ADAPTIVE QUERY PROCESSING IN DATA STREAM MANAGEMENT SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Shivnath Babu
September 2005

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom
(Department of Computer Science, Stanford University) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David J. DeWitt
(Department of Computer Science, University of Wisconsin-Madison)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Rajeev Motwani
(Department of Computer Science, Stanford University)

Approved for the University Committee on Graduate Studies.

Abstract

Many modern applications need to process *data streams* that consist of data elements generated in a continuous unbounded fashion. Examples of such applications include network monitoring, financial monitoring over stock tickers, sensor processing for environmental monitoring or inventory tracking, telecommunications fraud detection, and others. These applications have spurred interest in a new class of systems called *Data Stream Management Systems (DSMSs)*. DSMSs fulfill the data management needs of these applications arising from the continuous, unbounded, rapid, and time-varying nature of data streams. Conventional database management systems, which are designed to process queries over finite stored datasets, are ill-equipped to satisfy these new needs.

DSMSs enable users and applications to pose queries over data streams. These queries tend to be long-running since data arrives continuously, and are called *continuous queries*. A fundamental challenge faced by DSMSs is that stream characteristics (e.g., data distribution, arrival rate) and system conditions (e.g., query load, memory availability) may vary significantly over the lifetime of a continuous query. When stream characteristics or system conditions change, a query execution plan that was most efficient to process a continuous query before the change, may become very inefficient all of a sudden. Consequently, it is important for a DSMS to support *adaptive query processing*: The DSMS must be prepared to change the execution plan for a continuous query while the query is running, based on how stream characteristics and system conditions change. Without adaptivity, plan performance may drop drastically over time.

This thesis addresses the problem of processing continuous queries in a DSMS when stream characteristics and system conditions may vary unpredictably over time. We present a generic framework, called *StreaMon*, for adaptive query processing in a DSMS. StreaMon has three core components: (i) An *Executor*, which runs the current plan for each query, (ii) a *Profiler*, which collects and maintains statistics about current stream characteristics and system conditions, and (iii) a *Re-optimizer*, which ensures the current plans are the most efficient for current stream characteristics and system conditions. We instantiate the generic StreaMon framework for three distinct combinations of continuous query type and adaptivity need:

1. Adaptive processing of commutative filters over a stream to maximize throughput at all points in time.
2. Adaptive placement of subresult caches in pipelined plans for windowed stream joins to maximize throughput at all points in time.
3. Detecting relaxed constraints automatically in input streams and exploiting these constraints to reduce memory requirements in plans for windowed stream joins.

These three problems were motivated by their applicability to almost all the stream-based applications we studied. In each case we have instantiated the generic components of StreaMon in specific ways to address the particular continuous query type and adaptivity need. For each problem, we provide the definition and motivating examples, develop and analyze adaptive algorithms, and present implementation techniques and experimental results from the *STREAM* general-purpose DSMS prototype developed at Stanford.

Acknowledgments

I am extremely grateful to my Ph.D. adviser, Jennifer Widom, for all the guidance and help she has given me. She has been a wonderful role model to me as a researcher and a teacher. I will consider myself most fortunate if I can bring into my research, teaching, and advising even half the intensity and structure that she brings into hers. All the work presented in this thesis was done in collaboration with Jennifer.

I am grateful to have had the opportunity to work with David DeWitt, Rajeev Motwani, and Kamesh Munagala. David gave me the opportunity to spend a semester at the University of Wisconsin-Madison, and to get exposed to the system-building culture of Madison's Database group. Rajeev and Kamesh have helped me complement my system-oriented research with an understanding of the theoretical issues involved. A large portion of the work presented in this thesis was done in collaboration with Rajeev and Kamesh. I would like to thank David and Rajeev for kindly offering to serve as readers for this thesis.

I have greatly enjoyed the friendship of Kamesh Munagala and Chandra Nair. I would like to thank Arvind Arasu, Pedro Bizarro, Sorav Bansal, Ramesh Chandra, Shankar Ponnekanti, and Utkarsh Srivastava for all the help and encouragement they have given me. I am grateful to all members of Stanford's Infolab for numerous helpful suggestions and actions. In particular, I would like to thank Brian Babcock, Mayank Bawa, Mayur Datar, Prasanna Ganesan, Hector Garcia-Molina, Zoltan Gyongyi, Andy Kacsmar, Marianne Siroker, Qi Su, Qi Sun, and Sarah Weden.

Finally, I would like to thank my mother, father, and brother for their unconditional love and encouragement. I dedicate this thesis to them.

Contents

Abstract **iv**

Acknowledgments **vi**

1 Introduction **1**

1.1 Differences between a DSMS and a DBMS 3

1.2 STREAM DSMS and StreaMon 5

1.3 Motivating Example 6

1.4 Contributions and Outline of Thesis 9

1.5 Related Work 12

1.5.1 Data Stream Management Systems 12

1.5.2 Other Continuous Query Processors 13

2 The STREAM DSMS **15**

2.1 The CQL Continuous Query Language 15

2.1.1 Abstract Semantics 16

2.1.2 Concrete Language 17

2.1.3 Example CQL Queries 19

2.2 Query Plans and Execution 20

2.2.1 Operators 20

2.2.2 Queues 20

2.2.3 Synopses 21

2.2.4 Example Query Plan 22

2.2.5 Query Plan Execution 23

2.3 Performance Issues 25
2.3.1 Synopsis Sharing 25
2.3.2 Operator Scheduling 26
2.4 The STREAM System Interface 29
2.5 Related Work 31
2.6 Conclusion 32

3 Adaptive Query Processing with StreaMon 34
3.1 StreaMon 36
3.2 Adaptive Ordering of Pipelined Filters 38
3.3 Adaptive Caching for Windowed Stream Joins 42
3.4 Adaptive Memory Minimization using k-Constraints 47
3.5 Related Work 50
3.6 Conclusion 51

4 Adaptive Ordering of Pipelined Stream Filters 55
4.1 Introduction 56
4.1.1 Motivating Example 56
4.1.2 Filter Ordering 58
4.1.3 Adaptive Ordering of Filters 59
4.2 Notation for this Chapter 61
4.3 The A-Greedy Adaptive Filter Ordering Algorithm 62
4.3.1 The A-Greedy Profiler 63
4.3.2 The A-Greedy Re-optimizer 65
4.3.3 Convergence Properties 70
4.3.4 Run-time Overhead and Adaptivity 72
4.4 The Sweep Algorithm 73
4.4.1 Convergence Properties, Run-time Overhead, and Adaptivity 74
4.5 The Independent Algorithm 75
4.5.1 Convergence Properties, Run-time Overhead, and Adaptivity 76
4.6 The LocalSwaps Algorithm 77
4.6.1 Convergence Properties, Run-time Overhead, and Adaptivity 77

4.7 Experimental Evaluation 78

4.7.1 Summary of Experimental Results 80

4.7.2 Convergence Experiments 80

4.7.3 Run-time Overhead Experiments 83

4.7.4 Adaptivity Experiments 86

4.8 Summary of A-Greedy and its Variants 90

4.9 Related Work 92

4.10 Conclusion 93

5 Adaptive Processing of Windowed Stream Joins 95

5.1 Stream Joins and MJoins 96

5.1.1 Notation 99

5.2 A-Greedy for MJoins 101

5.2.1 Note on Join Graphs 102

5.2.2 Adaptive Ordering of Star Joins 103

5.2.3 Adaptive Ordering of Acyclic and Cyclic Joins 105

5.3 XJoins 105

5.4 Caches in MJoin Pipelines 109

5.4.1 Implementation of Caches in STREAM 112

5.5 Adaptive Caching 113

5.5.1 Cache Cost Model 114

5.5.2 Processing Windowed Stream Joins Adaptively 115

5.5.3 Estimating Cache Benefits and Costs Online 116

5.5.4 Offline Cache Selection Algorithm 117

5.5.5 Adaptive Cache Selection Algorithm 120

5.5.6 Implementation of Adaptive Caching in STREAM 122

5.6 Adaptive Memory Allocation to Caches 123

5.7 Globally-Consistent Caches 125

5.8 Experimental Evaluation 126

5.8.1 Experimental Setup 126

5.8.2 Summary of Experimental Results 128

5.8.3 Performance of Caching 128

5.8.4 Performance of Plans for Windowed Stream Joins 134

5.8.5 Adaptivity Experiments 137
5.9 Related Work 139
5.10 Conclusion 139

6 Exploiting k-Constraints for Continuous Queries 141

6.1 Introduction 143
6.1.1 Example 143
6.1.2 Challenges in Exploiting Stream Properties 146
6.1.3 Overview of Stream Constraints 147
6.1.4 Using Stream Properties for Reordering Tuples 149
6.1.5 Queries and Execution Overview 151
6.1.6 k-Mon: An Architecture for Exploiting k-Constraints 154
6.2 Preliminaries 156
6.3 Basic Query Execution Algorithm 158
6.3.1 Synopsis Reduction 166
6.3.2 Handling Windows over Streams 168
6.4 Overview of k-Mon's Implementation 170
6.5 Experimental Setup 171
6.6 Referential Integrity Constraints 172
6.6.1 Modified Algorithm to Exploit RIDS(*k*) 173
6.6.2 Implementing RIDS(k) Usage 174
6.6.3 Monitoring RIDS(*k*) 175
6.6.4 Experimental Analysis for RIDS(*k*) 176
6.7 Clustered-Arrival Constraints 179
6.7.1 Modified Algorithm to Exploit CA(*k*) 179
6.7.2 Implementing CA(*k*) Usage 181
6.7.3 Monitoring CA(*k*) 182
6.7.4 Experimental Analysis for CA(*k*) 183
6.8 Ordered-Arrival Constraints 184
6.8.1 Modified Algorithm to Exploit OAP(*k*) 185
6.8.2 Implementing OAP(k) Usage 185
6.8.3 Monitoring OA(*k*) 186

6.8.4 Experimental Analysis for OAP(*k*) 187
6.8.5 Modified Algorithm to Exploit OAC(*k*) 189
6.8.6 Implementing OAC(*k*) Usage 189
6.8.7 Experimental Analysis for OAC(*k*) 190
6.9 Computational Overhead 191
6.10 Constraint Combination 192
6.11 *k*-Constraints in the Linear Road Queries 195
6.12 Related Work 198
6.13 Conclusion 199

7 Future Directions in Adaptive Query Processing 201
7.1 Plan Logging 202
7.2 AQP in Conventional DBMSs 204
7.3 Conclusion 206

Bibliography 208

List of Tables

1.1 Differences between DBMSs and DSMSs 3

2.1 Operators used in STREAM query plans 21

4.1 Default values used in experiments 81

4.2 Run-time percentage breakdown for A-Greedy 84

4.3 Summary of performance of all algorithms 90

5.1 Notation used in this chapter 100

5.2 Relative stream arrival rates and pairwise join selectivities for the eight sample points in Figure 5.21. The stream arrival rates are relative to the rate of stream T 135

6.1 Tuple-processing times (microseconds) for different k 192

6.2 Summary of synopsis reductions 193

6.3 Results for Linear Road queries 197

List of Figures

1.1 Use of a Data Stream Management System for network monitoring 2

1.2 Query execution plan for example network-monitoring query 8

2.1 Data types and operator classes in abstract semantics 17

2.2 A simple query plan illustrating operators, queues, and synopses 22

2.3 A query plan illustrating synopsis sharing 27

2.4 Screenshot of the STREAM visualizer 30

3.1 Components of the generic StreaMon framework 36

3.2 A-Greedy 40

3.3 MJoin-based plan for example network-monitoring query 43

3.4 XJoin-based plans for the example network-monitoring query 44

3.5 A-Caching 45

3.6 k-Mon 49

4.1 Plan to process a pipelined filter query involving filters $F_1, F_2, \ldots, F_n$ over stream S. $f(1), f(2), \ldots, f(n)$ is a permutation of $1, 2, \ldots, n$ 57

4.2 Filters and sequence of input tuples in our example pipelined filter query . . 62

4.3 Profile window for our example pipelined filter query 65

4.4 Matrix view for our example pipelined filter query 67

4.5 Updating V on an insert to the profile window 67

4.6 Detecting a violation of the Greedy Invariant 68

4.7 Correcting a violation of the Greedy Invariant 69

4.8 Violation of the Greedy invariant 70

4.9 View entries maintained by A-Greedy variants 75
4.10 Architecture of the extended version of the `select` operator that uses A-Greedy to order filters 79
4.11 Effect of the number of filters under convergence 81
4.12 Effect of selectivity under convergence 82
4.13 Effect of correlation under convergence for $n = 3$ filters 83
4.14 Effect of correlation under convergence for $n = 8$ filters 83
4.15 Performance of all algorithms 84
4.16 Run-time breakdown for Figure 4.15 85
4.17 Effect of varying filter costs 85
4.18 Run-time breakdown for Figure 4.17 86
4.19 Adaptivity of A-Greedy versus Sweep 87
4.20 Adaptivity of A-Greedy versus LocalSwaps 87
4.21 Adaptivity of A-Greedy versus Independent 88
4.22 Varying the rate of change 88
4.23 Varying filter cost at $x = 10^5$ from Figure 4.22 89
4.24 Varying filter cost at $x = 10^6$ from Figure 4.22 89
4.25 Tradeoff spectrum for adaptive processing of pipelined filters 92

5.1 MJoin for an n-way windowed stream join 98
5.2 Plan and data used in examples 101
5.3 Acyclic, cyclic, and star joins 102
5.4 MJoin for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ 106
5.5 XJoins for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ 107
5.6 Tree of MJoins for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ 107
5.7 Plan with a $R_2 \bowtie R_3$ cache 111
5.8 Cache hierarchy from Example 5.5.1 118
5.9 State transition diagram for candidate caches 121
5.10 Architecture of the `mjoin` operator that uses A-Caching for cache selection 124
5.11 Varying cache hit probability (absolute graph) 129
5.12 Varying cache hit probability (relative graph) 130

5.13 Varying join selectivity (absolute graph) 130
5.14 Varying join selectivity (relative graph) 131
5.15 Varying update to probe ratio (absolute graph) 131
5.16 Varying update to probe ratio (relative graph) 132
5.17 Varying number of joins (absolute graph) 132
5.18 Varying number of joins (relative graph) 133
5.19 Varying join cost (absolute graph) 133
5.20 Varying join cost (relative graph) 134
5.21 Performance of join plans 135
5.22 Adaptivity to changing stream rate 137
5.23 Adaptivity to memory availability 138

6.1 Query execution plan for example windowed equi-join query 145
6.2 Query plan for example windowed join query 152
6.3 k-Mon 155
6.4 Procedure invoked when a tuple s arrives in stream S 163
6.5 Procedure to insert s into $\mathcal{S}(S).Yes$ 164
6.6 Procedure to insert s into $\mathcal{S}(S).No$ 164
6.7 Procedure to insert s into $\mathcal{S}(S).Unknown$ 165
6.8 Directed join graphs used in the examples 166
6.9 Directed join graphs used in the experiments 177
6.10 Memory reduction using RIDS(k) 177
6.11 Monitoring RIDS(*k*) 178
6.12 Memory reduction using CA(k) 183
6.13 Monitoring CA(*k*) 184
6.14 Memory reduction using OAP(k) 188
6.15 Monitoring OAP(*k*) 188
6.16 Memory reduction using OAC(*k*) 191
6.17 Effect of combining CA and OAC 194
6.18 Effect of combining CA and RIDS 195
6.19 *AccSeg* query from Linear Road 196

Chapter 1

Introduction

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* [51]. 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.

Conventional DBMSs are designed to process queries over finite stored datasets. However, many modern applications need to process *data streams* that consist of data elements generated in a continuous unbounded fashion. Examples of such applications include network-monitoring applications processing streams of network packets, financial monitoring over stock-ticker streams, environmental monitoring or inventory tracking over 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. Conventional DBMSs are ill-equipped to fulfill the needs of these applications. Therefore, a new class of systems—*Data Stream Management Systems (DSMSs)* [29, 30, 38, 88]—are being developed by the database research community to satisfy the requirements of stream-based applications. DSMSs enable users and applications to pose queries over data streams. These queries tend to be long-running since data arrives continuously, and are called *continuous queries*.

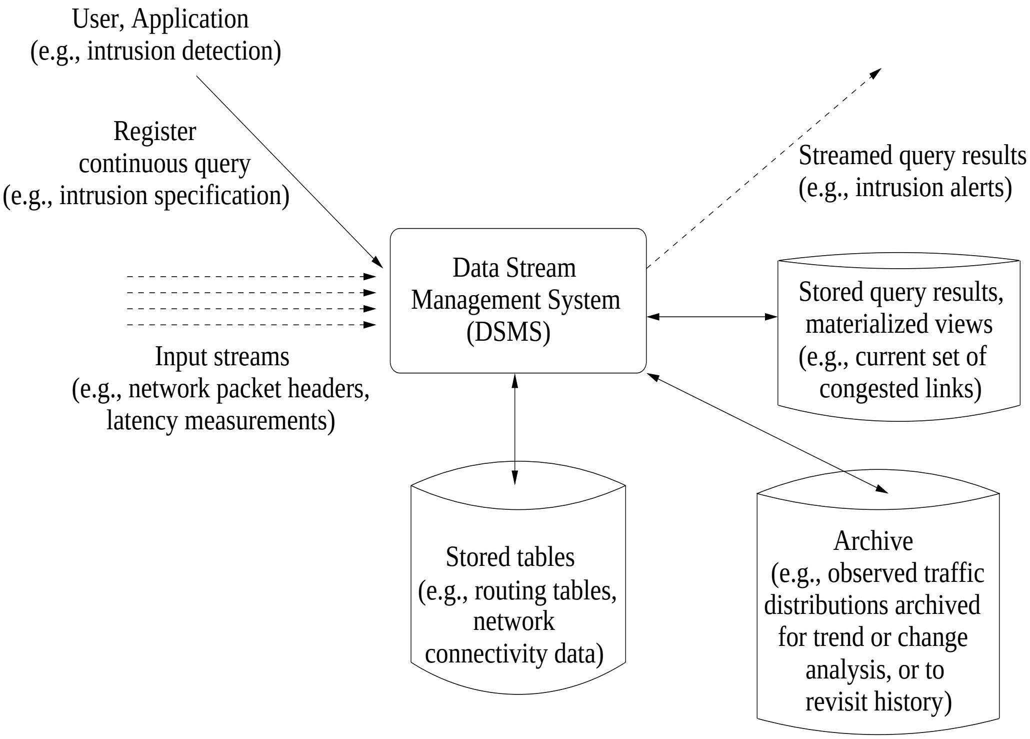


Figure 1.1: Use of a Data Stream Management System for network monitoring

Figure 1.1 shows how a DSMS is used by one of its target applications. The illustrative application used in Figure 1.1 is the monitoring of traffic in the large network of an Internet Service Provider (ISP) [27]. Continuous streams consisting of data such as packet headers and performance measurements are collected by data collectors placed in the network. These streams are sent to the DSMS for processing. In addition to the input streams, the DSMS also maintains conventional stored data, e.g., current routing tables or data about the current connectivity of routers, switches, and links in the network [25]. Network administrators or monitoring applications register continuous queries over the input data streams and other data managed by the DSMS. For example, a registered continuous query may specify a set of conditions that signal a specific type of attack on the network [23, 101].

The continuous queries registered at the DSMS are activated whenever new elements, or *tuples*, arrive in the input data streams, when one of the stored tables is updated, or when time advances. New results may be generated when a query is activated, which are usually

Component	DBMS	DSMS
Data	Finite data sets on disk, along with auxiliary structures like indexes	Continuous, rapid, and time-varying data streams
Query Semantics	One-time queries over data sets that (logically) do not change while a query runs	Continuous queries that produce results as stream tuples arrive, time advances, or stored data changes
Query Execution	*Pull-based*: Queries processed by scan-based or index-based operators that read data from disk	*Push-based*: Queries processed by incremental operators that are scheduled based on tuple arrival
Query Optimization	Query plan chosen once per query using statistics available on the data set, and system characteristics	Adaptive execution plans based on stream and system conditions as query runs

Table 1.1: Differences between DBMSs and DSMSs

output directly to users or applications as continuous data streams. The query result may also be stored and maintained at the DSMS, similar to *materialized views* in a conventional DBMS [59]. For example, the network administrator may want the DSMS to keep track of the set of congested links in the network.

In addition to real-time processing of the stream tuples as they arrive at the DSMS, these tuples or their aggregated summaries may be archived as shown in Figure 1.1. Stream archives serve two purposes. First, archives can be used for detecting changes or trends in the streams over time. For example, the network administrator may want an alert to be raised if the distribution of destination port numbers seen in the network packets at a router changes significantly over time. Second, archives are like data stored in a DBMS, so they can be queried and analyzed in a conventional manner.

1.1 Differences between DSMSs and Conventional DBMSs

DSMSs differ from conventional Database Management Systems (DBMSs) in many important components of data management. These differences are summarized in Table 1.1, and described next.

- **Data:** In conventional relational DBMSs, data is stored on disk as tables of tuples, or *relations*. The tuples in a relation may be organized on disk in specific layout patterns, e.g., tuples with identical values of an attribute may be clustered together for efficient access. Furthermore, auxiliary structures like indexes may be built on the relations to enable efficient retrieval of individual tuples. On the other hand, tuples in a stream arrive continuously at a DSMS, usually as data feeds over the network. The DSMS typically has no control over the arrival order, rate, or data distribution of the input streams. Furthermore, since continuous queries need real-time responses as data arrives at the DSMS, conventional DBMS data loading, storage, and retrieval techniques are not adequate to handle data streams in a DSMS.

