

**CERIAS Tech Report 2009-35**  
**Efficient query processing for rich and diverse real-time data**  
by Nehme, Rimma  
Center for Education and Research  
Information Assurance and Security  
Purdue University, West Lafayette, IN 47907-2086

**PURDUE UNIVERSITY**  
**GRADUATE SCHOOL**  
**Thesis/Dissertation Acceptance**

This is to certify that the thesis/dissertation prepared

By Rimma Nehme

Entitled Efficient Query Processing for Rich and Diverse Real-Time Data

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Elisa Bertino

Chair

Jennifer Neville

Elke Rundensteiner

Ahmed Elmagarmid

Sunil Prabhakar

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Elisa Bertino

Elke Rundensteiner

Approved by: William J. Gorman

Head of the Graduate Program

06/10/2009

Date

**PURDUE UNIVERSITY  
GRADUATE SCHOOL**

**Research Integrity and Copyright Disclaimer**

Title of Thesis/Dissertation:

Efficient Query Processing for Rich and Diverse Real-Time Data

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22*, September 6, 1991, *Policy on Integrity in Research*.\*

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Rimma Nehme

\_\_\_\_\_  
Signature of Candidate

06/10/2009

\_\_\_\_\_  
Date

\*Located at [http://www.purdue.edu/policies/pages/teach\\_res\\_outreach/c\\_22.html](http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html)

EFFICIENT QUERY PROCESSING  
FOR RICH AND DIVERSE REAL-TIME DATA

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Rimma Nehme

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2009

Purdue University

West Lafayette, Indiana

UMI Number: 3379699

All rights reserved !

INFORMATION TO ALL USERS !

The quality of this reproduction is dependent upon the quality of the copy submitted.

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



UMI 3379699

Copyright 2009 by ProQuest LLC. !

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



ProQuest LLC  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

*To Freddy*

## ACKNOWLEDGMENTS

I am extremely grateful to many people for all the guidance and help I have received throughout the period of my Ph.D. studies.

My deepest gratitude is to Dr. Elisa Bertino. I have been fortunate to have an advisor who gave me the freedom to explore on my own. I am grateful to her for holding me to a high research standard and having high expectations from me, while giving constant encouragement and support throughout my graduate studies.

I am indebted to my co-advisor Dr. Elke A. Rundensteiner for introducing me to the idea of pursuing the Ph.D. degree in the first place and for all the guidance and help she has given me over the years. Elke's insightful comments and constructive criticisms at different stages of my research were thought-provoking and they helped me focus my ideas and encouraged to try "risky" approaches. She has been a wonderful role model as a researcher and as a teacher.

During the three summers at Microsoft Research (MSR), I have worked with a wonderful group of people. In particular, I would like to thank my mentors there Dr. Nicolas Bruno and Dr. David Lomet for exposing me to a number of interesting research problems, engaging discussions and for inspiring me to do my very best. I will cherish my experience in the Data Management, Exploration and Mining (DMX) Group and the Database (DB) Group at MSR for the rest of my life. Working with this excellent group of people significantly contributed to my passion for database systems research.

I also would like to thank the current and the former members of the Indiana Center for Database Systems (ICDS) at Purdue University and the Database Systems Research Group (DSRG) at Worcester Polytechnic Institute with whom I have interacted during the course of my graduate studies. Particularly, I would like to acknowledge Ashish Kamra, Mariana Jbantova, Chris Mayfield, Hazem D. Elmeleegy,

Mohamed Y. Eltbakh, Dr. Hyo-Sang Lim, Venkatesh Raghavan, Dr. Luping Ding, Dr. Yali Zhu, Dr. Bin Liu, Dr. Maged El-Sayed, Dr. Song Wang, Di Yang, Karen Works, Mo Liu, Abhishek Mukerji, Mummoorthy Murugesan (and many others) for the many valuable discussions that helped me understand my research area better.

I would like to express my gratitude to the rest of my advisory committee at Purdue, Dr. Walid Aref, Dr. Ahmed Elmagarmid, Dr. Sunil Prabhakar and Dr. Jennifer Neville, for their help and invaluable comments and suggestions.

Finally, and most importantly, I would like to thank my husband Alfred for his love, support and the never-ending encouragement. His confidence in me has been absolutely invaluable throughout my Ph.D. studies. I dedicate this thesis to him.



## TABLE OF CONTENTS

	Page
LIST OF TABLES . . . . .	ix
LIST OF FIGURES . . . . .	x
ABSTRACT . . . . .	xiv
1 Introduction . . . . .	1
1.1 Data Stream Management Systems . . . . .	1
1.2 Emerging Real-life Streaming Applications . . . . .	3
1.3 Features for the Next Generation of Data Stream Management Systems . . . . .	6
1.3.1 Access Control for Streaming Data . . . . .	9
1.3.2 Tagging Streaming Data . . . . .	9
1.3.3 Diversity-Aware Query Processing . . . . .	10
1.4 Overview of Our Approach . . . . .	11
1.5 Contributions . . . . .	13
1.6 Summary and Outline . . . . .	17
2 Background and Related Work . . . . .	18
2.1 Data Stream Management Systems . . . . .	18
2.2 Query Optimization Techniques . . . . .	20
2.3 Adaptive Query Processing Techniques . . . . .	25
2.4 Streaming Metadata . . . . .	27
2.5 Learning Techniques . . . . .	27
2.6 Security and Access Control Enforcement . . . . .	29
2.7 Tagging Methods . . . . .	31
2.8 Summary . . . . .	33
3 Security and Access Control for Streaming Data . . . . .	34
3.1 Security in Data Stream Management Systems . . . . .	35
3.1.1 Challenges . . . . .	37
3.1.2 Our Contributions: The <i>FENCE</i> Framework . . . . .	38
3.2 Problem Formulation . . . . .	39
3.3 Overview of <i>FENCE</i> Framework . . . . .	41
3.3.1 <i>FENCE</i> Architecture . . . . .	41
3.3.2 An Instance of the <i>FENCE</i> Framework . . . . .	43
3.4 Dynamic Security Policy Model . . . . .	45
3.4.1 General Security Punctuation Schema . . . . .	45
3.4.2 Semantics of Security Punctuations . . . . .	46

	Page
3.4.3	Examples of Security Punctuations . . . . . 48
3.4.4	Security Punctuations Generation . . . . . 50
3.5	Security-Aware Continuous Query Processing . . . . . 51
3.5.1	Naive Approach . . . . . 51
3.5.2	Security Filter Approach (SFA) . . . . . 53
3.5.3	Query Rewrite Approach (QRA) . . . . . 57
3.5.4	Pros and Cons of QRA and SFA . . . . . 60
3.6	Experimental Study . . . . . 62
3.6.1	Experimental Setup . . . . . 62
3.6.2	Effectiveness of Security Punctuations . . . . . 65
3.6.3	Comparison of SA-CQP Methods . . . . . 67
3.6.4	Overhead of Security Enforcement . . . . . 68
3.6.5	Summary of Experimental Results . . . . . 70
3.7	Conclusion . . . . . 71
4	Tagging of Streaming Data . . . . . 72
4.1	Tagging in Data Stream Environments . . . . . 73
4.1.1	Challenges . . . . . 73
4.1.2	Alternative Tagging Methods . . . . . 74
4.1.3	Our Proposed Solution: The Stream Tag Framework . . . . . 76
4.1.4	Our Contributions . . . . . 78
4.2	Stream Tag Framework Overview . . . . . 78
4.3	Streaming Tags (or Tick-Tags) . . . . . 79
4.3.1	What is a Tick-Tag? . . . . . 79
4.3.2	Tick-Tag Physical Design . . . . . 81
4.3.3	Tag Query Language (TAG-QL) . . . . . 84
4.3.4	Tick-Tag Examples . . . . . 85
4.3.5	Tick-Tag Generation . . . . . 86
4.4	Tag-Based Query Processing . . . . . 87
4.4.1	Tag-Oriented Query Processing . . . . . 87
4.4.2	Tag-Aware Query Processing . . . . . 91
4.5	Physical Implementation . . . . . 94
4.6	Experimental Study . . . . . 96
4.6.1	Experimental Setup . . . . . 96
4.6.2	Cost of Tagger Operator . . . . . 100
4.6.3	Comparison of Tick-Tag Approach Against Alternatives . . . . . 102
4.6.4	Cost of Tag Join Operator . . . . . 104
4.6.5	Cost of Tag-Aware Join Operator . . . . . 105
4.7	Conclusion . . . . . 106
5	Diversity-Aware Query Processing . . . . . 107
5.1	Core Query Mesh (QM) . . . . . 107
5.1.1	Single versus Multiple Execution Plans . . . . . 107

	Page
5.1.2 Our Proposed Solution: The Query Mesh . . . . .	109
5.1.3 Challenges . . . . .	111
5.1.4 QM Architecture . . . . .	112
5.1.5 QM Assumptions . . . . .	113
5.2 The Query Mesh Optimizer . . . . .	113
5.2.1 Data Sampling . . . . .	113
5.2.2 Query Mesh Search Space . . . . .	114
5.2.3 Query Mesh Optimizer Sub-problems . . . . .	116
5.2.4 Query Mesh Cost Model . . . . .	118
5.2.5 Optimal Query Mesh Search Algorithm . . . . .	119
5.2.6 Query Mesh Search Heuristics . . . . .	121
5.3 The Query Mesh Executor . . . . .	127
5.3.1 Instantiation of Physical Infrastructure . . . . .	127
5.3.2 Physical Execution . . . . .	130
5.4 Query Mesh Experimental Study . . . . .	131
5.4.1 Experimental Setup . . . . .	133
5.4.2 Results and Analysis . . . . .	134
5.4.3 Summary of Core QM Experimental Conclusions . . . . .	142
5.5 QM Conclusion . . . . .	142
5.6 Self-Tuning Query Mesh (ST-QM) . . . . .	143
5.6.1 Motivation for Adaptivity . . . . .	144
5.6.2 Adaptive Multi-Plan Query Processing . . . . .	145
5.6.3 Our Proposed Solution: ST-QM . . . . .	148
5.7 Overview of Self-Tuning Query Mesh . . . . .	149
5.7.1 The Main Idea . . . . .	149
5.7.2 Query Mesh Concept Drifts . . . . .	149
5.7.3 ST-QM Architecture . . . . .	151
5.8 ST-QM Monitor . . . . .	153
5.8.1 Input Data Monitoring . . . . .	153
5.8.2 Execution Statistics Monitoring . . . . .	156
5.9 ST-QM Analyzer . . . . .	157
5.9.1 Phase I: Concept Drift Detection . . . . .	158
5.9.2 Phase II: Tuning Recommendations . . . . .	160
5.10 ST-QM Actuator . . . . .	163
5.10.1 Physical Execution of Adaptivity . . . . .	163
5.10.2 State Management and Adaptivity . . . . .	164
5.11 Self-Tuning Query Mesh Experimental Study . . . . .	165
5.11.1 Experimental Setup . . . . .	165
5.11.2 Results and Analysis . . . . .	168
5.11.3 Summary of ST-QM Experimental Conclusions . . . . .	175
5.12 ST-QM Conclusion . . . . .	176
5.13 Uncertainty-Aware Query Mesh (UA-QM) . . . . .	177

	Page
5.13.1 Problems with Uncertainty Ignorance . . . . .	178
5.13.2 Challenges . . . . .	180
5.13.3 Our Proposed Solution: UA-QM . . . . .	181
5.14 UA-QM Framework . . . . .	182
5.14.1 UA-QM Architecture . . . . .	182
5.14.2 Uncertainty Cases in Query Mesh . . . . .	183
5.14.3 Reasoning About Uncertainty . . . . .	185
5.14.4 Absolute Uncertainty . . . . .	186
5.14.5 Relative Uncertainty . . . . .	189
5.15 UA-QM Optimization . . . . .	190
5.16 UA-QM Execution . . . . .	197
5.16.1 Runtime Infrastructure . . . . .	198
5.16.2 Belief Space Handling . . . . .	199
5.17 UA-QM Conclusion . . . . .	200
6 Conclusion and Future Research Directions . . . . .	202
6.1 Summary . . . . .	202
6.2 Future Research Directions . . . . .	203
6.2.1 Security Extensions in DSMSs . . . . .	203
6.2.2 Tagging Extensions in DSMSs . . . . .	204
6.2.3 Query Mesh Extensions . . . . .	205
6.2.4 Generalized Punctuation (GPUNCT) . . . . .	207
LIST OF REFERENCES . . . . .	209
VITA . . . . .	226

## LIST OF TABLES

Table	Page
3.1 Example of query rewriting. . . . .	58
3.2 Default experimental parameters. . . . .	61
3.3 Dynamic properties of security policies. . . . .	62
4.1 Traditional tags versus streaming tags. . . . .	80
4.2 Overview of key TAG-QL statements. . . . .	85
4.3 Examples of tag-oriented queries. . . . .	88
4.4 Examples of tag-aware queries. . . . .	92
4.5 Tag examples used in the experiments. . . . .	99
5.1 Defaults used in the experiments. . . . .	132
5.2 Distribution statistics. . . . .	133
5.3 Default experimental parameters. . . . .	167
5.4 Distribution statistics and parameters. . . . .	170
5.5 Selectivity intervals for various subsets. . . . .	187

## LIST OF FIGURES

Figure	Page
1.1 Simple view of a typical streaming environment. . . . .	2
1.2 Example 1: Ubiquitous healthcare application. . . . .	4
1.3 Example 2: Location-based application. . . . .	6
1.4 Traffic example with tags on location updates. . . . .	7
1.5 Conceptual differences between “Streams 1.0” and “Streams 2.0” systems.	8
1.6 Overview of a query mesh solution. . . . .	11
1.7 Query mesh overview. . . . .	12
2.1 Query mesh versus other optimization techniques. . . . .	22
3.1 Conceptual idea of security-aware continuous query processing (SA-CQP).	41
3.2 Overview of <i>FENCE</i> architecture. . . . .	42
3.3 General security punctuation schema. . . . .	46
3.4 Enforcement of security punctuations in an “immediate” and “deferred” manner. . . . .	48
3.5 Query processing with <i>sps</i> . . . . .	53
3.6 <i>SFA</i> -based SA-CQP. . . . .	54
3.7 <i>SS<sup>+</sup></i> execution in <i>SFA</i> . . . . .	55
3.8 Processing of <i>sp</i> in <i>SS<sup>+</sup></i> . . . . .	55
3.9 Processing of “immediate <i>sps</i> ”. . . . .	56
3.10 <i>QRA</i> -based SA-CQP. . . . .	58
3.11 SA-CQP using <i>QRA</i> . . . . .	59
3.12 Algorithm for query rewriting. . . . .	60
3.13 Experimental results. . . . .	67
3.14 Security enforcement overheads. . . . .	69
4.1 Stream Tag Framework (STF) overview. . . . .	79

Figure	Page
4.2 <i>Tick-tag</i> schema. . . . .	82
4.3 Interpretations based on tag signs. . . . .	83
4.4 Tag-oriented algebra (examples). . . . .	89
4.5 Tag-aware join example. . . . .	93
4.6 Tagger operator algorithm. . . . .	95
4.7 Tag join operator algorithm. . . . .	97
4.8 Experimental Queries. . . . .	98
4.9 Tag properties. . . . .	99
4.10 Cost of tagging operator. . . . .	101
4.11 Comparison of alternatives (output rate) . . . . .	103
4.12 Comparison of alternatives (memory) . . . . .	103
4.13 Cost of tag join. . . . .	104
4.14 Cost of tag-aware join. . . . .	105
5.1 Optimizer producing logical <i>QM</i> solution. . . . .	110
5.2 Core query mesh framework. . . . .	112
5.3 Lattice-shaped query mesh search space. . . . .	116
5.4 Optimal <i>QM</i> search algorithm . . . . .	120
5.5 <i>II</i> search strategy for <i>QM</i> . . . . .	124
5.6 <i>SA</i> search strategy for <i>QM</i> . . . . .	125
5.7 Hybrid search strategy for <i>QM</i> . . . . .	127
5.8 Query mesh execution example. . . . .	128
5.9 <i>QM</i> physical runtime infrastructure. . . . .	129
5.10 Physical instantiation of the <i>Self-Routing Fabric</i> infrastructure. . . . .	129
5.11 Experimental distributions. . . . .	133
5.12 <i>QM</i> with different start solutions. . . . .	135
5.13 Impact of start solutions on routes. . . . .	136
5.14 Effect of the search strategy. . . . .	137
5.15 Query mesh experimental results. . . . .	137

Figure	Page
5.16 Comparison of runtime overheads. . . . .	138
5.17 Overhead of runtime classification. . . . .	142
5.18 Virtual and real concepts in Query Mesh. . . . .	147
5.19 Concept drift “spectrum”. . . . .	150
5.20 Self-tuning Query Mesh framework. . . . .	152
5.21 ST-QM process flow. . . . .	154
5.22 Route-driven sampling. . . . .	155
5.23 QM concept drift detection. . . . .	158
5.24 QM tuning recommendations. . . . .	161
5.25 Physical execution of QM adaptivity. . . . .	164
5.26 Experimental distributions. . . . .	166
5.27 Experimental results. . . . .	169
5.28 ST-QM overhead. . . . .	174
5.29 Overhead when no adaptation is needed. . . . .	175
5.30 Geo-social networking query example. . . . .	179
5.31 <i>UA-QM</i> architecture. . . . .	183
5.32 Uncertainty scenarios in <i>QM</i> . . . . .	185
5.33 Symmetric modeling of <i>SINs</i> and <i>CINs</i> . . . . .	187
5.34 Belief function for a relative uncertainty. . . . .	188
5.35 Cases for uncertainty intervals’ overlaps. . . . .	190
5.36 Example of computing SBF. . . . .	191
5.37 <i>UA-QM</i> optimization approaches. . . . .	191
5.38 <i>UA-QM</i> Optimization. . . . .	192
5.39 Computing uncertain routes. . . . .	192
5.40 Computing selectivity intervals. . . . .	193
5.41 Conceptual idea of uncertain routes and classifier. . . . .	193
5.42 Merging point estimate into a <i>SIN</i> . . . . .	194
5.43 Computing uncertain classifier. . . . .	195



Figure	Page
5.44 Merging uncertain query meshes. . . . .	197
5.45 Resolving belief functions (route example). . . . .	198
5.46 <i>UA-QM</i> execution. . . . .	201
6.1 Operator queue management. . . . .	206
6.2 Switch and operator-assisted tuning. . . . .	207

## ABSTRACT

Nehme, Rimma Ph.D., Purdue University, August 2009. Efficient Query Processing for Rich and Diverse Real-Time Data. Major Professors: Elisa Bertino and Elke A. Rundensteiner.

In recent years, data streams have become ubiquitous as technology is improving and the prices of sensor, location-tracking and portable devices are falling. Examples of streaming data include observations from sensor networks, location updates from GPS devices, measurements from health monitoring devices, status updates, comments and self-expressions from users on the web, e.g., by “twittering”. The current state-of-the-art Data Stream Management Systems (or short DSMSs) typically consider a very simple streaming environment, where data streams transmit only data tuples, and based on these arriving data tuples, continuous queries are evaluated on the server. For execution of a continuous query, typically, a *single* execution plan (based on the latest overall statistics) is employed for processing *all* arriving data.

We believe that such “first-generation” DSMSs (or as we also refer to them “Streams 1.0” systems), while enabling users and applications to pose queries over data streams, are, however, ill-equipped to support many of the functionalities crucial to the newly-emerging streaming applications, e.g., ubiquitous healthcare or geo-social networking. Motivated by the growing trend of such new stream-based applications, in this thesis, we propose to equip DSMSs with several important functionalities, namely:

1. the access control enforcement for security of streaming data
2. the tagging of streaming data for producing “richer” and more meaningful results, and

3. the diversity-aware query processing for efficient processing of queries, where subsets of data may exhibit distinct statistical properties.

For each of the above features, we provide the concrete problem definition, the motivating examples, develop and analyze algorithms, and present the experimental results using a general-purpose DSMS prototype. We believe that the ideas presented in this thesis can significantly contribute the development of the “next generation” of DSMSs – or the so-called “Streams 2.0” systems.

## 1 INTRODUCTION

In this chapter, we begin in Section 1.1 by highlighting the bigger picture of streaming technology. Next, in Section 1.2, using the real-life application examples, we motivate the need for supporting security and tagging of streaming data, and the necessity for instrumenting DSMSs with query processing techniques designed to efficiently handle diverse data. In Section 1.3, we present the set of novel features for the “next generation” of DSMSs – the so-called “Streams 2.0” systems. In Section 1.4, we give an overview of our approach to support the proposed features. We summarize the main contributions of the dissertation in Section 1.5. Finally, Section 1.6 gives the outline for the rest of the dissertation.

### 1.1 Data Stream Management Systems

Database Management Systems (DBMSs) have played a central role in information technology over the last three decades. DBMSs simplify the storage, maintenance, and analysis of large datasets. When users or applications need to handle significant amounts of data, they first load the data into a DBMS. Once loaded into the system, the data or selected portions of it can be retrieved on demand easily and efficiently through queries to the DBMS. Such queries are expressed in query languages like SQL [1]. A query submitted to a DBMS is processed over the current dataset stored in the DBMS. The results computed for the query are then returned back to the corresponding user or application.

While conventional DBMSs are designed to process queries over *finite* stored datasets, many modern applications need to process *data streams* that consist of data elements generated in a continuous unbounded fashion. Examples of such applications include health-monitoring applications that process streams of tuples describing

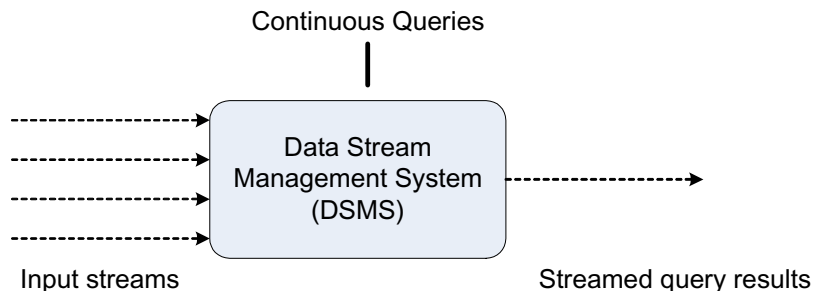


Figure 1.1. Simple view of a typical streaming environment.

the vital signs of patients, financial monitoring applications that detect patterns over the stock-ticker streams, environmental monitoring applications that track physical phenomena using the streams of observations generated by sensors, and many others. These applications have new data management needs that arise from the continuous, unbounded, rapid, and time-varying nature of data streams [2]. Since conventional DBMSs are ill-equipped to fulfill the needs of these applications, a new class of systems – called *Data Stream Management Systems* (or short DSMSs) [3–9] – have been developed by the database research community as well as by several commercial vendors [10–14] to satisfy the requirements of stream-based applications. DSMSs enable users and applications to pose queries over infinite data streams and to receive results in near-real-time. These queries tend to be long-running, since data arrives continuously, and are called *continuous queries*.

Figure 1.1 shows a high level overview of a typical data stream environment. Continuous streams consisting of data tuples, e.g., sensor measurements, latest stock prices, or moving objects’ location updates, are collected by monitoring devices and sent to a DSMS for processing. In addition to the input streams, the DSMS may also maintain conventional stored data, e.g., in the case of location-based services, this can be a table storing the road network of a city, or in the case of a financial application, this can be a table storing information about the companies and what they do for business. Users or monitoring applications register continuous queries over the input data streams and the other data managed by the DSMS. For example, a registered continuous query for a location-based application may specify a set of

conditions that signal a potential road congestion, or a pattern for dangerous driving behavior (to alert the nearby police and prevent potential accidents). The continuous queries registered on the DSMS are evaluated on the new data tuples when they arrive in the input data streams. Query results are usually output directly to users or applications in the form of continuous data streams as well, as shown in Figure 1.1.

## 1.2 Emerging Real-life Streaming Applications

In this section, we present several real-life application examples that call for new features that the current state-of-the-art DSMSs are ill-equipped to support. Based on the current growth and the popularity trends of these applications, we believe that they are here to stay and grow, and thus, it is crucial that the DSMSs become instrumented with the much-needed features described in this section.

*Example 1. Ubiquitous Healthcare:* In ubiquitous healthcare [15–17], tiny sensors are attached to a patient to gather information on bodily conditions such as temperature, heart rate, blood pressure, chemical levels, breathing rate and volume, and almost any other physiological characteristic that provides information that can be used to diagnose health problems. These sensors are worn on (or possibly implanted in) the body, or installed in patients’ homes and workplaces. A family doctor can remotely monitor patients, and provide general health advice and remote diagnosis while saving the patients a trip to the doctor’s office, and thus providing cheaper and better healthcare services. Figure 1.2 visually depicts the main idea of a ubiquitous healthcare application.

While remote health monitoring provides a lot of convenience, at the same time, detailed body sensor data may be combined with data from infrastructure sensors, which can provide a “life log” or an “activity diary” of the patients in real-time. If such sensitive personal data is shared among interested parties – such as employers, insurance companies, drug companies and the government, to name a few – the possibility of abuse or discrimination can be great.

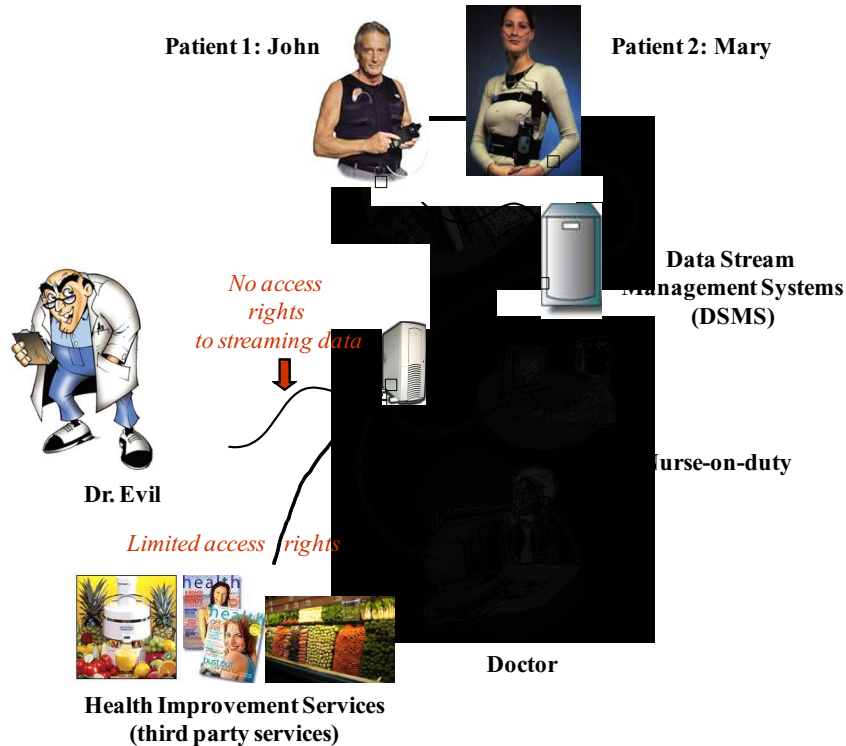


Figure 1.2. Example 1: Ubiquitous healthcare application.

The patients, transmitting their health data through their monitoring devices, should have the ability to continuously regulate who can access their real-time health information *at all times* and adjust it as necessary, based on time, location, observed data values, psychological state or any other reasons. The United States Health Insurance Portability and Accountability Act (HIPAA) empowers patients to specify their access control preferences on their health information, and the doctor's office must not release that information to any entity to whom the patient does not want his or her information to be disclosed. Furthermore, a patient should be able to ask the doctor's office at any time (including *in real-time*, e.g., from his or her portable device) to show that they indeed did not release his or her information to the parties the patient did not approve.

In the near future, the goal of ubiquitous health applications is to provide ubiquitous care, where patients receive tele-prescription and tele-infusion of drugs (e.g., a low glucose level may cause automatic injection of insulin for a diabetic patient) re-

motely and immediately. In order to achieve accountability, it is important to know who has gained access to the sensitive health data, when and under what circumstances.

Another important factor to consider is that patients are involved in various activities in their every-day lives, which might cause changes in their typical health data values. Consider a patient carrying a health monitoring device that measures his heart rate. When a heart rate becomes abnormal, the doctor gets alerted about it. The patient, to prevent unnecessary concerns, can attach a “tag” to his real-time streaming measurements stating “Running”. The doctor or a nurse knows (based on the tag) what is causing the change in the health measurements. This helps prevent second-guessing and avert issuing unnecessary alerts. Similarly, if a diabetic patient’s blood sugar rises slightly above the norm, and she attaches a tag to her streaming data that she is at a birthday party, the doctor may not need to be alerted, if the patient’s blood sugar has gone up only slightly, and there are friends and relatives around, in case of an emergency. In general, if patients were able to attach additional semantics (information) to their real-time streaming health data, the DSMS may exploit this extra-information in possibly averting unnecessary emergency situations and produce more informative results. The more informative results can facilitate in providing better and less expensive health services.

*Example 2. Location-Based Applications:* Recent improvements in location-based technologies and the drop in prices of location-tracking devices have spurred a new wave of mobile applications, providing location-based services and enabling *geo-social networking* scenarios e.g., [18, 19]. Figure 1.3 shows an example of a location-based application, where a cell-phone user is searching for nearby quiet coffee shops that currently have promotions (sales or discounts). Similarly, a user may request to find any of her friends in the vicinity, to possibly invite them to join her for a coffee.

Location-based applications naturally raise security and privacy concerns. Users consider their physical location and travel patterns highly privacy-sensitive and demand solutions that are able to protect their information. Therefore, it is essential to





Figure 1.3. Example 2: Location-based application.

provide support for users to be able to frequently change their access control policies based on their preferences, to restrict who can “see” their real-time information, e.g., where they are, whom they are with, or what they are doing.

To motivate the need for being able to attach additional semantics to a stream transmitting location-based information, consider another example (a traffic scenario) depicted in Figure 1.4. An accident occurs on a highway which causes extensive traffic jams. People stopped far away from the scene of the accident wonder what is causing the stopped traffic: an accident, a road construction, or a “curiosity factor”? A driver close to the accident attaches a tag to his location update describing what he is observing: “Accident, 2 cars, Near Exit 12”. Using this tag information, other drivers may determine how to proceed: take the nearest exit, notify others about their delay, or possibly even help if medical assistance is needed.

### 1.3 Features for the Next Generation of Data Stream Management Systems

Given that in the last several years the number of data sources that continuously generate data has increased significantly, the research on data streams has grown substantially. Based on current trends, we can clearly observe that the newly-

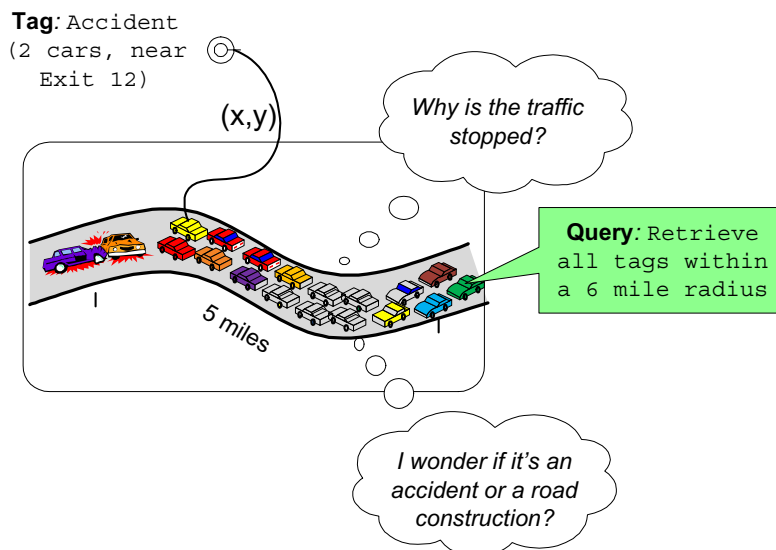


Figure 1.4. Traffic example with tags on location updates.

emerging applications are entering a new, more “social” and “participatory” phase, where users are not passive observers, but rather active contributors of real-time information. These trends lead to a feeling that the data stream management research needs to enter a “second phase” to support the needs of these emerging applications (see Figure 1.5), where in addition to sending their “regular” streaming data, users can state who can access their information and under what circumstances and adjust it in real-time. Furthermore, users should be able to attach additional information to their real-time data as well as other (arbitrary) streaming data, thus further enriching the streaming data with an additional semantics.

We can observe from the examples in Section 1.2 that the newly emerging applications call for new “user-centric features” inside DSMSs, namely the provisions for security (specifically, the access control) and the ability to attach additional semantics to their real-time streaming data by tagging. As a result of the above functionalities and often, based on data content alone, data streams may exhibit characteristics, where certain data subsets (based on either the content, the security policies or the associated tags) may be quite distinct, i.e., have unique statistical properties – and each such distinct subset can benefit from a different, tailored to its local statistics,

“2.0” is really about **people**



“Streams 1.0”		“Streams 2.0”
Send: - data only	Data Provider Mode	Send: - data - security policies - tags attached to data
Query: - data only	Query Specifier Mode	Query: - data and tags separately - data with respect to tags - tags with respect to data
--	Security	Security-aware query processing with respect to: - data access control policies - query authorizations
Single plan for all data (“diversity-oblivious”)	Query processing	Different plans for distinct subsets of data (“diversity-aware”)

} contributions of this thesis

Figure 1.5. Conceptual differences between “Streams 1.0” and “Streams 2.0” systems.

query execution plan. Clearly, current DSMSs either provide little or no support for the above-mentioned functionalities [20].

Figure 1.5 roughly characterizes the conceptual differences between the so-called “Streams 1.0” and “Streams 2.0” DSMSs. Similar to the emergence of Web 2.0 on the Internet [21], where novel applications and services, such as blogs, video sharing, social networking and podcasting have contributed to a more dynamic, user-driven, and socially connected Web, we envision that DSMSs need to be instrumented to provide similar in spirit “user-centric” functionalities with regard to streaming (real-time) data. In the rest of the section, we describe the features we address in this thesis,

namely the access control, the tagging, and the diversity-aware query processing in the context of data stream environments.

### 1.3.1 Access Control for Streaming Data

One of the biggest challenges in dynamic data stream environments in the context of security is access control enforcement – the ability to permit or deny a request to perform an operation (e.g., a read operation on a data tuple). Given the *long-running* nature of continuous queries, the content of the streaming data and along with it its “sensitivity” may change frequently over the lifetime of query execution. Furthermore, continuous queries on the server may also experience frequent changes in their access control privileges (access authorizations), while they are being executed. Such changes in security privileges may be due to mobility and varying context of the users receiving the results of continuous queries. For example, query results may be accessed via mobile phones, PDAs or iPhones from any place and at any time. As such, the users sending their streaming data can be rightly concerned about a possible unauthorized access to their real-time information and a potential violation of their privacy. One of the major challenges in this context comes from the fact that the security policies of both data and queries can be *concurrently* very dynamic. Enforcing security, while still guaranteeing near real-time response to queries presents a great challenge in a DSMS.

### 1.3.2 Tagging Streaming Data

Tagging in the general domain is known by a few different names, such as content tagging, collaborative tagging, and social tagging [22]. Informally, a tag is a relevant keyword or term associated with or assigned to a piece of information (a stream, a tuple, an attribute in a tuple, a particular value, etc.), that describes the item and enables the keyword-based classification and search of information.

Tags can be useful in many applications. Tags on streaming data can enrich existing stream-based applications, e.g., [23–25], and can enable and inspire novel useful services as described in Section 1.2. Other ways of leveraging tags associated with streaming data may include: (1) *Stream data tracking*, where tagged objects can be located and tracked unambiguously. (2) *Creation of rich user profiles*, where information about a user’s interests, mood, observations, and character can be revealed based on the tags employed by him or her in real time, and used in privacy preservation or tailored services. (3) *Exploration and browsing of streaming data*, which can be achieved by exploiting tags as a navigation mechanism allowing users to find related streaming data based on the tags (see Section 4.4.1). (4) *Social communication*, where by allowing other people to tag a specific subset of real-time data with their own tags, one can find out what different people think about the same piece of information. For instance, one scientist’s opinion (expressed via a tag) on real-time measurements in an experiment might vary significantly from the tags attached by other researchers. In general, we envision stream tagging being useful in almost any application in which streams are produced or consumed.

### 1.3.3 Diversity-Aware Query Processing

Most modern query optimizers determine a *single* “best” plan at compile time for executing a given query [26]. The execution cost for alternative plans is estimated and the one with the overall cheapest cost is chosen. The cost typically is estimated based on the *average statistics of the data as a whole*, as the objective is to find *one* plan for all data. Such optimization approach largely relies on the *uniformity* of attribute values. As a result, the single plan approach ignores the fact that various data tuples from the same stream may have distinct statistical properties (e.g., frequencies, correlations, etc.). While in some cases such simple and “monolithic” approach to execution plan selection is adequate, the strong assumption of uniformity is often

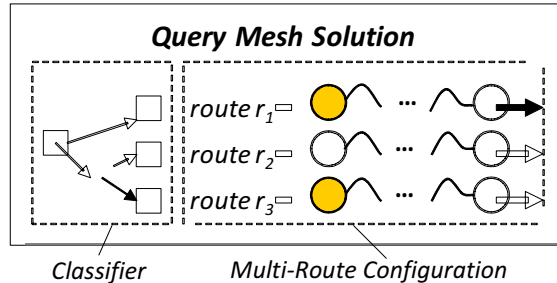


Figure 1.6. Overview of a query mesh solution.

unrealistic in practice and, in fact, rather unlikely for unbounded streams of data, which potentially could seriously limit query performance [20, 27–29].

As the third part of this thesis, we describe a novel data-diversity-aware query processing approach using our proposed *Query Mesh* (or *QM*, for short) model. The conceptual idea of a query mesh is visually depicted in Figure 1.6. *QM* takes a practical middle-ground strategy between the two query optimization extremes – the “monolithic” single execution plan solutions, used in nearly all commercial DBMSs [30–32] and the extremely reactive Eddies and its descendants [33, 34]. A query mesh consists of two complementary components: (1) a set of *pre-computed plans* (or routes), where each plan is optimized for a subset of data with certain statistical properties, and (2) a *classifier* component to determine which data tuples should be processed using which of the existing routes<sup>1</sup> (see Figure 1.7 for a more detailed view).

#### 1.4 Overview of Our Approach

The tagging and security features can be implemented in a Data Stream Management System using a “layered” approach (e.g., outside the query processor). For instance, [35–37] are examples of this type of implementation in the context of security. However, this may severely limit the performance of continuous queries, and such approach does not give enough flexibility in optimizing the issued queries.

<sup>1</sup>In the paper, we refer to the combination of a classifier and a set of execution routes – a *QM solution*.

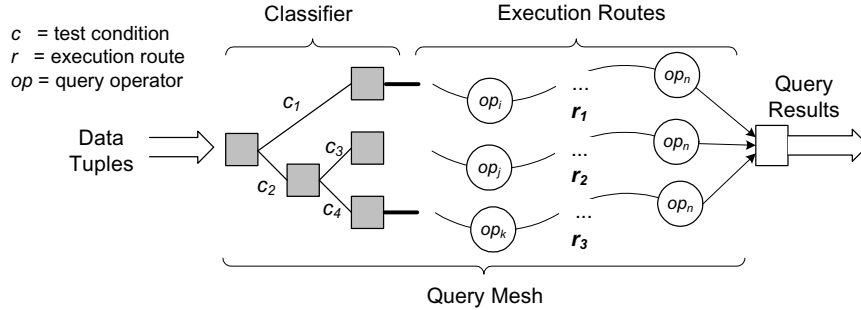


Figure 1.7. Query mesh overview.

The most crucial feature of any DSMS is to produce query results in near-real-time. In many scenarios, if a query result is slightly delayed, it becomes useless by the time it reaches the user. By adding the provisions for tagging and security in DSMS, we increase the overhead in the system. Thus, it is extremely important that, while extending the capabilities of a DSMS, the results are still produced in a timely and an efficient manner. Our approach is to integrate tagging and security processing into continuous query processing and optimization inside DSMS, which allows more efficient query execution. Specifically, we introduce novel streaming metadata objects that are embedded inside data streams and encapsulate the security and the tagging algorithms in physical query operators that can be part of query execution plans. We call these new operators security-aware and tag-aware and tag-oriented operators. Tag-oriented operators are the novel operators that explicitly operate on the streaming tags (e.g., tag selection, tag join), whereas tag-aware operators are regular continuous query operators (selection, join) that have been instrumented to correctly propagate streaming tags in the query pipeline (Section 4.4). Integrating security and tagging metadata and algorithms into the continuous query processor in DSMS has the following advantages:

- The security-aware and tag-aware query is under the optimizer's control; best algorithms and strategies can be chosen based on the estimated cost and other execution environment variables.

- The security-aware and the tag-aware operators can be shuffled with other operators in a query evaluation plan for better performance (e.g., pushing down security or tagging predicates). This flexibility is not possible when these features are implemented outside the query processing mechanism.
- The security and tagging functionality is general enough and highly applicable to many queries in different contexts. This native support for security and tagging greatly simplifies the development of many emerging applications, by pushing this logic inside the streaming engine.

Finally, our diversity-aware query processing approach – the *Query Mesh (QM)* – is general and can be easily integrated with security and tagging mechanisms. Query mesh can support data-diversity based on data content or the security policies or the attached semantic tags for efficient continuous query processing.

## 1.5 Contributions

The main contributions of this dissertation are as follows:

- *Security*. We address the problem of continuous access control enforcement in dynamic data stream environments, where both data and query security restrictions may potentially change in real-time. The distinguishing characteristics of our solution include: (1) the *stream-centric* approach to model dynamic security policies, (2) the *symmetric security model* for both continuous queries and streaming data, and (3) the *security-aware query processing methods*, that can optimize the execution based on data-related as well as security-related selectivities. Specifically, the contributions can be summarized as:
  - *Symmetric Security Punctuation Model*. We model both data and query security restrictions symmetrically in the form of security metadata, called “security punctuations”, streaming together with the data instead of being persistently stored on the server. We distinguish between two types of



security punctuations, namely, the data security punctuations (or short, *dsps*) which represent the access control policies of the streaming data, and the query security punctuations (or short, *qsps*) which describe the access authorizations of the continuous queries running on the server.

- *Security-Aware Query Processing.* For efficient execution of continuous queries, we propose security-aware query processing methods, namely: (1) the Security Filter Approach (or short SFA), and (2) the Query Rewrite Approach (or short QRA). We discuss the pros and the cons of both of these methods.
- *Implementation and Experiments.* We have implemented our solution in a prototype DSMS and have carried out extensive performance evaluation. The results of our experimental study show that our proposed approach has low overhead and can result in great performance benefits compared to alternative security solutions for streaming environments.
- *Tagging.* We show that supporting tagging as a DSMS functionality enables many interesting and useful applications. We show the advantages of supporting tagging on the query operator level inside the continuous engine in contrast to implementing tagging using alternative options. The contributions of our tagging solution can be summarized as follows:
  - *Streaming Tag Model.* We describe the streaming tag metadata model for tagging various streaming objects (e.g., tuples, data values, etc.). Tags are embedded inside streams and support a wide variety of user-based semantics.
  - *Tag Query Language.* We introduce a Tag Query Language (or short TAG-QL) that enables declarative specification and querying of streaming tags.
  - *Tag Query Processing* In DSMS equipped with tagging, users can attach and explicitly query streaming tags. We propose a *tag-oriented query algebra* that enables this functionality. We have also extended support for

implicit querying of tags on DSMS, where continuous query results are enriched with the tags of the base data. We describe the extensions to the continuous query algebra to enable *tag-aware query processing* and support correct propagation of tags in the query pipeline.

- *Implementation and Experiments.* To illustrate the feasibility, we have implemented our approach in a prototype DSMS. Our experimental analysis shows scalability and benefits of the streaming tag approach, and the costs associated with tag-awareness.
- *Diversity-Aware Query Processing.* As the third part of this dissertation, we introduce a novel *Query Mesh (QM)* model, which addresses the problem of efficient query processing when data is diverse. Instead of forcing all data to be processed by the same single plan, a query mesh solution effectively supports the concurrent usage of multiple plans to evaluate a query. The contributions of our proposed Query Mesh approach are as follows:
  - *Query Mesh Model.* *QM* employs a practical middle-ground strategy between the two query optimization extremes – the solutions that employ a “monolithic” single execution plan strategy for all input data, e.g., nearly all commercial DBMSs [30–32], and the systems like Eddies that employ a fine-grained “plan-less” approach, where instead of predetermined plans, at runtime the Eddy operator determines, one-at-a-time, the next operator, that the tuples must visit for processing [34]. *QM* provides the middle-ground by using *multiple pre-computed plans*, each optimized for a subset of data with certain statistical properties, and the *classifier* component to determine which data subsets should be processed by which of the pre-computed routes. The *QM* framework, implemented in a continuous query processing engine [8], has been shown to be very effective for both real and synthetic data compared to the single plan and the Eddy-based query processing alternatives.

- *Query Mesh-Based Optimization.* We formulate the complexity of the *QM* search space (Section 5.2.5), and develop the algorithm *Opt-QM* that finds optimal query meshes. *Opt-QM*, however, may be not feasible in practice due to its exhaustive nature when enumerating the search space. As viable alternatives, we propose several effective cost-based search heuristics to find good quality *QMs* efficiently.
- *Query Mesh-Based Execution.* For efficient query execution, we propose the *Self-Routing Fabric (SRF)* infrastructure. The novelty of *SRF* lies in its support for execution of multiple routes in parallel without physical constructing their topologies and without using a central router operator like Eddy. We describe other advantages of *SRF* in Section 5.3.
- *Self-Tuning Query Mesh.* We present a *Self-Tuning Query Mesh* infrastructure (or short *ST-QM*) that continuously adapts the query mesh solution to changing data subsets’ characteristics and to system conditions, e.g., memory, CPU resources availability. The fundamental challenge for self-tuning query mesh is the problem of determining the discrepancy between the previously learned query mesh model and the current model of the new data, what we denote as *optimization concept drift* problem.
- *Uncertainty-Aware Query Mesh.* To address the issue of uncertainty (about data input rate, operator selectivities, attribute values and their distributions) that naturally arises during query optimization in the streaming context, we present uncertainty-aware extension to the query mesh model, called *Uncertainty-Aware Query Mesh* (or *UA-QM*, for short). The goal of *UA-QM* is two-fold: (1) to model and measure various types of uncertainty to represent real-life scenarios in streaming environments more accurately and (2) to process data in an uncertainty-aware and multi-plan fashion.
- *Implementation and Experiments.* We thoroughly evaluate *QM* approach through experiments comparing it to the state-of-the-art techniques, namely

the systems using a single plan computed a priori and the systems discovering routes at runtime. Our results show that *QM* results in substantial performance improvements over the competitor systems. We also demonstrate *QM* effectiveness by measuring its runtime overhead.

## 1.6 Summary and Outline

In this section we have presented the real-life motivating application examples calling for new features to be added to Data Stream Management Systems (DSMSs), namely the access control for protecting sensitive streaming data, the tagging for attaching additional semantics to streaming data, and the data diversity-aware continuous query processing. Efficient handling of security, tagging and data diversity in DSMSs enables many key stream-based applications including location-based services, ubiquitous healthcare, environmental monitoring and many others. In this chapter, we gave an overview of our proposed approaches to address these problems and briefly summarized our contributions.

The rest of this dissertation is organized as follows. Chapter 2 highlights the related work and provides the necessary background for this dissertation. Chapter 3 presents our solution for access control enforcement in data stream environments. Chapter 4 presents our solution for tagging of streaming data. Chapter 5 describes our diversity-aware query processing mechanism using the query mesh model. Finally, Chapter 6 concludes by giving a summary of the dissertation and outlines extensions to the work presented in this dissertation.

Parts of this dissertation have been published in conferences and have been submitted to journals; the security for data streams and its details have been published in ICDE-2008 [38], the diversity-aware query processing in EDBT-2009 [27], the tagging and the extensions to the security and the query mesh models have been submitted to several publication venues [39–41]. The stream-centric approach to security in a DSMS will be demonstrated in ACM SIGMOD-2009 [42].

## 2 BACKGROUND AND RELATED WORK

In this chapter, we present an overview of the state-of-the-art research related to the work in this dissertation. This chapter is organized as follows. Section 2.1 describes different prototypes of DSMSs from both the academia and the industry. Brief survey of query optimization approaches is given in Section 2.2. In Section 2.3, we discuss adaptive query processing algorithms techniques for efficient processing of long-running continuous queries. Section 2.4 surveys various metadata mechanisms in current DSMSs. Section 2.5 describes related work in the context of learning approaches as applicable to our diversity-aware query processing approach. In Sections 2.6, we describe the related work in security, in particular, in access control. Section 2.7 provides the related work and necessary background on tagging. Finally, Section 2.8 summarizes the chapter.

### 2.1 Data Stream Management Systems

A number of general-purpose DSMSs have been proposed in the literature that target different application domains, where continuous data streams arise and queries must be evaluated continuously, including Aurora [3], Gigascope [5], CAPE [8], NiagaraCQ [9], Nile [7], STREAM [43], and TelegraphCQ [4]. All of these systems share similar goals, however, each of them has distinct characteristics and algorithms to achieve these goals.

While STREAM supports a declarative language for specifying arbitrarily complex continuous queries, Aurora supports a workflow-style boxes and arrows interface for specifying continuous queries. Aurora has limited support for adaptive query processing, but richer support for distributed query processing and tolerance to failures [44].

The TelegraphCQ system is built on the Eddies adaptive query processor [34]. Eddies is an integrated optimizer, executor, and operator scheduler<sup>1</sup>. On the other hand, most DSMSs take a modular approach to query processing, having separate optimization, execution, and scheduling components. Unlike many DSMSs, which were developed from scratch, TelegraphCQ was developed as an extension to the PostgreSQL relational DBMS [45]. The pros and cons of building a DSMS on top of an existing DBMS are documented in [46].

Nile [7] has a rich support for exploiting sharing of data and computation while processing multiple continuous queries concurrently. To support the efficient and correct pipelined execution of sliding window queries over multiple data streams, Nile employs the *Negative Tuple Approach* [7]. Negative tuples are the tuples that are generated whenever a tuple expires out of a sliding window. Each operator in the pipeline reacts differently whenever it receives a negative tuple to counteract the effect of the corresponding positive tuple that just expired out of the window. Although negative tuples guarantee correct execution of query pipelines, they induce performance overhead. Nile provides several optimization techniques to reduce the performance overhead induced by negative tuples. Another distinct feature of Nile is its predicate windows [47]. In contrast to sliding windows that limit the focus of queries on streams to the most recent tuples, predicate windows can select tuples of interest that meet a certain select or join predicate.

CAPE [8] focuses on adaptive query processing at many levels. In particular, CAPE emphasizes the following features: (1) Intra-operator adaptivity, where CAPE exploits metadata knowledge about the data streams to reduce resource usage and improve execution efficiency of operators. (2) Plan-level adaptivity, where CAPE supports online query re-optimizations and plan migration. (3) System-level adaptivity, where CAPE supports adaptive distribution of the query plan among multiple machines for load balancing.

---

<sup>1</sup>We discuss Eddies in more detail in Section 2.3.

GigascopE is a DSMS tailored to the network monitoring domain [5]. GigascopE supports a declarative query language (GSQL) which is less expressive than SQL [1] or CQL [48]. GigascopE also takes a two-level approach to query processing, where sub-queries are pushed down to the network interface level to eliminate unneeded input stream tuples as quickly and efficiently as possible.

Apart from DSMSs, continuous queries are also used in content-based filtering and publish-subscribe systems, e.g., [49–53]. Two systems, OpenCQ [54] and NiagaraCQ [9], support continuous queries for monitoring persistent data sets spread over a wide-area network, e.g., web sites over the Internet. Unlike many DSMSs supporting more expressive declarative languages for continuous queries, both OpenCQ and NiagaraCQ support continuous queries specified in an Event-Condition-Action (ECA) format typically used for triggers. Furthermore, OpenCQ and NiagaraCQ lack many features present in general DSMS systems, e.g., flexible query plans and operator scheduler, adaptive processing of commutative filters and stream joins.

One way of thinking about materialized views [55] and triggers [56] in conventional in database management systems is as “continuous queries” that need to be processed whenever the base data changes or a monitored event happens [57]. However, materialized views and triggers are insufficient to meet the needs of stream-based applications easily and efficiently.

## 2.2 Query Optimization Techniques

Query optimization is a well-studied area, with most efforts primarily concentrating on the “monolithic” strategy of optimizing a *single query execution plan for all data* [58–62].

To better illustrate where our research on diversity-aware query optimization fits among the existing techniques, we classify these techniques along two dimensions: the timing of the optimization decision and the granularity of optimization (see Fig-

ure 2.1<sup>2</sup>). Some database systems determine query execution plans in advance (at compile time), while others forego pre-computed plans and “route” tuples on-the-fly (at runtime). We observe that these approaches have a close resemblance to network communication methods which can be described as: (1) *connection-oriented*, where a connection with the receiver is established in advance before passing any data, or (2) *connection-less*, where data is sent without establishing an a priori connection, and the next hop of a packet is determined by a router during the transmission at runtime [63]. The parallel between route optimization in networking and query optimization in databases is evident. Query operators (for a given query) can be viewed as a “network” and a query plan as a specific “route” through all operators in the “network”. Given this parallel, we classify the state-of-the-art techniques according to the optimization time dimension as “*route-oriented*” and “*route-less*” solutions<sup>3</sup>. Database systems, including major commercial DBMSs [30–32] that determine a query plan a priori, employ the *route-oriented* paradigm. Systems, like Eddy [34] and its descendants, that at runtime decide for every data tuple which operator should process it next, fall under the *route-less* category [34, 64, 65].

In practice, almost all route-oriented solutions pre-compute a *single route* leading to their main disadvantage – optimization coarseness [66, 67]. Having a fully established plan a priori, however, has a number of advantages: all tuples follow the same execution plan, which is fully known, and the execution tends to be “overhead-free”. Furthermore, data tuples’ sizes do not need to be extended to store any optimization-related metadata (e.g., tuple lineage).

On the other hand, “route-less” systems, like Eddies [34], tend to use multiple routes by default. Such systems based on observed conditions decide at runtime which operator should process a tuple next [34, 64], thus discovering different execution routes for tuples on-the-fly. Unfortunately the “optimization decision” (i.e., which operator should process a data tuple next) is made continuously, and in the worst case

---

<sup>2</sup>Figure 2.1 is not meant to be an exhaustive survey. It merely provides an intuitive idea of where we think our solution fits compared to the state-of-the art techniques.

