown in Figure 2.1. Query developers provide *symbolic* models defining a modeled stream attribute in terms of other attributes on the same stream and a variable t. For example in Figure 2.1, stream A has a modeled attribute $A.x$ defined in terms of *coefficient* attributes $A.x$ and $A.v$. We allow the self-reference to attribute $A.x$

since we build *numerical* models from actual input tuples where the values of all coefficient attributes are known. In this example, the model $A.x = A.x + A.vt$ represents the x-coordinate a moving object as its position varies over time from some initial position. We consider time-invariant piecewise polynomial models since they are often used for simple and efficient approximation. The symbolic form of a general nth degree polynomial for a modeled attribute a is: $a(t) = \sum_{i=0}^{n} c_{a,i} t^i$. To ensure a closed operator set, we restrict the class of polynomials supported to those with non-negative exponents, since it has been shown that semi-algebraic sets are not closed in the constraint database literature [57]. In historical processing our modeling component computes coefficient attribute values internally.

Temporal attributes.. We assume each input stream S includes two temporal attributes, a reference attribute denoting a monotonically increasing timestamp globally synchronized across all data sources, and a delta attribute T. Pulse uses the reference timestamp's monotonicity to bound query state and delta timestamps for simplified query processing. Our models are piecewise functions, that is they are made up of *segments*. Denoting r as a reference timestamp and t^l, t^u as offsets, a segment, $s \in S$, is a time range $[r + t^l, r + t^u)$, for which a particular set of coefficients for a modeled attribute, $\{c_i\}$, are valid (written as $s = ([t^l_i, t^u_i), c_i) = ([t^l, t^u), c)_i$). In the remainder of this work, we drop the reference timestamp r from our time ranges for readability. Thus, we can represent an attribute's model over the lifetime of an application as a sequence of segments: $S = (([t^l, t^u), \{c\})_i, ..., ([t^l, t^u), \{c\})_j)$. We adopt the following update semantics. For two adjacent input segments overlapping temporally, the successor segment acts as an update to the preceding segment for the overlap, that is $\forall i, j : [t^l, t^u)_i \cap [t^l, t^u)_j \neq \emptyset \wedge [t^l, t^u)_i < [t^l, t^u)_j \Rightarrow (([t^l_i, t^l_j), c_i), \ldots, ([t^l, t^u), c)_j)$. This captures the uniqueness properties of an online piecewise function, where pieces appear sequentially.

2.2.2 Key attributes.

Pulse's data streams contain exactly two other types of attributes, keys and unmodeled attributes. Keys are discrete, unique attributes and may be used to represent discrete entities, for example different entities in a data stream of moving object locations. Unmodeled attributes are constant for the duration of a segment, as required by our time-invariant models. We omit details on the operational semantics of the core processing operators with respect to key and unmodeled processing due to space constraints. Our general strategy is to process these using standard techniques alongside the modeled attributes.

The underlying principle of our continuous-time processing mechanism is to take advantage of the temporal continuity provided by the input streams' data models in determining the result of a query. The basic computation element in Pulse is a simultaneous equation system that is capable of performing computation on continuous functions corresponding to operations performed by the relational algebra. In this section we describe how we construct these equation systems from our data models for core operators such as filters, aggregates and joins, and how we are able to compose these equation systems to perform query processing.

2.3 Selective Operators

Selective operators, such as stream filters and joins, produce outputs upon the satisfaction of a predicate comparing input attributes using one of the standard relational operators (i.e., $<, \leq, =, != , \geq, >$). We derive our equation system by transforming predicates in a three step process. Consider the a predicate with a comparison operator R, relating two attribute x, y as xRy. Our transformation is:

	General form
1. Rewrite in difference form	x - y R 0
2. Substitute continuous model	x(t) - y(t) R 0
3. Factorize model coefficients	(x-y)(t) R 0

We provide an example of these steps as applied to a join operator in Figure 2.1. The above equation defines a new function, $(x - y)(t)$, from the difference of polynomial coefficients that may be used to determine predicate satisfaction and consequently the production of results. Note that we are able to simplify the difference form into a single function by treating the terms of our polynomials independently. Depending on the operator R and the degree of the polynomial, there are various efficient methods to approach the above equation. In the case of the equality operator, standard root finding techniques, such as Newton's method or Brent's method [13], solve for points at which $(x - y)(t) = 0$. We may combine root finding with sign tests to yield a set of time ranges during which the predicate holds. We illustrate this geometrically in Figure 2.2.

The above difference equation forms one row of our equation system. By considering more complex conjunctive predicates, we arrive at a set of difference equations of the above form that must all hold simultaneously for our selective operator to produce a result. That is, given the following predicate and models: $x_1 R_1 y_1 \wedge x_2 R_2 y_2 \wedge \ldots \wedge x_p R_p y_p$, where $\forall i . x_i = \sum_{j=0}^{d} c_{x,i}^{j} t^i$, and $\forall i . y_i = \sum_{j=0}^{d} c_{y,i}^{j} t^i$, and $c_{x,i}^{j}$ is the jth coefficient in a segment, we derive

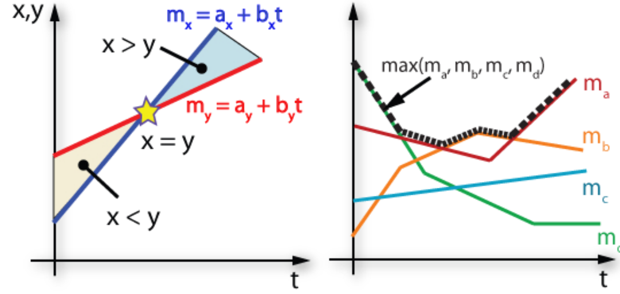

Figure 2.2: A geometric interpretation of the continuous transform, illustrating predicate relationships between models for selective operators, and piecewise composition of individual models representing the continuous internal state of a `max` aggregate.

the following equation system:

$$\begin{bmatrix} c^0_{x,1} - c^0_{y,1} & \dots & c^d_{x,1} - c^d_{y,1} \\ c^0_{x,2} - c^0_{y,2} & \ddots & \vdots \\ \vdots & & \\ c^0_{x,p} - c^0_{y,p} & \dots & c^d_{x,p} - c^d_{y,p} \end{bmatrix} \mathbf{t} \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_p \end{bmatrix} \mathbf{0}$$
$$= \mathbf{Dt}\ \mathbf{R}\ \mathbf{0} \tag{2.1}$$

In the above equation, $\mathbf{t}$ represents a vector comprised of powers of our time variable (i.e., $[t, t^2, t^3, \dots]'$). Thus the above equation system has a single unknown variable, namely a point in time t. We denote the matrix $\mathbf{D}$ our difference equation coefficient matrix. Under certain simplified cases, for example when $\mathbf{R}$ consists solely of equality predicates (as would be the case in a natural or equi-join), we may apply efficient numerical algorithms to solve the above system (such as Gaussian elimination or a singular value decomposition). A general algorithm involves solving each equation independently and determining a common solution based on intersection of time ranges. In the case of general predicates, for example including disjunctions, we apply the structure of the boolean operators to the solution time ranges to determine if the predicate holds. Clearly, Equation 2.1 may not have any solutions indicating that the predicate never holds within the segments' time ranges for the given models. Consequently the operator does not produce any outputs.

Pulse uses update segments to drive the execution of our equation systems. Consider the arrival of a segment, with time range $[t_0, t_1)$. For a filter operator, we instantiate and solve the equation system from the contents of the segment alone, ensuring that the solution for

the variable t is contained within $[t_0, t_1)$ (for both point and range solutions). For a join, we use equi-join semantics along the time dimension, specifically we execute the linear system for each segment $[t_2, t_3)$ held in state that overlaps with $[t_0, t_1)$ (for each attribute used in the predicate). In our solver, we only consider solutions contained in $[t_0, t_1) \cap [t_2, t_3)$.

2.4 Aggregate Operators

Aggregation operators have a widely varying set of properties in terms of their effects on continuous functions. In this section we present a continuous-time processing strategy for commonly found aggregates, namely min, max, sum, and average. At a high-level, we handle min and max aggregates by constructing an equation system to solve when to update our aggregate's internal state, while for sum and average, we define continuous functions for computing the aggregate over windows with arbitrary endpoints (i.e., continuous windows).

2.4.1 Min and Max Window Functions

The case of a single model per stream is trivial for min and max aggregates as it requires computing derivatives on polynomial segments to determine state updates. We focus on the scenario where a data stream consists of multiple models due to the presence of key attributes. The critical modification for these aggregates lies in the right-hand side of the difference equation, where we now compare an input segment to the partial state maintained within the operator from aggregating over previous input segments. We denote this state as $s(t)$, and define it as a sequence of segments: $s(t) = (([t^l, t^u), c)_1, ([t^l, t^u), c)_2 ..., ([t^l, t^u), c)_n)$, where each $([t^l, t^u), c)_i$ is a model segment defined over a time range with coefficients c_i. For example with a min (or max) function, the partially aggregated model $s(t)$ forms a lower (or upper) envelope of the model functions as illustrated in Figure 2.2. Thus we may write our substituted difference form as $x(t) - s(t)\ R\ 0$. This difference equation captures whether the input segment updates the aggregated model within the segment's lifespan. We use this difference equation to build an equation system in the same manner as for selective operators.

2.4.2 Sum and Average Window Functions

The sum aggregate has a well-defined continuous form, namely the integration operator. However, we must explicitly handle the aggregate's windowing behavior especially since sum and average aggregate along the temporal dimension. To this end, we define *window*

functions, which are functions parameterized over a window's closing timestamp to return the value produced by that window. At a high level, window functions help to preserve continuity downstream from the aggregate. We now describe how we compute a window function for sums.

We assume a window of size w and endpoint t, and consider two possible relationships between this window and the input segments. The lifespan of a segment $[t^l, t^u)$ may either match (or be larger than) the window size w, or be smaller than w. In the first case, we may compute our window results from a single segment. Specifically, we claim that a segment covering $[t^l, t^u)$ may produce results for a segment spanning $[t^l + w, t^u)$, since windows closed in this range are entirely covered by the segment. We define the window function for this scenario as:

$$wf_{sum}(t) = \int_{t-w}^{t} \sum_{i=0}^{d} c_i t^i dt = \sum_{i=i}^{d+1} \frac{c_{i-1}}{i} t^i \tag{2.2}$$

which is parameterized by the closing timestamp t of the window. In the scenario where a window spans multiple segments, we divide the window computation into three sub-cases: i) segments $[t_1^l, t_1^u)$ entirely covered by the window, ii) segments $[t_2^l, t_2^u)$ overlapping with head of the window t, and iii) segments $[t_3^l, t_3^u)$ overlapping with the tail of the window $t - w$. In the first sub-case, we compute the integral value for the segment's lifespan and denote this the constant C. In the second sub-case, we use the window function defined in Equation 2.2, and refer to this as the *head integral*. For the third sub-case, we apply an integral spanning the common time range of the segment $[t_3^l, t_3^u)$, and window: $\int_{t-w}^{t_3^l} \sum_{i=0}^{d} c_i t^i dt$. We refer to this integral as the *tail integral*. Note that for a given segment t_3^l is known and fixed. However we are still left with the term $t - w$ in our formula, but can leverage the window specification which provides a fixed value of w to express the result of the integral, by expanding terms of the form $(t - w)^i$ for $i > 0$ by the binomial theorem. This yields the following window function for windows spanning multiple segments: $wf_{sum}(t) = \int_{t-w}^{t_3^l} \sum_{i=0}^{d} c_i t^i dt + C + \int_{t_2^l}^{t} \sum_{i=0}^{d} c_i t^i dt$

For every input segment $[t_i^l, t_i^u)$ at the aggregate, we compute and cache the segment integral C, in addition to a function for the tail integral. This metadata is to be used by windows functions produced by future updates arriving at the aggregate. Finally we produce a window function for the input segment itself that spans all windows contained in its time range by fetching segment integrals and tail integrals for the set of windows $[t^l - w, t^u - w)$. While the above discussion concerned a sum function, these results may easily be applied to compute window functions for averages as $wf_{avg} = \frac{wf_{sum}}{w}$.

Operator	Inputs	State	Implementation	Outputs
Filter	x_i	–	$\mathbf{D} = [x_i - c_i]$; **solve DtR0**	$\{(t, x_i)\vert\mathbf{DtR0}\}$
Join	x_i on left input y_i on right input	order-based segment buffers, $S_x = \{([t^l, t^u), s_x)\vert t^l > t_y\}$ $S_y = \{([t^l, t^u), s_y)\vert t^l > t_x\}$	align x_i, y_i w.r.t t; $\mathbf{D} = [x_i - y_i]$; **solve DtR0**	$\{(t, x_i, y_i)\vert\mathbf{DtR0}\}$
Aggregate **min, max**	x_i	state model, $S = \{([t^l, t^u), s)\vert t^l > t_x - w\}$	align x_i, s_i w.r.t t $\mathbf{D} = [x_i - s_i]$; **solve DtR0**	$\{(t, s_i)\vert\mathbf{DtR0}\}$
Aggregate **sum, avg**	x_i	segment final $C = \int_{t^l}^{t^u} \sum_{i=0}^{d} x_i t^i$, $wf_{tail} = \int_{t-w}^{t} \sum_{i=0}^{d} x_i t^i dt$	$wf_{sum} =$ $wf_{tail} + C + wf_{head}$	$([t_l, t_u], wf_{sum})$
Aggregate group-by, function f	x_i per group	state for f per group	hash-based group-by, impl for f per group	outputs for f, per group

Table 2.1: Operator transformation summary.

Symbol definitions: x_i, y_i are polynomial coefficients for attributes x, y; $t = [t^l, t^u)$ is the valid time range for a segment; t_x, t_y denote the reference timestamps for the latest valid times for attributes x, y; (t, x) is the segment itself as a pair of valid times and coefficients; $(t, s_x)_i$ is a segment of attribute x that is kept in an operator's state; s_i are the coefficients of these state segments.

Transformation Limitations.

Frequency-based aggregates are those that fundamentally depend on the number of tuples in the input stream. Examples include count, frequency moments, histograms etc. Certain aggregation functions can be viewed as mixed aggregates if they depend on both the content and the frequency, for example a sum aggregate may have larger values for high rate data streams (assuming positive numbers). Presently, our framework does not handle frequency oriented aggregates, and can only handle mixed aggregates when their computation involves all tuples in the relation (and thus all points on the continuous function) like sum and average. Figure 2.1 summarizes Pulse's selective and aggregate operator transforms.

2.5 Continuous Segment Processing

Pulse performs operator-by-operator transformation of regular stream query instantiating an internal query plan comprised of simultaneous equation systems. Each equation system is closed, that is it consumes segments and produces segments, enabling Pulse's query processing to use segments as a first-class datatype. However, depending on the operator's characteristics, an equation system may produce an output segment whose temporal validity is a single point. This occurs primarily with selective operators involving at least one equality comparison. The reduction of a model to a single point limits the flow of models

through our representation, since the remaining downstream operators can only perform discrete processing on this intermediate result.

Once the processed segment reaches an output stream, we produce output tuples via a sampling process. For selective operators, this requires a user-defined sampling rate. We note that for an aggregate operator producing query results, there is no explicit need for a application-specified output rate. This may be inferred from the aggregate's window specification, and in particular the slide parameter which indicates the periodicity with which a window closes, and thus the aggregate's output rate.

2.6 Enforcing Precision Bounds

To handle differences between our continuous-time models and the input tuples, Pulse supports the specification of accuracy bounds to provide users with a quantitative notion of the error present in any query result. We consider an absolute error metric and *validate* that continuous-time query results lie within a given range of results produced by a standard stream query. One validation mechanism could process input tuples with both continuous-time and regular stream queries and check the results. However, this naive approach performs duplicate computation, offsetting any benefits from processing inputs in a continuous form.

Our validation mechanism checks accuracy at the query's inputs and completely eliminates the need for executing the discrete-time query. We name this technique query inversion since it involves translating a range of output values into a range of input values by approximately inverting the computation performed by each query operator. Some operators that are many-to-one mappings, such as joins and aggregates have no unique inverse when applied to outputs alone. However we may invert these operators given both the outputs and the inputs that caused them, and rely on continuity properties of these inputs to invert the output range. Query inversion maintains these inputs as query lineage, compactly as model segments.

We use accuracy validation to drive Pulse's online predictive processing. In this scenario, Pulse only processes queries following the detection of an error. We note that accuracies may only be attributed to query results if the query actually produces a result. Given the existence of selective operators, an input tuple may yield a null result, leaving our accuracy validation in an undefined state. To account for this case, we introduce *slack* as a continuous measure of the query's proximity to producing a result. We define slack as:

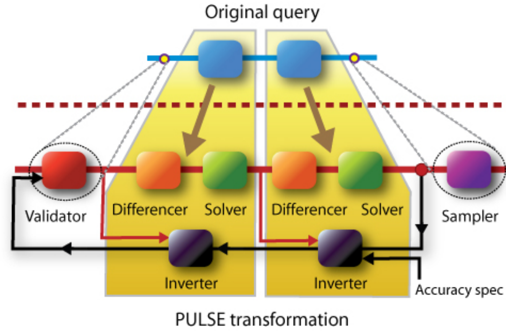

Figure 2.3: High level overview of Pulse's internal dataflow. Segments are either given as inputs to the system or determined internally, and processed as first-class elements.

$$\begin{aligned}&\mathbf{slack} = \min_t \|\mathbf{Dt}\|_\infty \\ &\text{s.t} \;\; t \in \bigcap [t^l, t^u)_i \qquad \forall i.[t^l, t^u)_{update} \cap [t^l, t^u)_i \neq \emptyset\end{aligned}$$

Above, we state that we only compute slack within valid time ranges common with the update segment causing the null (for stateful operators). Using the maximum norm ensures that we do not miss any mispredicted tuples that could actually produce results. Following a null any intermediate operator, Pulse performs slack validation, ignoring inputs until they exceed the slack range. Thus Pulse alternates between performing accuracy and slack validation based on whether previous inputs caused query results. Figure 2.3 provides a high-level illustration of Pulse's internal dataflow, including the inverter component that maintains lineage from each operation and participates in both accuracy and slack bound inversion.

2.6.1 Query Output Semantics

We briefly discuss the semantics of the outputs produced by continuous-time data processing. While a complete discussion of the topic lies outside the scope of this paper, we make several observations in the context of comparing and understanding the operational semantics of a continuous-time processor in comparison to a discrete-time processor. Clearly, the two modes of processing are not necessarily operationally equivalent on a given set of inputs. They may differ in the following ways.

Observation 1: Pulse may produce false positives with respect to tuple-based processing. If Pulse's query results are not discretized in the same manner as the input streams, Pulse may produce results that are not present under regular processing of the input tuples. For

example, consider an equi-join that is processed in continuous form by finding the intersection point of two models. Unless we witness an input tuple at the point of the intersection, Pulse will yield an output while the standard stream processor may not, resulting in a superset output semantics.

Observation 2: Pulse may produce false negatives with respect to tuple-based processing. False negatives occur when the discrete-time query produces results but Pulsedoes not, yielding a subset output semantics. This may occur as a result of precision bounds which allow any tuple lying near its modeled value to be dropped. Any outputs that may otherwise have been caused by the valid tuple are not necessary, and therefore omitted. Again the difference in result sets arises from a lack of characterizing discretization properties.

2.6.2 Query Inversion

We describe query inversion as a two-stage problem, first as a local problem for a single operator, and then for the whole query, leveraging the solution to the first problem operator to produce an inversion data flow.

Bound inversion problem: *given an output value and a range at an operator, what range of input values produces these output values?* This problem may have many satisfying input ranges when aggregates and joins are present in the query. For example, consider a sum aggregate, and the range $[5, 10]$ as the output values. There are infinitely many multisets of values that sum to 5, (e.g. the sets $\{4, 1\}$ and $\{-1, -2, 8\}$). The fundamental problem here is that we need to identify a unique inverse corresponding to the actual computation that occurs (motivated by continuity for future bound validation). We use the following two properties to perform this restriction:

Property 1: continuous-time operators produce temporal subranges as results. This ensures that every output segment is caused by a unique set of input segments.

Property 2: modeled attributes are functional dependents of keys throughout the dataflow. Each operator in our transformation preserves a functional dependency between keys and segments by passing along the key values that uniquely identify a segment.

These properties ensure we are able to identify the set of input segments for operations involving multiple segments (joins and aggregates) through segments' time ranges and key values. Providing we maintain the input keys and segments used to produce an intermediate operator's results (i.e., the lineage of a segment), we are able to identify the cause of each output segment, making query inversion an issue of accessing lineage. We remark that the cost of maintaining lineage is less prohibitive than with regular tuples due a segment's

compactness (a full analysis of the lineage requirements lies outside this paper's scope).

Given both the input models and the output models, solving the bound inversion problem then becomes an issue of how to apportion the bound amongst the set of input models. We describe *split heuristics* in the next section to tackle this problem. Our run-time solution to the bound inversion problem is dynamic and expressive, providing the ability to adapt to changing data distributions. By considering both the input and output segments during inversion, we are able to support different types of bounds including both absolute and relative offset bounds.

Query inversion problem: *given a range of values on each attribute at a query's output, what ranges of query input values produce these outputs?* Query inversion determines the appropriate context for performing bound inversion, given the query's structure. In particular we focus on addressing attribute aliasing, and attribute dependencies caused by predicates, as shown in the following example. Consider the query (omitting windows and precision bounds for readability):

```
select a, b as x, d from R join S
    where R.a = S.a and R.a < S.d
```

Here, a new attribute x is declared in the results' schema, and is an alias of the attribute b. We must track this data dependency to support query inversion on error bounds specified on attribute x, and refer to this metadata as bound *translations*. The second type of dependency concerns query where-clauses. In this example, the attribute $S.d$ is not part of the query's results, but constrains the results via its presence in a predicate. We track these dependencies and refer to them as *inferences*. During the inversion process, we apportion bounds to attributes such as $S.d$, inferring the values they may take. Pulse computes the translation and inference metadata as part of the planning phase, and passes this metadata to inverter operators which actually perform the computation for query inversion.

2.6.3 Split Functions

In this section, we present two heuristics for allocating value ranges of an operator's output attributes to its input attributes for both accuracy and slack bounds. Pulse supports the specification of user-defined split heuristics by exposing the a function interface for the user to implement, for an absolute error metric. We describe our heuristics in terms of the function signature (simplified to a single modeled attribute a for ease of understanding):

$$\{(ik_p, [i_a^l, i_a^u]), ..., (ik_q, [i_a^l, i_a^u])\} =$$
$$split(ok, oc, [o^l, o^u], \{(ik_p, ic_a) \ldots, (ik_q, ic_a)\})$$

where $(ik_p, [i_a^l, i_a^u])$ are the bounds allocated to input attribute a for key p. Also, ok, oc denote the keys and coefficients of the output segment, $[o^l, o^u]$ the output bound, and finally $\{(ik_p, ic_a) \ldots (ik_q, ic_a)\}$ the keys and coefficients of the input segments producing the output. Note that the result of our split function includes both the set of input keys that we split over, in addition to the bounds. Thus bounds are only allocated to the keys that actually cause the output.

Equi-split: this heuristic assigns the output error bound uniformly across all input attributes. Specifically, it implements the following split heuristic:

$$(ik_p, [i_a^l,\, i_a^u]) = [\tfrac{o^l}{n}, \tfrac{o^u}{n}]$$
$$\text{where} \quad a \in D(o), n = |\{ik_p \ldots ik_q\}| * |D(o)|,$$
$$D(o) = translations(o) \cup inferences(o)$$

The above equation specifies the uniform allocation of a bound to each key and attribute dependency.

Gradient split: this heuristic attempts to capture the contribution of each particular input model to the output result. Formally, the heuristic computes:

$$(ik_p, [i_a^l, i_a^u]) = \tfrac{d(ic_a)}{dt} * [\tfrac{o^l}{\sum_{m \in I} ic_m}, \tfrac{o^u}{\sum_{m \in I} ic_m}]$$
$$\text{where} \quad a \in D(o)$$
$$D(o) = translations(o) \cup inferences(o)$$
$$I = \{(ik_p, ic_a), \ldots, (ik_q, ic_a)\}$$

The above equation specifies that each bound allocated is the product of the gradient of a single segment with respect to the global segment of all input keys contributing to the result.

Both of the above schemes are conservative in the sense that they preserve two-sided error bounds, and ensure that the error ranges allocated on input attributes do not exceed the error range of the output attribute. A more aggressive allocation scheme may reduce two-sided error bounds to a one-sided error, for example in the case of inequality predicates, thereby improving the longevity of the bounds. In general, the efficiency of validating query processing is fundamentally an optimization problem and our current solution lays the framework for further investigation.

2.7 Experimental Evaluation

We implemented Pulse as a component of the Borealis [1] stream processing engine. This implementation provides full support of the basic stream processing operators including filters, maps, joins and aggregates and extends our stream processor's query language with accuracy and sampling specifications. Pulse is implemented in 27,000 lines of C++ code and adds general functionality for rule-based query transformations to Borealis, in addition to specialized transformations to our equation systems. We note that Pulse requires a small footprint of 40 lines in the core stream processor code base indicating ease of use other stream processors. In these experiments, Pulse executes on an AMD Athlon 3000+, with 2GB RAM, running Linux 2.6.17. We configured our stream processor to use 1.5GB RAM as the page pool for allocating tuples.

Our experiments use both a real-world dataset and a synthetic workload generator. The synthetic workload generator simulates a moving object, exposing controls to vary stream rates, attribute values' rates of change, and parameters relating to model fitting. Our real-world datasets are traces of stock trade prices from the New York Stock Exchange (NYSE) [67], and the latitudes and longitudes of naval vessels captured by the Coast Guard through the Automatic Identification System (AIS) [108].

2.7.1 Operator Benchmarks

Our first results are a set of microbenchmarks for individual filters, joins and aggregates. We investigate the processing throughput for fixed size workloads from our moving object workload generator, under a varying model expressiveness measured as the number of tuples that fit a single model segment. The workload generator provides two-dimensional position tuples with a schema: x, y, v_x, v_y denoting x- and y-coordinates in addition to x- and y-velocity components.

Filter. Figure 2.4 demonstrates that the continuous-time implementation of a filter requires a strong fit in terms of the number of tuples per segment, between the model and the input stream. The continuous-time operator becomes viable at approximately 1050 data points per segment. This matches our intuition that the iterations performed by the linear system during solving dwarfs that performed per tuple by an extremely simple filter operation.

Aggregate. Figure 2.5 compares the continuous-time aggregate's throughput for the min function under varying model fit settings. We also illustrate the cost of tuple-based processing at three window sizes for comparison. The window size indicates the number of

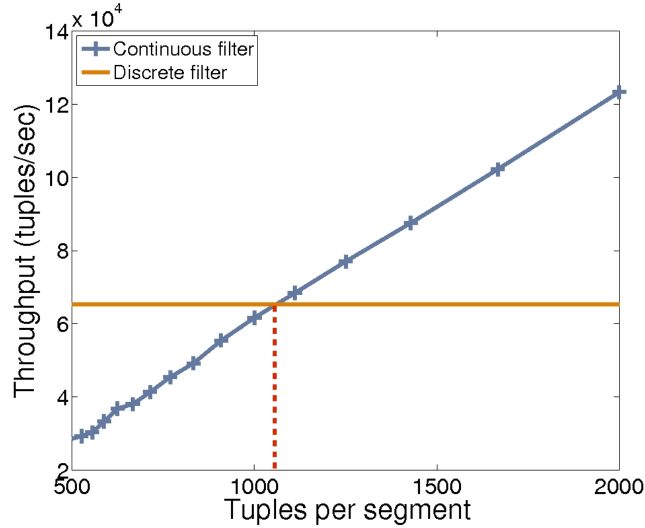

Figure 2.4: Microbenchmark for a filter operator with a 1% error threshold.

open windows at any point in time, and thus the number of state increments applied to each tuple. This benchmark shows the continuous-time aggregate provides higher throughput at approximately 120-180 tuples per segment for different windows. Thus we can see that the model may be far less expressive (by a factor of 5x) for our processing strategy to be effective. This improvement primarily arises due to the increased complexity of operations per tuple performed by an aggregate, in comparison to linear system solving. Figure 2.6 illustrates the operator's processing costs as window sizes vary from 10 to 100 seconds. Here the cost of a tuple-based aggregate is clearly linear in terms of the window size, while the cost of our segment-based processing remains low due to the fact we are only validating the majority of tuples, and not solving the linear system for each tuple. We demonstrate that Pulse outperforms tuple processing at window sizes beyond 30 seconds, and is able to achieve a 40% cost compared to regular processing at a 100 second window.