- **Query Semantics:** Queries in conventional DBMSs are *one-time queries* posed over the data stored in the DBMS at a point of time. One-time queries may be specified in a conventional declarative query language like SQL [51]. While the data in the DBMS may be updated over time, the output of a one-time query Q_o is the result produced by running Q_o over the snapshot of relations in the DBMS when Q_o was issued. On the other hand, a continuous query Q_c in a DSMS is posed over one or more unbounded streams, possibly with relations too. Q_c is registered at the DSMS by a user or application. Once registered, Q_c runs and generates results continuously until it is deregistered. Continuous queries can often be expressed in conventional languages used to specify materialized views [59] and triggers [117]: The query of interest can be specified as a materialized view, and the updates to the view can be captured and manipulated using triggers. However, continuous queries in stream-based applications routinely need constructs like windowed stream joins, history of updates to a table, nonblocking partial sorts, and others [5, 29]. While some of these constructs can be specified using arbitrary combinations of materialized views and triggers, such circuitous ways of expressing common constructs prevent the system from optimizing queries effectively. In order to enhance system usability and overall performance, it is important to make continuous queries over streams expressible in direct and simple ways. Consequently, new query languages that extend conventional languages and semantics have been proposed for continuous queries, e.g., [5].

- **Query Execution:** *Query plans* for executing one-time queries in DBMSs are based on reading relations from disk using scan-based or index-based access methods, and processing the data in memory [57]. Because of the long-running nature of continuous queries, the uncontrollable and time-varying nature of data streams, and the need for real-time responses in stream applications, new techniques for query execution are necessary in DSMSs. Query plans in DSMSs must process input stream tuples in an online and incremental manner so that new results can be generated in real-time. Query plans must be flexible enough to support fine-grained optimization and scheduling decisions to adapt to current stream and system conditions. Furthermore, DSMSs need to handle stream arrival rates that may exceed the system's capacity to provide exact results for all registered continuous queries. For example, a DSMS may degrade the accuracy of query results using techniques for approximate query answering, to reduce resource requirements during periods of high load [15, 107].

- **Query Optimization:** Conventional DBMSs use a *plan-first execute-next* approach to process a query Q [100]. The query optimizer enumerates candidate query plans for Q from the many different possible plans, each plan having different performance characteristics. The optimizer then estimates the cost of executing each candidate plan (e.g., time to completion) based on the current data and system conditions. The plan with the lowest estimated cost is chosen for executing Q. The chosen plan is then run until all query results are produced. In DSMSs, stream conditions (e.g., data distribution, arrival rate) and system conditions (e.g., query load, memory availability) may vary significantly over the lifetime of a long-running continuous query. Therefore, it is important for a DSMS to support *adaptive* approaches to query processing [10, 21, 64]: The DSMS must be prepared to change the execution plan for a continuous query while the query is running, based on how stream and system conditions change. Without adaptivity, plan performance may drop drastically over time as stream or system conditions change.

1.2 STREAM DSMS and StreaMon

The STREAM (*Stanford Stream Data Manager*) project at Stanford [88] is addressing new challenges in data management and query processing that arise in the context of DSMSs. As part of this project we built a general-purpose prototype relational DSMS, also called

STREAM, that supports a large class of declarative continuous queries over data streams and conventional stored relations. The STREAM prototype targets environments where streams may be rapid, stream and system conditions may vary over time, and system resources may be limited.

STREAM supports an expressive SQL-based declarative language called *CQL* (*Continuous Query Language*) for registering continuous queries over streams and relations [5]. CQL queries are translated into query plans composed of *operators*, *queues*, and *synopses*. These plans are flexible enough to support reorganization and fine-grained scheduling decisions. For improved performance, STREAM shares state and computation within and across query plans when multiple continuous queries are running simultaneously. Due to the long-running nature of continuous queries, STREAM provides tools for administrators and users to monitor and manipulate query plans as they run. Chapter 2 describes the STREAM system in more detail.

To maintain good performance under unpredictable and volatile stream and system conditions, STREAM supports adaptive techniques for processing continuous queries. The overall framework for adaptive query processing in STREAM is called *StreaMon*. StreaMon includes a variety of techniques to handle different types of continuous queries and adaptivity needs:

1. Adaptive ordering of filter operators in pipelined plans over a single stream [18]

2. Adaptive ordering of join operators in multiway stream join plans [18, 19]

3. Placement of *subresult caches* adaptively in multiway stream join plans to minimize recomputation of intermediate results [19]

4. Reducing run-time memory requirements for join and aggregation operators over streams by exploiting stream data and arrival properties [20]

1.3 Motivating Example

As illustrated in Figure 1.1, one application of a DSMS is to support traffic monitoring for a large network such as the backbone of an ISP [27, 38]. We use the following example continuous query from this application to illustrate some of the challenges in processing continuous queries over data streams:

> *Monitor the total traffic from a customer network that went through a specific set of links in the ISP's network within the last 10 minutes.*

A network analyst might pose this query to detect service-level agreement violations, to find opportunities for load balancing, to monitor network health, or for other reasons [14, 46].

Let C denote the link carrying traffic from the customer network into the ISP's network. Let B be an important link in the ISP's network backbone, and let O be an outgoing link carrying traffic out of the ISP's network. Data collection devices on these links collect packet headers (possibly sampled [46]), do some processing on them (e.g., to compute packet identifiers [46]), and then stream them to the DSMS running the continuous query [27, 38, 46, 90]. Thus, we have three input streams which for convenience we also denote C, B, and O. Each tuple in these streams contains a packet identifier `pid` and the packet's `size` in bytes. The above continuous query can be posed in CQL as:

Select sum(C.size)
From C [Range 10 minutes], B [Range 10 minutes], O [Range 10 minutes]
Where C.`pid` = B.`pid` and B.`pid` = O.`pid`

This continuous query first joins streams C, B, and O on `pid` with a 10-minute *sliding window* of tuples on each stream. (The *Range* clause in CQL specifies a time-based sliding window [5].) The join output is then aggregated to continuously compute the total common traffic. A similar query could be used in sensor networks, e.g., to monitor moving objects and their paths [6, 61].

Figure 1.2 shows a multiway join plan for executing this query. Each stream feeds a 10-minute sliding window, W_c, W_b, and W_o for streams C, B, and O respectively. Each window contains a hash index on `pid`. When a tuple t is inserted into W_c, the other two windows are probed with t.`pid` in some order, e.g., the order W_o, W_b is used in Figure 1.2. If both windows contain a tuple matching t.`pid` (`pid` is unique), then the joined tuple is sent as an insertion to the aggregation operator *sum(size)* which maintains the sum incrementally. Otherwise, processing on t stops at the first window that does not contain a

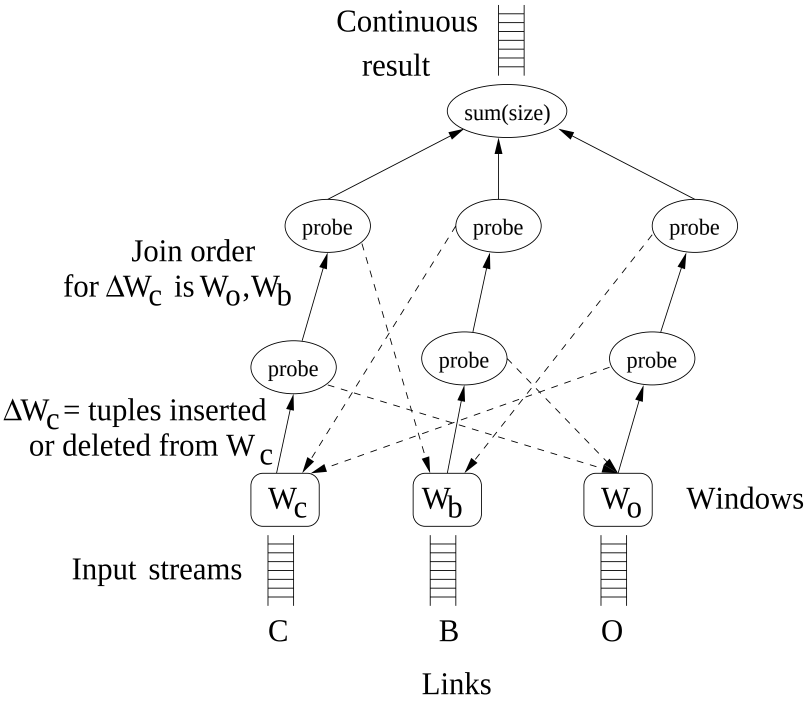


Figure 1.2: Query execution plan for example network-monitoring query

matching tuple. Similar processing occurs when a tuple t is deleted from W_c, i.e., when t becomes more than 10 minutes old and expires from the sliding window. Furthermore, similar processing occurs for insertions and deletions to the other two windows as shown in Figure 1.2. Note that each input stream in Figure 1.2 has its own order for probing the other windows.

The pipelined multiway join plan in Figure 1.2 is called an *MJoin* [114]. An MJoin requires a join order for each input stream which is used to join insertions and deletions in that stream with the windows of the other streams.

Suppose the current configuration of routing tables in the ISP's network is such that a significant fraction of the network packets arriving on link C is routed out of the network through link O, but almost none of the packets on link B pass through O. Then, the join order W_b, W_c for insertions and deletions in W_o can be significantly more efficient than the order W_c, W_b. However, over time the routing patterns can change (e.g., because of network congestion, changes in traffic distribution, attacks on the network, or hardware failures) so that almost all of the packets on link B now get routed out of the network through link O. Then, the join order W_c, W_b may quickly become much more efficient than the previously-optimal order W_b, W_c. Thus, changes in routing patterns may require corresponding changes in join orders to maintain good performance.

Suppose the rate of traffic on link B is much higher than that on links C and O. (Links in the core of the network often carry the aggregate of traffic across multiple incoming links.) Then, a more efficient plan P' than the one in Figure 1.2 may *materialize* the join of windows W_c and W_o, i.e., $W_{co} = W_c \bowtie_{pid} W_o$ is maintained as a materialized view [59]. The benefit of materializing W_{co} is that P' can process the high rate of insertions and deletions in W_b with a single probe into W_{co}, eliminating separate probes into W_c and W_o. Such plans that contain materialized state also need to be chosen in an adaptive fashion, e.g., because traffic rates can be extremely variable and bursty [118].

Streams C, B, and O exhibit some interesting properties. First, the packets being monitored flow through link C to link B to link O. Thus, a tuple corresponding to a specific `pid` appears in stream C first, then a joining tuple may appear in stream B, and lastly in stream O. Second, if the latency of the network between links C and B and between links B and O is bounded by d_{cb} and d_{bo} respectively, then a packet that flows through links C, B, and O will appear in stream B no later than d_{cb} time units after it appears in stream C, and in stream O no later than d_{bo} time units after it appears in B. Both of these properties, if "known" to the continuous query processor, can be exploited to reduce the memory requirement for the windows in Figure 1.2 significantly. For example, when a tuple b arrives in stream B and no joining tuple exists in the window on C, b can be discarded immediately because a tuple in C joining with b should have arrived before b. By detecting and exploiting the stream properties, the overall memory requirement for the plan in Figure 1.2 can be reduced by two orders of magnitude, as we will show in Chapter 6. However, routing paths and latencies in the network may change over time, so stream properties have to be detected and exploited in an adaptive manner.

1.4 Contributions and Outline of Thesis

The network-monitoring example highlights the important challenges in processing continuous queries over streams that we address in this thesis. This section summarizes our contributions and gives an overview of the chapters that follow.

- **The STREAM DSMS:** Chapter 2 describes our contributions in the overall design and implementation of the STREAM DSMS. We describe the CQL declarative query language and its semantics for continuous queries over streams and relations. We discuss how declarative CQL queries are translated into flexible query plans composed of operators, queues, and synopses, and how STREAM supports sharing of data and computation across multiple query plans. We describe how operators in query plans are scheduled so that peak memory usage of plans is minimized. Finally, we discuss STREAM's interface for run-time monitoring and manipulation of query plans.

- **The StreaMon Framework for Adaptive Query Processing:** As motivated in the network-monitoring example, adaptive approaches to query processing are required in DSMSs to handle fluctuating stream and system conditions over the lifetime of a continuous query. Chapter 3 describes the generic StreaMon framework for adaptive query processing in DSMSs. StreaMon is based on continuous monitoring for changes in stream and system conditions, and re-optimizing query plans quickly and efficiently when changes are detected. Chapters 4–6 describe three instantiations of the generic StreaMon framework for different classes of continuous queries and adaptivity needs.

- **Adaptive Ordering of Pipelined Filters:** Chapter 4 considers the class of continuous queries called *pipelined filters*, where a continuous stream of tuples is processed by a set of commutative filters. (If a set of filters is commutative, then the result of evaluating any of the filters on a tuple is independent of the order in which the filters are evaluated on the tuple.) We focus on the problem of ordering the filters adaptively to minimize processing cost in an environment where stream and system conditions may vary unpredictably over time. Our core algorithm for this problem, *A-Greedy*, adopts the generic StreaMon framework for adaptive query processing. When stream and system conditions are steady, the orderings converged on by A-Greedy are provably within a cost factor ≤ 4 of optimal. The other contribution of Chapter 4 is the identification of a three-way tradeoff in pipelined-filter processing among provable convergence to good orderings, run-time overhead, and speed of adaptivity. We present a suite of variants of A-Greedy that lie at different points on this tradeoff spectrum. Finally, we present a thorough performance evaluation of our algorithms in the STREAM DSMS.

- **Adaptive Processing of Multiway Stream Joins:** Chapter 5 considers the problem of processing continuous multiway *stream join* queries in environments where stream and system conditions can be unpredictable and volatile. We first consider MJoin plans for stream joins (recall Figure 1.2) and show how the A-Greedy algorithm can be used for adaptive ordering of join operators in MJoin pipelines. The theoretical guarantees provided by A-Greedy hold for a large class of stream joins. We show how MJoins may not be the most efficient way of processing stream joins under all scenarios. We describe our *A-Caching* algorithm that addresses this problem by placing *subresult caches* adaptively in MJoin pipelines. A-Caching enables our stream join plans to adapt over the entire spectrum between subresult-free MJoins and conventional tree-shaped join plans with materialized subresults at every intermediate node. Also, caches are advantageous from the adaptivity perspective because they can be added, populated incrementally, and dropped with little overhead. Like A-Greedy, A-Caching instantiates the generic StreaMon framework. Finally, we report a thorough experimental evaluation of A-Caching based on an implementation in the STREAM DSMS.

- **Exploiting k-Constraints over Data Streams:** Query plans for continuous queries often need to maintain significant amounts of run-time state over arbitrary data streams. For example, the plan in Figure 1.2 needs to store all tuples in the current windows over the streams. However, streams may exhibit certain data or arrival patterns, or *constraints,* that can be detected and exploited to reduce state considerably, without compromising correctness. Rather than requiring constraints to be satisfied precisely, which can be unrealistic in a data stream environment, we introduce k*-constraints,* where k is an *adherence parameter* specifying how closely a stream adheres to the constraint. (Smaller k's are closer to strict adherence and offer better state reduction.) Chapter 6 presents a query processing architecture, called k*-Mon*, that detects useful k-constraints automatically and exploits the constraints to reduce run-time state in query plans for continuous queries. Experimental results from an implementation of

k-Mon in the STREAM prototype DSMS show dramatic state reduction, while only modest computational overhead is incurred for the constraint monitoring and query execution algorithms.

- **Future Research Directions in Adaptive Query Processing:** Chapter 7 concludes this thesis by describing how adaptive query processing as a general technique applies beyond the processing of continuous queries in DSMSs, e.g., in the context of conventional query processing in DBMSs. We also outline future work in adaptive query processing.

1.5 Related Work

This section covers work related to data stream systems and to continuous query processing in general. Details of work related to the specific techniques proposed in this thesis are covered in the respective chapters.

1.5.1 Data Stream Management Systems

STREAM is a general-purpose DSMS that targets different application domains where continuous data streams arise and support for continuous queries is needed. Other general-purpose DSMSs include *Aurora* [29], *TelegraphCQ* [30], *Nile* [60], and *CAPE* [121]. All of these systems share important goals with STREAM. However, each of them differs from STREAM in some important aspects.

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. Unlike STREAM, Aurora has limited support for adaptive query processing, but richer support for distributed query processing and tolerance to failures [22].

The TelegraphCQ system is built on the Eddies adaptive query processor [10]. The relationship between Eddies and StreaMon from the perspective of adaptive query processing is covered in Section 3.5 in Chapter 3. Eddies is an integrated optimizer, executor, and operator scheduler. On the other hand, STREAM takes a modular approach to query processing, having separate optimization, execution, and scheduling components. Consequently, e.g., STREAM can support multiple scheduling policies [12, 13]. Unlike STREAM, which was developed from the ground up, TelegraphCQ was developed as an extension to the *PostgreSQL* relational DBMS [92]. The pros and cons of building a DSMS on top of an existing DBMS are documented in [79].

The Nile [60] and CAPE [121] DSMSs support many of the same features as STREAM. Unlike STREAM, Nile has limited support for adaptive query processing, but richer support for exploiting sharing of data and computation while processing multiple continuous queries concurrently. CAPE supports adaptive processing at many levels, but takes a different approach from StreaMon, as discussed in Section 3.5 in Chapter 3.

Gigascope is a DSMS tailored to the network monitoring domain [38]. Gigascope supports a declarative query language (*GSQL*) which is less expressive than CQL. Unlike STREAM, Gigascope takes a two-level approach to query processing where subqueries are pushed down to the network interface card to eliminate unneeded input stream tuples as quickly and efficiently as possible.

1.5.2 Other Continuous Query Processors

Apart from DSMSs, continuous queries are used in content-based filtering and publish-subscribe systems, e.g., [3, 43, 91, 99, 108]. Two recent systems, *OpenCQ* [82] and *NiagaraCQ* [34], support continuous queries for monitoring persistent data sets spread over a wide-area network, e.g., web sites over the Internet. While STREAM supports an expressive declarative language for continuous queries, both OpenCQ and NiagaraCQ support continuous queries specified in an event-condition-action format commonly used for triggers. Furthermore, OpenCQ and NiagaraCQ lack many of STREAM's features, e.g., flexible query plans and operator scheduler, adaptive processing of commutative filters and stream joins, and an interface to monitor running query plans. Like NiagaraCQ, STREAM has support for shared evaluation of common subexpressions across multiple continuous queries. However, unlike NiagaraCQ, STREAM does not yet support adaptive regrouping of queries based on changes in workload, data, or system conditions.

Materialized views [59] and triggers [117] in conventional DBMSs are effectively continuous queries that need to be processed whenever the base data changes or a monitored event happens. However, recall from Section 1.1 that materialized views and triggers are insufficient to meet the needs of stream-based applications easily and efficiently.

Chapter 2

The STREAM DSMS

In the previous chapter we motivated the need for Data Stream Management Systems (DSMSs) to meet the new requirements of stream-oriented applications. We now describe our contributions in the overall design and implementation of the STREAM general-purpose DSMS at Stanford.

Section 2.1 describes STREAM's CQL declarative query language and its semantics for continuous queries over streams and relations. CQL queries in STREAM are translated into query plans composed of operators, queues, and synopses. Details of query plans are presented in Section 2.2. Section 2.3 discusses two issues critical to STREAM's overall performance: sharing of data and computation among query plans, and an operator-scheduling algorithm that minimizes memory requirements for query plans under arbitrary stream arrival conditions. In Section 2.4 we describe the interface STREAM provides to DSMS administrators and users to monitor and manipulate query plans as they run. Related work is discussed in Section 2.5, and we conclude in Section 2.6.

2.1 The CQL Continuous Query Language

For simple continuous queries over streams—e.g., a query that filters a stream S, or joins S with an unvarying table—it can be sufficient to use a relational query language such as SQL, replacing references to relations with references to streams, and streaming new tuples in the result. However, as continuous queries grow more complex, e.g., with the addition of aggregation, subqueries, windowing constructs, and joins of streams and time-varying relations, the semantics of a conventional relational language applied to these queries quickly becomes unclear [5]. To address this problem, in the STREAM project we have defined a formal *abstract semantics* for continuous queries, and we have designed CQL, a concrete declarative query language that implements this semantics. While CQL is not a focus of this thesis, the query classes we consider in Chapters 4–6 are strict subsets of CQL. We now give a short overview of CQL and its semantics; full details appear in [5]. Detailed examples of the CQL queries considered in this thesis are given in Chapters 4–6.

2.1.1 Abstract Semantics

Our abstract semantics is based on two data types, *streams* and *relations*, which are defined using a discrete, ordered *time domain* Γ:

- A *stream* S is an unbounded bag (multiset) of *pairs* $\langle s, \tau \rangle$, where s is a tuple and $\tau \in \Gamma$ is the *timestamp* that denotes the logical arrival time of tuple s on stream S.

- A *relation* R is a time-varying bag of tuples. The bag of tuples at time $\tau \in \Gamma$ is denoted $R(\tau)$, and we call $R(\tau)$ an *instantaneous relation*. Note that our definition of a relation differs from the traditional one which has no built-in notion of time.

The abstract semantics uses three classes of operators over streams and relations:

- A *relation-to-relation* operator takes one or more relations as input and produces a relation as output.

- A *stream-to-relation* operator takes a stream as input and produces a relation as output.

- A *relation-to-stream* operator takes a relation as input and produces a stream as output.

Stream-to-stream operators are absent—they are composed from operators of the above three classes. These three classes are “black box” components of our abstract semantics:

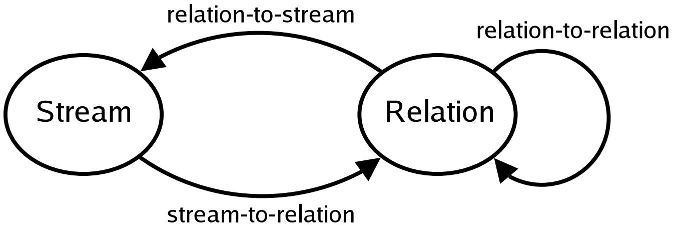


Figure 2.1: Data types and operator classes in abstract semantics

the semantics does not depend on the exact operators in these classes, but only on generic properties of each class. Figure 2.1 summarizes our data types and operator classes.