<sup>3</sup>By “oriented”, we mean that routes (i.e., query plans) are established in-advance.



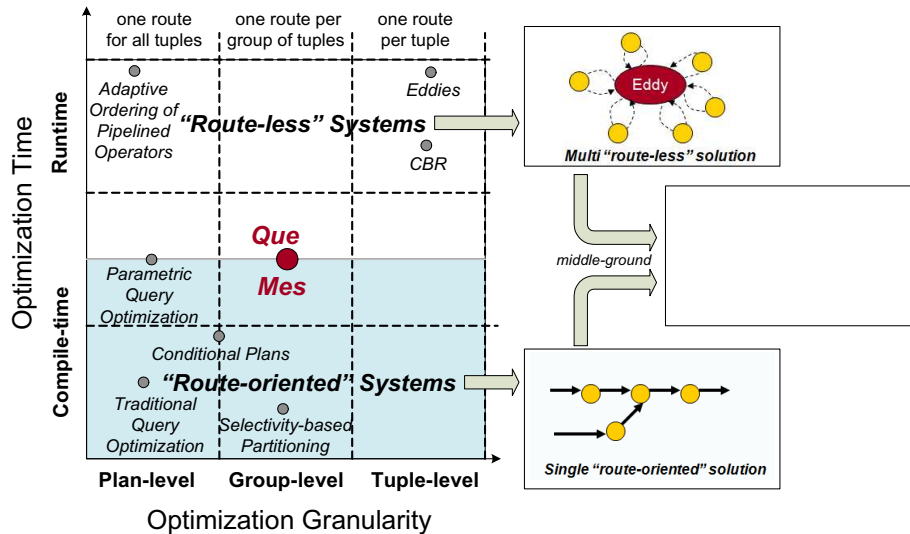


Figure 2.1. Query mesh versus other optimization techniques.

for every individual tuple, thus resulting in an unavoidable per-tuple route discovery overhead. Such systems typically do not exploit the observation that stable conditions tend to dominate query execution time, and that tuples with identical content or similar statistical properties are likely to be best served by the same route [28, 68]. Furthermore, individual tuples’ sizes tend to be larger, as tuples must now carry their individual “itineraries” (i.e., their lineage) depicting their current processing state.

In summary, optimizing too frequently as in the multi route-less approach may discover several plans but may also result in wasted resources. However, optimizing too coarsely as in the systems pre-computing a single plan may miss critical opportunities to improve query execution performance. We thus propose a practical middle ground approach between these two extremes in the form of a “*multi route-oriented*” solution called *Query Mesh* (or *QM*, for short).

Our work on diversity-aware query processing is related to the concept of *horizontal partitioning* [69]. Conceptually, the main idea here is to partition data so that different partitions can be processed using different execution plans. Selectivity-based partitioning scheme [29] – an instance of horizontal partitioning approach –

adopts a *divide-and-union* approach. A relation is partitioned according to selectivities, and subsequently the query is rewritten as a union of constituent queries over the computed partitions. The approach presented in [29], however, only focuses on the partitioning algorithm, rather than a systematic approach to generate different plans for different subsets of data – the focus of our work. In practice, the lack of a comprehensive system support for concurrent multi-plan execution may reduce (or completely cancel out) the effectiveness of such query processing strategy. The authors in [29] also do not address the issue of overhead associated with partitioning (classification) incurred at runtime when the data arrives and the important issue of concurrent multi-plan runtime execution efficiency. Our work addresses these issues and employs a very efficient novel “Self-Routing Fabric” data structure for efficient runtime execution.

*Conditional plans* [70] generalize serial plans by allowing different predicate evaluation orders to be used for different tuples based on the values of certain attributes. This class of plans can be beneficial when the attributes are highly correlated, and when there is a large disparity in the *acquisition* and evaluation costs of the predicates. Conditional plans primarily focus on often selecting a *single* and very *cheap* to *acquire* partitioning attribute, since query processing is done in the context of sensor networks. Such attribute is not necessarily the “best” splitting attribute in a more general context of DSMSs. In that respect, query mesh is a more general model that can employ an arbitrary number of attributes for partitioning (classification) of data. *QM*, being a general model, can exploit the data security and the tagging metadata (described in Chapters 3 and 4) to partition data into distinct data subsets.

A common problem with pre-computed solutions is that they might become inefficient over time. One approach to address inaccuracy or potential changes in an environment during query execution is through *eager re-optimization* and *runtime adaptation* of execution, e.g., [34, 71, 72]. Systems like IBM’s LEO (LEarning Optimizer) and more recently Microsoft SQL Server use monitoring and feedback to repair

incorrect cardinality estimations and statistics [73–75]. For a survey of adaptive query processing techniques, we refer the reader to [76].

Some works have proposed robust and uncertainty-aware solutions for query optimization e.g., [77–79], but they primarily focus on a *single*-plan strategy for query execution.

Proactive re-optimization in Rio [78] is a robust query optimization technique based on the concept of bounding boxes. Bounding boxes specify range of values that a parameter can take, thus representing uncertainty. The optimizer finds a (set of) plan(s) that behave well in the bounding box, and at run-time, if observed statistics fall inside the bounding box, the best plan in that box (based on the latest statistics) is chosen, else the re-optimization is invoked.

Error-aware optimization (eao) [80], similar to Rio, makes use of intervals over query cost estimates. Eao, however, focuses on memory usage uncertainty rather than selectivity uncertainty.

Babcock et.al. [77] tackle cardinality estimation uncertainty and consider a probability distribution over possible selectivities instead of a point estimate of selectivity. Using probability distribution, the optimizer selects the appropriate query plan after considering the relative importance of predictability vs. performance preference of the user. Prior to optimization, the user selects the trade-off between the two goals of predictability and performance (which could be at odds sometimes) to find the appropriate query plan.

Chu et al. [81] describe the least expected cost optimization technique. Here, instead of finding the lowest cost plan for the expected values of the parameters, the optimizer attempts to find the plan that has the lowest expected cost over the different values the parameters can take. The goal here is to find a “conservative” plan that is likely to perform reasonably well in many situations, rather than a more “aggressive” plan that may work better if the cost estimate is accurate, but much worse if the estimates are slightly off.

Ioannidis et al. [82] present parametric query optimization method, whereby multiple alternative plans are identified at compile-time, after which an actual *single* plan is chosen at run-time, when the actual parameter values are known.

A large body of work has been dedicated to extend support for uncertain data inside databases [83–86] including several efforts in building systems for managing uncertainty [87–89]. While many of these works study efficient algorithms for query processing on uncertain data, none of them actually consider uncertainty in the query processing itself, which is the focus of our paper.

To the best of our knowledge, none of the existing works tackle the problem of uncertainty when multiple query execution plans are employed concurrently for processing of distinct subsets of data.

### 2.3 Adaptive Query Processing Techniques

Related to our work are several techniques from adaptive query processing [3,6,76]. Here, at different times, tuples may be processed differently, as data statistics or system environment change. Similar to compile-time optimization, most adaptive query processing works still focus on adapting a single query plan as data properties and system conditions change at runtime [73,78].

A very recent survey of systems and techniques for adaptive query processing is given in [90]. Previous work on adaptive query processing considers primarily traditional relational query processing. One technique, which is being incorporated into commercial databases, is to collect statistics about query subexpressions during execution and to use the accurate statistics to generate better plans for future queries [73,91]. Two other approaches [92,93] re-optimize parts of a query plan following a (blocking) materialization point, based on accurate statistics collected on the materialized subexpression.

Convergent query processing is proposed in [68,94]: a query is processed in stages, each stage leveraging its increased knowledge of input statistics from the previous

stage to improve the query plan. The algorithms proposed in [68,94] do not extend to continuous queries and provide no guarantees on convergence. Reference [95] explores the idea of moving execution to different parts of a query plan adaptively when input relations transferred from remote nodes incur high latency. The POP approach adds checks to conventional query plans in DBMSs to detect sub-optimality during query execution, invoking re-optimization if required [79].

The previously-mentioned Eddies architecture [4, 33, 34, 65, 96, 97] enables fine-grained adaptivity by eliminating query plans, by instead routing each tuple adaptively across operators that need to process it. Eddies [34], which can potentially adapt at the tuple granularity, is observed to mostly be using a single plan for nearly all tuples as also indicated in [28]. Closely related to the *QM* paradigm is the content-based routing (CBR) extension of Eddies [28] that considers not only properties of operators (such as their selectivities and backlog) but also the content of the data. CBR, an extension to Eddies, however inherits several problems associated with Eddies, such as expensive on-the-fly decision-making and often unnecessary tuple-level granularity of adaptivity. [33] adds batching to the Eddies routing to reduce the tuple-level routing overhead. This work differs from ours in that it is still “route-less”. Further, the batching process is still neither content- nor route-based: batches of  $k$  tuples (i.e., continuous chunks of tuples that happened to arrive together in time) are routed together to aim to reduce the rather significant overhead associated with Eddies. Our *QM* solution is more coarse-grained than Eddies in that at a given point in time one execution plan is followed by all input tuples for a continuous query.

Apart from Eddies, the CAPE [8] and Gigascope [5] DSMSs support adaptive query processing over data streams. CAPE supports adaptive processing at the level of operators, e.g., within a join operator, as well at the level of query plans, e.g., switching among different plans for a query. CAPE also supports adaptive placement of query plan fragments across machines in a parallel processing environment. The two-level query processor in Gigascope can adapt the partitioning of work between the two levels, based on the characteristics of the input streams.

## 2.4 Streaming Metadata

*Punctuations* as sub-stream delimiters inside data streams have been first presented in [98]. *PJoin* [99] and *PWJoin* [100] apply punctuations to achieve join optimizations on streaming data. [101] uses punctuation-like annotations to inject dynamic schema-knowledge into XML stream to facilitate query optimization and out-of-order processing. [102] uses punctuations for execution safety checking of continuous join queries (CJQs). Punctuation uses continue to expand beyond their original semantics – delimiting epochs in the stream.

The authors in [103] propose a feedback mechanism based on punctuations that flow against the stream direction, and carry an intent, as opposed to embedded punctuations [104, 105]. DSMSs have focused on collecting and distributing feedback information by statically placing monitors in the query plan and directly sending parameter changes to operators. The approach in [103] differs significantly in at least two aspects: (1) the use of punctuation to convey the feedback messages, and (2) giving operators the ability to create, consume, and propagate feedback, eliminating the need for centralized managers.

AT&T’s Gigascope DSMS includes a type of punctuations – called “heartbeats” – that signal the passing of time, but their work does not exploit punctuations as feedback mechanisms for optimization [106].

This thesis is the first work proposing to use punctuations as (1) security constraints to enact access control policies, (2) as semantic labels (tags) to attach additional semantics to streaming data, and (3) as routing “itineraries” to process various subsets of data using different execution plans.

## 2.5 Learning Techniques

Related to the *QM* concept presented in this thesis are several techniques in machine learning. There has been plenty of work on building mining models over continuous data streams, including clustering [107, 108], decision trees [109] [110],

nearest neighbors [111], and heavy hitters [112, 113]. New algorithms have also been proposed for maintaining statistics over data streams, e.g., samples [114], histograms [115], and quantiles [116].

A number of algorithms have been proposed in the literature for extracting knowledge from data, using clustering [117, 118], classification [109, 110, 119], frequency counting [120, 121] and time series analysis techniques [122, 123]. These techniques can be integrated into the classifier component of the *QM*, the subject we plan to investigate further.

Decision tree construction is an important problem in data mining [124–127]. Most of the proposed algorithms address the problem of decision tree construction for static data. The key issue in mining on streaming data is that only one pass is allowed over the entire data. Moreover, there is a real-time constraint, i.e., the processing time is limited by the rate of arrival of instances in the data stream, and the memory available to store any summary information may be bounded. For most data mining problems, a one pass algorithm cannot be very accurate. The existing algorithms typically achieve either a deterministic bound on the accuracy [108], or a probabilistic bound [110]. A good survey of data mining techniques for streaming environments is presented in [128].

Several methods have been proposed to deal with time changing concepts [109, 129, 130]. The two basic methods are based on *temporal windows*, where the window fixes the training set for the learning algorithm and *weighting tuples* that ages the training tuples by shrinking the importance of the oldest tuples. The time window method can be improved by adapting its size [129, 131].

VFDT [110] is a very fast decision tree algorithm for data-streams. The main innovation in VFDT is the use of the Hoeffding bound to decide when a leaf should be expanded to a decision node. Later, VFDT has been extended with the ability to detect changes in the underlying distribution of the examples. CVFDT [109] is an algorithm for mining decision trees from continuous-changing data streams. CVFDT works by keeping its model consistent with a sliding window of the most recent ex-

amples. When a new example arrives it increments the counts corresponding to the new example and decrements the counts to the oldest example in the window which is now forgotten. Each node in the tree maintains sufficient statistics. Periodically, the splitting-test is recomputed. If a new test is chosen, CVFDT starts growing an alternate subtree. The old one is replaced only when the new one becomes more accurate.

Other learning techniques in artificial intelligence, such as neural networks [132], intelligent agents and reasoning [133], intelligent information systems [134], logic and logic programming [135], planning and scheduling [136, 137], bayesian networks [138], genetic programming [139] have also received a lot of attention from the research community in the recent years.

Works dealing with uncertainty in classification and machine learning [140] focus primarily on the prediction accuracy of the models. Classical versions of classification algorithms typically are not designed to handle uncertainty [141]. To overcome this limitation, probabilistic decision trees [142], bayesian decision trees [143], and classifier ensembles [144] have been proposed to deal with classification of data with missing, imprecise, or updated attribute values.

## 2.6 Security and Access Control Enforcement

Agrawal et al. have coined the concept of Hippocratic databases [145] to incorporate the privacy protection within relational DBMS. Hippocratic databases use privacy metadata to represent the data owner's privacy preferences and the the data collector's privacy policies. The data is returned to users only when the policies meet the preferences. The work focuses on relational databases only and does not address the challenges present in the streaming context.

The problem of access control in dynamic environments has raised significant interests in research community in recent years [146–148]. [149] extends RBAC model to Temporal-RBAC, which supports periodic role enabling and disabling and tempo-



ral dependencies among permissions. GEO-RBAC [150] extends RBAC model with spatial awareness. For most of these access control models, however, the changes in the policies do not get reflected on the results until the query is re-executed after the change. While the focus of our work is to enforce access control throughout the long running time of continuous queries in a DSMS.

The notion of continuous access control has been introduced by Ravi Sandhu et al. as part of the *UCON* model [151, 152]. To the best of our knowledge, apart from the initial theoretical paper, our on continuous access control enforcement in DSMSs is the first real instance of the *UCON* model. One of the reasons for the lack of real systems is that it is difficult to implement in practice [151]. We believe that we are the first to do so.

Fine-grained access control in relational databases has received a lot of attention recently [153–155]. Fine-grained access control allows control of access at the granularity of individual rows, and to specific columns within those rows and is often required in many database applications. Wang et.al. [153] design a labeling scheme to hide information in a database. To answer queries the authors propose a query modification approach to evaluate the queries over tables with masked cells. Chaudhuri et.al. [154] propose a model for fine-grained authorization based on adding predicates to authorization grants. The model supports predicated authorization to specific columns, cell-level authorization with null-ification, authorization for function/procedure execution, and grants with grant option. The model also incorporates other novel features, such as query defined user groups, and authorization groups, which are designed to simplify administration of authorizations. The model is designed to be a strict generalization of the current SQL authorization mechanism. Kabra et.al. [155] make an observation that the majority of models for fine grained access control follow a view replacement strategy which suffers from the overhead of the access control predicates when they are redundant and potentially may leak information through channels such as user-defined functions, and operations that cause exceptions and error messages. The authors propose techniques for redundancy re-

removal and define when a query plan is safe with respect to UDFs and other unsafe functions. To address the potential information leakage, the authors propose techniques to generate safe query plans. To the best of our knowledge, none of works on fine-grained access control address the simultaneous enforcement of multiple policies (server side and client side) and typically consider a static relational database context instead of dynamic data streams – the focus of our work.

Another area of related work is the context-awareness in access control and context-aware adaptation of access-control policies, e.g., for crisis management (or emergency). The main idea here is to employ contextual parameters as inputs to the access control model (e.g., a context-sensitive RBAC model [156]). In our paper, we accommodate the requirements of context-aware access control by providing support for: (1) generation of security punctuations based on the real-time context data streams, and (2) support for the immediate enforcement of security policies to tackle emergency situations.

## 2.7 Tagging Methods

There are several ongoing projects that deal with annotation propagation and management for scientific databases, e.g., DBNotes [157], Mondrian [158], bdbms [159], and MMS [160]. Social bookmarking systems, such as *Flickr* [161], *Delicious* [162] and *Technorati* [163] support annotations of web resources and images with free-text keywords. For more examples of tagging systems and their taxonomy, we refer the reader to [164]. To the best of our knowledge, none of these existing works address the problem of tagging in the context of dynamic data stream environments.

Chi et al. [165] study the entropy of tagging systems, in an effort to understand how tags grow, and how the groupings of tags change over time and affect browsing of data. Halpin et al.’s work [166] looks at the nature of tag distributions with information theoretic tools. There has been some work on association rules in tagging systems, including [167, 167] and [168]. [168] primarily focuses on prediction of tags.

Oldenburg et al. [169] look at how to integrate tags across tagging systems by using Jaccard measure and discuss different types of tagging systems: social bookmarking, research paper tagging systems, but not DSMSs.

Research on self-describing streaming XML which can be viewed as “data tags” has received a lot of attention in recent years [170–172]. XML processing is typically more expensive compared to traditional stream data processing, and requires a special XML stream management functionality (in addition to the XML-aware optimizer and executor). Our proposed tagging approach is simpler in design and more light-weight compared to streaming XML, while at the same time it provides support for rich user-based tag semantics.

Relational data-bases have had an extraordinarily successful history of commercial success and fertile research. It is not surprising, therefore, that database researchers have attempted to understand annotations and “tagging” in the context of relational databases [157].

One of the biggest challenges in relational databases is the correct propagation of annotations through queries’ pipelines. This is similar to the problem we’ve discussed in the context of tag-aware query processing. In [157], a practical approach is taken to handling annotation in which an extension of SQL is developed supporting explicit user control over the propagation of annotations. The idea is to allow the user to control the flow of annotations by adding propagation instructions to the SQL query language. In our implementation, the tagging system performs (by default) the system-driven propagation, when processing tag-aware continuous queries. Adding support for user preferences regarding tag propagation in tag-aware queries is a subject of our future work.

Most of the work on annotations of relational data focuses on annotating individual values in a table. Geerts et al. [158] have taken a more sophisticated approach and provide support for annotating associations between values in a tuple. For example, in a query one might want to annotate fields  $A$  and  $B$  in the output with information that they came from input table  $R$ , and the fields  $B$  and  $C$  with information

that they came from table  $S$ . The authors introduce the concept of a “block” – a set of fields in a tuple to which one attaches an annotation and a “colour” which is essentially the content or some property of the annotation. They investigate both the theoretical aspects and the overhead needed to implement the system. Our approach supports various tagging granularity by using regular expressions in the *Applicability* field in the *tick-tags*, and to maintain the tags’ “lineage” we employ the streaming *stix* concept.

We are unaware of any work that addresses the problem of real-time data tagging in the context of DSMSs and provides support for both explicit and implicit tag querying. Furthermore, our proposed approach is unique in that it is stream-centric: tags attached to streaming data are interleaved with the actual data tuples in the data streams, and the processing of these streaming tags is encapsulated inside the tag-based query operators that can be combined with regular continuous query operators.

## 2.8 Summary

In this chapter, we have described work related to the concepts and algorithms proposed in this thesis, including an overview of various DSMSs, static and adaptive query optimization approaches, existing streaming metadata techniques in DSMSs. We have also discussed relevant access control solutions from the security area and the tagging and annotation approaches in various systems.

### 3 SECURITY AND ACCESS CONTROL FOR STREAMING DATA

In this chapter, we address the problem of *continuous access control enforcement* in dynamic data stream environments, where *both* data and query security restrictions may potentially change in real-time and must be enforced online.

We present the *FENCE* (short for *Continuous Access Control Enforcement in Dynamic Data Stream Environments*) framework that effectively addresses this problem. The distinguishing characteristics of *FENCE* include: (1) the *stream-centric* approach to dynamic security, (2) the *symmetric* security model for both continuous queries and streaming data, and (3) two alternative *security-aware query processing* methods, that can optimize the execution based on data-related as well as security-related selectivities. In *FENCE*, both data and query security restrictions are modeled symmetrically in the form of security metadata, called “security punctuations”. Security punctuations stream together with the data instead of being persistently stored on the server. We distinguish between two types of security punctuations, namely, the *data security punctuations* (or short, *dsp*s) which represent the access control policies of the streaming data, and the *query security punctuations* (or short, *qsps*) which describe the access authorizations of the continuous queries running on the server. With respect to the problem of efficient execution of continuous queries, we propose and compare two security-aware query processing methods, namely: (1) the *Security Filter Approach (SFA)*, and (2) the *Query Rewrite Approach (QRA)*.

The rest of this chapter is organized as follows. In Section 3.1, we motivate the need to address the problem of continuous and online access control enforcement for streaming data. We give the problem definition in Section 3.2. Section 3.3 presents the *FENCE* architecture. Section 3.4 describes our security model with data and query security punctuations and their semantics. Section 3.5 presents the alternative

security-aware query processing methods. We describe the results of our experimental study in Section 3.6. We conclude in Section 3.7.

### 3.1 Security in Data Stream Management Systems

Due to recent developments in pervasive and ubiquitous computing, many enterprises begin to provide high-quality services based on real-time data, e.g., patient monitoring, location-based services and ubiquitous social networking [2, 173, 174]. The information in such applications arrives in the form of infinite data streams to a DSMS, where continuous queries are evaluated.

One of the biggest challenges in such dynamic data stream environments is the access control enforcement – the ability to permit or deny a request to perform an operation (e.g., a read operation). Given the *long-running* nature of continuous queries, the content of the streaming data and along with it its “sensitivity” may change frequently over the lifetime of query execution. Furthermore, queries on the server may also experience frequent changes in their access control privileges while being executed. Such changes in security privileges may be due to mobility and varying context of the users receiving the results of continuous queries: query results may be accessed via mobile phones, PDAs or iPhones from any place and at any time. Clearly, the users sending their streaming data can be rightly concerned about possible unauthorized accesses to their real-time information and potential violations of their privacy. One of the major challenges here comes from the fact that the security policies of both data and queries can be concurrently very dynamic.

**Example 1:** *Ubiquitous healthcare system.* Healthcare systems support real-time monitoring and access to vital signs data of patients by doctors, emergency personnel, and pharmacies. Consider a physician executing a continuous query  $Q$  that monitors the health state of his patients, e.g., heart rate, blood pressure, etc. Over time, while the query  $Q$  is being executed, the physician may continuously acquire different

roles<sup>1</sup>, which may have different access privileges, e.g., a hospital employee ( $R_1$ ), a doctor with unrestricted access ( $R_2$ ), or a doctor with restricted access ( $R_3$ ). Possibly, multiple combinations of these roles can be active at any time depending on the policy and the doctor’s context. While working in the emergency room, the doctor’s active set of roles may be:  $\{R_1, R_2\}$ . When entering an insurance building to settle a claim, the doctor’s active set of roles immediately changes to:  $\{R_3\}$ . In the evening, when the doctor comes back home, his active role set becomes:  $\{R_1\}$ . The patients, transmitting their data through their monitoring devices, should have the ability to continuously regulate in which role their doctor can access their real-time health information.

**Example 2: Location-Based Services.** Recent improvements in location-based technologies and the drop in prices of location-tracking devices have spurred a new wave of mobile services, such as location-based services and geo-social networking applications [19]. Such applications naturally raise privacy concerns. Users consider their physical location and travel patterns highly privacy-sensitive and demand solutions that are able to protect their information. Therefore, it is essential to provide support for users to be able to frequently change their access control policies based on their preferences, to restrict who can “see” their real-time information (e.g., where they are, whom they are with, or what they are doing).

Based on these real-life examples, we can observe that dynamic changes in policies are natural and represent an essential part of an access control environment in data streams. We can also observe that changes in security may arise not only because of (1) the dynamic preferences of the users sending their data (i.e., the data providers) but also from (2) the dynamic privileges of the users receiving the results of continuous queries running on DSMS (i.e., the query specifiers). To the best of our knowledge, our work is the first to address the problem of *online access control enforcement* with *concurrent* dynamic changes in security for *both data and queries*.

---

<sup>1</sup>Here, we assume the system is using a role-based access control (RBAC) model [175].

### 3.1.1 Challenges

Due to the characteristics of streaming data, there are a number of inherent challenges that make continuous access control enforcement a challenging task.

- *Fast data arrival rate.* A common characteristic of data streams is a high data volume and a rapid arrival rate [2]. It is not feasible to store all data from all streams and take random accesses to the data as it is done in traditional databases. Therefore, the security policies associated with a data must be determined as fast as possible and the speed of the access control enforcement algorithm must be faster than the incoming data rate.
- *Single scan of data.* Due to the massive volumes of data, there may be not enough space to store all streaming data and its security policies (which may be numerous and of fine granularity). Therefore, one scan of data and its security restrictions with compact memory usage is required.
- *Dynamic changes in security.* The widespread usage of portable devices and the users' mobility are likely to lead to frequent changes in transmitted streaming data and possibly its "sensitivity". In addition to that, the mobility and the changing context of the users receiving the results may translate into frequent changes in the access control authorizations. Thus, an access control enforcement mechanism must be adaptive to runtime changes in security.
- *Correctness of enforcement.* The foremost challenge is the prevention of any information leaks that may occur when access is no longer authorized. It is also important to ensure that the access to data is not denied, when an access privilege has, in fact, been granted, especially when it is crucial to see the data *immediately* (e.g., in the case of an emergency). At any time, only the data elements that satisfy both the query and the data security policies at the same time must be returned as query results.



- *Low overhead.* The results in streaming environments are expected to be produced in near-real-time. Since access control enforcement is nothing but an added “overhead” compared to the traditional continuous query processing, its cost must be as low as possible not to decrease the utility of the DSMS.

### 3.1.2 Our Contributions: The *FENCE* Framework

To address the above-mentioned challenges, we propose the *FENCE* (short for *Continuous Access Control Enforcement in Dynamic Data Stream Environments*) framework that supports the online enforcement of changes in the security policies of the data as well as in authorizations of the continuous queries while they are being executed. *FENCE* employs the *Security Punctuation (SP)* model [38] for both the streaming data and the continuous queries. Furthermore, *FENCE* enables a much richer security semantics for various applications’ needs. These features introduce new technical challenges for which we present our solutions in the rest of this chapter. Our major contributions can be summarized as follows:

- *FENCE* models *both* data-side and query-side dynamic security restrictions *symmetrically* using streaming “security punctuations”<sup>2</sup> metadata. *FENCE* extends the *SP Framework* [38] scheme by distinguishing between the two types of *sps*, namely, the *data security punctuations (dsps)* and the *query security punctuations (qsps)* to enforce security for both data and queries in a simple and efficient manner.
- *FENCE* framework supports security-aware continuous query processing with combined *dsps* and *qsps*. Compared to [38], which supports only *one* security-aware query processing method, *FENCE* is equipped with two adaptive techniques, namely: (1) the *Security Filter Approach (SFA)*, and (2) the *Query*

---

<sup>2</sup>We chose to name the streaming security metadata “security punctuations” (or short *sps*), because by introducing *sps* into data streams, we subdivide (i.e., punctuate) infinite data streams into finite partitions with associated access control policies.

*Rewrite Approach (QRA)*. We discuss the advantages and the limitations of each of the methods, and describe how both methods can support security-aware and compliant query processing, and can adapt to both data as well as security-related selectivities.

- Since in data stream environments, the access control policies may change in the middle of query execution, *FENCE* distinguishes between two types of security policy enforcement semantics, namely the *deferred* and the *immediate* enforcement. In the former, the access control policies are enforced on only the data tuples that arrive *after* the policy change. Alternatively, in immediate enforcement (e.g., in emergency scenarios), the access control is enforced *instantly* including the tuples that have arrived *before* the policy change and are not yet returned as query results. We formally address this issue and provide an efficient solution to support both types of security policy enforcements.
- We have implemented *FENCE* in a general DSMS prototype [8]. Our experimental study shows that *FENCE* efficiently supports access control on data streams with data and query security policy changes and security-related overhead with *sps* is low relative to continuous query execution cost.

### 3.2 Problem Formulation

To formulate the problem we address in this chapter, we first give the definition for the concept of *continuous query processing* (or CQP for short). In traditional CQP, continuous queries are registered in DSMS, and only the data tuples that satisfy the predicates of the continuous queries are produced as results. We call these predicates – *query predicates* – and formally define CQP as follows:

**Definition 3.2.1 (Continuous Query Processing (CQP))** *Suppose that a data element  $d=(v_1, v_2, \dots, v_n)$  from a data stream has  $n$  attributes and a query predicate  $\varphi_Q(attr_1, attr_2, \dots, attr_n)$  on  $d$  represents the condition of a given continuous query*

$Q$ . Then, whenever  $d$  arrives, the continuous query processing mechanism produces  $d$  as a result of  $Q$  if and only if  $\varphi_Q(v_1, v_2, \dots, v_n)$  is true.

In *Security-Aware Continuous Query Processing* (SA-CQP), in addition to the query predicates, there is an additional type of predicates, called *security predicates*, which determine whether the query may access the arriving data tuples based on the current access control policies. We distinguish between two types of security predicates, namely: (1) the *data-side security predicates*, which represent the data provider's security policies on the streaming data and (2) the *query-side security predicates*, which describe the query specifier's current access authorizations. Continuous queries, registered by a user (i.e., query specifier) implicitly acquire the access authorizations of that query specifier. Consequently, SA-CQP enforces access control on data streams by only producing the results that satisfy both the query predicates and the security predicates at the same time. SA-CQP can be formally defined as follows:

**Definition 3.2.2 (Security-Aware Continuous Query Processing (SA-CQP))**

Suppose that a data element  $d = (v_1, v_2, \dots, v_n)$  from a data stream has  $n$  attributes, a query predicate  $\varphi_Q(attr_1, attr_2, \dots, attr_n)$  on  $d$  represents the condition of a given continuous query  $Q$ , and a security predicate  $\varphi_S(attr_1, attr_2, \dots, attr_n)$  on  $d$  represents a security policy  $S$ . Then, whenever  $d$  arrives, the security-aware continuous query processing returns  $d$  as a result of  $Q$  if and only if  $\varphi_Q(v_1, v_2, \dots, v_n) \wedge \varphi_S(v_1, v_2, \dots, v_n)$  are both true.

Figure 3.1 visually depicts the SA-CQP concept. Query predicates are denoted by  $\varphi_Q$ , and security predicates  $\varphi_S$  are composed of two types of security predicates, namely the *data-side security predicates* and the *query-side security predicates* denoted by  $\varphi_{ds}$  and  $\varphi_{qs}$ , respectively.

In our work, we address one of the key aspects of security in data stream environments, namely, the *dynamic changes in security policies* (specifically, the changes in access control), while continuous queries are being executed. Dynamic security means that during query execution access control policies affecting the processing

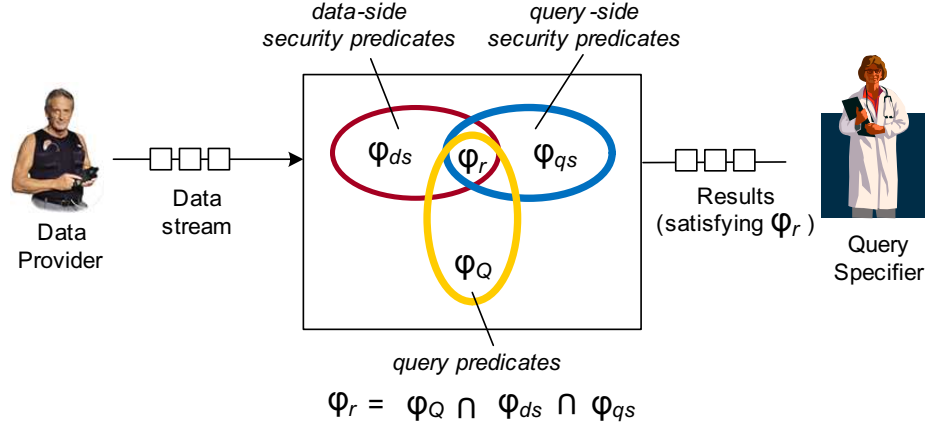


Figure 3.1. Conceptual idea of security-aware continuous query processing (SA-CQP).

(and the results) of the query may frequently change to support the real-time needs of users and the requirements of applications. *Data-side dynamic security* represents the changes in the data providers' security preferences and *query-side dynamic security* represents the changes in the query specifiers' access authorizations. In our work, we provide a solution for SA-CQP in the presence of both types of dynamic security.

### 3.3 Overview of *FENCE* Framework

In this section, we present the general *FENCE* architecture and then describe a specific instance of the framework that we consider in the rest of the paper.

#### 3.3.1 *FENCE* Architecture

To model dynamic access control in a data stream environment, *FENCE* extends the concept of security punctuations introduced in [38]. *Security punctuations* (or short, *sps*) are meta-data embedded inside data streams that describe the following aspects: (a) who has access rights, (b) to which streaming data objects, and (c) when. Compared to the original *sps* in [38], which only describe the data-side security policies, *FENCE* extends the *sp* paradigm to model both the *data-side* dynamic security policies as well as the *query-side* dynamic access authorizations that may be both

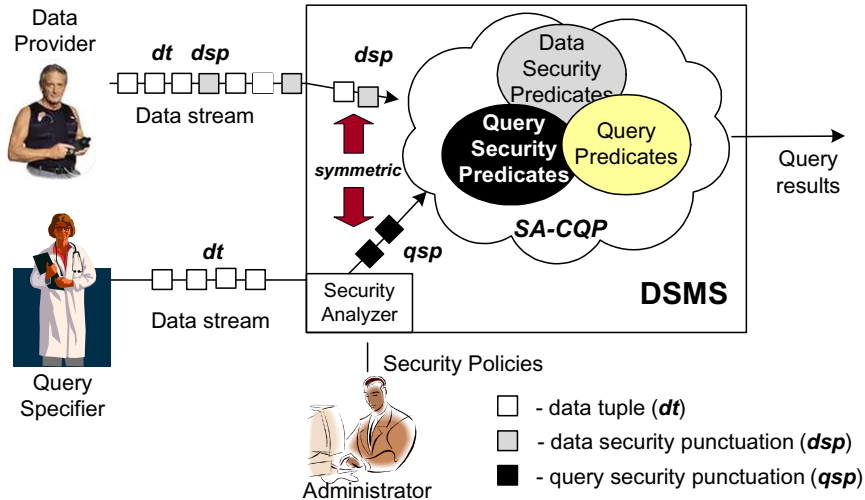


Figure 3.2. Overview of *FENCE* architecture.

continuously changing. By uniformly representing the security settings for data and queries using a single concept, namely the security punctuations, *FENCE* facilitates a simpler security model, code re-use and enables similar security processing for both data and queries. Using streaming *sps*, a DSMS can support *online* flexible, dynamic, and fast access control enforcement over infinite data streams, while queries are being evaluated. We will discuss the concept of security punctuations, as applicable to data and queries, in Section 3.4.

Figure 3.2 shows a high level overview of the *FENCE* architecture. In a typical streaming environment, we distinguish between three types of users: (1) The *data provider* – a user continuously sending his or her streaming data with the interleaved *sps*, that describe the real-time security preferences on his or her streaming data. (2) The *query specifier* – a user who registers a continuous query on the server to be evaluated on the incoming streaming data. As can be seen from Figure 3.2, a query specifier also streams his or her real-time context (via a data stream), based on which *qsps* that describe the real-time access privileges of the continuous query are generated. For example, if a query specifier is a physician and he is out on a lunch break, the current location of the physician (“outside the hospital premises”) will generate a punctuation that limits the data (the health information of his patients)

the doctor can access from his portable device. (3) The *DSMS administrator* is a user responsible for registering security policies that guarantee that correct privileges are given to the queries based on the context of the query specifiers. The *Security Analyzer* component in Figure 3.2 is responsible for generating correct *qsps* according to the organization’s security policy registered by the DSMS administrator. Both data and query-side security are symmetrically modeled by security punctuations. This *symmetry* facilitates a simpler model and similar processing for *both* data and query security metadata inside DSMS. When streaming security punctuations arrive to the system, the query processor interprets *sps* as security predicates by processing the data-side and the query-side *sps* alike, and then, produces results that satisfy both the query predicates as well as the security predicates (as shown in Figure 3.2). In *FENCE*, *dsps* are assumed to be directly generated by the data providers<sup>3</sup> and *qsps* can either arrive from the query specifiers (or from a third-party security service) or most likely are generated locally in DSMS by the *Security Analyzer* module based on the query specifiers’ current context<sup>4</sup>. We discuss the different possible scenarios of *sp* generation in Section 3.4.4.

### 3.3.2 An Instance of the *FENCE* Framework

*FENCE* is a general framework and is not restricted to any particular data or access control model. But to make our discussion concrete, here, we describe an instance of *FENCE* framework, with a specific data and an access control model that we will consider in the rest of the paper.

---

<sup>3</sup>The data security punctuations (*dsps*) may also be generated on the DSMS by evaluating continuous security policy queries on the incoming data streams (see Section 3.4.4), but for simplicity of discussion, we assume that the *dsps* arrive to the DSMS already interleaved with the data.

<sup>4</sup>We assume that the context of the query specifiers is represented by additional incoming data streams, e.g., stream of location updates.

## Data and Query Model

We consider a centralized DSMS processing long-running select-project-join (SPJ) queries on a set of infinite data streams. A continuous data stream  $S$  is a potentially infinite sequence of tuples that arrive over time. The general schema of tuples in a data stream is described by:  $[sid, tid, A, ts]$ , where  $sid$  is the stream identifier,  $tid$  is the tuple identifier,  $A$  is a set of attribute values in the tuple, and  $ts$  is the timestamp of the tuple. As commonly considered in other streaming systems, e.g., [43, 176], the timestamps of the stream elements are assumed to be ordered. For simplicity of discussion, we consider a single continuous query  $Q_i$ , registered by a query specifier in the DSMS, to be executed over data streams  $A, B, \dots, Z$ . The security restrictions applicable to the query specifier get implicitly inherited by  $Q_i$ . Query  $Q_i$  is represented by a query execution plan composed of operators  $op_1, \dots, op_k$ , where each operator acquires the security restrictions associated with the query  $Q_i$  for which it processes the incoming data tuples.

## Access Control Model

An access control policy specifies who has access to which objects and when. In its general form, an access control policy can be described by a triple  $\langle object, subject, operation \rangle$ . An *object* is an entity that contains the information. Examples of objects in data stream environments are: streams, tuples, tuple attributes and data values. A *subject* may invoke a request to access an object to perform an operation, e.g., a “read operation” on a data tuple. The subjects in *FENCE* are the query specifiers. Subjects acquire the access rights which are the set of privileges that they can hold and execute on an object. In our work, we consider a read right (operation) only. Due to the fact that just about all stream systems are read-only, this is a natural focus. However, the model can be easily extended to support other operations as well, such as update, delete, etc. Access to an object implies the right to use the information

it contains. An access is granted, if the corresponding subject owns a permission for the requested operation. Authorization is the granting of the access permissions.

As an example of an access control model, we consider a *Role-Based Access Control (RBAC)* model [175] in our work, and show how it can be implemented in *FENCE*. RBAC is one of the most well-known and widely-used access control models in modern systems today [175]. The main idea of RBAC is to introduce *roles* as an abstraction layer to decouple subjects and permissions [175]. Under the assumption of using RBAC, the streaming *dsps* describe which roles have currently the access rights to which streaming objects, and the streaming *qsps* depict the current roles of a continuous query. Query specifiers activate their default roles when they sign into the DSMS. We require that each query specifier belongs to at least one role. However, this assignment may change while the query specifier is receiving the results of his or her continuous query.

### 3.4 Dynamic Security Policy Model

In this section, we present the schema and the semantics of the security punctuations in *FENCE*. We provide examples of various *sps* and describe the scenarios of *sp* generation.

#### 3.4.1 General Security Punctuation Schema

In *FENCE*, we employ security punctuations to model symmetrically both the data and the query-side dynamic security restrictions. Such symmetric model makes SA-CQP simpler and allows security-related code re-use in DSMS. We call the *sps* representing data provider’s preferences for security – the *data security punctuations (dsps)* and the *sps* representing the query specifiers’ access privileges – the *query security punctuations (qsps)*. Figure 3.3 shows a general *sp* schema applicable to both *dsps* and *qsps*. We discuss each field in the *sp* schema next.



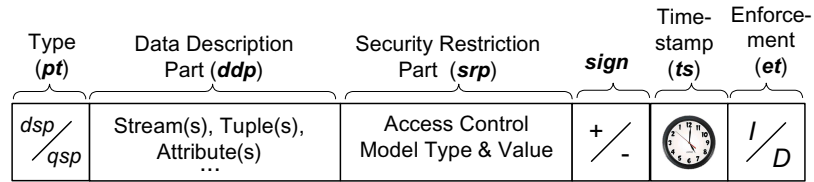


Figure 3.3. General security punctuation schema.

- *Punctuation Type (pt)*: describes whether the punctuation is a data or a query security punctuation.
- *Data Description Part (ddp)*: specifies which object(s) the access control policy applies to, e.g., which stream(s), tuple(s), or tuple attribute(s) [38]. For compactness of storage, we use *regular expressions* to describe objects and their policies inside *sps*.
- *Security Restriction Part (srp)*: denotes both the access control model type and the subjects authorized by the policy. Since we use RBAC in this work (see Section 3.3.2), the *srp* specifies RBAC as the model type and a set of role(s) that are authorized by the *sp*.
- *Sign*: indicates whether the authorization represented by the *sp* is positive or negative (see [177] for more details).
- *Timestamp (ts)*: records the time when the *sp* was generated.
- *Enforcement (et)*: indicates the security policy enforcement setting. We distinguish between two types of enforcement, namely the *Deferred (D)* enforcement and the *Immediate (I)* enforcement. We describe the details of this attribute in Section 3.4.2.

### 3.4.2 Semantics of Security Punctuations

A security policy may be expressed by one or more *sps* and may apply to zero or more tuples. A set of consecutive *dsps* or *qsps* form a “*batch*” of *sps* which is interpreted as a single access control policy or a complex authorization. All *sps* of

the same policy (or authorization) have the same timestamp  $ts$  – the time when the security policy was created and the  $sps$  were generated. If there is no  $sp$  authorizing the access to an object, “denial-by-default” is enforced, i.e., an access to a streaming object is denied unless explicitly allowed.

Another important semantic attribute is the access control policy’s *enforcement* setting. In traditional DBMSs, the enforcement semantics of security policies is clear – a policy applies to *all* data (i.e., the entire dataset) stored in the system. Furthermore, the policies do *not* change in the middle of query execution, and even if they do, they are not reflected on the results until the query is executed over again. In contrast, in DSMSs, the semantics is not quite so clear. Since data streams are infinite and queries are continuously being evaluated, whenever a new  $sps$  (with a new policy) arrives, there may be data tuples (that have arrived before the  $sp$ ) and are in the pipelines of the continuous query execution plan, that according to the  $sp$  are no longer accessible (the reverse may also be true, and the previously inaccessible tuples may now be accessed by the query).

To properly reflect the users’ security preferences in the system, we introduce two ways of enforcing a security policy, namely the *deferred* and the *immediate* enforcements, specified in the *et* attribute of an  $sp$ . In the case of the deferred enforcement, a policy represented by an  $sp$  applies only to the data that arrives *after* the  $sp$ , i.e., the tuples whose timestamps are greater than that of the  $sp$ . This type of enforcement is the most frequent case, and is needed for applications that need to protect the “future” data. For example, if a user carrying a cell phone device enters a casino, he or she may want to instantly prevent others from knowing his precise whereabouts. Thus, an  $sp$  with the deferred enforcement will be injected into his stream transmitting the user’s real-time location updates. With the immediate enforcement setting, the new policy affects both the (near past) data that has arrived to the DSMS *before* the current  $sp$  as well as to the future data that follows *after* it. Hence, the policy here may apply to both the “historic” and the “future” data. This type of enforcement is needed for the applications that demand the immediate reflection of the

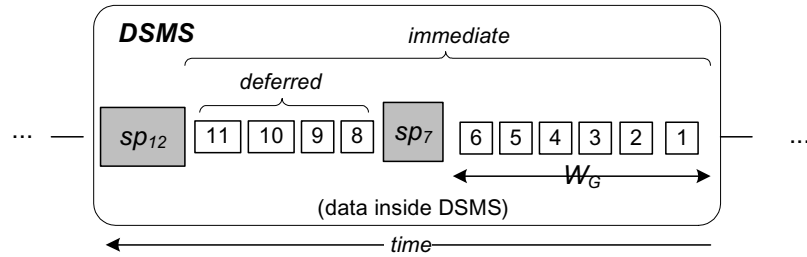


Figure 3.4. Enforcement of security punctuations in an “immediate” and “deferred” manner.

policy changes on the query results, without waiting for the arrival of new data. For example, in some applications, e.g., health monitoring, or financial applications, users cannot afford to wait for the arrival of new streaming data after a policy changes. In a healthcare application, this could be a matter of saving a patient’s life.

Figure 3.4 visually illustrates the differences between the immediate and deferred enforcement semantics. Consider the security punctuation  $sp_7$ , where 7 is the timestamp of the  $sp$ . Tuples denoted by the integers 1 through 6 represent the data tuples that have arrived before the  $sp_7$ , and tuples 8 through 11 after the  $sp_7$ . With the immediate enforcement,  $sp_7$ ’s policy will apply to tuples 1 through 11. With the deferred enforcement,  $sp_7$ ’s policy will only apply to tuples 8 through 11. To support the immediate security enforcement, we only consider the streaming data that is currently inside the DSMS (see Figure 3.4). A recent past data window can be further customized based on the application needs, e.g., last 1 hour of data only. To enable the immediate security policy enforcement, we maintain a global window  $W_G$  of the streaming data in DSMS, and  $W_G$  periodically slides, purging the data tuples that have expired from the “recent past” data window.

### 3.4.3 Examples of Security Punctuations

Consider the following data streams:  $S_1$  is the heart data stream,  $S_2$  represents the blood pressure data stream and  $S_3$  is the respiration data stream. Let  $R =$

$\{D_1, D_2, D_3, D_4, D_5\}$  be the set of roles in DSMS<sup>5</sup>. The following *dsp*s and *qsps* may specified<sup>6</sup>:

#### Data Security Punctuations

***dsp*<sub>1</sub>**:  $\langle \text{dsp} | S_1, *, * | D_2 | + | 12:00:00PM | D \rangle$

only queries registered by a cardiologist (role  $D_2$ ) can query the stream  $S_1$  (heart rate) after this punctuation arrives (due to deferred semantics, i.e.,  $\text{dsp}_1.et = D$ ).

This is an example of a *stream granularity policy*.

***dsp*<sub>2</sub>**:  $\langle \text{dsp} | *, [30, 210], * | D_4 | + | 12:00:00PM | D \rangle$

only queries registered by a general physician (role  $D_4$ ) can access data tuples (from any data stream) of patients with ids between 30 and 210, after this punctuation arrives ( $\text{dsp}_2.et = D$ ). This is an example of a *tuple granularity policy*.

***dsp*<sub>3</sub>**:  $\langle \text{dsp} | \{S_1, S_2\}, *, \{HeartBeat\} | \{D_2, D_5\} | + | 12:00:00PM | I \rangle$

only a cardiologist ( $D_2$ ) or a nurse-on-duty ( $D_5$ ) can query the heart beat from streams  $S_1$  and  $S_2$ . This is an example of an *attribute granularity policy*.

#### Query Security Punctuations

***qsp*<sub>1</sub>**:  $\langle \text{qsp} | \text{null} | D_1 | + | 12:00:00PM | D \rangle$

the query acquires a role of a dermatologist ( $D_1$ ) with deferred enforcement, i.e., the role applies to the query after the arrival of  $\text{qsp}_1$  and will pertain to the data tuples with the timestamp greater than  $\text{qsp}_1.ts$ .

***qsp*<sub>2</sub>**:  $\langle \text{qsp} | S_1, *, * | D_4 | + | 12:00:00PM | D \rangle$

the query acquires a general physician ( $D_4$ ) role and the current authorization of role  $D_4$  is the permission to only access stream  $S_1$  (heart data stream). The enforcement is deferred.

---

<sup>5</sup>The roles can be as follows:  $D_1$  = dermatologist,  $D_2$  = cardiologist,  $D_3$  = hospital employee,  $D_4$  = general physician, and  $D_5$  = nurse-on-duty.

<sup>6</sup>The different fields in an *sp* are separated by a vertical bar “|”.

$qsp_3$ :  $\langle qsp | \text{null} | \{D_2, D_5\} | \text{null} | 12:00:00PM | I \rangle$

the query now acquires roles  $D_2$  and  $D_5$  with an immediate enforcement.

### Combination of DSPs and QSPs

To determine which data tuples, the query currently has access to, the intersection of the data and the query security punctuations must be evaluated [38, 178]. Only if the intersection between the policies of  $dsp$ s and the authorizations of  $qsp$ s is non-empty, the access to the streaming data elements is granted. For example, if  $dsp_1$  and  $qsp_2$  arrive at the same time, the query access to the data tuples following the  $dsp_1$  will be denied. Although both  $sps$  grant a permission to access stream  $S_1$ , the  $qsp_2$  indicates that the current role of the query is  $D_4$ , and the access control policy of the data allows only the role  $D_2$  to access the stream  $S_1$ .

#### 3.4.4 Security Punctuations Generation

Until now, we have assumed that  $sps$  are manually generated by users and arrive to DSMS for processing already interleaved with the data. However, in some applications, such scenario is not feasible, and users may gain improper access privileges. To handle real and more complex real-life scenarios, here we discuss the more sophisticated approaches for generating  $sps$ . The discussion below is applicable to both  $dsp$ s and  $qsp$ s due to the symmetric property of the model.

- **Time-driven:**  $Sps$  can be generated at a specific time or periodically. For example, every  $\Delta$  time units, a security module on the data provider's device may generate an  $sp$  that carries the latest user-specified policy. The time-driven method allows users to periodically generate  $sps$  to automatically enforce current access control policies.

- **Value-driven:** Alternatively, *sps* can be generated based on the observed data values. For example, an *sp* can be inserted into a data stream whenever an attribute value has exceeded a predefined threshold.
- **Query-driven:** A more advanced approach is to generate appropriate policies for a specific context by evaluating a special type of continuous queries – the *Continuous Security Policy Queries* (or short *CP-Queries*). A *CP-Query* consumes data streams describing a user’s context (e.g., current location, activity, or any other context) or a data stream specific to a particular domain and produces as output a security metadata stream – a stream composed of only *sps*. We can easily support a *CP-Query* approach in DSMSs, since the processing of such queries is almost identical to the regular continuous queries, except the produced results are *sps* here instead of regular data tuples.

### 3.5 Security-Aware Continuous Query Processing

In order to support efficient security-aware continuous query processing (SA-CQP), the following key issues must be addressed: (1) how should the query predicates and the security predicates be evaluated together, (2) how should the security predicates be adapted, whenever a new *dsp* or a *qsp* arrives, and (3) how should the immediate and the deferred enforcement semantics be efficiently and correctly implemented. In this section, we address the above issues by presenting the two alternative query processing methods. To motivate our proposed approaches, we first begin with the naive method.

#### 3.5.1 Naive Approach

A naive method for query processing with dynamic security takes a very simple approach: it completely separates the access control processing from regular CQP. Such strategy evaluates security predicates at a designated point – either before or

after the query plan execution. The former and the latter strategies are also known as *pre-filtering* and *post-filtering*, respectively [38]. Figure 3.5 illustrates the naive approach along with the *FENCE* approach, which integrates the access control processing with the continuous query processing. Here,  $A, \dots, Z$  represent the regular input data streams with the embedded *dsps* from data providers, and  $C$  represents a stream transmitting *qsps*.

In naive pre-filtering method, a security filter, which discards data elements that do not satisfy the security predicates is placed *before* the query execution plan. Therefore, only the data tuples the query has the access rights to access can enter the query plan, e.g., [179, 180]. The post-filtering method is the reverse of the pre-filtering: the query predicates are evaluated first, and then the results get filtered *post-mortem* based on the access control policy of the data and the access rights of the query. In both the pre- and the post-filtering methods, the fixed placement of the access control filters may add significant processing overheads and considerably limit the query performance. If the access control policies are “loose” (i.e., query specifier has access to nearly all data), but the query predicates are very selective, the pre-filtering method may result in a heavy security-related processing overhead prior the query execution plan. This may be unnecessary, if the query predicates end up discarding the majority of the tuples anyways. In contrast, the post-filtering method may introduce unnecessary processing overhead, when very expensive query predicates (e.g., joins, groups by) are evaluated first, only for the tuples to be discarded later by the security predicates, because the query does not have access rights to them.

In the following sections, to overcome the limitations of the naive method, we propose two efficient SA-CQP methods employed in *FENCE*, namely the *Security Filter Approach (SFA)* and the *Query Rewrite Approach (QRA)*. Both *SFA* and *QRA* have a key advantage – they *integrate* security processing together with traditional query processing and can adapt to not only data-related but also to security-related selectivities. Such deep integration with traditional continuous query processing can help reduce the waste of resources, when few subjects have access rights to data or

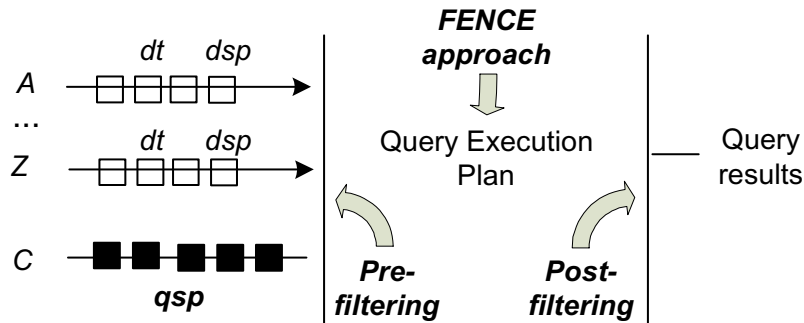


Figure 3.5. Query processing with *sps*.

minimize the security-related processing overhead when the query predicates are very selective.