Join. Figure 2.7 displays the throughput achieved by a continuous-time join compared to a nested loops sliding window join as the number of tuples per segment is varied. The join predicate compares the x and y positions of objects in our synthetic workload. Figure 2.7 shows that our join implementation outperforms the discrete join at 1.45 tuples per segment for a window size of 0.1s. This occurs because a nested loops join has quadratic complexity in the number of comparisons it performs, as opposed to the complexity of a validation operation which is linear in the number of model coefficients. Figure 2.8 illustrates the difference in processing cost under varying stream rates, and clearly shows Pulse's

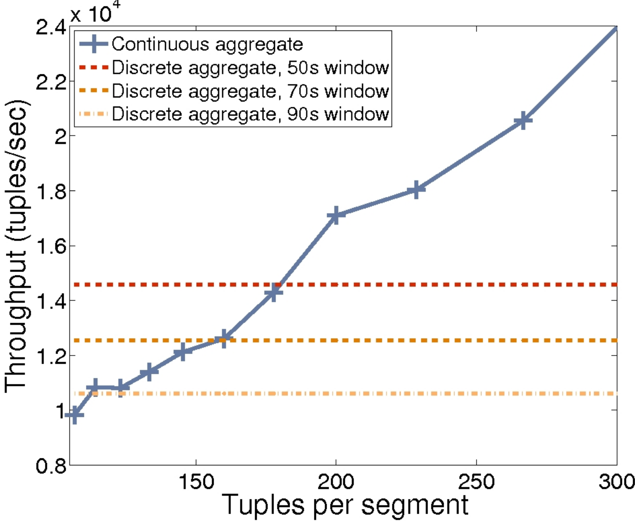

Figure 2.5: Microbenchmark an aggregate operator with a 1% error threshold.

significantly lower overhead. The processing cost of our mechanism remains low while the tuple-based cost increases quadratically (despite the linear appearance, we verified this in preliminary experiments while extending to higher stream rates). We plan on investigating this result with other join implementations, such as a hash join or indexed join, but believe the result will still hold due to the low overhead of validation compared to the join predicate evaluation.

Historical processing. Figure 2.9 presents throughput and processing cost results from the historical application scenario. In these results, we present the cost of performing model fitting, via an online segmentation-based algorithm [53] to find a piecewise linear model to the input data, in addition to processing the resulting segments. We consider a min aggregate, with a 60 second window, and a 2 second slide. Tuple processing reaches a maximum throughput of 15,000 tuples per second before tailing off due to congestion in the system as processing reaches capacity. In contrast, segment processing continues to scale beyond this point, demonstrating that the data modeling operation does not act as a bottleneck with this workload. The nested plot of modeling throughput, which executes our model fitting operator alone, illustrates that this instead happens at a higher throughput of approximately 40,000 tuples. This result indicates that data fitting is indeed a viable option in certain cases, and that simplistic modeling techniques such as piecewise linear models are indeed able to support high-throughput stream processing.

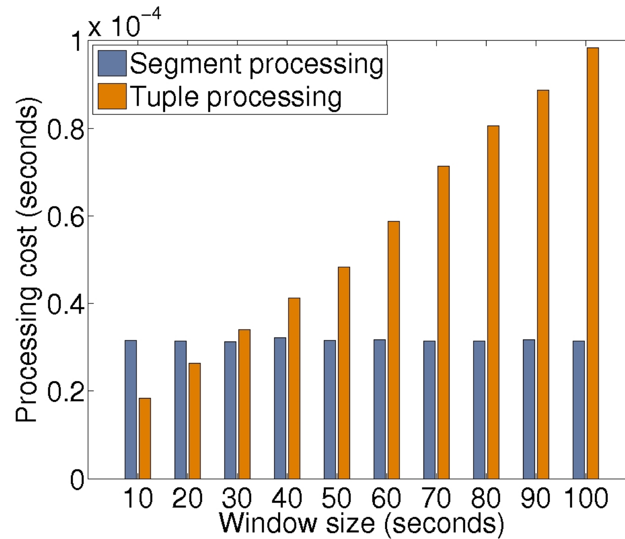

Figure 2.6: Continuous-time and discrete processing overhead comparison for an aggregate operator.

2.7.2 Query Processing Performance

We extracted the NYSE dataset of stock trade prices from the TAQ3 data release for January 2006, creating workloads of various sizes for replay from disk into Pulse. The schema of this dataset includes fields for *time, stock symbol, trade price, trade quantity.* In our experiments on this dataset, we stream the price feed through a continuously executing moving average convergence/divergence (MACD) query, a common query in financial trading applications. The MACD query is as follows (in StreamSQL syntax):

```
select symbol, S.ap - L.ap as diff from
  (select symbol, avg(price) as ap from
    stream S[size 10 advance 2]) as S
join
  (select symbol, avg(price) as ap from
    stream S[size 60 advance 2]) as L
on (S.Symbol = L.Symbol)
where S.ap > L.ap
```

This query uses two aggregate operations, one with a short window to compute a short-term average, and the other with a long window to compute a long-term average, before applying a join operation to check for the presence of a larger short-term average.

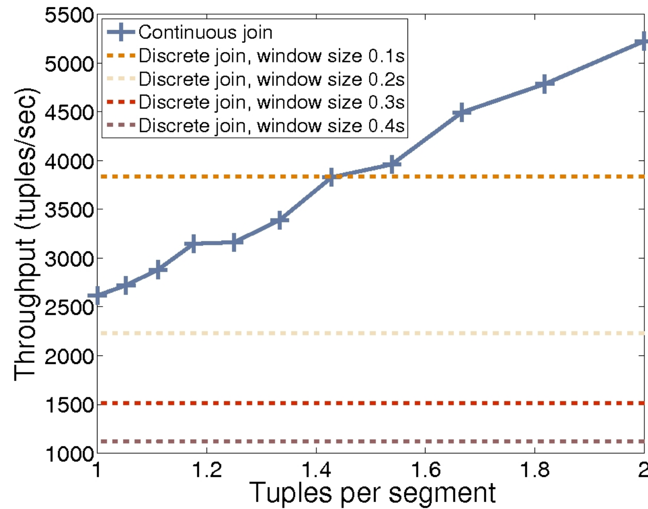

Figure 2.7: Microbenchmark for a join operator with a 1% error threshold.

The AIS dataset contains geographic locations and bearings of naval vessels around the coasts of the lower 48 states over a 6-day period in March 2006, totaling to approximately 6GB of data. We extracted a subset of the data for replay, with the following schema: *vessel id, time, longitude, longitudinal velocity, latitude, latitudinal velocity*. We then use the following query to determine if two vessels were following each other:

```
select Candidates.id1, Candidates.id2, avg(dist)
  (select S1.id as id1, S2.id as id2,
     sqrt(pow(S1.x-S2.x,2) + pow(S1.y-S2.y,2)) as dist
   from S[size 10 advance 1] as S1
     join S as S2[size 10 advance 1]
     on (S1.id <> S2.id))[size 600 advance 10]
  as Candidates
group by id1, id2 having avg(dist) < 1000
```

The above query continuously tracks the proximity of two vessels with a join operation and computes the average separation over a long window. We then apply a filter to detect when the long-term separation falls below a threshold.

2.7.3 NYSE and AIS Processing Evaluation

In this section we compare the throughput of the NYSE and AIS datasets and queries as they are replayed from file.

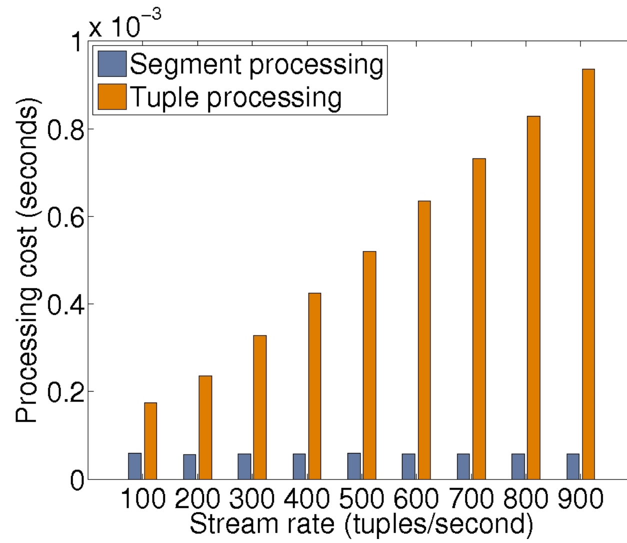

Figure 2.8: Continuous-time and discrete processing overhead comparison for a join operator.

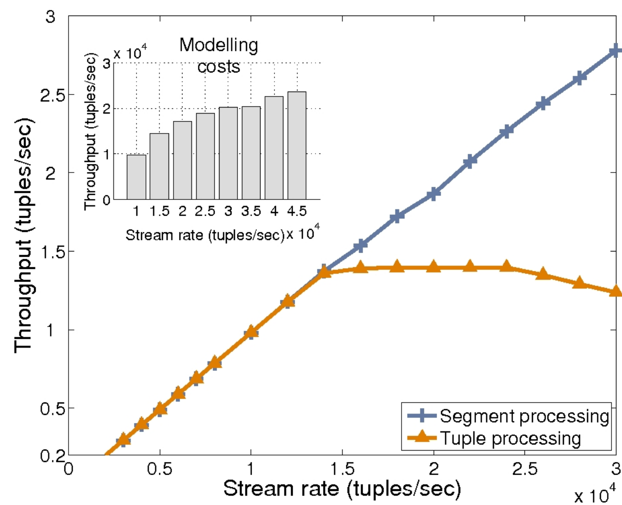

Figure 2.9: Historical aggregate processing throughput comparison with a 1% error threshold.

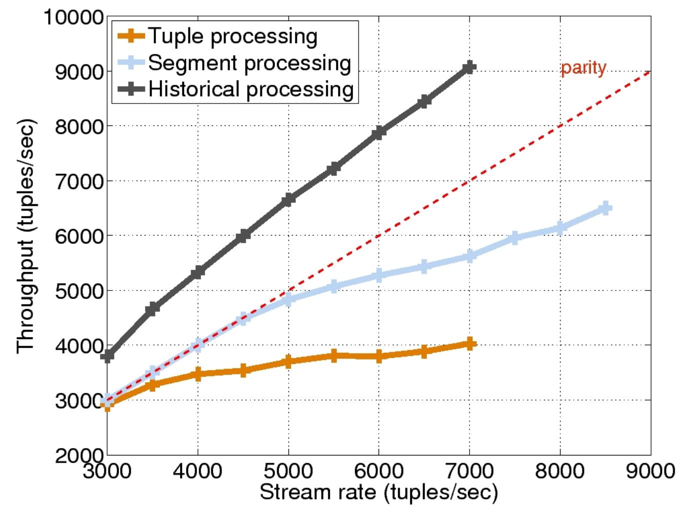

Figure 2.10: Continuous-time processing of the NYSE dataset, with a 1% error threshold.

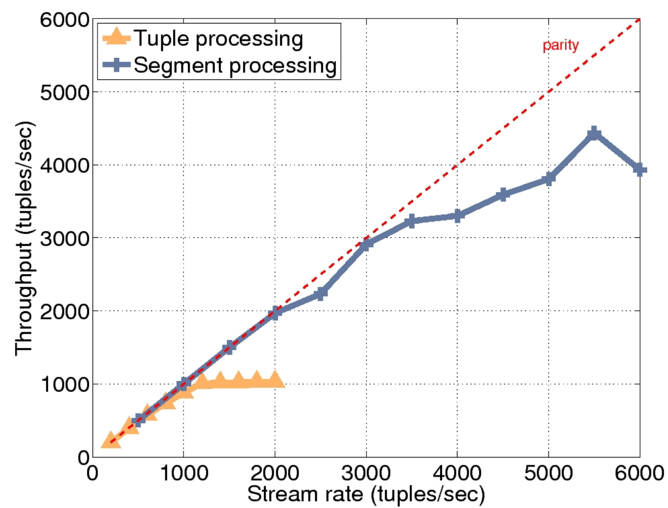

Figure 2.11: Continuous-time processing of the AIS dataset, with a 0.05% error

Figure 2.10 compares the throughput Pulse achieves while processing the NYSE dataset in comparison to standard stream processing. In this experiment, we set error thresholds to 1% of the trade's value. We see that the tuple-based MACD query tails off at a throughput of approximately 4000 tuples per second. We ran no further experiments beyond this point as the system is no longer stable with our dataset exhausting the system's memory as queues grow. In contrast the continuous-time processor is able to scale to approximately 6500 tuples per second, and similarly begins to lead to instabilities beyond this point. As a further comparison, we plot the historical processing performance that represents the throughput of processing segments alone (without modeling). This reflects performance achieved following an offline segmentation of the dataset. Historical processing scales well in this range of stream rates due to the lack of any validation overhead. Also note that we achieve greater throughput than parity, due to lower end-to-end execution times for our fixed workload, through the early production of results from sampling the linear models.

Figure 2.11 compares throughput in the AIS dataset, for an error threshold of 0.05%. This plot illustrates that the original stream query tails off after a stream rate of 1100 tuples per second, achieving a maximum throughput of approximately 1000 tuples per second. In contrast, Pulse is able to achieve a factor of approximately 4x greater throughput with a maximum of 4400 tuples per second. We note the lower stream rate in the AIS scenario in comparison to the NYSE scenario, due to the presence of a join operator as the initial operator in the query (the MACD query has aggregates as initial operators). The segment processing technique reaches its maximum throughput without any tail off since it hits a hard limit by exhausting the memory available to our stream processor while enqueuing tuples into the system.

2.7.4 Precision Bound Sensitivity Evaluation

Figure 2.12 displays the end-to-end processing latency achieved by Pulse for the MACD query on the NYSE dataset under varying relative precision bounds. The inset figure displays the number of precision bound violations that occurred during execution on a logarithmic scale. This figure demonstrates that Pulse is able to sustain low processing latencies under tight precision requirements, up to a threshold of 0.3% relative error for this dataset. The inset graph shows that as the precision bound decreases, there are exponentially more precision violations. Beyond a 0.3% precision, the processing latency increases exponentially with lower errors due to the queuing that occurs upon reaching processing capacity.

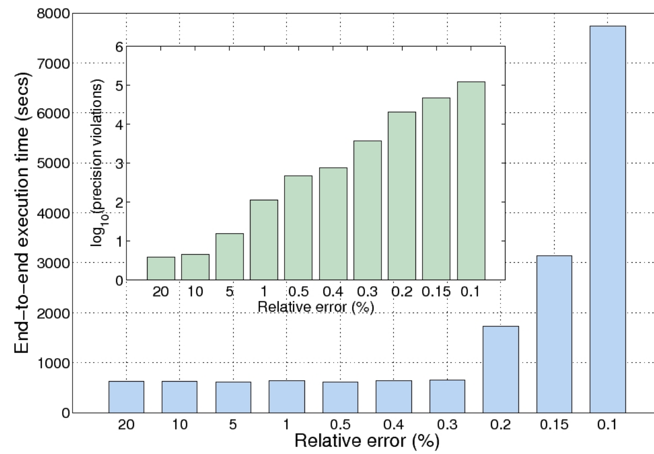

Figure 2.12: Continuous-time processing of the NYSE data at 3000 tuples/second.

In summary, our experimental results show that continuous-time processing with segments can indeed provide significant advantages over standard stream processing providing the application is able to tolerate a low degree of error in the results. In certain scenarios Pulse is capable of providing up to 50% throughput gain in an actual stream processing engine prototype, emphasizing the practicality of the our proposed mechanisms.

Chapter 3

Selectivity Estimation for Continuous Function Processing

In the previous chapter, we presented a system overview and the core query executor of the Pulse stream processor. We demonstrated how Pulse is able to leverage both the compact data representation provided by polynomial segments and user-defined precision bounds to significantly improve query processing performance in comparison to standard tuple-based processing techniques found in today's stream processors. We now address the question of how to improve Pulse's scalability beyond the basic segment-oriented query executor, and describe two additional components of our system. These are a query optimizer, designed specifically to provide support for multi-query optimization in our context, and a selectivity estimator that supports this query optimizer by capturing segment-based processing selectivities through the use of various mathematical properties of our polynomial segments. In this chapter, we start by introducing the problem of cost-based query optimization and present an overview on the type of cost model and optimization technique we consider, before focusing on the first stage of this problem – deriving a selectivity estimator customized to our equation systems.

3.1 Cost-Based Query Optimization

Both traditional database architectures and stream processing engines heavily utilize cost-based query optimization techniques that at their core rely upon a cost model to capture the desired optimization metric. In our work we consider a processing cost metric so that we may reduce the processing overhead of queries by determining advantageous query execution

plans. The processing cost of a query is typically defined based on operators' execution costs and the cardinality of intermediate results, or rate of dataflow between these operators. This latter term relies on the concept of selectivities, defined for an individual operator as the ratio of the number of outputs to the number of inputs. We now briefly describe some of the salient features of the cost model used as our objective function.

3.1.1 Risk-Based Cost Models

Pulse treats polynomial segments as first-class entities during query processing, where each operator consumes and produces segments. This immediately leads to a different perspective on the definition of operator selectivities. While we can still define selectivities abstractly as the ratio of outputs to inputs, our selectivities are defined in terms of segments and represents the segmentation resulting from both equation system solving and transformation operations. Loosely speaking, an input segment gets further segmented downstream, but may sometimes remain intact, for example when the entire segment satisfies an equation. We refer to this latter case as *subsumption*.

One immediate consequence of segment-selectivities is that we must revisit standard selectivity estimation techniques, re-evaluating how we can compute these selectivities from data distributions. We present an overview of this issue below. Selectivity estimation techniques by their nature are inaccurate for a variety of reasons, including their inability to handle correlations and data dependencies present in the input dataset as a result of adopting attribute independence assumptions. Due to these erroneous estimations, our cost model adopts the notion that executing a plan is *risky*, in that the plan may impose a different overhead than that determined by the optimizer. Pulse's optimizer searches over potential plans factoring in both costs and risks associated with query processing. Here, we briefly discuss the relationship between risk and cost.

We can immediately make some intuitive qualitative comments on our objective function. Clearly we would like to evaluate plans that have both low cost and low risk, and during our search for plans we would generally like to prune away plans with higher cost and risk than the best known plan. Figure 3.1i. illustrates these two categories on a risk-cost space in comparison to the best known plan as the low risk and high risk plans respectively. The remaining categories are those plans that offer a tradeoff between cost and risk over the best known plan.

Our objective function uses a weighted linear combination to represent the tradeoff between two plans' costs and risks, and defines a system parameter allowing a database

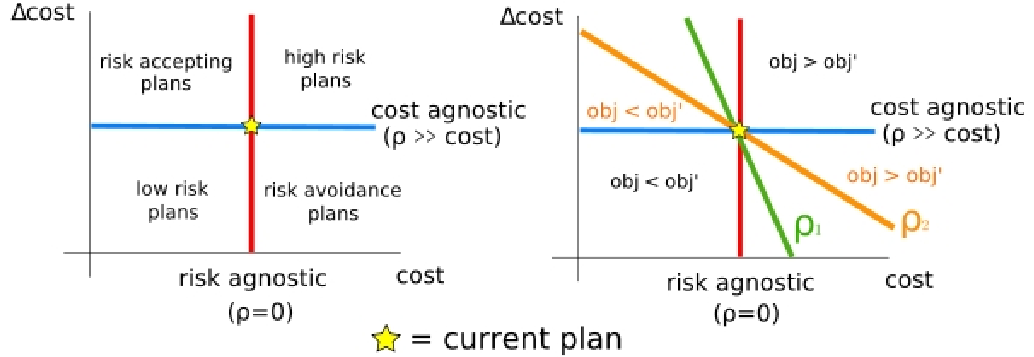

Figure 3.1: Qualitative comparisons of query plans on a risk-cost space.

administrator to customize the objective. This is represented algebraically as: $objective = (1-\rho)cost + \rho risk$, where ρ is the risk aversion factor. This risk aversion factor helps us to define equivalent plans as those with equivalent objective values. The parameter lies in the range $[0, 1]$, and indicates a risk-agnostic objective when $\rho \to 0$, or a cost-agnostic object when $\rho \to 1$. Figure 3.1ii. illustrates the effect of varying this parameter in terms of the fraction of plans explored from the risk accepting and the risk avoidance categories.

3.1.2 Multi-Query Optimizer Overview

Our query optimizer focuses on the challenge of scaling the number of continuous-time queries by detecting and sharing any work common across multiple queries. We present the design of an adaptive optimization algorithm designed to apply to standing queries, modifying these queries using localized heuristics that reduce processing overheads over a sequence of multi-query transformations. Our algorithm executes periodically, collecting statistics during this optimization period before making its decisions based on these statistics at the end of the period. The algorithm makes it decisions by evaluating a set of conditions that are heuristics derived from simplifications of our risk-based cost model. Each condition is associated with a multi-query transformation that is applied upon triggering the condition's threshold.

These conditions rely heavily on operator selectivities in their definitions, leading to the challenge of how to obtain these selectivities. One solution is an instrumentation-based approach where we collect selectivities from running queries. However this does not help the optimizer determine the costs of plans it would potentially like to run but do not currently exist in the system. We describe a selectivity estimator to aid in costing non-existing plans.

Our selectivity estimator requires input stream distributions for attributes used in queries as part of its computation, and in turn derives distributions for intermediate query results. The basic technique we employ for selectivity estimation is to represent our polynomial segments in the dual-space (a high-dimensional space representing polynomial coefficients), and defines methods in the dual-space to compare polynomials and determine when they intersect each other or exhibit a subsumption relationship. We extend these comparison operators to apply to histogram bins, allowing us to derive intersection and subsumption frequencies given the input polynomial distributions.

In the remainder of this chapter, we describe our selectivity estimation technique, starting with our representation of a set of segments collected over an optimization period. Our representation utilizes histograms and we next discuss how we can derive histograms based on the computation performed by relational operators, enabling us to capture the distributions of any intermediate results in our query plans. We can use these intermediate result distributions to compute cardinality and thus selectivity estimates.

The next chapter focuses on our multi-query optimization techniques. For this, we start by describing the three adaptive optimization techniques supported, two modes of sharing operators common across multiple queries, as well as an operator reordering technique. We then present the full details of our processing cost model that factors in this set of techniques, before describing the various adaptive conditions we use, as well as the query modifications they entail.

3.2 Segmentation-Oriented Selectivity Estimation

There are many methods for estimating operators' selectivities based on the characteristics of the input data distributions and deriving the effects of operators' functionalities upon these distributions. Traditional methods include histograms, where a histogram of input data is maintained over the base relations, and subsequently query operators are applied to bin boundaries to model operators' output distributions. Histogram-based estimation techniques typically assume a uniform density within the bin, and consequently leverage this to approximate query results when applying operators, to yield a selectivity estimate.

Histograms are intended to capture longer-term data distributions, and while in a stream environment the data distribution might change over time, the basic assumption is that the data distribution applies for long enough for optimization to be worthwhile. In this section we focus on selectivity estimation for the predictive processing applications. Selectivity

estimation in our context boils down to determining intersection or subsumption of polynomials with respect to the query, in our difference equations. Thus we need to define a method for approximating the number of intersections or subsumptions, to determine the resulting downstream segmentation (i.e. selectivity) of an input segment at an operator. While determining intersections of polynomials can be done by testing for roots of their difference, we need a mechanism that lets us approximate intersection counts between sets of polynomials, for example for join operators.

The first part of this chapter describes our approach for computing this approximation of the number of segments at any point within the query, and we start by describing our method for representing polynomials which allows us to compare polynomials, and more importantly allows us to compare the effects of applying relational operators on these polynomials.

3.2.1 Dual Space Representation of Polynomials

We consider a parameter space for our representation of large sets of polynomials, where each polynomial is a point whose coordinates are given by the values of its coefficients. Thus our parameter space has dimensions for our polynomials' variables and degrees, defining a high-dimensional space representing the universe of coefficient values. For example, given the set of polynomials $\{x + y, x + x^2 + x^3\}$, where x and y are variables, our parameter space has dimensions c_0, x, y, x^2, x^3, and in this case, two coordinates $c_1 = (0, 1, 1, 0, 0)$ and $c_2 = (0, 1, 0, 1, 1)$ representing the two polynomials respectively. The dimension c_0 represents a constant term that can be added to any polynomial, for example the term 10 in $x+x^2+x^3+10$. As shown, the coordinates represent the coefficients of these polynomials in terms of the parameter space's dimensions. In the moving-object database literature this parameter space is commonly referred to as the *dual-space* representation of polynomials [88]. Note that constants are also points in the dual space, where only the first (zeroth) coefficient has a non-zero value, while all other coefficients are constrained to have a zero value.

We can immediately reason about intersecting and subsumed pairs of polynomials in the dual space. Given the coordinates for a single polynomial, we can partition the dual space into regions indicating other polynomials which this specific polynomial subsumes, based on universally smaller or larger coefficient values. We refer to this as *extremal* subsumption. For example in Figure 3.2i., the polynomial p_1 has all of its coefficient values smaller than p, while p_2 has large coefficients, thus both of these are subsumed. However we cannot determine whether p subsumes or intersects with polynomials in the shaded regions of the

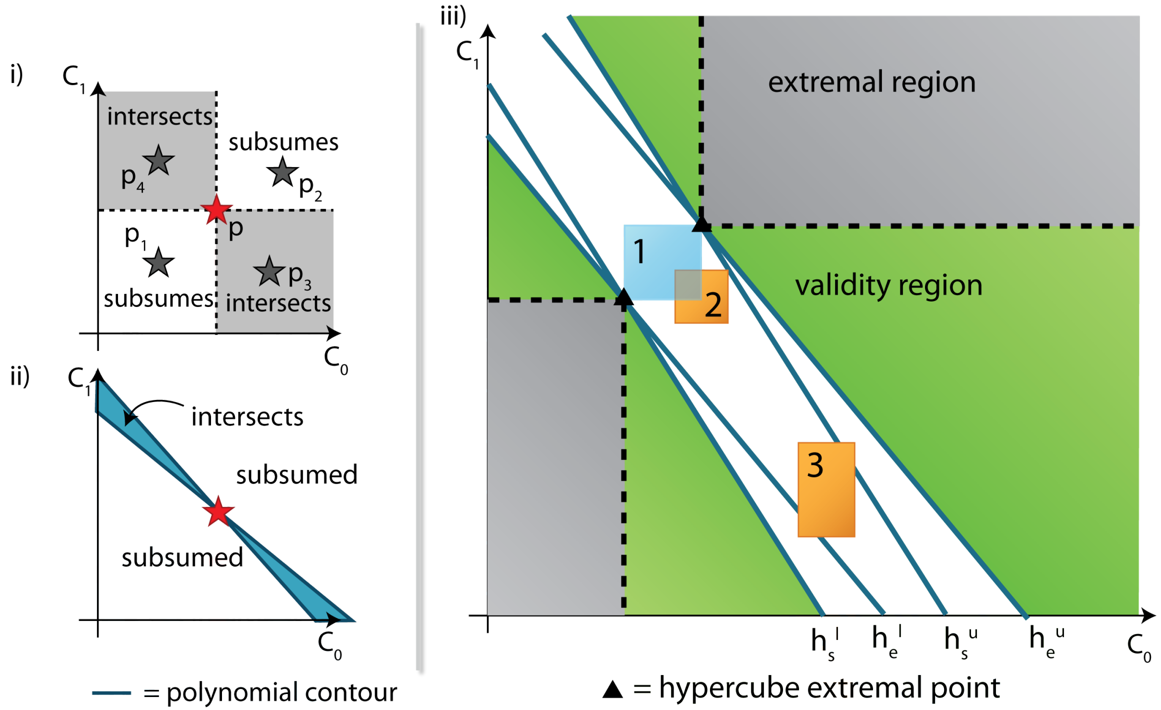

Figure 3.2: Selectivity estimation definition illustrations.

dual space, such as p_3 and p_4 as shown in the diagram. We use this form of dual-space partitioning to provide a coarse-grained filtering of potentially intersecting polynomials.

Now, the dual space represents the general form of a polynomial and does not include any information regarding the variables' domains. Given our inputs are piecewise polynomials, we would like to leverage the extents of each piece's variable domains to further limit the set of intersecting polynomials. In our case we use the validity time ranges of a segment (a 1-dimensional interval), while in the multivariate case, the valid variable domain is a hypercube [92]. We can then apply the following reasoning to both cases.

For a given polynomial and a lower bound on the valid range (for example, the starting timestamp of a segment), we can geometrically represent all other polynomials with the same value at this specific timestamp as a hyperplane in the dual space. Conceptually, this hyperplane uses constant values for the polynomials' variables, and treats the coefficients themselves as variables. For example, given a parameter space over variables $\{x, x^2, x^3\}$, a polynomial $x + x^2 + x^3$, with coordinate $c = (c_0, c_x, c_{x^2}, c_{x^3}) = (0, 1, 1, 1)$, and a lower valid bound $x = 2$, we can define a hyperplane with equation $2c_x + 4c_{x^2} + 8c_{x^3} = 14$. In the multivariate case, the lower valid bound would correspond to the lower extremal point

of the valid hypercube, yielding values for multiple variables. Note we obtain the constant 14 from the point known to satisfy the plane equation, that is the point at coordinate c. To reiterate, this hyperplane represents all other polynomials with equivalent values for a constant variable assignment, and we refer to this hyperplane as a polynomial *contour*. We use this hyperplane as a building block for the estimation technique for each operator.

We can also define a second hyperplane by considering the upper validity bound (i.e., the ending timestamp of a segment). In turn this defines a subspace between the starting timestamp's hyperplane, and the ending timestamp's hyperplane. This region indicates all coefficients whose linear combination results in a value that is equivalent to some value of our input polynomial, within this input's valid time range. We denote this region as the *validity*-subspace of a polynomial segment. Thus, two polynomials intersect only if they mutually lie within the subspace defined by the hyperplanes for each polynomial. If this property does not hold, the polynomials subsume each other within their respective time ranges. This second step refines the previous parameter-space partitioning, leading to precise intersection test for polynomial segments.

3.2.2 Selectivity Estimation in the Dual Space

So far we have described primitive properties for points in the dual space. We can also represent long-term statistics about segments in the dual space, specifically as multidimensional histograms of segments' coefficients, gathered over a time window. The histogram bins are hypercube regions in the dual space as defined by a bin's boundaries. Each bin maintains the total occurrence frequency of any polynomial with coefficients lying within the hypercube region. We denote the minimal corner (i.e. the hypercube corner with the minimal values of all coefficients), and the maximal corner as the *extremal* points of this hypercube. We define the extremal subsumption regions in terms of these extremal points of the histogram bin, as shown in Figure 3.2iii. Before we describe how we can use the validity region containment test described above on histogram bins, we briefly return to the challenge of selectivity estimation, and describe how this test can be used to determine selectivities for various operator's equation systems.