A continuous query Q is a tree of operators belonging to the above classes. The inputs of Q are the streams and relations that are input to the leaf operators, and the output of Q is the output of the root operator. The output is either a stream or a relation, depending on the class of the root operator. At time τ, an operator of Q logically depends on its inputs up to τ: tuples of S_i with timestamps $\leq \tau$ for each input stream S_i, and instantaneous relations $R_j(\tau')$, $\tau' \leq \tau$, for each input relation R_j. The operator produces new outputs corresponding to τ: tuples of S with timestamp τ if the output is a stream S, or instantaneous relation $R(\tau)$ if the output is a relation R. The behavior of query Q is derived from the behavior of its operators in the usual inductive fashion.

2.1.2 Concrete Language

Our concrete declarative query language, *CQL* (for *Continuous Query Language*), is defined by instantiating the operators of our abstract semantics. Syntactically, CQL is a relatively minor extension to SQL.

Relation-to-Relation Operators in CQL

CQL uses SQL constructs to express its relation-to-relation operators, and much of the data manipulation in a typical CQL query is performed using these constructs, exploiting the rich expressive power of SQL.

Stream-to-Relation Operators in CQL

The stream-to-relation operators in CQL are based on the concept of a *sliding window* [14] over a stream, and are expressed using a window specification language derived from SQL-99:

- A *tuple-based sliding window* on a stream S takes an integer $N > 0$ as a parameter and produces a relation R. At time τ, $R(\tau)$ contains the N tuples of S with the largest timestamps $\leq \tau$. It is specified by following S with "`[Rows N]`." As a special case, "`[Rows Unbounded]`" denotes the append-only window "`[Rows` ∞`]`."

- A *time-based sliding window* on a stream S takes a time interval ω as a parameter and produces a relation R. At time τ, $R(\tau)$ contains all tuples of S with timestamps between $\tau - \omega$ and τ. It is specified by following S with "`[Range` ω`]`." As a special case, "`[Now]`" denotes the window with $\omega = 0$.

- A *partitioned sliding window* on a stream S takes an integer N and a set of attributes $\{A_1, \ldots, A_k\}$ of S as parameters, and is specified by following S with "`[Partition By` $\mathtt{A}_1$`,...,`$\mathtt{A}_k$ `Rows N]`." It logically partitions S into different substreams based on equality of attributes $A_1, \ldots, A_k$, computes a tuple-based sliding window of size N independently on each substream, then takes the union of these windows to produce the output relation.

Relation-to-Stream Operators in CQL

CQL has three relation-to-stream operators: *Istream*, *Dstream*, and *Rstream*. *Istream* (for "insert stream") applied to a relation R contains the pair $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau) - R(\tau - 1)$, i.e., whenever s is inserted into R at time τ. *Dstream* (for "delete stream") applied to a relation R contains $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau - 1) - R(\tau)$, i.e., whenever s is deleted from R at time τ. *Rstream* (for "relation stream") applied to a relation R contains $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau)$, i.e., every current tuple in R is streamed at every time instant.

2.1.3 Example CQL Queries

Example 2.1.1 The following continuous query filters a stream S:

Select Istream(*)
From S [Rows Unbounded]
Where $S.A > 10$ and $S.B < 20$ and $S.C > 40$

Stream S is converted into a relation by applying an unbounded (append-only) window. Each of the three filters "$S.A > 10$", "$S.B < 20$", and "$S.C > 40$", is a relation-to-relation operator. The conjunction of these filters, as specified in the Where clause of the query, is a relation-to-relation operator formed by composing the three filters. This operator operates over the relation produced by the unbounded window on stream S, producing a relation whose tuples satisfy all the three filters. The inserts to this filtered relation over time are streamed in the query result by the Istream relation-to-stream operator.

CQL includes a number of syntactic shortcuts and defaults for convenience, which permit the above query to be rewritten in the following more intuitive form:

Select *
From S
Where $S.A > 10$ and $S.B < 20$ and $S.C > 40$

Example 2.1.2 The following continuous query is a *windowed join* of two streams S_1 and S_2:

Select *
From S_1 [Rows 1000], S_2 [Range 2 Minutes]
Where $S_1.A = S_2.A$ and $S_1.A > 10$ and $S_1.B < 20$

The answer to this query is a relation. At any given time, the answer relation contains the join (on attribute A, with $A > 10$ and $S_1.B < 20$) of the last 1000 tuples of S_1 with the tuples of S_2 that have arrived in the previous 2 minutes. If we prefer instead to produce a stream containing new A values as they appear in the join, we can write "Istream($S_1.A$)" instead of "*" in the Select clause.

2.2 Query Plans and Execution

When a continuous query specified in CQL is registered with the STREAM system, a *query plan* is compiled from it. Query plans are composed of *operators*, which perform the actual processing, *queues*, which buffer tuples (or references to tuples) as they move between operators, and *synopses*, which store operator state.

2.2.1 Operators

Recall from Section 2.1 that there are two fundamental data types in our query language: streams, defined as bags of tuple-timestamp pairs, and relations, defined as time-varying bags of tuples. We unify these two types in our implementation as sequences of timestamped tuples, where each tuple additionally is flagged as either an *insertion* ($+$) or *deletion* ($-$). We refer to the tuple-timestamp-flag triples as *elements*.

Streams only include $+$ elements, while relations may include both $+$ and $-$ elements to capture the changing relation state over time. Queues logically contain sequences of elements representing either streams or relations. Each query plan operator reads from one or more *input queues*, processes the input based on its semantics, and writes any output to an *output queue*. Individual operators may materialize their relational inputs in synopses (see Section 2.2.3) if such state is useful.

The operators in the STREAM system that implement the CQL language are summarized in Table 2.1. In addition, there are several *system operators* to handle "housekeeping" tasks such as marshaling input and output and connecting the query plans together. During execution, operators are *scheduled* individually, allowing for fine-grained control over queue sizes and query latencies. Scheduling algorithms are discussed later in Section 2.3.2.

2.2.2 Queues

A queue in a query plan connects a "producing" plan operator O_P to its "consuming" operator O_C. At any time a queue contains a (possibly empty) collection of elements representing a portion of a stream or relation. The elements that O_P produces are inserted into the queue and buffered there until they are processed by O_C.

Name	Operator Type	Description
`select`	relation-to-relation	Filters elements based on predicate(s)
`project`	relation-to-relation	Duplicate-preserving projection
`binary-join`	relation-to-relation	Joins two input relations
`mjoin`	relation-to-relation	Multiway join (MJoin) from [114]
`union`	relation-to-relation	Bag union
`except`	relation-to-relation	Bag difference
`intersect`	relation-to-relation	Bag intersection
`antisemijoin`	relation-to-relation	Antisemijoin of two input relations
`aggregate`	relation-to-relation	Performs grouping and aggregation
`duplicate-eliminate`	relation-to-relation	Performs duplicate elimination
`seq-window`	stream-to-relation	Implements time-based, tuple-based, and partitioned windows
`i-stream`	relation-to-stream	Implements *Istream* semantics
`d-stream`	relation-to-stream	Implements *Dstream* semantics
`r-stream`	relation-to-stream	Implements *Rstream* semantics

Table 2.1: Operators used in STREAM query plans

2.2.3 Synopses

Logically, a *synopsis* belongs to a specific plan operator, storing state that may be required for future evaluation of that operator. (In our implementation, synopses are shared among operators whenever possible, as described later in Section 2.3.1.) For example, to perform a windowed join of two streams, the join operator must be able to probe all tuples in the current window on each input stream. Thus, the join operator maintains one synopsis for each of its inputs. On the other hand, operators such as selection and duplicate-preserving union do not require any synopses.

The most common use of a synopsis in our system is to materialize the current state of a (derived) relation, such as the contents of a sliding window or the relation produced by a subquery. In Chapter 5, we will see synopses used to store partial intermediate join results in plans for multiway stream joins. Synopses used in query plans as part of join and aggregation operators always maintain indexes (e.g., hash indexes) on the respective join and grouping attributes to enable the operators to access selected subsets of the tuples efficiently. Synopses also may be used to store a summary of the tuples in a stream or relation for approximate query answering, e.g., see [88].

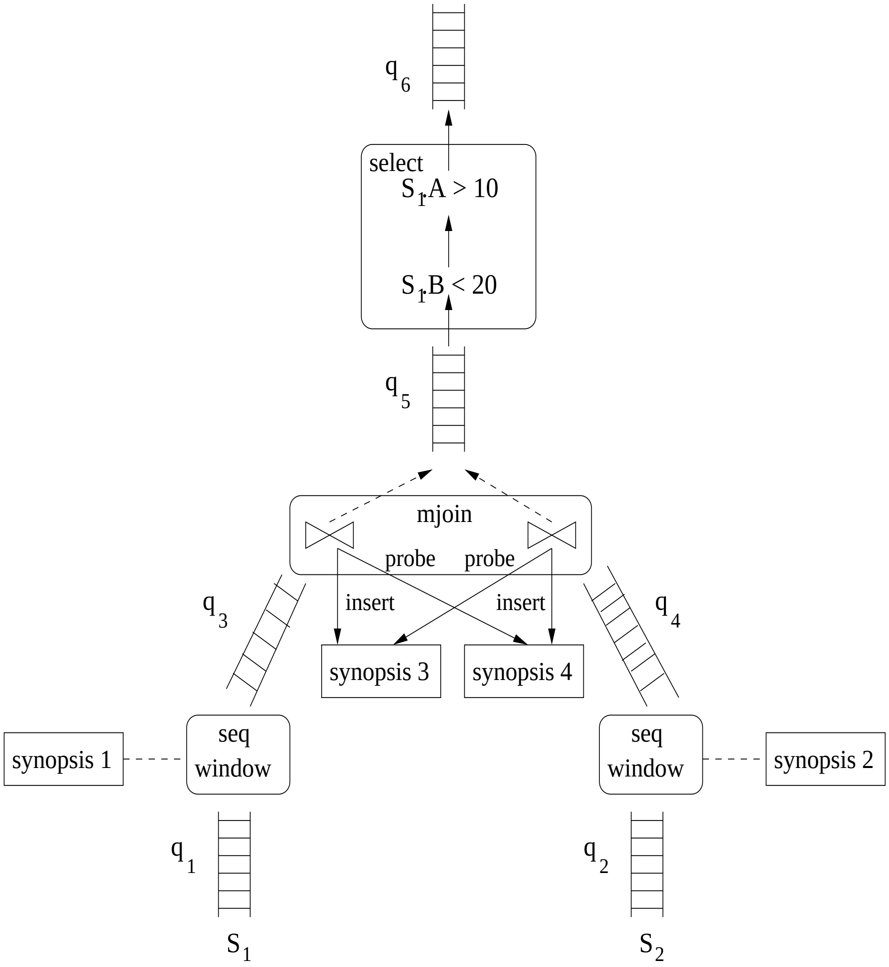


Figure 2.2: A simple query plan illustrating operators, queues, and synopses

Performance requirements often dictate that synopses (and queues) must be kept in memory, and we tacitly make that assumption throughout this thesis. Our system does support overflow of these structures to disk, although currently it does not implement sophisticated algorithms for minimizing I/O when overflow occurs, e.g., [109].

2.2.4 Example Query Plan

When a CQL query is registered, STREAM constructs a query plan: a tree of operators, connected by queues, with synopses attached to operators as needed. As a simple example, a plan for the query from Example 2.1.2 is shown in Figure 2.2.

There are four operators in the example plan: a `select`, an `mjoin`, and one instance of `seq-window` for each input stream. Queues q_1 and q_2 hold the input stream elements which could, for example, have been received over the network and placed into queues by a system operator (not depicted). Queue q_3, which is the output queue of the (stream-to-relation) operator `seq-window`, holds elements representing the relation "S_1 [Rows 1000]." Queue q_4 holds elements for "S_2 [Range 2 Minutes]." Queue q_5 holds elements for the joined relation "S_1 [Rows 1000] $\bowtie_{S_1.A=S_2.A}$ S_2 [Range 2 Minutes]". From these elements in q_5, Queue q_6 holds the elements passing the `select` operator which evaluates the conjunction of filters "$S_1.A > 10$" and "$S_1.B < 20$". q_6 may lead to an output operator sending elements to the application, or to another query plan operator within the system.

The $A > 10$ filter of the `select` operator can be pushed down into one or both branches below the `mjoin` operator, and also below the `seq-window` operator on S_2. However, tuple-based windows do not commute with filter conditions [5], and therefore the `select` operator cannot be pushed below the `seq-window` operator on S_1.

The plan has four synopses, $synopsis_1$–$synopsis_4$. Each `seq-window` operator maintains a synopsis so that it can generate "$-$" elements when tuples expire from the sliding window. The `mjoin` operator maintains synopses materializing each of its relational inputs for use in performing joins with tuples on the opposite input. Since the `select` operator does not need to maintain any state, it does not have a synopsis.

Note that the contents of $synopsis_1$ and $synopsis_3$ are similar (as are the contents of $synopsis_2$ and $synopsis_4$), since both maintain a materialization of the same window, but at slightly different positions of stream S_1. Section 2.3.1 discusses how we eliminate such redundancy by sharing synopses. (In the plan in Figure 1.2 for our example network-monitoring query from Chapter 1, we assumed that the synopses are shared between the join operator and the respective window operators.)

2.2.5 Query Plan Execution

When a query plan is executed, operators in the plan are scheduled to execute in turn. Continuing with our example from the previous section, when the `seq-window` operator on S_1 is scheduled, it reads stream elements from q_1. Suppose it reads element $\langle s, \tau, + \rangle$. It inserts tuple s into $synopsis_1$, and if the window is full (i.e., the synopsis already contains 1000 tuples), it removes the earliest tuple s' in the synopsis. It then writes output elements into q_3: the element $\langle s, \tau, + \rangle$ to reflect the addition of s to the window, and the element $\langle s', \tau, - \rangle$ to reflect the deletion of s' as it exits the window. Intuitively, the arrival of $\langle s, \tau, + \rangle$ in stream S_1 causes the insertion of s into the window and the deletion of s' from the window, all at logical time τ (recall Section 2.1.1). Therefore, the timestamps of both the output elements produced are the same, and also equal to that of the corresponding input element. The other `seq-window` operator is analogous.

When scheduled, the `mjoin` operator reads the earliest element across its two input queues. If it reads an element $\langle s, \tau, + \rangle$ from q_3, then it inserts s into $synopsis_3$ and joins s with the contents of $synopsis_4$, generating output elements $\langle s{\cdot}t, \tau, + \rangle$ for each matching tuple t in $synopsis_4$. Intuitively, the arrival of element $\langle s, \tau, + \rangle$ in stream S_1 at logical time τ causes the insertion of the tuples $s \cdot t$ in the join result at logical time τ. Similarly, if the `mjoin` operator reads an element $\langle s, \tau, - \rangle$ from q_3, it generates $\langle s{\cdot}t, \tau, - \rangle$ for each matching tuple t in $synopsis_4$, denoting the deletion of these tuples from the join result at logical time τ. A symmetric process occurs for elements read from q_4.

Note that the `mjoin` operator processes input elements in nondecreasing timestamp order across its input queues. Therefore, if the elements in each input queue are themselves in nondecreasing timestamp order, then the `mjoin` will produce its output elements also in nondecreasing timestamp order. This property of processing stream elements in timestamp order is characteristic of all STREAM operators. However, we cannot assume that the stream elements coming from external sources will always arrive in timestamp order. STREAM's approach to this problem is to use an "input manager" that will reorder elements if they are not in order, and also add timestamps to input tuples if the source does not provide them. This approach is based on monitoring and using the properties of input streams such as the degree of out-of-order arrival within a stream and the timestamp skew in the arrival of different streams. Full details of this approach are provided in [103], and are not a focus of this thesis.

Continuing with the discussion of the execution of the plan in Figure 2.2, the `select` operator dequeues elements from q_5. This operator tests the tuple in each element against its selection predicate, and enqueues the identical element into q_6 if the test passes, discarding it otherwise. Unlike the `seq-window` and `mjoin` operators, the `select` operator is *stateless*, that is, it does not maintain any run-time state based on the tuples processed so far. Recall that stateless operators in STREAM do not require any synopses.

Note that the semantics of each of the above operators depends only on the timestamps of the elements it processes, not, e.g., on the "wall-clock" time at the DSMS. This property is also characteristic of all STREAM operators, and it allows the order and duration for which plan operators are scheduled to have no effect on the data in the query result. However, scheduling strategies are important because they dictate performance metrics such as latency and resource utilization. Scheduling is discussed further in Section 2.3.2.

2.3 Performance Issues

In the previous section we introduced the basic architecture of our query execution engine. However, simply generating the most straightforward query plan corresponding to a query and executing it as described can be very inefficient. In this section, we discuss two ways in which we improve the performance of our system: by eliminating data redundancy (Section 2.3.1) and by scheduling operators to most efficiently reduce intermediate state (Section 2.3.2).

2.3.1 Synopsis Sharing

In Section 2.2.4, we observed that multiple synopses within a single query plan may materialize nearly identical relations. In Figure 2.2, $synopsis_1$ and $synopsis_3$ are an example of such a pair.

We eliminate this redundancy by replacing the two synopses with lightweight *stubs*, and a single *store* to hold the actual tuples. These stubs implement the same interfaces as non-shared synopses, so operators can be oblivious to the details of sharing. As a result, synopsis sharing can be enabled or disabled on the fly.

Since operators are scheduled independently, it is likely that operators sharing a single synopsis store will require slightly different views of the data. For example, if queue q_3 in Figure 2.2 contains 10 elements, then $synopsis_3$ will not reflect these changes (since the mjoin operator has not yet processed them), although $synopsis_1$ will. When synopses are shared, logic in the store tracks the progress of each stub, and presents the appropriate view (subset of tuples) to each of the stubs. Clearly the store must contain the union of its corresponding stubs: A tuple is inserted into the store as soon as it is inserted by any one of the stubs, and it is removed only when it has been removed from all of the stubs.

To further decrease state redundancy, multiple query plans involving similar intermediate relations can share synopses as well. For example, suppose the following query is registered in addition to the query in Section 2.2.4:

Select	A, Max(B)
From	S_1 [Rows 200]
Group By	A

The window on S_1 in this query is a subset of the window on S_1 in the other query. Thus, the same data store can be used to materialize both windows. The combination of the two query plans with both types of sharing is illustrated in Figure 2.3.

2.3.2 Operator Scheduling

An operator consumes elements from its input queues and produces elements on its output queue. Thus, the global operator scheduling policy can have a large effect on memory utilization, particularly with bursty input streams.

Consider the following simple example. Suppose we have a query plan with two operators, O_1 followed by O_2. Assume that O_1 takes one time unit to process a batch of n elements, and it produces $0.2n$ output elements per input batch (i.e., its *selectivity* is 0.2). Further, assume that O_2 takes one time unit to operate on $0.2n$ elements, and it sends its output out of the system. (As far as the system is concerned, O_2 produces no elements, and therefore its selectivity is 0.) Consider the following bursty arrival pattern: n elements arrive at every time instant from $t = 0$ to $t = 6$, then no elements arrive from time $t = 7$ through $t = 13$. Under this scenario, consider the following scheduling strategies:

- *FIFO scheduling*: When batches of n elements have been accumulated, they are passed through both operators in two consecutive time units, during which no other element is processed.

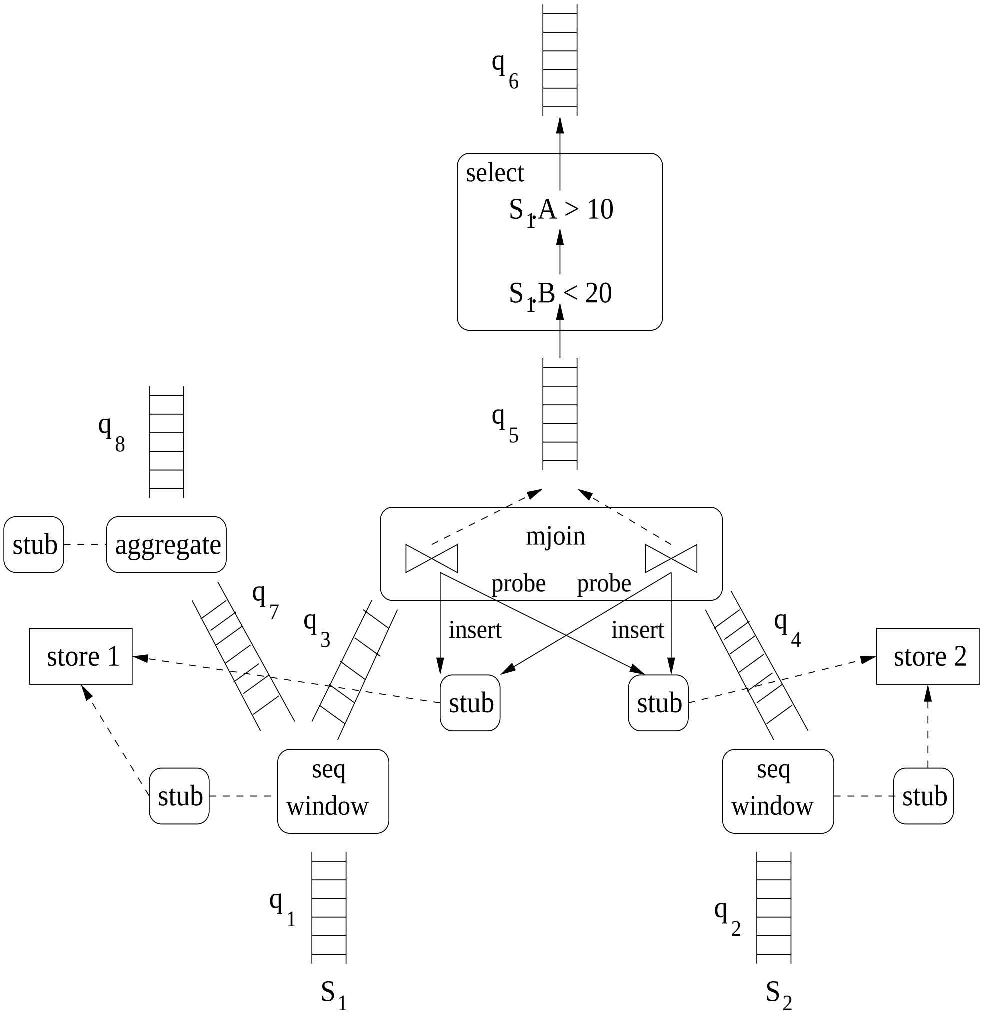


Figure 2.3: A query plan illustrating synopsis sharing

- *Greedy scheduling*: At any time instant, if there is a batch of n elements buffered before O_1, it is processed in one time unit. Otherwise, if there are more than $0.2n$ elements buffered before O_2, then $0.2n$ elements are processed using one time unit. This strategy is "greedy" since it gives preference to the operator that has the greatest rate of reduction in total queue size per unit time.