### 3.5.2 Security Filter Approach (SFA)

The main idea of the *SFA* is to introduce a special physical operator that performs access control-based filtering into the query execution plan. We call this new operator the *Security Shield Plus* ( $SS^+$ ) operator<sup>7</sup>, and it is handled just like any other traditional query operator in query processing and query optimization.  $SS^+$  operator can be viewed as a “select operator” that filters input data tuples based on the security predicates determined based on the arrived *dsps* and *qsps*. The filtering condition of  $SS^+$  changes dynamically whenever a new *dsp* or a *qsp* arrives. Figure 3.6 shows how the *SFA*-based SA-CQP works with the  $SS^+$  operators. The triangle-shaped operators in the figure are the  $SS^+$  operators, filtering data based on the security predicates of the query. Just like for an ordinary select operator, the location of  $SS^+$  in the query plan is determined by the query optimizer according to the selectivities of the security predicates. If the selectivity is high, the  $SS^+$  operator is pushed down in the query plan to come before the operators with lower selectivity (similar to “selection pushdown”). Contrary to the traditional select operator, however,  $SS^+$  is a stateful operator: it stores the most recently arrived *dsps* and *qsps* in its buffers:  $Buffer_{dsp}$

<sup>7</sup> $SS^+$  is similar in spirit to the initially proposed *Security Shield* (*SS*) operator in [38], however, its semantics is more sophisticated: it provides support for both *dsps* and *qsps* with much richer semantics.



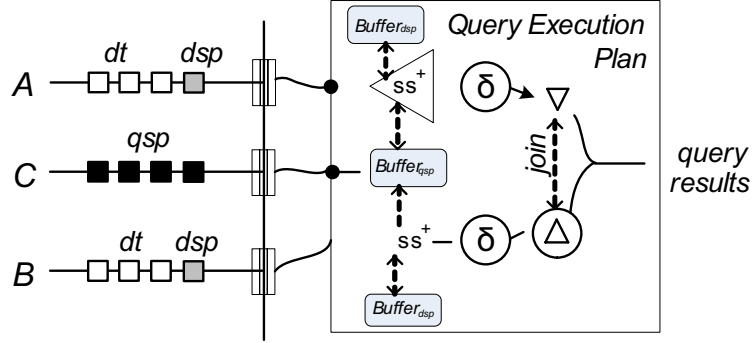


Figure 3.6. *SFA*-based SA-CQP.

and  $Buffer_{qsp}$ , respectively<sup>8</sup>, and computes their intersection to determine the current security predicates for filtering of data.

Since the rest of processing is similar to traditional CQP, we only explain the execution of  $SS^+$  operator<sup>9</sup>. Figure 3.7 shows the pseudocode for  $SS^+$  execution. If the input to  $SS^+$  is a security punctuation ( $dsp$  or  $qsp$ ), the *security\_predicates* variable, that represents the current intersection of the data and the query security policies, are updated to reflect the changes in policies (Step 3). If the input is a data tuple, it is propagated to the next operator in the pipeline if and only if the data tuple’s security policy satisfies the *security\_predicates* condition, otherwise, the tuple is discarded (Step 8).

Figure 3.8 shows the algorithm for computing the *security\_predicates* in  $SS^+$  operator. For every newly arrived  $sp$ , the intersection with the opposite type of  $sps$  (stored in the security buffer) is performed, e.g.,  $qsp \cap Buffer_{dsp}$  (Step 2) and  $dsp \cap Buffer_{qsp}$  (Step 5). This intersection is stored in the *security\_predicates* variable, which represents the current filtering condition.

If an  $sp$  has the immediate enforcement setting (Step 4 in Figure 3.7), in addition to the change in the *security\_predicates* variable (to be reflected on the future

<sup>8</sup>As mentioned in Section 3.4.1, all  $sps$  that belong to the same policy have the same timestamp  $ts$ . Therefore,  $Buffer_{dsp}$  and  $Buffer_{qsp}$  store the  $sps$  for data and queries respectively, that have arrived most recently and have the same  $ts$ .

<sup>9</sup>To preserve the correct security semantics during execution, traditional query operators in *SFA* need to be modified to become “security punctuation-aware”, as described in [38].

```

SSPlusExecution (o streaming object)
01 if (o.type == ‘‘security punctuation’’)
02   sp ← o
03   security_predicates ← ProcessSp(sp)
04   if (sp.et == ‘‘immediate’’)
05     ProcessImmediateSp(sp)
06 if (o.type == ‘‘data tuple’’)
07   if (o does not satisfy security_predicates)
08     discard o
09   else
10     propagate o

```

Figure 3.7.  $SS^+$  execution in  $SFA$ .

```

ProcessSp (sp security punctuation)
01 if (sp.pt == ‘‘qsp’’)
02   security_predicates ← Intersect(sp, Bufferdsp)
03   update Bufferqsp with sp
04 else if (sp.pt == ‘‘dsp’’)
05   security_predicates ← Intersect(sp, Bufferqsp)
06   update Bufferdsp with sp
07 return security_predicates

```

Figure 3.8. Processing of  $sp$  in  $SS^+$ .

data), the historic data in the  $W_G$  window (which has arrived earlier than the  $sp$ ) must be re-processed to be affected by the changes in the policy. To efficiently handle the immediate enforcement, we introduce a notion of the ‘‘narrowing intersection scope’’ for security policies. Informally, the narrowing means that the updated *security\_predicates* (representing the access control filtering condition) are more selective than prior to the change. Definition 3.5.1 formally describes the narrowing scope.

**Definition 3.5.1** Let  $\varphi_i$  be the security predicate at time  $ts_i$  and  $\varphi_j$  at time  $ts_j$  ( $ts_i < ts_j$ ). If the security predicate has changed between time  $ts_i$  and  $ts_j$  (i.e.,  $\varphi_i \neq \varphi_j$ ), and if  $\exists$  any data tuple  $d$  that makes  $\varphi_j(d) = \text{false}$  but makes  $\varphi_i(d) = \text{true}$ , the policy scope is said to be narrowing.

Figure 3.9 shows the algorithm for processing of the immediate *sps* using the notion of narrowing to optimize the immediate enforcement execution. If the scope is narrowing, it means that there may be tuples in the query pipeline that might have already passed through the  $SS^+$  (based on the earlier security policies), but are now no longer accessible (because of the narrowed intersection of the policies). Therefore, to immediately enforce the access control in such case, it is enough to consider only the data tuples that have already passed the  $SS^+$  in the query plan. To prevent that data from being returned as results, the new  $SS^+$  operator is activated at the root of the query execution plan until all of the pipelined data (that has arrived prior to the *sp*) is processed (Step 3). Otherwise, if the policy is *not* narrowing, the tuples stored in the  $W_G$  must be re-processed using the plan, to immediately see the results (that otherwise would possibly not be produced), thus reflecting the updated security policy (Steps 5 and 6). This can be done by clearing the data tuples from all of the query operators' queues, and then, feeding the data tuples from the recent past window  $W_G$  into the query plan.

```

ProcessImmediateSp (sp security punctuation)
01 scope  $\leftarrow$  DeterminePolicyScope(sp)
02 if (scope == ‘‘narrowing’’)
03   ActivatePostFiltering(sp)
04 else
05   discard all data tuples in the query plan
06   start query processing with  $W_G$ 

```

Figure 3.9. Processing of ‘‘immediate *sps*’’.

By encapsulating all security processing inside  $SS^+$  operators, *SFA*-based security-aware continuous query processing (or SA-CQP) can interleave the execution of security predicates with traditional query predicates. *SFA*, however, may require substantial modifications to the codebases of the current DSMSs (see Section 3.5.4). In the next section, we propose another SA-CQP approach that minimizes the need to modify existing DSMSs significantly and largely reuses the existing query processor infrastructure inside DSMS as it is.

### 3.5.3 Query Rewrite Approach (QRA)

The main idea behind the *QRA*-based SA-CQP comes from the observation that the enforcement of the dynamic security policies can be seen as the dynamic “rewriting of queries”<sup>10</sup>. According to the SA-CQP definition (in Section 3.2), we consider a query registered in DSMS that consists of query predicates and security predicates, where the security predicates are updated whenever a new *sp* arrives. A DSMS can support dynamic security changes in SA-CQP by creating a “new” query with the integrated in it the latest security predicates and replacing with it the current query. Table 3.1 shows an example of query rewriting. Here, after the arrival of *sps*, the original query predicate  $(R.a = S.a) \wedge (0 < R.b < 100) \wedge (0 < S.c < 100)$  is rewritten into  $(R.a = S.a) \wedge (0 < R.b < 50) \wedge (50 < S.c < 100)$  to reflect the access control policies described by the  $dsp_1$  and the  $qsp_1$ .

Figure 3.10 gives an overview of the *QRA*-based SA-CQP. Compared to the *SFA*, where  $SS^+$  operators process *sps* in the query plan, the *QRA* uses a centralized module, called the *Query Rewriting Module (QRM)*, to process arriving *sps*. *QRM* consumes *dsp*s and *qsp*s immediately upon their arrival to the system and stores them in the global  $Buffer_{dsp}$  and the  $Buffer_{qsp}$ , respectively. *QRM* also stores traditional query predicates in the  $Buffer_{query}$ . Whenever new *sps* arrive, *QRM* rewrites the cor-

---

<sup>10</sup>Query rewriting is generally used to compose queries or manage views. In our work, we exploit the query rewriting concept for the purpose of combining security and query predicates to adapt to dynamic changes in the access control policies and authorizations.

Table 3.1  
Example of query rewriting.

Original Predicates	Rewritten Predicates
$Q.p_1 \rightarrow (R.a = S.a)$	$Q.p_1' \rightarrow R.a = S.a$
$Q.p_2 \rightarrow (0 < R.b < 100)$	$Q.p_2' \rightarrow 0 < R.b < 50 // Q.p_2 + dsp_1$
$Q.p_3 \rightarrow (0 < S.c < 100)$	$Q.p_3' \rightarrow 50 < S.c < 100 // Q.p_3 + qsp_1$
$dsp_1 \rightarrow (0 < R.b < 50)$	
$qsp_1 \rightarrow (50 < S.c < 100)$	

responding query using the information stored in these buffers. Regular data stream tuples are processed by the query processor in the same way as in the traditional DSMSs. We note that *sps* are not sent into the query execution plan, but rather consumed by the *QRM* module to generate a new query. In that regard, the regular continuous query operators do not need to be “security punctuation-aware” as in the *SFA*.

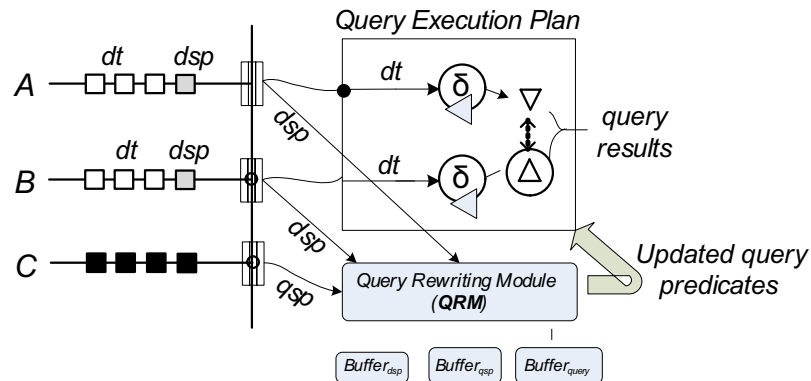


Figure 3.10. *QRA*-based SA-CQP.

Figure 3.11 shows the pseudocode for the *QRA*-based SA-CQP algorithm. Compared to the *SFA* which implements SA-CQP using  $SS^+$  operators, *QRA* realizes SA-CQP by executing the *QRM*, which may rewrite the query (and consequently its execution plan) in an effort to combine the security predicates with the query predicates. In the algorithm in Figure 3.11, if the input is an *sp*, the *QRM* rewrites the continuous query plan to integrate the new security predicates into the execution

plan (Step 3). The processing for the immediate enforcement is the same as in the *SFA*-based algorithm. If the input is a regular data tuple, the query processor evaluates it as in the regular continuous query processing (Step 7). We note that, in the context of *QRA*, the *QRM* is a separate module from the query processor module, thus, largely not requiring any modifications to the DSMS query optimizer and query executor modules.

```

QRA_SA-CQP (o streaming object)
01 if (o.type == ‘‘security punctuation’’)
02     sp ← o
03     RewriteQuery(sp)
04     if (sp.et == ‘‘immediate’’)
05         ProcessImmediateSp(sp)
06 else if (o.type == ‘‘data tuple’’)
07     process o as in normal continuous query processing

```

Figure 3.11. SA-CQP using *QRA*.

Figure 3.12 shows the query rewriting algorithm used by the *QRM*. Just like in the *SFA* algorithm, the *dsp*s and the *qsp*s are intersected to produce the updated security predicates (Step 2 and 5). In addition to this intersection, *QRA* also intersects the security predicates with the query predicates to produce the final predicates to be used in the query execution (equivalent to  $\varphi_r$  in Figure 3.1) (Step 8). The rewritten continuous query is optimized by the optimizer, and the new plan replaces the previously used execution plan (Step 9).

*QRA* has the advantage of minimizing the need to modify the existing DSMS components, e.g., query algebra, optimization rules and statistics, optimizer and the executor. Since the approach produces a new (rewritten) form of the same query to adapt to dynamic security policies, the existing query processor and the optimizer can be largely re-used (as they are) to implement the SA-CQP. In the next section, we discuss the pros and the cons of the *SFA* and the *QRA* methods in more detail.

```

RewriteQuery (sp -- security punctuation)
01 if (sp.pt == ‘‘qsp’’)
02   security_predicates ← Intersect(sp, Bufferdsp)
03   update Bufferqsp with sp
04 else if (sp.pt == ‘‘dsp’’)
05   security_predicates ← Intersect(sp, Bufferqsp)
06   update Bufferdsp with sp
07 query_predicates ← Find(Bufferquery, GetCurrentQuery())
08 new_Q ← Intersect(security_predicates, query_predicates)
09 new_qp ← Optimize(new_Q)

```

Figure 3.12. Algorithm for query rewriting.

### 3.5.4 Pros and Cons of QRA and SFA

The major difference between the *QRA* and the *SFA* is the abstraction level of the security predicates. In the *QRA*, security predicates are represented as logical conditions of a query. In contrast, in the *SFA*, security predicates are encapsulated in separate physical ( $SS^+$ ) operators in the query execution plan. As a result, this difference contributes to both the pros and the cons of the approaches. The main advantage of the *QRA* is that the existing query processor infrastructure can be largely re-used as it is, since the *sps* are not propagated into the query execution plan. Here, a query plan consist of only “traditional” continuous query operators and the dynamic changes in the access control are implemented by the *Query Rewriting Module* (*QRM*). The *QRM* is nearly all that is needed to be added to the system in this case. Conceptually, the *QRA* treats the existing query optimizer as the “black box” and invokes it as a sub-routine, with the query specification that integrates both the query and the security predicates. Such approach is faithful to the goal of minimizing code changes in the existing systems, but may result in a blow-up in the optimization time by a factor equal to the number of sub-routine invocations. In the worst case, this may happen every time a new *dsp* or a *qsp* arrives to the

Table 3.2  
Default experimental parameters.

Parameter	Value	Description
$dsp/t$	1:10	Average $dsp$ to tuple ratio
$qsp/(dsp/t)$	1:100	Average $qsp$ to ( $dsp$ + tuple ratio)
$\varphi_{ds}$	5 roles	Average size (in # of roles) of $dsp$ s
$\varphi_{qs}$	10 roles	Average size (in # of roles) of $qsp$ s
$P_{ds}$	<i>tuple-level</i>	Data-side policy applicability (i.e., policy level)
$P_{qs}$	<i>role-change</i>	Query-side authorization
$et$	Deferred	Default enforcement semantics
$ W_G $	1000 tuples	Size of $W_G$ window

system. Clearly, the main disadvantage here is that this approach is not very robust to dynamic changes in security. Potentially, every new  $dsp$  and  $qsp$  may lead to the optimizer re-invocation, the query plan rewriting, and the physical query plan migration, thus consuming the precious resources from evaluating continuous queries.

The main advantages of the *SFA* include its high performance and robustness to dynamic changes in security policies. Whenever a new  $qsp$  or  $dsp$  arrives, only the  $SS^+$  operators are affected, to reflect the changes in security policy, and the rest of the query plan does not need to be modified. The *SFA* approach is also more amenable to shared query processing in the case of multiple queries. If queries have the same query predicates (even if their authorizations are different), the processing can be shared with the proper security filters installed before and after the shared sub-part in the execution plan. Introducing new operators into the query algebra and making the existing query operators security-aware, however, brings several disadvantages. The query optimizer must now become aware of these new operators and their semantics, and must also adjust the cost model to reflect the streaming  $sps$ ' statistics and the cost of their processing by  $SS^+$  and regular (now security-aware) continuous query operators. In summary, the codebase of DSMS may need to undergo significant changes to accommodate the security-awareness inside the query processor.



Table 3.3  
Dynamic properties of security policies.

<b>Frequency variation:</b> $(1/1) \rightarrow (1/10) \rightarrow (1/30) \rightarrow (1/50) \rightarrow (1/100)$
<b>Scope variation:</b> $( R  = 1) \rightarrow ( R  = 10) \rightarrow ( R  = 50) \rightarrow ( R  = 100)$
<b>Intersection variation:</b> $(\varphi_{ds \cap qs} = 0) \rightarrow (\varphi_{ds \cap qs} = 0.5) \rightarrow (\varphi_{ds \cap qs} = 1)$

### 3.6 Experimental Study

In this section, we report the results of our experimental evaluation of *FENCE*. The three questions that we address in this section are summarized below:

- How effective is security punctuation mechanism, with embedded into streams (*dsp*s and *qsp*s), compared to alternatives? (Section 3.6.2)
- How do the *SFA* and the *QRA* methods compare against each other, and against the naive approach in terms of query performance? (Section 3.6.3)
- How big is the overhead of access control enforcement relative to the cost of the continuous query execution, i.e., SA-CQP vs. regular CQP? (Section 3.6.4)

#### 3.6.1 Experimental Setup

We have implemented *FENCE* in a prototype DSMS called *CAPE* [8]. All our experiments are run on a machine with Java 1.6.0.0 runtime, Windows Vista with Intel(R) Core(TM) Duo CPU @1.86GHz processor and 2GB of RAM. For the experiments, we consider a geo-social networking application scenario described in Section 3.1. The goal in such application is to enable social networking combined with location-based monitoring without leaking any sensitive information as determined by the users' and the application's policies. For data, we use the *Network-based Moving Objects Generator* [181] to generate data streams with total of 110K moving objects (e.g., people driving in cars with GPS devices, pedestrians walking on the streets with mobile phones) in the city of Worcester, MA USA. We have instrumented the

generated data streams with two additional attributes, namely the “age” and the “interests”. The values for the “age” attribute follow a normal distribution with mean  $\mu = 20$ , and the values for the “interests” attribute are randomly generated out of 10 possible choices, e.g., **dating**, **friendship**, **movies**, etc. Each data tuple also contains a timestamp.

In our setup, we have considered a scenario, where *dsps* are generated by the data providers on their physical devices, and the streaming data arrives to the DSMS with already interleaved *dsps*. To embed the corresponding *dsps* (and *qsps*) into the data streams, we have written a separate *sp generator* application, that, given the different values for the input parameters, produces security policies with desired characteristics. The *dsps* in the data streams describe the *tuple-granularity* access control policies, i.e., a security policy applies to an entire tuple, and thus, implicitly to all of its attributes. We chose the tuple level policy, because it is likely to be the most common granularity of security in such mobile environments. All tuple policies are described by a *single dsp* or a *single qsp*. We decided to represent policies using a single *sp* as this, in our belief, is likely to represent the most frequent case. Occasionally, more complex security policies may require multiple *sps* to represent it. Roles in our experimental setup,  $R_1, R_2 \dots R_n$  represent the various *real-life* “roles” of subjects encountered in a geo-social networking application, e.g.,  $R_1$  may represent a “family member”,  $R_2$  a “friend”,  $R_3$  a “stranger”,  $R_4$  a “co-worker”, and so on.

When determining a query for our experiments, we envisioned a query that allows people to “connect” to each other based on similar interests, proximity in age and current geographic location (for example, to spontaneously meet at a nearby coffee shop). We thus use the following query in our experiments:

```
SELECT * FROM S1, S2, CoffeeShops AS CS
WHERE distance(S1.loc, S2.Loc) < 5 AND
maxdistance(S2.loc, S3.Loc, CS) < 5 AND
```

```
intersect(S1.interests, S2.interests) AND
difference(S1.age, S2.age) < 10
```

This query may be executed by a user in a geo-social networking application, where one stream ( $S1$ ) represents his or her data stream and another data stream ( $S2$ ) of other users.

To simulate dynamic *query-side* security policies, our *sp generator* inserts a new *qsp* periodically into the stream transmitting *qsps*. *qsps* are generated using random role assignments from  $R_1 \dots R_{30}$ . The default *dsp* to tuple ratio is 1:10, which means that there is one *dsp* per 10 tuples. The average policy size in *dsps* is 5 roles. The *qsp* to data ratio is 1:100, which means that for every 100 data stream elements (data tuples and *dsps*), a new *qsp* is generated. The *qsps* depict only role changes. For simplicity, we have omitted the changes in the privileges of the roles that could be specified in the *qsps* (e.g., by an organization). Unless mentioned otherwise, the default parameters and their values used in the experiments are as specified in Table 3.2.

To simulate dynamic changes in security policies, we use our *sp generator* application to imitate the different possible real-life scenarios. The generation of the security policies with changing characteristics is managed as follows: the *sp* generator starts with an initial set of parameters, and over time the parameter values are varied, e.g., for frequency variation, the transition:  $(1/1) \rightarrow (1/10) \rightarrow (1/30) \rightarrow (1/50) \rightarrow (1/100)$  (as illustrated in Table 3.3), which means that initially *sp* to tuple ratio was 1 to 1 (i.e., every tuple has a unique policy), after some time it changes to 1 to 10, and then to 1 to 30 and so on. This process is repeated continuously for the entire query execution duration. The values of *sp* generator parameters are changed every 10K tuples. Other kinds of transitions in the security policies are depicted in Table 3.3.

### 3.6.2 Effectiveness of Security Punctuations

Our first set of experiments compares the performance of the three alternative access control enforcement mechanisms for streaming data: (1) the non-streaming, (2) the tuple-embedded, and (3) the security punctuation-based described in [38] (see Figure 3.13(a) and 3.13(b)). We refer to them in the charts as *non-streaming*, *tuple* and *sp*, respectively. In the case of non-streaming method, the tuple policies arrive to the system separately from the data. We assume that in the non-streaming case, users specify their policies using SQL, and send them separately over the network. When SQL-based access control policies are received, the non-streaming method parses them and stores them in the global policy hash table. For each arriving data stream tuple, the non-streaming method checks this policy table to retrieve the relevant security policies. To perform access control-based filtering, the relevant policies are intersected to determine if the access should be granted. In our setup, we have simulated this approach by using a separate stream of *dsps* (to imitate separately arriving policy specifications), and have introduced a small delay (a few seconds) to account for parsing of SQL. In the tuple-embedded approach, the tuples' schema is extended by adding an additional attribute to store the access control policies of the tuples. Thus, each individual tuple carries its access control policy, i.e., all authorized roles that are allowed to access it. In the *sp* approach, *dsps* are interleaved with the streaming data. For the query-side dynamic policy changes, in all three cases, we use a stream of *qsps* to simulate the changes in the roles of the executing query.

Figure 3.13(a) compares the average output rate for the three alternatives. The bottom axis denotes the *sp* to tuple ratio (e.g., 1/10 means for every 10 tuples, there is an *sp*). As can be seen, the average output rate using the *sp* approach is significantly higher than for the alternative methods (ranging from 30%-55% compared to the non-streaming approach and 8% to almost 70% to the tuple-based approach, with various *sp* to tuple ratios). Obviously, the lower the *sp* to tuple ratio, i.e., the more policies are shared by data tuples, the more advantageous the *sp* approach becomes. The security

policy frequency has almost no effect on the tuple-based approach. This is expected, because access control policies are stored in their entirety in each tuple, regardless of whether a set of consecutive tuples may share the same policy or not. When policies are the same for streaming data tuples, the tuple-based approach is thus significantly “penalized”. Here, the security processing for each tuple is done as if each tuple has a unique policy, and a lot of unnecessary overhead is thus incurred. The other two methods exploit the shared storage and the shared processing of security policies. In the non-streaming method, a single representation of each policy is maintained in the global policy hash table, and the security-related processing can be shared by (multiple) consecutive tuples with the same policy that arrive to the system. In the *sp* model, sharing policies is easy, and since they are already interleaved with the data, processing overhead is minimized (e.g., determining which policies are applicable to which data tuples), since *sps* always precede their associated data.

Figure 3.13(b) shows the average execution cost for the three alternatives, which follows a similar pattern as described above for the average output rate. The non-streaming method has the highest cost, which is somewhat expected. With frequent unique policies in the streams, more processing must be done by this method: policies must be parsed, stored, and when access control must be enforced, the policy table is probed, to find the policies and to intersect them. For the tuple-based method, the cost does not change, but it is higher than for the other two methods when the policies on the data tend to be similar and the *sp* to tuple ratio decreases (e.g., 1/50 case). The *sp* approach has a slightly higher cost for the 1/1 case, as there is a distinct *dsp* for each tuple, consuming more memory resources, and requiring more processing. However, if policies are shared, which is the most likely scenario in a typical application, the *sp* approach can significantly outperform the other two methods.

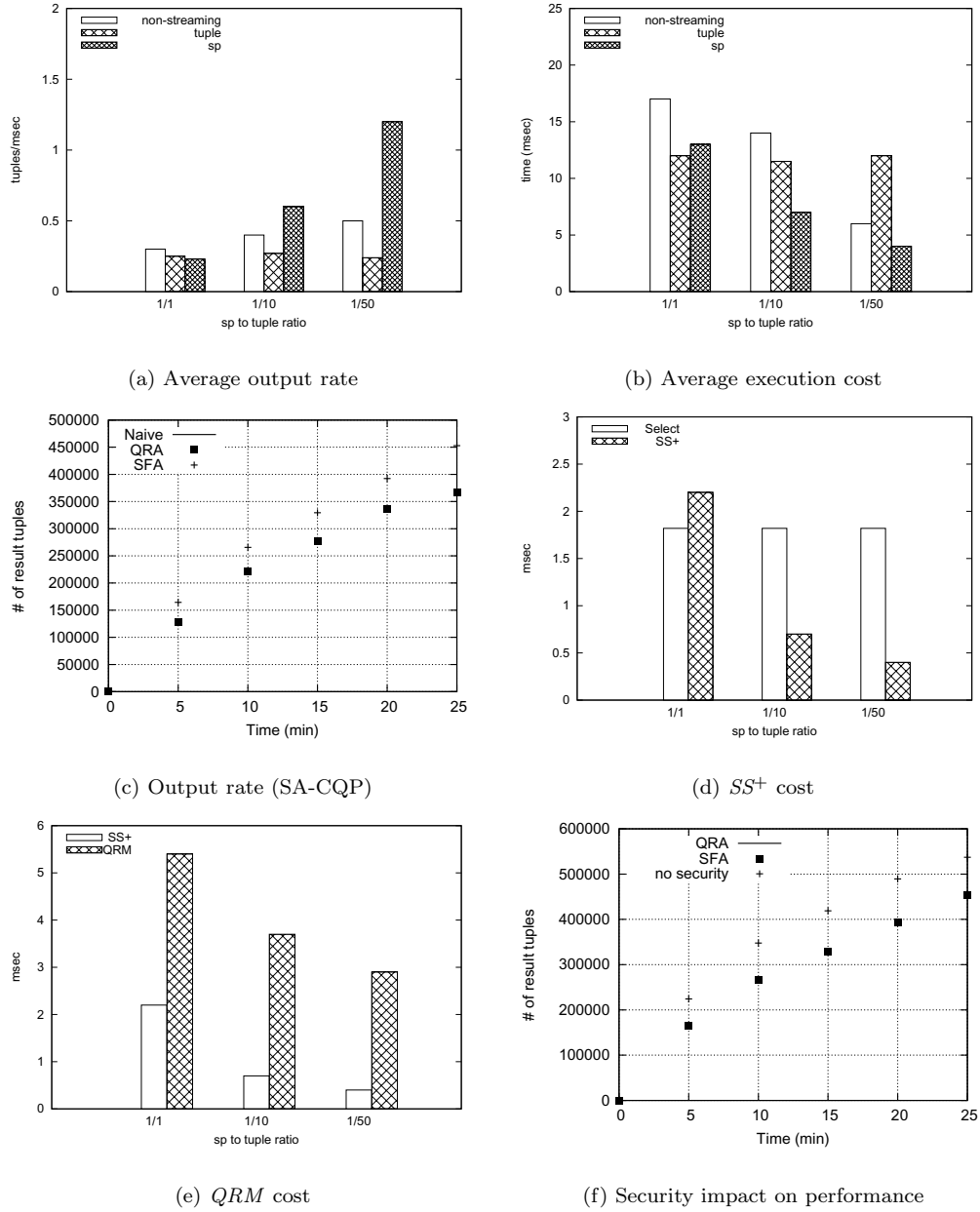


Figure 3.13. Experimental results.

### 3.6.3 Comparison of SA-CQP Methods

In this experiment, we compare the performance of the security-aware query processing alternatives described in Section 3.5, specifically the naive approach, the security filter approach (*SFA*) and the query rewriting approach (*QRA*). In the naive

approach, we have used the post-filtering method, where the access control-based filtering of query results is done after the query execution plan. We ran the query for 25 minutes several times, each time varying the dynamic security characteristics as illustrated in Table 3.3. Figure 3.13(c) shows the total number of result tuples produced over time when using these different query processing strategies. The lines represent the averages over all runs. As a general trend, we have observed that, the *SFA* has about 14 to 22% higher output rate than the *QRA* and 24 to 37% higher than the *Naive* approach. Clearly, we can see that the fixed placement of the security processing, can often limit the continuous query performance, especially if the security conditions are quite dynamic. When using the naive approach, the system cannot adapt to security-related selectivities, and the tuples that could have been filtered earlier may instead be in the query pipelines consuming valuable resources, like memory and CPU.

#### 3.6.4 Overhead of Security Enforcement

In this experiment, we evaluate the “overhead” of continuous access control enforcement on the continuous query performance. We begin by first measuring the processing costs specific to the *SFA* and the *QRA*. Then, we compare the output rate of the “security-free” continuous query execution (as a base case) to the output rates of the SA-CQP methods. Last, we present the overall overhead of these SA-CQP methods relative to the total execution cost of the query.

For *SFA*, we measure the cost of  $SS^+$  execution (Figure 3.13(d)). We have instrumented the  $SS^+$  operator code to measure the time spent to process a *dsp* or a *qsp*. We have also compared it to the cost of the regular select operator as the closest operator to  $SS^+$ . As can be seen from Figure 3.13(d), for the 1/1 *sp* to tuple ratio, the  $SS^+$  cost is close to the cost of the select operator. There is an additional overhead in  $SS^+$  to compute the filtering condition – the predicate based on the arriving *dsp*s and *qsp*s, that is not present in the regular selection and accounts for the disparity

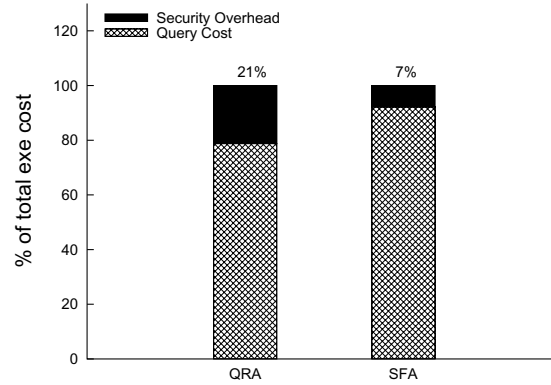


Figure 3.14. Security enforcement overheads.

in the 1/1 case. However, as one can observe, the more tuples share the same security policies, the smaller the overhead of the  $SS^+$  operator becomes compared to the selection.

In Figure 3.13(e), we present our comparison of the security processing cost in the *QRA* method, specifically, the average execution cost of the query rewriting module (*QRM*). Here, we only measure the cost of the query plan rewriting. We also contrast it to the average cost of the  $SS^+$  operator. As it can be seen, the *QRM* execution has a higher overhead compared to the  $SS^+$  execution (in some cases, by as much as 80%). The reason for such big cost discrepancy is that in  $SS^+$  only a policy intersection with the opposite buffer needs to be performed (which in most cases consists of only a single *sp*). Whereas the *QRM* module has to scan through all of the query predicates in the query to determine if the security predicates from the newly arrived *sp* can be combined with any of the existing query predicates. This operation is performed for every arrived *sp*, which explains the big cost discrepancy. The cost of the *QRM* execution can be improved by using more efficient internal data structures (e.g., an index on *sps* or the query predicates), but in the worst case, the *QRM* may still have to scan all query predicates.

To evaluate the impact of the continuous access control enforcement on the query performance, we ran the query with and without security-awareness enabled. Figure 3.13(f) shows the total number of tuples produced over time for both methods com-



pared to traditional continuous query execution as the baseline. We have abstracted the total security overhead in each method and illustrate it in Figure 3.14. As it can be seen, the query rewriting approach, on average, results in 21% overhead compared to the query execution cost, whereas the security filter approach in only 7%. The more frequently *sps* arrive (i.e., the more dynamic the policies are), the larger is the *QRM*'s overhead. In the case of  $SS^+$ , it basically replaces the *dsps* and *qsps* in its buffers and determines the latest policy intersection. If the selectivity of  $SS^+$  has not changed after the policy is updated, no further optimizations to the current execution plan is needed.

### 3.6.5 Summary of Experimental Results

The main results of our experimental study can be summarized as follows:

1. The symmetric security punctuation model with streaming *dsps* and *qsps* significantly outperforms the other alternatives, especially when more tuples share the same security policies.
2. The *SFA* approach results in up to 37% improvement over the naive approach and up to 22% over the *QRA* approach in the execution time and the output rate.
3. The runtime overhead of the continuous access control enforcement cost relative to the query execution cost is at most 21% for the *QRA*-based query processing and only 7% for the *SFA*-based query processing.
4. In general, the ability of the continuous query processor to adapt to not only data-related but also to security-related selectivities can significantly improve continuous query performance.

### 3.7 Conclusion

In this chapter, we have presented our solution to address the problem of continuous online access control enforcement in data stream environments, where both data and query security restrictions may change as the query is being evaluated. Our research motivation comes from the complicated access control requirements inherent in real-time streaming environments in the context of healthcare, location-based services and financial applications. We have proposed the *FENCE* framework, where data and query access control policies are modeled symmetrically using the data and the query security punctuations. We have implemented our proposed *FENCE* framework in a general DSMS prototype. Our experimental results show that our approach has low overhead and is suitable for data stream environments with dynamic security. We believe that our work makes an important contribution to both the databases and security fields in that it is the first to propose and implement a practical approach for online continuous access control enforcement where policies for data and queries may change concurrently.

## 4 TAGGING OF STREAMING DATA

In this chapter, we propose to enrich data streams with a new type of metadata called *streaming tags* or short *tick-tags*<sup>1</sup>. The fundamental premise of tagging is that users can label data using uncontrolled vocabulary, and these tags can be exploited in a wide variety of applications, such as data exploration, data search, and to produce “enriched” (with additional semantics) and thus, more informative query results. We focus primarily on the problem of continuous query processing with streaming tags and the tagged objects, and address the *tick-tag* semantic issues as well as the efficiency concerns. Our main contributions are as follows. We present the *Stream Tag Framework* (or short *STF*) that supports a *stream-centric* approach to tagging, and where *tick-tags*, attached to streaming objects, are treated as first-class citizens. Under STF, users can tag streaming objects at various granularity and can query tags *explicitly* as well as *implicitly* by outputting the tags of the base data together with continuous queries’ results. We have implemented STF in a prototype DSMS, and through a set of performance experiments, we show that the cost of stream tagging is small and the approach is scalable to a large percentage of tagged objects.

The rest of this chapter is organized as follows. We motivate the need to support streaming data tagging in Section 4.1. We give the overview of the Stream Tag Framework in Section 4.2. Section 4.3 describes our tagging model, the streaming tag concept and presents the tag query language called TAG-QL. Section 4.4 describes two tag-based query processing methods, namely the tag-oriented and the tag-aware query processing. Section 4.5 presents the physical implementations of the several key tagging operators.

---

<sup>1</sup>We chose the name “*tick-tags*” to capture the transient nature of attached labels and distinguish them from traditional “*tags*” (e.g., for web pages, images, files) that tend to be static and persistent.

We describe the results of our experimental study in Section 4.6. Finally, we conclude this chapter in Section 4.7.

## 4.1 Tagging in Data Stream Environments

Data streams are common in applications ranging from location-based services and traffic management to environmental and health sensing. Over the past few years, a large amount of research has been devoted to the design and development of DSMSs [4, 8, 43, 176]. With the exception of a few systems [38, 105, 182, 183], most DSMSs assume that data streams transmit exclusively data tuples (without any additional metadata embedded inside streams), and continuous queries are evaluated on the streaming data tuples.

We propose to enrich data stream environments with a special type of metadata called *streaming tags*, or short *tick-tags*<sup>2</sup>. An informal definition of tagging is the process of adding comments or labels to something. The problem of stream tagging is important, because high volume continuous data streams are ubiquitous, and stream processing applications are becoming vital in our every-day life (e.g., real-time traffic monitoring, emergency response and health monitoring). Tags on streaming data can enrich existing stream-based applications, e.g., [23–25], and can enable and inspire novel useful services as described in Section 1.2. We have also described many other ways of leveraging of streaming data tags in Section 1.3.2.

### 4.1.1 Challenges

Due to the inherent characteristics of streaming environments, the tagging of streaming objects is a challenging task. Stream data is typically characterized by large volumes, high input rates, is generated by multiple distributed data sources that rapidly send updates. Processing of continuous queries on streaming data requires

---

<sup>2</sup>The terms “*streaming tags*” and “*tick-tags*” represent the same concept in the context of our work and are used interchangeably.

near-real response time. Yet unlike traditional databases, data is not persistently stored on the server, but rather streamed through the DSMS once and then discarded. A system supporting tagging of streaming data must consider scalability, output rate, latency and resource utilization. To be useful in practice, a tagging mechanism must be able to support a variety of tagging granularities: users should be able to tag streams, tuples, attributes, or specific data values. For instance, if a set of data tuples correspond to a particular physical phenomenon (e.g., a hurricane), then it is useful to tag all those tuples with a single tag. Alternatively, if a particular data value is called into question, users should be able to attach a tag to an individual data value as well. Naturally, the more fine-grained the tagging is, the higher the overhead it may potentially incur. Furthermore, due to the infinite nature of streams and typically long-running continuous queries, frequent changes in data are likely to occur, which translates into possibly very frequent changes in tags' contents and statistics.

#### 4.1.2 Alternative Tagging Methods

To motivate our proposed solution, next we describe several alternative methods that could be used for tagging of streaming data.

- *Table Approach.* One solution is to build a separate global table in DSMS, where all tags arriving via a separate channel (e.g., another stream) are maintained. For each tag, the links to the appropriate streaming data elements in the form of query predicates are stored, as illustrated below:

Tag	Link To Data
"Running"	SELECT measurement FROM HeartRate WHERE time > 9:00AM and time < 9:30AM

Using this method, the tags are maintained separately from the streaming data. As a result, this method may potentially incur significant overheads. All tags arriving to DSMS (separately from the data) must be processed, and for every tag an entry in the central tag table must be created or updated. To identify the

data to which the tags are applicable to, a separate continuous query must be instantiated (in the worst case, one-per-tag) to find the streaming data elements that the tag corresponds to. If there are many tags, this may severely impact the performance of the system, as significant amount of resources would be taken away from evaluating continuous queries. Furthermore, after the steaming data passes through the DSMS, their respective tags must be deleted from the global tag table, thus further increasing the tag-related maintenance overhead.

- *Extended Data Tuples.* An alternative approach to tagging is to extend the schema of streaming data tuples by adding an additional attribute, where the tag information is stored. Here, tags physically are strongly coupled with the streaming data tuples. Although attractive, this approach has several limitations. First, by increasing the tuples' sizes, more memory and processing resources are consumed. Second, tags may apply to a collection of data tuples or data values, but using this method, tags would have to be duplicated, even if several tuples share the same tag. Furthermore and more importantly, when searching for tags or tagged data, every single tuple must be looked at to see if the tag content matches the search predicate. This approach suffers from the same problem as the “non-normalized” representation of data in relational databases, which calls for optimization of design [1].
- *Streaming XML.* Another possible solution for tagging is to exploit streaming XML [184, 185]. XML is human-legible and is designed to be self-describing. This enables the capability to define self-describing data elements by users. However, XML technology is complex and XML query processing (using either XQuery or XPath languages) is not intended to be evaluated over bursty streams. Even with the extensions supporting XML data streams such as [171, 172, 186], continuous processing of frequent XML-based tags is likely to be expensive and can seriously limit the performance of continuous queries.

- *Streaming Tags*<sup>3</sup>. Our proposed solution is to introduce a special type of streaming metadata called the *streaming tags* or short *tick-tags*. *Tick-tags* are embedded inside data streams and uniquely identify the streaming data objects (e.g., tuples, tuple attributes or data values) to which additional semantic labels are being attached. The advantage of the *tick-tag* approach is three-fold. First, *tick-tags* can be shared by several streaming objects, thus reducing memory and processing overheads. Second, *tick-tags*, interleaved with streaming data, facilitate a faster search for the objects they are applicable to. Furthermore, *tick-tags* can be just as dynamic as the streaming data and can be exploited in continuous query optimization similar to data tuples. The query optimizer can determine the best order of operators by considering both the data statistics as well as the streaming tags’ statistics. Finally, if users decide *not* to tag their data, then the data streams are identical to traditional data streams, and the existing query processing solutions for regular streaming environments are applicable as before.

#### 4.1.3 Our Proposed Solution: The Stream Tag Framework

Here, we present the *Stream Tag Framework* (or short *STF*) that provides full-fledged support for tagging of streaming data. In our endeavor, we strive to achieve the following goals.

- *Stream-centric tags*. Tags applicable to streaming objects are not transmitted and stored separately from the actual data, but rather interleaved with the data tuples inside data streams. Streaming tags have a transient nature – they are not stored permanently on the server, but rather “make a one pass” through the system and then may be discarded.

---

<sup>3</sup>We use terms “streaming tags” and “tick-tags” interchangeably. Both mean the same thing in the context of our work.

- *User-centric tags.* Different users may have unique understandings and explanations for the same piece of information, thus it is essential for a tagging framework to support “personalized” tags with respect to data. Users may also want to customize the time setting of their tags – whether they should be attached and be applicable only once or for some time in the near future. We refer to this feature – a *user-centric* tag semantics.
- *Explicit Querying of Tags.* Users or applications should be able to query streaming tags *explicitly*, in an ad-hoc or in a continuous manner. We call this feature – the *tag-oriented query processing* (see Figure 4.1). For example, a location-based application may specify a range query  $Q$ : *Continuously retrieve all streaming tags specified by users in the downtown of City X*<sup>4</sup>. Here the results of the query  $Q$  are characterized by a continuous stream of *tick-tags* that appear in the specified geographic region.
- *Enriched Query Results.* Regular continuous queries can also produce superior (tag-enriched) results [86, 157]. This functionality is enabled by *tag-aware query processing* (Figure 4.1). The goal here is to preserve the tags attached to the original data based on which the query results are computed. For example, if a tag calls into question the veracity of some streaming data value, one would like this information to be available to anyone who sees the results of a query based on this information. The main challenge in this context is to correctly propagate streaming tags through the query plan, while the tags’ corresponding data is being filtered, projected out or joined with other data tuples.
- *Tag Query Language.* Finally, a comprehensive tagging system must provide a high-level language to attach tags to streaming data, to query them or to specify that enriched (with tags) results should be produced for a given continuous query. For this purpose, we introduce a declarative *Tag Query Language* (or

---

<sup>4</sup>Here, we assume that *tick-tags* are attached to streaming data that has a location attribute.



short *TAG-QL*), which provides an intuitive interface for users to perform the above-mentioned actions.

#### 4.1.4 Our Contributions

The contributions of our *Stream Tag Framework (STF)* can be summarized as follows:

1. *Tag Model.* We introduce the notion of the *tick-tag* metadata for tagging various streaming objects (e.g., tuples, data values, etc.). *Tick-tags* are embedded inside data streams and support a wide variety of user-based semantics.
2. *Tag Query Language.* We introduce the *Tag Query Language* (or short, TAG-QL) that enables declarative specification and querying of streaming tags.
3. *Tag-Oriented Query Processing.* In STF, users can attach and *explicitly* query *tick-tags*. We describe the tag-oriented query algebra that enables this functionality.
4. *Tag-Aware Query Processing.* STF also supports *implicit* querying of tags, where continuous query results are enriched with the tags of the base data. We describe the extensions to the continuous query algebra to support the correct propagation of tags in a query pipeline.
5. *Implementation and Experiments.* To illustrate the feasibility, STF has been implemented in a prototype DSMS called *CAPE* [8]. Our experimental analysis shows scalability and benefits of the *tick-tag* approach, and the costs associated with the tag-awareness.

## 4.2 Stream Tag Framework Overview

Figure 4.1 shows a data stream environment with the STF (integrated inside DSMS) and the streaming *tick-tags* embedded inside data streams.

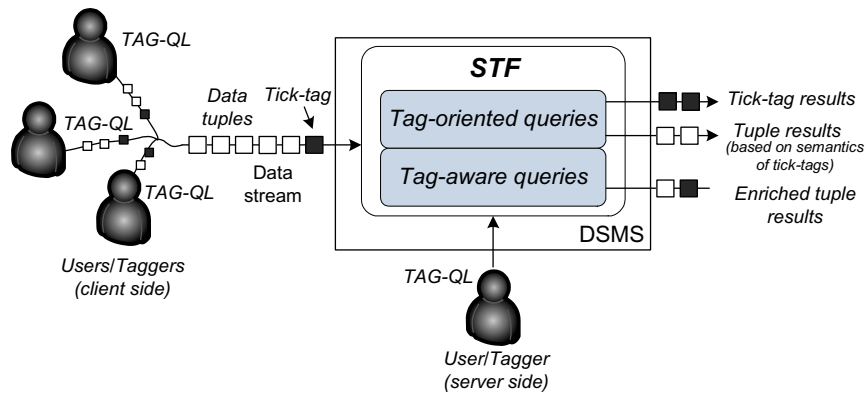


Figure 4.1. Stream Tag Framework (STF) overview.

We consider a centralized DSMS processing long-running queries on a set of data streams. A continuous data stream  $s$  is a potentially unbounded sequence of tuples that arrive over time. Tuples in the stream are of the form  $tuple = [sid, tpid, A, ts]$ , where  $sid$  is the stream identifier,  $tpid$  is the tuple identifier,  $A$  is a set of attribute values, and  $ts$  is the timestamp of the tuple.

A user  $u \in U$  can attach tags  $t \in T$  to streaming objects  $o \in O$  that can be of any granularity. A *taggable streaming object*  $o$  can be a (sub-)stream, a tuple, an attribute of a tuple, or a data value. An object is a piece of data to which additional information (via a tag) can be attached. An object  $o$  can have multiple tags at any given time and can be tagged in two ways: by a user providing the streaming data (on the client side) or by a user of the DSMS querying the streaming data (on the server side). Tagging itself can be performed in an ad-hoc manner, or it can also be continuously executed using a special type of continuous query called the *continuous tagging query* (see Section 4.3.5).

### 4.3 Streaming Tags (or Tick-Tags)

#### 4.3.1 What is a Tick-Tag?

*Tick-tags* are metadata tuples that attach additional information (a keyword, a label or a description) to streaming data objects. *Tick-tags* precede the streaming

objects they are applicable to (i.e., the physical data tuples containing the information to which the tag is attached), and the tuples in data streams are completely unaware of the *tick-tags*. In comparison to traditional keyword metadata, tags are not chosen from a controlled vocabulary defined by a single user, by an organization or by a third party [187, 188]. Instead (as it is also commonly done on the Web), users in their role as taggers can create tags of any content and attach them to streaming objects at any time. As a result, *tick-tags* contribute to a development of a real-time and continuously evolving *folksonomy* [164] – a rich way to characterize real-time data and means to discover interesting things about this data based on exploiting the collective knowledge of possibly many users.

*Tick-tags* have several distinguishing characteristics compared to their traditional counter-parts – the (static) tags used for tagging web pages, images, files, or relational data. Table 4.1 gives a brief comparison of dynamic *tick-tags* against traditional static tags.

Table 4.1  
Traditional tags versus streaming tags.

<i>Property</i>	<i>Traditional Tags</i>	<i>Streaming Tick-Tags</i>
<i>Persistence</i>	Permanent	Transient
<i>Locality</i>	(Most likely) stored separately from data	Interleaved with data
<i>Access</i>	Random access	Sequential access
<i>Input Rate</i>	Low	High
<i>Size</i>	Finite	Potentially infinite
<i>Tag Processing</i>	One-time	Continuous

As one can observe from Table 4.1, *tick-tags* inherit many characteristics of the dynamic streaming data that they are associated with. Namely, they are infinite, they arrive online, stay in DSMS only for a limited time and eventually get discarded by the system.

### 4.3.2 Tick-Tag Physical Design

The physical schema of a *tick-tag* is shown in Figure 4.2<sup>5</sup>.

- *Tagger Identifier (TID)* depicts the source of the *tick-tag*. The id of a tagger is globally unique and is determined by the system.
- *Applicability* describes the stream objects to which the tag is applicable to, e.g., a data value or a collection of tuples. To keep the objects' description compact, *regular expressions* [189] similar to [38, 105] are used in this field.
- *Content* is a string datatype and stores the actual tag value. Given that STF supports an uncontrolled vocabulary, this could be anything: a keyword “Accident”, a description “Nice Weather” or an emotional expression “Happy”, “Sad”.
- *Type* is used by the framework to *classify* streaming tags. There are a number of taxonomies for tags in the literature, e.g., [190, 191]. Although not the primary focus of this paper, we have added this field in the *tick-tag* schema to support future applications, such as reality mining [192] and tag-based data classification [193]. Our current implementation considers the following five types of tags, while the other types and classification algorithms will be a part of our future work:
  - *Objective*: Objective means a description, that does not depend on a particular user. For example, “Bad Smell” is not an objective tag (because one needs to know who thought it was bad), whereas “3 Car Accident” or “Electricity Loss” are objective tags.
  - *Subjective*: Subjective tag implies a personal opinion. For example, “Nice”, “Awful”, “Interesting”.
  - *Physical*: This type of tag describes something physically. For instance, “Broken Light” or “Icy Road”.

---

<sup>5</sup>The fields *not* specified by users are shaded in grey.

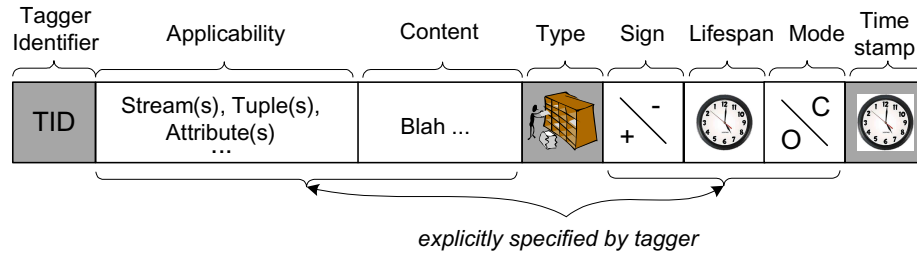


Figure 4.2. *Tick-tag* schema.

- *Acronym*: This type of tag is an acronym or a lingo that might mean various things. For example, “ZZZ” might mean going to sleep, “GFC” – going for coffee, and “911” – emergency or danger.
- *Junk*: The tag is meaningless or indecipherable. For example, “J” or “FJKDSLAD”.
- *Sign*. Since taggers use diversified vocabulary, often it may be difficult to generate an overall opinion or characterization based on the tags’ content [191]. Therefore, we have added a *sign* field to serve as a qualitative description of a *tick-tag*. *Sign* allows tags to rate and express opinions in a more *shareable vocabulary* than conventional tag content. Plus(+) or minus(-) values in the *sign* field easily characterize whether a tag has a positive(+) or negative(-) context. By counting the numbers of positive and negative tags, a representation of the overall opinion (or a “reputation”) or an assessment of the tagged information can be known. Tags without any value in the sign are considered as *neutral* and serve as regular content tags.

For example, consider an online auction system such as *eBay* [194]. This system monitors bids over items available for auction. One can imagine an *Auction* stream containing items to sell with a schema: *Auction*(*seller*, *product*, *product\_features*, *start\_price*, *time*). A collection of tags with different signs applicable to the objects in the stream *Auction* can give various interpretations as shown in Figure 4.3. The system could interpret the collection of positive and negative tags for objects here as:

Auction Schema	seller	product	product_features	start_price	time
<i>tuple</i>	145	Cell Phone G12	Motorolla, Touch screen	\$100.00	3:00:00AM
	10 tags	3 tags	20 tags	40 tags	5 tags
	$\frac{+}{8} \mid \frac{-}{2}$	$\frac{+}{3} \mid \frac{-}{0}$	$\frac{+}{16} \mid \frac{-}{4}$	$\frac{+}{1} \mid \frac{-}{39}$	$\frac{+}{0} \mid \frac{-}{5}$

Figure 4.3. Interpretations based on tag signs.

- “Most people like the seller.”<sup>6</sup>
- “Most people like the features of the product.”
- “Most people don’t like the start price of the product.”

The more tags there are, the more diverse interpretations can be made. A *sign* feature, thus, serves as a “bridge” for many diverse tags making them more shareable and enabling richer tag semantics.

- *Lifespan*. The lifespan of a *tick-tag* is the time interval during which the tag is active. A user specifies for how long (in the near future) the tag should be applicable to a streaming object. After the tag’s lifespan expires, the tag becomes inactive and the system garbage collects it. If only a single instance’s applicability is wanted (i.e., one pass through the system together with the data without any temporary delay in the system), the keyword “I” (meaning “Instant”) is specified in the field.
- *Mode*. The tag mode indicates a user’s preference regarding the combination of the tag with the earlier tags (those tags that are in the system and whose lifespans have not yet expired). “O” indicates “Overwrite”, and “C” means “Combine”<sup>7</sup> respectively. Taggers can specify the mode with respect to *their* tags only, i.e., a user’s tag cannot overwrite the tags generated by other users (taggers) applicable to the same streaming objects. We use *TID* field to track

<sup>6</sup>This could be based on the services or the quality of the products users might have purchased from the seller before.