While the histograms aid in computing the number of solutions resulting from any single operator, in order to derive histograms to perform selectivity estimation for a whole plan of operators, we require estimates of the solution ranges themselves, in addition to the number of solutions. For this purpose, we maintain an estimate of the inputs' variable domains for each histogram bin. In more detail, recall Pulse represents segments as a

triple of a reference timestamp, a starting offset, and an ending offset. In the predictive processing mode, we assume a constant input segment length, as given by our adaptive advance selection, implying that the ending offset of our input streams is bounded by this advance value. We can then define an estimate of the valid time range of all segments at the input streams as the lower bound of the starting offset, and the advance offset, and assign this offset timestamp pair to each bin. During the histogram derivation process we describe below, we will also derive solutions given this estimate of the input interval. Note there may be multiple solutions to an operator so our algorithm is capable of handling a list of solution intervals, and we note that at the inputs, we simply have a single element list.

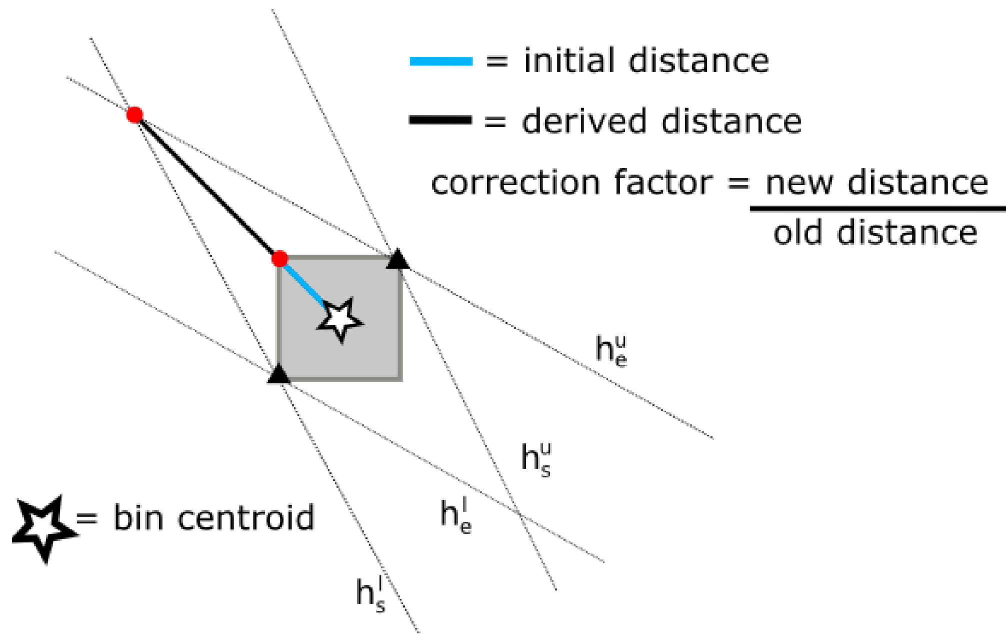

Figure 3.3: Bounding validity range correction factor.

In the same way that a valid time range defines two polynomial contours for a single polynomial, the list of estimated intervals associated with a bin defines a set of polynomial contour pairs. Strictly speaking, these contour pairs are defined with a uniform distribution within the bin's hypercube region (as given by the uniformity assumption within a histogram bin), however we make a simplifying assumption by selecting a bounding pair of contours at the bin's extremal point. These contours define the validity region for a bin, as shown in Figure 3.2ii. We also derive a simple metric to capture the degree of error present by adopting this bounding contour rather than a uniformly distributed set of contours,

and refer to this as the *correction factor* for a estimated interval. Figure 3.3 illustrates a geometric interpretation of the correction factor in 2-dimensions. Intuitively the correction factor represents the average displacement of points within the bin to the point defining the bounding contours. This average displacement is straightforward to compute as the displacement from the bin's centroid given uniformity assumptions. We choose a value of 1 as the initial correction factor, and derive correction factors depending on the operators present in the plan as part of deriving histograms and estimated solutions.

3.2.3 Query-Based Histogram Derivation

Given our representation of polynomial distributions as multidimensional histograms over the dual space, we consider the query expressions for which we are estimating selectivities, and the histograms we need to support this estimation. We primarily consider conjunctive filter predicates (i.e. range queries), join predicates, and min/max and sum aggregates. This leads to several estimation operation types.

Range queries involve comparison operations applied between a single variable and a constant that we allow to be composed with a conjunction operator. To support these conjunctive predicates, we require histograms for all attributes involved in the predicate. We first consider a single comparison operator, which involves an estimation operation between a histogram bin and a constant represented as a point in the dual space. For simplicity, and primarily to focus on the dual space estimation operations, we adopt the attribute independence assumption. In our case, this independence assumption does not mean attributes cannot be represented as polynomial functions of common variables. Rather, we view the coefficients of the polynomials as independent, implying that for a given attribute, the coefficients of this attribute are uniformly likely to appear with respect to the coefficients of any polynomials for other attributes. We can then use this uniformity property across attribute histograms to compute both the derived histogram resulting from applying the conjunctive predicate as well as its selectivity. Clearly this assumption may not hold in certain cases where the polynomial coefficients are defined on equivalent input stream attributes, for example $x = x_0 + v_x t$ and $y = v_x t$. In this case attributes x and y have common coefficients, v_x, hence will be dependent attributes. Handling such cases, for example with a static analysis of coefficient definitions, remains outside the scope of this thesis.

For stateful operators such as joins and aggregates, we derive the state in each case from the inputs to that operator as a distribution itself. In the case of a join, ignoring the effects of having update semantics, this turns out to be the actual input distribution itself

due to the fact that Pulse's join is a natural join on the temporal dimension. We describe how to derive the aggregate's state below. In both cases we arrive at a state distribution in addition to an input distribution, and subsequently apply estimation operations between pairs of histogram bins, one from the state distribution, and the other from the input distribution. For the join, we again allow conjunctive predicates and handle them in much the same way between conjuncts as we do for filters. Thus the estimation operation for the join requires histograms for all attributes present in the join predicate in addition to any attributes used in output expressions. For example the query `select R.a, R.c from R join S on (R.a = S.a and R.b = S.b)` requires histograms attributes for attributes $R.a, R.b, S.a, S.b$ as well as attribute $R.c$ so that it may compute a derived histogram for that attribute. For the aggregate, we the estimation requires histograms for all attributes used as part of the argument to the aggregation function. For example the query `select sum(x) from R` requires a histogram for attribute $R.x$, while the query `select sum(x+y) from R` requires a histogram for both $R.x$ and $R.y$.

3.3 Histogram-Based Estimation of Intermediate Result Distributions

We now return to the problem of detecting subsumption relationships using histogram bins. We define two classes of primitives that we use throughout the histogram derivation process. The first class of primitives are predicate properties for relating a bin b, its validity interval s and a single polynomial c in the dual space:

$$
\begin{aligned}
subsumes(b, c, s, op) &= extremal(b, c, op) \vee validity(b, c, s, op) \\
extremal(b, c, op) &= (\forall i.[c_i \leq l_i] \wedge op =' >') \vee (\forall i.[u_i \leq c_i] \wedge op =' <') \\
contained(c, b) &= \forall i.[l_i \leq c_i \wedge c_i \leq u_i]
\end{aligned}
$$

The $subsumes(b, c, s, op)$ predicate indicates if the coordinate c lies within either the extremal region or validity region of bin b. The $contained(c, b)$ predicate returns whether the coordinate c lies within bin b's hypercube region. We can define similar relationships between a pair of bins b, b' and a validity interval s for bin b:

$$subsumes(b, b', s, op) = extremal(b, b', op) \vee validity(b, b', s, op)$$
$$extremal(b, b', op) = (\forall i.[b_{i,u} < b'_{i,l}] \wedge op ='<') \vee (\forall i.[b'_{i,u} < b_{i,l}] \wedge op ='>')$$
$$overlaps(b, b') = \neg(\bigvee i[b_{i,u} < b'_{i,l} \vee b'_{i,u} < b_{i,l}])$$

The $subsumes(b, b', s, op)$ predicate indicates if the bin b' lies within the extremal region or validity region of bin b according to the relationship specified by op. The $overlap(b, b')$ predicate indicates whether the bin b' overlaps with bin b (note this predicate also holds if b' is contained within b).

The second class of primitives correspond to two types volume computations for determining the volume of the subsumed region of a bin b, based on its relation to both a single polynomial c, and another bin b'. Starting with the single polynomial case, we have:

$$V(b) = \prod_{i=1}^{n} b_{i,u} - b_{i,l}$$
$$V_{sub,i}(b, c, s) = \begin{cases} V_{sub,v} & \text{if } overlaps(b, validity_region(c, s)) \\ 0 & \text{otherwise} \end{cases}$$
$$V_{sub}(b, c, s) = \begin{cases} V(b) & \text{if } subsumes(b, c, s, op) \\ V_{sub,i}(b, c, s) & \text{if } contained(c, b) \\ 0 & \text{otherwise (note this considers a validity region around b, not c)} \end{cases}$$
$$V_{int}(b, c, s) = V(b) - V_{sub}(b, c, s)$$

Note that for $V_{sub,i}$, when the single polynomial's dual space coordinate lies within the bin, we leverage symmetry properties to compute the subsumed volume. That is due to the symmetric nature of validity subsumption between two coordinates, the set of points subsumed by the polynomial's coordinates are equivalent to the set of points that subsume the same coordinate.

We consider two further subsumption primitives that arise when operating on hypercube regions, namely that of when two hypercube regions overlap and subsumption volume within a validity region. We refer to the first case as intra-bin subsumption. Informally, intra-bin subsumption captures the extremal subsumption regions for each point within the hypercube, with respect to the points contained within another bin. For intuition, in the two-dimensional parameter space, this is:

$$\begin{aligned}
V_{sub,i}(b_1, b_2) &= V(b_2 \cap \{subs(x, y) : (x, y) \in b_1\}) \\
&= \int_{x_{1,l}}^{x_{1,u}} \int_{y_{1,l}}^{y_{1,u}} p_1(x, y) \times ((x - x_{2,l})(y - y_{2,l}) + (x_{2,u} - x)(y_{2,u} - y))dxdy \\
&= (x - x_{2,l})(y - y_{2,l}) + (x_{2,u} - x)(y_{2,u} - y)
\end{aligned}$$

where the simplification for the last step occurs due to the uniformity assumption within a bin. Generally, we can compute intra-bin subsumption as:

$$\begin{aligned}
V_{sub,i}(b_1, b_2) &= V(b_2 \cap \{subs(x_1 \ldots x_d) : (x_1 \ldots x_d) \in b_1\}) \\
&= \int \cdots \int_{x_{1,i,l}}^{x_{1,i,u}} p_1(x_1, \ldots, x_d) \times (\prod_{i=0}^{d} x_i - x_{2,i,l} + \prod_{i=0}^{d} x_{2,i,u} - x_i)dx_1 \ldots dx_d \\
&= \prod_{i=0}^{d} x_i - x_{2,i,l} + \prod_{i=0}^{d} x_{2,i,u} - x_i
\end{aligned}$$

The second primitive is needed due to the fact that with hypercube pairs, one bin's validity hyperplanes may intersect the other hypercube, indicating that the other hypercube is partially subsumed. Thus we need to compute the volume resulting from the intersection of a hyperplane and hypercube, depending on both the comparison operator and the relative position of the two hypercubes. We leverage existing tools for computing volumes of convex polytopes, such as polymake [36] or Vinci [16].
This leads to the following generalized subsumption volume computation method:

$$V_{sub}(b, b', s, s') = \begin{cases} V(b)V(b') & \text{if } subsumes(b, b', s, op) \vee subsumes(b', b, s', op) \\ V_{sub,i}(b, b')V_{sub,i}(b', b) & \text{if } overlaps(b, b') \\ V_{sub,v}(b, b', s)V_{sub,v}(b', b, s') & \text{otherwise} \end{cases}$$

$$V_{int}(b, b', s, s') = V(b)V(b') - V_{sub}(b, b', s, s')$$

Note that the subsumption primitive in the first condition above does not hold for partial subsumptions given validity volumes. Thus we only consider validity volumes when both hypercubes partially subsume each other, and not the asymmetric case when one subsumes the other but not vice-versa. We will now describe how these three primitive subsumption methods on histogram bins are used to compute the selectivities of filters, joins and aggregates.

3.3.1 Filters

Here, we define the estimation operation for a simple range predicate of the form $x < c$ where x is a modeled attribute, and then describe how we can use the derived histogram for each of these expressions in computing the selectivity of a conjunctive predicate. We start by computing estimate of the solution ranges arising in the presence of an intersection as follows:

$$t_b^{deriv} = \underset{\{[s_l,s_u]\}}{solve} \left[\sum_i w^i(r_i - c_i) < 0 \right] \cap t_b$$

We simply use the centroid of the bin, $r = \{r_1 \ldots r_n\}$, as a representative of all polynomials within the bin, and solve the difference equation with coefficients $\{r_i - c_i\}, i = 1 \ldots d$ to compute solutions, considering only those solutions contained in the bin's estimated input interval. Note that the coefficients $\{c_i\}$ correspond to a constant segment with value c. These solutions are then used as the estimated intervals for any downstream derivation. Let us define d_b^{est} as the number of solutions obtained for bin b. We can compute the derived frequency of a histogram bin as:

$$f_{b,s}^{deriv} = f_{b,s} c_f \left(\frac{V_{sub}(b,c,s) + d_{est}^b V_{int}(b,c,s)}{V(b)} \right)$$

Here, we see that the derived frequency is a product of the original frequency and correction factor when considering subsumed segments, provided the subsumption satisfies the comparison operator. When the constant's point is contained in the histogram bin, we consider both the extremal and validity regions defined at the point, and compute the fraction of the hypercube's volume that overlaps with these subsumed regions. These contribute $f_{b,s}c_f$ segments to the result, while the intersected regions contribute $f_{b,s}c_f d_{est}^b$, with both scaled by their respective volume fractions. Otherwise the constant's point lies entirely within the intersection region, and yields $f_{b,s} d_b^{est} c_f$ segments during the statistics gathering window. We can compute then operator's segment selectivity for a histogram H_x as:

$$\sigma_{x<c}(H_x, c) = \frac{1}{F} \sum_{b \in bins(H_x)} \sum_{s \in S} f_{b,s}^{deriv}$$

where $F = \sum_{b \in bins(H_x)} \sum_{s \in S} f_{b,s}$ is the total occurrence frequency for the histogram. The above selectivity equation computes an estimate of the total number of output segments

per histogram bin as a product of the occurrence frequency of the bin, and the number of solutions resulting from applying the operator to the bin's representative, provided the bin does not subsume the constant.

Finally we can compute a derived correction factor. Based on the description of the correction factor in Section 3.2.2, this turns out as:

$$\begin{aligned} e_{new} &= n \cdot (r - p) \\ c_{f,b}^{deriv} &= e_{new} / e_{old} \end{aligned}$$

where n is the hyperplane normal, r the bin centroid, p the known point on the hyperplane, and e_{old} the previous distance computed for the bin as a result of a derivation at an upstream operator. Note the distance e_{new} is maintained for each bin and solution to facilitate correction factor derivation at a downstream operator.

We can then use the definitions of these properties to compute the equivalent properties for a conjunctive predicate. Given our coefficient independence assumptions for different attributes, we can keep a histogram for a conjunctive predicate simply as a set of per-attribute histograms. We define a set C denoting the attributes used in a conjunctive predicate, for example given a predicate $x < c_1 \wedge y < c_2 \wedge z < c_3$, we have $C = \{x, y, z\}$. The set C represents the individual attribute histograms we access to compute a derived histogram. First, we compute the estimated solutions and correction factors for the derived bin:

$$\begin{aligned} t_b^{deriv,conj,a} &= \frac{1}{\prod_{a' \in C - \{a\}} F_{a'}^{deriv}} \sum_{\{b_1 \ldots b_k\} \in bins^{C-\{a\}}} \prod_i f_{b_i}^{deriv,a_i} \left[t_b^{deriv,a} \bigcap_i t_{b_i}^{deriv,a_i} \right] & \forall a \in C \\ f_{b,s}^{deriv,conj,a} &= \begin{cases} f_{b,s}^{deriv,a} & \text{if } t^{deriv,conj,a} \neq \emptyset \\ 0 & \text{otherwise} \end{cases} & \forall a \in C \\ c_{f,b}^{deriv,conj,a} &= \frac{1}{|bins^{C-\{a\}}|} \sum_{\{b_1 \ldots b_k\} \in bins^{C-\{a\}}} \prod_i c_{f,b_i}^{deriv,a_i} & \forall a \in C \end{aligned}$$

Above, in the derived histogram for the conjunctive predicate, we compute the average estimated solutions for each attribute's bin by considering bin combinations for other attributes and computing the average common solution amongst these bin combinations. Note these bin combinations correspond to a joint distribution across the conjunctive predicate's attributes. We use the term $bins^C$ to refer to bin combinations for attributes in the set C. We can then prune those bins with non-empty solutions. Finally, we compute each

attribute's correction factor in a similar fashion to the estimated solutions, again by considering bin combinations and computing the average product of correction factors across these bin combinations. For completeness we also describe how we can compute the selectivity of the conjunctive predicate p applied to stream S:

$$\sigma_p(H_S, p) = \prod_{a \in attrs(p)} \frac{1}{F} \sum_{bins(H_S(a))} \sum_{s \in S_a^{deriv,conj}} f_{b,s}^{deriv,conj,a}$$

where the term $attrs(p)$ refers to the attributes used in predicate p, $H_S(a)$ is the histogram for attribute a (i.e. a projected histogram given all attributes on stream S) and F is the total frequency of any attribute in H_S. We must ensure that this frequency is equivalent for all attributes by scaling the derived frequencies before the derived histogram can be used downstream. This is done with the aid of a normalizing constant:

$$\text{scale factor } = \frac{r}{F_a^{deriv}}$$

which uses the intermediate result cardinality $r = F\sigma_p(H_S, p)$.

3.3.2 Joins

We will now define the estimation operation for a comparison of two modeled attributes x, y of the form $x < y$. We support predicates specified as conjuncts of this basic comparison expression. We assume that histogram boundaries are aligned, otherwise this can easily be done by considering the union of bin boundaries and scaling frequencies according to the uniformity assumption within bins. We now derive histograms, estimated solutions and correction factors for a join operation on two bins a, b from the left and right inputs of the join operator respectively. We denote the input solutions for these bins as s_a, s_b respectively.

We start with the estimated solutions, stating that these can be computed by solving a difference equation between the two centroids of the bins $r_a = \{r_{a,i}\}$ and $r_b = \{r_{b,i}\}$, and considering only those solutions common to both sets of input solutions:

$$t_{a,b}^{deriv} = \underset{\{[s^l, s^u]\}}{solve} \left[\sum_i w^i (r_{a,i} - r_{b,i}) \leq 0 \right] \cap (s_a \cap s_b)$$

Next, we consider the derived histogram frequencies, which can be computed for a given pair of bins and solutions as follows:

$$f^{deriv}_{a,b,s_a,s_b} = f_a f_b c_{f,a} c_{f,b} \left(\frac{V_{sub}(a,b,s_a,s_b) + d^{est} V_{int}(a,b,s_a,s_b)}{V(a)V(b)} \right) \frac{|t^{deriv}_{a,b}|}{t_{opt}}$$

The above equation computes the derived frequency as a product of individual bin frequencies, scaled according to the ratio of the subsumed volume and intersecting volume between the two bins. Furthermore, we scale the derived frequency according to the input segment rate given by the ratio of the length of the derived solution range and the length of the optimization period t_{opt}. This is necessary due to the natural equality constraint on the valid time ranges present in Pulse's join, which implies that the segments collected on one stream are not compared to the entire set of segments collected on the other, but instead only compared to the number of segments sharing common valid ranges.

Next, we can compute the selectivity of the join predicate given two histograms H_x, H_y for attributes x, y respectively:

$$\sigma_{x<y}(H_x, H_y) = \frac{1}{F_x + F_y} \sum_{a \in bins(H_x)} \sum_{s_a \in S(a)} \sum_{b \in bins(H_y)} \sum_{s_b \in S(b)} f^{deriv}_{a,b,s_a,s_b}$$

where $F_x = \sum_{a \in bins(H_x)} \sum_{s_a \in S(a)} f_{a,s_a}$ and $F_y = \sum_{b \in bins(H_y)} \sum_{s_b \in S(b)} f_{b,s_b}$.

Finally we must also derive a correction factor for the new histogram bin, and do so as follows:

$$\begin{aligned} e_{new,a} &= n_a \cdot (r_a - p_a) \\ e_{new,b} &= n_b \cdot (r_b - p_b) \\ c^{deriv}_{f,a,b} &= e_{new,a}/e_{old,a} \times e_{new,b}/e_{old,b} \end{aligned}$$

Just as in the case of the filter's selectivity estimator, n_a, n_b are hyperplane normals, r_a, r_b are bin centroids, p_a, p_b are known points on the hyperplane, and $e_{old,a}, e_{old,b}$ are distances computed for upstream correction factors. These are each defined for bins in histograms H_x, H_y.

Finally, we support conjunctive predicates where each term of the predicate is a comparison of two modeled attributes as described above. The derived histogram's bin frequencies, estimated solutions and correction factors can all be computed in an identical way as with the filter operator. To recap this primarily involves considering bin-combinations of the derived histograms for each comparison term, determining averages of solutions and correction factors for those bins with common solutions, as well as scaling the respective frequencies based on the most selective comparison expression.

3.3.3 Aggregates

Our method for computing aggregates' segment selectivities performs two steps, the first is to obtain an instantaneous distribution of the aggregate function applied to segments corresponding to different key values, and the second is to then consider temporal aggregation. We describe these methods for both `min,max` aggregates and `sum,avg` aggregates.

Min/max aggregates

We begin by describing the contents of the instantaneous histogram derived following a min or max aggregate. In this histogram, we would like each bin to capture the occurrence frequency of the polynomials contained in the bin being the result of the aggregate, given input frequencies per key value. Thus we must compute the frequency with which the polynomials in a histogram bin dominate other polynomials according to whether we have a min or a max function. These frequencies can be computed with our subsumption methods. We consider a max aggregate, although the equations below can be trivially modified for a min aggregate. We define the frequency of a bin in the instantaneous histogram for key k as:

$$f_{b,s}^{inst}(k) = f_{b,k}(1 - \frac{\sum_{k' \in K} \sum_{b_d \in dominated(b,bins(k'))} \sum_{s' \in S(b_d)} f_{b_d,s',k'} [\![s \cap s' \neq \emptyset]\!]}{F_K})$$

where $K = keys - \{k\}$, $F_K = \sum_{k' \in K} \sum_{b \in bins(k')} f_{b,k'}$. Also, the function $dominated(b, bins(k'))$ returns the set of bins (for key k') dominated by the bin b and is computed as those bins lying completely within the extremal region of defined by the centroid of bin b.

Once we have the instantaneous distribution, for temporal aggregation, we explicitly leverage the constant advance that we use during predictive processing. We use this constant advance to determine the number of input segments that would present in a window, and given that our instantaneous distribution is derived from these inputs, we can derive a distribution of the coefficients in a window as a product of the instantaneous distribution. Note that this also relies on independence between the coefficients of sequential segments. Thus the distribution of inputs to the temporal aggregation is a joint distribution $H_w = \prod_{i=1}^{n} H_{inst} = H_{inst}^{n}$, where $n = w/adv$ for window size w and constant advance adv. However as we'll see, we can avoid computing this joint distribution, which can be expensive since its complexity $O(bins^n)$.

The critical functionality we capture as part of temporal max aggregation is the inclusion of constant segments resulting from an extremal instantaneous point lying within

the sliding window. We now describe how we compute the occurrence frequencies of these segments, and add them to the derived histogram in addition to the relevant portions of the instantaneous distribution. Based on our description of these constant segments, we add these following the arrival of a new input segment to the aggregate that is smaller than the window's existing maximum, that also does not cause the removal of the window's maximum. Thus we can compute the constant segments' frequencies based on computing the probabilities of these two conditions over the window distribution. The probability of the first event, where the addition of a segment does not cause the removal of the maximal segment is as follows. We label this event r:

$$\begin{aligned} p(r_{b,s}) &= p(\text{no extremal removal from b,s}) \\ &= p(\exists i \in [2 \ldots n].seg_1 < seg_i) \\ &= 1 - p(\forall i \in [2 \ldots n].seg_1 > seg_i) \\ &= 1 - (\sum_D f_{b,s}^{inst}/F_s)^{n-1} \end{aligned}$$

where $D = dominated(b, bins(H_{inst}))$. The probability of the second event, where the added segment is dominated by any existing segment in the window, is labeled a below and can be computed as:

$$\begin{aligned} p(a_{b,s}) &= p(\text{no extremal append to b,s}) \\ &= p(\exists i \in [2 \ldots n].seg_{n+1} < seg_i) \\ &= 1 - p(\forall i \in [2 \ldots n].seg_{n+1} > seg_i) \\ &= 1 - (\sum_D f_{b,s}^{inst}/F_s)^{n-1} \end{aligned}$$

where $D = dominated(b, bins(H_{inst}))$. We can then compute the constant segments' frequencies as:

$$f_{b,s}^{const} = \sum_{\{b' \in bins: max(b') \in [c_0^l(b), c_0^u(b)]\}} f_{b',s}^{inst} p(r_{b',s}) p(a_{b',s})$$

In the above equation, we use the property that the value of the constant segment is equal to the extremal value present in the window. Thus the frequency of a bin containing these constant segment has contributions from numerous other bins according to the maximal value, i.e. the maximal polynomial contour, associated with the bin.

Next, we describe our approach for computing the validity range of these constant segments, so that we may assign estimated solution intervals to the bins corresponding to these segments. We again use our independence assumption between sequential segments and state that the validity range of the constant segment follows a geometric distribution, corresponding to the number of segments added to the window prior to a new addition dominating the existing maximum of a window. Furthermore, we can use the number of segments present in a window under our constant advance assumption as an upper bound on the number of segments considered added in this geometric distribution. We can then state the following about the solution range of these constant segments, denoted $t_{b,s}^{const}$:

$$t_b^{const} = \frac{1}{n}\sum_{i=1}^{n}(E[t_b] \times (i-1) \times adv)p(a_{b,s})^{(i-1)}(1-p(a_{b,s}))$$

Above $E[t_b]$ is the expected validity range of a constant segment given upstream derived ranges from an input segment, and is defined as:

$$E[t_b] = \sum_{bins} [\![dominates(b,b')]\!]n + (1 - [\![dominates(b,b')]\!])min_{s \in S(b)}(s)$$

Finally, the correction factor for both regular bins and constant segment bins is simply the average correction factor of all bins contributing the maximal segment or constant.

$$c_{f,b,s}^{deriv} = avg_{\{b' \in bins : max(b') \in [c_0^l(b), c_0^u(b)]\}} c_{f,b,s}$$

Sum aggregates

We now turn our attention to sum aggregates, and apply a similar process of deriving an instantaneous distribution of the sum's value and then consider temporal aggregation over a sliding window. The instantaneous distribution can be computed by considering bin combinations for different key values. For each combination, we compute the set of valid ranges and track the extremal polynomials within each common range. We can compute the summation of these extremal polynomials, and their frequency as follows:

$$f_{b,s}^{inst} = \sum_{\{\{b_1 \ldots b_k\} \in bins^k :\}} f_{b^*} \prod_{i=1}^{k} \frac{f_{b_i,s}}{F_i}$$

where $b^* = argmax_{b_i}(\{f_{b_1} \ldots f_{b_k}\})$ and $F_i = \sum_{b \in bins(i)} f_b$.

Once we have obtained the instantaneous distribution, we can then use this in conjunction with the description of a sum's window function to derive the temporally aggregated output distribution. Recall that in a sum's window function we have three components: a head segment, a tail segment, and fully enclosed segments that contribute constants to the window's result. We can model the distributions of each of these components using our instantaneous distribution. Using the sequential segment independence assumption, both the head and tail segment distributions are the instantaneous distribution. This leaves the fully enclosed segments and the values they contribute. We capture this by considering the distribution of the definite integral of $n-2$ sequential segments (n segments per window, except for the head and tail segment).

$$f^C_{b,s} = \sum_{b \in bins(H_{inst}) : \sum_{s \in S}[\int_s \sum_{i=1}^{d} r_i t^i dt] \in [c^l_0(b), c^u_0(b)]} f^{inst}_{b,s}$$

The above defines frequencies for an instantaneous integral distribution, from which we derive the window integral distribution:

$$f^{C,w}_{b,s} = \sum_{\{\{b_1 \ldots b_{n-2}\} \in bins(H_C)^{n-2} : [\sum_{i=1}^{n-2} \sum_{j=1}^{d} r_j(b_i) t^j] \in [c^l_0(b), c^u_0(b)]\}} f_{b^*} \prod_{i=1}^{n-2} \frac{f_{b_i}}{F_i}$$

where $b^* = argmax_{b_i}(\{f_{b_1} \ldots f_{b_k}\})$ and $F_i = \sum_{b \in bins(i)} f_b$. Note that in the above equation we use the term $bins(H_C)^{n-2}$ to denote the $n-2$-way combinations of bins from the histogram H_C.