The following table shows the expected total queue size for each strategy, where each table entry is a multiplier for n.

Time	0	1	2	3	4	5	6	Avg
FIFO scheduling	1.0	1.2	2.0	2.2	3.0	3.2	4.0	2.4
Greedy scheduling	1.0	1.2	1.4	1.6	1.8	2.0	2.2	1.6

After time $t = 6$, input queue sizes for both strategies decline until they reach 0 after time $t = 13$. The greedy strategy performs better because it runs O_1 whenever it has input, reducing queue queue size by $0.8n$ elements each time step, while the FIFO strategy alternates between executing O_1 and O_2.

However, the greedy algorithm has its shortcomings. Consider a plan with operators O_1, O_2, and O_3. O_1 produces $0.9n$ elements per n input elements in one time unit, O_2 processes $0.9n$ elements in one time unit without changing the input size (i.e., it has selectivity 1), and O_3 processes $0.9n$ elements in one time unit and sends its output out of the system (i.e., it has selectivity 0). Clearly, the greedy algorithm will prioritize O_3 first, followed by O_1, and then O_2. If we consider the arrival pattern in the previous example then our total queue size is as follows (again as multipliers for n):

Time	0	1	2	3	4	5	6	Avg
FIFO scheduling	1.0	1.9	2.9	3.0	3.9	4.9	5.0	3.2
Greedy scheduling	1.0	1.9	2.8	3.7	4.6	5.5	6.4	3.7

In this case, the FIFO algorithm is better. Under the greedy strategy, although O_3 has highest priority, sometimes it is “blocked” from running because it is preceded by O_2, the operator with the lowest priority. If O_1, O_2 and O_3 are viewed as a single block, then together they reduce n elements to zero elements over three units of time, for an average reduction of $0.33n$ elements per unit time—better than the reduction rate of $0.1n$ elements O_1 provides. Since the greedy algorithm considers individual operators only, it does not take advantage of this fact.

This observation forms the basis of our *chain scheduling* algorithm [12]. Our algorithm forms blocks (“chains”) of operators as follows: Start by marking the first operator in the plan as the “current” operator. Next, find the block of consecutive operators starting at the “current” operator that maximizes the reduction in total queue size per unit time. Mark the first operator following this block as the “current” operator and repeat the previous step until all operators have been assigned to chains. Chains are scheduled according to the greedy algorithm, but within a chain, execution proceeds in FIFO order. In terms of overall memory usage, this strategy is provably close to the optimal “clairvoyant” scheduling strategy, i.e., the optimal strategy based on knowledge of future input, as shown in [12].

2.4 The STREAM System Interface

In a system for continuous queries, it is important for users, system administrators, and system developers to have the ability to inspect the system while it is running and to experiment with adjustments. To meet these needs, we have developed a graphical *query and system visualizer* for the STREAM system. The visualizer allows the user to:

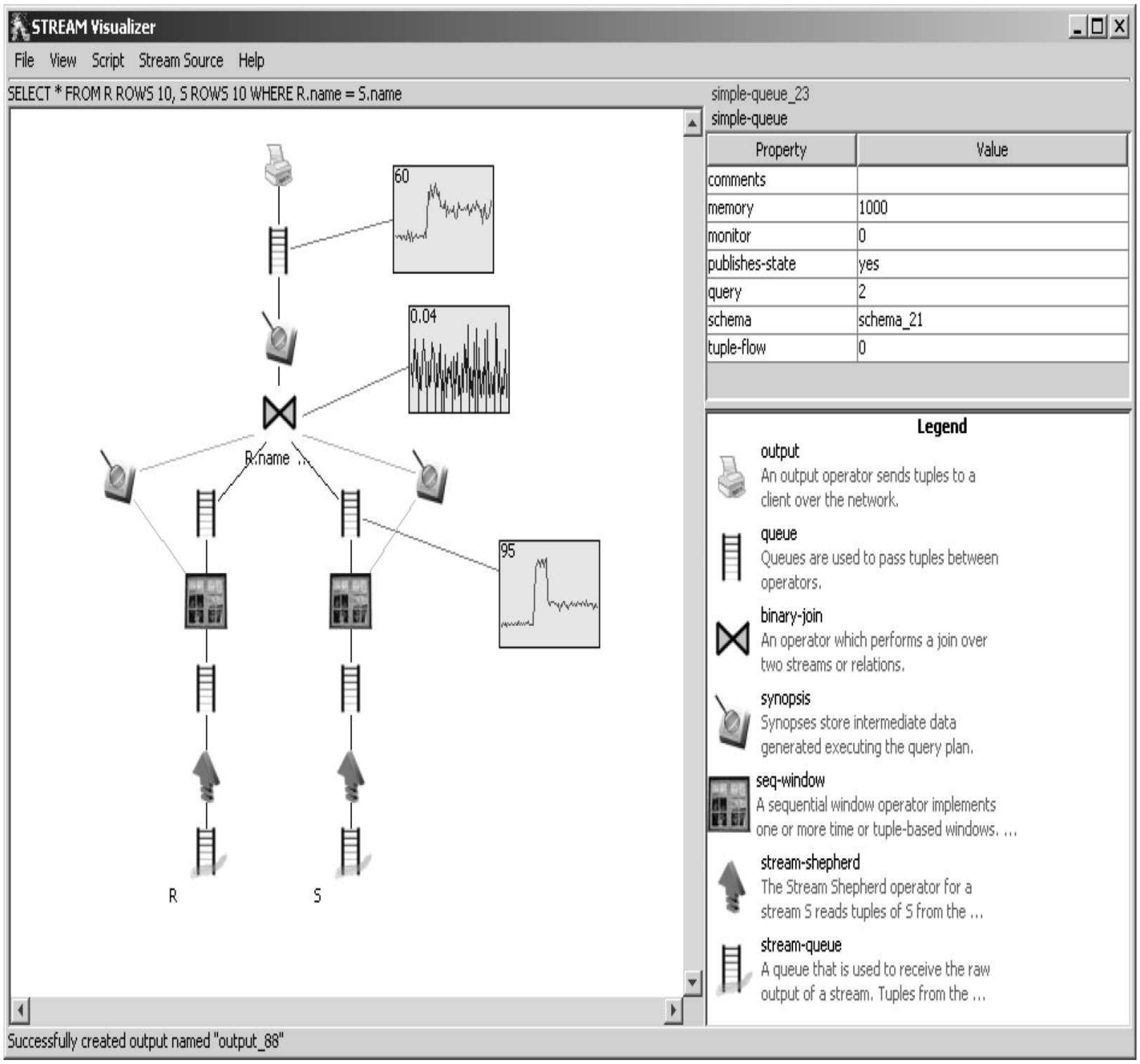
- View the structure of query plans and their component *entities* (operators, queues, and synopses). Users can view the path of data flow through each query plan as well as the sharing of computation and state within the plan.

- View the detailed properties of each entity. For example, the user can inspect the amount of memory being used (for queue and synopsis entities), the current throughput (for queue and operator entities), selectivity of predicates (for operator entities), and other properties.

- Dynamically adjust entity properties. These changes are reflected in the system in real time. For example, an administrator may choose to increase the size of a queue to better handle bursty arrival patterns.

- View *monitoring graphs* that display time-varying entity properties such as queue sizes, throughput, overall memory usage, and join selectivity, plotted dynamically against time.

A screenshot of our visualizer is shown in Figure 2.4. The large pane at the left displays a graphical representation of a currently selected query plan. The particular query shown is a windowed join over two streams, R and S. Each entity in the plan is represented by an icon: the ladder-shaped icons are queues, the boxes with magnifying glasses over them are synopses, the window panes are windowing operators, and so on. In this example, the



SELECT * FROM R ROWS 10, S ROWS 10 WHERE R.name = S.name

simple-queue_23
simple-queue

Property	Value
comments	
memory	1000
monitor	0
publishes-state	yes
query	2
schema	schema_21
tuple-flow	0

output
An output operator sends tuples to a client over the network.

queue
Queues are used to pass tuples between operators.

binary-join
An operator which performs a join over two streams or relations.

synopsis
Synopses store intermediate data generated executing the query plan.

seq-window
A sequential window operator implements one or more time or tuple-based windows. ...

stream-shepherd
The Stream Shepherd operator for a stream S reads tuples of S from the ...

stream-queue
A queue that is used to receive the raw output of a stream. Tuples from the ...

Successfully created output named "output_88"

Figure 2.4: Screenshot of the STREAM visualizer

user has added three monitoring graphs: the rate of element flow through queues above and below the join operator, and the selectivity of the join.

The upper-right pane displays the property-value table for a currently selected entity. The user can inspect this list and can alter the values of some of the properties interactively. Finally, the lower-right pane displays a legend of entity icons and descriptions for reference.

Our technique for implementing the monitoring graphs shown in Figure 2.4 is based on *introspection queries* over a special system-managed stream called *SysStream*. Every entity can publish any of its property values at any time onto *SysStream*. When a specific dynamic monitoring task is desired, e.g., monitoring recent join selectivity, the relevant entity writes its statistics periodically on *SysStream*. Then a standard CQL query, typically a windowed aggregation query, is registered over *SysStream* to compute the desired continuous result, which is fed to the monitoring graph in the visualizer. Users and applications can also register arbitrary CQL queries over *SysStream* for customized monitoring tasks.

2.5 Related Work

This section discusses work related to the components of data stream management that we considered in this chapter.

In Section 2.1 we gave an overview of STREAM's CQL declarative query language and formal abstract semantics for continuous queries. The design and implementation of CQL is not a focus of this thesis. The complete details of CQL, its abstract semantics, and its relationship to previous work on triggers, materialized views, and other languages for continuous queries are given in [5]. For example, in [5] we show that basic CQL (without user-defined functions, aggregates, or window operators) is strictly more expressive than *Tapestry* [108], *Tribeca* [106], GSQL [38], and materialized views over relations with or without *chronicles* [73]. We also discuss our choice to define a language based on both relations and streams, instead of taking a stream-only approach as in *TelegraphCQ* [30]. In this thesis we consider the processing of two classes of CQL queries: pipelined stream filters and multiway stream joins. Detailed examples of these query classes are given in Chapters 4–6.

STREAM processes continuous queries in an incremental fashion using flexible query execution plans that support adaptive re-optimization and fine-grained scheduling decisions. Related work in this category includes the processing of joins [56, 61, 114] and aggregation [44, 53] over streams. While the join operators considered in [56, 61, 114] are similar in many respects to STREAM's `mjoin` operator, they have no support for adaptive processing. References [44, 53] focus on giving approximate answers while processing aggregations over streams under resource constraints. Sharing of data and computation across multiple continuous queries is considered in detail in [8, 43]. Neither approximation nor resource sharing are a focus of this thesis.

The uncontrollable push-based arrival of data streams and the need for *quality of service* (*QoS*) guarantees for continuous queries have motivated work on QoS-aware scheduling of operators in DSMSs. In Section 2.3.2 we discussed our work on scheduling operators to minimize memory usage for bursty streams. We have also worked on multi-objective scheduling [13], e.g., scheduling to minimize memory requirements subject to latency constraints. Related work in this category includes scheduling strategies for shared window joins [62] and scheduling strategies for a variety of QoS metrics [28]. Operator scheduling strategies are not a focus of this thesis.

When incoming stream arrival rates exceed the DSMS's ability to provide exact results for the active queries, DSMSs perform *load-shedding* by introducing approximations that gracefully degrade accuracy [15, 107]. The techniques proposed in this thesis are for processing continuous queries with accurate results, although the techniques in Chapter 6 for exploiting stream-based constraints may inadvertently introduce approximations if the input stream properties change abruptly.

STREAM currently does not implement sophisticated algorithms for minimizing I/O when memory limitations force synopses and queues to be written to disk. Other work has addressed techniques for handling operators that join streaming data with data archived on disk [32], and stateful operators like windowed joins and aggregation that control how run-time state is spilled to disk when it no longer fits in the available memory [112, 114]. While the work in this thesis has been done in the context of a main-memory DSMS, some of our techniques extend to DSMSs that handle disk-resident data. Details will be provided in the respective chapters.

2.6 Conclusion

In this chapter we gave an overview of the overall design and implementation of the STREAM general-purpose DSMS. We described STREAM's CQL declarative query language for continuous queries and the details of query plans STREAM uses to execute CQL queries. We discussed two issues critical to STREAM's overall performance: sharing of data and computation among query plans, and an operator-scheduling algorithm that minimizes memory requirements for plans under arbitrary stream arrival conditions. Finally, we described STREAM's graphical interface for monitoring and manipulating query plans.

The STREAM prototype is now a mature system that can support stream-based applications. We conclude this chapter by outlining the major milestones in our development of the STREAM prototype.

1. **System demonstration:** The first major milestone in the development of the STREAM prototype was its demonstration at the ACM International Conference on Management of Data (SIGMOD), June 2003. This demonstration focused on the CQL language and the graphical interface for monitoring and manipulating query plans.

2. **Web-accessible demonstration:** In November 2003 we made the STREAM server accessible over the public Internet. Without installing any source code, users can connect using an HTML browser to a STREAM server running on a computer inside Stanford. Once connected, STREAM's graphical client is started on the user's local machine. With this client-server arrangement, users can try out almost all of the STREAM prototype's functionality, including running the Linear Road Benchmark for DSMSs [6].

3. **Release of source code:** The source code for the STREAM system was released publicly in late 2004. This release includes the code for the STREAM server, a simple textual client, and STREAM's graphical client. The graphical client incorporates the interface described in Section 2.4 for monitoring and manipulating query plans.

Chapter 3

Adaptive Query Processing with StreaMon

The previous chapter described how query plans composed of operators, queues, and synopses are used to execute continuous queries in the STREAM DSMS. Given a CQL query Q, we could use a conventional DBMS *plan-first execute-next* approach [100] to generate a plan for it. A query optimizer would first enumerate possible plans for Q. For example, for the stream filter query in Example 2.1.1 from the previous chapter, the optimizer has a choice of six execution plans corresponding to the six different orders in which the three filters can be evaluated on each tuple. While enumerating plans, the optimizer estimates the cost (e.g., number of processor cycles or wall-clock time) of each plan to process the input stream tuples arriving per unit time (e.g., every second). The plan with lowest estimated cost is then used to execute Q throughout its lifetime in the DSMS.

However, this conventional DBMS approach to query processing can cause severe performance degradation in a DSMS. The cost of a plan for a continuous query depends on the current stream conditions (e.g., data distribution, arrival rate) and system conditions (e.g., query load, memory availability). In a DSMS, stream and system conditions may vary significantly over the lifetime of a long-running continuous query. A plan that is most efficient for a continuous query Q at a point in time may become very inefficient to process Q when stream or system conditions change. Consequently, it is important for a DSMS to support *adaptive query processing* [10, 21, 64]: The DSMS must be prepared to change the execution plan for a continuous query while the query is running, based on how stream and system conditions change. Without adaptivity, plan performance may drop drastically over time.

In this chapter we present an overview of our generic framework, called *StreaMon*, for adaptive query processing in a DSMS. We first present the overall StreaMon framework in Section 3.1, then in Sections 3.2–3.4 we describe three instantiations of StreaMon. In each case we have instantiated the generic components of StreaMon in specific ways to address a particular continuous query type and adaptivity need. These three instantiations form the basis of the work discussed in more detail in Chapters 4–6.

Section 3.2 instantiates StreaMon to process commutative-filter queries adaptively over a stream, like the CQL query in Example 2.1.1 in Chapter 2. This query represents a very common class of continuous queries, which we call *pipelined filters*. The challenge in pipelined-filter processing is to maintain the order in which the filters are evaluated over input stream tuples that is most efficient for the stream and system conditions at any point in time.

Section 3.3 instantiates StreaMon to process windowed join queries adaptively over streams, like the CQL query in Example 2.1.2 in Chapter 2. An optimizer is faced with a large number of options while picking a plan for a windowed stream join, e.g., pipelined plans, full materialization of intermediate join results, partial caching of join results, memory allocation to multiple synopses, and others. Furthermore, unlike conventional DBMS optimizers, a continuous-query optimizer needs to consider the cost of switching to another plan when a change in stream or system conditions makes the current plan suboptimal.

Query plans for continuous queries often need to maintain significant run-time state in memory, which limits the number of queries the DSMS can process simultaneously. For example, the query plan for a windowed stream join query (see Figure 1.2 in Chapter 1) needs to store all tuples in the current windows over the streams. Section 3.4 describes an instantiation of StreaMon that addresses this problem. Our basic approach is to monitor input streams for various data or arrival patterns that can be exploited to reduce run-time state considerably, without compromising correctness. Our approach is designed to adapt to changes in such patterns over time.

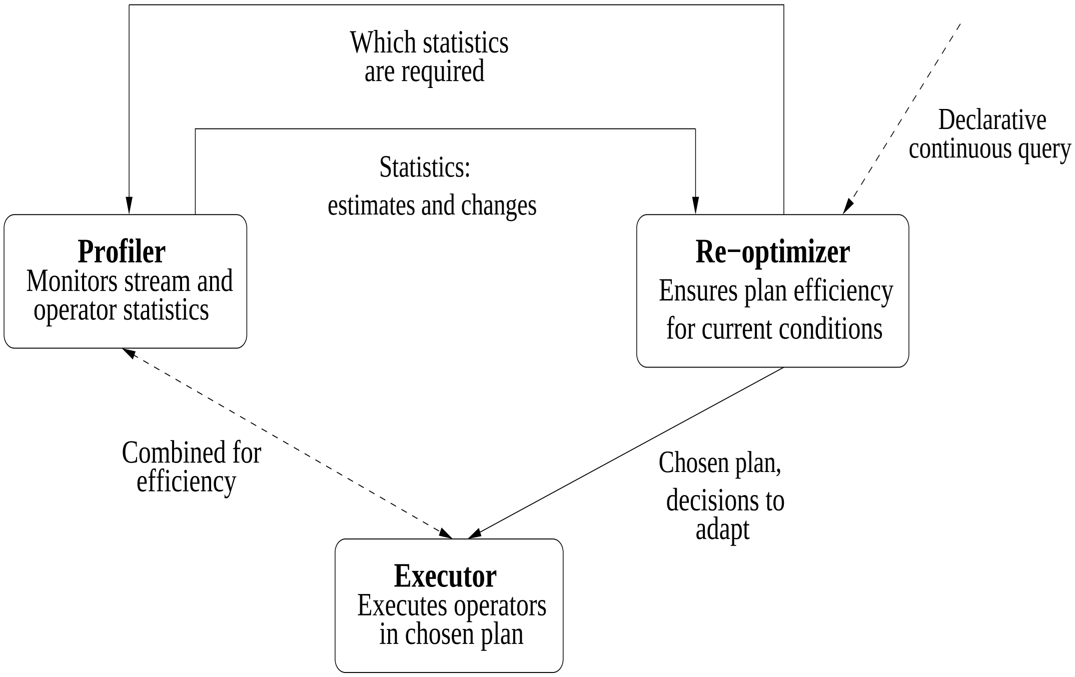

Which statistics are required
Declarative continuous query
Statistics: estimates and changes
Profiler
Monitors stream and operator statistics
Re−optimizer
Ensures plan efficiency for current conditions
Combined for efficiency
Chosen plan, decisions to adapt
Executor
Executes operators in chosen plan

Figure 3.1: Components of the generic StreaMon framework

3.1 StreaMon

We begin by presenting our generic StreaMon framework for adaptive query processing in a DSMS. StreaMon has three components as shown in Figure 3.1:

- An *Executor*, which produces query results by running currently-chosen plans on incoming stream tuples and relation updates.
- A *Profiler*, which collects and maintains statistics about current stream and system conditions.
- A *Re-optimizer*, which ensures that current plans and resource usage are the most efficient for current stream and system conditions.

A declarative continuous query Q is input to the Re-optimizer, which initially performs the same role as that of a conventional DBMS optimizer. The Re-optimizer first enumerates possible plans for Q. It then estimates the execution cost of each plan based on a cost model that predicts the *unit-time cost* [75] of each plan. The unit-time cost of a plan P represents the average cost in terms of execution time to process all of P's input tuples arriving at the DSMS in a single time unit. This metric, used commonly for continuous queries, will be used throughout this thesis. Motivation and details of this metric are given in [75] where it was first introduced.

The Re-optimizer picks the plan P_{min} with the minimum estimated unit-time cost based on the current stream and system conditions. P_{min} is given to the Executor, which starts using P_{min} to process Q over its input stream tuples and relation updates.

At the same time, the Re-optimizer informs the Profiler which statistics about the input streams and relations as well as the system are required to identify the best plan for Q. The estimated values that were used by the Re-optimizer for these statistics are also given to the Profiler. The Profiler then continuously monitors and maintains these statistics over time. If the value of any of these statistics changes by a significant amount, the Profiler informs the Re-optimizer. The Re-optimizer then repeats the plan enumeration and costing step with the new set of statistics to check whether the current plan P_{min} used by the Executor has become less efficient than the best plan P' for the new statistics. If P_{min} is less efficient than P', then the Re-optimizer tells the Executor to switch to P' for future processing of Q.