<sup>7</sup>Different semantics can be used to combine *tick-tags*.

the sources of tags (i.e., the taggers) for this purpose. This field enables users to retract their earlier tags or to add more elaborate descriptions via multiple tags.

- *Timestamp*. Timestamp describes the time when the tag was generated by a user (i.e., the tagger).

### 4.3.3 Tag Query Language (TAG-QL)

To enable users to attach and query streaming tags in an intuitive manner, STF provides a declarative language called the *Tag Query Language* (or TAG-QL for short). The syntax for attaching a *tick-tag* to a streaming object is shown below<sup>8</sup>:

```
ATTACH TAG <tag_content>
TO <object_description>
(WHERE <condition_description>)
(WITH
  TAG_SIGN = < + | - >
  TAG_LIFESPAN = <lifespan_value>
  TAG_MODE = <mode_value>)
```

The `<object_description>` describes the applicability of the tag, namely the object(s) to which the tag is being attached to. The `WHERE <condition_description>` clause is used to describe the conditions that the tagged data must satisfy. Implicitly, the `WHERE` clause also conveys the “location” in the stream, where the *tick-tag* will be inserted. Since *tick-tags* always come before the data they are applicable to, the `WHERE` clause states which data the tag should precede in the stream. The `<condition_description>` can be a simple condition or a nested sub-query. Other TAG-QL statements are illustrated in Table 4.2. We will describe them in detail and give query examples in the rest of the paper.

---

<sup>8</sup>The `WHERE...` and the `WITH...` clauses are optional here.

Table 4.2  
Overview of key TAG-QL statements.

<i>Syntax</i>	<i>Meaning</i>
ATTACH TAG ...	Attaches a tag to a streaming object
SELECT TAGS ...	Selects tags satisfying a search predicate
SELECT TAGGED OBJECTS...	Selects tagged objects
SELECT ... WITH TAGS	Returns tag-enriched query results

#### 4.3.4 Tick-Tag Examples

Here, we present several *tick-tag* examples to illustrate the syntax and the semantics of streaming *tick-tags*. Consider a data stream *Patients*(*sid*, *pid*, *measure*, *loc*, *time*) transmitting the real-time health measurements and the current locations of patients. The following *tick-tags* may be generated<sup>9</sup>:

$t_1$ : ■|-,-,-,loc.value|Panic Attack|■|-|1 min|0|■

represents a tag attached to the current *location* value of the user, and indicates that the user is having a panic attack at her current location. The user feels negative about this experience (- sign), which may also explain the changes in the health measurements (e.g., increase in the heart rate of the patient). The lifespan of the tag is 1 min, and it overwrites any other tags associated with this location value previously sent by the user. This is an example of a tag attached to a specific data value<sup>10</sup>.

$t_2$ : ■|-,-,measure,-|Running|■|+|30 min|0|■

is a tag attached to the heart rate measure attribute in the stream, and indicates that the user is currently running, which is something the user likes to do (as described by the negative '+' sign). The lifespan of the tag is 30 min (possibly indicating how long the user intends to exercise), and it overwrites any other tags associated with

<sup>9</sup>We only illustrate the fields specified by users. The system fields (that are not exposed to users) are denoted by ■.

<sup>10</sup>To attach a tag to an attribute value "<attribute\_name>.value" syntax is used, where <attribute\_name> is replaced with an actual attribute name.



the heart rate measure attribute specified by the user. This is an example of a tag attached to a tuple attribute. Using TAG-QL, the above tags are expressed as follows:

<pre> t<sub>1</sub>      ATTACH TAG           'Panic attack' TO           Patients.loc.value           WITH           TAG_SIGN = '-' AND           TAG_LIFESPAN = 1 min AND           TAG_MODE = OVERWRITE </pre>	<pre> t<sub>2</sub>      ATTACH TAG 'Running'           TO Patients.measure           WITH           TAG_SIGN = '+' AND           TAG_LIFESPAN = 30 min AND           TAG_MODE = OVERWRITE </pre>
---	---

The absence of the `WHERE...` clause in the TAG-QL statements above indicates that there are no constraints regarding which data values the *tick-tags* must precede. Thus, the *tick-tags* will be inserted into the stream interleaved with whatever the tuples happen to be transmitted at the time.

#### 4.3.5 Tick-Tag Generation

Users can create *tick-tags manually* (in an ad-hoc manner) as described above. Alternatively, users can perform *continuous tagging* by instantiating a special type of query, called the *Continuous Tagging Query*. A novel operator, called the *Tagger* operator always exists in such query. This operator consumes an input data stream, continuously evaluates the tagging condition, and produces the corresponding *tick-tags* that get inserted into the output stream and represent the tags being attached to the following after them data. We describe the physical implementation of the *Tagger* operator in Section 4.5. The processing of the tagging query is almost identical to an ordinary continuous query, except that the data tuples in the output stream are now interleaved with *tick-tags*. An example of a continuous tagging query expressed in TAG-QL is shown below:

```

ATTACH TAG 'Dangerous'
CONTINUOUSLY
TO Patients.pid.value

```

```

WHERE (SELECT pid
       FROM Patients
       WHERE measure > 80)
WITH
TAG_SIGN = '-'

```

The keyword `CONTINUOUSLY` in the TAG-QL statement above indicates that the tagging will be executed continuously, that is, a tag will be attached to every patient id (*pid*) value with the heart rate above the specified threshold. Specifically, a *tick-tag* (with the value “Dangerous”) will be created and inserted into the stream ahead of every tuple with the heart rate *measure* > 80 for the entire duration of the tagging query execution.

#### 4.4 Tag-Based Query Processing

We distinguish between two types of tag-based query processing in STF, namely the *tag-oriented* query processing and the *tag-aware* query processing.

##### 4.4.1 Tag-Oriented Query Processing

###### Expressing Tag-Oriented Queries in TAG-QL

In tag-oriented query processing, users or applications query *tick-tags explicitly*. Explicit tag querying is useful for the following two purposes: (1) to locate tags where the tag values themselves are of interest, e.g., *Show me all tags which have a sign = ‘-’*; and (2) to locate tags where the associated base data values are of interest, e.g., *Show me all data tuples that are tagged with the tags that have a value = ‘dangerous’*. Such explicit querying gives the ability to see what other streaming objects have been tagged with the same keyword or a sign, as well as browse through the tags related to the same streaming objects. For specifying such queries, TAG-QL provides “`SELECT TAGS`” and “`SELECT TAGGED OBJECTS`” statements. Queries  $Q_1$  and  $Q_2$  shown in Table 4.3 are examples of such tag-oriented queries.

Table 4.3  
Examples of tag-oriented queries.

<i>Syntax</i>	<i>Meaning</i>
$Q_1$ : SELECT TAGS FROM Patients WHERE OBJECT = Patients.measure AND TAG_SIGN = '-'	Finds all negative tags attached to the heart rate measure attribute in the <i>Patients</i> stream
$Q_2$ : SELECT TAGGED OBJECTS FROM Patients WHERE TAG = 'Emergency'	Returns tuples that contain objects tagged with word 'Emergency' in the stream <i>Patients</i>

### Tag-Oriented Query Algebra

Here, we describe several tag-oriented operators introduced into the continuous query algebra. Let  $t$  denote a tag,  $o$  – a streaming data object,  $T$  – a stream of tags,  $O$  – a stream of data objects, and  $p$  – a search predicate, which can be either on data objects (denoted as  $p_o$ ) or tags (denoted as  $p_t$ ). The following tag-oriented operations are defined in STF<sup>11</sup>:

**Tagger Operator** [ $TO(O, p_o, t) \rightarrow \overset{T'}{\frown} O$ , where  $\forall t_i \in T', t_i = t$ ]. Tagger operator is a unary operator that processes tuples on-the-fly, by attaching a tag  $t$  to an object  $o \in O$ , if  $o$  satisfies the condition  $p_o$ . As a result, the tagger operator inserts a tag  $t$  into the output stream preceding the object  $o$ . Figure 4.4(a) shows an example where the object  $o_2$  gets tagged with the tag  $t$ .

**Tag Selection** [ $TS(\overset{T}{\frown} O, p_t) \rightarrow T'$ ]. Tag selection is a unary operator that returns tags  $T'$  ( $T' \subseteq T$ ) (without their respective objects) that satisfy the tag search predicate  $p_t$ . Figure 4.4(c) illustrates an example, where tags  $t_1$  and  $t_2$  get returned as results by the tag selection operator based on the search predicate  $p_t$ .

---

<sup>11</sup>We denote an object  $o$  tagged with a tag  $t$  as  $\overset{t}{\frown} o$ , and a stream with embedded inside it *tick-tags* as  $\overset{T}{\frown} O$ .

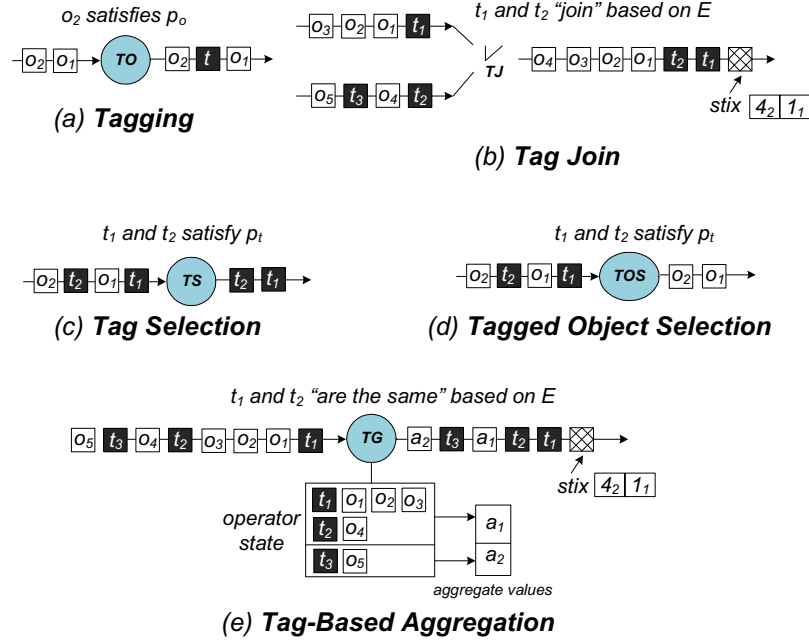


Figure 4.4. Tag-oriented algebra (examples).

**Tagged Object Selection**  $[TOS(\overset{T}{O}, p_t) \rightarrow O']$ . Tagged object selection is similar to tag selection operator ( $TS$ ), except that instead of tags it returns the tagged objects  $O' \subseteq O$ , whose tags satisfy the selection predicate  $p_t$  (for example, objects  $o_1$  and  $o_2$  in Figure 4.4(d) based on their respective tags  $t_1$  and  $t_2$ ). A variant of this operator returns the tagged objects together with their respective tags.

**Tag Join**  $[TJ(\overset{T_1}{O_1}, \overset{T_2}{O_2}, E) \rightarrow \overset{T'}{O'}]$ , where  $T' = E(T_1, T_2) \neq \emptyset$ . Tag join is a binary operator that joins two streams of objects, interleaved with *tick-tags*, based on the *tag join condition*  $E$ . The join condition  $E$  can be a tag equivalence, or a tag similarity, or some other tag join criteria. For example, a *tag equivalence* function  $TE(t_1, t_2)$  checks for the content equivalence of two tags, which can be word-based “Accident” = “Accident”, or semantics-based (e.g., when words mean the same thing) using the system’s dictionary<sup>12</sup>, e.g., “Accident” = “Disaster”, or by co-occurrence, if both tags contain the same objects. One of the simplest co-occurrence methods is using

<sup>12</sup>STF maintains the internal dictionary to support tag classification, tag equivalence and tag similarity function.

*absolute co-occurrence*, that is counting the number of times two tags are assigned to the same object. The similarity can also be estimated based on *relative co-occurrence*, also called “tag overlapping”, and can be measured by *Jaccard coefficient* [195]. If  $A$  and  $B$  are the collections of stream objects described by two tags, relative co-occurrence is then defined as:  $JC(A, B) = \frac{|A \cap B|}{|A \cup B|}$ . That is, relative co-occurrence is equal to the division between the number of objects in which tags co-occur, and the number of objects in which appear any one of two tags. For example, by this definition, the tag “Fireworks” could be equivalent to the tag “Cool”, if users had tagged all objects annotated with the term “Fireworks” with the tag “Cool.”

The result of the tag join operator is a single stream of tagged objects whose tags join based on the join function  $E$ . Figure 4.4(b) shows an example of a tag join operator output. Here, tags  $t_1$  and  $t_2$  from the two input streams join based on function  $E$ , and are sent to the output stream followed by their respective tagged objects  $o_1$ - $o_4$ . Note, that although the tags join, physically they are *not* combined into a single *tick-tag* tuple. The reason for such design is the need to preserve the correct base tags’ semantics. Since the tag join is based on only the *tick-tag*’s content field, we maintain the original *tick-tags* with their values (for attributes like lifespan, sign, mode, etc.) to ensure that the user-specified tag semantics is enforced correctly even after the tags join. After tag join, an additional streaming element is inserted into the stream – the so-called “streaming tag index” tuple (or short “*stix*”). The purpose of the *stix* is to store the references of the joined tags (which are placed consecutively together in the output stream) to their respective base data tuples. In the tag join example in Figure 4.4(b), the *stix* contains the following information:  $\boxed{4_2|1_1}$ <sup>13</sup>. The subscripts (<sub>1</sub> and <sub>2</sub>) are the indicies to the subsequent after the *stix tick-tags*, i.e., “<sub>1</sub>” refers to the first *tick-tag* and “<sub>2</sub>” to the second *tick-tag*, respectively. The values in each index refer to the offsets of the data tuples to which that *tick-tag* applies to. Thus, the above *stix* means the following: the first *tick-tag* applies to the tuples 1-3 ( $o_1$ - $o_3$ ) and the second *tick-tag* applies to the tuple 4 (i.e.,  $o_4$ ).

---

<sup>13</sup>The *stix* illustration should be read from right to left.

The main idea of the tag join operator is to combine two streams of tagged data based on the similarity of their embedded tags'. This operation can be useful for applications searching for related (based on the tags) streaming data that may arrive from various sources.

**Tag-Based Aggregation**  $[TG(\overset{T}{O}, E, G_T^{agg}) \rightarrow \overset{T}{O'}_{G_T^{agg}}]$ . This operator groups objects in a stream by their tags (the groups are based on the tag function  $E$ ) and incrementally updates the value of a given aggregate for each tag-based group (see Figure 4.4(e)). For every arrived tuple, the operator first adds it to the state buffer, and determines which group it belongs to (based on its tag's content and the tag similarity function  $E$ ), and then returns an updated result for this group (preceded by the subgroup's corresponding tags), which is understood to replace a previously reported answer for this group. It may happen that a data tuple may belong to several groups based on the attached tag to it. In this case, the operator picks the "closest" (again based on the function  $E$ ) tag group for the streaming object and updates the answer for that group. Objects without any attached tags can be either completely ignored by the operator or can be placed into a separate "non-tagged" objects group, for which the result is maintained similar to the tag-based groups. In the second case, a dummy *tick-tag* (with an empty content) is inserted prior to sending the answer to preserve correct semantics – to ensure that the earlier outputted tags are not applied to this aggregate answer.

#### 4.4.2 Tag-Aware Query Processing

##### Expressing Tag-Aware Queries in TAG-QL

In addition to the explicit querying of tags, users and applications may find it useful to receive continuous query results that are "enriched" with the tags associated with the original data, based on which the query results were produced. We call this functionality "*implicit tag querying*" and achieve it by performing the tag-aware query processing. To indicate that enriched results should be outputted for a given query,

Table 4.4  
Examples of tag-aware queries.

<i>Syntax</i>	<i>Meaning</i>
$Q_3$ : SELECT pid, loc, time FROM Patients WHERE measure > 80 WITH TAGS	Select-project query that will produce results with interleaved tags of the base data.
$Q_4$ : SELECT A.pid, B.pid FROM Patients A [1 min], Animals B [3 min] WHERE Dist(A.loc, B.loc) < 0.2 WITH TAGS	Join query that will produce results with interleaved tags of the base data.

a user simply adds a “WITH TAGS” statement when specifying a continuous query to the system as depicted in Table 4.4.

One of the immediate challenges in tag-aware query processing is the support for correct propagation of streaming tags through the continuous query pipeline, as data objects are being filtered, joined, or projected out. If one thinks of a tag as a form of mark-up on the streaming data, the key question here is how should that mark-up be transferred into the results of a query. We enable this functionality by adding the tag-awareness to continuous query operators.

#### Tag-Aware Query Algebra

**Projection** is a unary operator that processes tuples by discarding unwanted attributes. This operator simply propagates *tick-tags* and thereafter the projected tuples. If a *tick-tag* applies only to the projected attributes, it is discarded by the project operator as well.

**Selection** is a unary operator that drops tuples that do not satisfy the selection condition. A select operator delays a *tick-tag* propagation until at least one of the

tagged tuples that follow it satisfies the selection predicate. If all tagged tuples are filtered, their corresponding *tick-tag* is discarded then as well.

**Join** is a binary operator that joins the tuples of its input streams. If a tuple joins with another tuple, before being sent to the output stream the tags of the base tuples are physically arranged in a similar fashion as in the tag-oriented join (discussed in Section 4.4.1), with the following two main differences:

1. Since the data tuples (after the join) are physically combined into a single physical tuple, the *tick-tags* attached to the base data tuples will now refer to this (new) joined tuple. This reference is stored in the *stix* metadata tuple (described in Section 4.4.1) that gets inserted prior to the *tick-tags*.
2. In addition to maintaining the tuple-level granularity reference in *stix*, we now also store the references to the joined tuple attributes (that correspond to the base tuples' attributes) that the *tick-tags* apply to.

Figure 4.5 illustrates an example of a tag-aware join output.

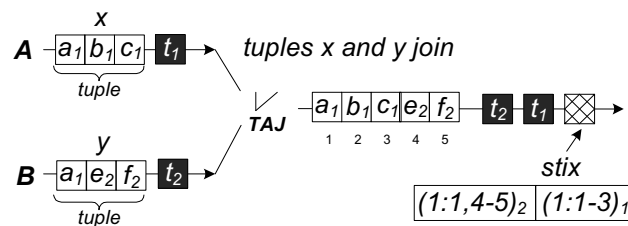


Figure 4.5. Tag-aware join example.

Here tuples  $x$  and  $y$  from streams  $A$  and  $B$  join based on the equality of the first attribute value  $a_1$ . Their respective base tuples' tags  $t_1$  and  $t_2$  precede the join tuple, and the *stix* stores the following information  $(1:1,4-5)_2|(1:1-3)_1$ , where “(1:1-3)<sub>1</sub>” means the first (after the *stix*) *tick-tag*  $t_1$  applies to the first tuple and to the attributes 1-3 in the join tuple. Similarly, “(1:1,4-5)<sub>2</sub>” means that the second *tick-tag*  $t_2$  applies to the first tuple and to the attributes 1, 4 and 5 in the join tuple, respectively.

**Aggregation.** In a tag-aware aggregation operator, each attribute domain is partitioned into attribute sub-groups, where each sub-group contains tuples with the same



attribute value. A result is calculated for each sub-group and then sent to the output stream preceded by the collection of tags that have arrived and have not yet expired from the window and are applicable to *any* object in that sub-group. The motivation for such comprehensive tag propagation is to make all tags associated with the base data (used to compute the outputted aggregate value) available with the aggregate query result.

#### 4.5 Physical Implementation

Here, we describe the physical implementation of two key operators in the tag-oriented algebra, namely the tagger operator and the tag join operator.

**Tagger Operator:** This operator is designed to continuously attach tags to streaming objects that satisfy tagging predicate  $p_o$ . Conceptually, the tagger operator is similar to the selection operator, except that it doesn't discard the tuples that don't satisfy the predicate  $p_o$ , but instead forwards them to the output stream without inserting a *tick-tag* ahead of them. Figure 4.6 shows the pseudocode for the tagger operator execution.

For every arrived data tuple, the tagger operator evaluates the tagging predicate to determine whether the streaming object should be tagged (Line 3). If yes, then a new *tick-tag* is created with the tag properties specified as parameters to the operator (Line 6). The operator assigns the current *query id* as the tagger identifier (*tid*) in the *tick-tag* (Line 5), and the time the *tick-tag* was created is stored in its timestamp field. The newly created *tick-tag* is then forwarded to the output stream followed by the data tuple (Lines 7-8). If the tagger operator receives a *tick-tag* as its input, it simply propagates it to the output stream (Line 11). One of the optimizations that can be employed by the tagger operator (the pseudocode is not shown), is to cache the last outputted *tick-tag*. If the next tuple satisfies the same tagging condition, and the regular expression in the applicability field of the already outputted *tick-tag* is suitable for the new tuple, then no additional *tick-tag* needs to be created and

```

TaggerOperator ( $p_o$  tagging predicate,  $c$  tag content,
 $s$  tag sign,  $l$  tag lifetime,  $m$  tag mode)
01 for (every new element  $e$  received from input stream)
02   if ( $e$  is a tuple) // input is a tuple
03     if ( $e$  satisfies  $p_o$ )
04        $ts = \text{getTime}(\text{now})$ 
05        $tid = \text{getCurrentQueryId}()$ 
06        $t = \text{CreateNewTickTag}(tid, P, c, s, l, m, ts)$ 
07       send  $t$  to output
08       send  $e$  to output
09     else send  $e$  to output
10   else // input is a tick-tag
11     send  $e$  to output // propagate tick-tag

```

Figure 4.6. Tagger operator algorithm.

sent to the output stream. The tuple will simply be forwarded to the output stream. The understanding here is that several tuples share the same *tick-tag* and follow it consecutively in the output data stream.

**Tag Join:** The *TagJoin* algorithm is shown in Figure 4.7. We present the *TagJoin* as a sliding window  $E$ -based join algorithm, where  $E$  is a tag join function (which can be a tag similarity, a tag equivalence function or any other tag join criteria as described in Section 4.4.1). In our pseudocode we describe the nested-loop version of the *TagJoin*. The optimized version of the operator employing an index on tuples and *tick-tags* in the window is a part of our future work. The *TagJoin* maintains a time-based sliding window. We employ a list structure to link all tuples and their *tick-tags* in a chronological order (most recent at the tail). *Tick-tags* are interleaved with tuples in the window, and, thus, the tuple list is “partitioned” by the *tick-tags* into segments, where the tuples in each segment may be tagged by the preceding them *tick-tags*. A collection of tuples between any two non-adjacent collections of *tick-tags*

is called a *tagged segment*. We discuss the processing of tuples and *tick-tags* from the input stream  $A$ . The processing for input stream  $B$  is similar due to the symmetric execution logic.

*Tag Collection.* As *tick-tags* arrive, they are stored in the sliding window. They represent the labels (annotations) for the upcoming data tuples (Lines 3-4).

*Invalidation.* When a new data tuple  $e_A$  is retrieved from the input stream  $A$ , it is used to invalidate the expired tuples from the head of the window of the stream  $B$  (Line 7). If all tuples from a tagged segment have been invalidated, their corresponding *tick-tags* are purged from the head of the window as well.

*Probe.* After the invalidation is done, the *tick-tag*(s) preceding the tuple  $e_A$  are used to probe the window of the stream  $B$ . For concreteness of discussion, let's consider there is a single tag  $t_A$  in the window of stream  $A$  that precedes tuple  $e_A$  and represents the tag attached to  $e_A$ . If  $t_A$  joins with *tick-tag*  $t_B$  from stream  $B$  based on the tag join function  $E$ , the *tick-tags* are placed consecutively together followed by their corresponding base tuples (see Section 4.4.1 for detailed explanation and an example of this step). The *stix* metadata tuple is then created (Line 17) to store the reference to the base tuples and is inserted into the output stream ahead of the "joined" *tick-tags*. The *stix*, the *tick-tags*, and their respective tuples are then forwarded to the output stream (Lines 18-19). If the tag join is empty (i.e., the tags do not join based on  $E$ ), then neither the *tick-tags* nor their respective tuples are forwarded to the output stream.

## 4.6 Experimental Study

### 4.6.1 Experimental Setup

We have implemented our proposed *Stream Tag Framework* in a DSMS prototype called *CAPE* [8]. We execute *CAPE* on Intel Pentium IV CPU 2.4GHz with 2GB RAM running Windows Vista and 1.6.0.0 Java SDK. For data, we use the *Network-based Moving Objects Generator* [181] to generate a moving objects dataset on which

```

TagJoin (A stream, B stream)
01  $W_A \leftarrow$  join time window for stream A
02  $W_B \leftarrow$  join time window for stream B
03 if (a new element  $e_A$  is received from stream A)
04   if ( $e_A$  is a tick-tag) // input is a tick-tag
05     TagCollection( $e_A$ , A,  $W_A$ )
06   else if ( $e_A$  is a tuple) // input is a tuple
07     Invalidate( $e_A$ , B,  $W_B$ )
           // retrieve tags that have arrived prior to tuple  $e_A$ 
08      $T_A \leftarrow$  GetTags( $e_A$ )
09     Probe( $T_A$ , A,  $W_A$ , B,  $W_B$ , E)
10 if (a new element  $e_B$  is received from stream B)
    // Similar to above

```

```

Probe ( $T_A$  - set of tick-tags from the current stream,
A - current stream,  $W_A$  - current stream window,
B - opposite stream,  $W_B$  - opposite stream window,
E - join condition)
11  $T_B \leftarrow$  GetTags( $B[W_B]$ )
12 for (every tick-tag  $t_A \in T_A$ )
13   for (every tick-tag  $t_B \in T_B$ )
14     if (Join( $t_A, t_B, E$ ) // tags join based on E)
15        $S_A \leftarrow A[W_A, t_A]$  // objects tagged by  $t_A$ 
16        $S_B \leftarrow B[W_B, t_B]$  // objects tagged by  $t_B$ 
17        $stix \leftarrow$  CreateNewStix()
18       send stix to output
19       send  $t_A, t_B, S_A, S_B$  tuples to output

```

Figure 4.7. Tag join operator algorithm.

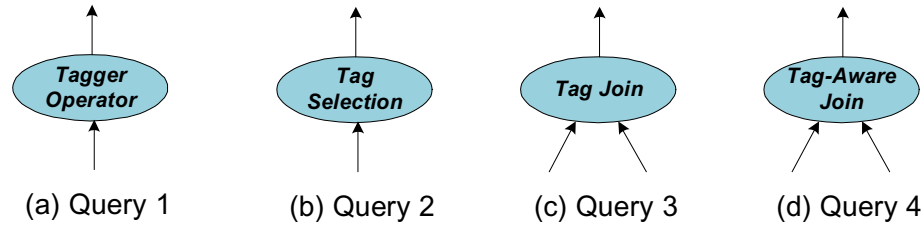


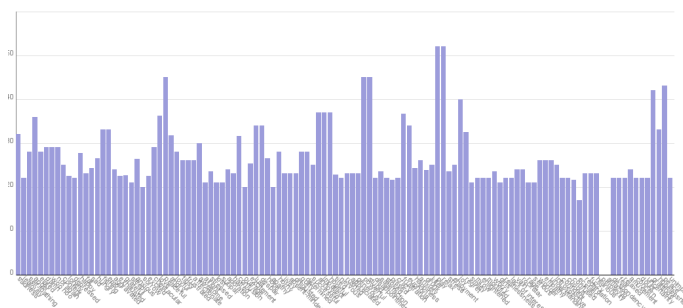
Figure 4.8. Experimental Queries.

we evaluate our experimental queries. The input to the generator is the road map of Worcester county, MA, USA. The output of the generator is a set of objects that move on the given road network and continuously send their location updates. We generate 100K of moving objects, which represent cars, cyclists, and pedestrians. Each tuple in the stream consists of the following fields: update type, object id, report number, object type, timestamp, current location, speed, and the location of the next destination node (see [181] for more details on the generator’s output). We break the moving objects stream up into several streams based on the ids of objects. Such setup simulates objects sending updates to different service providers or base stations and allows us to test join queries. Tuples’ arrival distribution is modeled using a Poisson distribution with a mean tuple inter-arrival rate equal to 10 milliseconds. Unless mentioned otherwise, the tagging is done at the tuple granularity and the *tick-tags* arrive to the DSMS already interleaved with the streaming data. We chose the tuple level, because it is likely to be the most common granularity of tagging. For comparison, we have implemented alternative tagging solutions described in Section 4.1.2, namely the table approach, the extended data tuple approach, and the streaming XML approach. In this section, we refer to them as *TABLE*, *TUPLE*, and *XML* respectively. Our technique is abbreviated as *TICK-TAG*.

For tag content, we employ the “emotion tags” dataset from the *ManyEyes* application [196] supported by IBM. Figure 4.9(a) shows the “tag cloud” for the emotion tags used in our experiments (with more frequent tags depicted in larger fonts). Figure 4.9(b) illustrates the overall tag distribution, and Table 4.5 lists some of the examples of the tag values.

absent-minded adoration affectionate aggravated alright ... amusement **anger** angry annoyed anxiety **anxious** apathy  
 appreciated **awesome** chestacular concern confident **confused** **confusion** content crazy curious damn depressed despair  
 dependency determined disappointment disgusted doubt **drunk** ecstatic elated emo empathy enjoyment enlightened euphoria **excited**  
**excitement** exhausted fat fear fishing frustrated **frustration** fuzzy generous girls **grateful** gratitude guilty **happiness**  
**happy** hatred hilarious hope hopeful hopelessness **horny** hungry impatient introspective jealous jivey **joy**  
 joyful killer lack of interest lonely lost **love** lust mortified mrow nervous not again outrage **peace** peaceful ... pissed play powerful rage  
 redardation **regret** relax **relaxed** restless rockstar rushed **sad** **sadness** satisfaction self-loathing serious **sleepy** smiley sorrow stoked  
 stress **stressed** stupid subpar **successful** tipsy **tired** uncertainty understanding unimpressed vain wonder woo hoo worry

(a) Tag cloud (emotion tags)



(b) Tag distribution (emotion tags)

Figure 4.9. Tag properties.

Table 4.5  
 Tag examples used in the experiments.

Most frequent emotions tags – <i>ManyEyes</i> dataset [196]	happy, sad, jealous, self-loathing, angry, elated, content, lonely, depressed, frustrated, aggravated, exhausted, grateful, sleepy, anxious, sorry, excited, anxious, loved, peaceful, joyful, tipsy, affectionate, cool, alright, stressed, lost, confused, outraged, despaired, hopeful, sympathetic, relaxed, unimpressed, ...
--	---

Four types of queries are used in our experiments which are depicted in Figure 4.8. Query 1 attaches tags (with values chosen at random) to streaming data tuples, with the tagging predicate being on the current location of moving objects. We use Query 1 to test the performance of our proposed tagger operator. Query 2 selects the tags satisfying the tag search predicate (the predicate is based on two types of emotions: “sad” and “angry” in a specific geographic area<sup>14</sup>) on the incoming stream with already interleaved tags. It is used to test the tag selection operation. Query

<sup>14</sup>A query of the form: “Continuously monitor all emotional tags (attached by people to their real-time information) that fall into ‘angry’ and ‘sad’ categories in downtown” may be executed by the law enforcement authorities to prevent potential violence or accidents in the city.

3 joins two streams of tagged tuples based on the tag equality function  $E$ , which is defined as semantics-based equivalence. Specifically, we have partitioned the emotion tags from the dataset [196] into 5 separate groups based on the type of the emotion, e.g., “happy”, “sad”, “neutral”, “angry”, etc., and in the tag-based join, we perform the join based on whether the tags belong to the same emotion group. For example, if tags “Joyful” and “Excited” belong to the same group “happy emotions”, then the two tags join, if they happen to arrive at the same time and are in the windows of the streams being joined. This type of query may be useful to find people who experience similar emotions at the same time, and can possibly help correlate it to their location or a nearby event. Finally, Query 4 performs a tag-aware join on two incoming streams of location updates based on the mutual proximity of the moving objects, e.g., two objects join, if they are within 0.1 miles from each other. Query 4 is used to test the cost of tag-awareness in the continuous join operator.

The real-life application (based on the data, tag values and queries described above) that we consider in our experimental setting is a *geo-social networking application*, e.g., [18, 197]. Here, users may want to tag their location updates with their emotions to update their friends on their well-being, or possibly look for someone to meet and socialize with in a given geographic area.

#### 4.6.2 Cost of Tagger Operator

Figure 4.10 compares the cost of our proposed *Tagger* operator to the cost of a regular *Selection* operator. In the case of selection operator, we process a regular data stream (without any *tick-tags* interleaved). We use Query 1 (from Figure 4.8) in this experiment. The selection predicate here is equivalent to the tagging predicate. The percentage of tagged objects is varied from 0% – none of the tuples are tagged to 100% – meaning all tuples are tagged (the same selectivity is for the selection operator). We use the selection operator here as the cost baseline to which we compare the tagger’s cost while varying the tagging frequency. The difference between the selection and the

tagger is as follows: (1) the selection operator discards the tuples that don't satisfy its predicate, whereas the tagger operator simply propagates them to the output stream (without tagging); (2) if the predicate is satisfied, the tagger inserts a *tick-tag* prior to the data tuple being tagged, whereas the selection simply propagates this tuple to the output stream. The cost for the selection operator increases, as the selectivity of the operator increases, largely due to more work being done by the operator when more tuples have to be propagated up-stream. Similarly, for tagger operator, as the percentage of objects being tagged increases, the tagger operator's cost increases. This is due to a larger number of *tick-tags* being generated (in the case of 100% tagging - twice as many streaming elements are enqueued to the output stream). The cost of the tagger is larger than of the selection (which is expected), on average, by 1.08x for 0% tagging<sup>15</sup> (when no *tick-tags* are inserted) by 1.8x for 100% tagging (where for every data tuple a *tick-tag* is inserted).

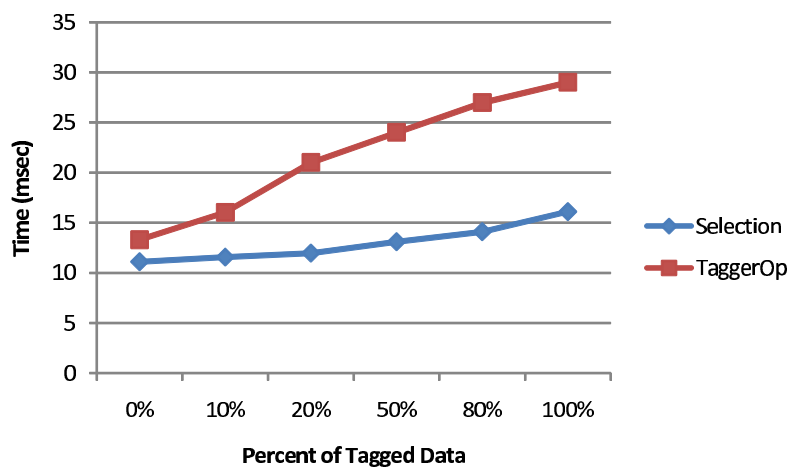


Figure 4.10. Cost of tagging operator.

<sup>15</sup>There are minor execution overheads in the tagger operator that are not present in the selection – e.g., propagation of (un-tagged) data elements upstream.



### 4.6.3 Comparison of Tick-Tag Approach Against Alternatives

The goal in this section is to compare the *TICK-TAG* approach against the alternative tagging solutions (described in Section 4.1.2), namely, the *TABLE*, the *TUPLE*, and the streaming *XML* methods. All these solutions were implemented in *CAPE* and re-use as much of the same code as possible for fair comparison. We use Query 2 (Figure 4.8) in this experiment, and present the average output rate and memory utilization results when performing *tag selection* using these methods. The query is a square region inside which we continuously monitor moving objects' tags. For the focal point of the tag selection query, we choose a random location on the road network (the same for all four cases) and consider it as the center of the query. The focal point the query is static, hence both the *tick-tags* as well as the moving objects that appear in the query region are the same for all four cases. The space is represented as the unit square, the query size is a square region of side length 2.

For *TABLE* tagging approach, we have a separate stream transmitting *tick-tags* that continuously arrive to the system. For every arrived *tick-tag*, we process it by inserting it into the global tag table and initiating an evaluation on the streaming data tuples to determine if the newly arrived tag is applicable to them. For the *TUPLE* approach, we have added an additional attribute in the stream's schema to store the tag value (in addition to all other tag parameters, e.g., mode, lifetime, to be fair in comparison with other approaches). For *XML* approach, we have embedded "xml tags" inside streaming tuples that are interleaved with regular data tuples, and process them when they arrive to the system<sup>16</sup>. Figures 4.11 and 4.12 compare the alternative approaches when varying the percentage of moving objects tagged from 0% to 100% in terms of average output rate and memory usage, respectively.

Figure 4.11 shows the average output rate results for the four different alternatives. In the figure, we measure the average number of result tuples produced per time unit. In Figure 4.12 we measure the memory usage by these different solutions over

---

<sup>16</sup>In the implementation, one XML tag is represented by two physical tuples – one storing the start xml tag and the other – the end tag.

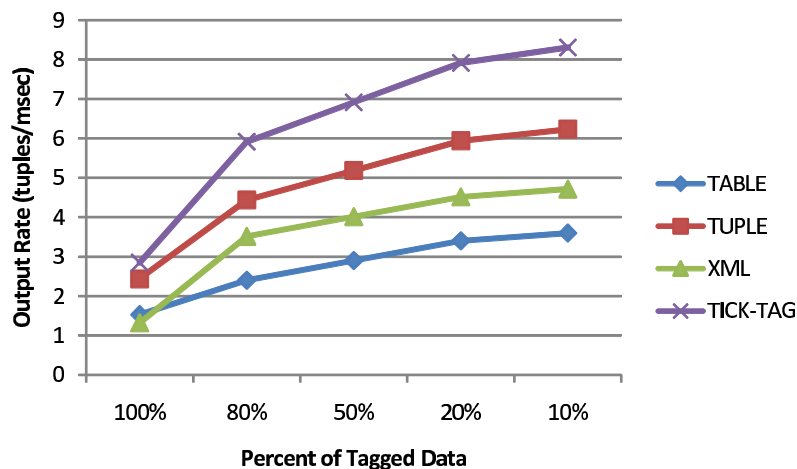


Figure 4.11. Comparison of alternatives (output rate)

time. We can see from both Figures 4.11 and 4.12 that the *TICK-TAG* approach results in higher output rate and smaller memory usage compared to the alternatives. The relative performance of *TICK-TAG* over the other tagging approaches increases with the increase of the number of streaming objects that can share their tags. The main reason is that the search cost of *TICK-TAG* is much lower than updating in search costs (to find the tagged objects) in the *TABLE* approach. Although *XML* and *TUPLE* approaches also take a “stream-centric” approach for tag implementation, they do not exploit the commonalities between the different tuples, and thus result in more memory being used and higher processing cost.

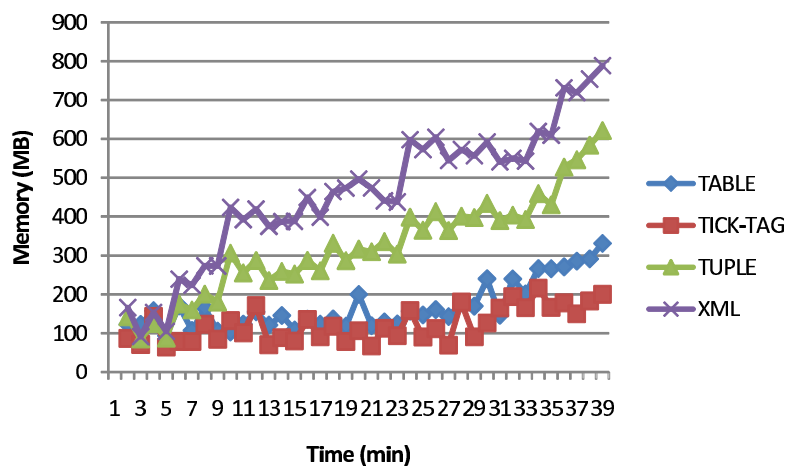


Figure 4.12. Comparison of alternatives (memory)

#### 4.6.4 Cost of Tag Join Operator

In this section we evaluate the cost of the tag join operator again with varying percentage of tagged data in both streams (from 0% to 100%). We use Query 3 in this experiment (Figure 4.8). Sliding windows are time-based and state buffers are implemented as linked lists. The tag join condition is described in Section 4.6.1. Figure 4.13 shows the average cost (computed after several runs) of the *TagJoin*. As

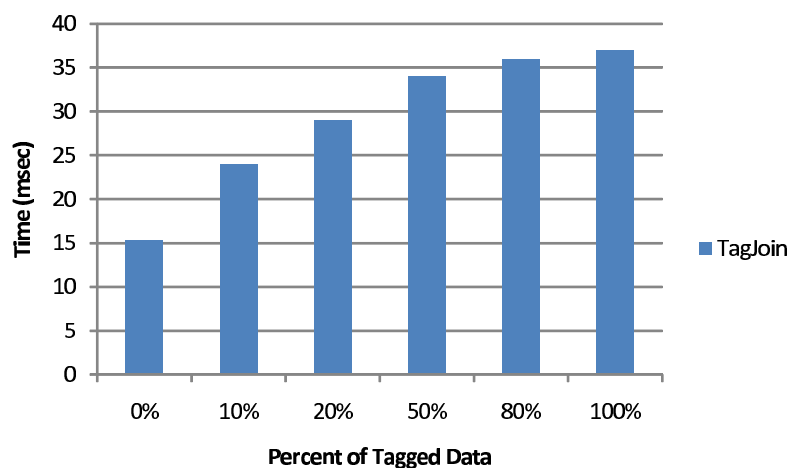


Figure 4.13. Cost of tag join.

can be expected, the cost of tag join increases as the percentage of tagged objects increases by about 65% when 100% of objects are tagged. The more tags are embedded inside data streams, the more overhead is incurred by the tag join. Also, in order to preserve the correct base tag semantics, e.g., tag lifetime, the operator continuously creates *stix* tuples that are used to maintain the references to the original data after the *tick-tags* and their respective data tuples get physically re-arranged in the stream as a result of the tag join. The more tags are interleaved in the streams, the more the join function  $E$  has to be invoked, the more *stix* elements will be created and the more elements will need to be enqueued into the output stream.

#### 4.6.5 Cost of Tag-Aware Join Operator

In this section we compare the cost of the *Tag-Aware Join* operator (described in Section 4.4.2) to a regular join operator using Query 4 from Figure 4.8. The goal of this experiment is to measure the overhead of tag-awareness in a join operator. Figure 4.14 shows the cost of the tag-aware join with respect to the regular join, when varying the percentage of tagged objects. We see that, the larger the number of tagged objects, the higher the cost of the tag-awareness. For 0%, the tag-aware join cost is nearly identical to the regular join operator cost, since there are no *tick-tags* in the streams, and the operator executes just like a regular join. Whereas for 100% of tagged objects, the tag-awareness incurs a “penalty” of processing a larger quantity of streaming *tick-tags*, incurring about 43% of additional cost for 100% of tagging. However, we believe, the case when 100% of data is tagged is highly unlikely in real-life applications. And the average overhead case, between 14%-33% for 20%-80% of tagged data seems to be a reasonable overhead for the added tag-awareness functionality and correct tag propagation through the query pipeline.

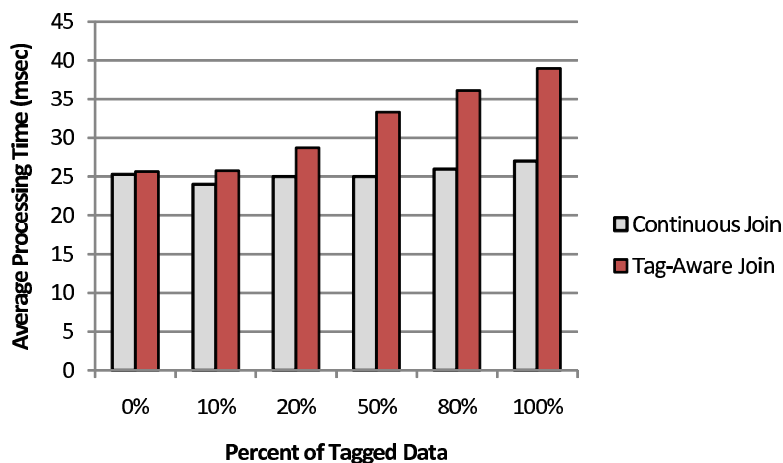


Figure 4.14. Cost of tag-aware join.

## 4.7 Conclusion

In this chapter we have proposed a tagging solution for streaming data using a special type of metadata called the *tick-tags*. *Tick-tags* can serve a variety of purposes, including labelling or describing some underlying real-time information, and serving as means of disseminating useful knowledge in addition to what is captured by the content of data tuples. Our experimental results show the scalability and performance benefits of the *tick-tag* approach compared to alternative solutions. We have also evaluated the costs of executing tag-aware and tag-oriented continuous queries.

## 5 DIVERSITY-AWARE QUERY PROCESSING

In this chapter we present our diversity-aware query processing solution termed the query mesh. We first present the core query mesh framework in Section 5.1. Section 5.6 discusses the self-tuning query mesh for adaptive query processing. Section 5.13 describes the uncertainty-aware query mesh to address the problem of uncertainty and imprecision in the multi-plan-based execution approach.

### 5.1 Core Query Mesh (QM)

#### 5.1.1 Single versus Multiple Execution Plans

As we have previously stated in the introduction, most modern query optimizers determine a *single* “best” plan at compile time for executing a given query [26]. The execution cost for alternative plans is estimated and the one with the overall cheapest cost is chosen. The cost typically is estimated based on the average statistics of the data as a whole as the objective is to find *one plan* for all data. However, significant statistical variations of different subsets of data may lead to poor query execution performance [20]. The main drawback here is the very coarse optimization granularity: *a single execution plan is chosen for all data*. Such “monolithic” approach can miss important opportunities for effective query optimization [20, 28, 29].

**Example:** Suppose we want to monitor stocks that exhibit “bullish” patterns, and appear recently in the news and in the latest “street research” e.g., blogs, popular web sites, etc. We can formulate such query as:

```
SELECT S.company_name, S.symbol, S.price
FROM Stock as S, News as N, StreetResearch as SR
WHERE matches(S.data, BullishPatterns)           /*op1*/
```

```
AND contains(S.sector, News[1 hour])           /*op2*/
```

```
AND contains(S.company_name, StreetResearch[3 hours]); /*op3*/
```

The lookup table *BullishPatterns* contains “bullish” patterns of stock behavior, e.g., “symmetrical triangle” or “falling wedge”, etc. [198]. Operator  $op_1$  performs a similarity-based join on the latest financial data of the incoming stock data tuples with the *BullishPatterns* table. Operators  $op_2$  and  $op_3$  perform the matches on the stock’s sector and the company name with the news sources’ data and the street research data<sup>1</sup>. Let  $c_i$  denote the current average processing cost per tuple for operator  $op_i$ , and  $\delta_i$  denote the current expected selectivity of  $op_i$ . Suppose it is a bull market (i.e., stocks are doing well) and the following conditions hold:  $c_1 > c_2 > c_3$  and  $\delta_1 > \delta_2 > \delta_3$ . Given these statistics, the best ordering of operators to process the data is  $op_3, op_2, op_1$ . Now suppose the news about poor crop harvest (e.g., due to severe weather conditions – floods, hurricanes, etc.) become public. Such news will most likely hurt companies in the agricultural and food sectors. Even if individually a company may be not affected, the price of a stock is often based on the health of its entire sector, and a company’s stock price may go up or down depending on whether investors think its industry will grow or contract. Thus, with the news of poor harvest, the stocks of the agricultural sector will likely result in fewer matches with *BullishPatterns* table) and occasionally may be mentioned in the news and blogs. Thus,  $\delta_1$  is likely to be high, and  $\delta_2$  and  $\delta_3$  still relatively low for such data tuples. So,  $op_1, op_2, op_3$  may be the most efficient ordering for processing stock tuples of this sector. Other sectors, e.g., high-tech, defense or financial services, however, may be completely unaffected, and hence for them  $op_3, op_2, op_1$  will remain the best ordering as before. If the system continues to use the overall statistics, and will still process all data using the  $op_3, op_2, op_1$  plan, this may significantly limit the query performance.

Other real-life examples where subsets of data may have different statistical properties are plentiful. In Internet and communication networks, network traffic tends to

---

<sup>1</sup>The results of such  $n$ -way join queries are frequently consumed by financial monitoring software, e.g., [199] for further analysis.

vary depending on its destination or its type (e.g., voice or multimedia or data). In case of a network congestion, different traffic packets will be discarded by routers with different probabilities. In real-time health monitoring, devices attached to people are likely to produce various values for different patients, depending on their age, gender, weight, etc.

We can observe from the examples above that real-life data tends to be non-uniform, i.e., there may be data subsets with distinct statistical properties, based on either the data content, or the access control policies or based on semantic tags (labels) associated with data tuples. Clearly, a single execution plan often is not likely to be able to serve well many of such rather diverse subsets of data, thus, leading to seriously inefficient query processing for some or possibly huge fractions of the actual data. Most current query optimization approaches do not focus on addressing such *intra-data* variations.

### 5.1.2 Our Proposed Solution: The Query Mesh

The main idea of *QM* is to determine *multiple execution routes* (or execution plans<sup>2</sup>), each optimized for a subset of data with distinct statistical properties. Then, a *classifier* model is inferred based on the computed set of routes<sup>3</sup> and the data characteristics (as depicted in Figure 5.1). The classifier is used for runtime classification of new data tuples to determine the best routes for their processing. While many classification models could be plugged-in into *QM*, e.g., neural networks, naive bayes, etc. [141], we employ a *decision tree* (*DT*) classifier. In our experiments, we have observed that the use of a *DT* classifier approach can “zero-in” on the sought-after route very quickly with typically a small number of comparisons. We describe other beneficial features of *DT* classifiers in Section 5.2.3, thus further justifying our choice.

---

<sup>2</sup>In the context of this thesis, we use the terms “routes” and “plans” interchangeably. Both denote the same concept here.

<sup>3</sup>We also refer to a set of multiple execution routes as a *multi-route configuration*.



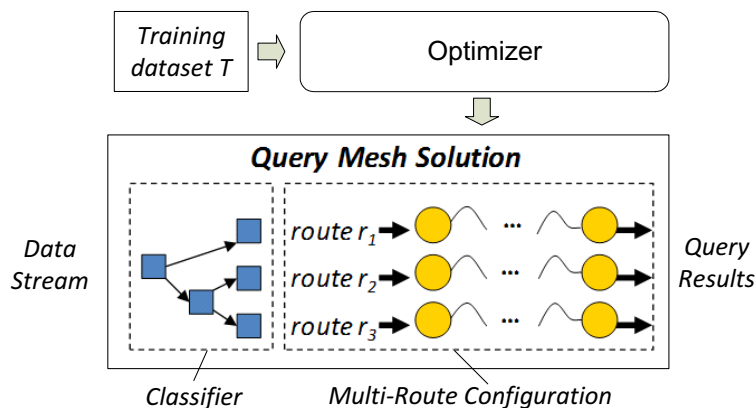


Figure 5.1. Optimizer producing logical  $QM$  solution.

Finding an optimal  $QM$  solution for a given query is an expensive process, largely due to the combinatorial explosion in the search space (Section 5.2.2). We formulate the complexity of the  $QM$  search space (Section 5.2.5), and develop the algorithm  $Opt-QM$  that finds optimal query meshes.  $Opt-QM$ , however, may be not feasible in practice due to its exhaustive nature when enumerating the search space. As viable alternatives, we propose several effective cost-based search heuristics to find good quality  $QMs$  efficiently.

In order to determine the best query mesh, the query optimizer uses a *training dataset*  $T$  (see Figure 5.1) that represents the data and its distribution expected to come in the future – a common approach in many database systems [28,200] and in prediction models in data mining alike [201]. For streaming databases, which are the focus of our paper, relying on samples of data is unavoidable, since it is impossible to “see” all of the streaming data a priori. Since, in addition to multiple execution routes used for query processing, a  $QM$  solution also includes the classifier component, the classifier cost must be considered during query optimization, as classification is now a part of the overall query execution process. The  $QM$  optimization problem can then be stated as follows: *for a given query  $Q$  and a representative dataset  $T$ , the optimizer*

must find a query mesh solution  $QM$  composed of a multi-route configuration  $R$  and a classifier  $C$  that results in the lowest execution cost for tuples in  $T$ .

For query execution, we have implemented a novel runtime infrastructure called the *Self-Routing Fabric (SRF)* that efficiently executes multiple routes in parallel without constructing their physical topologies. Instead, the *SRF* design enables query operators to “self-route” data in a distributed manner with near-zero overhead.