Subsequently we can derive the window distribution by combining with the integral distribution of the head and tail segments. Note that in these distributions, the integration operation simply transforms bins by dividing their respective coefficients with the degree of the variable associated with the coefficient's dimensions. Furthermore, in the case of the tail segment, we actually have a negative integral distribution computed as $H_{tail} = H_C - H_{head}$ where H_{head} is the integrated head segment distribution. Furthermore, note that the head and tail segments share the same estimated validity ranges due to their derivation from the instantaneous distribution. Thus we can simply use these estimated solutions for the derived histogram:

$$t^{deriv}_b = t_b$$

Also, the correction factor is again the average of all contributing bins.

$$c^{deriv}_{f,b,s} = avg_{b \in bins(H_{inst}): \sum_{s \in S}[\int_s \sum_{i=1}^{d} r_i t^i dt] \in [c_0^l(b), c_0^u(b)]} c_{f,b,s}$$

3.3.4 Histogram-Based Selectivity Bounds

In addition to selectivity estimations, as we shall see below, our optimizer's cost model attempts to use the risk involved in selectivity estimation as part of its choice of query plan. We represent risk as a function of two selectivity bounds and compute these bounds based on an extreme cardinality estimation technique presented in [14]. While the authors consider the containment assumption for equi-joins in this work, the key idea we use is that of computing selectivities based on assuming extremal skew between histogram bins rather than scaling independently when computing selectivities and using this to bounds. We primarily describe how we apply the technique to conjunctive predicates, and refer to the above paper for how to address join graphs. We do not consider single attribute filters and aggregate functions with a single attribute argument, since these simply propagate extremal selectivities involved in the attribute used. We do not consider more complex aggregates that use multiple attributes.

For conjunctive predicates, we define two properties σ_{min} and σ_{max} representing selectivity bounds as follows. We assume we have both original $f_{b,s}$ and derived frequencies $f^{deriv}_{b,s}$ for each individual comparison term in the conjunctive predicate. Let's consider σ_{max} first, where we pick the comparison term with largest total derived frequency F^{deriv}_{max}. This acts as an upper bound on the result cardinality if we ignore any other comparison terms in the conjunctive predicate. The key idea is that we then pick histogram bins for other terms with largest frequency within this result cardinality limit, and subsequently compute the selectivity for these other terms using only the chosen bins.

As an example, consider the conjunctive predicate $x < 100 \wedge y < 50 \wedge z < 10$ posed on a stream histogram of cardinality 1000. Furthermore, suppose the term $x < 100$ has the largest derived frequency (i.e. is the least selective), with $F^{deriv}_{max} = 400$ as the upper bound on the result cardinality. We sort the derived frequencies of attributes y and z, picking bins in descending order of derived frequency such that the total frequency is F^{deriv}_{max}. Let us denote this set of frequencies for attribute y as $DF_y = \{f^{deriv}_1, \ldots, f^{deriv}_n\}$, and similarly DF_z for attribute z. Now for each chosen bin, we also require their original frequencies and denote these as $OF_y = \{f_1, \ldots, f_n\}$ and OF_z. We can then compute an upper bound on the selectivity of the term $y < 50$ as $\frac{\sum DF_y}{\sum OF_y}$ and similarly for the term $z < 10$ as $\frac{\sum DF_z}{\sum OF_z}$. Note

that these are overestimates due to the our greedy selection of bins in descending frequency order. Finally we can compute our upper bound for the predicate as:

$$\sigma_{max} = \frac{F^{deriv}_{max}}{F} \times \frac{\sum DF_y}{\sum OF_y} \times \frac{\sum DF_z}{\sum OF_z}$$

We now present the general form of this selectivity bounds:

$$\begin{aligned}\sigma_{max}(H,p) &= \prod_{a \in attrs(p)} \frac{\sum_B f^{deriv}_{b,s}}{\sum_B f_{b,s}} \\ \text{where } &\min_{B \subset bins(H(a))} \sum_B f_{b,s} \text{ s.t } \sum_B f^{deriv}_{b,s} = R \\ \text{and } R &= \arg\max_{a \in attrs(p)} F^{deriv}, a\end{aligned}$$

Note that it is unlikely that we will find bins matching the constraint that their total frequency sums to R. Instead we simply apply our greedy selection as described above until we have at least met a sum of R, and simply scale the frequencies of the last bin chosen.

For σ_{min}, we start with the comparison term with lowest total derived frequency, and instead pick bins for other terms in ascending order of derived frequency.

$$\begin{aligned}\sigma_{min}(H,p) &= \prod_{a \in attrs(p)} \frac{\sum_B f^{deriv}_{b,s}}{\sum_B f_{b,s}} \\ \text{where } &\max_{B \subset bins(H(a))} \sum_B f_{b,s} \text{ s.t } \sum_B f^{deriv}_{b,s} = R \\ \text{and } R &= \arg\min_{a \in attrs(p)} F^{deriv}, a\end{aligned}$$

3.4 Experimental Evaluation

In this section, we present our experimental evaluation of our selectivity estimator, implemented in the Pulse framework, on top of the Borealis stream processing engine. We evaluate the selectivity estimator's characteristics both in terms of accuracy and overhead on synthetic datasets and an real-world workload based on the NYSE dataset seen in the previous section. We compare our selectivity estimator to a simple sampling-based technique for estimation. These experiments are performed on a Intel Core 2 Duo T7700 2.4 GHz processor, 4GB RAM, running a Linux version 2.6.24 kernel. Note that we used the summer 2008 release of Borealis, which is not designed for utilizing multiple cores.

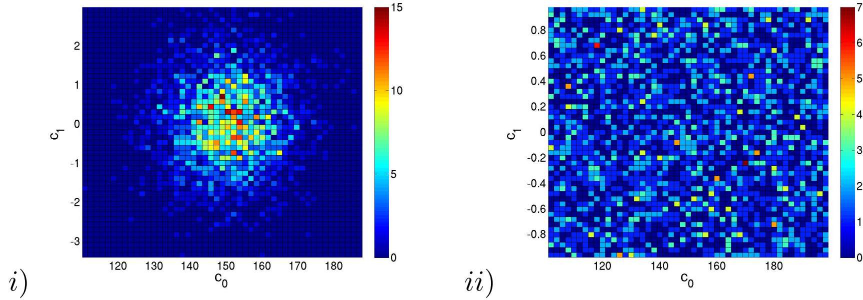

Figure 3.4: Parameter spaces of segments in the synthetic i) normal distribution, and ii) uniform distribution

Before we dive into the details of the evaluation, we briefly comment on the sampling technique used as a comparison point. There is a large volume of literature in the database community investigating sampling techniques, where these works are often diverse both in terms of the underlying principles of the algorithms used for estimation, (e.g. wavelets [62], sketching [33], etc.) and in terms of the kinds of queries considered, commonly including sum and average aggregates, top-k queries, and so forth. We apply an extremely simple form of sampling, where we sample the input stream and execute the query plan as is on the sampled stream. We compare the cardinality of the result stream for a fixed size workload, to the sample size to compute a selectivity estimate. Note that as we shall see in the following sections, the queries used during selectivity estimation do not involve joins, hence we need not apply more complex join sampling techniques. We now study how our parameter-space histogram technique performs in comparison to such as sampling technique.

3.4.1 Dataset Analysis

We start with an analysis of the datasets used in these experiments, in particular looking at the parameter space of the polynomial segments present in the datasets being processed. We start with the synthetic dataset whose parameter spaces conform to the distribution used to generate the dataset. We use two synthetic datasets, whose polynomials consist of coefficients drawn from a normal distribution and a uniform distribution respectively. Their parameter spaces are shown in Figure 3.4, which depicts a histogram of the parameter space, using colors to represents the frequency of coefficients within each bin. Note that the segments in our datasets are simple linear functions, hence the figures represent a two-dimensional parameter space. The x-axis in these figures represent values of the coefficient c_0 while the y-axis represents values of a function f applied to the coefficient c_1, where

Distribution	Parameter	Value	Description
Uniform	$[l_0, u_0, l_1, u_1]$	$[100, 200, -1, 1]$	Domain bounds for c_0, c_1
Normal	$[\mu_0, \mu_1]$	$[150, 0]$	Mean values for c_0, c_1
Normal	$[\sigma_0, \sigma_1]$	$[50, 1]$	Standard deviations for c_0, c_1

Figure 3.5: Synthetic dataset generation parameters

$$f(x) = \begin{cases} k - \log(-x) & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ \log(x) + k & \text{otherwise} \end{cases}$$

where $k = \min(\log(x) | x > 0)$. The function f maps a two-sided wholly positive or wholly negative exponential function passing through the origin into a linear function passing through the origin, effectively allowing use to view positive and negative values of the c_1 coefficient in a logarithmic scale.

Now we illustrate the parameter spaces for the NYSE dataset. This dataset consists of the segments fit by an offline algorithm for the seven stocks with largest trading volume during a one month period on the NYSE in January 2006. Their parameter spaces are illustrated in Figure 3.7. The x- and y-axes in these figures are identical to those for the synthetic dataset parameter spaces. The key property to note is that these distributions are wide, and short – that is to say coefficient c_0 ranges over a far larger domain than coefficient c_1, that is the stock prices are generally slow-changing. This is not unexpected for a real world dataset, for example many physical properties such as temperatures, are slow-changing. Note that we reproduced this property in our synthetic datasets, as can be seen by the ranges of the c_1 domain for both normal and uniform distributions in Table 3.5. Additionally, we note that several of these parameter spaces exhibit the presence of frequency clusters along both the c_0 and c_1 dimensions. For example in Figure 3.7iii) there are 6 apparent clusterings, three along the c_0 dimension, each of which have similar clusters at an offset along the c_1 dimension. This suggests an oscillatory characteristic amongst the segments.

NYSE Dataset

With these datasets in hand, we turn our attention to evaluating the accuracy and performance of our estimator starting with the NYSE dataset. The query workload for which we determine selectivities is comprised of single-attribute filter operations, where we detect if the stock price crosses a threshold. For accuracy and performance metrics, we look at both

Distribution	μ_0	μ_1	σ_0^2	σ_1^2
s12	77.657	-0.00832	16.636	0.0207
s3762	22.914	-0.00111	3.498	0.000257
s4878	27.002	-0.00294	0.293	0.00478
s6849	38.404	-0.000608	0.986	0.000601
s6879	70.537	16.350	16.958	136314.256
s6973	127.607	0.0267	3.463	2.472
s8329	37.466	0.00341	10.976	0.0314

Figure 3.6: NYSE dataset characteristic parameters

relative and absolute errors, and estimator execution times. We study two independent variables influencing these accuracy and performance metrics. These are the actual selectivity of the query, which is controlled by adjusting the threshold value used in the query, as well as the binning scheme used by input stream histograms for derivation purposes. Each binning scheme adopts a grid-based approach to picking bin boundaries, differing exponentially in the bin widths along each dimension.

We illustrate three sets of results on the NYSE dataset. This starts with a takeaway result depicting the accuracy achieved by our histogram technique and the overhead in providing such a level of estimation accuracy, compared to the sampling technique. We summarize that our experiments show a negative result, namely our estimation technique is outperformed by a small but noticeable amount both in terms of accuracy and performance. The remaining two results provide experimental details for our technique, comparing the accuracy and performance metrics for varying query selectivities and bin widths.

Figure 3.8 compares the accuracy achieved and the overhead needed to produce the estimate for our histogram-based technique to a histogram technique enhanced by two sampling methods. The first sampling technique, referred to as segment sampling, randomly samples the input stream prior to building a histogram on the resulting sample of segments. This produces a histogram with fewer bins than simply building the histogram over the full input stream of segments. The second sampling technique, bin sampling, constructs the histogram as usual, but then applies a weighted sampling of the bins, to selectively include a subset of the bins in the final histogram used for estimation. Note that the overhead measured in this experiment is simply the computational overhead for the estimate, which in the case of our histogram-based technique involves processing each bin, and for sampling, applying query processing to each sample. It does not include the overhead of maintaining and updating histogram bins, nor that of collecting samples. Each point on the histogram

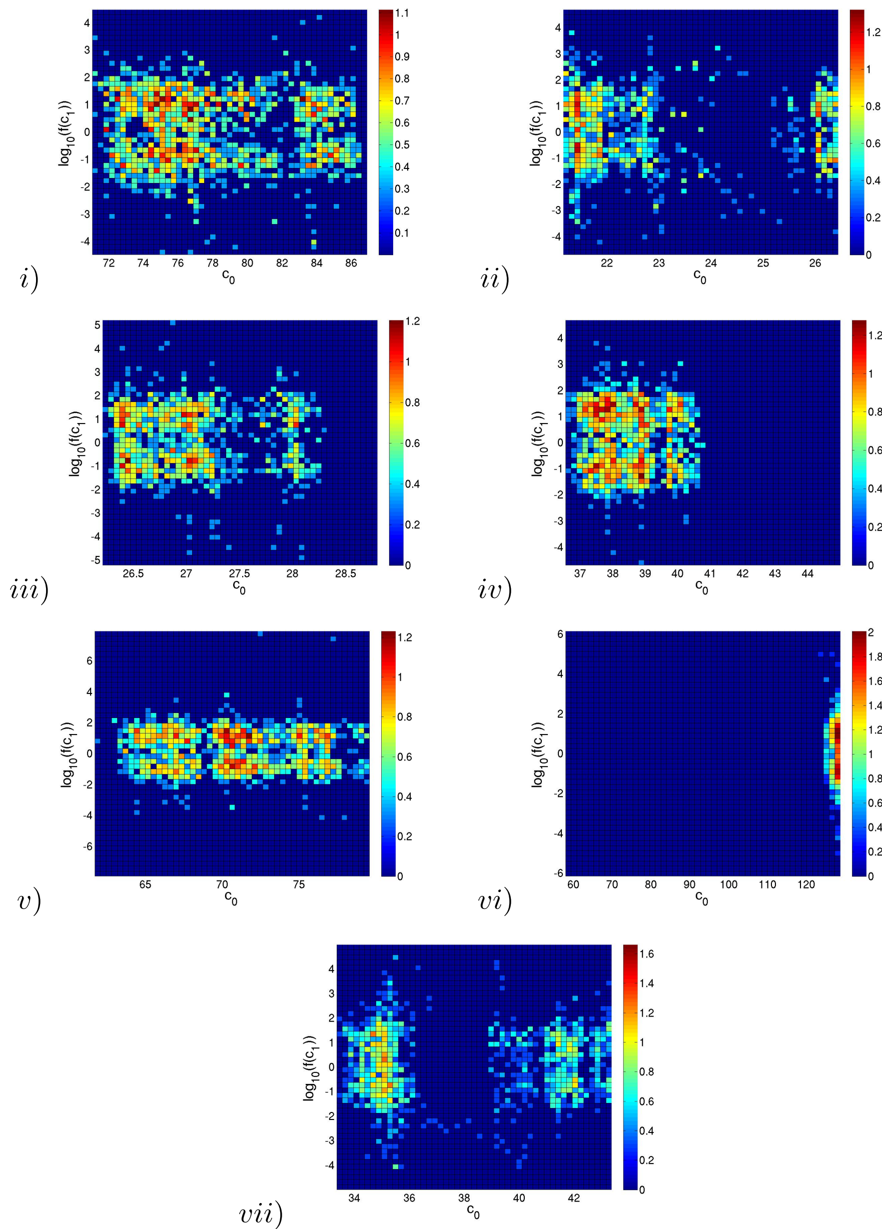

Figure 3.7: Segment parameter spaces for seven stocks from the NYSE dataset, stock ids are: i) 12, ii) 3762 iii) 4878 iv) 6849 v) 6879 vi) 6973 vii) 8239.

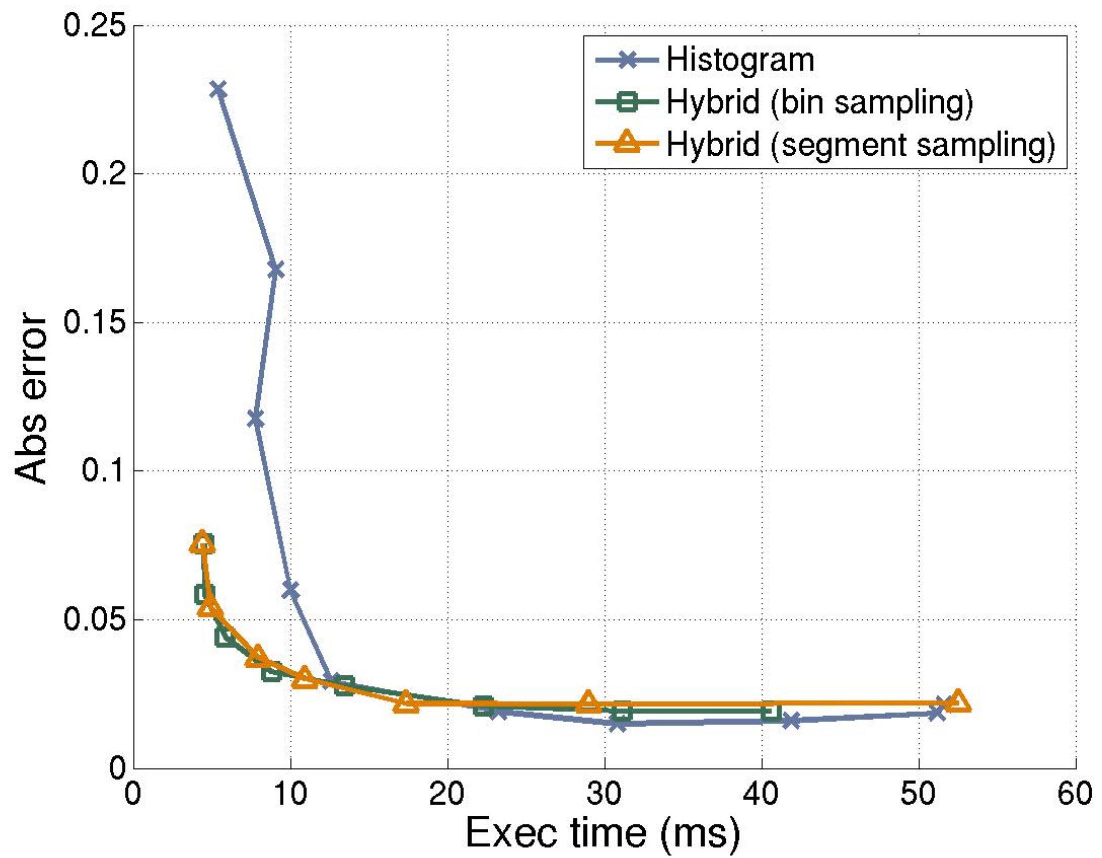

Figure 3.8: Comparison of accuracy vs. performance tradeoff for histogram technique and sampling on the NYSE dataset.

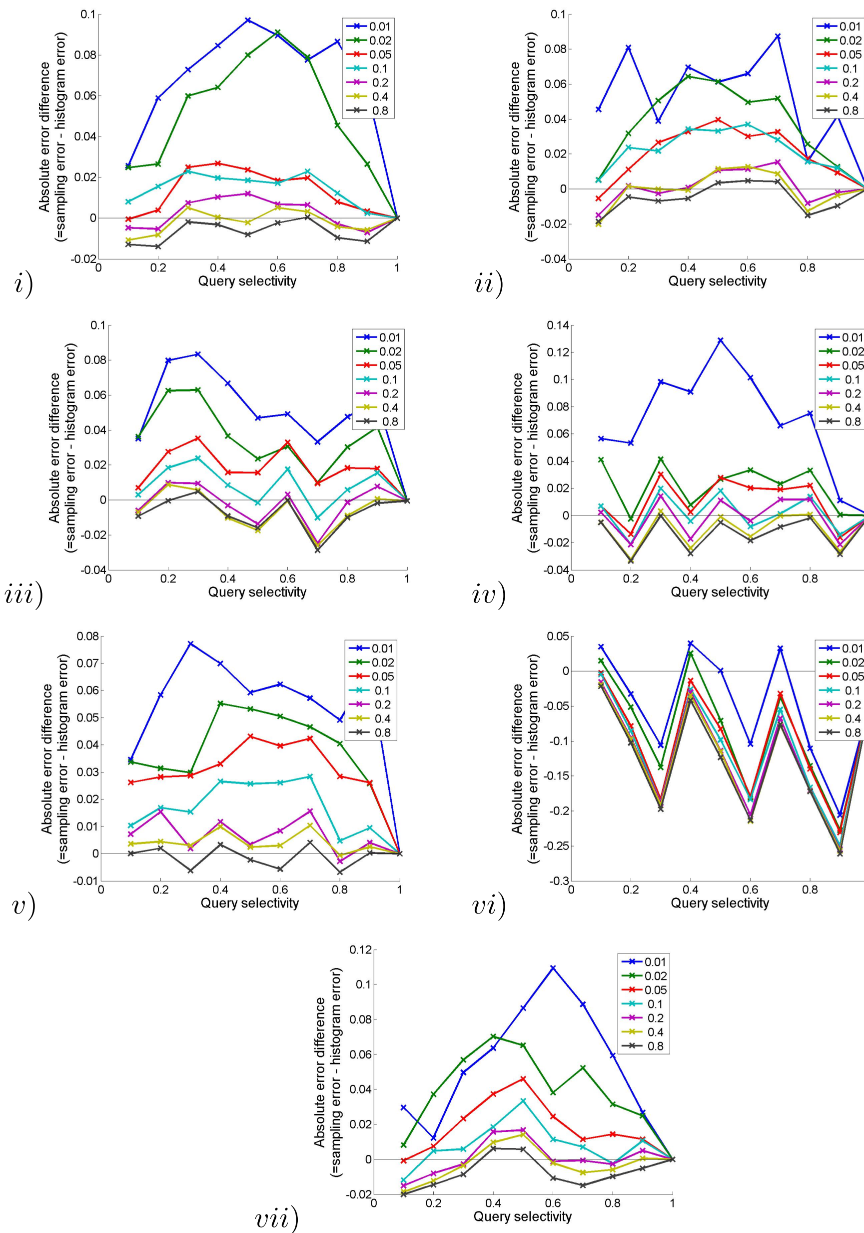

Figure 3.9: Absolute error differences between sampling at a variety of rates, and histogram-based estimation using a large number of bins.

curve corresponds to a particular binning scheme, averaged over a query workload whose selectivities range from 0.1 to 1.0, and the seven stock symbols with parameter spaces seen above. Points on the sampling curve correspond to varying sampling rates, also averaged over our query workload and stocks.

This figure shows that sampling significantly enhances estimation for the histogram method in the cases where the binning scheme uses a coarse grid, providing significantly greater accuracy, i.e. lower absolute errors, for an equivalent estimation overhead. The two sampling enhanced techniques differ amongst themselves at the finer-grained binning schemes where bin sampling is able to provide the same level of accuracy with significantly lower overhead, due to its more refined ability to prune bins from the histogram, and subsequently provide retain more relevant bins in this histogram for estimation. We finally remark, that while the histogram enhanced with bin sampling provides the best result in terms of both accuracy and overhead, both of the other techniques are clearly capable of providing low overall accuracy within a fairly small range of overheads.

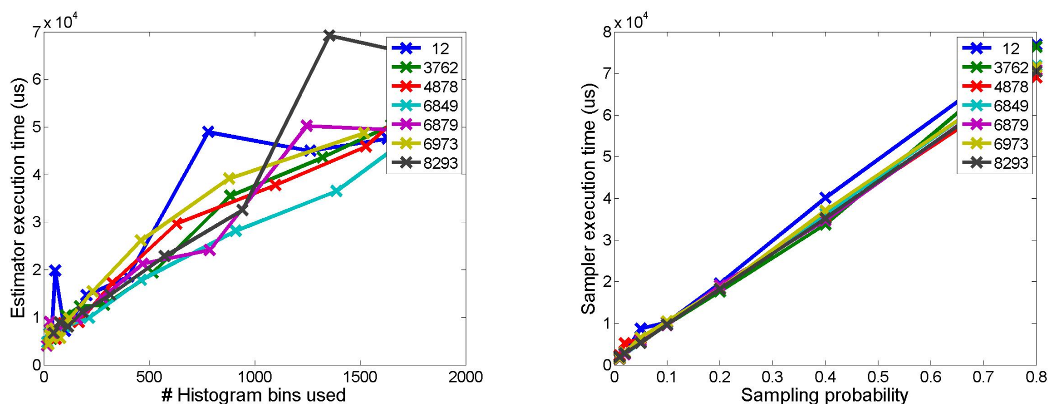

Figure 3.10: Estimation evaluation overhead for i) histogram-based and ii) sampling-based techniques on the NYSE dataset.

Figure 3.9 illustrates the difference in absolute errors for our selectivity estimates for each stock's dataset. In each plot, we compare the sampling method evaluated at a variety of sampling rates to the binning scheme using the largest number of bins. Positive absolute error differences indicate where our technique outperforms the associated sampling method. Note these plots focus on the issue of accuracy in isolation and do not factor in the overhead of performing the estimation. This figure illustrates the error for several query workload selectivities, and is intended to highlight any sensitivities or biases in the estimator under varying query selectivities. The primary note we make here is that the estimator is fairly

robust towards varying query workloads. In many of these figures, absolute errors start off at low values, increase towards the midpoint of the selectivity domain, and finally decrease as selectivity approaches a value of 1. This trend can be understood by considering the primary source of error in our histogram-based technique, which is the approximation of an individual segment's valid hyperplanes by considering bin hyperplanes. Our technique incurs greatest error when the majority of bins use these hyperplanes for selectivity estimation, as opposed to pruning or subsumption directly performed by the hypercube corresponding to histogram bins. This naturally occurs at the midpoint of the actual selectivity domain. Figure 3.9vi) however does not follow such a trend, and in the behavior exhibited there can be attributed to the selectivity estimates computed by our histogram-based technique. For this particular stock's parameter space, the choice of binning scheme results in highly discretized set of selectivity estimates, essentially a step function. Thus at certain steps, our estimator's accuracy improves significantly, and the cause of such a step function is that the binning scheme and the bin widths chosen result in a highly discretized parameter space with few active bins. Subsequently, neighboring threshold values have the same impact in terms of pruning or subsuming these few bins, leading to a step function for our histogram-based estimates.

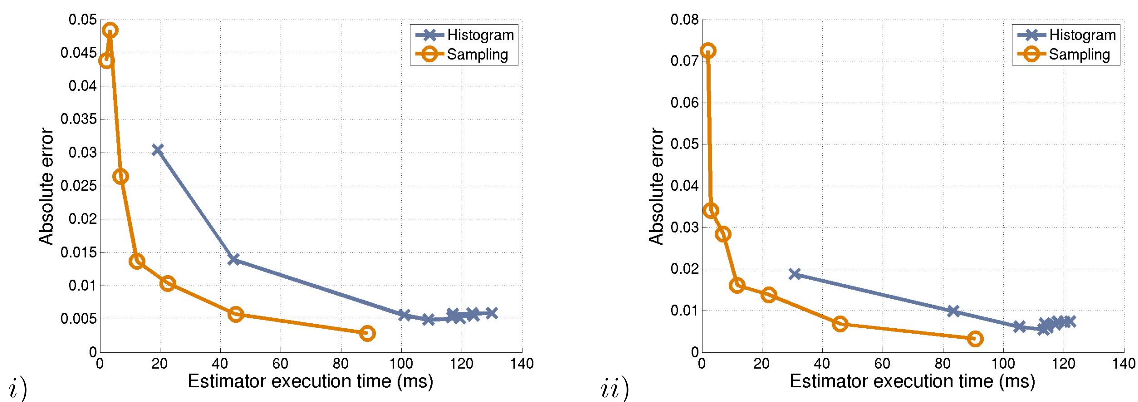

Figure 3.11: Comparison of accuracy vs. performance tradeoff for histogram technique and sampling on the synthetic dataset.

Figure 3.10 shows the overhead in estimating selectivities using our histogram-based technique as well as sampling. In Figure 3.10i) we have a line for each stock symbol, at varying binning schemes. These binning schemes differ in the number of histogram bins present in the domain of coefficient values, with each binning scheme containing twice as many bins as the previous scheme. Given this number of available bins in the coefficient

value domain, the x-axis on this graph plots the number of bins actually used, that is the coefficient values often sparsely populate their domain, hence not all bins end up containing any segments. The general trend here is that the overhead grows as a linear function of the number of bins used. This matches our expected computational complexity arising from looping over such bins, performing pruning, subsumption and intersection tests as the main work during histogram derivation for a predicate. We notice the volatile nature of these plots and remark that in our initial inspection, the source of such volatility appears to arise from the choice of experimental setup parameters relating to the interaction between various components of the Borealis framework (for example sleep times between loading a query from a client, and initiating the estimation in the backend server process). This is somewhat to be expected in experimenting and developing with complex systems, and we view the analysis of such interactions beyond the scope of this thesis. Figure 3.10ii) displays the overhead for sampling-based estimation, for a variety of sampling rates indicating the fraction of the input workload used for estimation. The overhead exhibits a linear relationship to the sampling rate that is to be expected, and can be explained by the fact that we are processing a linear system of equations for each sampled segment. Note that due to the different independent variables on these plots, it is difficult to directly compare overheads for the two techniques, rather this must be done factoring in accuracies as shown in the accuracy and overhead figure above (Figure 3.8).