The steps outlined above are the generic steps involved in the operation of StreaMon for a continuous query. Some of these steps may incur significant run-time overhead if executed naively. The instantiations of StreaMon that we describe in this thesis incorporate many techniques to reduce this potential overhead. While the complete details of these techniques are deferred to the respective chapters, we give some examples here.

The Profiler and the Executor are often combined in part so that many of the statistics required for plan costing can be simply observed as part of regular query execution. Such an interaction usually reduces run-time overhead significantly compared to having the Profiler compute these statistics directly. As another example, the instantiation of StreaMon for pipelined filters implements a technique that avoids the need to repeat plan enumeration each time stream or system conditions change. With this technique, the new best plan can be computed in an incremental fashion starting from the current plan. Furthermore, in Chapter 7 we propose a new technique called *plan-logging* that reduces StreaMon's run-time overhead gradually over time for long-running continuous queries.

As we have just seen, the *run-time overhead* incurred to maintain adaptivity is an important metric in adaptive query processing. Two other metrics are important as well: *convergence properties* and *speed of adaptivity*. The convergence properties of an adaptive query processing algorithm are the properties of the plan that the algorithm converges to when input stream and system conditions are stable (unvarying). For example, the property of the converged plan that we consider in this thesis is the ratio of this plan's cost to the cost of the optimal plan for the stable stream and system conditions. Other convergence properties of interest might be, e.g., the steady-state memory requirement, or average latency of tuple flow through the plan. We do not expect stream and system conditions to change all the time, so periods of change will be separated by (possibly long) periods when conditions are stable. The convergence properties of an adaptive algorithm indicate how good we can expect the plan performance to be during these stable periods. When conditions change, the speed of adaptivity dictates how quickly the algorithm will converge to its new selected plan for the new conditions.

We have observed a three-way tradeoff in adaptive query processing among the run-time overhead, convergence properties, and speed of adaptivity metrics. For example, algorithms that adapt quickly and have good convergence properties typically require significantly more run-time overhead. If we have strict limitations on the run-time overhead, then we usually must sacrifice either speed of adaptivity or convergence properties. Adaptive algorithms in StreaMon are designed around this three-way tradeoff. For some of the query classes we consider, we have developed a suite of adaptive algorithms that lie at different points along this tradeoff spectrum. More details of the three-way tradeoff and how it affects our approach are given in Chapters 4 and 5.

Having given an overview of StreaMon, we are ready to describe our instantiations of the generic components of StreaMon for specific classes of continuous queries.

3.2 Adaptive Ordering of Pipelined Filters

The first class of continuous queries we consider for adaptive processing is called pipelined filters. A pipelined filter query is a conjunction of commutative filters over a single stream, as illustrated by the CQL query in Example 2.1.1 in Chapter 2. This query is reproduced here for convenience:

Select *

Select	*
From	S
Where	$S.A > 10$ and $S.B < 20$ and $S.C > 40$

This query operates on stream S. Each of the filters in the query $S.A > 10$, $S.B < 20$, and $S.C > 40$ will evaluate to true or to false on a tuple of S, and the result of the query is a stream containing S tuples that satisfy all three filters. Because the filters are commutative, they can be evaluated in any order on each S tuple without affecting the correctness of the query result. Consequently, we have six possible execution plans for this query, corresponding to the six $(= 3!)$ ways in which we can order three filters.

The STREAM system will use a plan with a single `select` operator to process this filter query (recall Section 2.2.4 from Chapter 2). However, at any point in time, the `select` operator needs to pick one of the six possible filter orders to process incoming S tuples. The unit-time cost of these plans differ based on the selectivity and the per-tuple processing time of each filter. (The selectivity of a filter is the probability that it will evaluate to true on a random input tuple.)

Let us denote the filters $S.A > 10$, $S.B < 20$, and $S.C > 40$ as F_1, F_2, and F_3 respectively. Let d_i denote the probability that filter F_i will evaluate to false on a random tuple in stream S. That is, the selectivity of F_i is $1 - d_i$. Let t_i denote the average processing time per tuple for filter F_i. Since all three filters are simple boolean predicates on a single attribute, we will assume that $t_1 = t_2 = t_3$.

Suppose $d_3 > d_1$ based on the current data characteristics of tuples arriving in stream S. That is, the probability of $S.C$ being greater than 40 is less than the probability of $S.A$ being greater than 10 for a random S tuple arriving at the DSMS. In this scenario, it is clearly more efficient on average to evaluate F_3 before F_1 on incoming S tuples. Therefore, F_3 should come before F_1 in the filter ordering used for the query.

Now suppose $d_2 > d_3 > d_1$. Extending our above reasoning, the best filter ordering is F_2, F_3, F_1, i.e., F_2 is evaluated first on an incoming tuple $s \in S$, followed by F_3 if s passes F_2, and finally F_1 if s passes both F_2 and F_3. However, this ordering may be suboptimal compared to F_2, F_1, F_3 if filters F_2 and F_3 are *correlated* (nonindependent). For example, the probability that a tuple $s \in S$ has $s.C > 40$ may be very high if we already know that $s.B < 20$. In general, we say that filters F_i and F_j over stream S are correlated

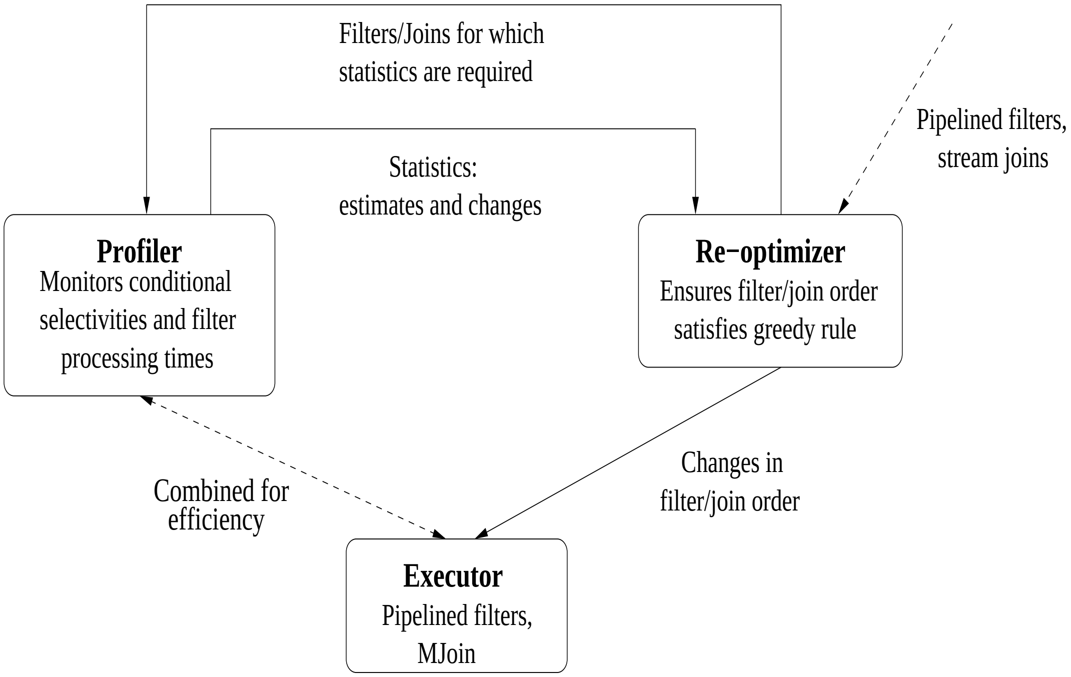

Filters/Joins for which statistics are required
Pipelined filters, stream joins
Statistics: estimates and changes
Profiler
Monitors conditional selectivities and filter processing times
Re−optimizer
Ensures filter/join order satisfies greedy rule
Combined for efficiency
Changes in filter/join order
Executor
Pipelined filters, MJoin

Figure 3.2: A-Greedy

if $d_{ij} \neq d_i \times d_j$, where d_{ij} is the probability that both F_i and F_j will evaluate to false on a random tuple in S. As we illustrated above, ordering filters without accounting for correlations can be significantly suboptimal.

Now suppose the distribution of data among incoming S tuples changes so that the values of d_2 and d_3 drop all of a sudden. Because of this change, the DSMS should switch to using the filter evaluation order F_1, F_2, F_3 which is much more efficient for the new conditions than the previously optimal F_2, F_1, F_3.

The above example motivates the problem of selecting good orderings for potentially-correlated pipelined filters, and adapting over time as stream or system conditions change. We developed the *A-Greedy* algorithm to address this problem. A-Greedy instantiates the generic StreaMon framework as shown in Figure 3.2.

The job of A-Greedy's Re-optimizer for a pipelined filter query over a stream S is to ensure that at any point in time the filter ordering used to process incoming S tuples is efficient for the current stream and system conditions. Unfortunately, it has been shown that the problem of finding the optimal ordering of a set of correlated pipelined filters is NP-Hard [89]. A-Greedy overcomes this hurdle using a simple greedy algorithm for ordering the filters that takes into account the conditional (joint) selectivities for correlated filters. This greedy algorithm is very efficient, and more importantly, this polynomial-time algorithm for the NP-Hard filter-ordering problem gives an ordering whose unit-time cost is provably within a constant factor 4 of the cost of the optimal ordering. In experiments the greedy algorithm usually finds the optimal ordering.

The Re-optimizer is able to make the strong theoretical claim about the filter ordering, and achieve good performance in practice, but it requires taking into account conditional filter selectivities as well as filter processing times. Furthermore, to adapt to changes in stream and system conditions, we must monitor continuously to determine when statistics change. In particular, we must detect when statistics change enough that the current ordering is no longer consistent with the ordering that would be selected by the greedy algorithm. A-Greedy's Profiler is responsible for keeping online estimates of statistics and detecting when they change, as indicated in Figure 3.2.

For n filters, the total number of conditional selectivities is $n2^{n-1}$. For small n, it may be feasible for the Profiler to maintain online estimates of all these selectivities. However, this task quickly becomes infeasible as n increases, and also is impractical for an online adaptive approach. A-Greedy's Profiler addresses this problem by maintaining a summary structure that keeps track of a selected quadratic number of conditional filter selectivities. Details of this structure as well as the techniques used by the Profiler for efficiently approximating the conditional selectivities are given in Chapter 4. Roughly, statistics collection is based on sampling as well as the observation of regular query execution to reduce the run-time overhead of the Profiler.

Based on the summary structure maintained by the Profiler, the Re-optimizer can detect quickly when statistics have changed such that the current ordering used by the Executor is no longer consistent with the ordering that would be selected by the greedy algorithm. This structure also enables the Re-optimizer to compute the new greedy ordering efficiently when statistics change. When a change is detected, the Re-optimizer informs the Executor which then switches to the new greedy ordering. Switching to a new ordering in pipelined filters does not require any change in run-time state, so it involves little overhead.

The complete details of A-Greedy will be described in Chapter 4. Next we describe our instantiation of StreaMon for processing a more complex class of continuous queries adaptively.

3.3 Adaptive Caching for Windowed Stream Joins

Our second instantiation of StreaMon considers the adaptive processing of multiway windowed stream join queries. The example network-monitoring query in Section 1.3 from Chapter 1 is an instance of a windowed stream join. The query is reproduced here for convenience:

Select sum(C.size)
From C [Range 10 minutes], B [Range 10 minutes], O [Range 10 minutes]
Where C.`pid` = B.`pid` and B.`pid` = O.`pid`

Recall that this continuous query joins streams C, B, and O on attribute `pid` with a 10-minute sliding window of tuples on each stream. The MJoin-based plan in Figure 1.2 for this query is repeated in Figure 3.3 for convenience. This plan contains a pipeline of join operators for each input stream, which is used to join insertions and deletions in that stream with the windows of the other streams. For example, when a tuple $c \in C$ is inserted into the window W_c, the other two windows are probed with c.`pid` in some order, e.g., the order W_o, W_b is used in Figure 3.3. If both windows contain tuples matching c.`pid`, then joined tuples are created and sent as insertions to the aggregation operator. Otherwise, processing on c stops at the first window that does not contain a matching tuple.

A join pipeline in an MJoin is a pipeline of commutative operators, similar to pipelined filters. However, unlike a filter, a join operator may produce an arbitrary number of output tuples per input tuple. Although the mapping between join pipelines and pipelined filters is not exact, the A-Greedy algorithm can be extended in a relatively straightforward manner to order join operators in MJoins, as indicated in Figure 3.2. Full details of this extended A-Greedy algorithm are given in Chapter 5.

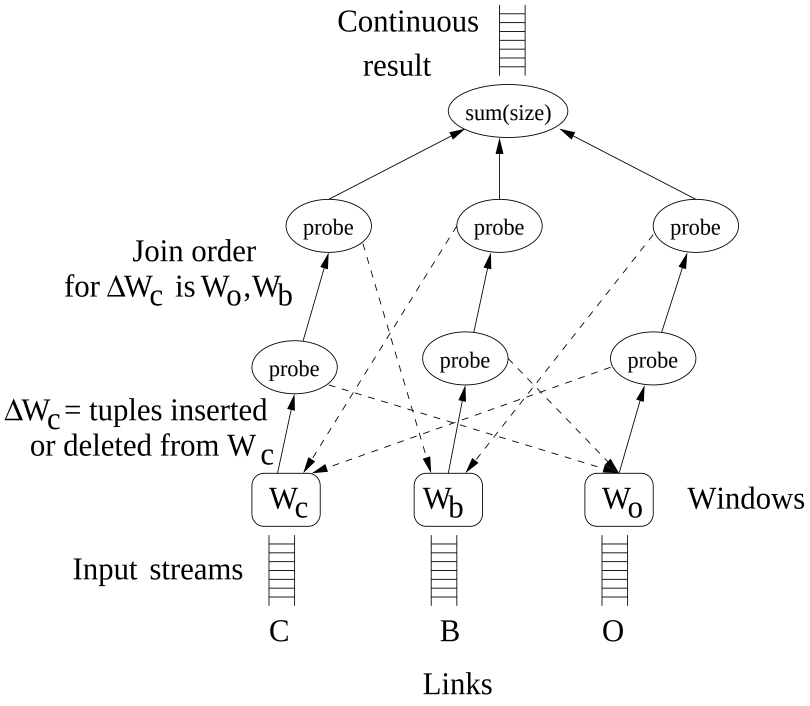


Figure 3.3: MJoin-based plan for example network-monitoring query

The combination of MJoin plans and the extended A-Greedy algorithm for adaptive ordering of MJoin pipelines gives us a complete approach for adaptive processing of multiway windowed stream joins. However, MJoins may not be the most efficient way of processing windowed stream joins under all scenarios because MJoins do not maintain intermediate join results. For example, consider again the MJoin in Figure 3.3. Each tuple $c \in C$ will be joined first with W_o, and the resulting tuples will be joined with W_b. The intermediate join results generated—$c \bowtie W_o$, $c \bowtie W_o \bowtie W_b$—are discarded after c is processed. Not maintaining such intermediate results can limit performance in scenarios where values of join attributes repeat in incoming tuples in C. For example, if $c' \in C$ arrives after c with the same join-attribute values as c, then the MJoin will repeat the computation it did for c, to generate the same intermediate join results again.

In contrast to the multiway MJoins, an *XJoin* [112], which is a tree of two-way joins, maintains a *fully-materialized* join result for each intermediate two-way join in the plan. Figure 3.4(a) shows an XJoin-based plan for our example network-monitoring query. This

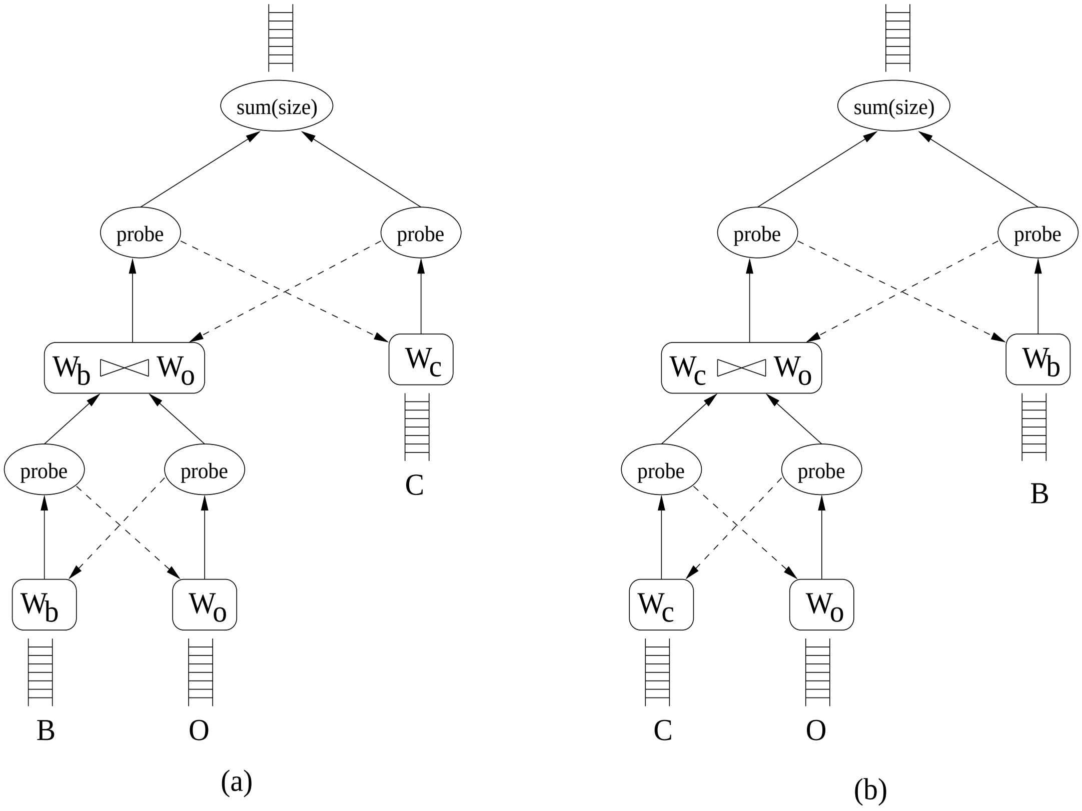


Figure 3.4: XJoin-based plans for the example network-monitoring query

XJoin uses a left-deep tree and maintains one intermediate join result—the join of windows W_b and W_o—indicated in the figure as $W_b \bowtie W_o$. Each new tuple $c \in C$ can now be joined with $W_b \bowtie W_o$ directly instead of being joined with W_b and W_o separately. Consequently, joining tuples in W_b and W_o need not be computed again and again when multiple tuples in stream C have the same join-attribute values. (Note that join plans in conventional DBMSs are usually trees of two-way join operators, like XJoins. To our knowledge, multiway join operators like MJoins are not used in conventional DBMSs.)

While XJoins can address the recomputation problem of MJoins, the extra run-time state carried by XJoins may hinder adaptivity when stream or system conditions change. For example, if stream B has a long burst of tuple arrivals, then we may want to switch from the XJoin in Figure 3.4(a) to the XJoin in Figure 3.4(b) because the latter can process B tuples efficiently. Such plan switching can be expensive if plans carry a lot of state [121]:

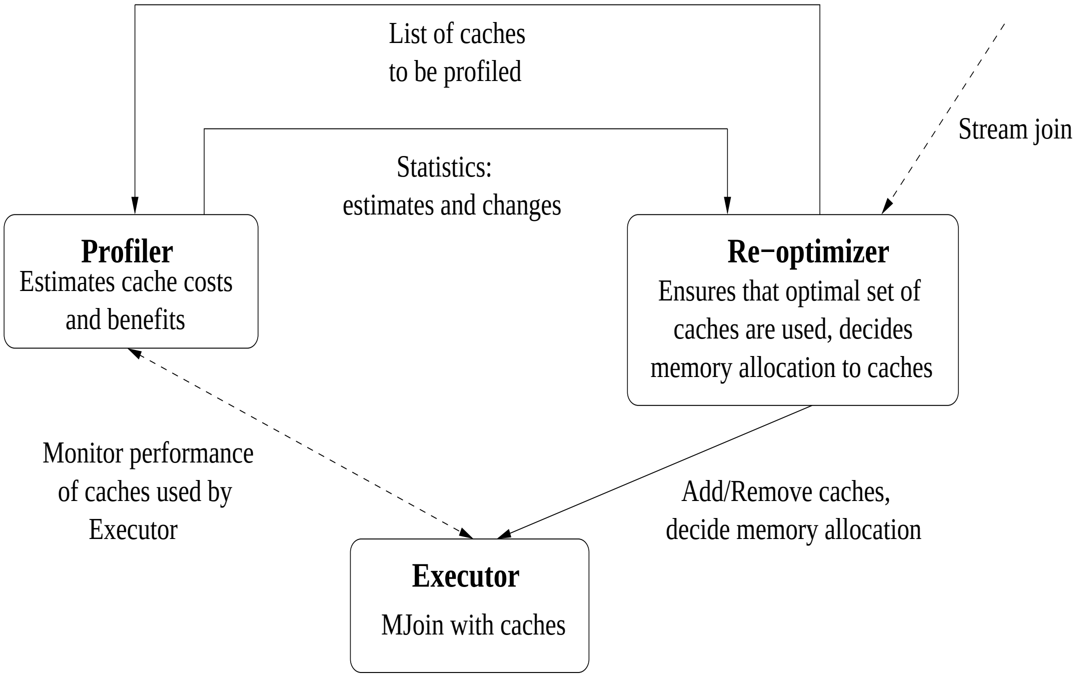


Figure 3.5: A-Caching

The state in the old plan has to be disposed (e.g., to reclaim memory) and the complete state in the new plan has to be generated and materialized before we can start processing new input tuples. In volatile stream and system conditions, it is important to keep the cost of switching between plans low, so as to avoid a significant pause in throughput when conditions change.

It follows from the above discussion that extra state in query plans may be required for good performance when stream and system conditions are steady, but extra state can increase the cost of switching between plans. We have developed a new approach for windowed stream joins, called *A-Caching*, that addresses these conflicting requirements of multiway windowed stream joins. A-Caching instantiates the three generic components of StreaMon as shown in Figure 3.5.