### 5.1.3 Challenges

Several practical issues make this problem challenging: (1) Finding the optimal solution is complex, because there is a combinatorial explosion of all possible execution routes for all possible subsets of training data to consider. (2) Selecting a good quality training dataset is a challenging task. Training dataset’s size and accuracy directly affect the size of the  $QM$  search space and the quality of the resulting  $QM$  solution. With a smaller training dataset, we may be able to enumerate all possible  $QM$  solutions, but the accuracy with respect to the real data may be low. Whereas with a larger training set, the accuracy may significantly improve, but at the cost of an extremely large search space, making it impossible to enumerate all solutions. (3) Most challenging of all is the fact that the classifier model and the number and the choice of particular execution routes are strongly dependent on each other. A change in one component may cause a modification to the other, subsequently affecting the cost of the overall  $QM$  solution. Such interplay between routes and classification introduces a dilemma: how should a  $QM$  solution be computed. Should the training dataset get partitioned first, and then the optimizer would compute the routes for the different partitions? Or alternatively, should some effective routes be computed first, and then the tuples from the training dataset would get assigned to one of the established routes? Clearly, finding a good  $QM$  solution is a complex problem. (4) Finally, query execution with multiple concurrent routes is a challenge. If the

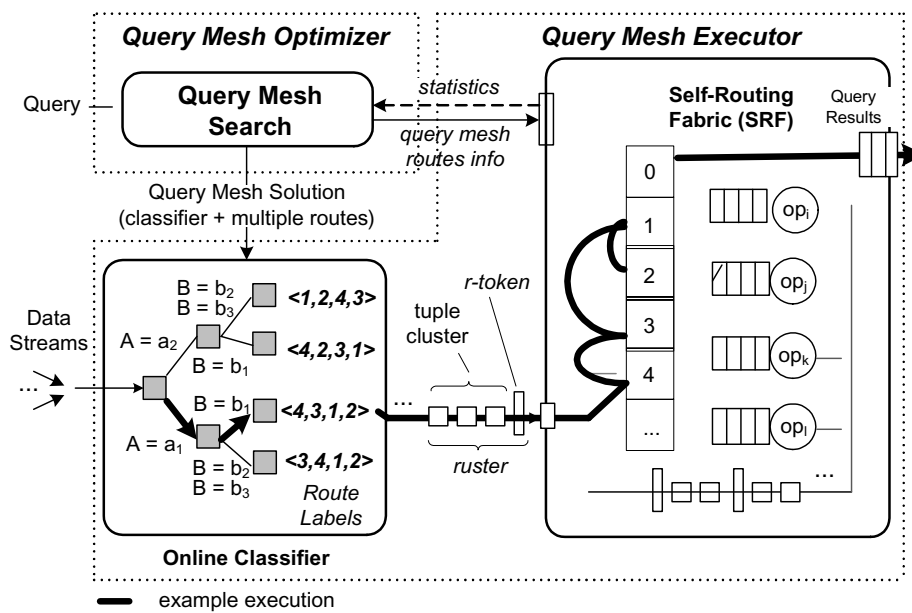


Figure 5.2. Core query mesh framework.

execution infrastructure is not well-designed, the benefits of using multiple distinct routes for different subsets of data may be completely lost.

#### 5.1.4 QM Architecture

The *QM* framework consists of two primary components: the *query mesh optimizer* and the *query mesh executor* (see Figure 5.2). For a given query, the query mesh optimizer computes the query mesh off-line using training tuples and statistics. For every query mesh solution analyzed by the optimizer, the classifier model is induced and the routes are computed. To avoid redundant recomputations, the optimizer employs several caching techniques for sub-components of the query mesh (e.g., routes, or parts of the classifier) that don't change as it traverses the search space. The query mesh executor takes the query mesh configuration produced by the optimizer and instantiates the physical runtime infrastructure. Next, we describe each of the query mesh components more in detail.

### 5.1.5 QM Assumptions

We focus on select-project-join (SPJ) queries in our work, thus, the set of admissible operators in *SRF* includes: selection filter, projection, and a join. To implement joins, we use *one-way-join-probe* (*OJP*) operators inside the *SRF*. *OJPs* are similar in spirit to SteM operators<sup>4</sup> [65], and essentially correspond to a half of a traditional join operator, and are formed over a base stream, supporting the *insert* (build), *search* (probe), and *delete* (eviction) operations for window purging. The choice for such join implementation was largely influenced by our current system implementation [8] and the fact that it is one of the representative approaches of the state-of-the-art. While a number of other alternatives to join implementation are possible, since this was not the focus of our work, we decided to go with the existing system setup. The solution works, and we leave the exploration of other state management techniques in *QM* as a part of our future work.

## 5.2 The Query Mesh Optimizer

### 5.2.1 Data Sampling

The selection of the training data, i.e., which sampling technique should be employed to accurately depict the distribution of the data, is a research topic in its own right. For our work, we have explored several techniques from statistics, including random sampling with cross-validation and sampling with bootstrapping [201]. Using these methods, the system can estimate how well the selected training dataset is going to represent future as-yet-unseen data, and re-sample the data until the desired accuracy is achieved. In practice, the training dataset may also be collected using statistics from previous (historical) execution runs of the query or by employing a similar approach to *plan staging* [76], where optimization and execution are interleaved. The first stage of query processing may use a traditional single plan approach

---

<sup>4</sup>We ensure that no duplicate or missing results are produced similar to SteMs [65].

for query execution, while simultaneously collecting data statistics and training data. Using the samples from the first stage as the training set, a  $QM$  solution can be computed by the optimizer and then used for the execution in the subsequent  $QM$  stage.

### 5.2.2 Query Mesh Search Space

Given a query and collected training data (Figure 2.1), we now study how many possible multi-route configurations (i.e., different routes' combinations) the optimizer may have to enumerate through to find the best one. The expected execution costs of the routes, including the classifier would have to be estimated for these configurations in order to find the optimal  $QM$  solution. Thus, the set of all possible multi-route configurations comprises the  $QM$  search space.

The cardinality of the training dataset has a direct impact on the size of the  $QM$  search space. The larger the cardinality of the training dataset the more possible data subsets, routes, their statistics, and their different combinations may need to be evaluated by the optimizer. Hence, the training dataset must be selected wisely to be compact yet sufficiently representative of the real data. To reduce the size of the search space, we perform *data condensing* [202] on the set of sampled real data tuples. The condensing step aims at selecting a small subset of tuples without a significant degradation in accuracy in order to reduce both storage and processing time. Within the condensing techniques, the approaches can be categorized into two main groups. First, the schemes that select a subset of the original tuples [203, 204], and second the adaptive schemes that modify or generate them [205, 206]. In both cases original data “densities” (i.e., value frequencies) [207] get associated with the condensed training tuples. In our implementation, we went with the second scheme. We keep our discussion on training set condensation brief here. For more details on the condensing algorithms, we refer the reader to [207]. After condensing the training data, each condensed training tuple serves as an abstraction for a subset of

the original sample dataset. In the rest of the paper, we will refer to a condensed tuple as a *training tuple*  $t$  and a reduced dataset that consists of such condensed tuples, summarizing their respective original data tuples, as a *training dataset*  $T$ .

Since routes are computed based on the training dataset  $T$ , the spectrum of possible multi-route configurations ranges from *an individual route per each training tuple* in  $T$  to a *one route for all tuples* in  $T$ . Let  $n$  denote the cardinality of the training tuple set  $T$ , i.e.,  $n = |T|$ . The upper-bound for all possible multi-route configurations corresponds to the number of distinct possible ways of assigning  $n$  distinguishable tuples to one or more routes. The number that describes this value is the *Bell number* ( $B_n$ ) [208], which represents the number of different *partitions* of a set of  $n$  elements. A multi-route configuration, which represents the set of execution routes, is a *partition* of the training tuple set  $T$ , defined as a set of non-empty, pair-wise disjoint subsets of  $T$  whose union is  $T$  (see Figure 5.3). For example,  $B_3 = 5$ , since the 3-element set  $\{1, 2, 3\}$  can be partitioned in 5 distinct ways:  $\{\{1\},\{2\},\{3\}\}$ ,  $\{\{1\},\{2,3\}\}$ ,  $\{\{2\},\{1,3\}\}$ ,  $\{\{3\},\{1,2\}\}$  and  $\{\{1,2,3\}\}$ <sup>5</sup>. The *Bell number* describes the size of  $QM$  search space, i.e., the total number of all possible partitions for an arbitrary training dataset  $T$ . Mathematically, the Bell number is represented as the sum of *Stirling numbers* of the second kind [208]. The Stirling number  $S(n, k)$  is the number of ways to partition a set of cardinality  $n$  into exactly  $k$  nonempty subsets. The problem is clearly challenging, as  $B_n$ 's complexity is exponential as described below:

$$B_n = \sum_{k=1}^n S(n, k) = \sum_{k=1}^n \left( \frac{1}{k!} \sum_{j=1}^k (-1)^{k-j} \binom{n}{k} j^n \right). \quad (5.1)$$

Figure 5.3 illustrates the lattice-shaped  $QM$  search space for a set of training tuples of size  $n=4$ . It also shows two examples of partitions with 2 and 4 routes respectively. The total number of different partitions here equals 15 ( $B_4=15$ ).

Points  $A, B, C$  and  $D$  in Figure 5.3 (on the right) denote four different  $QM$  solutions.  $A, C$  and  $D$  are the *neighbor solutions* to  $B$ , as indicated by the pairwise connecting edges to  $B$ . A single basic transformation e.g., a split of a subset, or merge of two

<sup>5</sup>For brevity, we denote  $\{\{1\},\{2,3\}\}$  as "1/23".

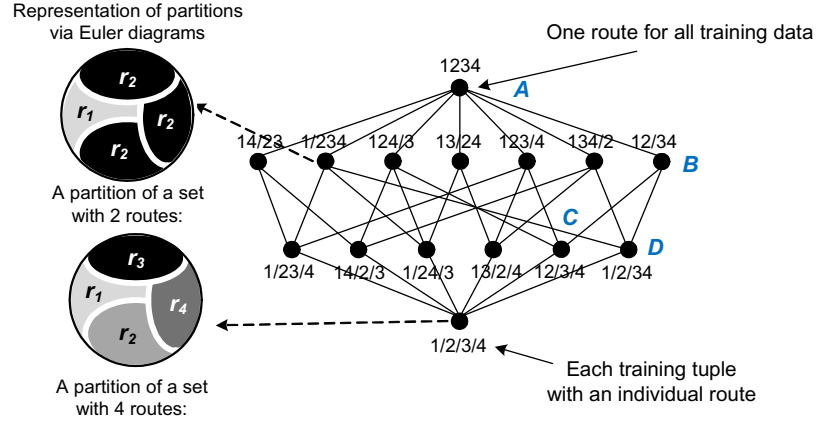


Figure 5.3. Lattice-shaped query mesh search space.

subsets, is needed to transition from a  $QM$  solution to its neighbor, e.g., “12/34”  $\rightarrow$  “12/3/4”.

### 5.2.3 Query Mesh Optimizer Sub-problems

In this section, we first address two main sub-problems of  $QM$  selection. We then proceed with the cost model and the search algorithms used by the  $QM$  optimizer.

**Route Selection Sub-problem:** One of the main sub-components of  $QM$  solution is a multi-route configuration composed of a set of execution routes. Let  $O = \{op_1, \dots, op_n\}$  be the set of operators in a query, where  $op_i \in O$  ( $1 \leq i \leq n$ ) is  $\sigma$ ,  $\pi$  or  $\bowtie$  operators, then a route  $r_q$  depicts an operator ordering  $r_q = \langle op_1, \dots, op_n \rangle$ . Finding the optimal ordering of operators for query optimization is a well-studied topic in database research. In our work, we consider the computation of a single best route (for a subset of data) as a “black box” computation. That is, the optimizer invokes an existing procedure to compute a route based on available statistics using any of the state-of-the-art techniques, e.g., [66, 209, 210]. For example, similar to [58], the best order of operators can be determined by an increasing order of *operator rank*. The rank of an operator  $op_i$  is defined by  $rank(op_i) = \frac{c(op_i)}{1-\delta(op_i)}$ , where  $c(op_i)$  is the cost of operator  $op_i$  and  $\delta(op_i)$  is its selectivity ( $0 \leq \delta(op_i) \leq 1$ ). Alternatively, the optimizer

can also use the dynamic programming [210] or the transformation-based [66] methods to determine the routes.

***Classifier Selection Sub-problem:*** The second main sub-component of *QM* solution is classifier. Inducing classifier model based on the computed routes (as discussed above) and the training data attributes' values is another problem that must be addressed. There is a wide range of classifiers available in the literature, each with its strengths and weaknesses. Determining a suitable classifier for a given problem is still more an art than a science [141]. This is due to the fact that classifier performance and quality depend greatly on the characteristics of the data to be classified [201]. In our work, we employ a *decision tree* (*DT*) classifier. *DT* is attractive for several reasons:

- First, complex decisions can be approximated by the conjunction of simpler local decisions at various levels of the tree.
- Second, in contrast to other classifiers, where each data tuple is tested against all classes, thereby reducing efficiency, in *DT* classifier, a data tuple is tested against only certain subsets of test conditions, thus eliminating unnecessary computations.
- Because most tuple features are deterministic and often common to a group of tuples, a *DT*-based classifier tends to be very efficient (as was also confirmed by our experimental study in Section 5.4).

The algorithm for *DT* induction executes in a top-down recursive divide-and-conquer manner. At the start, all training tuples are at the root. Then, they get partitioned recursively by the tree induction algorithm based on selected test attributes. Test attributes are selected on the basis of the *entropy*-based measure, called *information gain* [141]. The leaf nodes of the *DT* contain the route identifiers for the execution routes that will be assigned to the tuples that reach those leaf nodes after classification. Conditions for stopping *DT* growth are: (1) all training tuples



for a given node belong to the same route, (2) there are no remaining attributes for further partitioning, then *majority voting* [141] is employed for assigning a class label to the leaf, or (3) there are no training tuples left.

#### 5.2.4 Query Mesh Cost Model

Next, we describe the cost model used by the optimizer to compare query mesh solutions when searching for the best one. The expected cost of a  $QM$  consists of three main parts:

(1) *Cost of routes*: Each execution route  $r_q$ , composed of a sequence of operators, has a per-tuple cost  $c(r_q)$  to process a tuple using that route.  $c(r_q)$  represents the expected time to process a single tuple to completion, meaning either to output the tuple as a result or to drop it using  $r_q$ . The cost of  $r_q$  is commonly calculated using two quantities: (i) *cost of operator*  $c_i(op_i)$ , which represents a per-tuple cost of  $op_i \in r_q$ , and (ii) *selectivity of operator*  $\delta(op_i)$ , which is defined as the fraction of tuples that are expected to satisfy  $op_i$ .

(2) *Cost of classification*: Since each arriving tuple must be processed by the classifier, the classification cost must be included in the overall expected processing cost. The classification cost is defined as the cost of a path  $p(DT|r_q)$  from the  $DT$  root to the leaf node with a route label  $r_q$ . The cost of a path  $c(DT|r_q)$  is a function of the number of nodes in the path, combined with the cost of computation at each test node in the path:  $c(DT|r_q) = \sum_{i=1}^{|p(DT|r_q)|} c(node_i)$ .

(3) *Multi-route overhead*: Maintaining multiple execution routes introduces system overhead, e.g., memory, processing, scheduling, etc. For simplicity of presentation, we abstract all overheads associated with a route into a single variable  $OVH$  representing the average overhead per route. The total cost of a  $QM$  can then be described as:

$$cost(QM) = \sum_{q=1}^{|R|} f_q * [c(DT|r_q) + c(r_q|q)] + |R| * OVH. \quad (5.2)$$

where  $q$  represents a distinct subset of data that gets assigned a route  $r_q \in R$ , and  $|R|$  is the number of routes in the  $QM$ ,  $f_q$  is the expected fraction of tuples from the training dataset  $T$  to be processed by a particular route  $r_q$ , and  $c(r_q|q)$  is the expected cost of the route for a subset  $q$ .

### 5.2.5 Optimal Query Mesh Search Algorithm

As the baseline, we now introduce the *Opt-QM* algorithm (pseudo-code shown in Figure 5.4) which is guaranteed to find optimal  $QM$  solution. *Opt-QM* traverses the lattice-shaped search space (see Figure 5.3 for an example) starting from the “bottom” point, where each training tuple has an individual route to the other extreme, where all data is processed using the same (single) route.

In the algorithm in Figure 5.4, first, the *Opt-QM* computes the *power set*  $P(T)$  for the given training dataset  $T$  (Line 3). The power set of  $T$  is the set of all subsets of  $T$ . Using the training dataset, the statistics for the subsets in the power set are calculated (Line 5). The training tuples are processed by the operators, to estimate the operators’ costs and selectivities. The routes are computed similar to the methods discussed in Section 5.2.3 (Line 6). After all possible subsets have been established and the best routes for those subsets have been computed, the algorithm iterates through all possible *partitions* that represent the possible multi-route configurations for the set  $T$ . For each partition  $p$  composed of the subsets  $\{S_i \dots S_j\}$  (Line 8), a union of the routes  $R$  for the subsets is computed (Line 9) and the classifier is induced (Line 10). A new  $QM$  solution is constructed with the routes  $R$  and the classifier  $C$  (Line 11), and its cost estimated (as described in Section 5.2.4). If the new  $QM$  has the smallest cost compared to the solutions seen so far, it is kept (Lines 13-14), otherwise it is discarded (Line 16). After the exhaustive enumeration of all possible configurations, the algorithm returns the  $QM$  solution with the smallest cost as a result.

```

OptQM ( $T$  training dataset)
01  $bestQM = null$ 
02  $bestQMCost = \infty$ 
03  $P(T) = ComputePowerSet(T)$  // compute the power set of  $T$ 
04 for (each set  $S \in P(T)$ )
05    $S.stats = ComputeStats(S)$ 
06    $S.r = BestRoute(S.stats)$  // compute the best route for  $S$  based on  $S.stats$ 
07 repeat
08   let  $p = \{S_i \dots S_j\}$  //  $p$  is a partition of the power set  $P(T)$ 
09    $R = BestRoutes(p)$  // union the best routes for  $\{S_i \dots S_j\}$ 
      // induce classifier based on the training data and the routes
10    $C = InduceClassifier(p)$ 
11    $QM = NewQMSolution(C, R)$ 
12   if ( $QM.cost < bestQMCost$ )
13      $bestQM = QM;$ 
14      $bestQMCost = QM.cost$ 
15   else
16     discard  $QM$  // better  $QM$  has been already found
17 until (all partitions  $p \in P(T)$  enumerated) // total  $B_n$  of them
18 return  $bestQM$ 

```

Figure 5.4. Optimal  $QM$  search algorithm

*Complexity Analysis:* The complexity of the *Opt-QM* is  $O(B_n * E)$ , where  $B_n$  is the *Bell number* (described in Section 5.2.2) and represents the upper-bound of all possible multi-route configurations (query meshes) for the set  $T$ .  $E$  is the time complexity of the “black-box” route computation algorithm and depends on the algorithm employed by the optimizer to find a route, e.g.,  $E = O(n2^n)$  for dynamic programming [210], or  $E = O(n^2)$  for the rank-based ordering algorithm [58, 211]. Clearly with large training datasets, the *Opt-QM* algorithm is not scalable in practice. The problem of finding one optimal route alone is already known to be *NP-hard* [58].

By adding the multi-route factor, the complexity of the problem increases further. Consequently, both the exponential running time and the space requirements provide a strong motivation to design efficient search heuristics as more practical alternatives for query mesh optimization.

### 5.2.6 Query Mesh Search Heuristics

Here we propose a series of cost-based heuristics that find a good quality  $QM$  solution in reasonable time without exhaustive enumeration of the search space. The heuristics have the following three main steps:

- **Step 1:** A *start QM solution* is chosen, its cost is computed, and it is set as the best solution found so far,  $bestQM = QM$ .
- **Step 2:** A *search strategy* is iteratively applied to traverse the  $QM$  search space to find another solution  $QM'$ .
- **Step 3:** The cost of  $QM'$  is computed and compared to the cost of the  $bestQM$  found so far. If  $QM'$  has a smaller cost, the  $bestQM$  is replaced with  $QM'$ . Steps 2-3 are repeated until a *stop condition* is reached.

Although the steps above sound simple, deciding on what is the best strategy for each of these steps is non-trivial. Furthermore, all three steps in unison have a great impact on the quality of the final  $QM$  solution. Typically, after a start solution is chosen, the search strategy performs walks in the search space via a series of “moves”. The number of moves is limited (by the stop condition), and if a poor start solution is picked at the start, the search strategy might not be able to reach a good quality  $QM$  before the search terminates. In the rest of the section, we propose various schemes to address the following questions: (1) How to pick a promising *start QM solution*? (2) What effective *search strategies* can be employed to *improve* the start solution? (3) Finally, when should the search for  $QM$  terminate, i.e., what should be the *stop condition*?

## Selecting a Start Solution

Given that a query mesh model represents an interplay between data values, their frequencies and the best execution routes for their processing, one possible approach to start the search is to pick a start solution based on data content. Given a training dataset  $T$ , a *content-driven approach* (or short *CDA*) partitions training tuples based on the similarity of their values: each attribute domain is partitioned based on the pre-defined threshold (e.g., for a discrete domain it could be a set of values, for a continuous domain it could be a simple range, etc.) that define how “close” the training tuples are to one another based on their content. After the content-based groups have been determined, the best route for each group is computed, and the classifier model induced to complete the  $QM$  solution. The motivation here is that similar content likely means similar selectivities, and thus the same best routes.

An alternative approach is to first compute the routes for each of the training tuples separately<sup>6</sup>. Thereafter, the tuples can be “grouped-by” the similarity of their respective routes, thus forming the groups composed of “route-equivalent” tuples. Lastly, the classifier induction is performed. We call this method the *route-driven approach* (or *RDA*, for short). The *RDA* takes the “reverse” approach compared to the *CDA*. The motivation here is that tuples with different values may still share the same best route. Consider, for instance, moving objects applications. Here, geographically distant areas may still have similar distributions of moving objects, and query processor can thus exploit the same execution route for objects with very different location values.

Other  $QM$  start solution approaches may also include *random-pick* (*RP*), where a  $QM$  with the smallest cost out of  $x$  randomly selected solutions is chosen. Another alternative is to pick a  $QM$  solution, where all data has one route (the top of the lattice in Figure 5.3) and the classifier is empty. This solution is called *extreme-1-*

---

<sup>6</sup>After *data condensing* (described in Section 5.2.2), training tuple abstracts a set of sampled real data tuples. When estimating statistics and computing routes for training tuples, their respective real data tuples are used.

*route* (or *E1R*) approach, and corresponds to a single plan execution strategy, just like in traditional query optimization.

One interesting observation can be made here: the above methods share some similarities with methods in cluster analysis [201]. For example, *CDA* resembles the *partitioning method* in cluster analysis driven by the content. *RDA* resembles the *density-based method* in clustering. Finally, *E1R* can be viewed as the “root” solution in a hierarchical clustering analysis.

The advantage of *CDA* is that it is intuitive, rather simple, and can produce distinct subsets quickly. The disadvantage, however, is that the quantity and the quality of routes are largely dependent on the partitioning function. The advantage *RDA* is that it can find groups of arbitrary “shape” (if represented visually as clusters), i.e., tuples with very different content may still be found to belong to the same group and help create a more efficient *QM* solution. *E1R* works well where the number of clusters is not known in advance, and the algorithm relies on the search strategy (described next) to divide the data into particular sub-sets to determine the next *QM* solution. Based on the resemblance to the cluster analysis techniques, we believe the same guiding principles as when selecting a cluster analysis technique for a given dataset [212] can be applied here when choosing among the strategies for *QM* start solution.

### The Search Strategy

There exists a large number of search algorithms in the literature, each with its strengths and weaknesses. In *QM* for a search strategy, we chose randomized search strategies, e.g., based on iterative improvement, simulated annealing, hill-climbing, etc., which guarantee to find a good *QM* solution in reasonable amount of time [213]. They can serve as viable alternatives to the exhaustive *QM* search. As an example of a search strategy, we illustrate in Figure 5.5 the pseudo-code for the iterative improvement *QM* search strategy (*II-QM*). In our system, we have also

```

II-QM (bestQM a start query mesh solution)
01 bestQMCost = bestQM.cost
02 while (not stop_condition)
03   QM = start solution (e.g., chosen at random,
                        or using methods from Section 5.2.6)
04   while (not local_minimum(QM))
05     QM' = random solution in NEIGHBORS(QM)
06     if (cost(QM') < cost(QM))
07       QM = QM'
08   if (QM.cost < cost(bestQM))
09     bestQM = QM
10 return bestQM

```

Figure 5.5. *II* search strategy for *QM*

implemented *Simulated Annealing QM* strategy (Figure 5.6), where a great deal of random movement in the search space is tolerated.

*Iterative Improvement.* The inner loop of *II-QM* is called a local optimization, which starts at a certain start *QM* solution and improves it by repeatedly accepting random downhill moves (i.e., *QMs* with decreasing costs) until it reaches a local minimum. *II-QM* repeats these local optimizations until a stop condition is met. Then it returns the local minimum with the lowest cost found. As time approaches  $\infty$ , the probability that *II-QM* will visit the global minimum approaches 1. The procedure is repeated various times, each time starting at a new *QM* start solution<sup>7</sup>, until a stop condition is met. Then the algorithm compares the local minima it found and chooses the solution with the lowest cost. If there were enough repetitions of the first steps, the algorithm has found a solution that is close to the global minimum.

*Simulated Annealing.* In simulated annealing (*SA*) search strategy, initially the “temperature”  $\tau$  parameter is set to high. Thus, a great deal of random movement in

<sup>7</sup>One good strategy in *II-QM* is to try the different start *QM* selection approaches and return the *QM* with the smallest cost after several iterations.

```

SA-QM (bestQM a start query mesh solution)
01 QM = bestQM
    // Choose an initial (high) temperature  $\tau > 0$ 
    // Choose a value for  $\rho$ , the rate of cooling parameter
02 Choose a random neighbour of QM and call it QM'
    // Calculate the cost difference in the query meshes:
03  $\delta = \text{cost}(QM') - \text{cost}(QM)$ 
    // Decide to accept the new query mesh or not
04 if ( $\delta \leq 0$ )
05     QM = QM' // QM' is better than or is the same as QM
06 else
07     QM = QM' with probability  $e^{-\frac{\delta}{\tau}}$ 
08 if (stop_condition is met)
09     exit with QM as the final solution
10 else
11     reduce temperature by setting  $\tau = \rho * \tau$ , and go to Step 2

```

Figure 5.6. *SA* search strategy for *QM*

the search space is tolerated. Over time the “temperature” parameter is lowered, and thus less and less random movement is allowed, until the solution settles into a final “frozen” state. This allows the heuristic to sample the solution space widely when the “temperature” is high, and then gradually move towards simple steepest ascent/descent as the “temperature” cools. Thus the search can move out of local optima during the high temperature phase. The *SA* algorithm accepts a worsening move with a certain probability. This probability declines as  $\tau$  declines, by analogy the randomness in the movements decreases as the temperature falls. When  $\tau$  is small enough the algorithm accepts only the improving moves. Figure 5.6 sketches the pseudo-code for *SA*-based query mesh search strategy.



*Hybrid Search.* Alternatively, we can employ a hybrid search strategy, the pseudocode for which is shown in Figure 5.7. The hybrid algorithm takes three input parameters:  $T$ ,  $m$ , and  $H$ , where  $T$  is the training tuple set,  $m$  describes the cardinality (or the upper-bound) of the subsets to be fully enumerated, and  $H$  is the list of the search heuristics (with their stop conditions) to employ after the full enumeration completes. The main idea of the hybrid is to first start with the full enumeration strategy, then choose the best candidate solution as input to the first search heuristic in the list. Heuristics are then executed one after another, each improving the  $QM$  solution and returning its output as the input to the next heuristic in the list until all heuristics have executed. We denote the size of the training set using  $n = |T|$ . The input parameters  $m$  and  $H$  control the overall search strategy for the best  $QM$ . At one extreme, if  $m = n$ , then the search procedure takes the full enumeration approach to find the optimal query mesh. On the other hand, if  $m = 0$ , the procedure corresponds to a pure heuristic-based search (described in Section 5.2.6). If  $m = k$ , where  $0 < k < n$ , then the search procedure first enumerates in a *bottom-up*<sup>8</sup> fashion the query mesh configurations up to the size  $k$  of subsets. Then the “best so far” query mesh is used as the start solution for the heuristic-based search. The value of  $m$  relative to  $n$  reflects the desired degree of completeness of enumeration. The issue of heuristically determining an appropriate value of  $m$  depends on the characteristics of the training set data. For now, we assume the query mesh optimizer can adjust this value heuristically to vary the nature of enumeration from quick and heuristic to accurate and exhaustive. This covers the entire search spectrum from pure heuristic to the full enumeration search.

### Selecting a Stop Condition

The *stop condition* largely depends on the  $QM$  search strategy employed. In general, query mesh search may stop when either  $k$  iterations have gone by, or the solution did

---

<sup>8</sup>Alternatively, we can take a *top-down* approach where initially a single route is assigned to the entire training set. Subsequently, the training set is broken down into subsets and routes are computed for the subsets.

```

HybridSearch (m subset size or the upper-bound for full enumeration,
T training dataset, H search heuristic)
01 n = |T| // size of the training tuple set
02 if (m = 0)
03     bestQM = RANDOM-PICK // see “Selecting a Start Solution” Section
04 else
05     bestQM = EXTREME-N-ROUTES // see “Selecting a Start Solution” Section
06 while (size of subsets in bestQM ≤ m)
07     execute QM_Exhaustive_Enumeration(bestQM)
08 if (m < n)
09     continue with the heuristic H(bestQM) // where H ∈ {II, SA, ...}
10 return bestQM

```

Figure 5.7. Hybrid search strategy for *QM*.

not improve in the last several rounds indicating that the search process has reached a plateau. Alternatively, the search can be time-bounded or resource-bounded, e.g., when memory or CPU utilization limits are reached.

To summarize, given a finite amount of time, the quality of the resulting *QM* found by a heuristic depends on the start solution, the connectivity of the search space determined by the neighbors of each solution, the search strategy and its step size, the stop condition, and, finally, the quality of the cost model employed by the optimizer.

## 5.3 The Query Mesh Executor

### 5.3.1 Instantiation of Physical Infrastructure

Given a logical query mesh specification (the classifier and the set of routes) found by the optimizer, the *QM* executor takes it as an input and instantiates the physical *QM* runtime infrastructure. The runtime infrastructure consists of the *Online Clas-*

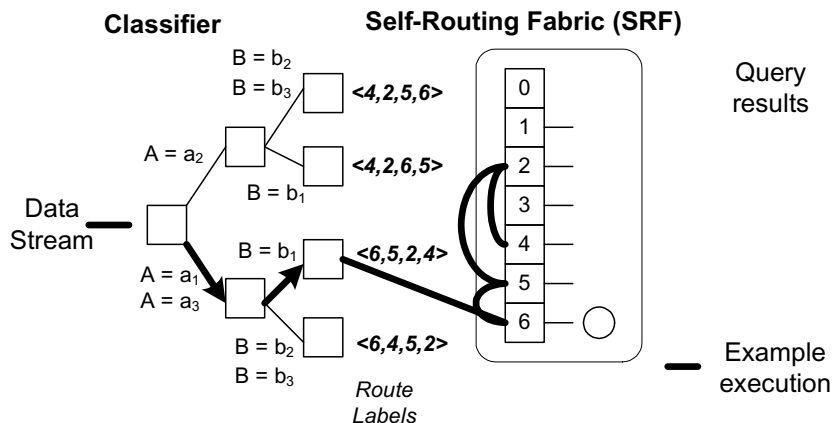


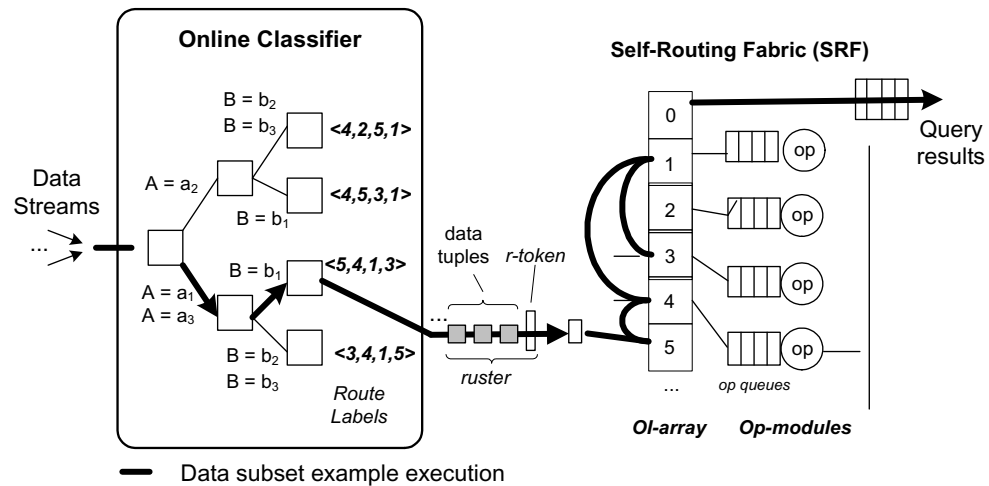
Figure 5.8. Query mesh execution example.

*sifier* operator and the *Self-Routing Fabric (SRF)*<sup>9</sup> infrastructure inside which all the query operators are instantiated (Figure 5.9).

The *SRF* infrastructure consists of two main elements (see Figure 5.9): (1) The *Operator Index Array (OI-array)* which stores the pointers to all query operators. Each index  $i$  corresponds to a unique operator  $op_i$ . Index “0” is reserved for the *SRF* global output queue, where query result tuples are placed to be sent to application(s). (2) The *Operator Modules (Op-modules)* which are the actual operators processing the tuples, e.g., *selection*, *projection*, etc.

Before starting query execution, *SRF* infrastructure must be instantiated. Figure 5.10 shows the pseudo-code for constructing *SRF* for a given query. The algorithm takes as an input a logical specification of *QM* from the optimizer. First it iterates through all operators in the routes of *QM*, and computes a union of all operators denoted by  $O$  (Lines 2-6). Thereafter, the *OI-Array* is instantiated (Line 5) with the “0<sup>th</sup>” index being reserved (Line 6), and a physical instance of each operator is created and assigned to one position in the array (Lines 7-9). The *OPT* hash table stores the mapping between the logical operators and their physical instances (Line 8), and is used at the end of the instantiation to update the logical routes with their physical counterparts (Line 13). To keep the description concise, we skip the pseudo-code for

<sup>9</sup>The name “*Self-Routing Fabric*” was chosen, because the infrastructure enables the operators to self-route the tuples according to their best routes without any central router operator like Eddy.

Figure 5.9. *QM* physical runtime infrastructure.

**ConstructSRF** (*QM* logical query mesh solution)

```

01 OPT = new operator translation hash table
02 for (each  $r \in QM.R$ )
03   for (each op  $\in r$ )
04      $O = Union(op)$ 
05 SRF.OI-Array = new Array[1 + |O|]
06 op_index = 1 // 0-th index is reserved for result tuples
07 for (each op  $\in O$ )
08   OPT[op] = op_index
09   SRF.OI-Array[op_index++] = new PhysOp(op)
10 ReplaceLogicalLabelsInOnlineClassifier(OPT)
11 return;

```

Figure 5.10. Physical instantiation of the *Self-Routing Fabric* infrastructure.

label replacement, but the main idea is to translate and update of the classifier labels with operators' physical ids in the *SRF*. This step is done only once and only in the *SRF* instantiation to eliminate the burden of continuous logical-to-physical operator translation for every classified tuple at runtime.

### 5.3.2 Physical Execution

When new tuples arrive, they first get processed by the online classifier operator to determine the routes to be used for their processing. To keep memory and CPU overhead minimal, the routing decisions are applied at the granularity of groups of tuples, denoted as “*routable clusters*” (or short “*rusters*”), rather than individual tuples. Thereafter, the tuples are forwarded into the *SRF* for the actual query evaluation according to their routes (see Figure 5.9). In contrast to simple tuple batches e.g., in [33], *rusters* are based on the semantics of sharing the same best execution route. Hence, a *ruster* concept is different from both a batch of tuples that happen to arrive contiguously together in time, as well as from a traditional cluster, e.g., grouping tuples based on similar values. Tuples with very different data values may still be assigned to the same *ruster*. Hence, the term “*ruster*” allows us to depict the concept of such tuple grouping precisely.

Tuples are classified using a *tumbling classification window*  $W^{TC}$ . We use a tumbling window, because it partitions a stream into non-overlapping consecutive windows, so that a tuple is classified only once. If tuples within a time window are known to be correlated, then the classification overhead can be minimized by classifying one tuple per window and then sending the rest of the tuples on the same route as the classified tuple. The pre-computed route for a *ruster* is stored in a *route token* (short *r-token*). *R-tokens* are metadata tuples, similar in spirit to streaming punctuations [105] and are embedded inside data streams, thus partitioning the infinite data streams into finite *rusters*. What distinguishes the *r-tokens* from other streaming metadata is that (i) they are “self-describing” as they carry *routing instructions* for streaming data to convey to operators and (ii) the routes in the *r-tokens* are specified in the form of an *operator id stack* based on the design of the *SRF*.

An example of runtime execution is depicted by a thick black arrow in Figure 5.9. Consider an *SRF* with the operator index array as follows:  $OI\text{-array}[1] = op_i$ ,  $OI\text{-array}[2] = op_j$ ,  $OI\text{-array}[3] = op_k$ ,  $OI\text{-array}[4] = op_l$ ,  $OI\text{-array}[5] = op_m$ . Then a

route  $r = \langle op_m, op_l, op_i, op_k \rangle$  will be encoded in an *r-token* as a stack  $\langle 5, 4, 1, 3 \rangle$ , where ‘5’ is the first operator in the route and ‘3’ is the last. The top of the stack represents the index of the operator in the *SRF*. A *ruster* is always routed to the operator that is currently the top node in the routing stack. *OI-array* enables the knowledge of the “location” of other operators. After an operator is done processing the *ruster*, the operator “pops” its index from the top of the routing stack in the *r-token*, and then forwards the *ruster* to the next (now the top) operator. If *all* tuples from a *ruster* are dropped by an operator  $op_i$ , then the *ruster* is not processed any further and its *r-token* is discarded. When the *r-token* operator stack is empty, the *ruster* tuples are forwarded to the global output queue reserved by the index “0” and then to applications.

The novelty of our proposed infrastructure lies in the physical separation between the component that determines which routes should be used for execution and the component that actually physically executes the routes based on the logical specifications. Such architecture easily supports concurrent execution of multiple routes. Furthermore, *SRF* eliminates a central router operator (like Eddy), and removes the “backflow” bottleneck problem present in the systems based on Eddy, where tuples are continuously sent back to the Eddy to determine which operator should process them next [33]. We also believe that our runtime infrastructure offers many other potential benefits for adaptivity and multi-query shared processing.

#### 5.4 Query Mesh Experimental Study

Here, we describe our experimental evaluation of the core query mesh framework implemented inside Java-based continuous query engine called CAPE [8]. To evaluate *QM*, we compare its performance against competitor systems, such as the solution employing “single plan for all data” [62] (SP, for short) and the “multi-route-less systems” [34] (or MR, for short) discovering routes at runtime. In the rest of the section, we will refer to them using their abbreviated names, SP and MR, respectively.

Table 5.1  
Defaults used in the experiments.

Parameter	Value	Description
$D$	<i>Poisson</i>	Default data distribution
$ A $	6	# of attributes in tuple schema
$ S $	1000 tuples	Size of sample tuple set
$ T $	10 tuples	Size of training tuple set (after data condensing)
<i>Start Solution</i>	<i>Route-Driven</i>	$QM$ start solution strategy
<i>Search Strategy</i>	<i>II-QM</i>	$QM$ search strategy
<i>Stop Condition</i>	$k = 3$	# of iterations in the search
$W^{TC}$	1,000 tuples	Classification window size
<i>Ruster size</i>	100 tuples	<i>ruster</i> size

For SP, we use a multi-way join (MJoin) [62] operator. MJoin is a generalization of symmetric binary join algorithms, providing the best plan for each stream, and thus it became our choice for SP (as the closest competitor to query mesh). For MR, we use Eddy framework with CBR routing [28], which is one of the closest approaches to  $QM$ . To ensure the even comparison, all systems were implemented in CAPE, and their implementation used as much of the same codebase and data structures as possible. We use a Round-Robin scheduler in all three systems, which cycles over the list of active operators and schedules the first operator ready to execute. When scheduled, an operator runs for a fixed amount of time bounded by  $|T_{dq}|$ , the number of tuples that an operator may dequeue from its input queue in each execution epoch. Round-Robin was chosen as it has a desirable property of avoiding starvation: no operator with tuples in its input queue goes unscheduled for an unbounded amount of time. In addition to comparison against the alternative systems, we also demonstrate the effectiveness of  $QM$  by measuring its runtime overheads.

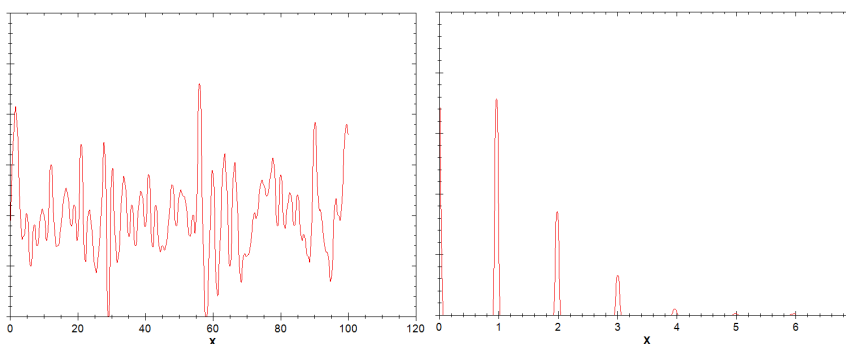


Figure 5.11. Experimental distributions.

Table 5.2  
Distribution statistics.

Data Distributions		
Name	Parameters	Application Examples
<i>Uniform</i>	$\alpha \in \{\dots, \beta-1, \beta\}$ $\beta \in \{\alpha, \alpha+1, \dots\}$ $X \in \{\alpha, \dots, \beta-1, \beta\}$	<ul style="list-style-type: none"> <li>• Long-term patterns of data</li> <li>• Distribution of moving objects in some geographic areas</li> </ul>
<i>Poisson</i>	$0 < \lambda < \infty$ $X \in \{0, 1, \dots\}$	<ul style="list-style-type: none"> <li>• Service times in a system</li> <li>• # people arriving at a counter</li> <li>• # of times a web server is accessed per minute</li> </ul>
<b>Uniform</b> ( $\alpha = 0, \beta = 100$ ): <i>min</i> : 0.0, <i>max</i> : 100.0, <i>med</i> : 49.0, <i>mean</i> : 49.7, <i>ave.dev</i> : 25.2, <i>st.dev</i> : 29.14, <i>var</i> : 849.18, <i>skew</i> : 0.05, <i>kurt</i> : -1.18.		
<b>Poisson</b> ( $\lambda = 1$ ): <i>min</i> : 0.0, <i>max</i> : 7.0, <i>med</i> : 1.0, <i>mean</i> : 0.97, <i>ave.dev</i> : 0.74, <i>st.dev</i> : 1.01, <i>var</i> : 1.02, <i>skew</i> : 1.17, <i>kurt</i> : 1.89		

#### 5.4.1 Experimental Setup

All our experiments are run on a machine with Java 1.6.0.0 runtime, Windows Vista with Intel(R) Core(TM) Duo CPU @1.86GHz processor and 2GB of RAM. We use  $N$ -way join queries in our experiments of the form:

```
select * from S1, S2, S3, ... SN
where S1.col1 = S2.col1 and
```



$S_2.col2 = S_3.col1$  and

$S_3.col2 = S_4.col1$  and ...

$S_{N-1}.col2 = S_N.col1$

Such queries represent a core query type in database and data stream management systems alike and are frequently used to discover correlations across data coming from different sources. In the context of financial applications, such query may be useful for applications making a decision on a stock purchase, e.g., to estimate expected fall/rise in stock prices based on the arriving news. For sensor data, this type of query may be useful to detect fire “hotspots” or for automatic temperature control inside buildings. The specific query we use is an equi-join of 5 streams, i.e.,  $S_0 \bowtie S_1 \dots S_4 \bowtie S_5$ . We use synthetic data sources for our experiments, similar to many other systems’ evaluations e.g., [28, 64, 65]. Using synthetic data allows us to manage data properties that are hard to control in real-life data<sup>10</sup>.

We employ several well-known data distributions to establish the data skew. Specifically, we use *Uniform* and *Poisson* distributions (Figure 5.11 visually illustrates these distributions). These distributions model many real-life phenomena (a few examples are listed in Table 5.2). The default data properties, distribution parameters and system parameters used in the experiments are depicted in Table 5.3 and Table 5.4.

## 5.4.2 Results and Analysis

### QM Optimizer: Effect of the Start Solution

Here, we evaluate the effects of different *start solution* approaches (described in Section 5.2.6) on the structure of *QM* solution when used in a heuristic-based search.

---

<sup>10</sup>We have also experimented with real-life data, and the results were very encouraging: the trends were similar to synthetic data.

The sliding windows in the queries are based on the timestamps present in the data (as opposed to the clock times when tuples arrived to the system during a particular test run). In this way, we ensured that the query answers were the same regardless of the rate at which the dataset is streamed to the system or the order of tuple processing.

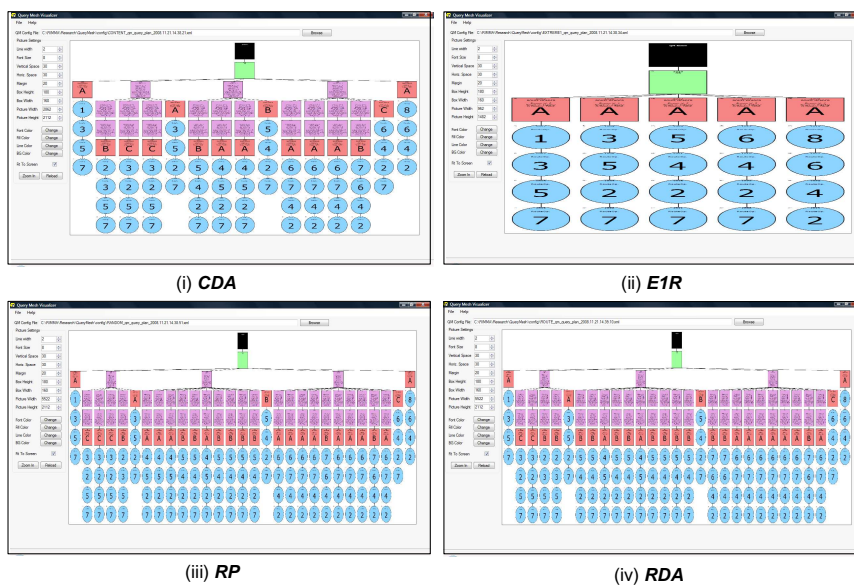


Figure 5.12.  $QM$  with different start solutions.

We provide the qualitative rather than the quantitative analysis here, as it clearer depicts the differences among the approaches. In all cases, the search strategy ran with a single iteration, thus after selecting the start solution the search terminated. We have implemented a  $QM$  visualizer application that given a logical query mesh solution found by the optimizer graphically displays its structure. Figure 5.12 shows the snapshots of  $QMs$  when four different start solutions were used:  $CDA$ ,  $E1R$ ,  $RP$  and  $RDA$ , respectively. Although, it is not possible to show all the details of the classifier nodes and operators in the routes, we thought that showing the overall  $QM$  structure could give the reader a better idea of what a  $QM$  solution may look like. Black node represents the root of the classifier in  $QM$ . Green node is a global OR distinguishing between different streams' data. Purple nodes represent the internal test nodes of the  $DT$  classifier, and red nodes are the leaves of the classifier. Routes composed of query operators are depicted by blue nodes.

Figure 5.13 provides intuitive examples for why different start solutions approaches result in such different  $QMs$ . Here, the data tuples are represented as dots and the dashed lines separate data into subsets with distinct routes according to the approach.

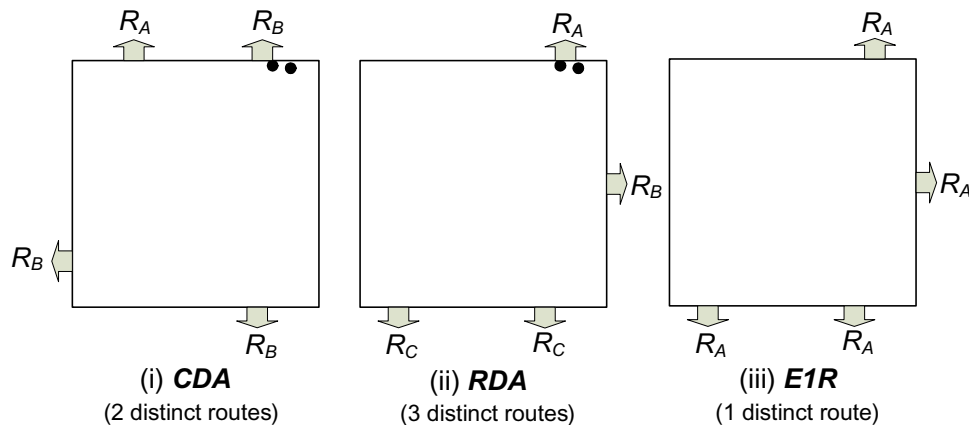


Figure 5.13. Impact of start solutions on routes.

In *CDA*, we partition quantitative attribute domains into ranges, and based on the statistics of different content-based partitions compute routes (in the example, we have two routes  $R_A$  and  $R_B$ ). Here, the subsets of data (and as a result, the final routes and the overall *QM* solution) are largely dependent on the partition function employed. In *E1R*, all data subsets, regardless how distinct they are, will be processed using one route ( $R_A$ ). In *RDA*, distinct subsets will be determined based on similar data statistics (or data “densities”). Clearly, a start solution approach has a great impact on the final *QM* solution, in particular, the count and the quality of the routes found and the classifier model induced.

#### QM Optimization: Effect of the Search Strategy

Here, we evaluate how *QM* solution changes after a search strategy is applied to the start *QM* solution. For this purpose, we have picked two *QMs* produced by *E1R* and *RDA*, as examples, and ran the *II-QM* algorithm with the stop condition  $k = 10$  iterations. Figure 5.14(top) shows the *QM* change with *E1R* start solution and Figure 5.14(bottom) shows the resulting *QM* with *RDA* start approach. As can be seen, the search strategy improves the initially picked *QM* solution, and the resulting *QMs* are

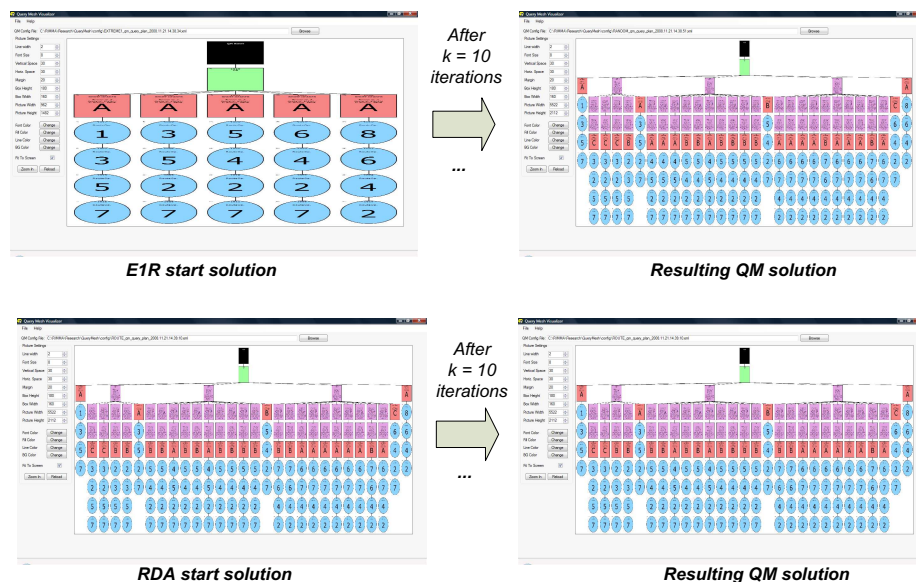


Figure 5.14. Effect of the search strategy.

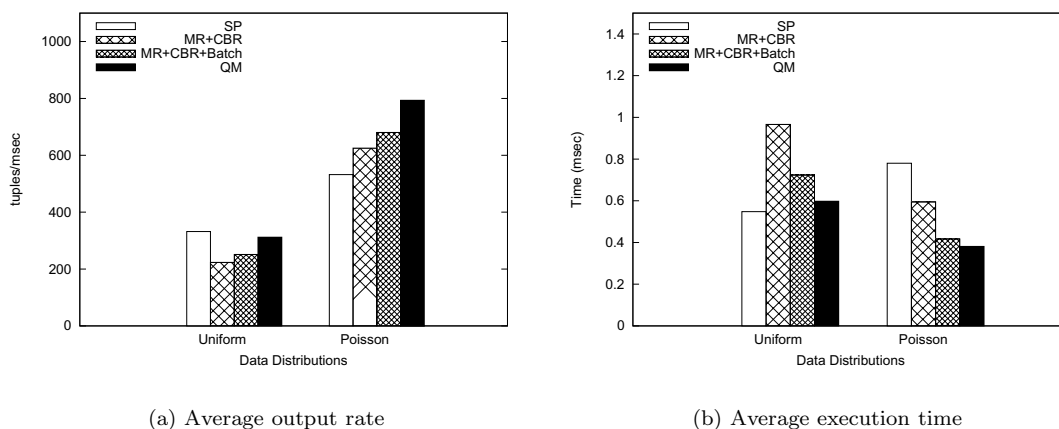


Figure 5.15. Query mesh experimental results.

similar, although not identical (the classifier model varied slightly). Given a limited amount of search time (bounded by our stop condition), still the start solution has a great impact on the final *QM* solution produced by the optimizer and the efficiency of the optimizer to produce good *QMs* quickly.

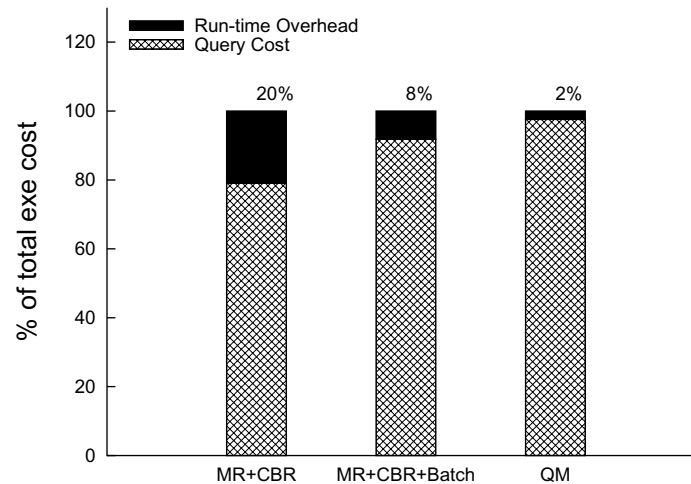


Figure 5.16. Comparison of runtime overheads.

#### QM Optimization: Classifier Structure and Size

Here, we analyze how the classifier model may vary relative to the features of data.