3.4.2 Synthetic Dataset

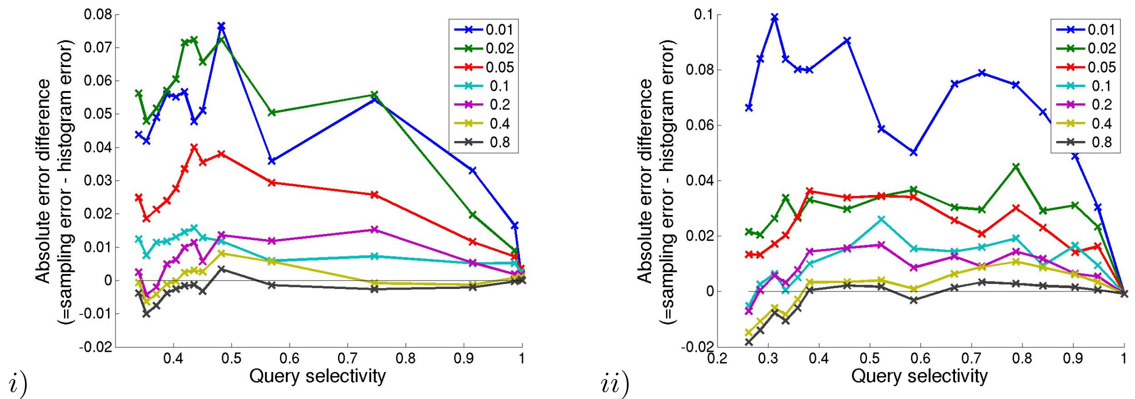

Figure 3.12: Absolute error differences between sampling at a variety of rates, and histogram-based estimation using a large number of bins on the synthetic dataset.

We now investigate these properties and metrics for the synthetic dataset with parameter

spaces shown in Figure 3.4, starting with our accuracy and tradeoff results. Figure 3.11i) shows the accuracy achieved by the histogram-based technique, in its original form, and enhanced with bin-sampling in comparison to the overhead of estimation for each method for a normally distributed parameter space, while Figure 3.11ii) applies to a uniformly distributed parameter space. Points in both diagrams are averaged over varying query selectivities, and each point corresponds to a different binning scheme and sampling rate of the histogram- and sampling-based lines respectively. These figures corroborate that sampling-enhanced selectivity estimation dominates the histogram-based method, providing both better accuracy with lower overheads. We notice the tighter clustering of points towards the larger overhead values for the histogram-based method in both diagrams. We also remark that the separation between the two lines in this comparison of tradeoffs appears greater than that seen with the NYSE dataset. This is caused by the binning schemes all utilizing a large number of bins, whereby almost every segment in the input workload is binned into unique histogram bins.

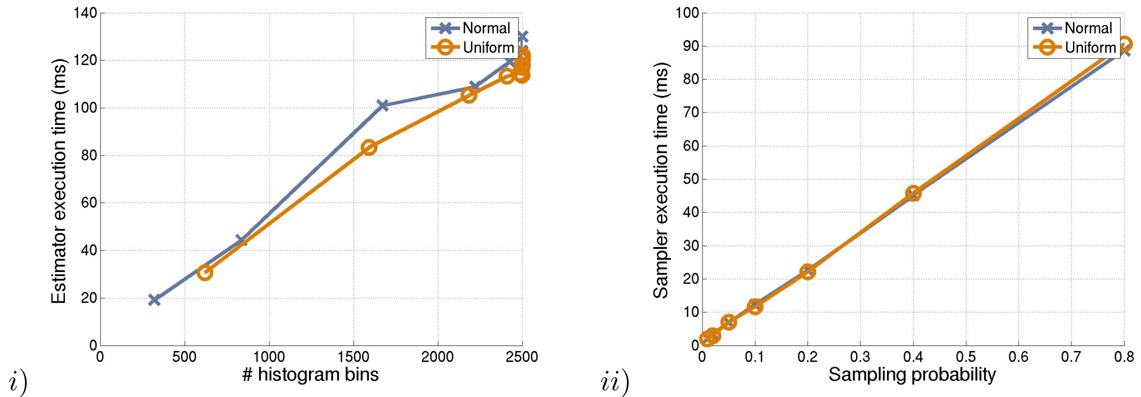

Figure 3.13: Estimation evaluation overhead for i) histogram-based and ii) sampling-based techniques on the synthetic dataset.

In Figure 3.12, we analyze the accuracy achieved by both techniques in more detail, plotting the difference in absolute errors of the estimation techniques. Specifically, we compute error differences between sampling at a variety of rates and the binning scheme employing the smallest bin widths, for both normal and uniform parameter spaces. In both plots we display error differences for set of queries with actual selectivities ranging from 0.1 to 1. The results shown here exhibit trends similar to those for the NYSE dataset, namely that absolute errors are larger near in the central part of the selectivity range, and tail off towards the upper extreme of this range. We also note that in terms of accuracy alone,

our estimator performs reasonably well on these synthetic datasets, with our histogram technique proving advantageous even in comparison to sampling up to 40% for both normal and uniform datasets at selectivities above 0.4. Indeed our technique is marginally worse than an 80% sample rate in both cases.

Finally, Figure 3.13 illustrates the estimation overhead for the histogram-based technique for varying bin widths, and for sampling over several sampling rates. Both results here exhibit similar trends to those from the NYSE dataset, namely that the histogram estimator's overhead is linear in terms of the number of bins actually populated in the histograms, while the sampler's overhead is linear in terms of the sample size. We note that in Figure 3.13i) several of the binning schemes result in input histograms where each segment is assigned to a unique bin. This again highlights the dependency of our technique on the choice of binning strategies, and the difficulty in picking binning schemes.

Chapter 4

Adaptive Multi-Query Optimization with Segment Selectivities

With a selectivity estimation technique for our segment-based processing in hand, we can now turn to the issue of developing a cost model for query processing to guide cost-based query optimization. In this work, we focus on sharing and ordering mechanisms from the many choices of query optimization techniques. Our reasons for doing so include the fact that these techniques are two of the most common semantic optimization techniques at the query level, and thus are significantly influenced by selectivities as defined in our query processing context. In this section, we start with describing how ordering and sharing techniques can be applied to our queries for performance optimization, before moving on to describing a cost model representing the processing overhead and capturing the effects of these optimization mechanisms on processing overhead. We then present adaptive query optimization algorithms based on this cost model that monitors properties of the running queries, to detect performance thresholds violations, and then performs a set of local query adaptations to improve system performance.

In the following two sections, we consider two sharing techniques, one which relies on using covering subexpressions for common query plan subtrees, denoted *cover-sharing*, and another which eliminates duplicates between compatible (but independent) query plan subtrees prior to processing the element. We refer to the later as *union-sharing*. Both of these cases apply to query elements that are compatible in terms of their schemas, and the attributes processed. The related literature [97] includes descriptions of compatible

expressions based on their signatures of input attributes used, and algorithms for finding these compatibilities.

4.1 Covered Sharing

In more detail, given a set of compatible subexpressions across a set of queries, our cover-sharing technique uses a covering subexpression that produces a superset of intermediate results for all query elements preceding each compatible expression, all the way to the inputs. Note that at query inputs, stream processing engines typically duplicate streams into each query's execution contexts (i.e. queues), much in the same way that traditional databases use multiple scan operators while processing many queries unless explicitly optimized. Thus covering all elements to the input streams allows us to eliminate this input duplication.

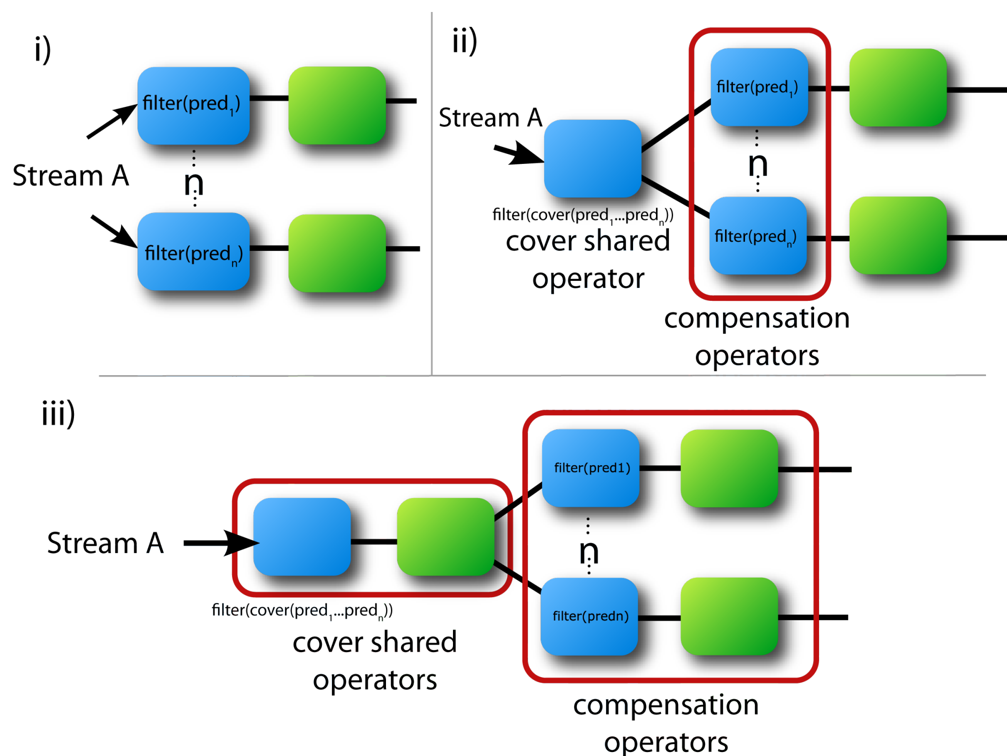

Figure 4.1: Cover-sharing example.

We'll now present a simple example of cover-sharing. Figure 4.1 provides an illustration of this example of a pair of cover-shared queries, outlining how two query plans can be merged into a single one through the use of covering operators from the input streams. We start with two separate operator chains in step i), and in step ii) we share the first operator

by computing a covering expression for the various predicates, and adding compensation operators following the cover-shared operator. In step iii), as an abstract example, we show the plan supposing we are able to share the second operator. This requires that the compensation predicates are able to commute with the second operator. Note that in the cover-sharing algorithm, all operators to the input stream A are also cover-shared.

The tradeoffs in utilizing a cover-sharing mechanism lies in the fact that the covering expression is produces a superset of intermediate results, in addition to requiring compensation operators downstream from the covering expression that are capable of filtering this superset to the exact outputs generated by each individual query. Intuitively, if there is not much overlap in the intermediate results produced by operators, the covering expression will eliminate much duplicate processing, while we still incur the overhead of the compensation operators. We note here that there is no need to limit ourselves to a single covering expression. In our system, we consider the use of multiple covering expressions over various subsets of queries, leading to the problem of how to appropriately select both the queries to cover as well as the covering expression itself. For example, given a set of queries $Q = \{q_1 \ldots q_n\}$ that each have a compatible operator, we can divide the set Q into m disjoint subsets $\{Q_1 \ldots Q_m : Q_i \subset Q\}$, where each subset Q_i is processed by an *instance* of the shared operator with its own covering expression. We refer to the set Q_i as the set of consumers for the shared instance, and m the sharing degree of the operator.

Covering Expression Computation. The final mechanism-related concern we describe for cover-sharing is that of how to compute a covering expression for a given consumer set. We primarily focus on conjunctive predicates found in filters and joins and return to this momentarily. We remark that aggregates are only compatible if they use the same aggregation function with identical argument expressions, for example the aggregate expressions `sum(a)` and `sum(a+b)` are not compatible. Now we return to the case of conjunctive predicates for compatible operators. Note that by the definition of compatibility [97], the consumer set must have a common set of attributes across each of their predicates. We focus on only the comparison terms in the conjunction that refer to one of these common attributes. The remaining terms in any consumer's conjunction is processed by that consumer's compensation operator.

In the case of both filters and joins, each of these common attributes will be compared to a constant (e.g. `x < 5`) or another common attribute (in the same input stream for filters, for example `R.x < R.y`, and potentially in different input streams for joins, such as `R.x < S.y`). Constant comparisons can be handled in a straightforward manner provided we have identical comparison operators – we simply find a subsuming constant. Thus for three

queries with constant comparisons `x < 5`, `x < 7` and `x < 20`, we use the expression `x < 20`. To handle pairs of modeled attributes, we first identify if the same pairs are compared across all consumers, using the same comparison operator. If this is the case, we simply find a subsuming comparison expression. Otherwise we add a disjunction of the two comparison terms.

For example consider the comparison terms `x < y`, `3x < 2y`, `4x > y`, and `3x > 3y`, each posed by a different query. The covering expression is the term:

$$(((x > 0 \lor y > 0) \land x < y) \lor ((x < 0 \lor y < 0) \land 3x < 2y)) \qquad \text{cover expr part i.}$$
$$\lor\ (((x > 0 \lor y > 0) \land 3x > 3y) \lor ((x < 0 \lor y < 0) \land 4x > y)) \qquad \text{cover expr part ii.}$$

We make two remarks for the above. First, the goal here is to minimize the number of disjunctive terms added to the common expression between the original terms, since we solve a separate equation system for each disjunction. The above covering expression actually includes all the original terms so it may not seem clear how this simplifies evaluation. The answer is as follows: the general structure above is that we add covering expression identification terms, such as $(x > 0 \lor y > 0)$, for each original term. These should be checked at the solution boundary to determine the covering expression. While this term must be computed for segments that take on both positive and negative values, if either the segment for `x` or `y` is entirely positive or negative, we can use boolean short-circuiting to evaluate only one equation system. Thus in the above example we would only evaluate one equation system for part i. of the covering expression and another for part ii. of the covering expression. This is necessary since the two parts have opposing comparison operators between attributes `x` and `y`.

Note that each cover-shared operator also keeps track of the compensation operators that must execute to distinguish each consumer individually. A downstream operator can only be implemented with a cover-sharing algorithm if these compensation operators commute with the downstream operator. For example if the current shareable operator is a filter and the downstream shareable operator is an aggregation, the downstream operator cannot be implemented in cover-shared fashion since the filter's compensation operators (which will be filters themselves) cannot be pushed beyond any shared aggregate.

4.2 Union Sharing

The idea behind union-sharing is to perform ad-hoc, adaptive sharing at an arbitrary shareable operator in the query plan. Assume we have a set of (potential) consumers $\{c_1, \ldots, c_n\}$

currently executing a set of shareable operators in non-shared fashion. Furthermore, suppose each consumer applies this non-shared operator to its unique set of inputs $\{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$. As suggested by its name union-sharing involves computing a union of the intermediate inputs to each consumer, namely $\mathcal{I} = \bigcup \mathcal{I}_k, k = 1 \ldots n$. The key is that applying this union operator produces a superset of any single consumer's intermediate inputs, that is $\mathcal{I}_k \subseteq \mathcal{I}, k = 1 \ldots n$. This superset is processed by both a shared covering operator, and then a compensation operator for each consumer. In a similar fashion to cover-sharing, we also consider selecting multiple shared instances as well as picking their associated consumer sets, and requiring compensation operators to distinguish each query's exact output set. We draw our inspiration for the union-sharing technique from related work with on-the-fly sharing for aggregate operators [55], adapting the techniques to work with polynomial segments.

At a high-level, processing with a covering operator and a compensation operator is similar to cover-sharing, but the key difference between the mechanisms is the use of a union operator to construct an intermediate input superset, rather than covering each expression upstream through to the input streams. Thus union-sharing creates a covering expression for only a specific shareable operator, while every consumer continues to execute any upstream operators independently. We now dive into more detail on this intermediate input superset construction.

There are a few subtleties to applying a union operator for the purpose of sharing. In the Borealis stream processor, tuples are copied into each individual query's execution context, hence simply applying a union produces duplicate tuples. We would like our intermediate input superset to be duplicate-free, otherwise we would process essentially the same input multiple times with the shared operator as result of copied inputs. Thus we apply a duplicate elimination algorithm following this union, removing equivalent intermediate results. We now provide intuition on the benefits of union-sharing before describing the duplicate elimination algorithm.

The tradeoffs in union-sharing are also the cost of processing the shared operator over a superset of results, and then distinguishing each query's results with compensation operators. In comparison to the cover-sharing, union-sharing provides a more adaptive technique that can be applied at arbitrary compatible operators with fewer changes to the running individual queries and merged queries, rather than requiring all upstream operators to be cover-shared. However union-sharing incurs the cost of explicitly performing duplicate elimination, which is much simpler in the case of cover-sharing since this simply means eliminating the aforementioned copy operation at the input streams. Figure 4.2 illustrates

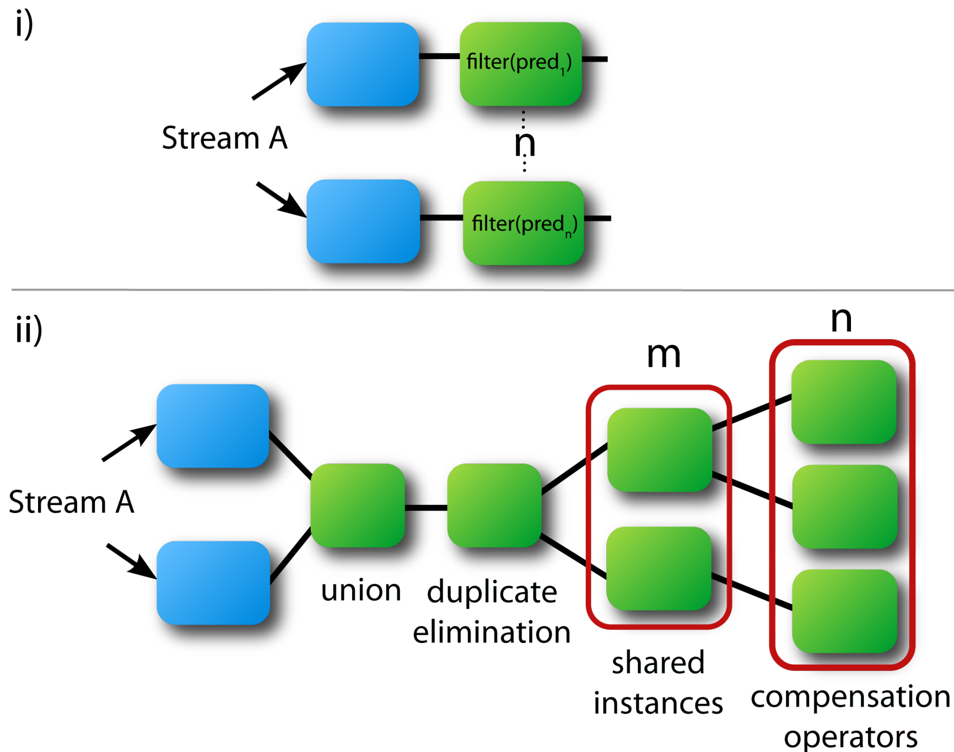

Figure 4.2: Union-sharing example.

the concept of union-sharing, explicitly highlighting the points where both the union and duplicate operators are applied, as well as the shared and compensation operators.

Duplicate Elimination and Data Sharing. We now return to the duplicate elimination we perform, since this is custom to our context of processing polynomial segments. We define duplicate segments as those that share both equivalent coefficients and valid time ranges. Duplicates primarily occur due to subsumed segments following processing by any operator. We can detect duplicates in a straightforward manner following a union operator by maintaining a buffer based on the monotonic reference timestamp, and checking and eliminating from the contents of this buffer periodically. The buffer can then be flushed and any non-duplicate inputs can be processed by the shared instances downstream.

We also describe data sharing as an extended, aggressive form of duplicate elimination to reduce the size of the intermediate input superset. One way to view data sharing is as a dual to the query sharing described above. Data sharing uses the fact that our segments are a compact representation of a set of tuples, and in particular our segments may exhibit covering properties. We define covering as a two segments with equivalent coefficients, but

where one segment's valid time range is a strict subset of the other.

Now one option for data sharing would be to decompose any set of covering segments S (i.e. a set of segments where each member covers or is covered by another) into disjoint segments. Each disjoint segment would be covered by multiple members of S, and we could then share the processing of these members by instead processing the disjoint segment. However as it turns out from a simple analysis, the increased segmentation from creating disjoint segments is identical to the benefits of sharing, leading to no net gains. The analysis is straightforward, each covering set member that uses a disjoint segment can share processing for that disjoint segment. However the creation of a disjoint segment also leads to a residual segment for the covering set member, eliminating the benefit of sharing the disjoint segment. Note that this residual must exist, otherwise two covering set members are duplicates in the sense of equivalent coefficients and valid time ranges.

Instead the approach we use is semantic data-sharing, which requires extending the shared operator and compensation operator's algorithm. The idea again is to use covering segments, but rather than decomposing these into disjoint segments, we produce the maximally covering segment m for the set S, as well as keeping tracking of the valid time range for each member of S for use in compensation operators. The key idea here is that we use the segment m to quickly detect when we need to perform no work for the covering set S. For example consider a filter on m. If this filter does not produce a result for m, it will not produce any results for the set S. However if it does produce a result for m, then we must perform more work, at each compensation operator, to determine each individual result for the set S. Thus we must maintain the valid time range for each member of S with m. This technique also applies to joins and their predicates, providing we assume both filters and joins apply predicates and maintain state on individual key values in the presence of keys. Note for joins, this property also helps to ensure that the shared operator's windows cover those windows of the compensation operator.

Aggregates turn out to be more complex as a result of windows maintained over multiple key values. We only consider min and max aggregates, and do not apply data sharing to sum aggregates. Consider two segments a and b with the same key value k, where a covers b, and a is an input to a shareable aggregate agg_1, and b an input to agg_2. Now suppose we have a shared implementation of these two aggregates, and consider adding segment a to the shared operator's window. The problem lies in the fact that segments for another key value may contribute to the window for agg_2 in the difference of a and b's valid time ranges. This limits the properties we can conclude about the outcome of adding b to the window for agg_2 by considering a and the shared operator's window.

In particular we can state the following. For a min aggregate, using covering segments allows us to compute the minimal envelope at a shared operator, across all consumers of that operator. Subsequently considering covering segment updates for a min aggregate allows us to determine when the covering segment causes no state updates. However for max aggregate, we can compute the maximal envelope across all consumers. In turn, this allows us to determine when a covering segment updates state across all consumers, subject to the consumer's valid time range.

Summary. In summary, we use both duplicate elimination and data sharing to aggressively reduce the size of the intermediate inputs to our shared operators. This handles both the issue of copied tuples for multi-query optimization and exploits the fact that we have a compact representation of data. Data sharing when applied simultaneously with query sharing allows us to perform the following semantic optimizations. For filters and joins, we can eliminate redundant processing by detecting when inputs produce no results for any consumer. For min aggregates, we can detect when inputs produce no state updates for any consumer, and for max aggregates we detect inputs that produce state updates for any consumer.

4.3 Ordering

The basic idea in exploiting processing orders is that eliminating segments from processing as early in the pipeline as possible reduces processor load, that can instead be spent working on segments which produce outputs. The difficulties in determining a suitable order lies in the large number of possible orders, and subsequently the large number of conditional selectivity distributions. Since optimizers assume independence assumptions between attributes, attribute dependencies late in the operator chain are incorrectly estimated, leading to suboptimal plans in the presence of high degrees of correlation after the initial operator.

Prior work has studied this problem in the case of streamed filters [8], using monitoring conditions that maintain an invariant property on operators' selectivity-cost ratios, in a single query's evaluation order. Specifically, the conditions maintain that each operator has a greater drop probability-to-cost ratio than any downstream operator. We leverage the monitoring conditions developed in this work, integrating their execution with our conditions for detecting sharing opportunities, and ensuring that any adaptations performed provide a correct implementation of the query in the presence of shared work.

4.4 Processing Cost Model

Recall the high-level structure of our cost model as defined in Section 3.1.1 as consisting of both processing cost and risk metrics. We define our processing cost and risk metrics as $c(est)$ and Δc, denoting the cost as a function of the estimated selectivities, and the risk as the extremal range of costs. We use these two terms in our objective, and define them as follows:

$$
\begin{aligned}
cost =& (1-\rho)c(est) + \rho\Delta c \\
c(est) =& c_{cov}(est) + c_{un}(est) + c_{ns}(est) \\
c_{cov}(est) =& \sum_{(e,Q)\in A} \sum_{(inst,S)\in inst(e)} \sigma_{up}^{est}(e, inst)(c_{cv} + \sigma_{cv}^{est}(e, inst)\sum_{q\in S} c_{cp}) \\
c_{un}(est) =& \sum_{(e,Q)\in B} \sigma_{up}^{est}(e, union(e))(c_{un} + c_{dup} + \sigma_{dup}^{est} \times \sum_{(inst,S)\in inst(e)} c_{cv} + \sigma_{cv}^{est}(e, inst)\sum_{q\in S} c_{cp}) \\
c_{ns}(est) =& \sum_{(e,Q)\in C} \sigma_{up}^{est}(e, op(e))c_e \\
\Delta c =& c(max) - c(min) \\
=& c_{cov}(max) - c_{cov}(min) + c_{un}(max) - c_{un}(min) + c_{ns}(max) - c_{ns}(min)
\end{aligned}
$$

where

$$
\begin{aligned}
\sigma_{up}(e, o) =& \sum_{m_1\in in(b_1)\wedge q(m_1)\in q(b_1)} \cdots \sum_{m_k\in in(b_k)\wedge q(m_k)\in q(b_k)} \\
& \prod_{inst_{i,a}\in U_A} \sigma_{inst_{i,a}} \prod_{(i,inst_{i,b})\in U_B} \sigma_{dup_i}\sigma_{inst_{i,b}} \prod_{j\in U_C} \sigma_j \\
U_A =& \{inst_{i,a} : i < e \wedge (i, Q) \in A \wedge (inst_{i,a}, S_{i,a}) \in inst(i) \wedge inst_{i,a} \prec^* o \wedge inst_{i,a} \prec^* m_{min}\} \\
U_B =& \{(i, inst_{i,b}) : i < e \wedge (i, Q) \in B \wedge (inst_{i,b}, S_{i,b}) \in inst(i) \wedge inst_{i,b} \prec^* o\} \\
U_C =& \{j : j < e \wedge (j, Q) \in C \wedge j \prec^* o \wedge j \prec^* m_{min}\} \\
\prec^* =& \cup_{i\in\{A,B,C\}} \prec^i
\end{aligned}
$$

$$
\prec((o, Q), (o', Q')) = \begin{cases} 1 & \text{if } o < o' \wedge ((Q' \subseteq Q \wedge o' : \{\mathcal{A}, \mathcal{C}\}) \vee (Q \subseteq Q' \wedge o' : \mathcal{B})) \\ 0 & \text{otherwise} \end{cases}
$$

In the above formalization, we define processing cost $c(est)$ as a summation of three terms representing the cost of query elements processed with a cover-sharing algorithm

(c_{cov}), a union-sharing algorithm (c_{un}), and finally elements that are not shared and execute as normal (c_{ns}). Each of these three components range over the expressions and consumer sets of query elements implemented with the associated algorithm, which are represented above as the sets A, B, C. Specifically A is the set of expressions and consumer queries that are implemented with a cover-sharing algorithm, B the same for the union-sharing algorithm, while C tracks the non-shared expressions. Throughout this section we use the notation A (or B, C) to describe the set of expressions and consumer queries implemented with cover-sharing, and $\mathcal{A}$ or $(\mathcal{B}, \mathcal{C})$ to state the type of algorithm used for a specific expression and consumer query (i.e. $i : \mathcal{A}$ states an instance i is implemented in cover-shared form). The shared implementations additionally allocate the consumer queries amongst multiple instances, creating a second-level grouping of consumer queries that the c_{cov} and c_{un} terms range over. The set of instances for an expression e is denoted by $inst(e)$, and consists of an instance element and assigned consumer query set pairs $(inst, S)$.

Each of the three cost components c_{cov}, c_{un}, c_{ns} compute a processing cost as a product of upstream selectivities and per-element costs. The per-element cost of a cover-shared implementation includes the cost of the covering instance (c_{cv}) and any compensation elements (c_{cp}), while a union-shared implementation includes the union and duplicate elimination elements (c_{un}, c_{dup}) in addition to the covering and compensation elements. For non-shared elements, the per-element cost is simply that of the original element.

The term $\sigma_{up}(e, l)$ refers to the combined selectivity of all upstream elements from the element l processing expression e. This indicates the fraction of the query's inputs that a specific element processes. We distinguish between the expression e and the element l since there may be multiple elements processing the same expression, for example multiple shared instances. The element l from which we compute upstream selectivities includes instances in the case of cover-sharing (denoted as $inst$), the union operator for union-sharing, $union(e)$, and the non-shared operator $op(e)$. The definition of σ_{up} is based on three sets U_A, U_B, U_C which refer to the set of upstream elements implemented in cover-shared, union-shared and non-shared fashion respectively. The sets capture additional topological information to represent the dataflow between element instances, through the relationships expressed by the $\prec^*$ operator.

This operator defines a reachability relationship, where $a \prec^* b$ that states that an element b is reachable (i.e. supports dataflow) from element a. We use the term $\prec^*$ to denote the transitive closure of the operator $\prec$ that defines the relationship for neighboring elements. The $\prec$ operator holds whenever the expression ordering relationship $<$ holds (this relationship is defined on the original query plans where there are no shared operators),

and the containment property on consumer sets shown in the definition of $\prec$ holds. This containment property depends on the mechanism used for the element denoted by the second operand, and requires the consumer set to be a subset of any upstream consumer set for cover-shared and non-shared elements, and a superset in the case of union-shared elements.

The risk metric Δc is defined as the range of possible costs, $c(max) - c(min)$, associated with a given plan that can arise due to inaccuracies present in selectivity estimation. In order to compute this risk metric, we require upper and lower bounds on our selectivity estimates. These bounds are used in computing upstream selectivities in place of average selectivities, yielding cost bounds when combined when per-element costs. In the above formalization, we omit explicitly defining these cost bounds for each term in the cost metric and only present the definition for the upper bound on the cover-sharing algorithm ($c_{cov}(max)$). Note that these terms use the selectivity bounds $\sigma^{max}, \sigma^{min}$ described in Section 3.3.4.