The plan used by A-Caching's Executor for a windowed stream join is basically an MJoin, but the MJoin pipelines may contain *join result caches*. A cache in a join pipeline corresponds to a segment of join operators in the pipeline. Intuitively, the cache stores join results computed by these operators in the recent past. If a tuple t being processed currently by the pipeline has the same join-attribute values as a tuple t' processed in the recent past, then the cache enables us to save the regular join computation for t: the tuples joining with t can be simply looked up in the cache where they would have been placed when t' was processed.

Adding caches to MJoin pipelines gives many advantages:

1. The resulting plan space captures a large spectrum of windowed stream join plans that includes both MJoins and XJoins as well as arbitrary combinations of MJoins and XJoins.

2. Unlike the fully-materialized state in XJoins, caches can be added, populated incrementally, and dropped with little overhead. Therefore, the cost of switching from one MJoin with caches to another is low, enabling fast adaptivity when stream or system conditions change.

3. Unlike XJoins, the performance of MJoins with caches degrades gracefully when memory availability decreases.

However, it is a nontrivial problem to add and remove caches adaptively in the MJoin pipelines used by the Executor, based on stream and system conditions. This problem is addressed collectively by A-Caching's Re-optimizer and Profiler as illustrated in Figure 3.5.

A-Caching's Profiler monitors the *cost* and *benefit* of using different caches in MJoin pipelines. Intuitively, the benefit we get from using a cache in a join pipeline is the unit-time cost of the pipeline without the cache minus the unit-time cost with the cache. The cost of using a cache in a join pipeline is the unit-time cost of ensuring that the cached contents are current. If a cache is actually being used by the Executor in a pipeline as part of regular join processing, then its cost and benefit can be observed directly. It is more challenging to estimate the cost and benefit of a cache that is not being used, as we discuss in Chapter 5.

A-Caching's Re-optimizer is responsible for ensuring that the set of caches used in the join pipelines at a point of time is the best for current conditions. This *cache selection* *problem* is NP-Hard. Details of the problem as well as algorithms for it are described in Chapter 5. Furthermore, since the memory in a DSMS must be partitioned among all active continuous queries [29, 88], it may be that we do not have sufficient memory to store all caches selected by our cache selection algorithm. Therefore, the Re-optimizer is also responsible for dynamically allocating memory to the caches to maximize overall benefit under constraints on memory availability.

Chapter 5 will describe A-Caching in detail. Next we describe our instantiation of StreaMon to minimize memory requirements for windowed stream join plans.

3.4 Adaptive Memory Minimization using k-Constraints

Query plans for continuous queries often need to maintain significant run-time state in memory, which limits the number of queries the DSMS can process simultaneously. Consider our example network-monitoring query used in the previous section. An MJoin-based plan to execute this query is shown in Figure 3.3. This plan has to store all tuples in the ten-minute sliding window over each stream because an incoming tuple in a stream can join with any tuple in the windows over the other streams. Therefore, the total memory requirement for this plan is roughly the sum of the sizes of the three windows (plus extra memory for hash indexes on the windows), which can be quite large if the streams have high arrival rates.

Recall from Section 1.3 in Chapter 1 that the input streams C, B, and O involved in this query exhibit some interesting properties. First, the packets monitored by the query flow through link C to link B to link O. Thus, a tuple corresponding to a specific `pid` appears in stream C first, then a joining tuple may appear in stream B, and lastly in stream O. Second, if the latency of the network between links C and B and between links B and O is bounded by d_{cb} and d_{bo} respectively, then a packet that flows through links C, B, and O will appear in stream B no later than d_{cb} time units after it appears in stream C, and in stream O no later than d_{bo} time units after it appears in stream B. As we showed in Section 1.3, these two properties can be exploited to reduce the overall memory requirement for the query plan significantly. For example, when a tuple b arrives in stream B and no joining tuple exists in the window on C, b can be discarded immediately because a tuple in C joining with b should have arrived before b.

This example shows the importance of detecting and exploiting stream properties to reduce memory requirements in plans for continuous queries. Based on a study of several stream-based applications, we identified a set of basic *constraints* that individually or in combination capture the majority of properties useful for memory reduction in continuous queries [102]. For example, one of the basic constraints we identified is *ordering*, where the arrival of stream tuples is in increasing (or decreasing) order of the value of one or more attributes.

It is unrealistic to expect streams to satisfy strict constraints at all times, due to variability in data generation, network load, scheduling, and other factors. However, streams frequently come close to satisfying constraints, and we want to be able to take advantage of these situations. To address this problem, we developed the notion of *k-constraints*, where $k \geq 0$ is an *adherence parameter* capturing the degree to which a stream or joining pair of streams adheres to the strict interpretation of the constraint. The constraint holds with its strict interpretation when $k = 0$. For example, k*-ordering* specifies that out-of-order stream tuples are no more that k tuples apart.

We developed a query-processing architecture, called *k-Mon*, that can detect useful k-constraints automatically in input data streams and exploit these constraints to reduce run-time state in plans for continuous queries. Since stream properties may change over time, it is important for k-Mon to take an adaptive approach to query processing. Therefore, k-Mon instantiates the generic StreaMon architecture as shown in Figure 3.6.

Continuous queries are registered with k-Mon's Re-optimizer, which generates an initial STREAM query plan based on any currently known k-constraints on the input streams in the query. The Executor begins executing this plan. At the same time, the Re-optimizer informs the Profiler about constraints that may be used to reduce the memory requirement for this query.

The Profiler monitors input streams continuously and informs the Re-optimizer whenever k values for potentially-useful constraints change. As usual, we combine constraint monitoring with query execution whenever possible to reduce the monitoring overhead. The Re-optimizer adapts to the changes detected by the Profiler by adjusting the k values used by the Executor for memory reduction. Obviously if a k value is very high (e.g., when a constraint does not hold at all, k grows without bound), the memory reduction obtained

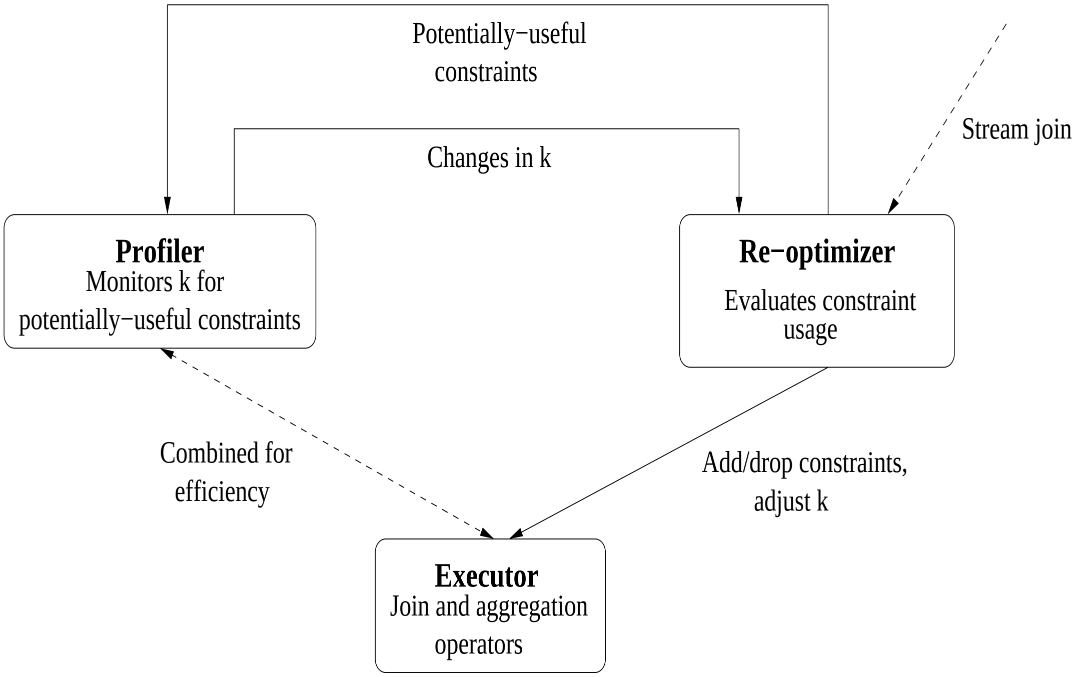

Potentially−useful constraints
Changes in k
Stream join
Profiler
Monitors k for potentially−useful constraints
Re−optimizer
Evaluates constraint usage
Combined for efficiency
Add/drop constraints, adjust k
Executor
Join and aggregation operators

Figure 3.6: k-Mon

from using the constraint may not be large enough to justify the extra computational cost. Therefore, the Executor ignores constraints with k values higher than some threshold.

The query execution algorithm used by k-Mon's Executor assumes adherence to k-constraints within the values for k given by the Profiler. Based on these properties, the algorithm identifies and eliminates run-time state that is not necessary for processing future stream tuples, reducing the overall memory requirement. For example, the packets monitored by our example join query flow through link C to link B. Thus, a tuple c corresponding to a specific pid will arrive in stream C before its joining tuple b (b.pid = c.pid) arrives in stream B. Also, c will not join with any tuple in B that arrives after b. Therefore, the tuples c and b can be discarded after the join results generated by them are produced, reducing the size of the stored windows. In this scenario, k-Mon will detect that the join between streams C and B is *one-one*, and that a stream-based *referential-integrity constraint* holds on this join with $k = 0$; full details are provided in Chapter 6. Based on the detected k-constraints, k-Mon will automatically give the memory reduction we described above.

k-Mon's Executor may not always produce accurate query results. Specifically, if the Profiler underestimates k for some constraint, particularly if k increases rapidly, then during query execution some state may be discarded that would otherwise be saved if the true (higher) k value was known. In this scenario, *false negatives* (missing tuples) can occur in the results produced for windowed stream joins. In many stream applications, modest inaccuracy in query results is an acceptable tradeoff for more efficient execution [44], especially if the inaccuracy persists for only short periods. Our example network-monitoring query clearly has this property. Nevertheless, the monitoring algorithms used by k-Mon's Profiler incorporate techniques to ensure that the inaccuracy in query results is minimized. We defer the details to Chapter 6 where we give the complete description of k-Mon.

3.5 Related Work

A very recent survey of systems and techniques for adaptive query processing is given in [16]. 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 [26, 105]. Two other approaches [71, 74] 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 [70, 72]: 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 [70, 72] do not extend to continuous queries and provide no guarantees on convergence. Reference [113] explores the idea of moving execution to different parts of a query plan adaptively when input relations brought from remote nodes incur high latency.

The *POP* approach adds *checks* to conventional query plans in DBMSs to detect suboptimalities during query execution, invoking re-optimization if required [85]. The *Rio* system proposes *proactive re-optimization* where the uncertainty in input statistics is taken into account to choose plans that are robust to estimation errors, and also to drive the collection of accurate statistics during query execution using random-sample processing techniques [17].

The novel Eddies architecture [10, 30, 41, 84, 94, 110] enables fine-grained adaptivity by eliminating query plans, instead *routing* each tuple adaptively across operators that need to process it. StreaMon's overall approach differs from Eddies in two significant ways:

1. StreaMon 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.

2. Recall from Section 3.1 that algorithms in StreaMon are designed around a three-way tradeoff in adaptive query processing among the three important metrics: run-time overhead, speed of adaptivity, and convergence to good execution plans.

Nevertheless, some of our adaptive algorithms can be used as routing schemes in Eddies.

Apart from Eddies, the CAPE [121] and Gigascope [38] DSMSs support adaptive query processing over data streams. Like StreaMon, 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. Unlike StreaMon, these systems do not take into account the tradeoff among the three metrics for adaptive query processing; e.g., see Sections 4.3–4.6 in Chapter 4.

There has been plenty of work on building mining models over continuous data streams, including clustering [1, 58], decision trees [45, 68], nearest neighbors [77], and heavy hitters [36, 37]. New algorithms have also been proposed for maintaining statistics over data streams, e.g., samples [116], histograms [40], and quantiles [7]. These algorithms can be used by StreaMon's Profiler for maintaining statistics about current stream and system conditions. Currently, the Profiler uses low-overhead techniques based on sampling, e.g., see Section 4.3.1 in Chapter 4.

3.6 Conclusion

Conventional DBMSs use a plan-first execute-next approach to process a query Q: the query optimizer enumerates candidate query plans for Q, estimates the execution cost of each candidate plan based on the current data and system conditions, chooses the plan with the lowest estimated cost, and runs this plan to completion. However, this approach can cause severe performance degradation in a DSMS where stream and system conditions may vary significantly over the lifetime of a long-running continuous query. A plan that is most efficient for a continuous query Q at a point in time may become very inefficient to process Q when stream or system conditions change. Consequently, it is important for a DSMS to change the execution plan for a continuous query while the query is running, based on how stream and system conditions change. Without such adaptive query processing, plan performance may drop drastically over time.

In this chapter we presented an overview of our generic StreaMon framework for adaptive query processing in a DSMS. We presented the three components of StreaMon and described the interactions among these components. We then described three instantiations of StreaMon. In each case we instantiated the generic components of StreaMon in specific ways to address a particular continuous query type and adaptivity need.

Our first instantiation of StreaMon, called A-Greedy, was for a very common and relatively simple class of continuous queries called pipelined filters. We described how A-Greedy maintains the order in which the filters are evaluated over incoming stream tuples so that the order is efficient for the current stream and system conditions. Our second and third instantiations of StreaMon—A-Caching and k-Mon—were for a more complex class of continuous queries, namely, multiway windowed stream joins. A-Caching places subresult caches adaptively in MJoin pipelines, enabling our windowed stream join plans to adapt over the entire spectrum between subresult-free MJoins and conventional tree-shaped join plans with materialized subresults at every intermediate node. Also, caches are advantageous from the adaptivity perspective because they can be added, populated incrementally, and dropped with little overhead. k-Mon detects useful k-constraints automatically over input data streams and exploits these constraints to reduce run-time state in query plans for continuous queries.

All our three instantiations follow the same overall structure of StreaMon. However, the algorithms used by the Re-optimizer, the Profiler, and the Executor in each case are specific to the continuous query type and adaptivity need being addressed. Also, these three instantiations differ to some extent in the interactions among the three components.

For example, the Re-optimizer and the Profiler are more closely integrated in A-Greedy than in A-Caching or k-Mon. The closer interaction helps A-Greedy to compute the new best plan incrementally after a change in stream or system conditions, while the others have to repeat plan enumeration and costing. While more details will be provided in the respective chapters, intuitively, the tighter coupling between these components in A-Greedy is possible because of the simpler and smaller-sized space of plans being considered.

The three combinations of continuous query type and adaptivity need that we considered for instantiating StreaMon were motivated by their occurrence in almost all the stream-based applications we studied [102] as well as by the relative simplicity of their problem definitions. There are many other combinations that could be considered for instantiating StreaMon. For example:

1. Aggregation over streams is a very common type of continuous query, so a DSMS can benefit from processing these queries adaptively [36]. One of the main challenges in instantiating StreaMon for this query type is to develop techniques that the Profiler can use to approximate the stream data distribution efficiently and with quality guarantees.

2. When continuous queries are very complex (e.g., with many subqueries), the space of plans StreaMon's Re-optimizer needs to consider, and the number of statistics the Profiler needs to monitor, become very large. This problem also arises when we consider the sharing of computation across multiple continuous queries [34], since the set of queries considered together are similar to considering one very complex query. In these scenarios, it is usually not feasible to enumerate all possible plans during re-optimization. For example, the Re-optimizer may now need to keep track of the history of plan selection, to enable low-overhead plan selection when re-optimization is invoked under statistical conditions seen previously. In Chapter 7 we propose extensions to StreaMon to incorporate such techniques.

3. A relevant problem in the modern systems architecture context is to instantiate StreaMon for the adaptive processing of long-running continuous queries on clusters of workstations [9, 121]. The cluster setting generates a new set of plan choices that the Re-optimizer needs to consider, and a new set of time-varying parameters that

the Profiler needs to monitor. For example, the Re-optimizer must now consider how operators and synopses can be placed on different machines for balancing the overall load, and how to handle variations in load. The Profiler now needs to monitor, e.g., the number and capacity of machines, the load on machines, and the communication bandwidth across machines.

Chapter 4

Adaptive Ordering of Pipelined Stream Filters

The previous chapter described the generic StreaMon framework for adaptive query processing in a DSMS. We now describe our first instantiation of StreaMon to process a specific class of continuous queries adaptively. We consider a common class of continuous queries, called pipelined filters, where a stream of tuples is processed by a set of commutative filters. We focus on the problem of ordering the filters adaptively to minimize processing cost in an environment where stream and system conditions vary unpredictably over time. Our core algorithm for this problem, A-Greedy, follows the StreaMon framework introduced in the previous chapter. We describe how A-Greedy instantiates the Executor, Profiler, and Re-optimizer components of StreaMon.

We analyze A-Greedy in terms of the three-way tradeoff we identified in the previous chapter among convergence properties, run-time overhead, and speed of adaptivity. We then present a suite of variants of A-Greedy that lie at different points on this tradeoff spectrum. Finally, we present a thorough performance evaluation of our algorithms in the STREAM DSMS.

4.1 Introduction

A *pipelined filter query* Q is a conjunction of commutative filters $F_1, F_2, \ldots, F_n$ over a single data stream S. (Recall that if a set of filters is commutative, then the result of evaluating any of the filters on a tuple is independent of the order in which the filters are evaluated on the tuple.) Q would be represented in CQL as:

Select *
From S
Where F_1 and F_2 and $\cdots$ and F_n

Each filter operates on one S tuple at a time and evaluates to true or false. The result of Q is a stream containing all S tuples for which all the filters $F_1, F_2, \ldots, F_n$ evaluate to true.

The STREAM system uses a plan containing a single `select` operator to process this filter query, as shown in Figure 4.1. To process an input tuple $s \in S$, the `select` operator will evaluate the filters on s one by one in some chosen order. This evaluation continues until a filter evaluates to false on s, which means that s is not in the query result, or all filters evaluate to true on s, which means that s is in the query result.

4.1.1 Motivating Example

Consider the following example continuous query from a network intrusion detection application in an enterprise [23, 101]:

> *Track packets with destination address matching a prefix in a table T, and containing the 100-byte and 256-byte sequences "0xa. . .8" and "0x7. . .b" respectively as subsequences.*

This query can be expressed in CQL as follows:

Select *
From Packets P
Where *lookup*(P.destAddr, T) = true and *match*(P.data, "0xa. . .8") = true and
match(P.data, "0x7. . .b") = true

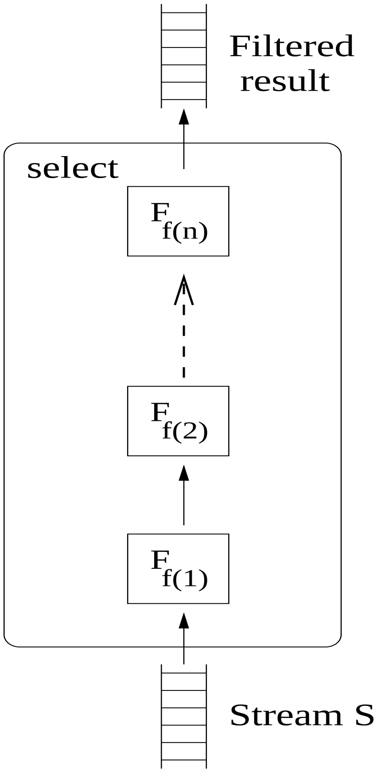


Figure 4.1: Plan to process a pipelined filter query involving filters $F_1, F_2, \ldots, F_n$ over stream S. $f(1), f(2), \ldots, f(n)$ is a permutation of $1, 2, \ldots, n$

In this representation, the *lookup*(a, T) function returns true for packets whose destination address a matches one of the addresses in table T. The *match*(I, p) function returns true if the pattern p occurs at least once in the input string I.

This continuous query is trying to detect a specific type of attack on the enterprise's network. The indicator of this attack is the arrival of network packets containing a well-defined *signature* that are traveling to the machines specified in the table T. Some recent network attacks have targeted database servers [23], so T may contain the addresses of the machines running database servers in the enterprise. The 100-byte and 256-byte sequences represent the signature of packets constituting the attack. For example, this attack may be exploiting a bug in a database server where the server behaves in an uncharacteristic manner when an input contains these two sequences.

This example query Q is a pipelined filter query over the stream of network packets. Note that Q is a conjunction of three commutative filters, each of which is an expensive user-defined function. Because of filter commutativity, the query result will be the same regardless of the order in which these three filters are evaluated on the incoming network packets. However, overall processing costs for Q can depend on how the filters are ordered, and the best ordering can depend on current stream and filter conditions.

Let us denote the *lookup* filter in the query as F_1, and the two *match* filters as F_2 and F_3 respectively. Let d_i denote the probability that filter F_i will *drop* an incoming packet, i.e., d_i is the probability that filter F_i will evaluate to false on a random packet. (Note that once a filter evaluates to false on a packet, we do not need to process the packet further.) Also, let t_i denote the average processing time per packet for filter F_i. Suppose $d_3 > d_1$ and $t_3 < t_1$ based on the current stream conditions. That is, the likelihood of filter F_3 dropping an incoming packet is higher than that for F_1, and F_3 is cheaper to evaluate then F_1. In this scenario it is clearly more efficient on average to evaluate F_3 before F_1 on incoming network packets.

Now suppose $d_2 > d_3 > d_1$ and $t_2 < t_3 < t_1$. Extending our above reasoning, the best filter ordering is F_2, F_3, F_1, i.e., F_2 is evaluated first on an incoming packet p, followed by F_3 if p passes F_2, and finally F_1 if p passes both F_2 and F_3. However, this ordering may be suboptimal compared to F_2, F_1, F_3 if filters F_2 and F_3 are correlated. (Recall from Chapter 3 that filters F_i and F_j over stream S are correlated if $d_{ij} \neq d_i \times d_j$, where d_{ij} is the probability that both F_i and F_j will evaluate to false on a random tuple in S.) For example, the probability that a packet contains the (attack) byte sequence “0x7. . .b” (from F_3) may be very high if we already know that the packet contains the byte sequence “0xa. . .8” (from F_2). Therefore, F_3 may drop almost none of the packets passed by F_2, making the filter evaluation order F_2, F_3, F_1 less efficient than F_2, F_1, F_3.