*QM* decision tree classifier consists of a root, a number of branches composed of internal nodes and leaves. *DT* can contain both categorical and numeric (continuous) information in the nodes of the tree. Quantitative data types (ordinal and continuous) are “binned” into categories that are used in the creation of branches – or splits – in the decision tree. Similarly, the categorical data is collapsed into groupings of categories - to enable them to form the branches of the decision tree). We have observed that, on average, the size of the learned decision tree varied between 51 and 70 total nodes (that’s with streams’ schemas consisting of 6 attributes as depicted in Table 5.3), with about 1-6 distinct subsets (described by the number of leaves) per stream and 1-4 unique routes per stream. If the cardinality of the multi-route configuration is 1 (which may happen when all data is uniform or there is a query constraint, e.g., a plan consists of a single operator or a certain operator order *must* be used), then clearly there is no need for a *DT*, hence  $height(DT) = 0$ . Otherwise, the height of the classifier very much depends on the features of the data and the number of routes and

their distribution with respect to the training data. Generalizing about the expected height of a  $QM$  classifier for an arbitrary dataset is extremely challenging. However, implicit preference for a small decision tree is built-in into the classifier induction algorithm by default [141], which guarantees minimum classification cost, regardless of what data properties may be.

#### Query Mesh Execution: Average Output Rate and Processing Time

In this experiment, we compare  $QM$  against competitor approaches, namely SP and MR. We execute MR in two modes: (i) with batching [33] and (ii) without batching. The batch size is set to 100, which is similar to max *ruster* size parameter in  $QM$  (see Table 5.3), and is designed to reduce MR execution overhead. We executed query for 25 minutes several times, using these different solutions, and show the results, averaged over all those runs. Figure 5.15(a) compares the average output rate, the average execution time per tuple is presented in Figure 5.15(b), and the run-time execution overheads present in these systems are in Figure 5.16.

From Figure 5.15(a), we can observe that for Uniform distribution, on average,  $QM$  is between 2.2 - 8% worse than SP in output rate, better by 39% in output rate than MR+CBR without any batching, and better by 24% than MR+CBR with batching. With Uniform distribution, most of the time,  $QM$  uses a single route per stream. Occasionally, due to sampling, we have noticed two routes per stream in  $QM$ . For Poisson distribution, we have observed that  $QM$  on average has 27% higher output rate than MR+CBR without batching, 18% higher than MR+CBR with batching and 44% higher output rate than SP. The average execution time per tuple (in Figure 5.15(b)) follows a similar trend.

The results show that  $QM$  approach does not incur significant performance penalty if datasets are not skewed, and can give great improvements if they are. The “worst case” scenario for  $QM$  is when all data has uniform distribution and no distinct routes are needed. If there are no distinct subsets in the data, no benefit can be gained from

trying to find distinct routes during optimization and during execution, processing *rusters* of tuples that all have the same route.

#### QM Execution: Runtime Overheads

Here, we compare runtime overheads of *QM* and MR<sup>11</sup>. Figure 5.16 reports the overheads per workload of tuples relative to the total execution cost. A workload in this experiment is a set of data tuples received and processed during a time interval of 1 minute. For both MR and *QM*, we considered any execution that is not the actual query processing (i.e., the processing by the query operators on the data) to be a runtime overhead. We have instrumented the code to determine the time spent by each procedure contributing to such overhead in both cases. The relative runtime overhead is depicted in black in Figure 5.16.

MR suffers from continuous re-optimization and re-learning overheads. In MR, Eddy operator continuously profiles operators and identifies “classifier attributes” to partition the data into tuple classes that may be routed differently [28]. Continuous overhead of re-computing classifier attributes based on runtime information may often be unnecessary as the best classifier attribute for an operator does not change very often, as was also indicated in [28]. Furthermore, MR continuously experiences the “backflow” overhead, where tuples get continuously routed back to the Eddy operator that has to re-examine the tuples and forward them to the next operator for processing. The overhead is  $O(n+1)$  time, where  $n$  equals the number of operators and 1 accounts for the first time a tuple from an input stream gets processed. Batching attempts to reduce MR overhead. However, batching in Eddy [33] is still very naive: every  $b$  tuples, i.e., a continuous chunk of tuples that happened to arrive together in time are batched and routed together. Without any batching, the runtime overhead in MR+CBR algorithm amounted to nearly 20% of the total execution cost. These

---

<sup>11</sup>We did not measure SP runtime overheads, as all data is processed using one route here, and thus no overhead regarding how the data should be processed is incurred at runtime.

overheads limit query performance and the benefit that can be obtained from a better adaptive policy in Eddy.

In *QM*, on the other hand, tuples are grouped together into the same *ruster* based on the classification. The classifier model implicitly considers the data values and the similarity of statistics to assign routes to data, and thus tuples classified and grouped into *rusters* are guaranteed to share the same best route. Furthermore, *QM*'s runtime infrastructure based on *SRF* enables the query operators to route data tuples in a distributed fashion nearly overhead-free, thus eliminating the “backflow” overhead problem associated with Eddies. The only small runtime overhead incurred in *QM* is the probing of the classifier to determine the execution plan for arriving data. The classification overhead, however, was measured to be very small, on average, only 2% of the query execution cost (Figure 5.16). The classifier in our case is small in height (maximum 2-3 levels high) and *DT* traversal is rather quick and cheap. Additional system overheads include scheduling an additional operator (i.e., the online classifier) by the scheduler. Since the processing of the tuples by the classifier is fast, the operator, when scheduled, completes its work very quickly giving majority of the execution time to query operators.

We have evaluated the overhead of the online classifier relative to the overall query execution cost. Figure 5.17 shows the classifier overhead for various join queries.

As can be seen, online classification has a very low relative overhead ranging from 2% for a 10-join query up to 4% for a 4-join query. We have also observed that the decision tree classifier tends to be small in height (maximum 2-3 levels high) and *DT* traversal is thus quick and cheap. Additional system overhead of the online classifier corresponds to scheduling an additional operator by the scheduler. Since the processing of the tuples by the classifier is fast, the operator, when scheduled, completes its work very quickly giving majority of the execution time to query operators.



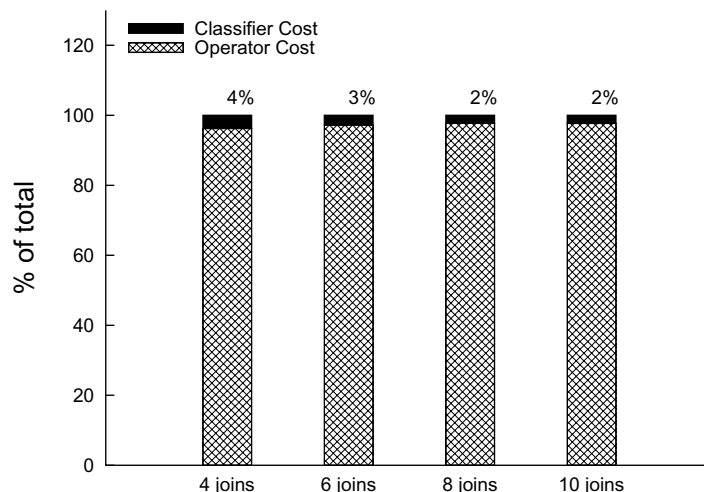


Figure 5.17. Overhead of runtime classification.

### 5.4.3 Summary of Core QM Experimental Conclusions

The main points of our experimental study can be summarized as follows:

1. *QM* can give up to 44% improvement in execution time and output rate.
2. Even if data is not skewed, *QM*'s performance in the worst case will be at most 2-8% slower than a single plan approach.
3. The runtime overhead of *QM* is very small (2% to 4% at most) relative to the overall query processing cost.
4. The actual route execution in *SRF* (forwarding of data by an operator to the next operator in the route) is nearly negligible resulting in 0.01% of total query execution cost.

## 5.5 QM Conclusion

Here, we have proposed a novel query processing approach called *Query Mesh* (or *QM*). *QM* approach answers a central need to have a middle-ground solution between the plan-based systems using a single plan and the continuously re-optimizing solutions that may employ different plans for different data. Furthermore, *QM* offers numerous advantages over the state-of-the-art techniques. First, *QM* uses machine

learning techniques to discover the relationship between the data and the resulting routes to find the best processing strategy for different subsets of data. Second, *QM* is comprehensive and addresses both query optimization and execution. Third, *QM* execution infrastructure facilitates shared operator processing and has near-zero route execution overhead. Our most important contribution was to show that *QM* implemented in a prototype DSMS can achieve significant performance improvements over alternative solutions, and thus presents a potential as a paradigm for query optimization.

In the future we plan to address the issue of uncertainty in *QM*. The current setup assumes a “perfect” knowledge scenario when computing *QM*, which may not be the fact in real-life. We plan to address scenarios, where computed routes may be uncertain (e.g., due to lack of statistics), classifier may be uncertain (e.g., due to several possible best route alternatives for a subset of data), and how the *QM* optimizer the *QM* executor should handle such cases.

## 5.6 Self-Tuning Query Mesh (ST-QM)

As we have described at the beginning of this chapter, in real-life applications, different subsets of data may have distinct statistical properties, e.g., various websites may have diverse visitation rates, different categories of stocks may have dissimilar price fluctuation patterns. For such applications, it can be fruitful to eliminate the commonly made *single execution plan* assumption and instead execute a query using several plans, each optimally serving a subset of data with particular statistical properties. Furthermore, in dynamic environments, data properties may change continuously, thus calling for adaptivity. The intriguing question is: can we have an execution strategy that

1. is plan-based to leverage on all the benefits of traditional plan-based systems,
2. supports multiple plans each customized for different subset of data, and yet

### 3. is as adaptive as “plan-less” systems like Eddies?

While the proposed *Query Mesh (QM)* approach provides a foundation for such an execution paradigm, it does not address the question of adaptivity required for highly dynamic environments. In this section, we fill this gap by proposing a *Self-Tuning Query Mesh (ST-QM)* – an adaptive solution for content-based multi-plan execution engines. *ST-QM* addresses adaptive query processing by abstracting it as a *concept drift problem* – a well-known subject in machine learning. Such abstraction allows to discard adaptivity candidates (i.e., the cases indicating a change in the environment) early in the process if they are insignificant or not “worthwhile” to adapt to, and thus minimize the adaptivity overhead. A unique feature of our approach is that all logical transformations to the execution strategy get translated into a *single inexpensive physical operation* – the classifier change. Our experimental evaluation using a continuous query engine shows the performance benefits of *ST-QM* approach over the alternatives, namely the non-adaptive and the Eddies-based solutions.

#### 5.6.1 Motivation for Adaptivity

Many modern applications deal with data that is updated continuously and needs to be processed in real-time [214–216]. Examples include network monitoring, financial monitoring, fraud detection, etc. Even if given a highly effective query execution strategy at the start, data and system characteristics may change considerably during the query lifetime, making it necessary to adapt the execution strategy. This pressing problem of adaptivity has become an important and active area of research in recent years [58, 72, 76, 96, 217]. Moreover, real-life datasets typically tend to be non-uniformly distributed [218], e.g., sensor networks, moving objects, etc. Enforcing a single query plan execution strategy, as is the defacto standard for most database technology, may lead to serious performance deterioration in situations where subsets of data may have very different statistics [20].

*Motivating Example.* Consider the following continuous query used in a financial monitoring application to correlate stock prices with current events:

```
SELECT *
FROM stocks, news, currency, blogs
WHERE blogs.subject = stocks.industry
      AND stocks.region = news.region
      AND news.country = currency.country
      AND stocks.percent_change > 15
```

To answer this query, the data may be acquired from several stock exchanges, geographically dispersed news sources and blogs that may be updated at various rates, e.g., based on the location or the time zone. Arriving from various data providers, the respective data subsets are likely to have different statistical properties, such as their data values, their frequency, and arrival rates. To complicate matters, in reaction to the same real-life events, prices of stocks may fluctuate rather differently over time. News about political instability in certain geographical regions may affect positively the stocks of defense-related companies while having an opposite or no effect on other sectors. Change in data values and their frequencies may lead to the disappearance of existing and the emergence of new statistically similar data subsets, consequently leading to changes in query execution statistics. To ensure good performance at all times, a database system must be capable to continuously identify such distinct data subsets and to adapt the execution strategy accordingly.

Unlike most adaptive solutions, e.g., [58,72,76], our work does not focus on adapting a *single execution plan* for a query, but rather on adapting the *multi-plan-based* (or we refer to it as *multi-route-based*) *execution strategy*<sup>12</sup> [219].

## 5.6.2 Adaptive Multi-Plan Query Processing

Given that *QM* employs a practical middle-ground strategy between the two query optimization extremes – the solutions that employ a “monolithic” single execution

---

<sup>12</sup>We use terms “plan” and “route” interchangeably. Both mean the same thing in the context of this paper. To prevent any confusion with Eddies-based systems [33,34], a route in our work is a fully pre-computed query plan.

plan strategy for all input data, e.g., nearly all commercial DBMSs [30–32], and the systems like Eddies that employ a fine-grained “plan-less” approach, where instead of predetermined plans, at runtime the Eddy operator determines, one-at-a-time, the next operator, that the tuples must visit for processing [34]. The open question now arises, if a multi-plan based execution strategy, such as *QM*, can be as adaptive as “plan-less” systems like Eddies? The need for adaptivity is evident. Even with an initial good choice of a *QM* solution, after some time, data characteristics, e.g., data values, their frequencies and execution statistics, such as operators’ costs and selectivities, may change considerably requiring to adapt the execution strategy. The fundamental challenge for *QM* adaptivity is the problem of determining the discrepancy between the previously constructed *QM* model<sup>13</sup> and the currently most suitable *QM* solution based on the new data characteristics, i.e., its values and its statistics. In machine learning, such discrepancy is called a *concept drift* [141]. Concept drifts happen when a model built in the past is no longer applicable to the current data.

In the context of *QM*, the change may occur at either the *target concept* level, i.e., the routes in the multi-route configuration, or at the underlying *data distribution* level, i.e., the data values and their frequencies (see Figure 5.18). The necessity to change the current model due to changes in the data distribution is called a *virtual concept drift* [220]. A *real concept drift* may occur for instance when more accurate statistics become available during execution and the routes in *QM* should be adapted based on this new information. Virtual and real concept drifts often occur together. We refer to such case as *hybrid concept drift* [141]. From a practical point, a concept drift (real, virtual, or both) gives a good indication that the current *QM* solution needs to be adapted. A concept drift implicitly indicates that either the data values, their frequencies or execution statistics have changed. Thus the predictions made by the current *QM* solution become less accurate as the time passes, e.g., data may be assigned to “wrong” subsets and less efficient execution plans may be used for

---

<sup>13</sup>A *QM* solution represents a particular “model” of execution, as determined by the classifier and the set of execution routes. In machine learning, this term is commonly used to refer to classifier-based systems.

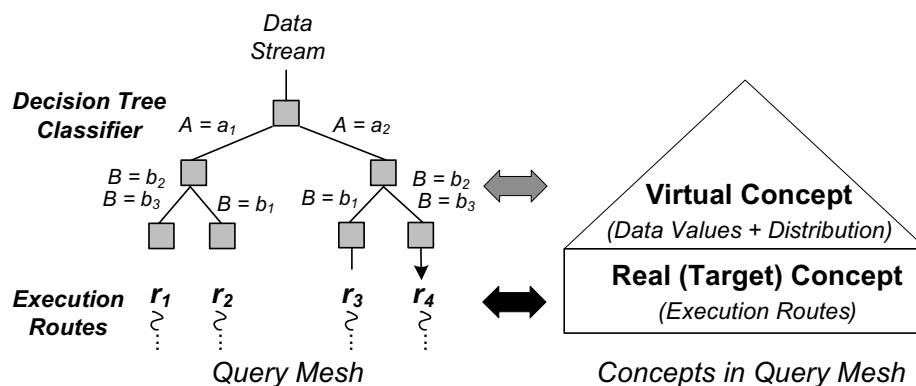


Figure 5.18. Virtual and real concepts in Query Mesh.

processing of those data tuples. Hence, detecting concept drifts can serve as a good signal indicating a possible need to adapt.

Multi-route adaptivity is a more complex problem compared to a single plan adaptivity and brings several new challenges. First, we must continuously find and deploy the best execution solution where multiple plans are used concurrently. The majority of current adaptive solutions [76] are inapplicable here, as these methods are designed to support only a single plan. Second,  $QM$  employs a classifier as a component of query processing infrastructure. Therefore the classification cost must be taken into account by the  $QM$  optimizer. Furthermore, a  $QM$  may need to be adapted not only when statistics change, but also when data values change (even if statistics stay the same), because such change has a direct impact on the classifier accuracy<sup>14</sup> and the overall performance of  $QM$  solution. Therefore, monitoring data values is as important as monitoring statistics. Finally, the physical execution of  $QM$  adaptation itself must be inexpensive to make it practical for dynamic environments where query results must be produced in near-real time. In summary, the key challenges include: (1) how and when to determine that the current  $QM$  solution is no longer adequate, (2) how to determine the new “best”  $QM$  solution based on the new data values and

<sup>14</sup>The classifier is constructed based on data values.

the updated statistics, and (3) how to efficiently execute the physical migration from the current  $QM$  to a new  $QM$  solution while the query is being executed.

### 5.6.3 Our Proposed Solution: ST-QM

We address the above-mentioned challenges by proposing a self-tuning framework for  $QM$  called *ST-QM*. The techniques presented in this work are discussed in the context of stream environments and multi-plan query processing, however, in principle, they can be applicable to other systems as well. In summary, the contributions of this paper are:

1. We abstract the adaptivity of a multi-plan solution  $QM$  as a *concept drift* problem. Our approach, based on monitoring and detection of concept drifts, can discard many insignificant adaptivity cases early, and thus minimize the adaptivity overhead.
2. We present algorithms to efficiently determine *virtual* and *real* concept drifts in  $QM$  used to determine if and how the execution strategy should be adapted.
3. The key feature of our adaptive method is that all *logical* transformations to the current execution solution are translated into a *single physical operation* – the change of the classifier, without effecting the rest of the execution infrastructure. This makes physical adaptivity extremely lightweight.
4. We thoroughly evaluate the *ST-QM* approach through experiments. Our results show that *ST-QM* is very effective in adapting to different kinds of concept drifts, its overhead is minimal, and the physical actuation of adaptivity has nearly negligible cost.

## 5.7 Overview of Self-Tuning Query Mesh

### 5.7.1 The Main Idea

The following is the problem we tackle in the context of adaptive query processing using Query Mesh model:

**Multi-Route AQP Problem:** For a given query  $Q$  and its multi-plan solution  $QM$  computed at time  $t_i$  based on the representative dataset  $T$  and statistics  $H$ , continuously detect a concept drift when a new sample dataset  $T'$  and statistics  $H'$  become available at time  $t_j > t_i$ . If a concept drift has occurred, find a new solution  $QM'$  based on  $H'$  that results in the lowest execution cost for tuples in  $T'$ . If the estimated  $cost(QM') < cost(QM)$ , replace  $QM$  with  $QM'$ .

The goal of Self-Tuning Query Mesh framework (or short  $ST-QM$ ) is to detect  $QM$  concept drifts and to adapt the current  $QM$  solution correspondingly to best suit the observed drift. Our approach is unique in that we view the problem of adaptive query processing (AQP) as a *concept drift* problem from machine learning [221]. This abstraction of the AQP gives several advantages to the adaptive system. First, if we discard an adaptivity case due to an absence or a presence of a small concept drift, it is likely not going to lead to a better  $QM$  solution, because we've discarded insignificant changes in the environment. If we do not discard a case, then there is a high chance that it is worthwhile to analyze further. In the end, there are fewer cases  $ST-QM$  has to analyze and the ones that do get analyzed further are all promising. Second, techniques from machine learning and data mining fields addressing the concept drift detection and analysis can be leveraged here to determine if adaptation is needed and how to best adapt to the observed drift.

### 5.7.2 Query Mesh Concept Drifts

Given the two kinds of concepts in  $QM$  (*virtual* and *real*), the following three cases may occur:



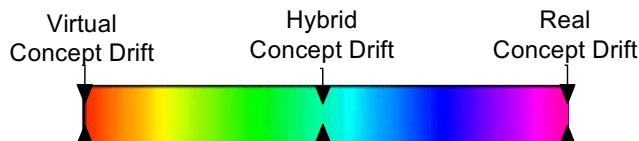


Figure 5.19. Concept drift “spectrum”.

**Case 1:** *Virtual Concept Drift.* This indicates that data values and/or their frequencies have changed, but the execution statistics of the new data subsets stay the same, thus making the previously computed routes still applicable. One example when such scenario may occur is when a better quality (i.e., more representative) training dataset is collected over time. In this case, the execution statistics of the subsets might not change significantly, yet the *QM* classifier can be further fine-tuned by integrating new data values. For example, a new *DT* sub-tree can be added or the nodes can be “pushed-up” or “down” for faster classification. Another example of this case (based on the application mentioned in Section 5.6.1) is when a stock exchange opens and starts streaming its data. The streaming data values from the recently opened stock exchange get combined with the streaming data from other previously streaming stock exchanges (e.g., from other regions). Here the new stock data values, e.g., symbols, location, etc., will appear in the data streams, yet the underlying distribution and the statistically similar data subsets are likely to stay unchanged.

**Case 2:** *Real Concept Drift.* This case means that the data values stay unchanged, but their execution statistics (e.g., selectivities or operator costs) begin to vary, thus requiring the execution routes to be adapted. This scenario tends to be less frequent, but may arise when the optimizer used a rough approximation of data subsets’ statistics, and then more accurate statistics become available as a result of the query execution feedback. The updated statistics enable the “tune-up” of the execution routes. Using the financial application example, this case may happen when a sole stock market is being monitored. Here, the data values, e.g., stocks being sold on

this stock exchange, become available as soon as it opens and are unlikely to change significantly during the day. Yet, the statistics for the new data might not be very accurate at the beginning of the execution. However, the longer the query runs, the more accurate estimations can be made. Another example when this case may occur is when better routes are found through *route exploration* described in Section 5.8.2.

**Case 3: Both Virtual and Real Concept Drifts.** We refer to this case – the hybrid concept drift, and it happens when both the data distribution and the execution statistics change, consequently leading to alterations in the execution routes and the classifier. Using financial application example, this case may happen when during the after-market trade hours, important news become public, which may have a significant impact on the stock prices of some industries. Since not all stocks participate in the after-hours trading, the data distribution changes after the markets close. Furthermore, the real-life news may impact the prices of only certain types of stocks. In this case, both the data distribution and the execution statistics may change significantly, thus requiring both the classifier and the set of execution routes in *QM* to be adapted.

The three cases described above are not independent. Virtual and real concept drifts are the special cases of the hybrid concept drift. The three cases compose a comprehensive “spectrum” of changes that may occur in a system (Fig. 5.19): specifically, a change in data values and their frequencies, a change in execution statistics and a change in both.

### 5.7.3 ST-QM Architecture

*ST-QM* adds three new components to the core *QM* framework: *ST-QM Monitor*, *ST-QM Analyzer* and *ST-QM Actuator* (shaded grey in Figure 5.20). We have designed *ST-QM* to be highly modular, enabling adaptivity functionality to be turned on/off with complete transparency to the core *QM* framework (bottom of Figure 5.20). The architecture is easily extensible: new algorithms and metrics can be added

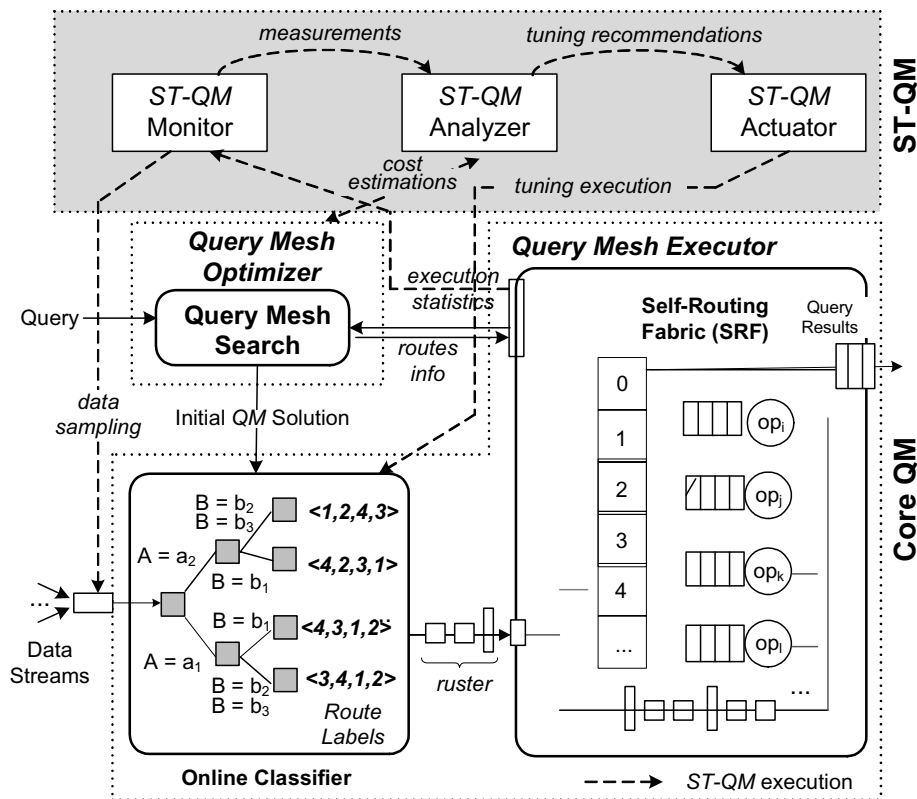


Figure 5.20. Self-tuning Query Mesh framework.

without much disturbance to the rest of the system. We describe the functionality of each *ST-QM* component next.

*ST-QM Monitor* continuously samples data and execution statistics that will be used to determine if a concept drift has occurred. Monitored parameters include data values, their frequencies, and the operators' costs and selectivities. Our monitoring approach is comparable to that of the existing systems, e.g., [58, 222] with a few distinct characteristics (see Section 5.8). Given the measurements from the *ST-QM Monitor*, the *ST-QM Analyzer* determines if a concept drift has actually occurred, how well the current *QM* solution is meeting its estimated costs and performance goals, and what (if anything) is going wrong. Based on the analysis, the *ST-QM Analyzer* makes recommendations if and how the *QM* solution should be adapted.

*ST-QM Actuator* takes these recommendations and physically adapts the *QM* solution. Figure 5.21 graphically depicts the flow of the entire process.

Monitoring, analysis, and actuation of adaptivity in *ST-QM* add overhead to query processing. Thus, to minimize the overhead, the following system requirements must be met: (1) monitoring must be light-weight, and only if significant changes are detected should the more expensive analysis process be invoked, (2) adaptivity candidates corresponding to insignificant changes in the environment must be discarded early, e.g., during monitoring or in the early analysis, without invoking the optimizer, (3) the decision to adapt should be made only if significant improvement in the performance is expected, and (4) the physical execution of adaptivity must be fast and inexpensive to be done online.

## 5.8 ST-QM Monitor

Monitoring aims to identify if the current *QM* solution is no longer consistent with the current data and its characteristics. What sets apart our monitoring goals from the existing systems, e.g., [58, 222] is that: (1) we monitor not only the change in data distributions and execution statistics but also in the data values, and (2) we focus not only on assuring the overall representativeness of a sample but also on ensuring that new, i.e., the never seen before data values are not gone undetected. *ST-QM Monitor* employs two complimentary techniques, namely the input data and the execution statistics monitoring.

### 5.8.1 Input Data Monitoring

For data monitoring, we sample the arriving to the server data to collect a *new training dataset*. This new dataset is analyzed to see if changes in the data values and their distributions have occurred. Monitoring data values (in addition to the distributions and execution statistics) has the advantage that the adaptive system can exploit this extra information, which is collected inexpensively, to minimize the

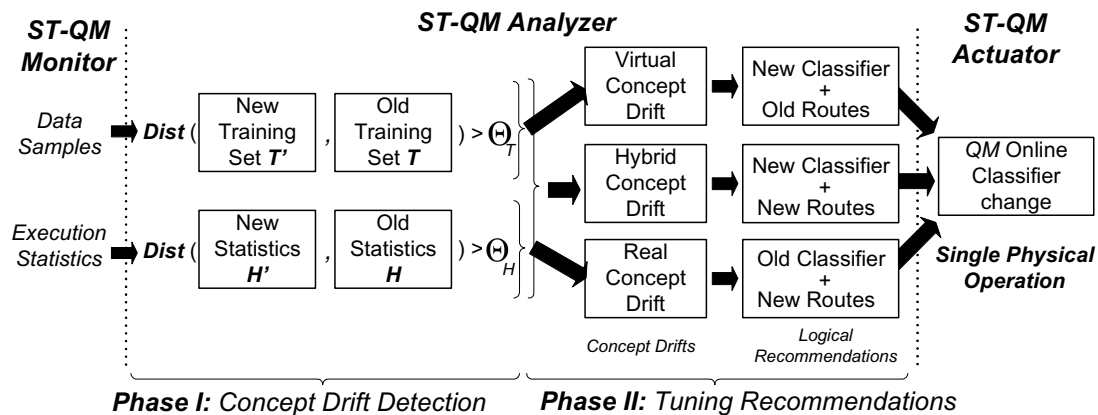


Figure 5.21. ST-QM process flow.

overhead of the more expensive execution statistics monitoring, e.g., when profiling operators or exploring for new execution routes alternatives. Often changes in data values implicitly indicate changes in data distributions. Consequently, this leads to a possible change in the execution statistics, since virtual and real concept drifts frequently occur together. If the system detects a change in data values, it may then employ a more expensive and detailed execution statistics monitoring to see if the routes may need to be adapted. Simple random and systematic sampling techniques can be used here for data sampling [200]. However they can miss potentially “important” training data trying to uniformly cover the entire sampling window. Thus, we’ve designed the following techniques:

*Classifier-driven sampling.* This type of sampling is based on the “importance” of tuple attributes. In *QM*, some attributes are naturally more important than others, e.g., when the decision tree (*DT*) classifier is constructed, a split criterion is used to select the best splitting attribute at each node. *Information gain*, *entropy*, or *gini index* measures of impurity can be used for this purpose [141,223]. Comparing the impurity value of a split attribute in the *DT* classifier for the old and the new samples of data can be a good indicator if the data distribution possibly changed. If the differences between the old and the new impurity measures for the same attributes

are significantly different, then the new sample is considered “interesting to analyze further” and thus is not discarded. The decision of how many impurity measures to compute, i.e., for how many *DT* nodes, and their relative importance in the overall sampling is parameterized.

*Route-driven sampling.* This sampling method resembles a *biased* sampling approach. It is guided by the *QM* execution routes and the expected percentage (*% expected*) of tuples to be processed by those routes. Here, each tuple from the new sample first probes the current *QM* classifier (Stage 1). After probing, tuple groups are formed, with each group being assigned to a particular route. If the difference between the actual and the expected route assignment fraction of tuples is less than the system-set threshold, then a random selection of *k* members from those groups is performed. If the difference is greater than the threshold, these tuple groups get a high “priority”, because they contain the different (from before) data and  $(k + k * (\% \text{ expected} - \% \text{ actual}))$  tuples are sampled from each such tuple group (Stage 2). The sub-sample size here is directly proportional to the observed frequency difference.

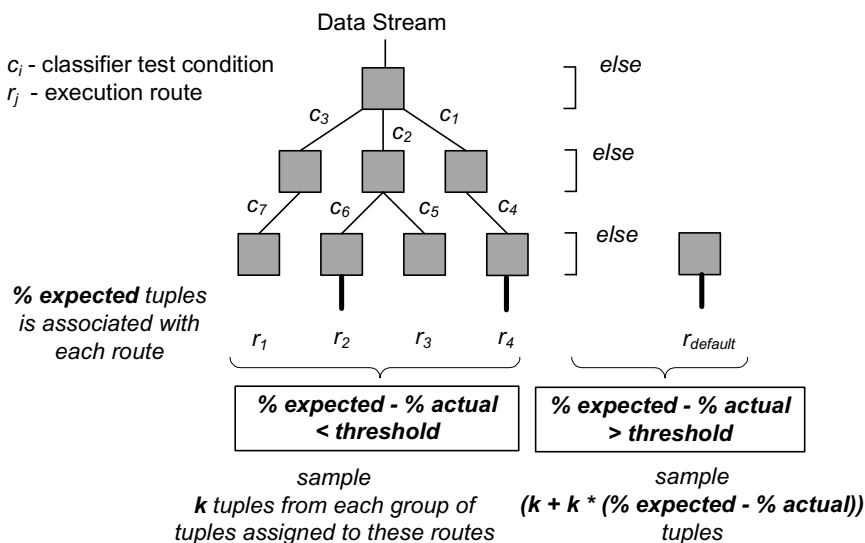


Figure 5.22. Route-driven sampling.

The motivation behind this method is the following: if the same fraction of tuples were assigned to the same routes, then the data distribution is unchanged. However, if the difference is significant, e.g., in the case of a special route called the *default route*<sup>15</sup>, then more tuples should be sampled, as this could be an indication of a virtual concept drift and possibly the classifier may need to be updated. Since this type of sampling is “biased” towards collecting previously unseen data, the new sample is treated as a compliment to the old training data set and the two sets are combined (unioned) to improve the overall quality of the training data and the resulting execution model.

### 5.8.2 Execution Statistics Monitoring

The statistics collected during execution are used to detect the presence of the *real* concept drift. Execution statistics monitoring consists of two complementary sub-parts: exploitation and exploration statistics monitoring.

*Exploitation Statistics.* Exploitation statistics monitoring tracks the selectivities and costs of operators when using the established execution routes. We instrument query operators to collect three types of statistics: (1) *independent* selectivities, (2) *correlated* selectivities and (3) operator costs (measured by wall-clock time). To compute independent selectivities, a *statistics bit* is turned on in the *r-token* of a randomly selected *ruster*, thus making it a special-purpose (*statistics*) *ruster*. If a tuple from a *statistics ruster* does not satisfy operator predicate, the tuple is not physically discarded, but rather marked as a “ghost” to be able to compute independent selectivities for other operators en-route. For correlated selectivities, the selectivity is computed using only the regular (“non-ghost”) tuples in the *statistics ruster*, i.e., the tuples that have not been discarded by any previous operators in the route. All three types of statistics are collected for each individual route by the operators.

---

<sup>15</sup>A *default route*  $r_{default}$  (illustrated in Figure 5.22) is an execution route based on the overall statistics of the data. It is used by the data that has similar statistics as the overall data statistics, as well as by the “new” data with properties (values and frequencies) that may have not been present when the *QM* was originally computed.

*Exploration Statistics.* The motivation for the exploration statistics lies in the fact that the only way to know precise costs of alternative strategies is through competitive execution [34]. For this purpose, we use *exploration rusters* – a small fraction of the input *rusters* that are randomly selected and assigned different from their current “best” routes, while monitoring the statistics along these routes. The exploration routes are determined by the *exploration policy*. *ST-QM* employs two exploration policies: (i) *random existing route*, where chosen *rusters* are sent on another randomly picked *existing* route; (ii) *random new route*, where *rusters* are sent on a randomly generated and currently *non-existing* in the *QM* solution route.

Devoting resources to exploration to obtain information about thus-unknown costs may help in finding better routes, but in the short term it detracts from exploitation – producing results with the current best routes. This is a classic exploration versus exploitation dilemma [76]. To address this problem, *ST-QM* adaptively determines the number of *rusters* used for exploration. The total number of exploration *rusters* (*TER*) depends on the value of a distance measure (described in Section 5.9) and is computed as:  $TER = DER + (\alpha * \mu)$ , where *DER* is the default number of exploration *rusters*,  $\mu$  is the value of the distance measure and  $\alpha$  is the fraction of *rusters* per distance unit. The larger the distance, the larger the number of *rusters* used for exploration. Exploration may also be applied selectively to only some *rusters*, to put more focus on exploring routes for certain subsets of data.

## 5.9 ST-QM Analyzer

The *ST-QM Analyzer* takes the data samples and the statistics from the *ST-QM Monitor* and based on them determines if any concept drifts have occurred. It then gives tuning recommendations based on the analysis. The execution consists of two phases: (1) *concept drift detection*, and (2) *tuning recommendations*.



## 5.9.1 Phase I: Concept Drift Detection

## Virtual Concept Drift Detection

The concept drift detection algorithm *CD-Detect* (in Figure 5.23) maps the problem of virtual concept drift detection to the problem of comparing two data samples  $T$  and  $T'$ .

```

CD-Detect ( $T$  old training set,  $T'$  new tuple sample,
 $H$  old statistics,  $H'$  new statistics)
01  $dist_{data} = ComputeDataDistance(T, T')$ 
02  $dist_{routes} = ComputeRoutesDistance(H, H')$ 
03 if ( $dist_{data} > \theta_{data}$ ) and ( $dist_{routes} > \theta_{routes}$ )
04   return Hybrid Concept Drift
05 else if ( $dist_{data} > \theta_{data}$ )
06   return Virtual Concept Drift
07 else if ( $dist_{routes} > \theta_{routes}$ )
08   return Real Concept Drift

```

Figure 5.23. QM concept drift detection.

The algorithm requires a distance measure  $dist_{data}$  which quantifies the difference between data samples  $T$  and  $T'$ . If  $dist_{data} > \theta_{data}$ , where  $\theta_{data}$  is the adjustable distance threshold, virtual concept drift is reported. The key to the change detection is the intelligent choice of the distance function to compute  $dist_{data}$ , which must accurately quantify a data change that may impact the current *QM*. The choice for the threshold  $\theta_{data}$  value defines the balance between the sensitivity and the robustness of the detection. The smaller  $\theta_{data}$ , the more likely we are to detect small changes in the data, but the larger is the risk of a false positive.

One common approach to measuring data differences is to first estimate the probability distributions of the data, and then compute the distance, such as the *Kullback-Leibler Divergence* or the *Jensen-Shannon Divergence* [224], between the estimated distributions. However, this approach is computationally impractical for large and

high dimensional data. The problem becomes even more challenging in streaming data environments, as the high speed makes it difficult for such expensive algorithms to keep up with the data [225]. To tackle this issue, we have designed two efficient methods:

*Misclassification Rate.* Misclassification rate or error rate  $\mathcal{E}$ , described as  $\mathcal{E} = (1 - \mathcal{A})$  where  $\mathcal{A}$  is the classifier accuracy, represents the fraction of total cases “misclassified” by the current *QM* classifier for the new data sample. The main idea here is to assign the execution routes to the tuples from the new data sample. Then the tuples from the new sample probe the current classifier, and the classifier’s misclassification rate, e.g., mean absolute error, is computed. The reason we assign the new sample tuples to the existing routes (even though we could possibly find better plans for their processing) is because we are checking for virtual concept drift with respect to the current target (i.e., the current set of execution routes).

*Signature-Based Method.* This method regards the decision tree classifier as a summarization of the distribution of data. Each leaf node contains a route label and the fraction of tuples expected to be processed by that route. Together, all the leaf nodes can be thought of forming a special “histogram” of route assignment frequencies. Then after probing the classifier, a signature is assigned to each data sample that depicts the route assignments frequencies. This way we evaluate data distribution changes by comparing these signatures. This method is extremely efficient, since all it requires is a quick probe of the classifier.

### Real Concept Drift Detection

Real concept drift occurs when execution statistics change significantly, consequently implying that the execution routes may need to be altered as well. Given the updated execution statistics, a new set of routes is computed and compared to the old set of routes. The goal here is *not* to estimate whether the *QM* solution with the new set of routes would necessarily be “better” (remember, the cost of a *QM* solution depends

on the combination of both the classifier and the routes' costs), but rather that the new routes are different (see Algorithm in Figure 5.23). Using such simple route difference approach allows *ST-QM* to minimize its overhead: since route computation is a fraction of the entire *QM* re-computation [219]. Next we discuss several possible choices for the route distance measure  $dist_{routes}$ .

*Number of Affected Routes.* This distance measure counts the number of routes that are different when comparing the old and the new sets of routes. Let  $R$  denote the old set of routes, and  $R'$  be the new set of routes. Then  $dist_{routes} = |R_{diff}| = |R' - R|$ , where  $\forall r \in R_{diff}, r \in R'$  and  $r \notin R$ . For example, if a route  $r$  has a different operator ordering or if a new route  $r$  exists in the new set as a result of exploration – all these changes contribute to the route distance measure. If a more fine-grained measure is needed, the approach can be extended to consider the count of the operators with significantly different selectivities and execution costs.

*Route Edit Distance.* This distance measure is based on the *edit distance* approach [226]. Here, the old and the new routes are mapped respectfully to the same data subsets, meaning these routes were considered as the best execution strategies for processing of the same data subset at different times. Routes represent operator sequences and can be described by the strings composed of operator identifiers. The edit distance between any two routes is then the number of operations required to transform one of them into the other. The examples of edit distances that can be used here include *Hamming distance*, *Levenshtein distance*, and many others.

## 5.9.2 Phase II: Tuning Recommendations

After *QM* concept drifts have been detected, the *ST-QM Analyzer* determines how to address them. In response to the concept drifts, *ST-QM Analyzer* may do the following: (1) ignore the concept drifts, if they are small or the benefits of adapting the current *QM* is not expected to give much performance improvement; (2) incrementally tune a sub-part of the *QM* solution, e.g., a classifier sub-tree or a route; (3) compute a

---

Algorithm **TR-*Produce***(*CD detected concept drift, T' new training dataset, H' latest execution statistics*)

```

1:  $QM$  = current query mesh solution used in execution
   /* VIRTUAL CONCEPT DRIFT RECOMMENDATIONS */
2: if ( $CD.Type == Virtual\ Concept\ Drift$ ) then
3:   Compute new classifier  $C'$  based on the training set  $T'$ 
4:   Let  $QM'$  = new query mesh solution with classifier  $C'$ 
5:   if ( $cost(QM') < cost(QM)$ ) then
6:     Recommend New Classifier  $C'$ 
7:   end if
   /* REAL CONCEPT DRIFT RECOMMENDATIONS */
8: else if ( $CD.Type == Real\ Concept\ Drift$ ) then
9:   Compute new set of routes  $R'$ 
10:  Let  $C' = current\ classifier\ QM.C$ 
11:  Update the target level of the classifier  $C'$  with routes  $R'$ 
12:  if (the target  $R'$  requires modification of classifier  $C'$ ) then
13:    Compute a new classifier  $C''$  based on the new target  $R'$ 
14:    Let  $QM'$  = new query mesh solution with classifier  $C''$ 
15:    if ( $cost(QM') < cost(QM)$ ) then
16:      Recommend New Classifier  $C''$  and New Routes  $R'$ 
17:    else
18:      Let  $QM'$  = new query mesh with routes  $R'$ 
19:    end if
20:    if ( $cost(QM') < cost(QM)$ ) then
21:      Recommend New Routes  $R'$ 
22:    end if
23:  end if
   /* HYBRID CONCEPT DRIFT RECOMMENDATIONS */
24: else if ( $CD.Type == Hybrid\ Concept\ Drift$ ) then
25:   Compute new  $QM'$  solution based on the training set  $T'$  and the new statistics  $H'$ 
26:   if ( $cost(QM') < cost(QM)$ ) then
27:     Let  $C' = classifier\ QM'.C$ 
28:     Let  $R' = set\ of\ routes\ QM'.R$ 
29:     Recommend New Classifier  $C'$  and New Routes  $R'$ 
30:   end if
31: end if

```

Figure 5.24. QM tuning recommendations.

---

new  $QM$  solution based on the updated statistics and consider to replace the current  $QM$  solution.

## Recommendation Algorithm

Figure 5.24 illustrates the pseudo-code for the *TR-Produce* algorithm<sup>16</sup> employed by the *ST-QM Analyzer* to produce tuning recommendations. Similar to many adaptive solutions, *ST-QM* uses the *QM optimizer cost model* [219] to compare the current execution to what was originally expected or what is estimated to be possible [76]. The recommendation algorithm has the following cases:

**Case 1:** *Virtual Concept Drift Recommendation.* If a virtual concept drift is detected, first a new classifier  $C'$  for the new training set  $T'$  is computed. Then the cost of the new query mesh (with the new classifier  $C'$ )  $QM'$  is determined and compared to the cost of the current  $QM$ . If the new  $QM'$  has a smaller cost, the new classifier  $C'$  is recommended.

**Case 2:** *Real Concept Drift Recommendation.* If a real concept drift has been detected, the target level (i.e., the routes) in the  $QM$  classifier are updated. If this update does not require the modification of the rest of the classifier, and if the  $QM'$  solution with new routes  $R'$  has a smaller cost than the current  $QM$  solution, then the new routes  $R'$  are recommended. If the classifier needs to be adjusted (e.g., if some routes are now shared by several groups or if some routes are removed), this case is then handled as a hybrid concept drift.

**Case 3:** *Hybrid Concept Drift Recommendation.* If a hybrid concept drift has been detected, a new  $QM$  solution with the new classifier and the new set of routes is computed, its cost is estimated and compared to the current  $QM$  solution's cost. If the newly computed  $QM'$  has a smaller cost, then both the new classifier  $C'$  and the new routes  $R'$  are recommended<sup>17</sup>.

To evaluate the benefit of a recommendation, *ST-QM Analyzer* uses the metric, called improvement  $I$ :

$$I(QM, QM', T', H') = 100\% * \left( 1 - \frac{\text{cost}(QM', T', H')}{\text{cost}(QM, T', H')} \right)$$

<sup>16</sup>“*TR*” is the abbreviation for “Tuning Recommendations”.

<sup>17</sup>If either the classifier or the new set of routes have been computed in the earlier stages of analysis, e.g., during concept drift detection, they are cached and not recomputed in this phase.

where  $QM$  is the initial and  $QM'$  the recommended solution, and  $cost(QM', T', H')$  is the expected cost of evaluating a query under the  $QM'$  solution based on the training data set  $T'$  and the statistics  $H'$ . The *ST-QM Analyzer* computes the expected improvement value, and if the value is deemed as substantial, only then the recommendation is outputted.

## 5.10 ST-QM Actuator

### 5.10.1 Physical Execution of Adaptivity

*ST-QM Actuator* physically adapts the  $QM$  solution in the execution framework based on the recommendations received from the *ST-QM Analyzer*. As described in Section 5.9.2, *ST-QM Actuator* may receive the following three kinds of recommendations:

- **R<sub>1</sub>**. *New Classifier + Old Routes*
- **R<sub>2</sub>**. *Old Classifier + New Routes*
- **R<sub>3</sub>**. *New Classifier + New Routes*

The key characteristic of the *ST-QM* is that all three recommendations get translated into a *single physical operation* in the execution infrastructure, namely the change of the classifier in the online classifier operator. To accomplish this, only a simple pointer re-assignment to the new classifier object is needed (Figure 5.25). This single step is the actual execution of  $QM$  adaptivity and the implementation is trivial. What makes this possible is the architecture of  $QM$  framework. Although routes (i.e., query plans) are pre-computed, their topology is not physically constructed. Instead the *Self-Routing Fabric (SRF)* infrastructure provides distributed routing (i.e., forwarding of tuples to the operators in the plan) based on the plan specifications assigned by the classifier. This physical separation between the component that determines which plans should be used for execution and the component that actually executes the plans based on specifications, makes the  $QM$  adaptivity so light-weight. To change

the execution strategy, all the system needs to do is modify the specification of the plans (in the classifier).

Figure 5.25 illustrates an example of physical execution of *QM* adaptivity. The old classifier, marked by lighter grey, is replaced by the new classifier, and the *rusters* with new routes are sent into the self-routing fabric instantaneously. The attractiveness of

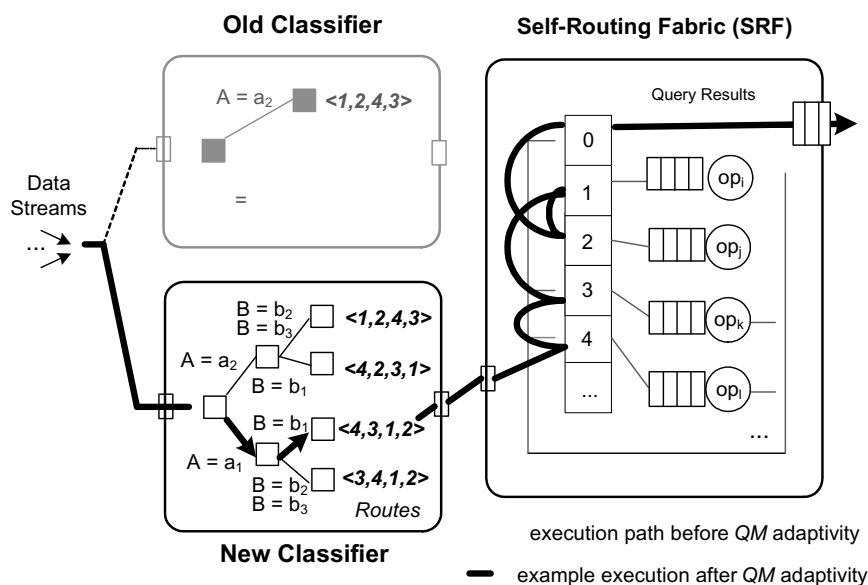


Figure 5.25. Physical execution of *QM* adaptivity.

our design is that we can easily switch between different multi-plan solutions. If the desired performance improvements after adaptivity are not gained, *ST-QM* can easily switch back to the previous *QM* solution. The architecture makes such behavior very flexible.

### 5.10.2 State Management and Adaptivity

One of the key questions that must be answered in adaptive systems is the problem of state management for stateful operators. We consider select-project-join (SPJ) queries. For joins, we employ *one-way-join-probe* (*OJP*) operators [219], similar in spirit to *SteMs* [65], which correspond to a half of a traditional join operator. There

is one *OJP* associated with each stream attribute that participates in the join. The *OJP* keeps track of the window of attribute values that have arrived on the stream and allow subsequent tuples from the other streams to probe these stored attribute values to search for a match. In the case of the join operator, the order in which tuples probe the *OJP* is irrelevant as long as each tuple passes through each *OJP* exactly once. This holds from the associativity and the commutativity property of the join operator [65, 217]. Without adaptive functionality, the core *QM* framework already supports concurrent plans with different operator ordering. Hence, adding adaptivity does not require any additional support. We plan, however, to investigate new state management techniques in our future work, to extend support to other types of queries.

### 5.11 Self-Tuning Query Mesh Experimental Study

We now describe our experimental evaluation of *ST-QM* implemented inside Java-based continuous query engine called CAPE [8]. To evaluate *ST-QM*'s design, we compare its relative performance against competitor systems, namely the non-adaptive *QM* and the adaptive “plan-less” Eddies [34] with CBR-based routing policy [28] – the closest solutions to *ST-QM*. To ensure the even comparison, all three systems were implemented in CAPE, and their implementation used as much of the same codebase and data structures as possible. We also demonstrate the effectiveness of *ST-QM* by measuring its overheads and the benefits of its concept drift abstraction approach.

#### 5.11.1 Experimental Setup

All our experiments are run on a machine with Java 1.6.0.0 runtime, Windows Vista with Intel(R) Core(TM) Duo CPU @1.86GHz processor and 2GB of RAM. Our experiments use  $N$ -way join queries which join incoming  $S_1 \dots S_N$  streams. The specific query we use is an equi-join of 10 streams, i.e.,  $S_0 \bowtie S_1 \dots S_9 \bowtie S_{10}$ .  $N$ -way join



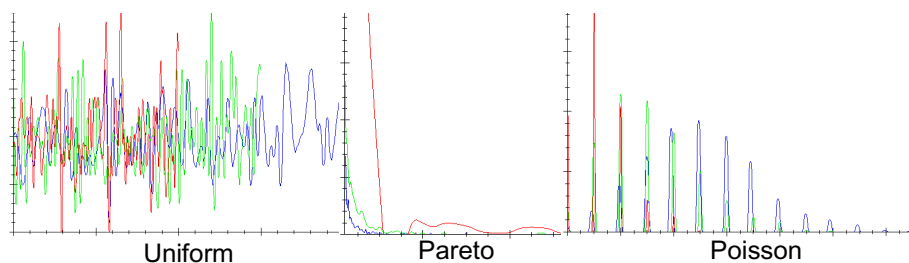


Figure 5.26. Experimental distributions.

queries are one of the core queries in database systems used to discover relationships across data or events coming from different data sources.

We use synthetic data sources for our experiments, similar to [28, 64, 65]. Using synthetic data allows us to manage data properties that are hard to control in real-life data. We employ several known data distributions to determine the skew of the data. Specifically, we use well-known distributions: *Uniform*, *Pareto* and *Poisson* [227] (see Figure 5.26). These distributions model many real-life phenomena (see Table 5.3 for examples). The default data properties, system parameters and distribution parameters used in the experiments are shown in Table 5.3 and Table 5.4<sup>18</sup>.

Each stream’s schema is composed of five attributes and a timestamp. For every join attribute column, integer-based values are generated using one of the above-mentioned distributions. The values of other attributes are correlated to the join attribute values, e.g., in a stream  $S(col_1, col_2, col_3, col_4, col_5)$ , if  $col_1$  is a join attribute, the values of  $col_2 \dots col_5$  are correlated to the values in the join attribute column according to the specified to generator correlation parameters. The default values are 50%, 30%, 15%, 5%. To make this more concrete, consider an example: value 100 is generated in the join attribute column based on the chosen distribution, then in another attribute column, 50% of the time value 99 will appear next to 100, 30% value 98, and so on<sup>19</sup>. For other attributes in the stream, the values are generated similarly.

<sup>18</sup>The different colors in Figure 5.26 illustrate how the distribution changes when the parameter values vary as described in Table 5.4.

<sup>19</sup>Values ‘99’ and ‘98’ were picked arbitrarily here to convey the example.

Table 5.3  
Default experimental parameters.

Parameter	Value	Description
<i>Ruster size</i>	100 tuples	Average <i>ruster</i> size
<i>Sample size</i>	100 tuples	Average sample size per stream
<i>Data monitoring</i>	<i>Route-driven sampling</i>	Data monitoring method. $k =  T / R $ , $\theta_{diff} = 0.2$
<i>Execution monitoring</i>	<i>Exploitation statistics</i>	No exploration is used
<i>dist<sub>data</sub></i>	<i>Signature-based</i>	Virtual concept drift detection method. $\theta_{data} = 0.1$
<i>dist<sub>routes</sub></i>	<i>Number of affected routes</i>	Real concept drift detection method. $\theta_{routes} = 0.2$
<i>Impr. I</i>	$I = 0.1$	Improvement parameter
Data Distributions		
Name	Parameters	Application Examples
<i>Uniform</i>	$\alpha \in \{\dots, \beta-1, \beta\}$ $\beta \in \{\alpha, \alpha+1, \dots\}$ $X \in \{\alpha, \dots, \beta-1, \beta\}$	<ul style="list-style-type: none"> <li>• Long-term patterns of data</li> </ul>
<i>Pareto</i>	$0 < \alpha < \infty$ $0 < \beta < \infty$ $\alpha \leq X < \infty$	<ul style="list-style-type: none"> <li>• Animal migration</li> <li>• Word frequencies</li> </ul>
<i>Poission</i>	$0 < \lambda < \infty$ $X \in \{0, 1, \dots\}$	<ul style="list-style-type: none"> <li>• Service times in a system</li> <li>• # of phone calls at a call center per minute</li> <li>• # of times a web server is accessed per minute</li> </ul>

We decided against generating random values in the non-join attribute columns, to avoid short and wide decision tree classifiers (e.g., a decision tree with height 1 and the test conditions based on all possible random values). The explanation for this is the following: if an attribute contains a lot of unique random values, the entropy value for this attribute column approaches 0. Since many splitting criteria in *DT* construction algorithms are *entropy*-based [141], the attribute with the most distinct values gets picked first, and the algorithm stops right there, thus resulting in a short and wide decision tree.