4.5 Detecting Optimization Opportunities

Given the above processing cost model, our algorithm executes as a monitoring algorithm, where we periodically collect operator costs and input stream distributions over a time window, and subsequently evaluate a set of conditions that focus on specific parts of the cost model to determine query adaptations. We present the monitored conditions with the view of solving two problems, the first of which involves determining both the sharing algorithm to use for each compatible expression and the global set of queries considered as consumers, while the second finds a finer-grained grouping of queries for each expression and chosen algorithm, and creates shared instances and covering expressions for each resulting query group. We note that every adaptation described below must preserve the plan's dataflow integrity, specifically by ensuring a consumer set containment property holds between neighboring elements following a move. This consumer set containment property states that for every instance i of a shared operator o with consumer set $Q_{o,i}$, any shared downstream operator and its instances must have a consumer set that is a subset of $Q_{o,i}$ before and after any adaptation. This property ensures that no shared instance of any query operator is missing any inputs it would process in unshared fashion.

4.5.1 Shared Operator Algorithm and Expression Adaptation Conditions

The first set of monitoring conditions and adaptations we describe optimizes the algorithm used for each compatible expression and query capable of sharing the expression. The conditions for each expression and query pair depends upon the algorithm currently chosen

for the query's upstream and downstream neighbors (from the compatible operator). Thus we present our algorithm as a state machine, consisting of states defined on the shareable operator and neighbors' currently employed sharing mechanisms, and define transitions between states as detecting when one mechanism becomes cheaper than the other.

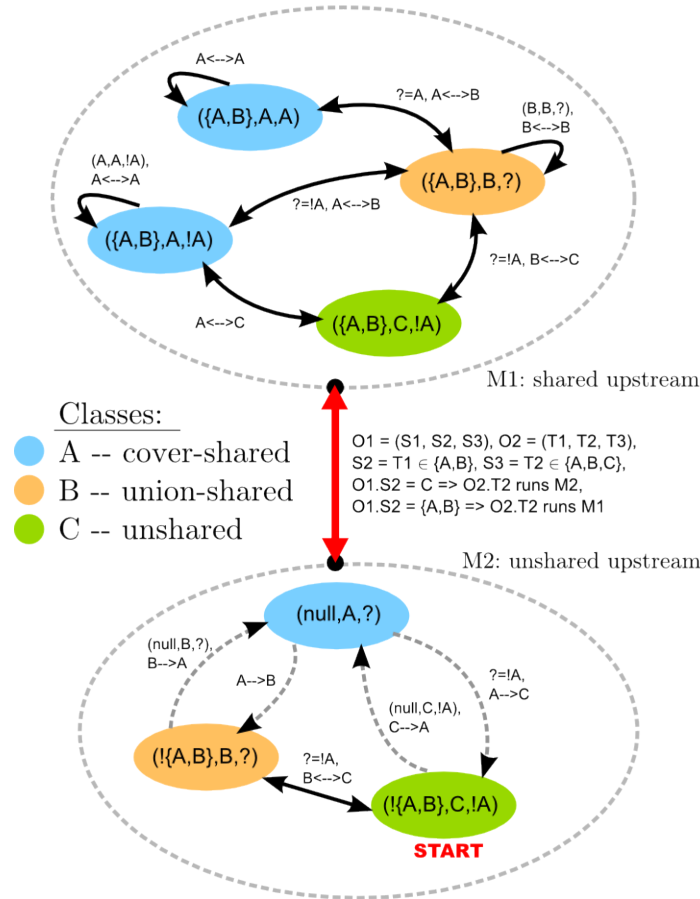

Figure 4.3: State transition diagram for selecting sharing algorithm type.

Specifically our algorithm uses two separate state machines, or *modes*, with the active mode depending on whether or not a shared mechanism is used for the upstream neighbor. The states and transitions of the two modes are represented pictorially in Fig. 4.3. In this diagram the triples representing the state label indicate the upstream operator's sharing state, the current operator's sharing state, and the downstream operator's sharing state. Note these states includes a wildcard pattern '?' indicating the state matches any algorithm type for the relevant neighbor, for example the state ({A,B},B,?) indicates that state applies for an operator currently implemented with union-sharing, whose upstream neighbor may

either be cover-shared or union-shared, and whose downstream neighbor can be implemented using any of the three algorithms. The critical aspects to note are that an operator can only use cover-sharing when any of its upstream operators are also shared (or has no upstream operators), and that the transitions monitored with an unshared upstream is simpler than with a shared upstream.

The general form of the conditions tested prior to applying any sharing mechanism transition shown in Fig. 4.3 is as follows. Consider a transition of the form $x \rightarrow y$, for a given expression and query q, where $x : \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, $y : \{\mathcal{A}, \mathcal{B}, \mathcal{C}\} - \{type(x)\}$, and also define X, Y, Z as the sets of queries implemented in the types corresponding to x, y and the remaining type. Then the cost prior to, and following the transition can be defined as:

$$
\begin{aligned}
&cond(x \rightarrow y) = \\
&\quad cost(X|Y, Z) - cost(X - \{q\}|Y \cup \{q\}, Z) + \\
&\quad cost(Y|X, Z) - cost(Y \cup \{q\}|X - \{q\}, Z) \\
&\quad cost(Z|X - \{q\}, Y \cup \{q\}) - cost(Z|X - \{q\}, Y \cup \{q\})
\end{aligned}
$$

Thus our condition is simply to determine whether or not the transition is advantageous, that is whether or not it reduces our optimization objective. This involves checking $cond(X \rightarrow Y) > 0$. To evaluate the above condition, we can use a delta form based on expanding the above terms with their definitions in the processing cost model. This delta form captures local changes for each algorithm type assignment of x, y. We omit this delta form here for readability purposes.

The cyclical transitions within a state (e.g. for the ({A,B}, A, !A), ({A,B}, A, A) and ({A,B}, B, ?) states in Fig. 4.3) indicate reordering transitions within the shared segment of the query plan. In particular, we consider reordering the element to which the state machine applies and its upstream neighbor. We return to these reordering conditions in the next section. Furthermore, note that for simplicity we omit the instance transitions described below in the next section. Instance transitions would add a cyclic arc at every state shown, and would involve allocating the query associated with the state machine to a different instance for executing the shared element.

The state machines shown describes the sharing mechanism used for a single query element. Each shareable element and query pair in our system evaluates these transition conditions to optimize the query element. Given that an element's mode is dependent upon the mechanism used for the upstream operator, whenever we apply a transition for a single

operator, we also update the active mode for any downstream neighbors. Furthermore, we check for applicable transitions for each expression and query in topological order, thereby ensuring that each state machine is in a stable active mode prior to applying transitions. We argue that by performing per-expression-query adaptations, our algorithm allows for a highly flexible configuration of multiple queries containing a sharing compatible expression. For example we are able to execute a shared expression using both cover-sharing and union-sharing mechanisms, with disjoint sets of queries using the two types of algorithms.

The algorithm described so far has the ability to adapt the sharing mechanism of an element to use a different, *existing*, shared instance. In its current form, our algorithm is unable to function from an initial state, where no shared instances execute and all elements are unshared. To support bootstrapping shared instances, we use an additional condition to monitor all unshared elements for a compatible expression, which when triggered transitions a set of unshared elements to execute through union-sharing, or cover-sharing if the upstream element also executes in shared fashion. Note that we view this condition as a completeness requirement, our optimizer should be able to handle any query plan as input. We plan to consider a separate initialization algorithm as part of future work to accelerating convergence, by determining a good initial plan to adapt.

4.5.2 Shared Instance Creation and Adaption Conditions

In the second level of sharing conditions, we define how we adapt instances of shared operators, and their associated consumer sets. These conditions apply at the granularity of instances and individual queries, and are instantiated once we have determined our initial plan. We now describe three conditions that are based on a simplification of the objective function to consider the contributions made by a single operator.

The first condition we describe considers the objective contribution of a single shared instance j, yielding the equation below for operator i. In this section we describe the approach for cover-shared instances. A similar method can be used for union-shared instances.

$$\sum_{q \in Q \wedge i \in q} \sigma^{cv}_{i,j} c^{cp}_{i,j} + c^{cv}_{i,j}$$

This equation states that the (independent) cost of an instance is given by the product of the covering operator's selectivity ($\sigma_{i,j}$) and the cost of the compensation operator for each member of the consumer set ($c^{cp}_{i,j}$), as well as the cost of the covering operator itself ($c^{cv}_{i,j}$). The shared instance j lowers the objective value when compared to the non-shared case

when the following property holds:

$$\sum_{q \in Q \wedge i \in q} \sigma^{cv}_{i,j} c^{cp}_{i,j} + c^{cv}_{i,j} < \sum_{q \in Q \wedge i \in q} c^{orig}_{i}$$
$$\Rightarrow \sigma^{cv}_{i,j} < \frac{\sum_{q \in Q \wedge i \in q} c^{orig}_{i} - c^{cv}_{i,j}}{\sum_{q \in Q \wedge i \in q} c^{cp}_{i,j}}$$

In this equation c^{orig}_{i} refers to per-segment processing cost of the original non-shared operator. The above equation yields a condition based on the selectivity of the covering operator ($\sigma^{cv}_{i,j}$), stating that it must be less than the term on the right-hand side for cover-sharing to be advantageous in comparison to non-shared execution. We explain the right-hand side as the difference between two terms: i) the ratio of the total original cost and compensation operator cost, and ii) the ratio of the covering operator and compensation operator costs. Note that this is relation to a given consumer set – if this condition does not hold, there is no benefit in using a shared instance for any of the consumers allocated to the instance. At this point we remove the instance from the system. We refer to this condition as the *lazy-instance* condition since it determines when to remove instances lazily, whenever they are no longer useful rather than when they are suboptimal.

Our second condition considers the contribution of a single consumer to a shared instance's objective value: $\sigma^{cv}_{i,j} c^{cp}_{i,j} + c^{cv}_{i,j}$. A consumer benefits from shared processing provided the following condition holds:

$$\sigma^{cv}_{i,j} c^{cp}_{i,j} + \frac{c^{cv}_{i,j}}{|Q_{i,j}|} < c^{orig}_{i} \Rightarrow \sigma^{cv}_{i,j} < \frac{c^{orig}_{i} - (c^{cv}_{i,j}/|Q_{i,j}|)}{c^{cp}_{i,j}}$$

where $Q_{i,j}$ is the consumer set for the instance j of operator i. This condition states that the covering operator's selectivity must be less than the ratio to the cost benefit from using a fraction of the covering operator instead of the original operator, and the compensation operator cost. If this condition does not hold, we attempt to find another instance where the operator can contribute positively. This requires estimating the selectivity change for both the instance we are switching from, and the instance we consider switching to. If there are no operators allocated to an instance following the switch, we remove that instance. Furthermore if there are no instances resulting in lower system cost, we create a new singleton instance with the operator as its only constituent. We refer to this condition as the *lazy-consumer* condition.

Unlike the lazy-instance and lazy-consumer conditions which attempt to ensure that all instance-consumer allocations provide some reduction to the objective over non-shared processing, our third condition focuses on aggressively minimizing the objective by ensuring that certain consumers are continuously reallocated to the instances that best reduce the objective. We refer to this condition as the *eager-switch* condition, and define its benefits with the following equation:

$$\Delta obj = \Delta inst_{target} + \Delta inst_{source} = (inst_{target}^{new} - inst_{target}^{old}) - (inst_{source}^{old} - inst_{source}^{new})$$

Above, we state that applying an eager-switch condition changes the objective function based on the cost changes of both the source instance and the target instance. The cost of an instance is defined given the processing cost model in Section 4.4, and can be simplified to one of $c_{cov}(est), c_{un}(est)$ depending on the sharing algorithm being used. Since this condition does not alter upstream operators, it can be rewritten in terms of the different covering predicate selectivities and compensation predicate costs and risks. We omit the derivation for readability purposes and simply present the condition as follows:

$$\begin{aligned}\Delta \sigma_1'^{cv} c_{1,sw}'^{cp}(I_{S,1}' + R_1') - \Delta \sigma_2^{cv} c_{2,sw}^{cp}(I_{S',2} + R_2) > \\ R_2 I_{S',2} + R_1 I_{S,1} - (R_2' I_{S',2} + R_1' I_{S,1}') + (R_2 c_2^{cv} - R_1' c_1'^{cv})\end{aligned}$$

where $1, 2$ indicate the source and target instances respectively, S, S' indicates the consumer set before and after the switch for the instance associated with the term, $I_{k,j}, I_{k,j}'$ the instance cost for the consumer set k and instance j before and after the switch respectively, and finally R_k, R_k' the risk for instance k also before and after the switch. For example the term $I_{S,1}'$ refers to the instance cost for the source instance before the switch, restricted to the consumer set for the source instance after the switch (i.e. with the transitioning consumer removed).

4.5.3 Reordering Conditions for Shared Operators

To the best of our knowledge, there has been limited work that investigates how to combine adaptive reordering techniques with adaptive sharing techniques. The application of both optimization techniques are not mutually compatible in their original forms, that is reordering cannot be applied regardless of the sharing configuration of the operator. The critical aspect turns out to be the concept of consumer sets, and the consumer set containment

property we describe must remain invariant throughout adaptations. Consider an instance j of operator i with consumer set $Q_{i,j}$ that is produces results for a downstream consumer set $Q_{i',j'}$ where according to consumer set containment, we have $Q_{i',j'} \subseteq Q_{i,j}$. Reordering instance j' to be evaluated prior to j would violate the consumer set containment property, unless we have a tighter property, where $Q_{i',j'} = Q_{i,j}$. This tighter property leads to our concept of chained instances in shared queries that are capable of supporting reordering because each element of the chain supports an identical consumer set.

We define A-chains and C-chains as sequences of elements that are implemented with a cover-sharing mechanism, or are non-shared. A-chains may be tree-structured, that is an instance with a given consumer set may feed multiple downstream instances that process disjoint subsets of consumers. Note that tree-structured C-chains indicate that the query developer has explicitly used a shared expression in the query plan. Reordering A-chains occurs at the instance granularity, and must maintain the plan structure outside the reordered elements. Thus any tree-structured A-chains must ensure the reachability relationship $\prec^*$ holds identically for any elements not involved in the reordering. This reachability preservation also applies for C-chains but naturally applies to the non-shared query elements as opposed to instances.

Our ordering conditions apply to sequential elements in either an A-chain or C-chain. These conditions simply compare the covering instance or non-shared operator's cost and selectivity ratio, ensuring that following property holds:

$$\frac{c_i}{1-\sigma_i} > \frac{c_j}{1-\sigma_j}$$

where i is the upstream element in the chain and j the downstream. Also c_i, c_j refer to either covering instance or non-shared processing cost, and $\sigma_i, sigma_j$ to either covering instance selectivity or non-shared selectivity depending on the type of chain under consideration. We note that reordering can be useful for enlarging the set of beneficial sharing configurations, by allowing compensation operations to be more costly. Finally, union-shared elements cannot be trivially reordered due to the union performed and the subsequent arbitrary consumer sets chosen by the covering instances. We consider the problem of reassigning consumer sets upstream from union-shared elements outside the scope of this work, and thus do not support their reordering.

4.6 Collecting Query Statistics

In addition to using a pure estimation-oriented approach to computing selectivities, Pulse is capable of leveraging profiling and instrumentation techniques of actual executing queries, for example in scenarios where the presence of data dependencies results in poor estimates, and thus can improve selectivity estimates of any operations processing these attributes downstream from the instrumentation point. Our query profiling mechanism focuses on the issue of determining which internal streams to instrument for gathering attribute distribution statistics. Our strategy is to tie this decision in with the query optimization problem, enabling us to balance the benefits attained from query optimization with the additional overhead arising from instrumentation. At a high level, our solution focuses on finding and instrumenting points within the query that are expensive, and difficult to model.

4.6.1 Cost model and collection mechanism

We capture the effects of gathering statistics in our cost model by considering a substitution function ϕ, which replaces any estimated selectivities with measured selectivities:

$$\phi(c_{cov}(est), L) = \sum_{(e,Q)\in A} \sum_{(inst,S)\in inst(e)} \phi_\sigma(\sigma^{est}_{up}(e, inst), L) \times (c_{cv} + \sum_{q\in S} c_{cp})$$

$$\phi_\sigma(\sigma^{est}_{up}(e, o), L) = \sum_{m_1\in in(b_1)\wedge q(m_1)\in q(b_1)} \cdots \sum_{m_k\in in(b_k)\wedge q(m_k)\in q(b_k)}$$

$$\prod_{inst_{i,a}\in U_A} (1 - l_{i,a})\sigma^{est}_{inst_{i,a}} + l_{i,a}\sigma_{inst_{i,a}}$$

$$\prod_{(i,inst_{i,b})\in U_B} (1 - l_{i,b})\sigma^{est}_{dup_i}\sigma^{est}_{inst_{i,b}} + l_{i,b}\sigma_{dup_i}\sigma_{inst_{i,b}}$$

$$\prod_{i\in U_C} (1 - l_i)\sigma^{est}_i + l_i\sigma_i$$

$$\text{where } l_{i,a} = \begin{cases} 1 \text{ if } ((i, a), \sigma_{inst_{i,a}}) \in L_A \\ 0 \text{ otherwise} \end{cases}$$

$$\text{where } l_{i,b} = \begin{cases} 1 \text{ if } ((i, b), \sigma_{dup_i}, \sigma_{inst_{i,b}}) \in L_B \\ 0 \text{ otherwise} \end{cases}$$

$$\text{where } l_i = \begin{cases} 1 \text{ if } (i, \sigma_i) \in L_C \\ 0 \text{ otherwise} \end{cases}$$

$$\text{and } \sigma_i = \text{measured selectivity for operator } i.$$

To summarize the above changes to the cost model, we first describe the sets L_A, L_B, L_C. These represent the selectivities collected for those instances implemented in cover-shared, union-shared and non-shared fashion respectively. For a cover-shared operator i, we have the covering instance's selectivity $\sigma_{inst_{i,a}}$. For a union-shared operator i, we collect the selectivity of the duplicate elimination operator σ_{dup_i} and the instance $\sigma_{inst_{i,b}}$. Finally for a non-shared operator i we simply collect that operator's selectivity σ_i. Note that we view these measured selectivities as averages collected over a time window. These measured selectivities have limited use, namely that they do not aid in the derivation of histograms downstream from the instrumentation point, and hence only impact downstream path selectivities through the change at the collection point.

Next, we can define 0-1 decision variables $l_{i,a}, l_{i,b}, l_i$ indicating if we collect selectivities for a particular operator and instance. We define the set L as the set of these decision variables. These decision variables are primarily used in the definition of path selectivities, in the term $\phi_\sigma(\sigma^{est}_{up}(e, o), L)$ above. In this term, we compute an upstream selectivity for an instance o of the operator e as the products of individual path selectivities, summed

over the various paths arising due to the presence of union-sharing (given the union-shared operators $b_1 \ldots b_k$). Here our measured selectivities replace the estimated selectivities for the instances or non-shared operators present in any path. We derive similar definitions of c_{un}, c_{ns}, $c(max)$, $c(min)$, σ_{up}^{min}, σ_{up}^{max} with similar substitutions of measured selectivities but omit here for readability.

As we stated, L represents the set of measured selectivities. We can then incorporate the function ϕ into a revised query optimization problem:

$$\begin{aligned}
&cost = \min_L(\phi(c, L) + c_{collect})\rho\phi(\Delta c, L) \\
&\text{where } \phi(\Delta c, L) = \phi(c(max), L) - \phi(c(min), L) \\
&\text{and } c_{collect} = \text{instrumentation cost}
\end{aligned}$$

4.6.2 Adaptation conditions

We can derive the benefit of collecting a set of statistics L in comparison to not collecting any statistics as:

$$\begin{aligned}
\Delta cost \quad &= c\rho\Delta c - (\phi(c) + c_{collect})\rho\phi(\Delta c) \\
&= \rho(c\Delta c - \phi(c)\phi(\Delta c) - c_{collect}\phi(\Delta c))
\end{aligned}$$

Due to the comparison to a query plan with no collected statistics, we refer to the above benefit as the independent benefit. This independent comparison explicitly simplifies the candidate instrumentation sets considered for collection, albeit at the expense of suboptimal instrumentation.

Now, the above formulation represents the benefit of collecting statistics, but does not indicate how to pick the set of operators L to instrument. One key issue here is that during the selection of the set L, we do not actually have measured selectivities requiring us to make assumptions on the impact of such measurements on the estimated selectivities. We adopt a simple approach where we assume our estimated selectivities are unchanged after gathering statistics, yielding $\hat{\phi}(c) = c$. This assumption allows us to simplify the goal of gathering statistics to that of minimizing the risk involved in the chosen plan, and we can now rewrite the independent benefit of instrumentation to our objective as:

$$\Delta\widehat{cost} = \rho(c\Delta c - \hat{\phi}(c)\hat{\phi}(\Delta c) - c_{collect}\hat{\phi}(\Delta c))$$
$$= \rho(c\Delta\hat{\phi} - c_{collect}\hat{\phi}(\Delta c)) \qquad \text{where}\Delta\phi = \Delta c - \phi(\Delta c)$$

Clearly, we would like to maximize the above benefit, and observe that at a local optimum we are unable to make further adaptations, represented as $\Delta\widehat{cost} < 0$. This condition allows us to derive the following property that any candidate for the set L should meet:

$$\rho\left(c\Delta\hat{\phi} - c_{collect}\hat{\phi}(\Delta c)\right) > 0 \Rightarrow \frac{c_{collect}}{c} < \frac{\Delta\hat{\phi}}{\hat{\phi}(\Delta c)}$$

The above property is based on comparing total query costs rather than at a sub-query granularity that is amenable to continuous monitoring to drive adaptation. Assuming constant collection costs, the left-hand side of the above relationship is a constant in the presence of a fixed query plan. Note that we could ask the question what is the best query plan that facilitates effective statistics collection, but we consider this issue beyond the scope of this work.

Using the fact that we have derived a model of independent statistics collection where the effects of statistics collection are compared to a query without any feedback, we can derive a sub-query granularity condition by simply evaluating the right-hand term of the above equation for various query elements (i.e. varying L), and comparing them to our collection cost threshold. Computing the right-hand term on a per-element level still requires computing selectivity products for the dataflow paths present in the query plan for each element considered for statistics collection. We enable our conditions to reuse partial selectivity products computed while varying L by memoizing these computations.

4.7 Condition Evaluation

To this point, we have described our motivation for designing and evaluating a set of monitoring conditions on certain query properties, and applying a variety of localized query adaptations following their satisfaction. These conditions may clearly affect each other, since they alter the query structure and affect the metrics being monitored. To conclude this section we describe an evaluation order that can handle these dependencies between conditions by invalidating any dependent conditions prior to applying any query modifications.

We state that the dependency ordering for a single operator is as follows, and then explain how we obtain this:

$$\text{sharing algorithm} \prec \text{lazy-instance} \prec \text{lazy-consumer} \prec \text{eager-switch} \prec \text{feedback}$$

Above the $\prec$ relationship states that the left-hand condition should be evaluated prior the right-hand condition due to dependencies in the cost components used in the right-hand condition. Note that any triggered conditions result in removal of any dependent conditions for that optimization period. This is a simplifying, conservative evaluation, since in certain cases these dependent conditions could be updated, but note that the evaluation of these dependent conditions may also require significant recomputation of selectivity estimates. Our sharing algorithm condition has all other condition types for the same operator and instance as its dependents. The structural changes caused by switching types, such as adding or removing plan elements for union-sharing, directly affects the localized cost model acting as the foundation for the other conditions. The lazy-instance conditions determine when to remove an instance, causing an lazy-consumer or eager-switch conditions for the consumer set associated with that instance to become invalid. Finally all of the aforementioned conditions that result in structural changes in the global query plan may invalidate any feedback selection conditions for removed elements.

Finally recall that for a given operator or shared instance, the state machine driving our sharing algorithm conditions matched against the upstream sharing mode, the monitored sharing mode, and the downstream sharing mode. Thus changes in a given operator's sharing mode results in resetting the sharing algorithm conditions for these neighbors. However we only update the neighbors' sharing algorithm conditions and not any of their dependents, since we do not actually change the state of the neighbors. Thus we can summarize our condition evaluation order in two stages: i) a topological traversal of the multi-query plan, and at each operator traversed ii) an evaluation of operator-specific conditions based on the above condition dependency order.

4.8 Experimental Evaluation

We have also implemented our optimizer in Pulse, using Borealis as the underlying stream processing engine. Our evaluation was performed on the same system setup as the selectivity estimator, namely on a Intel Core 2 Duo T7700 2.4 GHz processor, 4GB RAM, running a Linux version 2.6.24 kernel, with the summer 2008 release of Borealis. We now examine the benefits of our optimization mechanism, comparing the cover sharing mechanism to unshared query execution, when used to process queries of varying selectivities. The queries

considered here are a set of filters applying a threshold to stock prices in the NYSE dataset, resulting in covering and compensation operators that are also filters. We performed these experiments in batch fashion, that is the client enqueues all segments to be processed into the system and we record the duration of execution until the operator's queues have been emptied.

Figure 4.4 displays the total processing time for the NYSE dataset for a single stock symbol against the number of operators (identical to the number of queries) being processed. These operators have thresholds chosen randomly from a uniform distribution whose upper bound is indicated on each plot. These upper bounds correspond to thresholds yielding query selectivities in the range 0.1 to 1. Given our particular query workload, these upper bounds approximately correspond to the threshold used by the covering predicate, and thus the selectivity of the covering predicate. The lower bound is set to the minimal value such that all segments in the dataset are pruned by the bound. Figures 4.4i)-v), which correspond to upper bound selectivities of 0.1-0.5 all exhibit cases where a cover-shared execution outperforms the unshared equivalent. Beyond such selectivities, as seen in Figures 4.4vi)-x), cover-sharing does not turn out to be advantageous to unshared execution, primarily due to the additional cost of the covering operator, and its insufficient selectivity to overcome this additional cost in terms of executing the compensation operators. We also observe that at lower upper bound selectivities, the processing cost for cover-sharing tends to have a much shallower gradient with increasing numbers of operators, with Figure 4.4i) having an almost constant processing cost. This arises due to the fact that the covering operator dominates the total processing cost, and the compensation operators, which is the only part of the shared query that grows with the number of consumers, consumes a small fraction of processing resources. We briefly remark on the volatility seen in these experiments. In addition to the unpredictability arising from executing in an actual stream-processing prototype, there are additional sources of noise arising from our instrumentation technique. In particular, we remark that determining empty queues inside the engine requires gathering statistics at the client, and in its current form in Borealis, this statistics collection method proves to incur non-trivial overhead (approximately 200ms per probe). This limits the rate at which we can poll for empty queues, which when combined with our desire not to overburden the engine with continual polling, results in a precision of approximately 250ms in determining the time instant at which queues are empty.

Figure 4.5 is a pivoted plot of the previous figure, whereby we compare the difference in processing costs between the unshared and cover-shared execution mechanisms against the upper bound selectivity of the query workload. Each line in the figure represents the

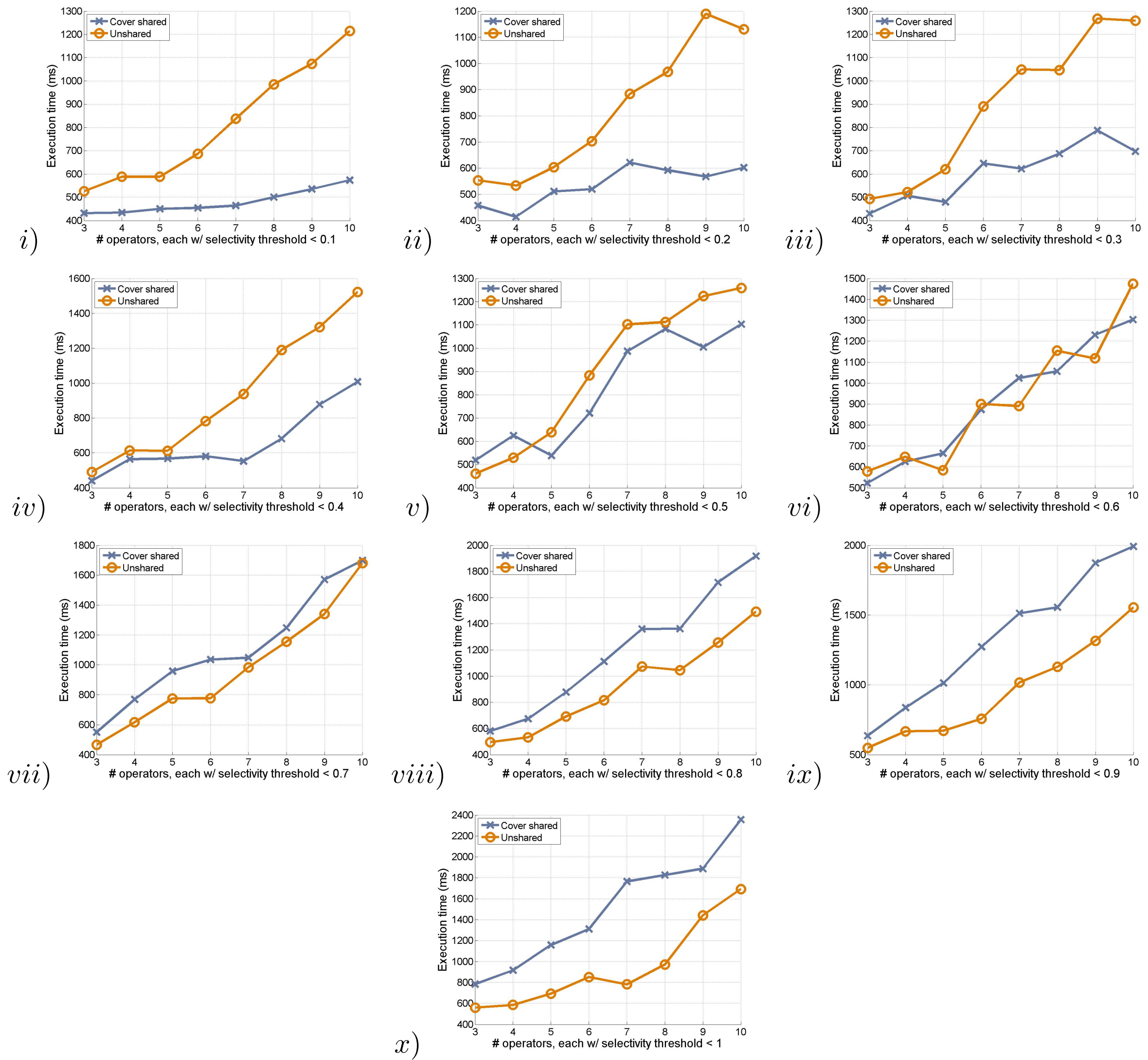

Figure 4.4: Comparison of cover-shared and unshared execution at a variety of upper bounds for query selectivities.