Now suppose the monitored attack starts on the enterprise’s network. Then, the values of d_2 and d_3 may drop suddenly, making the filter evaluation order F_1, F_2, F_3 much more efficient than the previously optimal F_2, F_1, F_3.

4.1.2 Filter Ordering

The above example motivates the problem of selecting good orderings for pipelined filters, and illustrates the challenges that we face. Specifically:

1. Ordering decisions are based on filter selectivities, and as we have seen, selectivities across filters may be nonindependent, or correlated. In conventional DBMSs, correlations are known to be one of the biggest pitfalls of query optimization [35, 105]. We are able to make strong theoretical claims about our ordering algorithms, and achieve good performance in practice, by monitoring and taking into account correlated filter selectivities.

2. For n filters, an exhaustive algorithm must consider $n!$ possible orderings. This approach may be feasible for one-time optimization and small n, but it quickly becomes infeasible as n increases, and also is impractical for an online adaptive approach. In this chapter we present an ordering algorithm that is provably within a small constant factor of optimal for the unit-time cost metric, and that nearly always finds the optimal orderings in practice.

3. Stream data and arrival characteristics may change over time [30, 83]. In addition, since we are considering arbitrary filters which could involve, for example, joins with other data or network communication, filter selectivities and processing times also may change considerably over time, independent of the incoming stream characteristics. Overall, an ordering that is optimal at one point in time may be extremely inefficient later on. Our adaptive approach for handling this problem is discussed next.

4.1.3 Adaptive Ordering of Filters

Even with an effective algorithm for filter ordering, we face additional challenges in the continuous query environment when stream and filter characteristics may change considerably during the lifetime of a query.

- **Run-time overhead:** Suppose we have established a filter ordering based on collected data and processing-time statistics. To adapt to changes in stream and filter characteristics, we must monitor continuously to determine when statistics change. In particular, we must detect when statistics change enough that our current ordering is no longer consistent with the ordering that would be selected by the greedy algorithm. Since the greedy algorithm is based on correlated filter selectivities, in theory we must continuously monitor selectivities for all possible orderings of two or more filters to determine if a better overall ordering exists. We provide a variety of techniques for efficiently approximating these statistics.

- **Convergence properties:** An adaptive algorithm continuously changes its solution to a problem as the input characteristics change. Thus, it is difficult to prove anything concrete about the behavior of such an algorithm. One approach for making claims is to prove *convergence properties* of an adaptive algorithm: Imagine that data and filter characteristics stabilize; then the adaptive algorithm would converge to a solution with desirable properties. Our core adaptive algorithm, A-Greedy, has good convergence properties: If statistics were to stabilize, A-Greedy would provably converge to the same solution found by the static greedy algorithm using those statistics, which is provably within a small constant factor of optimal.

- **Speed of adaptivity:** Even if an adaptive algorithm has good convergence properties, it may adapt slowly. In reality statistics are not expected to stabilize. Thus, an algorithm that adapts slowly may constantly lag behind the current optimal solution, particularly in a rapidly-changing environment. On the other hand, an algorithm that adapts too rapidly may react inappropriately to transient situations. We show how A-Greedy balances adaptivity and robustness to transient situations.

It turns out we face a three-way tradeoff: Algorithms that adapt quickly and have good convergence properties require significantly more run-time overhead. If we have strict limitations on run-time overhead, then we must sacrifice either speed of adaptivity or convergence. We develop a suite of variants on our statistics collection scheme and our A-Greedy algorithm that lie at different points along this tradeoff spectrum.

This rest of this chapter proceeds as follows. Section 4.2 establishes notation for this chapter. In Section 4.3 we present the A-Greedy algorithm and analyze its convergence properties, run-time overhead, and speed of adaptivity. Sections 4.4–4.6 present variants of A-Greedy. Section 4.7 presents an experimental evaluation of our algorithms based on an implementation in the STREAM prototype DSMS. We discuss work related to pipelined filters in Section 4.9, and conclude in Section 4.10.

4.2 Notation for this Chapter

We are given a pipelined filter query Q that applies the conjunction of n commutative filters $F_1, F_2, \ldots, F_n$ over input stream S. Each filter F_i takes a stream tuple $s \in S$ as input and returns either true or false. If F_i returns false for tuple s, then we say that F_i *drops* s. Tuple s is emitted in the continuous query result if and only if all n filters return true for s.

The STREAM query plan for executing Q consists of a `select` operator with one input queue and one output queue, as shown in Figure 4.1. The `select` operator contains an ordering of the filters, denoted $O = F_{f(1)}$, $F_{f(2)}$, $\ldots$, $F_{f(n)}$. (We use f throughout this chapter to denote the mapping from positions in the filter ordering to the indexes of the filters at those positions.) The operator reads tuples in timestamp order from its input queue and processes them using O. When a tuple $s \in S$ is processed by O, first filter $F_{f(1)}$ is evaluated on s. If it returns false (s is dropped by $F_{f(1)}$), then s is not processed further. Otherwise, $F_{f(2)}$ is evaluated on s, and so on.

Consider $O = F_{f(1)}, F_{f(2)}, \ldots, F_{f(n)}$. We use $d(i|j)$, $1 \leq j < i \leq n$, to denote the conditional probability that $F_{f(i)}$ will drop a tuple s from input stream S, given that s was not dropped by any of $F_{f(1)}$, $F_{f(2)}$, $\ldots$, $F_{f(j)}$. The unconditional probability that $F_{f(i)}$ will drop an S tuple is $d(i|0)$. We use t_i to denote the expected time for F_i to process one tuple. Note that $d(i|j)$ and t_i are expected to vary over time as input characteristics change; we always refer to their current values. With this notation we can compute the expected time for O to process an incoming tuple in S to completion (either emitted or dropped), denoted T_O, as:

$$T_O = \sum_{i=1}^{n} t_i D_i, \text{ where } D_i = \begin{cases} 1 & i = 1 \\ \prod_{j=1}^{i-1}(1 - d(j|j-1)) & i > 1 \end{cases} \tag{4.1}$$

Therefore, the unit-time cost of O is *rate*$(S) \times T_O$, where *rate*(S) is the rate at which tuples arrive in stream S. Our goal is to maintain filter orderings that minimize this cost at any point in time. Since *rate*(S) is independent of the filter ordering, we do not need to take *rate*(S) into account to compute the ordering that has the minimum unit-time cost.

Example 4.2.1 Figure 4.2 shows an example pipelined filter query containing four filters F_1–F_4 over a stream S. Filter F_i contains a hash-indexed table of tuples T_i such that F_i

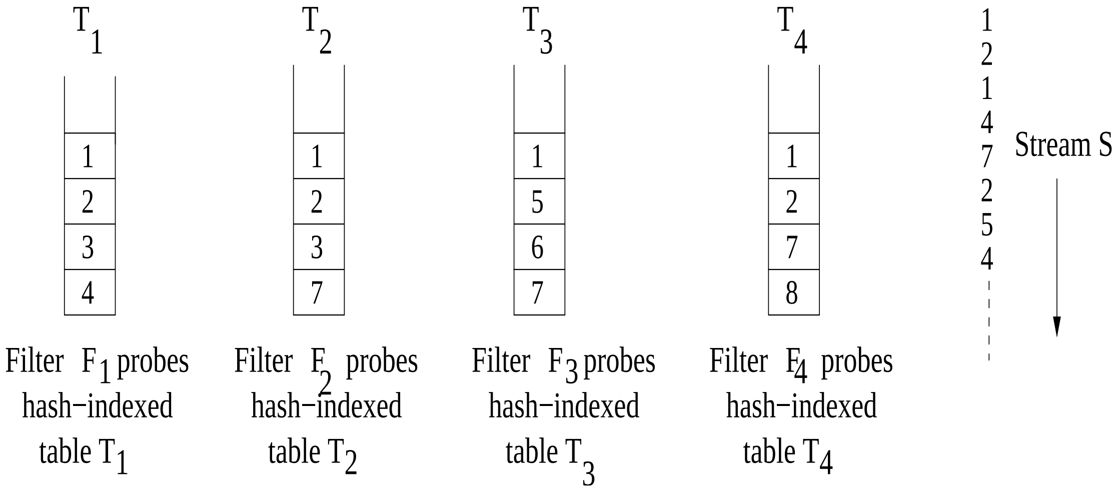

T1
1
2
3
4
T2
1
2
3
7
T3
1
5
6
7
T4
1
2
7
8
1
2
1
4
7
2
5
4
Stream S
Filter F1 probes hash−indexed table T1
Filter F2 probes hash−indexed table T2
Filter F3 probes hash−indexed table T3
Filter F4 probes hash−indexed table T4

Figure 4.2: Filters and sequence of input tuples in our example pipelined filter query

drops a tuple $s \in S$ if and only if T_i does not contain s. For example, when filter F_1 is evaluated on s, F_1 probes the hash index on table T_1. F_1 will drop s if s is not equal to one of the four tuples present in T_1.

Let us consider the number of hash probes required for different orderings to process the input sequence 1, 2, 1, 4, 7, 2, 5, 4, also shown in Figure 4.2. Note that all of these tuples except $s = 1$ are dropped by some filter in F_1–F_4. For $O_1 = F_1, F_2, F_3, F_4$, the total number of probes for the eight S tuples shown is 20. (For example, $s = 2$ requires three probes—F_1, F_2, and F_3—before it is dropped by F_3.) The corresponding number for $O_2 = F_3, F_2, F_4, F_1$ is 18, while $O_3 = F_3, F_1, F_2, F_4$ is optimal for this example at 16 probes. □

4.3 The A-Greedy Adaptive Filter Ordering Algorithm

In this section we develop A-Greedy, our core adaptive ordering algorithm for pipelined filters. Let us first consider a greedy algorithm based on stable statistics. Using our cost metric introduced in Section 4.2 and assuming for the moment uniform times t_i for all filters, a greedy approach to filter ordering proceeds as follows:

1. Choose the filter F_i with highest unconditional drop probability $d(i|0)$ as $F_{f(1)}$.

2. Choose the filter F_j with highest conditional drop probability $d(j|1)$ as $F_{f(2)}$.
3. Choose the filter F_k with highest conditional drop probability $d(k|2)$ as $F_{f(3)}$.
4. And so on.

To factor in varying filter times t_i, we replace $d(i|0)$ in step 1 with $d(i|0)/t_i$, $d(j|1)$ in step 2 with $d(j|1)/t_j$, and so on. We refer to this ordering algorithm as *Static Greedy*, or simply *Greedy*. We will see later that this algorithm has strong theoretical guarantees and very good performance in practice. Greedy maintains the following *Greedy Invariant* (*GI*):

Definition 4.3.1 *(***Greedy Invariant***)* $F_{f(1)}, F_{f(2)}, \ldots, F_{f(n)}$ *satisfies the Greedy Invariant if:*

$$\frac{d(i|i-1)}{t_{f(i)}} \geq \frac{d(j|i-1)}{t_{f(j)}}, 1 \leq i \leq j \leq n$$

□

The goal of our adaptive algorithm A-Greedy is to maintain, in an online manner, an ordering that satisfies the GI. In the next two sections we describe how A-Greedy achieves this goal by instantiating the Profiler and Re-optimizer components of the generic StreaMon framework from Chapter 3.

4.3.1 The A-Greedy Profiler

To maintain the GI, the Re-optimizer needs continuous estimates of (conditional) filter selectivities, as well as tuple-processing times. We consider each in turn.

Selectivity Estimates

Our greedy approach is based on $d(i|j)$ values, which denote the conditional probability that $F_{f(i)}$ will drop a tuple e, given that e was not dropped by any of $F_{f(1)}$, $F_{f(2)}$, …, $F_{f(j)}$. Selectivity generally refers to the inverse of drop probability—that is, the selectivity of filter $F_{f(i)}$ executed after $F_{f(1)}, F_{f(2)}, \ldots, F_{f(j)}$ is $1 - d(i|j)$, or the probability of a tuple passing the filter. For n filters, the total number of conditional selectivities is $n2^{n-1}$.

Clearly it is impractical for the Profiler to maintain online estimates of all these selectivities. Fortunately, to check whether a given ordering satisfies the GI, we need to check $(n+2)(n-1)/2 = O(n^2)$ selectivities only. Thus, by monitoring $O(n^2)$ selectivities, we can detect when statistics change sufficiently that the GI is violated by the current ordering.

Once a GI violation has occurred, to find a new ordering that satisfies the GI we may need $O(n^2)$ new selectivities in the worst case. Furthermore, the new set of required selectivities depends on the new input characteristics, so it cannot be predicted in advance. Our approach is to balance run-time monitoring overhead and the extra computation required to correct a violation: The Profiler does not estimate filter selectivities directly. Instead, it maintains a structure called the *profile window* that captures succinctly the statistics of tuples dropped in the recent past. The Re-optimizer can compute required selectivity estimates from the profile window. As we will see in Section 4.3.2, the Re-optimizer maintains a view over the profile window for incremental monitoring of conditional selectivities.

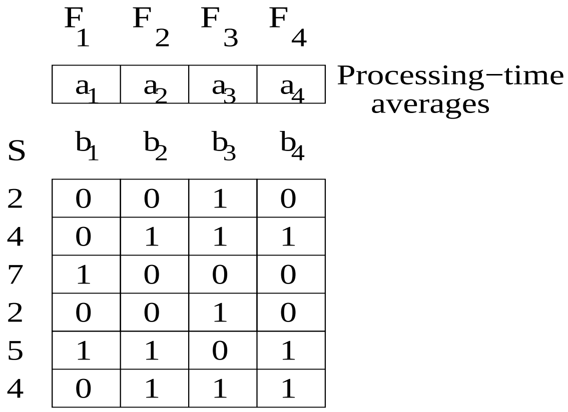
Specifically, the profile window is a sliding window of *profile tuples* created by sampling tuples from input stream S that get dropped during filter processing. A profile tuple contains n boolean attributes $b_1, \ldots, b_n$ corresponding to filters $F_1, \ldots, F_n$. Profile tuples are created as follows: When a tuple $s \in S$ is dropped during processing, s is profiled with probability p, called the *drop-profiling probability*. If s is chosen for profiling, processing of s continues artificially to determine whether any of the remaining filters unconditionally drop s. The Profiler then logs a tuple with attribute $b_i = 1$ if F_i drops s and $b_i = 0$ otherwise, where $1 \leq i \leq n$.

The profile window is maintained as a sliding window so that older input data does not contribute to statistics used by the Re-optimizer. This window could be time-based, e.g., tuples in the last 5 minutes, or tuple-based, e.g., the last 10,000 tuples. The window must be large enough so that:

1. Statistics can be estimated with high confidence.

2. Re-optimization is robust to transient bursts.

In our current implementation, we fix a single drop-profiling probability and profile window size for each pipelined-filter query. There may be advantages to modifying these parameters adaptively, which is outside the scope of this thesis.



S	b_1	b_2	b_3	b_4
2	0	0	1	0
4	0	1	1	1
7	1	0	0	0
2	0	0	1	0
5	1	1	0	1
4	0	1	1	1

Figure 4.3: Profile window for our example pipelined filter query

Example 4.3.1 Figure 4.3 shows the profile window for the example in Figure 4.2 when the drop-profiling probability is 1, i.e., all dropped tuples are profiled, and the profile window is a tuple-based window of size 6. □

Processing-Time Estimates

In addition to conditional selectivities, the GI is based on the expected time t_i for each filter F_i to process a tuple. Since overhead is incurred to measure and record processing times, we want to measure processing times for only a sample of tuples. Furthermore, so that old statistics are not used by the Re-optimizer, we maintain a sliding window of processing-time samples. While the sampling rate and sliding window for processing-time estimates need not be the same as for selectivity estimates, currently we use the same profile tuples as for selectivity estimates, and record the times as n additional columns in the profile window (not shown in Figure 4.3). We use a_i to denote the average time for F_i to process a tuple, computed as the running average of processing-time samples for F_i.

4.3.2 The A-Greedy Re-optimizer

The Re-optimizer's job is to ensure that the current filter ordering satisfies the GI. Specifically, the Re-optimizer maintains an ordering O such that O satisfies the GI for statistics estimated from the tuples in the current profile window. For efficiency, the Re-optimizer incrementally maintains a specific view over the profile window. We describe this view next, then explain how the view is used to maintain the GI.

The Matrix View

The view maintained over the profile window is an $n \times n$ upper triangular matrix $V[i,j], 1 \leq i \leq j \leq n$, so we call it the *matrix view*. The n columns of V correspond in order to the n filters in O. That is, the filter corresponding to column c is $F_{f(c)}$. Entries in the i^{th} row of V represent the conditional selectivities of filters $F_{f(i)}, F_{f(i+1)}, \ldots, F_{f(n)}$ for tuples that are not dropped by $F_{f(1)}, F_{f(2)}, \ldots, F_{f(i-1)}$. Specifically, $V[i,j]$ is the number of tuples in the profile window that were dropped by $F_{f(j)}$ among tuples that were not dropped by $F_{f(1)}, F_{f(2)}, \ldots, F_{f(i-1)}$. Notice that $V[i,j]$ is proportional to $d(j|i-1)$.

The Re-optimizer maintains the ordering O such that the matrix view for O always satisfies the condition:

$$\frac{V[i,i]}{a_{f(i)}} \geq \frac{V[i,j]}{a_{f(j)}}, 1 \leq i \leq j \leq n \tag{4.2}$$

By the definition of V, Equation (4.2) is equivalent to saying that O satisfies the GI for the statistics estimated from the tuples in the profile window. If the estimated $V[i,j]$ and a_i represent the real $d(j|i-1)$ and t_i respectively, then O satisfies the GI.

Example 4.3.2 Consider the scenario in Figure 4.2 and assume that all measured processing times are equal (not unreasonable for hash-probes into similarly-sized tables), so $a_i = 1, 1 \leq i \leq n$. Figure 4.4 shows the matrix view over the profile window in Figure 4.3. Ordering $O = F_3, F_1, F_2, F_4$ satisfies the GI. Recall from Example 4.2.1 that we computed this ordering as having the lowest number of hash probes to process the example input stream. □

Matrix view V must be updated as profile tuples are inserted into and deleted from the profile window. Figure 4.5 contains the pseudocode for updating V when a new profile tuple is recorded. The same code is used for deletion of a profile tuple, with line 10 replaced by $V[r,c] = V[r,c] - 1$.

F_3	F_1	F_2	F_4
4	2	3	3
	2	1	1
		0	0
			0

Figure 4.4: Matrix view for our example pipelined filter query

```
/** Input: Profile tuple $\langle b_1, \ldots, b_n \rangle$ inserted into the profile window */
/** Find the first filter $F_{f(dc)}$ that dropped the corresponding tuple */
$dc = 1$;
while ($b_{f(dc)} == 0$)
    $dc = dc + 1$;
/** Increment $V$ entries for filters that dropped the tuple */
for ($c = dc$; $c \leq n$; $c = c + 1$)
    if ($b_{f(c)} == 1$)
        for ($r = 1$; $r \leq dc$; $r = r + 1$)
            $V[r, c] = V[r, c] + 1$;
```

Figure 4.5: Updating V on an insert to the profile window

Violation of the Greedy Invariant

Suppose an update to the matrix view or to a processing-time estimate causes the following condition to hold:

$$\frac{V[i,i]}{a_{f(i)}} < \frac{V[i,j]}{a_{f(j)}}, 1 \leq i < j \leq n \tag{4.3}$$

Equation (4.3) states that $F_{f(j)}$ has a higher ratio of drop probability to processing time for tuples that were not dropped by $F_{f(1)}, F_{f(2)}$, . . ., $F_{f(i-1)}$ than $F_{f(i)}$ has. Thus, $O =$ $F_{f(1)}$, . . ., $F_{f(i)}, \ldots, F_{f(j)}, \ldots, F_{f(n)}$ no longer satisfies the GI. We call this situation a *GI violation at position* i. An update to V or to an a_i can cause a GI violation at position i either because it reduces $V[i,i]/a_{f(i)}$, or because it increases some $V[i,j]/a_{f(j)}$, $j > i$. Figures 4.6 and 4.7 contain pseudocode to detect and to correct, respectively, violations of GI at position i.

```
/** $old, new$: Values of $V[i,j]/a_{f(j)}$ before and after an update */
if ($new > old$ and $i < j$ and $V[i,i]/a_{f(i)} < new$)
    Violation at position $i$, so invoke Figure 4.7 with input $i$;
if ($new < old$ and $i == j$)
    for ($c = i + 1$; $c \leq n$; $c = c + 1$)
        if ($new < V[i,c]/a_{f(c)}$) {
            Violation at position $i$, so invoke Figure 4.7 with input $i$;
            return; }
```

Figure 4.6: Detecting a violation of the Greedy Invariant

Example 4.3.3 From Example 4.3.2 and Figure 4.4, O = F_3, F_1, F_2, F_4 satisfies the GI. Let the next two tuples arriving in S be 3 and 6 respectively, both of which will be dropped. Since the drop-profiling probability is 1, both 3 and 6 are profiled, expiring the earliest two profile tuples in the 6-tuple window. The resulting profile window is shown in Figure 4.8(a). Figure 4.8(b) shows the updated matrix view, which indicates a GI violation at position 1. (Recall that processing times are all 1.) The new ordering F_4, F_3, F_1, F_2 satisfying the GI and the corrected matrix view are shown in Figure 4.8(c). □

Discussion

From the algorithm in Figure 4.7 we see that if we reorder the filters such that there is a new filter at position i, then we may need to reevaluate the filters at positions $> i$ because their conditional selectivities may have changed. Each reevaluation requires a full scan of the profile window to compute the required conditional selectivities. Thus, the algorithm in Figure 4.7 may require anywhere from 1 to $n - i$ scans of the profile window, depending on the changes in input characteristics.