To simulate dynamic changes, the generation of data was managed as follows: the data generator starts with a data distribution and its initial distribution parameters; over time, the distribution parameters values are varied, e.g., for Poisson distribution, the transition:  $(\lambda = 1) \rightarrow (\lambda = 3) \rightarrow (\lambda = 5)$  (see Table 5.4), means that the initial distribution parameter value was 1, after some time it was changed to 3, and then to 5. This process is repeated continuously for infinite data streams. The values of distribution parameters are changed every 10K tuples across all streams.

The execution of *ST-QM* in CAPE [8] is split into two execution threads. The monitoring and the adaptivity actuation are interleaved with the query execution on one thread. The analysis of *ST-QM* (i.e., concept drift detection, optimizer calls and generation of tuning recommendations) is executed on another thread. The analysis and the optimizer search can sometimes be extensive [219], thus blocking the query executor from processing the arriving data tuples, while the system is being analyzed by adaptive component, is not practical. Hence, we separated *ST-QM* analysis into a separate thread, to prevent blocking of the query executor and to ensure that results are produced at all times.

### 5.11.2 Results and Analysis

#### Comparison Against Alternative Systems

In this experiment, we compare *ST-QM* design against the closest competitors, specifically the non-adaptive *QM* execution and the Eddy-based system with CBR-based routing [28]. The main difference between the implementations is that the non-adaptive *QM* evaluates the query using the same classifier and routes for the duration of the entire query execution. If data characteristics change, and the classifier does not have a sub-tree for the new data values, the “*default plan*” ( $r_{default}$ ) is used for processing of that data.  $r_{default}$  plan is based on the overall statistics of the data and is computed by the optimizer prior to query execution, just like in traditional query optimization. CBR-based execution is done in the context of Eddies. Eddy operator

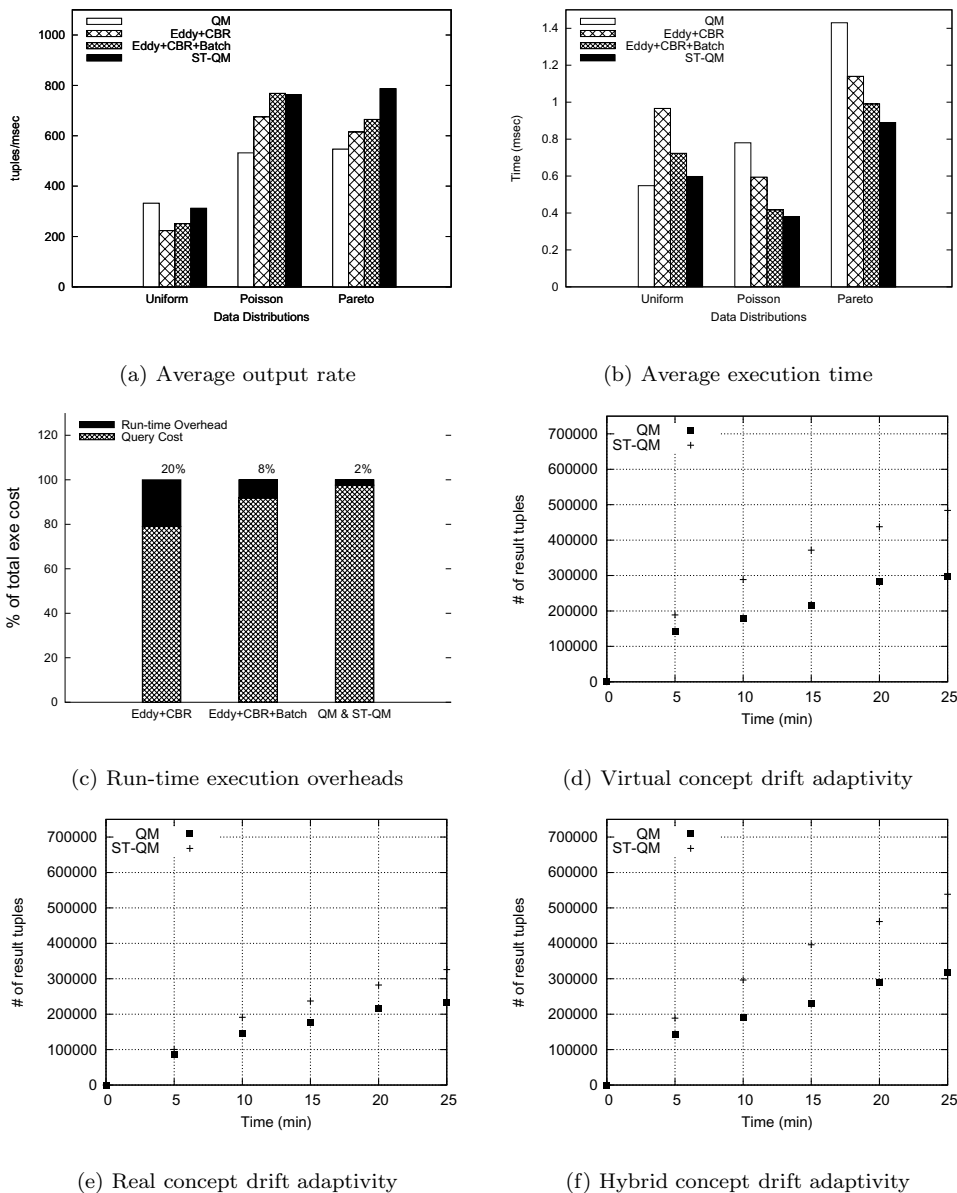


Figure 5.27. Experimental results.

continuously profiles operators and identifies “classifier attributes” to partition the data into tuple classes that may be routed differently [28]. We execute Eddy with CBR routing in two modes: (i) with batching and (ii) without batching [33]. The batch size is set to 100, which is similar to *ruster* max size parameter in *ST-QM* (see Table 5.3), and is designed to reduce execution overhead.

Table 5.4  
Distribution statistics and parameters.

<p><b>Uniform</b> (<math>\alpha = 0, \beta = 100</math>): <i>min</i>: 0.0, <i>max</i>: 100.0, <i>med</i>: 49.0, <i>mean</i>: 49.7, <i>ave.dev</i>: 25.2, <i>st.dev</i>: 29.14, <i>var</i>: 849.18, <i>skew</i>: 0.05, <i>kurt</i>: -1.18. <i>Distr. trans</i>: (<math>\alpha=0, \beta=100</math>)<math>\rightarrow</math>(<math>\alpha=0, \beta=150</math>)<math>\rightarrow</math>(<math>\alpha=0, \beta=200</math>)...</p>
<p><b>Pareto</b> (<math>\alpha = 1, \beta = 1</math>): <i>min</i>: 10.0, <i>max</i>: 6833.0, <i>med</i>: 19.0, <i>mean</i>: 73.56, <i>ave.dev</i>: 86.22, <i>st.dev</i>: 341.25, <i>var</i>: 116455.33, <i>skew</i>: 14.26, <i>kurt</i>: 240.2 <i>Distr. transitions</i>: (<math>\alpha=1, \beta=1</math>)<math>\rightarrow</math>(<math>\alpha=1, \beta=1.5</math>)<math>\rightarrow</math>(<math>\alpha=1, \beta=2</math>)...</p>
<p><b>Poisson</b> (<math>\lambda = 1</math>): <i>min</i>: 0.0, <i>max</i>: 60.0, <i>med</i>: 10.0, <i>mean</i>: 10.0, <i>ave.dev</i>: 7.2, <i>st.dev</i>: 9.8, <i>var</i>: 97.59, <i>skew</i>: 0.96, <i>kurt</i>: 0.88 <i>Distribution transitions</i>: (<math>\lambda = 1</math>)<math>\rightarrow</math>(<math>\lambda = 3</math>)<math>\rightarrow</math>(<math>\lambda = 5</math>)...</p>

We ran the query processor for 25 minutes several times, employing these different execution strategies, and show the results, averaged over all those runs. Figure 5.27(a) compares the average output rate, the average execution time per tuple is presented in Figure 5.27(b), and the run-time execution overheads present in these systems are in Figure 5.27(c)<sup>20</sup>.

From Figure 5.27(a), we can observe that for Uniform distribution, on average, *ST-QM* has 39% higher output rate than CBR without any batching, 24% higher than CBR with batching, and 6% lower than non-adaptive *QM*. In Uniform distribution, most of the time, the streams tend to have a single route. Occasionally, due to sampling, we have noticed two routes per stream in *ST-QM*. However, even with changes in the environment, the routes based on average statistics of the “old” data tend to be the same best routes for the “new” data. This explains the close output rate of *ST-QM* compared to non-adaptive *QM* for Uniform distribution. For Poisson distribution, *ST-QM* on average has 13% higher output rate than CBR without batching, 0-0.5% smaller rate than CBR with batching, and 43% higher output rate than non-adaptive *QM*. Here the simple batching of Eddies incidentally plays out

<sup>20</sup> “QM” in the charts refers to the non-adaptive *QM* execution.

very well, thus resulting in an average performance of *ST-QM* and Eddies being really close. For Pareto distribution, we observe that *ST-QM* on average has 27% higher output rate than CBR without batching, 18% higher than CBR with batching and 44% higher output rate than non-adaptive *QM*. The average execution time per tuple (in Figure 5.27(b)) follows a similar trend.

For Pareto and Poisson distributions, when a concept drift occurs, and most of the data gets processed by the default routes in non-adaptive *QM* system, this results in poor execution strategy, since the data properties have changed and the execution could be improved by determining the new data subsets and customizing the routes for them, as is done in *ST-QM*. CBR, on the other hand, suffers from continuous re-optimization and re-learning overheads (the relative overhead is depicted in black in Figure 5.27(c)). Implemented in the context of Eddies, CBR continuously experiences the “backflow” overhead, where tuples get continuously routed back to the Eddy operator that has to re-examine the tuples and forward them to the next operator for processing. The overhead is  $O(n+1)$  time, where  $n$  equals the number of operators and 1 accounts for the first time a tuple from an input stream gets processed. Without any batching, Eddy processing with CBR algorithm amounted to nearly 20% of the total execution cost.

Batching attempts to reduce Eddy overhead. However, batching in Eddy [33] is still very naive: every  $b$  tuples, i.e., a continuous chunk of tuples that happened to arrive together in time are batched and routed together. Without batching, the Eddy “backflow” overhead per workload of tuples  $W$  is  $O((n+1)*|W|)$ . With batching, the overhead gets reduced by the batch size  $b$ , resulting in the total overhead  $O((n+1)*|W|)/(b)$ . In practice, the batches might be smaller, depending on the arrival rates of the tuples. In *QM*, on the other hand, tuples are grouped together into the same *ruster* based on the classification, i.e., the data values and the similarity of statistics, and are thus guaranteed to share the same best route.

Eddies employing CBR also experience continuous overhead of re-computing classifier attributes based on runtime information, even though the best classifier attribute

for an operator does not change very often [28]. These overheads limit the benefit that can be obtained from a better adaptive policy in Eddy. Static *QM* and *ST-QM* also have a small runtime overhead, namely the probing of the online classifier to determine the execution plan for arriving data. The classification overhead, however, was measured to be very small, only 2% of the query execution cost (Figure 5.27(c)).

### Adaptivity to Concept Drifts

This experiment evaluates how *ST-QM* adapts to different concept drifts. We use non-adaptive *QM* execution as a base case to compare *ST-QM* results.

A *virtual concept drift* means that the data values change, but the distributions of the new content groups stay the same, thus affecting the classifier component but not the target routes. To simulate only virtual concept drifts, we generate data using one of the experimental distributions, and then over time replace the data values with different values, while maintaining the same distribution of data values. Thus, the content of data changes, but their frequencies stay the same. A real life example when this scenario may happen is the variation between the number of times a web server is accessed per minute. Depending on the day (e.g., work day or weekend), the hour (e.g., morning or evening) the values may be different, but the overall distribution typically tends to follow Poisson distribution [63]. We show the results for the Poisson distribution here, but similar trends have been observed for other distributions as well. Figure 5.27(d) shows the results for *ST-QM* compared to non-adaptive *QM*. *ST-QM* gives, on average, between 24% to 38% improvement over static *QM* execution.

In *real concept drift*, the data values stays constant, but the execution routes change. Real concept drift may occur due to changes in either the selectivities, the costs of query operators, or both. Typically, a change in selectivity indicates a change in the data distribution, and thus most likely a hybrid concept drift. Therefore, to simulate only real concept drifts, we vary the time it takes an operator to process a tuple over time (with non-changing data values) and report the effects on *ST-QM*'s

performance. To motivate the exploration of the space of higher operator costs, consider the following example: [228] describes multilingual query operators, e.g., *LexEQUAL* and *SemEQUAL*, for matching multilingual names and concepts, respectively. If over time, the user is not happy with the results produced by the queries composed of such operators, the user may increase the quality threshold [228], which may result in more detailed computations by such operators for certain phonemically close words. In our experiments, the increase in operator cost is obtained by running CPU intensive computations every time a tuple has to be processed by an operator, and varying this cost depending on the tuple’s data values. Figure 5.27(e) shows that *ST-QM* is quite effective at detecting and adapting to real concept drifts. On average, *ST-QM*’s approach results in 15 to 28% faster output rate than the non-adaptive *QM* case.

For hybrid concept drift, we varied both data values and operator costs. Figure 5.27(f) shows the results for continuous hybrid concept drift occurrence, i.e., when both virtual and real concept drifts take place together. We can observe, that *ST-QM* outperforms non-adaptive *QM* by 24% to 41% in hybrid concept drift case.

#### Run-time Overhead of *ST-QM*

*ST-QM* has three overheads: monitoring, analysis and actuation. We instrumented the code to determine the time spent by each of these overheads. Figure 5.28 reports the overheads per workload of tuples relative to the total execution cost. A workload in this experiment is a set of data tuples received and processed during time interval between any two *ST-QM* invocations.

The monitoring overhead per tuple was measured as the time taken by the function that performs sampling and makes the decision whether to discard or keep the sample (see Section 5.8.1). For execution statistics monitoring, we have instrumented each operator to measure the time spent computing the statistics (selectivities and execution cost) for each “statistics” *ruster* (Section 5.8.2). The analysis overhead was measured as the time taken by the function that performs concept drift detection, to



invoke the optimizer, and to produce tuning recommendations (see Section 5.9). The actuation overhead was measured as the time taken to replace the current classifier with a new classifier (described in Section 5.10).

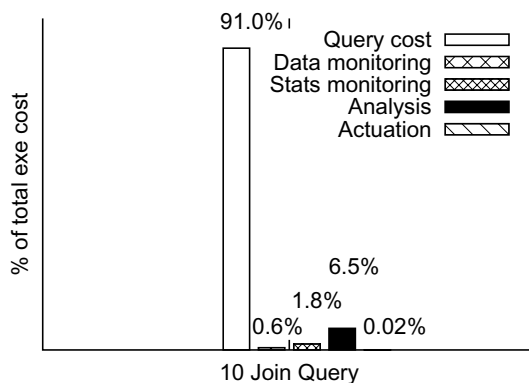


Figure 5.28. ST-QM overhead.

The total overhead (monitoring together with analysis and actuation) is 2.42% of the total execution time without optimizer invocation, and 8.92% with optimizer invocation. One important parameter to control the overhead of *ST-QM* is the size of the training tuple set or the new tuples' sample size. The more tuples get collected, the larger is the analysis overhead and the optimizer overhead. The optimizer overhead is especially sensitive to the size and type of training tuples collected, as was previously reported in [219]. A balance must be kept between the size and the quality of the training data.

In addition, we also measured the worst case scenario for *ST-QM*: when no concept drift occurs and the adaptation is not needed. If there are no changes in the environment, no benefit can be gained from changing to a different *QM* solution. Thus, differences in the output rates must be due to extraneous overhead (and not due to better decisions). For this experiment, we ran our experimental query over the Poisson-distributed dataset without any changes to the data and with *ST-QM* functionality enabled. Figure 5.28 displays the average over 5 runs of the query. When

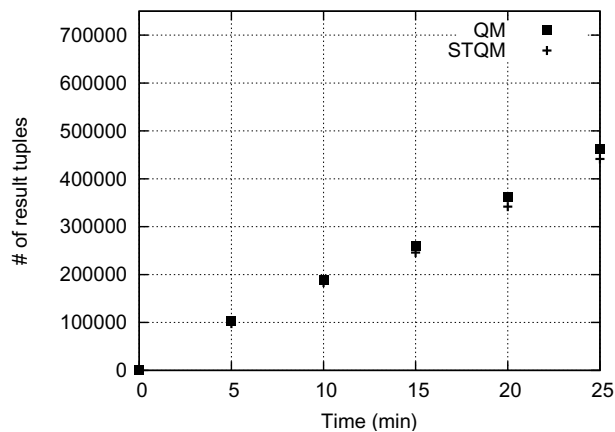


Figure 5.29. Overhead when no adaptation is needed.

no benefit is possible, *ST-QM* is on average between 2.2 - 4.8% worse than static *QM* in the total number of results produced. This result confirms that *ST-QM* approach has detected that changes were insignificant, based on its monitoring and concept drift detection and did not invoke the optimizer. By discarding such insignificant adaptivity cases early, it minimized its adaptivity overhead. This overhead can be further reduced in the system by minimizing the monitoring frequency of both data and execution statistics.

### 5.11.3 Summary of ST-QM Experimental Conclusions

The main points of our experimental study can be summarized as follows:

1. *ST-QM* can give up to 44% improvement in execution time and output rate.
2. *ST-QM* is highly adaptive to virtual, real and hybrid concept drifts and can result in some cases in up-to 41% improvement compared to non-adaptive *QM*.
3. The runtime overhead of *ST-QM* relative to query execution is small (at most 7%) . The actuation cost of physical adaptivity is nearly negligible resulting in 0.02% of total execution cost.

4. Even if no adaptivity is needed, *ST-QM*'s performance in the worst case will be at most 2-4% slower than of static *QM*.

### 5.12 ST-QM Conclusion

Here we addressed the problem of *adaptivity* in the multi-plan-based query processing engines. We have presented a *Self-Tuning Query Mesh (ST-QM)* architecture that uses multiple plans for processing different subsets of data, and yet is as adaptive as the “plan-less” systems. *ST-QM* increases the efficiency of query processing in highly dynamic environments, by adapting the multi-plan solution, so that different subsets of data may benefit from different execution plans over time. *ST-QM* approach is unique in that it abstracts the problem of adaptive query processing as a concept drift problem. Such abstraction allows *ST-QM* to discard adaptivity candidates early in the process, if the changes are insignificant to adapt to and thus minimize the adaptivity overhead. The key characteristic of the *ST-QM* approach is that all logical changes to the current *QM* solution get translated into a simple physical operation, namely the classifier change. Our most important contribution is that we have shown in our prototype implementation that *ST-QM* approach can be simultaneously inexpensive and adaptive. Our experimental study indicates that *ST-QM* can adapt to different types of concept drifts very efficiently. Furthermore, the run-time overhead of *ST-QM* execution is fully amortized by the performance benefits of the better multi-plan-based query processing.

Here, we address the problem of adaptive query processing on non-uniform data streams. We propose a *Self-Tuning Query Mesh* infrastructure (or short *ST-QM*) that continuously adapts to data streams' characteristics and to system conditions, e.g., memory, CPU resources availability. The fundamental challenge for self-tuning query mesh is the problem of determining the discrepancy between the previously learned query mesh model and the current model based on the characteristics of the new data and the system condition, what we denote as *optimization concept drift* problem. The self-tuning query mesh has the ability to judiciously determine *when*

and *how* to adapt its infrastructure to accurately match the changed concept of the data streams and employ the best query mesh for the current system conditions. *ST-QM* used a *three-fold* adaptation process - classifier tuning, multi-route configuration tuning, and runtime route tuning - to ensure efficient processing of continuous queries on non-uniform data streams. We have described the tuning techniques in *ST-QM* and have presented detailed experimental evaluation.

### 5.13 Uncertainty-Aware Query Mesh (UA-QM)

Recent years have witnessed the emergence of novel applications where incoming data arrives in the form of continuous data streams, for example, location-based services, sensor networks and financial tickers. Data in such applications tends to be non-uniformly distributed, and query processing can often benefit from employing *multiple execution plans*, each optimally serving a subset of data with distinct statistical properties. Recently proposed *Query Mesh (QM)* framework implements such *multi-plan* (or *multi-route*<sup>21</sup>) execution paradigm very efficiently. However, similar to most query processing systems, *QM* optimizer assumes all knowledge to be *certain* and *complete* when determining a low cost multi-route solution. Such assumption is unrealistic for streaming environments which are riddled with uncertainty, due to measurement inaccuracies, incomplete or unknown information or data arrival latencies. Here, we focus on the problem of uncertainty in the multi-route query processing context, and propose a novel *Uncertainty-Aware Query Mesh* solution (or short *UA-QM*) to address this problem. The goal of *UA-QM* is two-fold: (1) to model and measure various types of uncertainty to represent real-life scenarios in streaming environments more accurately and (2) to process data in an uncertainty-aware and multi-route fashion. We have implemented our approach in a prototype DSMS, and our experimental evaluation shows the benefits of our proposed *UA-QM* approach.

---

<sup>21</sup>We use terms “plans” and “routes” interchangeably in our work. Both mean the same thing in the context of this paper.

### 5.13.1 Problems with Uncertainty Ignorance

Compared to traditional relational databases where the entire dataset is present and so are the complete statistics about it, this luxury is not available in DSMSs. Here, the knowledge about the environment such as data input rate, operator selectivities, attribute values and their distributions is typically incomplete and is continuously changing. Thus, uncertainty naturally arises during query optimization in the streaming context. Most query processors in Data Stream Management Systems (similar to relational counterparts), however, consider all knowledge to be certain and complete during optimization phase, which may significantly limit query performance at runtime [77]. Existing solutions dealing with uncertainty e.g., [77–79] are primarily *single-plan*-based and focus on cardinality estimations for the data *as a whole*. A more complex structure and a different execution paradigm (that employs unique plans for distinct subsets of data) makes these uncertainty solutions insufficient for a multi-route solution like query mesh (see Section 5.14.2 for more detailed explanation).

As a motivating example, consider a geo-social networking application, such as *BrightKite* [18]. Here, an input data stream *people* may be transmitting real-time location updates from the users looking to get together to socialize in a given geographic area. A continuous query  $Q$  (shown at the top left in Figure 5.30) is executing to match people based on similar age, interests and location. For simplicity of discussion, we assume that the stream *people* has two distinct data subsets, denoted as the “*city*” and the “*suburbs*” subsets<sup>22</sup>. Figure 5.30 (the table at the bottom left) shows the selectivities of operators  $OP_1$ - $OP_3$  for each of the subsets, and the overall selectivity. Here, the selectivities are represented as “certain” point estimates – a common approach in most database systems. Assuming that operators have the same execution costs and only overall selectivities are considered, the best ordering for *people* stream tuples for query  $Q$  is  $OP_2$ ,  $OP_3$ ,  $OP_1$ . However, if we distinguish

<sup>22</sup>This could be people living in a city and in the suburbs.

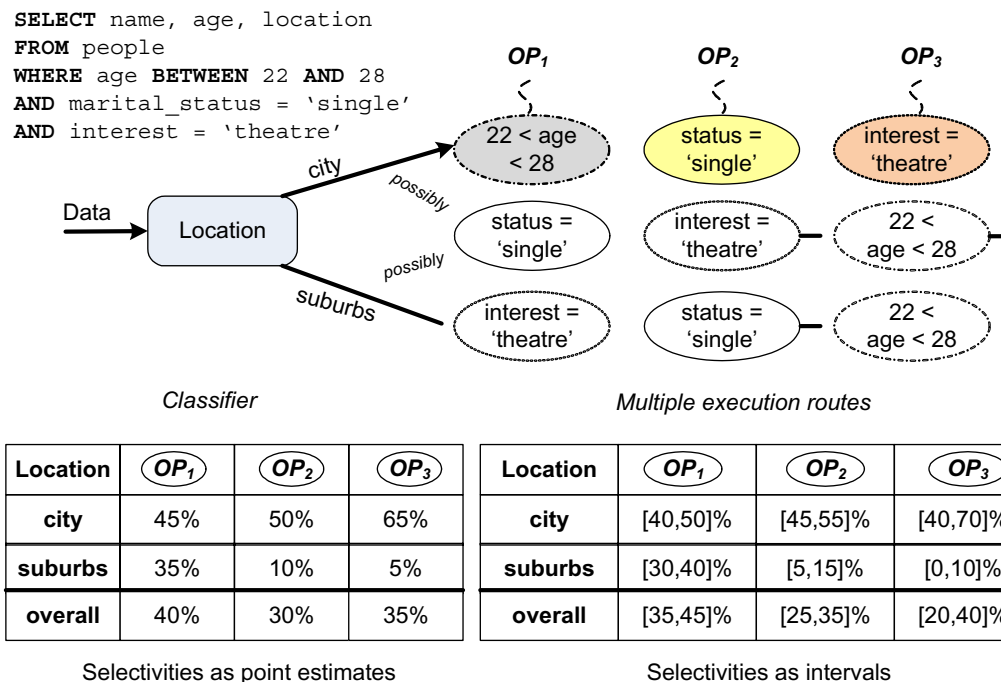


Figure 5.30. Geo-social networking query example.

operators' selectivities based on the different subsets, we can see that for the “*city*” tuples, the ordering  $OP_1, OP_2, OP_3$  will outperform  $OP_2, OP_3, OP_1$ , while  $OP_3, OP_2, OP_1$  will outperform  $OP_2, OP_3, OP_1$  for the “*suburbs*” tuples.

While the above example clearly motivates the benefit of processing different data using different plans, it has a major limitation: it completely ignores uncertainty. For example, uncertainty may be present in the operator selectivities due to varying estimates based on (several) data samples, each roughly approximating the real data, or due to data arrival latency. Uncertainty in operator statistics translates into uncertainty in execution routes. Furthermore, the classifier, which is used to assign routes to data tuples, due to the training set quality or its limited size, may contain some uncertainty as well.

Consider the same example in Figure 5.30 at the bottom right, where instead of certain point estimates, we use *selectivity intervals* to represent uncertainty in

operator selectivities. With uncertain selectivities, it becomes much more challenging to determine which data tuples should be processed using which of the present routes. For example, the “*city*” tuples may benefit from the ordering  $OP_1, OP_2, OP_3$ , but also in some cases, it could benefit from the alternative ordering  $OP_2, OP_3, OP_1$  (depicted by a dashed line) due to the overlap in the operators’ selectivity intervals. Similarly, some “*suburbs*” tuples may benefit from the alternative routes if uncertainty is considered during query optimization.

In addition to uncertainty in routes, classifier may also face uncertainty. For simplicity of presentation, the classifier in our example consists of a single test on the *location* attribute. However, in real-life scenarios, a classifier is likely to have multiple test nodes based on which tuples are assigned to their best routes. Thus, given uncertain routes and uncertain route assignments, as well as the training set only roughly approximating the real data, a classifier is likely to contain some uncertainty as well.

In summary, the problem with many current optimization techniques is that they are: (1) mostly uncertainty-oblivious; and (2) those that are uncertainty-aware, primarily focus on uncertainty in a single query plan execution strategy. Here, we propose to address the open problem of uncertainty in multi-route query processing in the context of data streams.

### 5.13.2 Challenges

A number of characteristics inherent in streaming environments make the problem of handling uncertainty in a multi-route solution a challenging task.

- *Fast data arrival rate.* A common characteristic of data streams is a high data volume and a rapid arrival rate. Therefore, uncertainty estimation algorithm needs to be as fast as possible and the speed of decision-making regarding uncertainty must be faster than the data incoming rate.

- *Different types of uncertainty.* Uncertainty in a multi-route solution may occur in both routes as well as in classifier which is responsible for assigning tuples to routes. Moreover, uncertainty may be “*absolute*” (with regard to actual values, such as statistics measurements, e.g., order of operators), or it may be “*relative*” (with regard to the best choice among multiple possible alternatives). Thus, an uncertainty mechanism must be able to model and measure various types of uncertainty in both routes and classifier as well.
- *User preferences.* Users executing continuous queries may have different preferences regarding the best way of dealing with uncertainty. Users may want to decide what should be a reasonable tradeoff between certainty vs. possibility (in other words, expectation and ambiguity) when their query is being executed. Thus, uncertainty mechanism should provide support for user preferences with regard to how to handle uncertainty during query processing.
- *Low overhead.* The results in streaming environments are expected to be produced in near-real time. Since the added uncertainty-awareness functionality adds processing and storage overheads (compared to regular “uncertainty-oblivious” query processing), the overhead must be as low as possible not to seriously impact the performance of DSMS.

We address the above-mentioned challenges in the context of data streams and present a solution, based on the multi-route query mesh model [219], which we call *Uncertainty-Aware Query Mesh (UA-QM)*.

### 5.13.3 Our Proposed Solution: UA-QM

*UA-QM* contributions can be summarized as follows:

1. *Model.* We propose uncertainty model where both *absolute* and *relative* uncertainties are modeled *symmetrically* for both execution routes and classifier in



a query mesh. Absolute uncertainties are represented using *uncertainty intervals* and relative uncertainties using *belief functions*. The *symmetric* property provides a simpler model and similar uncertainty processing in different components of query mesh. (Section 5.14).

2. *Optimization*. We describe uncertainty-aware optimization algorithms including the computation of multiple execution routes and classifier induction under various uncertainty scenarios (Section 5.15).
3. *Execution*. We discuss how uncertainty is handled at runtime when executing a continuous query in the *UA-QM* framework (Section 5.16).
4. *Experiments*. We have implemented *UA-QM* in a prototype DSMS called *CAPE* [8]. We present our experimental analysis showing the benefits of our proposed approach.

## 5.14 UA-QM Framework

### 5.14.1 UA-QM Architecture

Figure 5.31 gives an overview of *UA-QM* architecture which builds on top of the core query mesh framework [27, 219]. We have designed *UA-QM* to be highly modular, enabling uncertainty-awareness functionality to be turned on/off with complete transparency to the core *QM* framework (bottom of the Figure 5.31). The architecture is easily extensible: new algorithms, heuristics and metrics can be added without much disturbance to the rest of the system.

The key components of *UA-QM* include: (1) *QM optimizer* with uncertainty extensions, (shaded half-way in Figure 5.31) and described in Section 5.15, (2) *Belief Space* nodes, used to represent uncertainty in routes and classifier (Section 5.15), (3) *Belief Handler*, a component responsible for resolving uncertainty by taking into consideration user preferences (described in Section 5.16), and (4) *Uncertainty encoding*

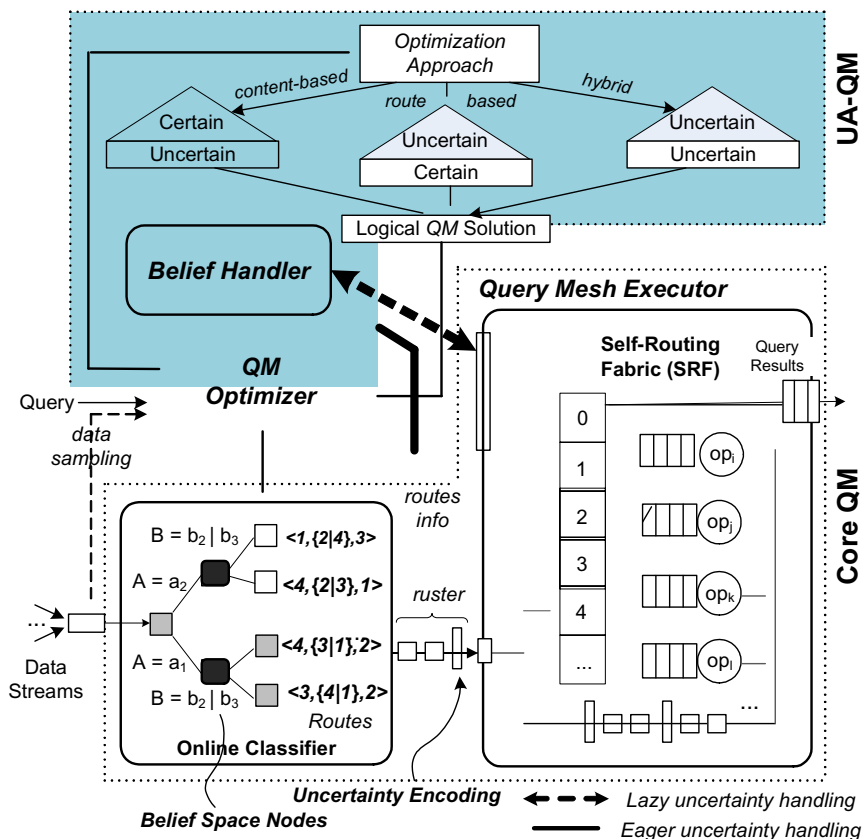


Figure 5.31. UA-QM architecture.

used in routes' specifications (discussed in Section 5.16). We present each of these components, their functionality and execution in detail in the rest of the paper.

#### 5.14.2 Uncertainty Cases in Query Mesh

Compared to a single plan solution, where uncertainty may be present in only one route, a multi-route solution may have uncertainty in several routes, as well as in the classifier. For classifier, we employ a *decision tree* model, as it is one of the most commonly used and efficient classification models. Under route uncertainty, we focus on uncertainty in *operators' selectivities*, and under classifier uncertainty, we consider

uncertainty in the *impurity measures* (specifically, in *information gain*<sup>23</sup> [141]) that have a direct impact on the structure and size of the classifier. Next, we describe the three possible uncertainty cases that may occur in a multi-route *QM* solution.

**Case 1:** *Certain Classifier and Uncertain Routes.* By uncertainty in routes we mean uncertainty in the routes' costs as a result of imprecision in individual operators' selectivities. This leads to ambiguity about the best operator ordering and it may occur when the query mesh optimizer uses a strategy similar to *Content-Learns* algorithm [28], or *Content-Based Approach (CBA)* [219] to find a low cost *QM* solution. The main idea of these optimization strategies is to partition the training dataset into groups based on the similarity of data values first, and then compute the routes for each content group. Different data values may imply unique distributions and statistics, and thus possibly various execution routes. Here, the statistics of the routes may be imprecise, yet the classification based on the training data and the partitions they belong to (determined based on the training data values) is considered to be certain (Figure 5.32(a)).

**Case 2:** *Uncertain Classifier and Certain Routes.* This scenario is the reverse of the above case. Here, routes are assumed to be certain and the classification may be uncertain (Figure 5.32(b)). This case may occur when the *QM* optimizer uses a *Route-Based Approach (RBA)* [219] to find the best multi-route query mesh solution for a given query. The main idea of this optimization strategy is to compute routes first, using all available statistics (possibly coming from multiple samples of data), and then assign the training data to the existing routes. The statistics used for routes' computations are assumed to be complete and reliable (after being collected over many runs of the same query), yet the training dataset (of limited size) used for the classifier induction may depict real data with some inaccuracy, thus resulting in classification model with uncertainty. This case may also occur when using a large training dataset for classifier induction is prohibitive [141], and some training data

---

<sup>23</sup>Other measures of impurity, such as *entropy* or *gini index* [141] could be used here as well.

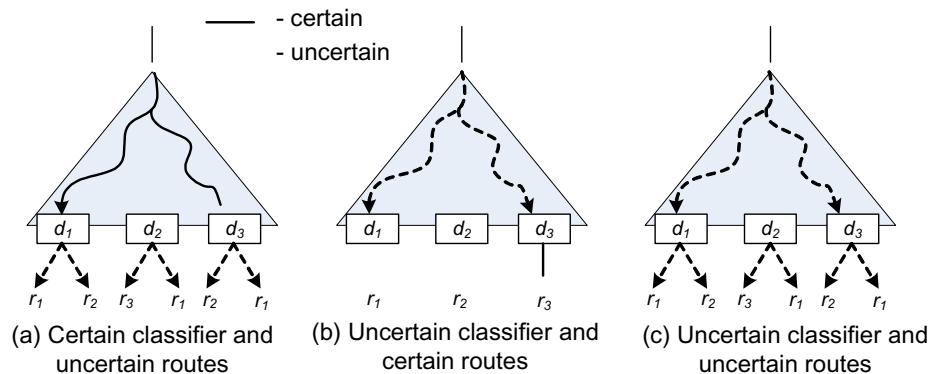


Figure 5.32. Uncertainty scenarios in  $QM$ .

must be eliminated from being used in classifier induction – the phenomenon known in machine learning as “pruning” [229].

**Case 3: Uncertain Classifier and Uncertain Routes.** The third case is the composite of the above two cases, where uncertainty is present in both the routes and the classifier (Figure 5.32(c)). Here, a data tuple may belong to more than one data subset (or a distinct group), and more than one route may be considered to be possible for processing of that subset. This case may occur when a *hybrid optimization approach* is used by the  $QM$  optimizer: two  $QM$  solutions are computed, one using the *CBA* method and another using the *RBA* method (as described above), and then “merged” to produce the “best” overall  $QM$  solution.

### 5.14.3 Reasoning About Uncertainty

In order to address the problem of uncertainty, we need a method for representing and measuring it. In robust query processing, the common approaches for modeling uncertainty include probability distributions (or short PDs) [77] or bounding boxes (or short BBs) (also known as bounding intervals) [78]. Both methods have their advantages and limitations. PDs, for instance, give an intuitive representation for users to decide how much uncertainty they are willing to “tolerate” when planning an

execution strategy at compile-time. BBs, on the other hand, easily capture variations and imprecisions in statistics, e.g., possible *min*, *max* and expected bounds. It also allows a query processor to check the latest statistics at runtime, and determine which *concrete* execution solution applicable within the bounding interval should actually be employed.

Our uncertainty model includes the strengths of both of the above approaches and enables both user-driven [77] and system-driven [78] responses to handling uncertainty. What sets our model apart from the existing techniques is that we model two types of uncertainty, namely the *absolute uncertainty* and the *relative uncertainty*, and we model them *symmetrically* for both routes and classifier in query mesh. We believe that such two-way uncertainty modeling can represent real-life scenarios more accurately, and the *symmetric* property facilitates a simpler model and similar processing for different components in query mesh.

Informally, absolute uncertainty represents the uncertainty in actual values (e.g., in operator statistics or in attribute impurity estimates), whereas relative uncertainty models the ambiguity about the choice among possible alternatives (e.g., the best order of operators in routes, or the choice of the best splitting attribute in classifier when multiple options are possible). In *UA-QM*, we employ *uncertainty intervals* for modeling absolute uncertainties (see Section 5.14.4) and *belief functions* from Belief Function theory to represent relative uncertainties (see Section 5.14.5).

#### 5.14.4 Absolute Uncertainty

We distinguish between two types of uncertainty intervals in *UA-QM*, namely the *selectivity intervals* and the *classification intervals* to model absolute uncertainty.

*Selectivity intervals* (or short *SINs*) represent uncertainty in operators' statistics. Table 5.5 shows an example. Here, the selectivity of  $OP_1$  for the distinct subset  $d_1$  might be not known with certainty, but it is known to be between 40% and 50%, and is represented by the interval [40,50]. Hence, the interval describes the possibility

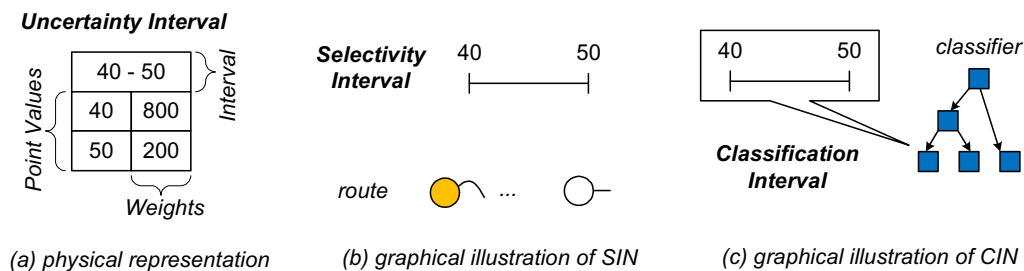


Figure 5.33. Symmetric modeling of *SINs* and *CINs*.

distribution of  $OP_1$  selectivity for subset  $d_1$ . Viewed in this perspective, the entries in Table 5.5 in the column *SIN* are the possibility distributions of the values of selectivities for different subsets of data.

Table 5.5  
Selectivity intervals for various subsets.

Stream	Operator	Subset	<i>SIN</i>
$S_1$	$OP_1$	$d_1$	[40,50]
		$d_2$	[30,40]
		$d_3$	[35,45]

*Classification intervals*<sup>24</sup> (or short *CINs*) represent uncertainty in the *information gain* values of different attributes that are used in determining the best splitting attribute when constructing *QM* classifier. One of the basic steps in decision tree classification is to select the splits based on attributes and data values that are used to predict membership in the terminal nodes of the decision tree classifier (in the context of our work, terminal nodes represent the various execution routes). In general terms, the split at each node found will generate the greatest improvement in predictive accuracy. This is usually measured with an impurity measure, which provides an indication of the relative “homogeneity” of tuples in the terminal nodes of

<sup>24</sup>A more precise term would be “*impurity measure intervals*”, but we decided to choose a more general name – “*classification intervals*” – to characterize where in *QM* the uncertainty takes place.

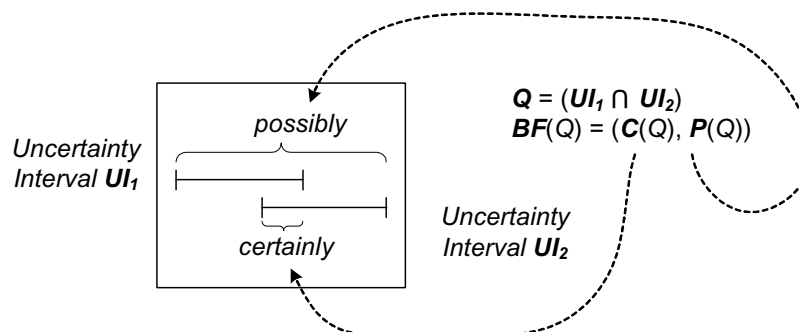


Figure 5.34. Belief function for a relative uncertainty.

the classifier. For example, if all training tuples in each terminal node have identical values, then node impurity is minimal, homogeneity is maximal, and prediction is perfect (at least for the case of training tuples used in the induction of classifier). We omit the discussion of information gain and how it is computed and refer the reader to [141] or any machine learning textbook.

A classifier interval  $CIN=[0.4,0.5]$  represents a range of impurity measure values between 0.4 and 0.5 (or in absolute terms  $[40,50]$ , meaning between 40% and 50%). Due to symmetric property of our model, both  $SINs$  and  $CINs$  are represented by the same physical data structure (shown on the left in Figure 5.33). Consider, for example, an uncertainty interval  $UI [40,50]$  (which could be either a  $SIN$  or a  $CIN$ ).  $UI.Interval$  stores the interval value, between 40 and 50.  $UI.PointValues$  represent the original point value estimates measured and used in determining the overall interval.  $UI.Weights$  correspond to the weights of their respective  $UI.PointValues$ . For example, in Figure 5.33 in the case of a  $SIN$ , the weight of 800 represents the count of tuples that reported to have the point estimate of selectivity equal 40% and the weight of 200 indicates, that 200 tuples contributed to the selectivity point estimate 50%. The weight values could be based on the absolute count of tuples from the samples that have resulted in that particular point estimate value, when the selectivity interval was computed. Alternatively, weights can be represented using relative percent values instead of absolute counts.

### 5.14.5 Relative Uncertainty

To model *relative uncertainty*, we use the concepts from the *Belief Function Theory* also known as *Dempster-Shafer Theory* [230,231]. Relative uncertainties in routes and in the classifier are expressed in the form of *belief functions*. Belief functions provide a very intuitive way to model ambiguity (compared to classical probability framework), and allow incorporating subjectiveness in uncertainty [77]. Furthermore, if evidences come from multiple sources (e.g., from different samples of data collected at different times), the model provides a flexible and adaptive way to combine those evidences<sup>25</sup>. Another attractive aspect of belief functions framework is its flexibility – it can be reduced to the Bayesian framework under certain conditions. Figure 5.34 shows a conceptual idea of a belief function. Here, two uncertainty intervals (which could be either *SINs* or *CINs*) are depicted as  $UI_1$  and  $UI_2$ . Under relative uncertainty, the question  $Q$  we are interested in is – *how large is the overlap between the two intervals  $UI_1$  and  $UI_2$ ?* This question  $Q$  denotes the intersection of the uncertainty intervals and precisely characterizes the relative uncertainty regarding  $UI_1$  and  $UI_2$ . Since both  $UI_1$  and  $UI_2$  are uncertain, the answer to this question can be constructed to have two parts, one relating to the *certainty*  $C$  in the answer and the other to its *possibility*  $P$ , and symbolically can be expressed as  $BF(Q)=(C(Q),P(Q))$ . The certainty parameter can be viewed as the expectation and the possibility parameter as the ambiguity. Similar to absolute uncertainties, relative uncertainties, are modelled symmetrically for both routes and classifier in  $QM$  as described below.

A *Selectivity Belief Function (SBF)* represents a belief in the intersection of any two *SINs*. A route (query plan) optimization algorithm<sup>26</sup> must determine the best order of operators. If any of the operators' *SINs* are overlapping, this translates into the uncertainty about which operator should come first. Figure 5.35 illustrates the possible cases for uncertainty intervals' intersections<sup>27</sup>: *UIs* may be completely *non-*

<sup>25</sup>Here, we use a basic form of belief functions. More advanced features like adding subjectiveness to evidence, etc., we reserve for our future work.

<sup>26</sup>This can be any of the state-of-the-art techniques from the literature, e.g., [58, 66, 209, 210].

<sup>27</sup>Due to model symmetry, the same logic applies to *CINs* as well.



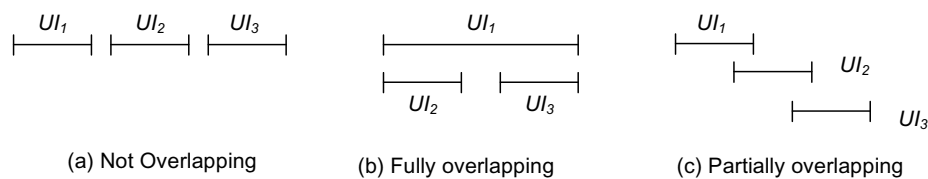


Figure 5.35. Cases for uncertainty intervals' overlaps.

*overlapping* (5.35a), *completely-overlapping* (5.35b) or *partially overlapping* (5.35c).

For overlapping selectivity intervals, *SBFs* are computed for the following three cases:

- (1)  $SBF_1$ : operator with  $SIN_1$  must come first
- (2)  $SBF_2$ : operator with  $SIN_2$  must come first
- (3)  $SBF_3$ : either of operators may come first

Figure 5.36 shows a concrete example of partially overlapping *SINs*. The values of *SBFs* for this example are as follows:  $SBF_1 = (800,1000)$ ,  $SBF_2 = (400,1000)$ , and  $SBF_3 = (800,2000)$  when expressed in absolute terms, and after normalization:  $SBF_1 = (0.8,1)$ ,  $SBF_2 = (0.4,1)$ , and  $SBF_3 = (0.8,2)$ .

*Classification Belief Functions* (*CBFs*) represent beliefs in the relative intersections of *CINs* and are modeled similar to *SBFs*. Thus, we omit the details of *CBFs* computation, as it is the same as for *SBFs*. *CBFs* represent relative uncertainty about impurity measures for various attributes, which translates into uncertainty about the best order of splitting attributes when computing the classifier.

In *UA-QM*, Belief functions (both *SBFs* and *CBFs*) are resolved to concrete answers, e.g., specific classification nodes and operators in the routes based on user preferences with respect to uncertainty. Belief functions can be resolved in an “eager” and a “lazy” manner during execution as described in Section 5.16.

## 5.15 UA-QM Optimization

In this section, we describe uncertainty-aware multi-route query optimization algorithm. We consider three possible uncertainty cases presented in Section 5.14.2.

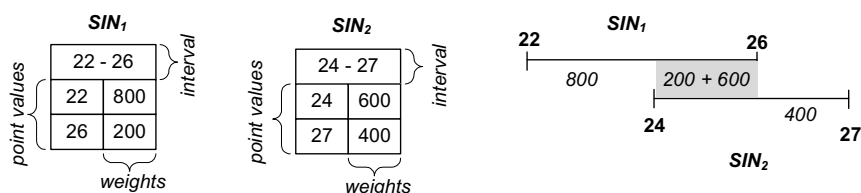


Figure 5.36. Example of computing SBF.

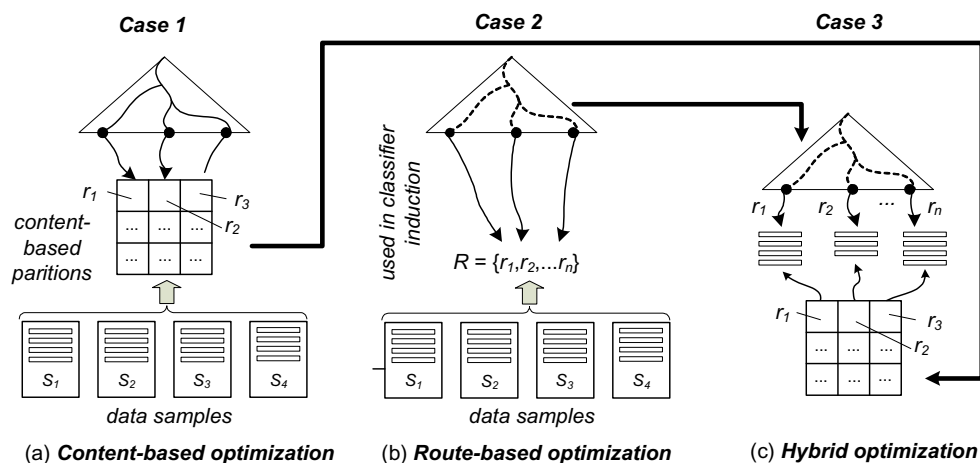


Figure 5.37. UA-QM optimization approaches.

Figure 5.38 illustrates the pseudocode for the overall uncertainty-aware multi-route query optimization algorithm and Figure 5.37 visually depicts the cases.

**Case 1: Certain Classifier and Uncertain Routes:** As we have previously stated, this case may occur when  $QM$  optimizer uses a *Content-Based Approach* (CBA) to find a good query mesh (Figure 5.37(a)). Using CBA, the algorithm first divides data into partitions based on similarity of values (Figure 5.38, Line 2), and then computes the execution routes for each content-based partition (Figure 5.38, Line 3).

*Computation of Uncertain Routes:* The pseudocode for the procedure computing uncertain routes is shown in Figure 5.39. It starts off by computing selectivity intervals for each operator and each content-based partition (Line 1). The operators are then ordered by the monotonically increasing selectivity intervals (Line 2). After the  $SIN$ s

```

UA-QM-Optimization ( $T$  training dataset represented
by a collection of data samples  $\{k_1, k_2 \dots k_n\}=T$ )
// Case 1: Certain Classifier and Uncertain Routes
01 if (optimization method == Content-Based)
02    $D = \{d_1, d_2 \dots d_n\}$  // content-based partitions
03    $R_u = \text{ComputeUncertainRoutes}(T, D)$ 
04    $C_c = \text{InduceCertainClassifier}(T, D)$ 
05   return new UA-QM( $C_c, R_u$ )
// Case 2: Uncertain Classifier and Certain Routes
06 else if (optimization method == Route-Based)
07    $(R_c, D) = \text{ComputeCertainRoutes}(T)$ 
08    $C_u = \text{InduceUncertainClassifier}(T, D, R_c)$ 
09   return new UA-QM( $C_u, R_c$ )
// Case 3: Uncertain Classifier and Uncertain Routes
10 else if (optimization method == Hybrid)
11   Let  $UA-QM_1$  = solution from steps 1-5
12   Let  $UA-QM_2$  = solution from steps 6-9
13   return new MergeUAQMSolutions( $UA-QM_1, UA-QM_2$ )

```

Figure 5.38. UA-QM Optimization.

```

ComputeUncertainRoutes ( $T$  training dataset,
 $D$  content-based partitions)
01  $SIN = \text{ComputeSelectivityIntervals}(T, D)$ 
02  $oSIN = \text{OrderSelectivityIntervals}(SIN)$ 
03  $uSIN = \text{GetOverlappingSelectivityIntervals}(oSIN)$ 
04  $SBF = \text{ComputeSBFsForSelectivityIntervals}(uSIN)$ 
05  $uR = \text{OrderOperatorsWithSINsAndSBFs}(uSIN, SBF)$ 
06 return  $uR$ 

```

Figure 5.39. Computing uncertain routes.

```

ComputeSelectivityIntervals ( $T$  training dataset,
 $D$  content-based partitions)
 $SIN$  -- a hashtable storing  $SIN$ s of all
operators for different subsets of data
01 for (each operator  $OP$ )
02    $SIN[OP] = \text{NULL}$  // No precomputed statistics
      // Compute selectivities using different samples
03   for each sample  $k_i \in T$ 
04     for each partition  $d_j \in D$ 
05       compute selectivity  $s(OP)$  for  $d_j$  based on  $k_i$ 
      // Merge point selectivity into selectivity interval
06        $sin = \text{MergeIntoSIN}(OP, d_j, s(OP), |k_i|)$ 
07        $SIN[OP, d_j] = sin$ 
08        $SIN[OP].\text{Update}(SIN[OP, d_j])$ 
09 return  $SIN$ 

```

Figure 5.40. Computing selectivity intervals.