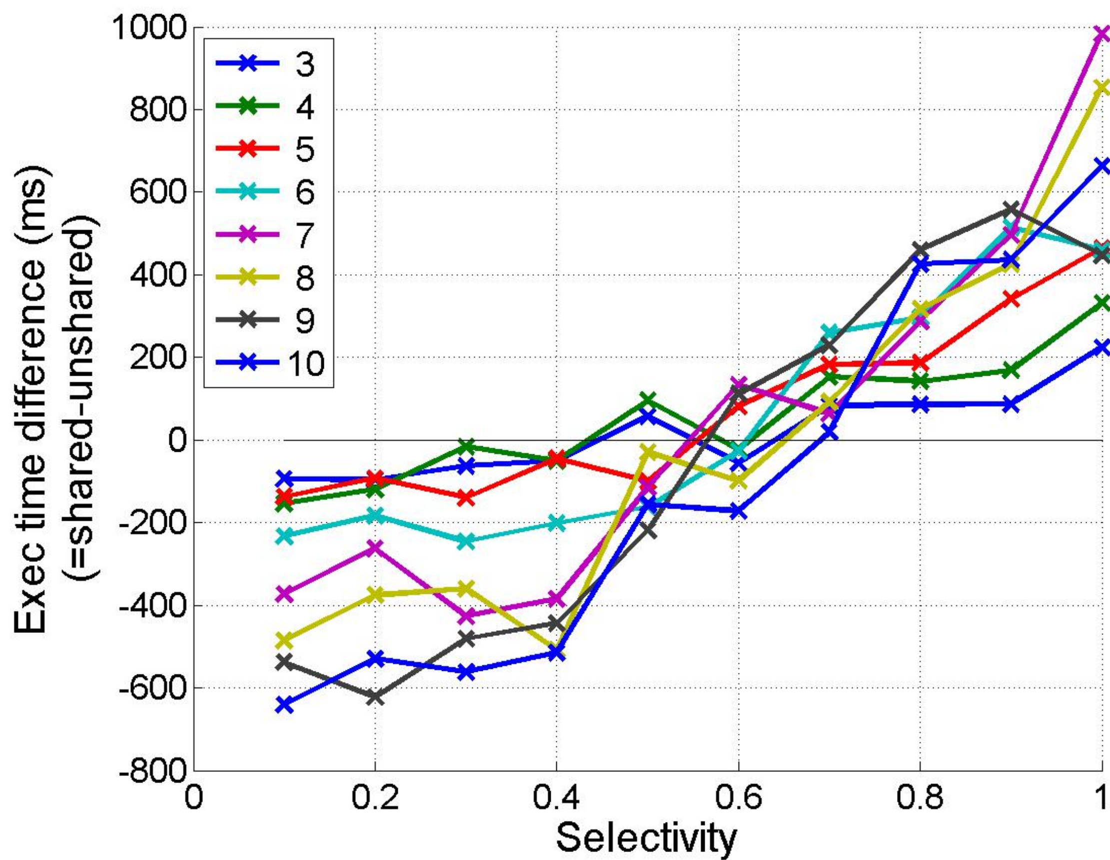

Figure 4.5: Processing cost differences between unshared and cover-shared execution for query workloads with a variety of covering selectivities.

Selectivity configuration for operator B	Upstream sharing mode	Selectivity configuration for operator A		
		High	Spread	Low
High	Cover	34.27	23.82	1.32
	Union	35.41	24.67	15.07
	Unshared	30.73	19.26	13.87
Spread	Cover	34.72	29.2	1.57
	Union	35.64	30.94	14.46
	Unshared	29.24	24.86	13.73
Low	Cover	3.05	3.14	1.45
	Union	20.14	19.08	14.52
	Unshared	28.7	21.42	14.37

Figure 4.6: Processing cost comparison for union-sharing and other mechanisms in a 2-operator chain, with varying selectivity configurations at each operator. Note here high selectivity configuration implies selectivities close to 0.0, spread selectivities indicates randomly chosen selectivities in [0.0, 1.0], and low refers to selectivities near a value of 1.0. Additionally the upstream operator, A, is cover-shared, and for this operator, the covering selectivity is roughly equivalent for high and spread selectivity configurations.

processing cost for a specific number of operators. The aim of this plot is to illustrate sensitivity to the upper bound selectivity, and thus covering selectivity. There are several trends to note here. First that the crossover selectivity, that is the selectivity beyond which unshared execution is cheaper than cover-shared execution, is approximately 0.6 for any number of operators. Based on the cover-shared cost model, this crossover selectivity should be dependent on the number of consumers using a shared operator, where larger numbers of shared operators should incur a higher crossover selectivity. In practice this crossover selectivity turns out to be lower as seen in our experiments due to the increasing system overheads of executing more complex queries, including scheduling and additional queuing arising from a longer pipeline of operators. Next, the range of processing cost differences over the selectivities show here is tends to be greater for a large number of consumers, than for fewer consumers. This arises due to the cumulative benefits and costs of dropping inputs to or processing inputs for increasing numbers of compensation operators.

Finally, Figure 4.6 displays a comparison of the processing costs for the union-sharing technique to both unshared and cover-shared execution of 10 shared queries, where each query is simply two filter operators connected as a chain. In this query, the upstream operator is shared in cover-sharing fashion, while the downstream operator is implemented using the technique stated in the table. Furthermore, we vary the predicates present in

both operators according to three simple configurations, first a high selectivity configuration where predicates in all 10 queries are close to 1.0, spread selectivity configuration where predicates have selectivities chosen uniformly randomly from [0.0, 1.0], and finally low selectivity where all predicates have selectivity close to 0.0. The results in Figure 4.6 show that in any situation where either operator A or B (i.e. the first or second shareable operator in the chain respectively) have a low selectivity configuration, cover-sharing dominates the other techniques, due to the fact that the covering predicate also has low selectivity.

Note that whenever both of the operators have either a high or spread selectivity configuration, unshared execution tends to perform better than both cover- and union-sharing. For cover-sharing, this is due to the corresponding high selectivity of the covering predicate (i.e. the covering predicate's selectivity will also be close to 1.0 in the case of a spread configuration), indicating that the covering operator is unable to eliminate common unnecessary inputs for the compensation operators for B. However in the case of union-sharing, this result arises from the relatively high cost of performing the union and duplicate elimination operations with respect to the entire query processing cost. We expect that as the cost of the downstream query increases relative to these union and duplicate elimination operations, union-sharing will outperform cover-sharing. Finally in the case where operator B has low selectivity and where operator A has high or spread selectivity, we see that while cover-sharing is the dominant technique, union-sharing significantly outperforms unshared execution, that is it offers a processing technique that need not require wholescale changes to the query plan, yet can still provide lower processing cost than either extreme of plan.

Chapter 5

Related Work

This thesis has been inspired by many pieces of work in the database community, ranging from the stream processing engines as the context of our own query processor, moving-object databases and constraint databases which have both used polynomial representations of data, and selectivity estimation and multi-query optimization techniques for traditional databases. Of particular note is the recent work on model-based databases, where we have contributed query processing techniques over attributes modeled as polynomials. We also discuss select mathematical software and computer algebra systems which provide the mathematical expressiveness we would like to realize.

5.1 Data Stream Management Systems

There has been much recent interest in performing push-based query processing over a sequence of continuously arriving input data. Stream processing engines have emerged as a database architecture to support continuous queries over these data streams with the key differences from traditional architectures including straight-through processing and temporal constructs such as windows and punctuations. Much of the initial work focused on architectural challenges in dealing with high-volume inputs, for example load shedding techniques, and load management. We now briefly describe some of the major efforts in this field primarily to provide background to the platform on which we've built our own query processor and to reaffirm the advantages of model-based processing for stream applications.

Aurora, Borealis. The Aurora and Borealis [2, 1] projects are two stream processing engines built by Brandeis, Brown and MIT to demonstrate the functionality and benefits of

stream processing engines in a single server and distributed configurations respectively. Aurora is a general purpose stream processor that implemented queries using a dataflow model and provided several relational-style operators for users to express their queries. These operators include the filters, maps, joins and aggregates we have considered for relational processing in Pulse. The Borealis stream processor extended Aurora with key distributed execution primitives, for example the ability to migrate operators across multiple hosts and a general set of dynamic query modification operations. In turn these facilitated novel distributed algorithms for load management [94, 10], distributed load shedding [46, 9] and high availability in the face of node and network failures. Pulse is built on top of the Borealis stream processing engine as a custom dataflow graph and as such, is able to transparently take advantage of the distributed execution supported by Borealis, including for example distributing the solver load across multiple machines. Furthermore Pulse uses models to significantly improve query processing performance by using a compact approximate data representation, which can be considered an alternative technique to reducing overheads by shedding load and providing a subset semantics in the results produced. The key point here is that a numerically approximate technique provides a different result model than the subset semantics, which may be more suitable for the continuous numerical attributes in the domains that we target.

Stanford Data Stream Manager (STREAM). The STREAM [65] project from the Stanford Database Group designed a stream processing engine that used a text-based query language, known as Continuous Query Language (CQL), which included stream-oriented extensions such as windows to SQL. In addition STREAM included a relational-style query processor, an adaptive query optimizer with techniques such as adaptive reordering, and approximate query processing with sampling and summaries. Another key contribution of the STREAM project included memory management techniques through a variety of scheduling policies, shared processing and exploiting order constraints present in data streams.

As a predecessor to the work on the STREAM project, the TRAPP project [70] investigated filtering techniques in a streaming environment. Stream filtering mechanisms [69, 48] apply delta processing techniques to perform stateful query processing by establishing a bound (or predictive model) between a data source and a data processor. This allows the source to avoid sending updates, providing these updates adhere to the predictive model. These techniques are similar to the functionality provided by our accuracy splitter component, but do not consider processing issues for whole queries or computing with data models.

TelegraphCQ. The TelegraphCQ stream processing engine [18] from the UC Berkeley Database Group presented the design of a stream processing engine on top of the Postgres RDBMS, which, in contrast to the from-scratch designs of the Aurora and STREAM projects, heavily utilized existing components of a database system to implement continuous query processing. Some of the novel features of TelegraphCQ included its adaptive query processing capabilities, for example with the use of Eddies [6] that dynamically route tuples to query plan operators, thus allowing a per-tuple operator ordering. Also the project investigated query processing strategies for hybrid queries combining streamed and archived data [19], for example using approximations of archived data to limited the cost of disk-based access while performing continuous query processing [20].

These three stream processing engines provide just a sample of the recent work on the topic, and other notable systems include NiagaraCQ [24], Gigascope [27], StatStream [98], Nile [44], System S [37]. Additionally complex event processing systems have extended the relational approach to provide support for pattern oriented queries, including SASE+ [93], and Cayuga [30] amongst several other systems.

5.2 Databases for Sensor Data Processing

Given the wide range of potential applications of sensor networks, there has been much recent interest in understanding how to best provide data management tools to query sensor data. In particular, we focus on the topic of model-based databases in this section. This topic is emerging as an umbrella for the many pieces of related works which look at mathematical representations of sensor data. We limit ourselves to consider those works relating to using piecewise polynomials as models, and briefly describe works using time series as well. The time series literature is significantly more extensive in terms of considering queries outside the relational algebra, and additionally there have been several works on other types of mathematical models that we omit here. To the best of our knowledge, Pulse is the first framework to process continuous queries directly on piecewise polynomials with simultaneous equation systems.

5.2.1 Model-Based Query Processing

Deshpande and Madden [32] present the MauveDB system which is capable of using user-defined interpolation and regression functions to provide regular gridded access to the raw data as a materialized view in the database system. The authors advocate the use of standard query processing techniques on top of this gridded view, and present efficient

techniques to maintain this view upon changes to the underlying interpolation functions. In contrast, rather than maintaining a gridded view using our polynomials and then applying standard relational operators to this view, we attempt to maintain a polynomial representation throughout the query processing pipeline, treating polynomials as first-class entities.

5.2.2 Querying Regression Functions

Neugebauer [66], and Lin and Risch [59] present interpolation techniques for base relations in relational databases. Neugebauer presents techniques for optimizing embedded interpolation functions that are essentially black-box interpolation functions in the query plan. Following the construction of an interpolated relation via these embedded functions, query processing proceeds using standard processing techniques. This is similar in principle to the MauveDB approach described above. In contrast Pulse proposes an alternative approach to query processing that can delay the interpolation later in the plan, providing significant efficiency gains. Lin and Risch describe a novel selection operator that is assigned the responsibility of performing the interpolation, as well as an indexing technique over segments to support such an operator. Their work does not discuss issues of delaying polynomial evaluation with respect to other operators as in Pulse, and we consider the issue of indexing orthogonal to the work presented here. There are a variety of indexing techniques that Pulse could leverage, but we also remark that indexing may be less beneficial than in traditional database scenarios due to the stream context we consider, and the typical high index update costs.

More recently FunctionDB [92] investigated the use of multivariate polynomials to represent schema attributes, and presented both an equation solving approach and an approximation algorithm based on adapting a rasterization algorithm to effectively perform hypercube space-filling of constraint solution regions. FunctionDB is the closest work to our own, and supports a more general polynomial datatype than that presented in Pulse. However it does not address the question of handling user-specified errors and leveraging these throughout the query plan for trading off query accuracy and performance.

5.2.3 Model-Driven Query Processing in Probabilistic Databases

The BBQ system [31] uses multivariate Gaussians to represent a joint probability distribution over several attributes and supports a variety of probabilistic queries on such attributes. The queries considered include range queries, point queries, and average aggregate queries.

Each of these query types either use a confidence interval and determines attribute values corresponding to the interval, or yields a probability value corresponding to the desired attribute values.

The issue of handling uncertain or probabilistic data can be considered orthogonal to our work, indeed there have been several pieces of recent work looking to design a data and query model for uncertain and probabilistic databases [28, 5, 11]. It is clear that models can play an important role in this context given the abundance of statistical and probabilistic techniques of representing data which use well-defined mathematical forms of probability distributions such as Gaussian, Poisson, exponential and Levy distributions amongst many others.

5.2.4 Querying Time-Series Models

Time series databases primarily focus on basic operations on a time-series datatype, and consider datatype-specific queries such as similarity search and subsequence matching [35], in addition to mining and analysis related queries [71, 83]. Time-series modeling techniques include various segmentation algorithms used to capture time series as piecewise linear models [82, 53]. Indeed, these approaches can be used internally by Pulse to construct piecewise polynomials, given a standard data stream consisting of discrete tuples. However, the query functionality typically considered in these mining and analysis works represents a disjoint set of functionality from relational algebra and does not address the issue of how to leverage temporal continuity in high-volume relational data stream processing.

5.3 Moving-object Databases

Moving object databases frequently leverage continuous-time models in the form of object trajectories during spatio-temporal query processing. Here, query types commonly include range search, range aggregation, spatial join [61] and nearest neighbor queries [96], and are often implemented through a specialized index structure for each query type. A survey of these indexes and access methods may be found in [64]. Tao et. al [85] exploit predictive functionality to trade off result accuracy for communication cost for range queries. In a stream processing context, the PLACE project [63] studies multi-query optimizations via sharing, and the role of uncertainty in spatio-temporal queries.

5.3.1 Moving-object Indexes

There are several works addressing the challenge of designing appropriate indexing techniques for trajectory data. The STRIPES index [73] maintains trajectories in the dual space, by using a multidimensional quadtree defined on this dual space. Furthermore, to handle the time dimension, the authors require that the objects issue periodic updates which in turn is used to define the lifetime of the index structure. Thus STRIPES maintains a small number of indexes based on which objects have recently issued updates. The PA-Tree [68] considers historical query processing of trajectory segments. This index uses polynomials to approximate multiple segments, while tracking the maximum error in the approximation. The authors then build the PA-Tree as a two-tiered parameter (i.e. dual) space index over the coefficients of the polynomial approximations, with first tier providing a low-dimensional approximation that can still be effectively indexed with R-trees. There are numerous other works on trajectory indexing such as the MVR-Tree, TPR*-Tree [87], SETI [17], etc. whose features are compared in the two works mentioned.

5.4 Constraint Databases

Constraint databases have provided inspiration for representing high-volume data streams in a compact way, and for our formulation of continuous queries as linear systems. The constraint database model and query language is introduced in [52]. The related literature has considered semi-algebraic constraints, demonstrating that such constraints are not closed under certain aggregation operators. By limiting our polynomials to those with positive integer powers, we ensure that we handle such aggregates in a closed manner.

5.4.1 Dedale

The DEDALE [41] project describes the implementation of a constraint database system that operates on infinite sets of tuples, simplifying tuples via normalization before processing via a constraint engine. DEDALE has also been applied to the problem of processing queries over interpolated data [42]. However, constraint databases have enjoyed limited popularity and use in today's database systems. We believe the reasons for this include the lack of extensive optimization techniques and cost models to aid in performing under-the-hood query optimization. In comparison to Pulse, constraint databases do not address stream applications and their high volumes of updates. Constraint databases were primarily considered for use in spatial applications whereas we are targeting finance, science and

engineering applications.

5.4.2 Optimization queries

While not strictly falling under the topic of constraint databases, we also briefly describe related work by Gibas et. al. [39] on optimization queries in the context of relational databases. In this work, the authors consider queries as an objective function and constraints placed on attributes present in a relational database, that is the database contains the set of possible solutions to the optimization problem. One class of objective functions considered is that of a polynomial objective. The authors describe a query processing algorithm that intelligently traverses an index based on the given query, and analyze the I/O performance of their algorithm. In Pulse the constraints themselves are polynomials, and our work focuses on the throughput of the system rather than I/O characteristics. Additionally our query processing mechanism is based on equation systems rather than pruning a search space through the use of solution bounds present in internal nodes in index structures.

5.5 Approximate Query Processing

The related literature on approximate query processing techniques is pertinent to the context of our work given our view that piecewise polynomials provide an approximation to the input data stream, and our desire to manage the propagation of errors during query processing to uphold user-specified precision bounds. The literature on approximate query processing is extensive, and we focus on more recent work addressing approximation for stream processing.

5.5.1 Stream Filtering

Stream filtering is used to describe a delta processing technique, typically executing in a distributed fashion, whereby a server maintains a set of objects, each according to a user-specified precision bound. Data sources selectively send updates to these objects to ensure the precision bound is met, while leveraging the ability to only send a small portion of the full set of updates for performance gains, including communication and processing overheads.

Olston and Widom present TRAPP [70] which investigates stream filtering for aggregation queries over a set of data objects, presenting an optimal algorithm to choose the update tuples for min and max aggregates without selection, and approximation algorithms sums

and aggregates with selections. The work also discusses some of the challenges involved with maintaining join-aggregate queries. Olston, Jiang and Widom [69] further investigate stream filtering addressing the question of how to handle multiple aggregate queries placed over different, but overlapping, sets of objects. They demonstrate the inefficiency of a simple uniform allocation strategy, presenting an algorithm that selects update rates for each data source based on an optimization problem that can be represented and solved using system of linear equations.

Jain et. al. [48] present a stream filtering algorithm that uses Kalman filters in a predictive approach to stream filtering. Here both the database server and data sources maintain a Kalman filter that is capable of providing predictions to a query processor to produce results for the user. The Kalman filter at the data source mirrors that at the server, so that it may reliably determine the effects of sending updates to the server, in terms of the accuracy of the predictions generated. These updates are then communicated whenever prediction accuracy drops below a user-specified precision bound. Unlike Pulse, this work applies standard query processing techniques following the generation of predictions from the Kalman filter, and does not consider how the relational algebra can be applied directly to the mathematical representation of Kalman filters. Cheng et. al. [25] investigate stream filtering in the context of entity-based queries (as opposed to numerical value-based queries which are the norm in the literature), considering non-value based errors such as rank-based errors and metrics defined in terms the fraction of false positive and negative query results. In summary their work on rank-based errors relies on maintaining a buffer of entities that exceeds the desired limit, which can be used to yield a ranked result set with error bounded by the buffer size. Updates can then be communicated to ensure the buffer contains the appropriate entities. Pulse is unable to support these types of queries and focuses primarily on continuous numerical attributes through its choice of polynomials as a model. Entity-based attributes require alternative models focusing on categorical data, such as classifiers including SVMs, decision trees or neural networks. Shah and Ramamithram [81] present a stream filtering technique assuming a query model of polynomial queries. Polynomial queries compute an arithmetic expression over a set of data items (variables) using a polynomial function. The authors argue that stream filtering systems should be concerned with both the number of updates communicated and the number of bound computations. They present an algorithm that uses a secondary (larger) bound at data sources to determine when a primary bound is valid at the server. In certain specialized cases, Pulse would be able to leverage this technique, however its goal is one of generality and Pulse is capable of handling relational queries rather than those based on specific arithmetic structure. In

essence Pulse's queries are sequences of polynomial queries, with constraints, and each stage of the query processing would be able to leverage techniques presented here.

5.5.2 Approximation in Stream Processing Engines

Investigating approximation techniques in stream processing engines has been a popular topic due to the high volumes of data that must be processed in stream applications, leading to heavily overloaded systems where a natural approach is to consider shedding load, and approximating query results.

The problem of load shedding in data stream processing, in simple terms, refers to the question of how to selectively eliminate tuples from the input workload given that this workload causes overload in the system. Tatbul et. al. [90] investigate random and semantic approaches to this problem in the Aurora stream processing engine. This work is further extended to handle nested aggregations [91] by applying load shedding to whole windows rather than individual tuples, and also to the distributed case [89], to achieve both processing and network overhead reductions, similar to stream filtering techniques described above. Other examples of load shedding include the work by Babcock et. al. [7] in the context of the Stanford STREAM project, and Reiss and Hellerstein [76] in the TelegraphCQ project.

Das et. al. [29] present a study of error metrics for sliding window join algorithms, acting as proponents for an archive metric which indicates the quality of a join result in terms of the amount of work required to finish an incomplete join (and thus has to be archived for processing later). This leads to the use of a MAX-subset metric which attempts to defer processing for tuples so as to ensure the join result contains as many result tuples as possible. The authors present several heuristics to obtain a MAX-subset result, as well as discussing the hardness of the problem.

As an example of another popular subtopic in approximate stream processing, Dobra et. al. [33] describe the use of pseudo-random sketch summaries of a data stream for processing aggregate queries. Sketches are essentially a compressed form of a model, and these works describe how aggregates can be computed directly from this model. More recent work has expanded the use of sketching techniques to a variety of other query types including join-aggregate queries [78], as well as for use in distributed settings [26]. Typically these works consider sum, count and average aggregates, and the authors derive the error bounds in their approximation through an analysis of the specifics of their sketching technique. Despite the improved accuracy of approximation with this limited set of queries, Pulse's

goal is that of generality, hence its choice of using polynomials which can be used in min and max aggregates, as well as predicates. One interesting topic of future research would be creating a hybrid query processor that is capable of leveraging multiple kinds of models, for example these sketches to handle the relevant aggregation portion of the query.

5.6 Selectivity Estimation

There are many related works on selectivity estimation techniques, and we now present a brief overview of those that inspired our own approaches. Earlier works on selectivity estimation assumed independence between a relation's attributes. Recently these approaches have been found to provide underestimates to actual selectivities in the presence of correlations and dependencies between the attributes. In the second subsection we describe works targeted at addressing this problem.

5.6.1 Selectivity Estimation for Intermediate Result Cardinalities

Poosala et. al [75] present a variety of histogram techniques for estimating selectivities, comparing the various histogram algorithms in terms of several metrics including storage costs, and construction and maintenance costs. The takeaways from this work include the advantageous use of sampling techniques during histogram construction, and the argument for the use of the MaxDiff histogram as the standard. Matias, Vitter and Wang [62] present wavelet-based histograms for use in selectivity estimation. The authors apply a wavelet decomposition to the cumulative distribution of the dataset, and build and maintain histograms over a subset of wavelet coefficients. These histograms can then be used to efficiently reconstruct the cumulative distribution for both selectivity and approximate query processing purposes. Bruno and Chaudhuri [14] present a technique for estimating the selectivities of intermediate expressions based on using SITs (statistics on intermediate tables) for subexpressions, and present an algorithm to select the subexpressions on which they capture statistics. The algorithm presented uses a greedy heuristic comparing the relative difference between upper and lower bounds on selectivities for two successive operators in a query plan.

5.6.2 Sampling-Based Selectivity Estimation

Much of the work in sampling-based estimation lies at the intersection of statistics and databases. The literature in the statistics community is far more extensive on sampling

techniques, estimators and their mathematical properties, and the work in the database community often leverages this literature at their intersection. We briefly outline several earlier works on applying simple statistical techniques for estimation, before addressing the question of how to actually perform sampling for estimation purposes.

Lipton et. al. [60] present a sampling technique for estimating cardinalities and selectivities for selections and joins in relational databases. This work represents one of the classical attempts at using sampling for estimation and contributed the idea that sampling input relations and processing queries over samples can yield useful cardinality estimates. Haas et. al. [43] present sampling-based estimators for the number of distinct values of an attribute in a relation, which can be useful in query optimization for determining both selection and join orderings. The estimator they present explicitly takes account of degree of skew present in the distribution to reduce the variance in estimates.

However there are issues with basic approaches of processing queries on uniform-random sampling on input relations for the case of joins and aggregations over selections, as shown by Chaudhuri et. al. [23, 21]. To this end, a variety of of specialized query processing techniques have been developed to support estimation through processing on samples [22, 58, 50, 95]. Pulse adopts the classical approach of histograms, investigating the use of sampling during histogram construction alone. While the idea of sampling is clearly applicable to Pulse, the issue lies in understanding how these specialized sampling techniques could be adapted to work with polynomials, and incorporating the structure of the polynomials (through their coefficients) into sampling techniques.

5.6.3 Handling Correlations and Functional Dependencies

Poosala and Ioannidis [74] compare the use of a multidimensional histogram and the singular value decomposition technique to explicitly approximate joint distributions that avoids requiring independence assumptions. Briefly, their conclusions state the use of multidimensional MaxDiff histograms outperform SVD techniques in terms of accuracy. In a follow-up paper, Bruno and Chaudhuri [15] investigate the use of all available SITs to minimize the number of independence assumptions made when evaluating conditional selectivities. Getoor, Taskar and Koller [38] looked at the use of graphical models in selectivity estimation, where the graphical models are able to both detect dependencies in the dataset and represent the joint distribution of the attributes involved in a factorized manner according to the structure of the graphical model. Ilyas et. al. [47] present the CORDS system that is a data-driven method for detecting correlations between a relation's attributes. The

technique presented is based on sampling attribute values and subsequently applying a chi-squared analysis to the sample. CORDS additionally analyzes the number of distinct values in a column to detect soft functional dependencies. At a high-level the goal here is to recommend correlated or functionally dependent column groups on which the system should maintain joint distribution statistics to reduce errors during selectivity estimation.

5.7 Multi-Query Optimization

Multi-query optimization has been applied to both the ad-hoc query processing found in traditional databases, and to stream processing environments. Note that while these techniques do not explicitly consider polynomial processing, our own technique is independent of the underlying query processor thus allowing comparison to the existing work.

5.7.1 Multi-Plan Optimization

Multi-query optimization is a well-studied topic in the context of traditional relational databases [72, 79], where many of these techniques involve analysis of plans for those queries concurrently posed to the system. Roy et. al. [77] present a set of multi-query optimization heuristics on the AND-OR DAG representation of a set of queries based on both the basic Volcano [40] algorithm for MQO, as well as their own greedy heuristic. The greedy heuristic selects an operator compatible for sharing one at a time, specifically the operator which maximally reduces the total cost of all plans. The authors then present several optimizations to further reduce the set of operators considered greedily based on several structural and cost properties of the AND-OR graph and the cost model. Zhou et. al. [97] present a solution to the MQO problem based on computing a good set of candidate subexpressions to consider using during the optimization phase of query compilation. Their contributions include both developing heuristics to efficiently determine this advantageous set of common subexpressions, as well as efficiently pruning this set during query optimization. Their technique requires no modification to existing query optimizers since they advocate simply extending the candidate plan set with the materialized results of the common subexpression. Our approach to multi-query optimization differs from these by focusing on adaptivity in its design, and by integrating its sharing mechanism with other basic plan optimizations such as reordering operators. As such we do not perform classical dynamic programming on query plans, nor does our heuristic operate at the whole plan level. Rather, our design applies a much finer-grained control, and consequently less drastic transitions of the query plan during optimization phases, but is capable of optimizing more frequently.

5.7.2 Sharing in Stream Processing Engines

Due to the recent emergence of stream processing engines, multi-query optimization in the stream context has not been investigated to the same depth as with classical databases. The Aurora stream processing engine [2] provides support for expressing queries that are directed acyclic graphs, where the end user is expected to provide the shared form of the query themselves as common subexpressions. Krishnamurthy, Wu and Franklin [56] present an adaptive sharing algorithm focusing on exploiting sharing opportunities between differing window specifications and for equivalent aggregates over different data partitions. The second technique for sharing work amongst equivalent aggregates forms the basis for our union-sharing technique which explicitly eliminates duplicates adaptively in the middle of the query network, without requiring any upstream sharing. In our work, we need not explicitly share work across different window specifications – this is one of the advantages of maintaining a continuous state model, where we intrinsically gain the benefits of shared windows given our polynomial representation. Furthermore our multi-query sharing technique novelly combines both cover-sharing and union-sharing using a set of simple heuristics for both determining which type of sharing to use, and how to allocate consumers to shared instances. Finally the key details of each algorithm are novel to our polynomial datatypes, such as the duplicate elimination algorithm and computation of a covering expression.

5.7.3 Staged Databases

Staged databases [45] represent a philosophically different approach to multi-query optimization, in that rather than applying a purely static plan-based approach, the authors design a highly adaptive run-time approach to detecting sharing opportunities based on operator micro-engines. This approach exploits "natural" sharing opportunities following the generation of a query plan, rather than explicitly searching the space of possible plans for shareable operators during query compile time. Pulsedoes not consider using such an adaptive MQO approach. Note that while such techniques avoid potentially expensive optimizations during query compile time, in the case of stream processing engines it may be advantageous to attempt more aggressive plan-based optimization give the relatively static nature of continuous queries. Furthermore plan-based MQO approaches can apply transparently of the underlying execution environment, for example on shared-nothing distributed stream processing engines.

5.8 Mathematical Software and Computer Algebra Systems

We now describe several related software systems from the symbolic and numerical mathematical computation community. In general these systems support a far wider range of mathematical operations and in particular their symbolic systems are capable of applying a greater range of equivalence and rewrite rules. Our position is that we do not wish to reinvent the wheel when it comes to mathematical software, but clearly we would like to leverage such systems while implementing queries that are both computationally intensive and also access high-rate stream datasets. We describe two categories of software, symbolics software and general purpose numerics software.

5.8.1 Mathematica and Maxima

We start by discussing the symbolic solving capabilities of these two mathematical software systems. Mathematica [109] is capable of solving a large number of different ODE types with its `DSolve` function, including first-order ODEs, linear second-order ODES, ODE systems and generally higher-order linear ODEs. Additionally `DSolve` can handle a small subset of PDEs including first-order linear and quasilinear PDEs, as well as a restricted form of a linear second-order PDE.

Maxima [101] is an opensource computer algebra system that is capable of analytically solving ODEs with its `ode2` method. The ODEs supported include linear, separable, and homogeneous first-order ODEs as well as constant coefficient, exact, and linear homogeneous second-order ODEs, and those second-order ODEs that can be solved by the variation of parameters method. Maxima also supports solving initial and boundary value problems symbolically, as well as computing numerical solutions with a 4th order Runge-Kutta method (the `rk` method). Other popular symbolic mathematical software includes Sage [106], GiNaC [105] and Axiom [99].

We also briefly discuss Mathematica's numerical solving capabilities through its `NDSolve` method. This method supports both ODEs and a restricted class of PDEs. For its ODE solver, Mathematica implements both an explicit and implicit Runge-Kutta method, with the explicit solvers including methods of order ranging from 2 to 9. For its PDE solver, Mathematica implements the method of lines which requires PDEs to have an initial value problem specified in at least one dimension, with a numerical ODE integrator applied over this dimension. The above feature set is only a partial description of those techniques most relevant to this thesis. We refer the reader to the software documentation [] for details on other methods.

5.8.2 Matlab and Octave

Matlab provides a suite of ODE solvers as described in [] for handling stiff and non-stiff ODEs. The `ode23` and `ode45` solvers handle non-stiff equations through the use of Runge-Kutta techniques, with the `ode45` solver based on both a 4th and 5th order Runge-Kutta method known as the Dormand-Prince method. We also note that the `ode15s` solver is capable of handling differential algebraic equations. We consider extending our support for differential equations to this as part of future work. Matlab also supports solving PDEs with the finite element method, and we use this as our backend solver for the elliptic PDEs described in our data model. The finite element method includes support for adaptive mesh refinement.

Octave is an open-source alternative to Matlab that also includes an ODE solver. This solver is based on the LSODE package [103]. Octave also supports differential algebraic equations using the DASPK package. There is limited support for solving PDEs in Octave, for example with the OctMesh [102] which acts as an Octave binding to the LibMesh library [100]. An alternative approach to using a generic numerical mathematical software system to solve PDEs is to use a library designed specifically for this purpose, for example the deal.ii library [104].

Chapter 6

Conclusions

We have presented the Pulse framework for performing relational-style query processing over data stream attributes modeled as piecewise polynomial functions of a time variable. Pulserepresents queries as simultaneous equation systems for a variety of relational operators including filters, joins and standard aggregates. We use piecewise polynomials to provide a compact, approximate representation of the input dataset and primarily consider supporting predictive processing with these polynomials, where the polynomials are used for extrapolation and subsequently validated against the actual data as it arrives into the system. Given that our polynomial models are dataset approximations, we provide query language extensions for users to specify precision bounds, and leverage these bounds during validating predicted results with actual results. The key idea here is that we can perform state-change processing where we only process queries if they significantly affect the result according to user-specified bounds. We describe the precision bound inversion problem as that of determining precision bounds at query inputs rather than outputs, and present a set of heuristics to compute input precision bounds. We have implemented Pulse on top of the Borealis stream processing engine and evaluated it on two real-world datasets from financial and moving object applications. Our experimental results show that Pulse is able to achieve significant performance improvements by processing relational queries directly on the mathematical representation of these polynomials and exploiting precision bounds in comparison to standard tuple-based stream processing.

The polynomial processing described in the first part of this thesis differs significantly from standard approaches to relational query processing with its use of equation systems solvers. Both standard databases and stream processing engines model and profile query processing performance internally, and subsequently apply a wide range of optimization

techniques to improve processing performance. The second part of this thesis visits this challenge in the context of our particular style of processing. We present the design of a selectivity estimator that is capable of computing selectivities for relational operators applied over piecewise polynomials, as well as a multi-query optimizer that uses these selectivities to determine when it is advantageous to construct a global query plan that shares work across individual queries. In summary, our selectivity estimator performs its work by maintaining histograms on a parameter space of the polynomials sampled at runtime. We then define histogram operations according to the processing performed by filters, joins and aggregates on individual histogram bins in this parameter space. Next, our multi-query optimizer considers two types of sharing algorithms which differ in where they perform input duplicate elimination prior to sharing work across multiple queries. Our sharing algorithm is implemented as a set of heuristic conditions that locally adapt a global query based on both statistics collected and selectivity estimates provided by our custom estimator, picking the adaptations to apply in a greedy manner.

We implemented our selectivity estimator and query optimizer within the Borealis stream processing engine, monitoring and manipulating queries prior to their execution in Pulse. The experimental results gathered with this implementation have both positive and negative aspects. We compared our selectivity estimation technique to a simple sampling-based technique for estimation. This comparison demonstrated a negative result, whereby our histogram-based estimation is outperformed by the sampling technique both in terms of accuracy and estimation overhead, for selective operators. Note that our technique still has several advantages over an approach such as sampling. Sampling is known to perform poorly on min and max aggregates, indeed there is no known bound for the error present in any estimate obtained via sampling. We believe our estimation technique will clearly outperform sampling in this case. Furthermore we believe our technique can provide additional benefits through its ability to provide both lower and upper bounding values on selectivities, and that these bounds will turn out to be tighter than those obtained from theoretical analysis of sampling-based estimation techniques. However a full investigation of these claims lies outside the scope of this thesis, and as such is an immediate topic for future work. The positive takeaway from our experimentation lies with the optimization mechanisms and algorithm. We have demonstrated that a set of shareable operators can benefit greatly from shared execution in the context of processing segments, on a real-world dataset. This applies when a covering operator has moderately low selectivity, and thus is likely to be applicable in actual queries posed by end-users.

While this thesis has solved critical functionality and scalability challenges for supporting

domain-specific knowledge through models expressed as polynomial functions in a stream processing context, the long-term vision of rich, extensive, support for using models as representations of data in a DBMS is still in its infancy in the current state-of-the-art, especially in terms of being a recognizable and usable feature in actual deployed systems. We outline a wide variety of topics for future investigation, ranging from the immediate to the fundamental challenges at hand.

Unmodeled attribute support and key attribute scalability.
Pulse as has been described in this thesis, operates purely on modeled attributes, and does not incorporate processing for attributes that are not associated with any model. We believe supporting standard query processing techniques on such attributes side-by-side with Pulse's processing is a key usability requirement in garnering adoption of model-driven databases in real-world applications. The main challenge in supporting unmodeled attributes is that they may greatly increase the degree of segmentation present in the input workload, for example the unmodeled attribute may take on many values within the valid time range of a segment, and requiring that segment to be represented as multiple segments. There are several simple approaches one could adopt to provide such functionality, for example tuples could consist of both modeled and unmodeled attributes, and operators could be separated into two parts depending on the type of attribute being processed. We could apply standard operators to transform unmodeled attributes, passing through the modeled attributes, and then apply Pulse's operators to handle these. An alternative approach is to view unmodeled attributes as constants within each segments valid time range, and then directly process such constant segments as part of Pulse.

Key attributes can also potentially impose scalability challenges in addition to unmodeled attributes, depending on the size of the key's attribute domain. Key attributes effectively define partitions for Pulse's query processing in the case of selective operators, and present a natural opportunity to exploit data parallelism in enabling multiprocessor or distributed processing to scale system throughput. However aggregate operations may wish to utilize models from all or groups of key values, creating data dependencies downstream from any parallel or distributed execution, complicating our efforts to scale the system. We foresee opportunities in effectively handling distributed processing of general filter-join-aggregate queries in Pulse, that will require novel, ideally incremental, processing techniques exploiting our internal representation of queries as equation systems.

Database support for attributes modeled by differential equations.
We view the polynomials described so far as one type of model, and have described how to support polynomials as a first-class datatype in Pulse. We now focus on challenges in providing database support for other model types, in particular looking at differential equations, and how they may be represented as datatypes and processed with relational-style queries. In a similar fashion to our polynomial segment data model, we envision allowing individual attributes in a schema to be represented as differential equations, including both ODEs and PDEs. In the case of ODEs, our system could support specification in two ways, first declaratively, as part of the query, as a linear differential equation $y^{(n)} = b(x) + \sum_{i=1}^{n} a_i(x)y^{(i)}$, with the functions $a_i(x)$ limited to polynomials, and secondly as a user-defined function to support more general non-linear ODEs. In the case of schema attributes represented by PDEs, two common types of PDEs are: i) elliptic equations and ii) PDEs with IVPs in at least one dimension. These correspond to the types of PDEs supported by Matlab [107] and Mathematica [109], two popular mathematical software packages. The solution techniques for the two PDEs are finite element methods, and finite difference methods respectively. As a key point, we remark that philosophy is that our contribution is not intended to be in advancing the state-of-the-art in numerical methods for solving PDEs, rather we want to leverage existing tools to solve PDEs representing a single attribute, and then consider this attribute in a relational data model, leaving us to focus on the interaction between query processing and existing well-known numerical methods.

Given these attributes which may be represented as a differential equation, our approach would require a differential equation solver as part of processing relational queries on such attributes. ODE and PDE solvers commonly handle two types of constraints placed on solutions, namely initial value problems and boundary value problems. Initial value problems pose a constraint on the solution's value at the lower boundary of the independent variable's valid range. We perceive a streaming nature to initial value problems, where a data stream specifies the initial values. A differential equation solver would then compute a solution for every input arriving on the data stream. Boundary value problems pose constraints on both the lower and upper boundary values of the independent variables' valid ranges. There are a variety of different boundary conditions including Neumann conditions, Dirichlet conditions and Cauchy conditions, each of which varies in the term they constrain between combinations of the dependent variable's value and derivative. Loosely speaking, boundary conditions define interpolation problems, where the differential equation solution represents dependent attribute values within the valid range. Intuitively boundary conditions arise in historical query processing where we have a dataset at hand, and we wish to interpolate

while respecting the values witnessed in the dataset.

Query processing techniques for handling the various forms of and problems posed on differential equations will encompass both analytical solutions and numerical approximations to the ODE or PDE. We remark that the outputs of many differential equation solvers falls into one of two basic types, symbolic expressions, which include exponential and trigonometric operators, and interpolating polynomials. Our query processor would need to be extended to evaluate relational operators over symbolic expressions, where we may have to handle operations combining both the symbolic form and the polynomial form. Furthermore, in leveraging existing tools providing numerical methods for solving both ODEs and PDEs, we believe that one key issue lies and in understanding the approximation provided by these techniques. The approximation should be captured in a revised error model, and leverage query semantics in conjunction with the revised definition of errors to control the tradeoff between solving accuracy and performance. The key idea here is that query-agnostic numerical methods are likely to generate solutions with unnecessarily high numerical precision, and that this can be avoided at run-time by dynamically configuring solver precisions based on user-specified tolerances and the continuous queries' intermediate results.

Expressive model types.

We believe that a major guiding principle in providing database support for other model types is that these datatypes should not be black boxes, rather they should expose sufficient semantics for query optimization, similar to enhanced abstract datatypes [80]. A critical part of this direction would be to investigate the design of an extensible interface and how that interface can support mechanisms for query optimization. One aspect of this challenge is to determine how we may leverage mathematical properties of these datatypes in defining query processing mechanisms and performing query optimization. For example, we consider it critical to determine the feasibility of an analytical solution due to the significantly simplified query processing and improved performance it provides, and envision accomplishing this with the aid of a computer algebra system (CAS) integrated with our stream processor.

We have a few initial insights into a data type interface, which is designed around two common properties of the aforementioned models. First we remark that each type of model supports a *sampling function* that we may use to drive a numerical solver. Next, we observe that numerical methods and solvers are predominantly iterative processes, and that this iteration is often driven by the semantics of the model, rather than data flow. Thus our abstraction decouples control flow from data flow to enable the iterative behavior

required to numerically evaluate models. Our framework exposes the following methods for a datatype to implement (in C++ syntax):

```
union InputValue { Tuple; Window; }
union TimeValue { TimePoint; TimeRange; }
union SolutionValue { Point; Segment; }

model_solver:
  void initModel(InputValue, TimeValue);
  boolean hasSolutions();
  pair<TimeValue, SolutionValue> getSolution();
  SolutionValue getSolution(TimeValue);
  boolean checkSolution(Segment, RelOp)
  Segment finalModel(TimeValue);
```

As part of future work, we intend to investigate both conversion abstractions that allow one model type to be cast to another, in addition to a modeling abstraction, where model types may be built or learned incrementally, although it is unclear how much commonality there is across types in these procedures. With this model-specific solving interface, we are able to define a standard numerical query solver based on the evaluation of difference predicates. Our general solver invokes the methods defined by each model type above, from the following pseudocode:

```
query_solver:
void solve_unary_equation(input):
  solver.init(input)
  while ( solver.hasSolutions() )
    solnTime, solnVal = solver.getSolution()
    if ( predicate(solnVal) )
      result = solver.final(solnTime)
      emit(solnTime, result)
    terminate()

void solve_binary_equation(inputA, inputB):
  if solverA.hasDifference(solverB)
    solve_unary_equation(
      solverA.difference(inputA, inputB))
  else
    solverA.init(inputA), solverB.init(inputB)
    while ( solverA.hasSolutions() )
```

```
solnATime, solnAVal = solverA.getSolution()
solnBTime, solnBVal = solverB.getSolution(solnATime)
if ( predicate(solnAVal, solnBVal) )
  result = output_attribute(
    solverA.final(solnBTime),
    solverB.final(solnBTime))
  emit(solnBTime, result)
terminate()
```

The above example shows symbolic manipulations with the `hasDifference` and `difference` functions. We show a difference equation with two variables being solved as a difference equation of one variable by computing a single model (segment) representing that difference. This solver also highlights other interesting challenges, including synchronization (note `solverB` samples at the time value yielded by `solverA`) and solver termination policies. Solver termination concerns how an optimizer might take advantage of mathematical properties to limit the solutions produced to exclude those that cannot be query results. We assume the existence of an `output_attribute` function above to apply any model transformations corresponding to non-selective operators (such as a `sum` aggregate). Note our interface is closed, that is our solver produces the same model type as its input.

```
initModel : set input segment as state
            determine multiplicity of roots
hasSolutions : return if any roots remain
getSolution : return time range and segment
              between last root and current root
checkSolution : check if entire segment
                satisfies relational operator
finalModel : return input segment
```

Figure 6.1: Example polynomial type sampling function.

It is my hope that this thesis has conveyed the potential benefits of embracing mathematical representations of data at the core of a database management system, and the tenets which have inspired the direction and execution of this research, namely that effective tools are built from extending the state-of-the-art both in terms of functionality and expressiveness, and from providing such functionality in an high-performance, usable manner to end users. I look forward to the opportunity to continue to design, architect and prototype such systems in the future.

Bibliography

[1] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Proc. of the Second Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 277–289, January 2005.

[2] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The International Journal on Very Large Data Bases*, 12(2):120–139, August 2003.

[3] Yanif Ahmad and Uğur Çetintemel. Declarative temporal data models for sensor-driven query processing. In *Proc. of the Fourth International Workshop on Data Management for Sensor Networks (DMSN'07)*, pages 37–42, September 2007.

[4] Yanif Ahmad, Olga Papaemmanouil, Uğur Çetintemel, and Jennie Rogers. Simultaneous equation systems for query processing on continuous-time data streams. In *Proc. of the 24th International Conference on Data Engineering (ICDE'08)*, pages 666–675, April 2008.

[5] Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In *Proc. of the 23rd International Conference on Data Engineering (ICDE'07)*, pages 1479–1480, April 2007.

[6] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 261–272, May 2000.

[7] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Proc. of the 20th International Conference on Data Engineering (ICDE'04)*, pages 350–361, March 2004.

[8] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 407–418, June 2004.

[9] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems*, 33(1):1–44, March 2008.

[10] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Contract-based load management in federated distributed systems. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 197–210, March 2004.

[11] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *The VLDB Journal: The International Journal on Very Large Data Bases*, 17(2):243–264, March 2008.

[12] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[13] Richard P. Brent. *Algorithms for Minimization without Derivatives*. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[14] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 263–274, June 2002.

[15] Nicolas Bruno and Surajit Chaudhuri. Conditional selectivity for statistics on query expressions. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 311–322, June 2004.

[16] Benno Bueller, Andreas Enge, and Komei Fukuda. Vinci: Computing volumes of convex polytopes. http://www.lix.polytechnique.fr/Labo/Andreas.Enge/Vinci.html.

[17] V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with seti. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[18] Sirish Chandrasekaran, Amol Deshpande, Mike Franklin, and Joseph Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[19] Sirish Chandrasekaran and Michael J. Franklin. Psoup: a system for streaming queries over streaming data. *The VLDB Journal: The International Journal on Very Large Data Bases*, 12(2):140–156, August 2003.

[20] Sirish Chandrasekaran and Michael J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 348–359, August 2004.

[21] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems*, 32(2):9, June 2007.

[22] Surajit Chaudhuri, Gautam Das, and Utkarsh Srivastava. Effective use of block-level sampling in statistics estimation. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 287–298, June 2004.

[23] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. On random sampling over joins. In *Proc. of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, pages 263–274, June 1999.

[24] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 379–390, May 2000.

[25] Reynold Cheng, Ben Kao, Sunil Prabhakar, Alan Kwan, and Yi-Cheng Tu. Adaptive stream filters for entity-based queries with non-value tolerance. In *Proc. of the 31st International Conference on Very Large Data Bases (VLDB'05)*, pages 37–48, August 2005.

[26] Graham Cormode and Minos Garofalakis. Approximate continuous querying over distributed streams. *ACM Transactions on Database Systems*, 33(2):1–39, June 2008.

[27] Charles D. Cranor, Yuan Gao, Theodore Johnson, Vladislav Shkapenyuk, and Oliver Spatscheck. Gigascope: high performance network monitoring with an sql interface. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, page 623, June 2002.

[28] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 864–875, August 2004.

[29] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 40–51, June 2003.

[30] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *Proc. of the Third Biennial Conference on Innovative Data Systems Research (CIDR'07)*, pages 412–422, 2007.

[31] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph Hellerstein, and Wei Hong. Model driven data acquisition in sensor networks. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 588–599, August 2004.

[32] Amol Deshpande and Samuel Madden. MauveDB: supporting model-based user views in database systems. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 73–84, June 2006.

[33] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 61–72, June 2002.

[34] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification.* Wiley-Interscience Publication, 2000.

[35] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, pages 419–429, May 1994.

[36] Ewgenij Gawrilow and Michael Joswig. Polymake: an approach to modular software design in computational geometry. In *Proceedings of the 17th annual symposium on Computational geometry*, pages 222–231, 2001.

[37] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, pages 1123–1134, June 2008.

[38] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *Proc. of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, pages 461–472, May 2001.

[39] Michael Gibas, Ning Zheng, and Hakan Ferhatosmanoglu. A general framework for modeling and processing optimization queries. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB'07)*, pages 1069–1080, September 2007.

[40] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proc. of the 9th International Conference on Data Engineering (ICDE'93)*, pages 209–218, April 1993.

[41] Stéphane Grumbach, Philippe Rigaux, Michel Scholl, and Luc Segoufin. The DEDALE prototype. In *Constraint Databases*, pages 365–382, 2000.

[42] Stéphane Grumbach, Philippe Rigaux, and Luc Segoufin. Manipulating interpolated data is easier than you thought. In *Proc. of the 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 156–165, September 2000.

[43] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 311–322, 1995.

[44] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, Mohamed Y. Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, R. Gwadera, Ihab F. Ilyas, Mirette S. Marzouk, and Xiaopeng Xiong. Nile: A query processing engine for data streams. In *Proc. of the 20th International Conference on Data Engineering (ICDE'04)*, page 851, March 2004.

[45] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, 2003.

[46] Jeong-Hyon Hwang, Ying Xing, Ugur Çetintemel, and Stanley B. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proc. of the 23rd International Conference on Data Engineering (ICDE'07)*, pages 176–185, April 2007.

[47] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 647–658, June 2004.

[48] Ankur Jain, Edward Y. Chang, and Yuan-Fang Wang. Adaptive stream resource management using kalman filters. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 11–22, June 2004.

[49] Shawn R. Jeffery, Minos N. Garofalakis, and Michael J. Franklin. Adaptive cleaning for rfid data streams. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB'06)*, pages 163–174, September 2006.

[50] Shantanu Joshi and Christopher Jermaine. Materialized sample views for database approximation. *IEEE Transactions on Knowledge and Data Engineering*, 20(3):337–351, March 2008.

[51] Bhargav Kanagal and Amol Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *Proc. of the 24th International Conference on Data Engineering (ICDE'08)*, pages 1160–1169, April 2008.

[52] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences.*, 51(1):26–52, 1995.

[53] Eamonn J. Keogh, Selina Chu, David Hart, and Michael J. Pazzani. An online algorithm for segmenting time series. In *Proc. of the 2001 (ICDM'01)*, pages 289–296, November 2001.

[54] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. Towards correcting input data errors probabilistically using integrity constraints. In *MobiDE '06: Proceedings of the 5th ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 43–50, June 2006.

[55] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 623–634, June 2006.

[56] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 623–634, June 2006.

[57] Gabriel M. Kuper, Leonid Libkin, and Jan Paredaens, editors. *Constraint Databases.* Springer, 2000.

[58] Per-Åke Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. Cardinality estimation using sample views with quality assurance. In *Proc. of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*, pages 175–186, June 2007.

[59] Ling Lin and Tore Risch. Querying continuous time sequences. In *Proc. of the 24th International Conference on Very Large Data Bases (VLDB'98)*, pages 170–181, August 1998.

[60] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD'90)*, pages 1–11, May 1990.

[61] Nikos Mamoulis and Dimitris Papadias. Slot index spatial join. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):211–231, January 2003.

[62] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *Proc. of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 448–459, June 1998.

[63] Mohamed F. Mokbel and Walid G. Aref. PLACE: A scalable location-aware database server for spatio-temporal data streams. *IEEE Data Eng. Bull.*, 28(3):3–10, 2005.

[64] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, 2003.

[65] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[66] Leonore Neugebauer. Optimization and evaluation of database queries including embedded interpolation procedures. In *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD'91)*, pages 118–127, May 1991.

[67] Inc. New York Stock Exchange. Monthly TAQ, http://www.nysedata.com/nysedata/.

[68] Jinfeng Ni and Chinya V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Transactions on Knowledge and Data Engineering*, 19(5):663–678, May 2007.

[69] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 563–574, June 2003.

[70] Chris Olston and Jennifer Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proc. of the 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 144–155, September 2000.

[71] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. Streaming pattern discovery in multiple time-series. In *Proc. of the 31st International Conference on Very Large Data Bases (VLDB'05)*, pages 697–708, August 2005.

[72] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Proc. of the 4th International Conference on Data Engineering (ICDE'88)*, pages 311–319, February 1988.

[73] Jignesh M. Patel, Yun Chen, and V. Prasad Chakka. Stripes: An efficient index for predicted trajectories. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 637–646, June 2004.

[74] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 486–495, August 1997.

[75] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, pages 294–305, June 1996.

[76] Frederick Reiss and Joseph M. Hellerstein. Data triage: An adaptive architecture for load shedding in telegraphcq. In *Proc. of the 21st International Conference on Data Engineering (ICDE'05)*, pages 155–156, April 2005.

[77] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 249–260, May 2000.

[78] Florin Rusu and Alin Dobra. Fast range-summable random variables for efficient aggregate estimation. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 193–204, June 2006.

[79] Timos K. Sellis and Subrata Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, June 1990.

[80] Praveen Seshadri. Enhanced abstract data types in object-relational databases. *The VLDB Journal: The International Journal on Very Large Data Bases*, 7(3):130–140, August 1998.

[81] Shetal Shah and Krithi Ramamritham. Handling non-linear polynomial queries over dynamic data. In *Proc. of the 24th International Conference on Data Engineering (ICDE'08)*, pages 1043–1052, April 2008.

[82] Hagit Shatkay and Stanley B. Zdonik. Approximate queries and representations for large data sequences. In *Proc. of the 12th International Conference on Data Engineering (ICDE'96)*, pages 536–545, February 1996.

[83] Jin Shieh and Eamonn J. Keogh. sax: indexing and mining terabyte sized time series. In *Proc. of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 623–631, 2008.

[84] Adam Silberstein, Alan Gelfand, Kamesh Munagala, Gavino Puggioni, and Jun Yang. Making sense of suppressions and failures in sensor data: A bayesian approach. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB'07)*, pages 842–853, September 2007.

[85] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 611–622, June 2004.

[86] Yufei Tao, George Kollios, Jeffrey Considine, Feifei Li, and Dimitris Papadias. Spatio-temporal aggregation using sketches. In *Proc. of the 20th International Conference on Data Engineering (ICDE'04)*, pages 214–226, March 2004.

[87] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 790–801, September 2003.

[88] Yufei Tao and Xiaokui Xiao. Primal or dual: which promises faster spatiotemporal search? *The VLDB Journal: The International Journal on Very Large Data Bases*, 17(5):1253–1270, August 2008.

[89] Nesime Tatbul, Ugur Çetintemel, and Stanley B. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB'07)*, pages 159–170, September 2007.

[90] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 309–320, September 2003.

[91] Nesime Tatbul and Stanley B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB'06)*, pages 799–810, September 2006.

[92] Arvind Thiagarajan and Samuel Madden. Querying continuous functions in a database system. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, pages 791–804, June 2008.

[93] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 407–418, June 2006.

[94] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. of the 21st International Conference on Data Engineering (ICDE'05)*, pages 791–802, April 2005.

[95] Fei Xu, Christopher Jermaine, and Alin Dobra. Confidence bounds for sampling-based group by estimates. *ACM Transactions on Database Systems*, 33(3):1–44, 2008.

[96] Man Lung Yiu, Nikos Mamoulis, and Dimitris Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):820–833, June 2005.

[97] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proc. of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*, pages 533–544, June 2007.

[98] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 358–369, August 2002.

[99] http://axiom.axiom-developer.org/. Axiom.

[100] http://libmesh.sourceforge.net. LibMesh.

[101] http://maxima.sourceforge.net. Maxima.

[102] http://octmesh.forja.rediris.es/. OctMesh.

[103] https://computation.llnl.gov/casc/odepack/. LSODE.

[104] http://www.dealii.org/. Axiom.

[105] http://www.ginac.de. Ginac.

[106] http://www.sagemath.org. Sage.

[107] The Mathworks, http://www.mathworks.com. Matlab.

[108] http://www.navcen.uscg.gov/enav/ais/default.htm. U.S. Coast Guard Navigation Center, Automatic Identification System.

[109] Wolfram Research, http://www.wolfram.com. Mathematica.