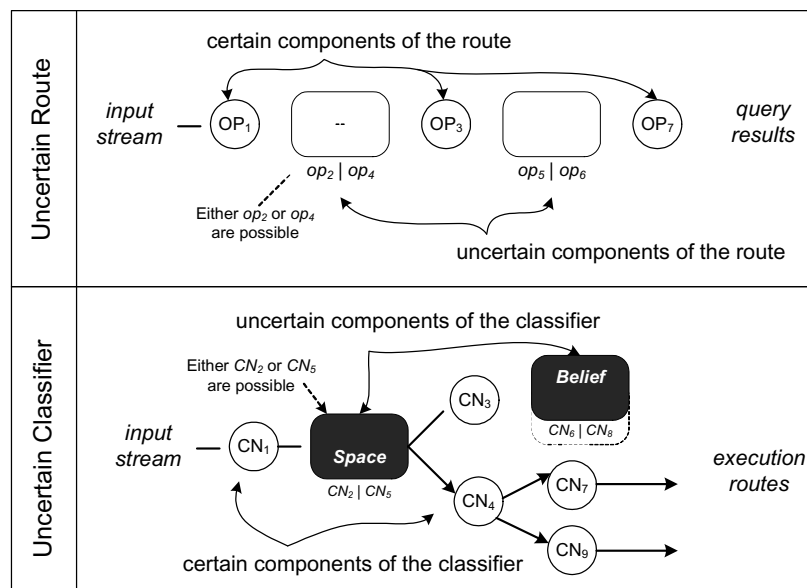


Figure 5.41. Conceptual idea of uncertain routes and classifier.

```

MergeIntoSIN (OP operator,  $d_j$  partition,
s selectivity value, w weight)
01 sin = SIN[OP,  $d_j$ ]
    // if SIN is null, initialize it
02 if (sin == null), then
03     sin.Min = sin.Max = s;
04     return sin;
    // if s is far from the SIN mid-point, return a "conflict"
05 mid = (sin.Max - sin.Min)/2;
06 if ( $\Delta = \text{diff}(s, \text{mid}) > \Delta_{MAX}$ ) then return null;
    // update SIN attributes
07 if ( $s > \text{sin.Max}$ ) then sin.Max = s
08 else if ( $s < \text{sin.Min}$ ) then sin.Min = s
09 sin.PointValue[s] = w
10 return sin

```

Figure 5.42. Merging point estimate into a SIN.

have been ordered, the algorithm determines if any of them are overlapping (Line 3). For all overlapping *SIN*s, *SBF*s are computed (Line 4) as described in Section 5.14.5. After the selectivity belief functions have been established, the algorithm determines the routes by ordering the operators. In every case, where uncertainty intervals are overlapping, i.e., a choice between the operators is uncertain (and there is a corresponding *SBF*), a “*belief space node*” is created in the route (pseudocode is not shown). Figure 5.41 (top) shows a conceptual idea of an uncertain route with belief space nodes.

To complete the uncertainty-aware *QM* solution, the optimizer induces the classifier<sup>28</sup> based on the training data tuples and the subsets they belong to (Fig. 5.38, Line 4), which are the certain partitions defined based on the data content.

<sup>28</sup>The induction algorithm for a “certain classifier” is the same as for regular decision trees, such as ID3, C4.5 from the literature.

```

ComputeUncertainClassifier ( $T$  training tuples,  $S$  - set of attributes
that may be tested by the decision tree,  $R$  - target route attribute
(predicted by the decision tree),  $M$  - data samples)
01  $Root = \text{DecisionTreeNode}(T)$ 
02 if (all tuples of  $T$  are assigned to the same route  $r_i$ )
03    $Root = \text{single-node tree with label} = r_i$ 
04 else if ( $S$  is empty)
05    $root = \text{single node tree with label} = \text{most common value of } R \text{ in } T$ 
06 else
07    $G \leftarrow \text{members of } S \text{ that maximize } \text{InfoGain}(T, A, M)$ 
08   if ( $|G| > 1$ ) // there are overlapping CINs
09      $BS$  is belief space node  $\forall A \in G$ 
10      $Root.\text{addBeliefSpaceNode}(BS)$ 
11      $Root = BS$ 
12     for (every  $A \in G$ )
13       perform the same steps as in Lines 15-22
14   else if ( $|G| == 1$ ) // there are no overlapping CINs
15      $A \in G$  is decision attribute for  $Root$ 
16     for (each possible value  $v$  of  $A$ )
17       add a branch below  $Root$  testing for  $A = v$ 
18        $T_v \leftarrow \text{subset of } T \text{ with } A = v$ 
19       if ( $T_v$  is empty)
20         below the new branch add a leaf with
           label = most common value of  $R \in T$ 
21       else
           // below the new branch add subtree
22        $Root.\text{addBranch}(\text{ComputeUncertainClassifier}(T_v, S - \{A\}), R)$ 
23 return  $Root$ 

```

Figure 5.43. Computing uncertain classifier.

*SIN Computation:* Figures 5.40 and 5.42 illustrate the pseudocode describing the details of *SIN* computation. In Figure 5.40, after the *SIN* hashtable that stores selectivity intervals for all operators is initialized (Fig. 5.40, Line 1), for each operator and partition combination, a point selectivity is estimated using every available collected data sample. The computed value for each data sample is then merged with other point estimates to form a selectivity interval for that operator with respect to that partition (Lines 3-6).

Figure 5.42 shows the pseudocode describing how a point estimate is merged with other estimates to form a *SIN* for an operator. First, if the *SIN* is *null*, then the point estimate  $s$  becomes the *min* and the *max* value of the *SIN* (Lines 2-4). If  $s$  is far from the mid-point of the *SIN*, this is viewed as a “*conflict*” with the current selectivity interval and a *null* value is returned (Lines 5-6). The *conflict flag* (or a null value returned by the procedure) indicates that the selectivity value is significantly different from the selectivity estimates from other samples in that *SIN*. The limit for how far a value may be from the mid-point without causing a conflict is controlled by the system parameter  $\Delta_{MAX}$ . A significant difference may indicate a possible sample outlier or a change in the environment, thus calling for more samples (representing the latest data) to be collected to be used in *QM* optimization. Otherwise, if  $s$  is not conflicting and  $s$  is either smaller or larger than the current *min* and *max*, these values are updated accordingly. The point value estimate with its weight, which corresponds to the cardinality of the sample (used in estimating that measure) is stored inside the *SIN* data structure (Lines (7-9)).

**Case 2: Uncertain Classifier and Certain Routes** This case occurs, when *QM* optimizer uses the *Route-Based* optimization approach (RBA) (see Section 5.14.2 and Figure 5.37(b)). Here routes are computed first, based on all available samples of data and their statistics (Figure 5.38, Line 7). The statistics are considered to be certain, and thus computed routes are assumed to be certain as well. To induce a classifier, the impurity measures for classifying attributes are computed based on all available data samples. Impurity measures for the same attribute may vary for each

data sample, thus leading classification intervals or *CINs*. The presence of *CINs* leads to an uncertain classifier. Figure 5.41 (bottom) shows a conceptual view of an uncertain classifier<sup>29</sup>.

*Computation of Uncertain Classifier:* Figure 5.43 shows the pseudocode for constructing uncertain classifier. First, root node is created (Line 1). If all training tuples are assigned to one route, then the classifier is a singleton, or we refer to it “empty” classifier. The understanding here is that all data should be processed using a single route.

**Case 3: Uncertain Classifier and Uncertain Routes** The third case for multi-route optimization is when both routes and classifier may be uncertain, as a result of merging uncertain *QM* solutions computed using two different optimization approaches (CBA and RBA) – to get a better overall *QM* solution (Figure 5.37(c)).

```

MergeUAQMSolutions (UA-QM1 - CBA-based QM,
UA-QM2 - RBA-based QM)
    // UA-QMfinal is the final (merged) QM
01 UA-QMfinal.C = UA-QM2.C // classifier from RBA-based QM
02 UA-QMfinal.R = MergeRoutes(UA-QM1.R, UA-QM2.R)
03 return UA-QMfinal

```

Figure 5.44. Merging uncertain query meshes.

*Merging Uncertainty-Aware QMs:* Figure 5.44 shows the pseudocode for merging two query meshes.

## 5.16 UA-QM Execution

*UA-QM Executor* receives logical *QM* specification from the optimizer and instantiates physical runtime infrastructure. Conceptually, at runtime, we may have either the classifier with belief space nodes or the routes with belief space nodes which

<sup>29</sup> *CN* stands short for “classifier test node”.



represent the uncertainty about the best option (among several possible alternatives). In this section, we describe how these belief space nodes are resolved at runtime to determine the concrete execution solution. In Section 5.16.2 we describe how beliefs are resolved in *UA-QM*.

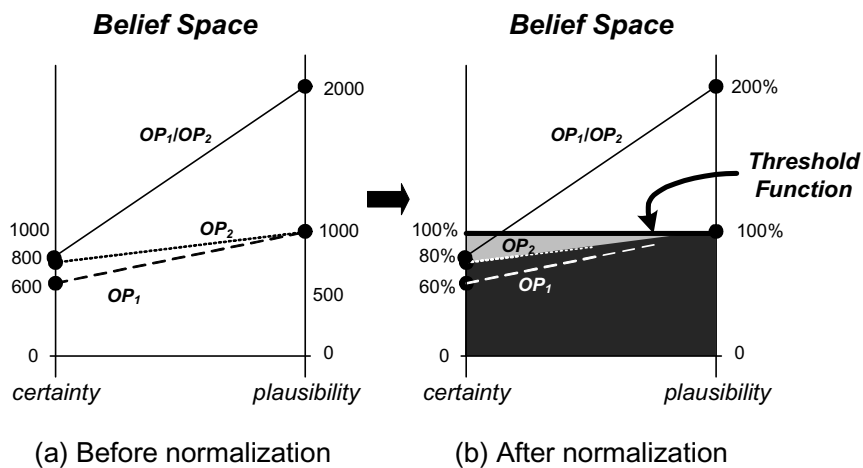


Figure 5.45. Resolving belief functions (route example).

### 5.16.1 Runtime Infrastructure

For efficient query execution, *QM* Executor uses an infrastructure, called the *Self-Routing Fabric (SRF)* (see Figure 5.31) [27,219], which implements query processing via multiple routes with near-zero route execution overhead. In contrast to current adaptive systems, *SRF* eliminates the expensive central data router operator, such as Eddy operator [33,34,97] and enables de-centralized self-routing of data by operators. Route specifications are encoded in meta-data tuples, called “*routing tokens*” (or short *r-tokens*). *R-tokens* are then embedded inside data streams along with their data tuples by the *online classifier operator*. To keep memory and CPU overheads minimal, the tuples are assigned to an existing route in groups called “*routable clusters*” or short “*rusters*” rather than individual tuples. *Rusters* distinguish themselves from traditional batching, e.g., [33,217], in that they are formed by probing the classifier. Hence, only the tuples that share the same best route get assigned to the same *ruster*.

To enable de-centralized routing, routes in the *r-tokens* are specified in the form of an *operator stack* based on the design of *SRF*. The stack nodes represent the indexes of the operators in the *SRF*, e.g., the *r-token*  $\langle 2,3,1,4 \rangle$  indicates that ‘2’ is the first operator in the route, ‘3’ is the next, and so on. A *ruster* is always sent to the operator that is currently the top node in the routing stack. After an operator is done processing the *ruster*, the operator “pops” the top of the routing stack – its unique identifier in the *r-token*, and then puts the *ruster* into the next (now the top) operator’s input queue. When the operator stack is empty, the *ruster* tuples are forwarded to the global output queue reserved by index “0” and then to the application(s).

To enable uncertain route specification, we introduce a small modification to the route encoding: the *r-token*  $\langle 2, \{ \mathbf{3} | \mathbf{1} \}, 4 \rangle$  indicates that ‘2’ is the first operator in the route, ‘3’ or ‘1’ is the next, etc. The encoding  $\{ \mathbf{3} | \mathbf{1} \}$  depicts the uncertainty about the order of operators.

### 5.16.2 Belief Space Handling

Figure 5.45 visually depicts belief functions using a “relationship graph” (or we denote it as “*belief space*”) where certainty vs. plausibility of each belief are plotted against each other. Thus, resulting BFs represent the uncertainty in the overlapping selectivity sub-intervals.

In this scenario, we have a certain classifier and uncertain routes, i.e., routes with “belief spaces”. In order to determine the concrete physical sequence of operators in an uncertain route, the user receiving results of the query provides a *threshold function* to specify how much he or she is willing to believe in (certainty and possibility of) a relative uncertainty. To make this more intuitive, let’s consider what is the act of believing: it consists of creating a threshold of belief at some probability, assembling evidence for the question, and when the probability exceeds that threshold, accepting it as true. Thus, a threshold function specifies the preference of the user with regard to

uncertainty and symbolically is described as  $TF=(C,P)$ , where  $C$  and  $P$  are certainty and possibility parameters.

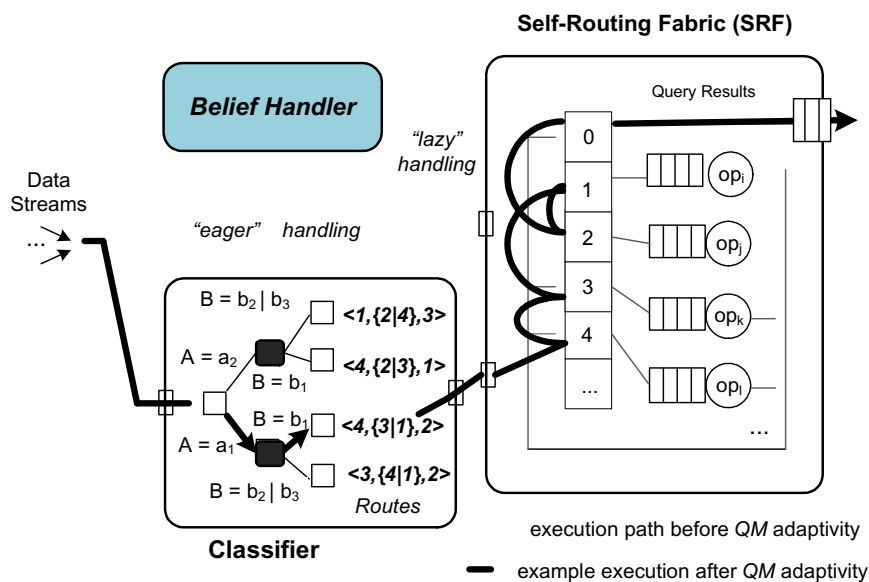
Given the threshold function, a concrete route is determined as follows (see Figure 5.45): for every belief space (depicting relative uncertainty in the order of operators in the route), and the specified threshold function we find the *closest belief function*. If we were to plot belief functions together with the threshold function, we would get something like in Figure 5.45. The threshold function line does not serve as a “cut off” parameter in a traditional sense of a threshold, but rather a desired belief of the user. In order to determine the closest belief function to the threshold function, we employ integration techniques from mathematics (since we need to find a line with the smallest area between that line and the threshold function line).

**Definition 5.16.1 (*Smallest Distance Property*)** Let  $A$  denote area,  $y(x) = BF(OP_i)$  represent the belief function of an operator  $OP_i$ , and  $z(x) = TF()$  represent the threshold function line. Let  $A_{OP_i} = \int_{x=0}^{x=1} y(x)dx$ , and  $A_{TF} = \int_{x=0}^{x=1} z(x)dx$ . We choose an operator  $OP_i$  if it satisfies the following property:  $A_{\Delta} = |A_{TF} - A_{BF}|$  is the smallest  $\forall$  BFs in the belief space.

Thus, using mathematical integration we can find the closest line (belief function) to the threshold function, and make a choice among possible alternatives while considering the preferences of the user.

### 5.17 UA-QM Conclusion

Uncertainty-awareness addresses a major limitation of the most of the existing query processing solutions which typically ignore uncertainty and could result in poor performance, or may lead to frequent re-optimizations, further impeding query performance. Here, we have proposed using the concepts of the belief function theory of evidence can be used as basis for a method that makes multi-route query optimization more robust to uncertainty – both in the plan computation as well as in plan assignment.



In this paper, we propose to address the challenges above and present a framework handling uncertainty problem in a multi-plan (or multi-route) query execution systems. To be practical, an uncertainty mechanism in a data stream environment must provide: (1) fast measurement of uncertainty, (2) efficient and meaningful representation of different types of uncertainty in a multi-route solution, (3) support for user preferences in uncertainty processing, (4) adaptivity to dynamic changes in uncertainty, and (5) very low overhead compared to traditional continuous query processing. Our uncertainty modeling procedure captures both absolute and relative uncertainty and is compatible with the architecture of existing query optimizers, allowing it to be easily integrated into traditional single-plan-based database management system.

## 6 CONCLUSION AND FUTURE RESEARCH DIRECTIONS

### 6.1 Summary

The main goal of this dissertation is to introduce several new features inside Data Stream Management Systems (DSMSs) that are becoming increasingly important requirements for many emerging stream-based applications. Specifically, we tackle the problems of the access control enforcement on streaming data, the tagging of streaming data and the diversity-aware query processing inside DSMSs. The three main contributions of this dissertation can be summarized as follows.

First, the dissertation addresses the problem of continuous access control enforcement in dynamic data stream environments, where both the data and the query security restrictions may potentially change in real-time. The proposed approach advances the state of the art of data stream management systems by introducing: (1) the stream-centric approach to dynamic security, (2) the symmetric security model for both continuous queries and streaming data, and (3) the alternative security-aware query processing methods, that can optimize the execution based on data-related as well as security-related selectivities. Experimental evaluation shows that our proposed approach outperforms other possible alternatives and the security-aware query processing can achieve significant performance benefits over the previously proposed naive pre-filtering and post-filtering methods.

Second, the dissertation proposes a solution for tagging streaming data using a special type of streaming metadata called a tick-tags. Tick-tags can serve a variety of purposes, including labelling or describing some underlying real-time information, and serving as means of disseminating useful knowledge in addition to what is captured by the content of data tuples. Exploiting tags embedded in a data stream increases the kinds of queries that can be executed over data streams. Our experimental results

show the scalability and performance benefits of the tick-tag approach compared to alternative solutions. We have also evaluated the costs of executing tag-aware and tag-oriented continuous queries.

The third contribution in this dissertation is the development of a data diversity-aware query processing framework, called query mesh, that enables different subsets of data to be processed by different query execution plans. We addressed the problem of optimization and efficient runtime execution using query mesh framework, the adaptivity to changing conditions at runtime, and the problem of imprecision and uncertainty in the context of query mesh. The query mesh framework is general, offering numerous advantages over current state-of-the-art solutions. It is applicable to streaming engines and potentially to relational DBMSs as well. Our preliminary experimental results verify the effectiveness of the query mesh approach and demonstrate its potential as a paradigm for continuous query optimization for rich and diverse data.

## 6.2 Future Research Directions

Next we describe the possible directions for future research based on the concepts presented in this dissertation.

### 6.2.1 Security Extensions in DSMSs

Security is paramount to the functioning of any system. In this thesis, we have considered only the problem of access control. However, a full-fledged DSMSs needs a comprehensive security support, which involves different issues of security [232]: authentication, authorization and access control, confidentiality and integrity, availability, auditing, privacy, physical, hardware security, and operating system security.

One interesting direction is to expand security punctuation mechanism further to support streaming data integrity, e.g., providing assurance that real-time data traffic is not altered during the transmission. Another area worth examining is the mining

of security preferences (depicted by streaming security punctuations) of the users and building security policy profiles, based on which the best query execution plans can be constructed in advance and anticipated to be used in the future.

### 6.2.2 Tagging Extensions in DSMSs

In our approach, we tag streaming data using words (strings of characters). An alternative to string-based tags could be object-based tags. An interesting research direction is the investigation whether streaming data could be tagged with objects instead of keywords. Instead of tagging objects with strings, which falls back on a simple full-text search, users could tag something with an actual representation.

Another interesting problem to investigate is whether the *tick-tag* awareness can also be used in query optimization at compile-time when determining a query execution plan, as well as at runtime (similar to *punctuations* [105]) to adapt the query execution strategy based on the observed streaming *tick-tags*. Clearly, not all punctuations are useful to a particular query, and it would be useful to make a determination of when they are. That is, we would like to answer the question “Can stream query  $Q$  benefit from a particular set of punctuations?” To that end, we first define punctuation schemes to specify the collection of punctuations that will be presented to a query on a particular data stream. We show how both punctuations and query operators induce groupings over the items in the domain of the input(s). We show that a query benefits from an input punctuation scheme (in terms of being able to produce a given output scheme), if each set in the groupings induced by the operators of the query is covered by a finite number of punctuations in the scheme—a kind of compactness.

Finally, real-time tag mining and tag classification can be of interest to many stream-based applications, e.g., real-time auction monitoring, social networking, and scientific monitoring. This problem refers to extending the support for real-time mining and classification to streaming tags.

### 6.2.3 Query Mesh Extensions

While so far, this thesis has focused on the core issues of the query mesh optimization and execution, many other topics could potentially be explored in this context.

One interesting direction is to employ learning methods towards different problems inside data stream engine. One such problem in continuous environments is resource limitations problem. Currently there is an underlying assumption that a query mesh produces exact answers under constrained resources. Since query processing eventually may “fail” due to finite resources, alternate solutions may ultimately need to be employed. Here, to reduce the volume of data processed under duress, learning methods may be employed towards adaptive load shedding, results approximation, and disk-spilling. This direction would also help in expanding the understanding of the power and limitations of machine learning techniques in the “guts” of a database engine.

A very important problem is shared multi-query processing using query mesh paradigm. Here, methods for sharing execution routes and classification would need to be investigated not only among subsets of tuples for one query but also among different query meshes for multiple queries executing on the system.

In the long term, it is essential to expand the query mesh scope to consider processing in the context of large-scale distributed environments. Here, issues like resource discovery, business process coordination and resource negotiation must be tackled. Resilience under failures of the network (unreliable communication), servers (fault tolerance), or components of a server (recovery) are critical services, which I would like to explore in my future work.

In both static and streaming databases, a query plan operator typically maintains a *single* queue (see Figure 6.1(a)). Such setup has several disadvantages in the context of multi-route execution: (1) only a single type of scheduling policy can be used by



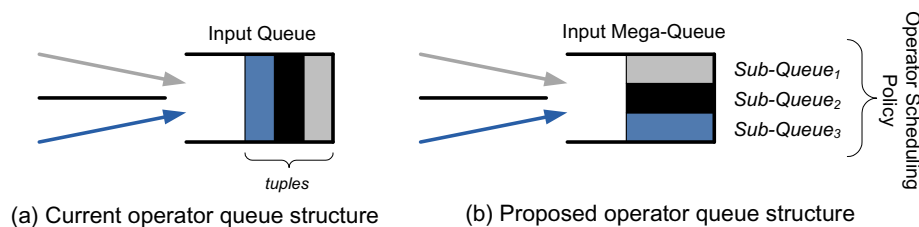


Figure 6.1. Operator queue management.

an operator to dequeue tuples, namely First-In-First-Out (FIFO), and (2) operator processing can be biased towards “bursty” inputs.

Another direction worth examining is the so-called *horizontal queue partitioning* scheme (see Figure 6.1(b)), where the operator input queue is partitioned into multiple sub-queues. Based on horizontally partitioned queues, operators can perform queue management for further improvement for query mesh-based execution. With horizontal queue partitioning approach, there are many possibilities for scheduling tuples for execution by the operators, e.g., FIFO, Priority Queuing, Fair Queuing, Round Robin Weighted Fair Queuing, etc [63]. Other advantages of the proposed operator queue management scheme include: (1) operator can control average queue size, (2) bursts can be absorbed without dropping tuples, (3) operators can prevent bias against “bursty” inputs, and (4) operators can possibly “punish” bursty flows.

The idea of *operator-assisted monitoring* approach, inspired by the resource management approach in *ATM networks* [63] (see Figure 6.2(a)) could potentially also be beneficial in the context of query mesh. The performance metadata, denoted *resource management (RM)* cells, are injected into the network by the intermediate switches and routers and are used to convey network status (e.g., available bandwidth, congestion levels) to the source and destination systems. In the query mesh context, the main idea is for operators to attach their status information, e.g., number of tuples in the queue, current processing rate, etc., to the performance metadata tuples (*p-tokens*) streaming together with the data, so that other operators (e.g., the operators next in route) can become aware of this performance information and possibly exploit

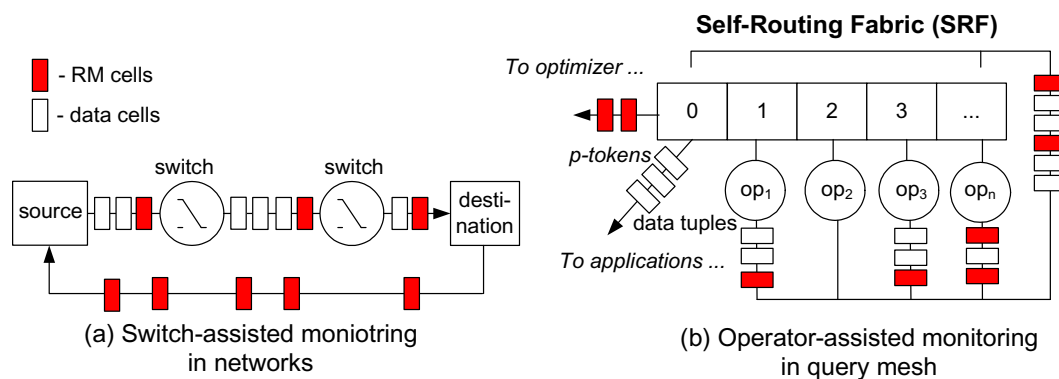


Figure 6.2. Switch and operator-assisted tuning.

it by adapting the routes of the future *rusters* that may contain those bottlenecked operators.

The attractiveness of such operator-assisted performance and state information “diffusion” is in the fact that in a sense it comes for “free”. Since the tuple *rusters* may still have to be routed to other operators, a small “performance signal” with a timestamp can be interleaved with streaming tuples, and then the operators that are left in the route can become “aware” of the current situation at the other operator(s). They can use this information (within a time window) to possibly “detour” some *rusters* if the immediate operators are “overloaded”. Furthermore, these runtime performance metadata can be streamed back to the optimizer to determine how the overall query mesh (not just the runtime routes) can be adapted.

Finally, while our focus in this thesis is on applying diversity-aware query processing in a DSMS, the query mesh model can also be useful in conventional DBMSs, addressing the issue that optimizers sometimes pick plans that perform poorly compared to the actual best plan.

#### 6.2.4 Generalized Punctuation (GPUNCT)

One potentially very effective research direction to explore is the idea of a *generalized punctuation* in DSMSs. To make the idea more intuitive to the reader, consider

the concept of a *Generalized Search Tree (GiST)* [233] in Database Management Systems, which is an index structure supporting an extensible set of queries and data types. The GiST is an extensible data structure, which allows users to develop indices over any kind of data, supporting any lookup over that data. It unifies a number of popular search trees in *one* data structure (the list includes R-trees, B+-trees, hB-trees, TV-trees, Ch-Trees, partial sum trees, ranked B+-trees, and many others), eliminating the need to build multiple search trees for handling diverse applications.

The similar in spirit idea could be extended to the concept of punctuations in Data Stream Management Systems. We refer to this notion – a *generalized punctuation* or *GPUNCT* for short. In a single data structure, the GPUNCT can provide various punctuation logics required by a DSMS, thereby unifying disparate punctuation-based mechanisms. GPUNCT can be used to easily implement a range of well-known punctuation mechanisms, including as sub-stream delimiters inside data streams [98], security metadata [38], feedback mechanisms [103], routing lineage [27], and many others; it can also allow for easy development of specialized metadata for new data types or queries. GPUNCT, like GiST, would represent an example of software extensibility in the context of DSMSs. It will enable the smooth evolution of DSMS towards supporting new punctuation-based algorithms. This would allow authors of new punctuation-based algorithms to focus on implementing the novel features of the new punctuation type – for example, the way in which subsets of the data should be described for search – without becoming experts in DSMS internals.

## LIST OF REFERENCES

## LIST OF REFERENCES

- [1] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, NJ, USA, 2001.
- [2] Lukasz Golab and Tamer Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.
- [3] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams – A new class of data management applications. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 215–226, 2002.
- [4] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.
- [5] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 647–651, 2003.
- [6] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.
- [7] 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 *Proceedings of the International Conference on Data Engineering (ICDE)*, page 851, 2004.
- [8] Elke A. Rundensteiner, Luping Ding, Timothy M. Sutherland, Yali Zhu, Bradford Pielech, and Nishant Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1353–1356, 2004.
- [9] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 379–390, 2000.

- [10] StreamBase. <http://www.streambase.com/>.
- [11] Coral8. <http://www.coral8.com/>.
- [12] Truviso. <http://truviso.com/>.
- [13] Progress Aparma. <http://www.progress.com/apama/index.asp>.
- [14] Andrew Witkowski, Srikanth Bellamkonda, Hua-Gang Li, Vince Liang, Lei Sheng, Wayne Smith, Sankar Subramanian, James Terry, and Tsae-Feng Yu. Continuous queries in Oracle. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1173–1184, 2007.
- [15] My Heart. <http://www.hitech-projects.com/euprojects/myheart/>.
- [16] PIPS. <http://www.pips.eu.org/>.
- [17] Proactive Health. <http://www.intel.com/research/prohealth/>.
- [18] Bright Kite. <http://brightkite.com/>.
- [19] The Carbon Project. <http://www.thecarbonproject.com/>.
- [20] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems (TODS)*, 9(2):163–186, 1984.
- [21] Daniel Lewis. What is Web 2.0? *Crossroads*, 13(1):3–3, 2006.
- [22] Hak Lae Kim, John G. Breslin, Sung-Kwon Yang, and Hong-Gee Kim. Social semantic cloud of tag: Semantic model for social tagging. In *Proceedings of the Agent and Multi-Agent Systems: Technologies and Applications (KES-AMSTA)*, pages 83–92, 2008.
- [23] Liming Chen and Craig Roberts. Semantic tagging for large-scale content management. In *Proceedings of the Web Intelligence*, pages 478–481, 2007.
- [24] Pirjo Näkki, Sari Vainikainen, and Asta Bäck. Experiences of semantic tagging with Tilkut. In *Proceedings of the International Conference on Entertainment and Media in the Ubiquitous Era (Mindtrek)*, pages 167–171, 2008.
- [25] Simone Braun, Valentin Zacharias, and Hans-Jörg Happel. Social semantic bookmarking. In *Proceedings of the Practical Aspects of Knowledge Management (PAKM)*, pages 62–73, 2008.
- [26] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, New York, NY, USA, 2000.
- [27] Rimma V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. Self-tuning query mesh for adaptive multi-route query processing. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 803–814, 2009.
- [28] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 757–768, 2005.

- [29] Neoklis Polyzotis. Selectivity-based partitioning: A divide-and-union paradigm for effective query optimization. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 720–727, 2005.
- [30] Microsoft SQL Server. <http://www.microsoft.com/sql/default.mspx>.
- [31] DB2. <http://www.ibm.com/software/data/db2/>.
- [32] Oracle. <http://www.oracle.com/index.html>.
- [33] Amol Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49, 2004.
- [34] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 261–272. ACM, 2000.
- [35] W. Lindner and J. Meier. Towards a secure data stream management system. In *Proceedings of the Trends in Enterprise Application Architecture (TEAA)*, pages 114–128, 2005.
- [36] Wolfgang Lindner and Jörg Meier. Securing the borealis data stream engine. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 137–147, 2006.
- [37] Barbara Carminati, Elena Ferrari, and Kian Lee Tan. Enforcing access control over data streams. In *Proceedings of the Symposium on Access Control Models and Technologies (SACMAT)*, pages 21–30, 2007.
- [38] Rimma V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. A security punctuation framework for enforcing access control on streaming data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 406–415, 2008.
- [39] Rimma V. Nehme, Hyo-Sang Lim, and Elisa Bertino. Fence: Continuous access control enforcement in dynamic data stream environments. Submitted to the International Conference on Very Large Data Bases (VLDB), 2009.
- [40] Rimma V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. Tagging stream data for rich real-time services. Submitted to the International Conference on Very Large Data Bases (VLDB), 2009.
- [41] Rimma V. Nehme, Karen E. Works, Elke A. Rundensteiner, and Elisa Bertino. Query mesh: Multi-route query processing technology. Submitted to the International Conference on Very Large Data Bases (VLDB), 2009.
- [42] Rimma V. Nehme, Hyo-Sang Lim, Elisa Bertino, and Elke A. Rundensteiner. Streamshield: A stream-centric approach towards security and privacy in data stream environments. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2009.
- [43] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The stanford stream data manager. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, page 665, 2003.

- [44] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the Conference on Symposium on Networked Systems Design and Implementation*, pages 197–210, 2004.
- [45] The PostgreSQL object relational database management system. <http://www.postgresql.org>, 2009.
- [46] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, 2003.
- [47] Thanana M. Ghanem. Supporting predicate-window queries in data stream management systems. In *Proceedings of the International Conference on Data Engineering Workshops*, page 139, 2006.
- [48] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.
- [49] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 53–64, 2000.
- [50] Yanlei Diao and Michael J. Franklin. Query processing for high-volume xml message brokering. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 261–272, 2003.
- [51] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring xml data on the web. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 437–448, 2001.
- [52] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 469–478, 1991.
- [53] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. Continuous queries over append-only databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 321–330, 1992.
- [54] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 11(4):610–628, 1999.
- [55] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
- [56] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, San Francisco, CA, USA, 1996.



- [57] Shivnath Babu. *Adaptive Query Processing in Data Stream Management Systems*. PhD thesis, Stanford University, 2005.
- [58] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 407–418, 2004.
- [59] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 168–177, 1991.
- [60] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 128–137, 1986.
- [61] Arun N. Swami and Balakrishna R. Iyer. A polynomial time algorithm for optimizing join queries. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 345–354, 1993.
- [62] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 285–296, 2003.
- [63] James F. Kurose and Keith Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley, Boston, MA, USA, 2002.
- [64] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 948–959, 2004.
- [65] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 353–365, 2003.
- [66] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 209–218, 1993.
- [67] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.
- [68] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to source properties in processing data integration queries. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 395–406, 2004.
- [69] Paul De Bra and Jan Paredaens. Horizontal decompositions and their impact on query solving. *SIGMOD Record*, 13(1):46–50, 1982.

- [70] Amol Deshpande, Carlos Guestrin, Wei Hong, and Samuel Madden. Exploiting correlated attributes in acquisitional query processing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 143–154, 2005.
- [71] Chung-Min Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 161–172, 1994.
- [72] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. Adaptively reordering joins during query execution. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 26–35, 2007.
- [73] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 19–28, 2001.
- [74] Volker Markl and Guy Lohman. Learning table access cardinalities with LEO. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 613–613, 2002.
- [75] Ning Zhang 0002, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. Statistical learning techniques for costing xml queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 289–300, 2005.
- [76] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
- [77] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 119–130, 2005.
- [78] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. Proactive re-optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 107–118, 2005.
- [79] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust query processing through progressive optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 659–670, 2004.
- [80] Efstratios Viglas. *Novel query optimization and evaluation techniques*. PhD thesis, University of Wisconsin – Madison, 2003.
- [81] Francis C. Chu, Joseph Y. Halpern, and Johannes Gehrke. Least expected cost query optimization: What can we expect? In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 293–302, 2002.
- [82] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. *VLDB Journal*, 6(2):132–151, 1997.
- [83] Norbert Fuhr. A probabilistic framework for vague queries and imprecise information in db. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 696–707, 1990.

- [84] G. Grahne. *Problem of Incomplete Information in Relational Databases*. Springer-Verlag, New York, NY, USA, 1991.
- [85] Lyublena Antova, Christoph Koch, and Dan Olteanu. Query language support for incomplete information in the maybms system. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1422–1425, 2007.
- [86] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *VLDB Journal*, 17(2):243–264, 2008.
- [87] Laks V. S. Lakshmanan, Nicola Leone, Robert B. Ross, and V. S. Subrahmanian. Probview: A flexible probabilistic database system. *ACM Transactions on Database Systems (TODS)*, 22(3):419–469, 1997.
- [88] Jihad Boulos, Nilesh N. Dalvi, Bhushan Mandhani, Shobhit Mathur, Christopher Ré, and Dan Suciu. MYSTIQ: A system for finding more answers by using probabilities. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 891–893, 2005.
- [89] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 262–276, 2005.
- [90] Shivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 238–249, 2005.
- [91] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 263–274, 2002.
- [92] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 299–310, 1999.
- [93] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 106–117, 1998.
- [94] Zachary George Ives. *Efficient query processing for data integration*. PhD thesis, University of Washington, 2002.
- [95] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 130–141, 1998.
- [96] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 49–60, 2002.

- [97] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 333–344, 2003.
- [98] Peter A. Tucker, David Maier, and Tim Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Engineering Bulletin*, 26(1):33–40, 2003.
- [99] Luping Ding, Nishant Mehta, Elke A. Rundensteiner, and George T. Heineman. Joining punctuated streams. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 587–604, 2004.
- [100] Luping Ding and Elke A. Rundensteiner. Evaluating window joins over punctuated streams. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 98–107, 2004.
- [101] Leonidas Fegaras, David Levine, Sujoe Bose, and Vamsi Chaluvadi. Query processing of streamed xml data. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 126–133, 2002.
- [102] Hua-Gang Li, Songting Chen, Junichi Tatemura, Divyakant Agrawal, K. Selçuk Candan, and Wang-Pin Hsiung. Safety guarantee of continuous join queries over punctuated data streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 19–30, 2006.
- [103] Rafael J. Fernandez-Moctezuma, Kristin A. Tufte, and Jin Li. Inter-operator feedback in data stream management. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2009.
- [104] Peter Tucker. *Punctuated data streams*. PhD thesis, OGI School of Science & Engineering at Oregon Health and Science University, 2005.
- [105] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3):555–568, 2003.
- [106] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A heartbeat mechanism and its application in Gigascope. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1079–1088, 2005.
- [107] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. On demand classification of data streams. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 503–508, 2004.
- [108] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O’Callaghan. Clustering data streams. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 359–366, 2000.
- [109] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 97–106, 2001.
- [110] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 71–80, 2000.

- [111] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, and Rui Zhang. Approximate nn queries on streams with guaranteed error/performance bounds. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 804–815, 2004.
- [112] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic udafs at streaming speeds. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 35–46, 2004.
- [113] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 464–475, 2003.
- [114] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Transactions in Mathematics Software*, 11(1):37–57, 1985.
- [115] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 635–644, 2002.
- [116] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.
- [117] Moses Charikar, Liadan O’Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, pages 30–39, 2003.
- [118] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O’Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3):515–528, 2003.
- [119] João Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 523–528, 2003.
- [120] Graham Cormode and S. Muthukrishnan. What’s hot and what’s not: Tracking most frequent items dynamically. *ACM Transactions on Database Systems (TODS)*, 30(1):249–278, 2005.
- [121] Pierre-Alain Laur, Richard Nock, Jean-Emile Symphor, and Pascal Poncelet. Mining evolving data streams for frequent patterns. *Pattern Recognition*, 40(2):492–503, 2007.
- [122] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah, and Jianyong Wang. Multi-dimensional regression analysis of time-series data streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 323–334, 2002.
- [123] Valery Guralnik and Jaideep Srivastava. Event detection from time series data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 33–42, 1999.

- [124] Johannes Gehrke, Venkatesh Ganti, Raghu Ramakrishnan, and Wei-Yin Loh. Boat – optimistic decision tree construction. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 169–180, 1999.
- [125] Johannes Gehrke, Raghu Ramakrishnan, and Venkatesh Ganti. Rainforest – A framework for fast decision tree construction of large datasets. *Data Mining Knowledge Discovery*, 4(2-3):127–162, 2000.
- [126] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 18–32, 1996.
- [127] John C. Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 544–555, 1996.
- [128] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *SIGMOD Record*, 34(2):18–26, 2005.
- [129] Ralf Klinkenberg. Learning drifting concepts: Example selection vs. example weighting. *Intelligent Data Analysis*, 8(3):281–300, 2004.
- [130] Miroslav Kubat and Gerhard Widmer. Adapting to drift in continuous domains (Extended Abstract). In *Proceedings of the European Conference on Machine Learning (ECML)*, pages 307–310, 1995.
- [131] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(1):69–101, 1996.
- [132] Erich Schikuta and Paul Glantschnig. An object-relational neural network database type. In *Artificial Intelligence and Applications*, pages 31–37, 2007.
- [133] José Cascalho and Helder Coelho. Tuning agents’ behaviours using embedded attributes. In *Artificial Intelligence and Applications*, pages 157–162, 2007.
- [134] Stepas Janusonis and Liudas Leonas. Future trends in self-formation. In *Artificial Intelligence and Applications*, pages 627–632, 2005.
- [135] David Gilbert and Christopher J. Hogger. Logic for representing and implementing knowledge about system behaviour. In *Advanced Topics in Artificial Intelligence*, pages 42–49, 1992.
- [136] Jirí Lazanský. Practical applications of planning tasks. In *Advanced Topics in Artificial Intelligence*, pages 238–244, 1992.
- [137] Menno Heeren. Ant system for solving reactive scheduling problems in value added chains. In *Artificial Intelligence and Applications*, pages 6–11, 2005.
- [138] Ivan Stajduhar and Ivan Bratko. Likelihood based classification in bayesian networks. In *Artificial Intelligence and Applications*, pages 367–372, 2007.
- [139] Richard J. Povinelli. Book review: Foundations of genetic programming. *Genetic Programming and Evolvable Machines*, 5(3):319–320, 2004.
- [140] Durga Shrestha. Machine learning approaches for estimation of prediction interval for the model output. *Neural Networks*, 19(2):225–235, 2006.

- [141] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, NY, USA.
- [142] Ilyes Jenhani, Nahla Ben Amor, and Zied Elouedi. Decision trees as possibilistic classifiers. *International Journal of Approximate Reasoning*, 48(3):784–807, 2008.
- [143] Siegfried Nijssen. Bayes optimal classification for decision trees. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 696–703, 2008.
- [144] Nick Street and YongSeog Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 377–382, 2001.
- [145] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 143–154, 2002.
- [146] Carlos Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. SPL: An access control language for security policies and complex constraints. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2001.
- [147] Piero A. Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):1–35, 2002.
- [148] Dominik Raub and Rainer Steinwandt. An algebra for enterprise privacy policies closed under composition and conjunction. In *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)*, pages 130–144, 2006.
- [149] Elisa Bertino, Piero A. Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, 2001.
- [150] Maria Luisa Damiani, Elisa Bertino, Barbara Catania, and Paolo Perlasca. GEO-RBAC: A spatially aware RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 10(1), 2007.
- [151] Jaehong Park and Ravi Sandhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the Symposium on Access Control Models and Technologies (SACMAT)*, pages 57–64, 2002.
- [152] Jaehong Park and Ravi Sandhu. The UCON-ABC usage control model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):128–174, 2004.
- [153] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 555–566, 2007.
- [154] Surajit Chaudhuri, Tanmoy Dutta, and S. Sudarshan. Fine grained authorization through predicated grants. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1174–1183, 2007.

- [155] Govind Kabra, Ravishankar Ramamurthy, and S. Sudarshan. Redundancy and information leakage in fine-grained access control. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 133–144, 2006.
- [156] Arun Kumar, Neeran M. Karnik, and Girish Chaffle. Context sensitivity in role-based access control. *Operating Systems Review*, 36(3):53–66, 2002.
- [157] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. *VLDB Journal*, 14(4):373–396, 2005.
- [158] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. Mondrian: Annotating and querying databases through colors and blocks. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 82, 2006.
- [159] Mohamed Y. Eltabakh, Mourad Ouzzani, and Walid G. Aref. bdbms – A database management system for biological data. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 196–206, 2007.
- [160] Divesh Srivastava and Yannis Velegrakis. Intensional associations between data and metadata. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 401–412, 2007.
- [161] Flickr. <http://flickr.com/>, 2009.
- [162] Delicious. <http://del.icio.us/>.
- [163] Technorati. <http://www.technorati.com/>.
- [164] Cameron Marlow, Mor Naaman, Danah Boyd, and Marc Davis. HT06, tagging paper, taxonomy, Flickr, academic article, to read. In *Proceedings of the Conference on Hypertext and Hypermedia (Hypertext)*, pages 31–40, 2006.
- [165] Ed H. Chi and Todd Mytkowicz. Understanding the efficiency of social tagging systems using information theory. In *Proceedings of the Conference on Hypertext and Hypermedia (Hypertext)*, pages 81–88, 2008.
- [166] Harry Halpin, Valentin Robu, and Hana Shepherd. The complex dynamics of collaborative tagging. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 211–220, 2007.
- [167] Christoph Schmitz, Andreas Hotho, Robert Jschke, and Gerd Stumme. Mining association rules in folksonomies. In *Proceedings of the International Federation of Classification Societies (IFCS)*, pages 261–270, Heidelberg, 2006.
- [168] Paul Heymann, Daniel Ramage, and Hector Garcia-Molina. Social tag prediction. In *Proceedings of the International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 531–538, 2008.
- [169] Steffen Oldenburg, Martin Garbe, and Clemens H. Cap. Similarity cross-analysis of tag/co-tag spaces in social classification systems. In *Proceeding of the ACM Workshop on Search in Social Media (SSM)*, pages 11–18, 2008.
- [170] Dan Olteanu, Tim Furche, and François Bry. An efficient single-pass query evaluator for XML data streams. In *Proceedings of the Symposium on Applied computing (SAC)*, pages 627–631, 2004.



- [171] Feng Peng and Sudarshan S. Chawathe. XPath queries on streaming data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 431–442, 2003.
- [172] Xiaogang Li and Gagan Agrawal. Efficient evaluation of XQuery over streaming data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 265–276, 2005.
- [173] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 407–418, 2006.
- [174] Mohamed H. Ali, Walid G. Aref, Raja Bose, Ahmed K. Elmagarmid, Abdel-salam Helal, Ibrahim Kamel, and Mohamed F. Mokbel. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1295–1298, 2005.
- [175] David F. Ferraiolo, Ravi S. Sandhu, Serban I. Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.
- [176] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.
- [177] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. An extended authorization model for relational databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 9(1):85–101, 1997.
- [178] Duminda Wijesekera and Sushil Jajodia. A propositional policy algebra for access control. *ACM Transactions on Information and System Security (TISSEC)*, 6(2):286–325, 2003.
- [179] Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. Secure xml querying with security views. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 587–598, 2004.
- [180] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 551–562, 2004.
- [181] Thomas Brinkhoff. Generating network-based moving objects. In *Proceedings of the Statistical and Scientific Database Management (SSDBM)*, page 253, 2000.
- [182] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: A new architecture for high-performance stream systems. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1(1):274–288, 2008.

- [183] Luping Ding and Elke A. Rundensteiner. Evaluating window joins over punctuated streams. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 98–107, 2004.
- [184] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. FluXQuery: An optimizing XQuery processor for streaming xml data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1309–1312, 2004.
- [185] Sujoe Bose and Leonidas Fegaras. Data stream management for historical xml data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 239–250, 2004.
- [186] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic query optimization for XQuery over xml streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 277–288, 2005.
- [187] Bettina Berendt and Christoph Hanser. Tags are not metadata, but “just more content” – to some people. In *Proceedings of the International Conference on Weblogs and Social Media (ICWSM)*, 2007.
- [188] Ronen Feldman, Binyamin Rosenfeld, Moshe Fresko, and Brian D. Davison. Hybrid semantic tagging for information extraction. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 1022–1023, 2005.
- [189] Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming-Wei Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10(4):407–437, 2005.
- [190] Morgan Ames and Mor Naaman. Why we tag: Motivations for annotation in mobile and online media. In *Proceedings of the Conference on Computer Human Interaction (CHI)*, pages 971–980, 2007.
- [191] Scott A. Golder and Bernardo A. Huberman. The structure of collaborative tagging systems. *Proceedings of the Computing Research Repository (CoRR)*, 2005.
- [192] Nathan Eagle and Alex Pentland. Reality mining: Sensing complex social systems. *Personal and Ubiquitous Computing (PUC)*, 10(4):255–268, 2006.
- [193] Robert Jäschke, Leandro Balby Marinho, Andreas Hotho, Lars Schmidt-Thieme, and Gerd Stumme. Tag recommendations in social bookmarking systems. *Artificial Intelligence Communications*, 21(4):231–247, 2008.
- [194] EBay. <http://www.ebay.com/>.
- [195] Wen-Liang Hung and Miin-Shen Yang. Similarity measures between type-2 fuzzy sets. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 12(6):827–842, 2004.
- [196] Many Eyes DS. <http://manyeyes.alphaworks.ibm.com/>.
- [197] Google Latitude. <http://www.google.com/latitude/intro.html>.
- [198] TradingMarkets. <http://www.tradingmarkets.com/>.

- [199] Trending. <http://www.trending123.com/>.
- [200] Richard J. Lipton, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 116(1&2):195–226, 1993.
- [201] David J. Hand, Padhraic Smyth, and Heikki Mannila. *Principles of data mining*. MIT Press, Cambridge, MA, USA, 2001.
- [202] Dymitr Ruta. Dynamic data condensation for classification. In *Proceedings of the Artificial Intelligence and Soft Computing (ICAISC)*, pages 672–681, 2006.
- [203] B.V. Dasarathy. Minimal consistent set identification for optimal NN decision systems. *IEEE Transactions on Systems, Man, and Cybernetics (TSMC)*, 24(3):511–517, 1994.
- [204] Takekazu Kato and Toshikazu Wada. Direct condensing: An efficient voronoi condensing algorithm for nearest neighbor classifiers. In *Proceedings of the International Conference on Pattern Recognition (ICPR)*, pages 474–477, 2004.
- [205] C. H. Chen and Adam Józwiak. A sample set condensation algorithm for the class sensitive artificial neural network. *Pattern Recognition*, 17(8):819–823, 1996.
- [206] José Salvador Sánchez. High training set size reduction by space partitioning and prototype abstraction. *Pattern Recognition*, 37(7):1561–1564, 2004.
- [207] Maria Teresa Lozano. *Data Reduction Techniques in Classification Processes*. PhD thesis, Jaume University, 2007.
- [208] Martin Klazar. Bell numbers, their relatives, and algebraic differential equations. *Journal of Combinatorial Theory Series*, 102(1):63–87, 2003.
- [209] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 34–43, 1998.
- [210] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 23–34, 1979.
- [211] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 267–276, 1993.
- [212] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience, New York, NY, USA, 1990.
- [213] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 312–321, 1990.
- [214] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems (TODS)*, 29:162–194, 2004.

- [215] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.
- [216] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 37–48, 2002.
- [217] Kajal T. Claypool and Mark Claypool. Teddies: Trained eddies for reactive stream processing. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 220–234, 2008.
- [218] Kien A. Hua, Yu lung Lo, and Honesty C. Young. Considering data skew factor in multi-way join query optimization for parallel execution. *VLDB Journal*, 2(3):303–330, 1993.
- [219] Rimma V. Nehme, Elke A. Rundensteiner, Karen E. Works, and Elisa Bertino. Query mesh: An efficient multi-route approach to query optimization. Technical Report CSD TR #08-009, Department of Computer Science, Purdue University, April 2008.
- [220] Gerhard Widmer and Miroslav Kubat. Effective learning in dynamic environments by explicit context tracking. In *Proceedings of the European Conference on Machine Learning (ECML)*, pages 227–243, 1993.
- [221] Alexey Tsymbal. The problem of concept drift: Definitions and related work. Technical Report TCD-CS-2004-15, Department of Computer Science, University of Dublin, Trinity College, 2004.
- [222] Surajit Chaudhuri, Arnd Christian König, and Vivek R. Narasayya. Sqlcm: A continuous monitoring framework for relational database engines. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 473–485, 2004.
- [223] Donald Michie, D. J. Spiegelhalter, C. C. Taylor, and John Campbell, editors. *Machine learning, neural and statistical classification*. Ellis Horwood, Upper Saddle River, NJ, USA, 1994.
- [224] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. Wiley-Interscience, New York, NY, USA, 1991.
- [225] Haixun Wang and Jian Pei. A random method for quantifying changing distributions in data streams. In *Proceedings of the Principles of Data Mining and Knowledge Discovery (PKDD)*, pages 684–691, 2005.
- [226] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 550–559, 2004.
- [227] I. M. Chakravarti, R. G. Laha, and J. Roy. *Handbook of Methods of Applied Statistics*, volume I. John Wiley and Sons, New York, NY, USA, 1967.
- [228] A. Kumaran, Pavan K. Chowdary, and Jayant R. Haritsa. On pushing multilingual query operators into relational engines. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 98, 2006.

- [229] Ian H. Witten and Eibe Frank. Data mining: Practical machine learning tools and techniques with Java implementations. *In SIGMOD Record*, 31(1):76–77, 2002.
- [230] Philippe Smets. The transferable belief model. *Artificial Intelligence*, 66(2):191–234, 1994.
- [231] Philippe Smets. The application of the transferable belief model to diagnostic problems. *International journal of intelligent systems*, 13:127–157, 1998.
- [232] Matthew A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley, Boston, MA, USA, 2002.
- [233] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 562–573, 1995.

VITA

## VITA

Rimma V. Nehme was born in the small town of Baranovitchi, USSR (currently Belarus). She spent her childhood travelling all over the Soviet Union and East Germany because of her father's job in the military. In 1997, during her senior year in high-school, Rimma came to the United States. As a result, Rimma earned two high school diplomas in 1998, in the US and in Belarus.

Being completely fascinated with America, Rimma decided to pursue her undergraduate studies at Hillsdale College, MI in August 1998. In 2001, after three years, Rimma was granted her B.S. degree with the *Magna Cum Laude* (high honor) in Computational Mathematics. Upon graduation, Rimma joined the EMC Corporation as a software engineer in Hopkinton, MA working on the online management of enterprise storage devices. After spending several years in the industry, Rimma decided to pursue graduate studies part-time in Worcester Polytechnic Institute (WPI) in Worcester, MA and received the masters degree in Computer Science in 2005. While at WPI, Rimma shaped her research attitude and developed significant interest in database systems. Rimma then came to Purdue University to pursue her Ph.D. degree in the fall of 2005. Rimma published several papers in core database technology. In 2006, 2007 and 2008, Rimma spent three summers at Microsoft Research in Redmond, WA, one of the top research labs in the field world-wide. During her summer internships, Rimma interacted with many world-class researchers and further built her database systems experience. Rimma V. Nehme graduated with a Ph.D. degree in Computer Science from Purdue University in August 2009 and joined the Microsoft Jim Gray Systems Lab in Madison, Wisconsin USA.