The adaptive ordering can thrash if both sides of Equation (4.2) are almost equal for some pair of filters. To avoid thrashing, we flag a violation and invoke the algorithm in Figure 4.7 only if:

$$\frac{V[i,i]}{a_{f(i)}} < \alpha \frac{V[i,j]}{a_{f(j)}}, 1 \leq i < j \leq n \tag{4.4}$$

```
/** Input: $O$ violates the Greedy Invariant at position $i$ */
$maxr = i$;
for ($r = i$; $r \leq maxr$; $r = r + 1$) {
    /** Greedy Invariant holds at positions $1, \ldots, r - 1$ and at
     *  $maxr + 1, \ldots, n$. Compute $V$ entries for row $r$ */
    for ($c = r$; $c \leq n$; $c = c + 1$)
        $V[r, c] = 0$;
    for each profile tuple $\langle b_1, b_2, \ldots, b_n \rangle$ in the profile window {
        /** Ignore tuples dropped by $F_{f(1)}, F_{f(2)}, \ldots, F_{f(r-1)}$ */
        if ($b_{f(1)} == 0$ and $b_{f(2)} == 0$ and $\cdots$ and $b_{f(r-1)} == 0$)
            for ($c = r$; $c \leq n$; $c = c + 1$)
                if($b_{f(c)} == 1$) $V[r, c] = V[r, c] + 1$;
    }
    /** Find the column $maxc$ with maximum $\frac{V[r,maxc]}{t_{f(maxc)}}$ */
    $maxc = r$;
    for ($c = r + 1$; $c \leq n$; $c = c + 1$)
        if ($V[r, c]/a_{f(c)} > V[r, maxc]/a_{f(maxc)}$) {
            $maxc = c$;
            if ($maxc > maxr$)    $maxr = maxc$;
        }
    if ($r \neq maxc$) {
        /** Current filter $F_{f(maxc)}$ becomes the new $F_{f(r)}$.
         *  We swap the filters at positions $maxc$ and $r$ */
        for ($k = 0$; $k \leq r$; $k = k + 1$)
            Swap $V[k, r]$ and $V[k, maxc]$;
    }
```

Figure 4.7: Correcting a violation of the Greedy Invariant

Parameter $\alpha \leq 1$, called the *thrashing-avoidance parameter*, is a constant to reduce the possibility of thrashing. α has a theoretical basis as we show in the next section.

We now analyze the behavior of A-Greedy with respect to the three important properties introduced in Section 4.1: guaranteed convergence to a good plan in the presence of a stable environment, run-time overhead to achieve this guarantee, and speed of adaptivity as statistics change.

S	b_1	b_2	b_3	b_4
7	1	0	0	0
2	0	0	1	0
5	1	1	0	1
4	0	1	1	1
3	0	0	1	1
6	1	1	0	1

Profile window

(a)

F_3	F_1	F_2	F_4
3	3	3	4
	3	2	2
		0	0
			0

Matrix view V
(Violation in first row)

(b)

F_4	F_3	F_1	F_2
4	3	3	3
	1	1	0
		1	0
			0

Matrix view V
(After correction)

(c)

Figure 4.8: Violation of the Greedy invariant

4.3.3 Convergence Properties

We say the stream and filter characteristics have *converged* when the following three conditions hold over a long interval of time:

C_1: The data distribution of stream tuples is constant.

C_2: For every subset of filters $F_1, F_2, \ldots, F_n$, the data distribution of input tuples passing all of the filters in the subset is constant.

C_3: The expected processing time t_i for each filter F_i is constant.

When stream and filter characteristics converge, A-Greedy soon produces the same filter ordering that would be produced by the Static Greedy algorithm (Section 4.3), namely the ordering that guarantees the GI (Definition 4.3.1). We prove that this ordering is within a small constant factor of the optimal ordering according to our cost metric.

Theorem 4.3.1 *When stream and filter characteristics converge, the cost of a filter ordering satisfying the GI is at most four times the cost of the optimal filter ordering.*

Proof: The proof proceeds to show that when stream and filter characteristics converge, that is, when Conditions C_1–C_3 above hold, the filter ordering problem and Static Greedy reduce respectively to the pipelined set cover problem and a greedy approximation algorithm for it from [89].

In an instance $P(E, W, U, C)$ of pipelined set cover, we are given a set $E = \{e_1, e_2, \ldots, e_m\}$ of m (possibly infinite) weighted tuples with respective weights $W = \{w_1, w_2, \ldots, w_m\}$. Also given are n sets $U = \{S_1, S_2, \ldots, S_n\}$ with respective unit-weight processing costs $C = \{c_1, c_2, \ldots, c_n\}$. Each set contains zero or more tuples in E. The solution desired for $P(E, W, U, C)$ is a pipeline $S_{f(1)}, S_{f(2)}, \ldots, S_{f(n)}$ of sets in U that minimizes the *pipelined cost* of processing E:

$$\sum_{i=1}^{n} c_{f(i)} \sum_{j=1}^{m} \{w_j | e_j \notin \bigcup_{l=1}^{i-1} S_{f(l)}\} \tag{4.5}$$

Intuitively, each set S_i in the pipeline drops the tuples in E that S_i contains from further processing. Thus, S_i processes those tuples in E that are not dropped before S_i appears in the pipeline. S_i contributes $c_i W_i$ to the total pipelined cost, where W_i is the total weight of tuples it processes.

We show how the problem of finding the optimal ordering of n filters $F_1, \ldots, F_n$ on stream S when stream and filter characteristics converge, reduces to solving $P(E, W, U, C)$. By Condition C_1, S can be reduced to a weighted set of m tuples with weights equal to their probability of arrival in S, corresponding to E and W respectively. By Conditions C_2 and C_3, filters $F_1, \ldots, F_n$ with tuple-processing times $t_1, \ldots, t_n$ correspond to the input sets $U = \{S_1, S_2, \ldots, S_n\}$ and their costs $C = \{c_1, c_2, \ldots, c_n\}$. From Equation 4.1, the cost of an ordering O is the sum of processing times for the filters with each filter F_i processing the tuples that are not dropped by any filter before F_i in O. Thus, the cost of $F_{f(1)}, F_{f(2)}, \ldots, F_{f(n)}$ is equivalent to the pipelined cost of the corresponding pipeline of sets $S_{f(1)}, S_{f(2)}, \ldots, S_{f(n)}$.

Reference [89] presents a greedy approximation algorithm for $P(E, W, U, C)$ which builds the pipeline one set at a time. (Note that pipelined set cover is an offline problem where the input is given in advance.) Suppose the algorithm has added sets $S_{f(1)}, S_{f(2)}, \ldots, S_{f(j-1)}$ in order to the pipeline so far. At position j in the pipeline, the algorithm chooses the set $S_i \in \{S_1, S_2, \ldots, S_n\} - \{S_{f(1)}, S_{f(2)}, \ldots, S_{f(j-1)}\}$, that maximizes the ratio W_i/c_i, where $W_i = \sum_{k=1}^{m}\{w_k | \; w_k \in S_i - \cup_{l=1}^{j-1} S_{f(l)}\}$. Intuitively, S_i maximizes the total weight of newly covered tuples per unit processing time. Note that W_i/c_i for $P(E, W, S, C)$ is equivalent to the ratio $d(i|j-1)/t_i$ which determines the filter at position j in Greedy.

Reference [89] shows that the greedy approximation algorithm for $P(E,\ W,\ U,\ C)$ produces a pipeline whose cost is at most four times the cost of the optimal pipeline. Since we have shown that the ordering produced by Greedy (that satisfies the GI) when input characteristics converge is equivalent to the pipeline produced by the greedy approximation algorithm for $P(E, W, S, C)$, and that costs of solutions are directly comparable between the problems, the same guarantee holds for Greedy as well. □

The key to the above proof was showing that the filter ordering problem is equivalent to the pipelined set cover problem from [89]. The equivalence of pipelined filters and pipelined set cover brings out some interesting characteristics of our problem. Pipelined set cover is *MAX SNP-hard* [89], which implies that any polynomial-time ordering algorithm can at best provide a constant-factor approximation guarantee for this problem. Reference [47] shows that the factor 4 in Theorem 4.3.1 is tight. However, a nice property shown in [89] is that the constant factor depends on problem size, e.g., for 20, 100, and 200 filters the bound is 2.35, 2.61, and 2.8 respectively; the theoretical constant 4 requires an infinite number of filters. Irrespective of these theoretical bounds, our experiments in Section 4.7 show that A-Greedy usually finds the optimal ordering.

Recall from the end of Section 4.3 that we introduce a constant $\alpha \leq 1$ to avoid thrashing behavior, replacing equation (4.3) with equation (4.4) in determining whether the GI is violated. When this parameter is incorporated, the constant 4 in Theorem 4.3.1 is replaced with $\frac{4}{\alpha}$ [89]. Here too, the constant depends on the number of filters, and generally the optimal ordering is found in practice.

4.3.4 Run-time Overhead and Adaptivity

The overhead of adaptive ordering using A-Greedy is the run-time cost of the Profiler and the Re-optimizer. It can be divided into four components:

1. *Profile-tuple creation.* The Profiler creates profile tuples for a fraction of dropped tuples equal to the drop-profiling probability. To create a profile tuple for a tuple e dropped by $F_{f(i)}$, the Profiler needs to additionally evaluate the $n-i$ filters following

$F_{f(i)}$, recording whether the filter is satisfied and measuring the time for each filter to process e.

2. *Profile-window maintenance.* The Profiler must insert and delete profile tuples to maintain a sliding window. It also must maintain the running averages a_i of filter processing times as tuples are inserted into and deleted from the profile window.

3. *Matrix-view update.* The Re-optimizer updates V (Figure 4.5) whenever a profile tuple is inserted into or deleted from the profile window, accessing up to $n^2/4$ entries.

4. *Detection and correction of GI violations.* The Re-optimizer must detect violations (Figure 4.6) caused by changes in input characteristics, and correct them (Figure 4.7). To detect a violation on an update, up to n entries in V may need to be accessed, while it may require up to $n - i$ full scans of the profile window to correct a GI violation at position i.

A change in stream or filter characteristics that causes a GI violation will cause a violation of Equation (4.2) and will be detected immediately. Furthermore, A-Greedy can correct the violation immediately (Figure 4.7), because any additional conditional selectivities required to find the new filter ordering can be computed from the existing profile window. Thus, A-Greedy is a very rapidly adapting algorithm.

4.4 The Sweep Algorithm

As we have seen in the previous section, A-Greedy has very good convergence properties and extremely fast adaptivity, but it imposes nonnegligible run-time overhead. Thus, it is natural to consider whether we can sacrifice some of A-Greedy's convergence properties or adaptivity speed to reduce its run-time overhead. It is possible to do so, as we demonstrate through several variants of A-Greedy presented in this and the next two sections.

A-Greedy detects every violation of GI in an ordering O as soon as it occurs by continuously verifying the relative ordering for each pair of filters in O. Suppose instead that at a given point in time we only check for violations of GI involving the filter at one specific position j, with respect to any filter at a position $k < j$. By rotating j over $2, \ldots, n$, we can still detect all violations, although not as quickly as A-Greedy. This approach is taken by our first variant of A-Greedy, which we call *Sweep*.

Sweep proceeds in stages. During a stage, only the filter at a position j in O, i.e., $F_{f(j)}$, is profiled. (Details of profiling are given momentarily.) A stage ends either when a violation of GI at position j is detected, or when a prespecified number of profile tuples are collected with no violation. Sweep maintains a weaker invariant than the GI.

Definition 4.4.1 *(***Sweep Invariant***) Let $F_{f(j)}$ be the currently profiled filter. $F_{f(1)}$, …, $F_{f(n)}$ satisfies the Sweep Invariant if:*

$$\frac{d(i|i-1)}{t_{f(i)}} \geq \frac{d(j|i-1)}{t_{f(j)}}, 1 \leq i < j$$

□

Within each stage, Sweep works like A-Greedy: The Profiler collects profile tuples and the Re-optimizer detects and corrects violations of the Sweep Invariant efficiently by maintaining a matrix view V over the profile tuples. To maintain the Sweep Invariant when $F_{f(j)}$ is being profiled, the Re-optimizer needs to maintain $V[i,i], 1 \leq i \leq j$ and $V[i,j], 1 \leq i < j$ only, as shown in Figure 4.9(a). Furthermore, to maintain these entries, only attributes $b_{f(1)}, b_{f(2)}, \ldots, b_{f(j)}$ in the profile window are required. The Sweep Profiler collects profile tuples by profiling dropped tuples with the same drop-profiling probability as the A-Greedy Profiler. However, for each profiled tuple, the Sweep Profiler needs to additionally evaluate $F_{f(j)}$ only.

4.4.1 Convergence Properties, Run-time Overhead, and Adaptivity

Like A-Greedy, if characteristics stabilize then Sweep provably converges to an ordering that satisfies the GI. Furthermore, the run-time overhead of profiling and re-optimization is lower for Sweep than for A-Greedy: For a profiled tuple that is dropped by $F_{f(i)}$, the A-Greedy Profiler evaluates $n - i$ additional filters, while the Sweep Profiler evaluates at most one additional filter. Similarly, the worst-case cost of updating V for profile-tuple insertions and deletions is $O(n^2)$ for A-Greedy, but $O(n)$ for Sweep.

The lower run-time overhead of Sweep comes at the cost of reduced adaptivity speed compared to A-Greedy. Since only one filter is profiled in each stage of Sweep, an ordering

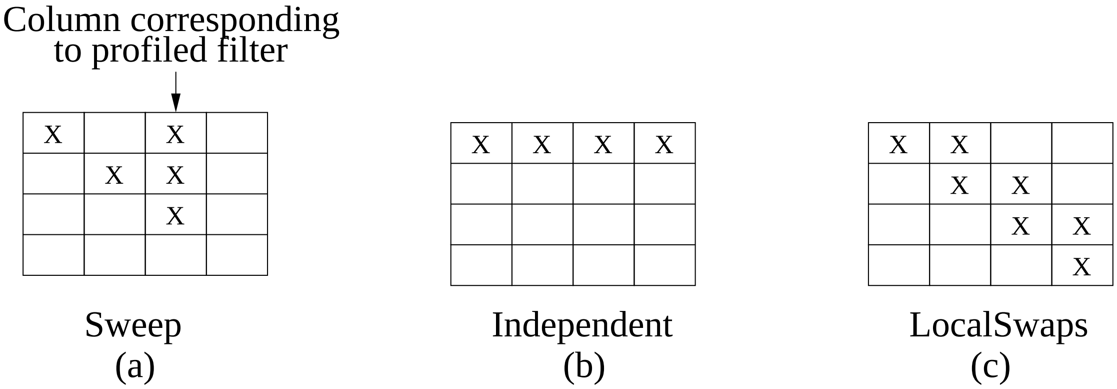


Figure 4.9: View entries maintained by A-Greedy variants

that violates the GI can remain undetected for a relatively long time (up to $n - 2$ stages of Sweep), and can have arbitrarily poor performance. Even after the violation is detected, Sweep may take longer to converge to the new GI ordering than A-Greedy would.

4.5 The Independent Algorithm

A-Greedy is designed specifically to perform well in the presence of nonindependent filters. Suppose we assume the filters are independent, an assumption made frequently in database literature although it is seldom true in practice [35, 105]. Under the independence assumption, conditional selectivities reduce to simple one-filter selectivities, and the GI reduces to the following invariant:

Definition 4.5.1 *(***Independent Invariant***)* $F_{f(1)}$, $F_{f(2)}$, ..., $F_{f(n)}$ *satisfies the Independent Invariant if:*

$$\frac{d(i|0)}{t_{f(i)}} \geq \frac{d(j|0)}{t_{f(j)}}, 1 \leq i \leq j \leq n$$

□

The *Independent* algorithm maintains the above invariant irrespective of whether the filters are actually independent or not. Like Sweep, Independent can be implemented as a variation of A-Greedy. To maintain the Independent Invariant, profile tuple creation is the same as in A-Greedy, but since conditional selectivities are not needed, the drop-profiling probability can be lower than in A-Greedy. Furthermore, the Re-optimizer needs to maintain the entries in the first row of V only, as shown in Figure 4.9(b). The original Eddies algorithm for adaptive ordering [10], ordering algorithms proposed recently for multiway stream joins [56, 114], and certain ordering algorithms for relational joins [78], all are similar to the Independent algorithm.

4.5.1 Convergence Properties, Run-time Overhead, and Adaptivity

Convergence properties of the Independent algorithm depend on whether the assumption of filter independence holds or not. If the filters are all independent, then Independent converges to the optimal ordering (not just the GI ordering) [63], and so does A-Greedy. However, if the filters are not independent, then the cost of the ordering produced by Independent can be $O(n)$ times worse than the GI ordering, as we show in the following example.

Example 4.5.1 Consider the pipelined filters in Example 4.2.1, but with n filters instead of four. Let input characteristics converge such that tuples arriving in S are distributed uniformly in $[1,\ 2,\ \ldots,\ 100]$. Let F_1–F_{n-1} contain $\{1,\ 2,\ \ldots,\ 49\}$, let F_n contain $\{50,\ 51,\ \ldots,\ 100\}$, and assume uniform processing times ($t_i = 1$, say) for all filters. Since $F_1, \ldots, F_{n-1}$ all have a higher probability of dropping a tuple than F_n, Independent will converge to an ordering O_i that is a permutation of $F_1, \ldots, F_{n-1}$ followed by F_n. The expected number of filters processed per input tuple is $0.49n + 0.51$. On the other hand, A-Greedy will converge to an ordering O_g starting with one of $F_1, \ldots, F_{n-1}$, then F_n, then the remaining filters in some order. The expected number of filters processed per input tuple is 1.49, which is optimal. □

Independent cannot guarantee the good convergence properties of A-Greedy or Sweep, as we have just seen. However, like A-Greedy it adapts to changes instantaneously, and with a lower drop-profiling probability and maintenance of V limited to the first row, it has lower run-time overhead.

4.6 The LocalSwaps Algorithm

The three algorithms we have seen so far all detect invariant violations involving filters F_i and F_j that are arbitrarily distant from one another in the current ordering. An alternative approach is to monitor “local” violations only. The *LocalSwaps* algorithm maintains the following invariant.

Definition 4.6.1 *(***LocalSwaps Invariant***)* $F_{f(1)}$, $F_{f(2)}$, $\ldots$, $F_{f(n)}$ *satisfies the LocalSwaps Invariant if:*

$$\frac{d(i|i-1)}{t_{f(i)}} \geq \frac{d(i+1|i-1)}{t_{f(i+1)}}, 1 \leq i < n$$

□

Intuitively, LocalSwaps detects situations where a swap between adjacent filters in O would improve performance. To maintain the LocalSwaps Invariant, the Re-optimizer needs to maintain only the entries in two diagonals of V, as shown in Figure 4.9(c). This reduces the cost of profiling: For each profiled tuple dropped by $F_{f(i)}$, the LocalSwaps Profiler needs to additionally evaluate $F_{f(i+1)}$ only. Furthermore, when a profile tuple is inserted into or deleted from the profile window, only a constant number of V entries need be updated.

4.6.1 Convergence Properties, Run-time Overhead, and Adaptivity

Unlike our other algorithms, the convergence behavior of LocalSwaps depends on how the stream and filter characteristics change. In the best cases, LocalSwaps converges to the same ordering as A-Greedy. However, in some cases the LocalSwaps ordering can have $O(n)$ times higher cost than the GI ordering, as we show in the following example.

Example 4.6.1 Consider the pipelined filters in Example 4.2.1, but with n filters instead of four. Let input characteristics converge such that tuples arriving in S are distributed uniformly in $[1,\ 2,\ \ldots,\ 100]$. Let F_i contain $\{1, 2, \ldots, 100\} - \{(i-1)\delta, \ldots, i\delta\}$, where $\delta = 100/n$, and , and assume uniform processing times ($t_i = 1$, say) for all filters. Suppose initially A-Greedy and LocalSwaps are both using ordering $O = F_1, F_2, \ldots, F_n$ satisfying their respective invariants. Suppose the tuples in F_n change so that F_n now contains

$\{(n-2)\delta, \ldots, (n-1)\delta\}$ and this is the only change before stabilization. The LocalSwaps Invariant still holds for $F_1, F_2, \ldots, F_n$, so LocalSwaps does not modify O. However, A-Greedy will modify O to $O' = F_n, F_{n-1}$, then the remaining filters in some order, which is optimal. The expected number of filters processed per input tuple for O and O' are $50(n+1)$ and $1 + 200/n$ respectively. □

Clearly, LocalSwaps has lower run-time overhead than A-Greedy because of its lower profiling and view-maintenance cost. However, as with any algorithm that is restricted to local moves, LocalSwaps may take more time to converge to the new plan when stream and filter conditions change, and it may get stuck in a local optima as shown in Example 4.6.1.

4.7 Experimental Evaluation

We have implemented the four adaptive ordering algorithms—A-Greedy, Sweep, Independent, and LocalSwaps—in the STREAM prototype DSMS. Our basic approach was to incorporate each of these algorithms into STREAM's `select` operator. Recall that pipelined filter queries in the STREAM system are processed by plans consisting of the `select` operator. We implemented four extended versions of the `select` operator, corresponding to our four adaptive algorithms for ordering the filters in the operator. In our implementation, the code of the `select` operator instantiates the StreaMon framework for adaptive processing. For example, Figure 4.10 shows how we extended the `select` operator to implement A-Greedy based on the StreaMon framework. The contents of this figure follow from Sections 4.3.1 and 4.3.2. The `select` operator was extended along similar lines to implement Sweep, Independent, and LocalSwaps.

We now report the results from our experimental study of A-Greedy and its variants with respect to their convergence properties, run-time overhead, and speed of adaptivity. Our basic experimental setup consists of $n+1$ synthetic streams $S_0, S_1, \ldots, S_n$ and pipelined filter queries of the form:

Select *
From S_0
Where F_1 and F_2 and $\cdots$ and F_n

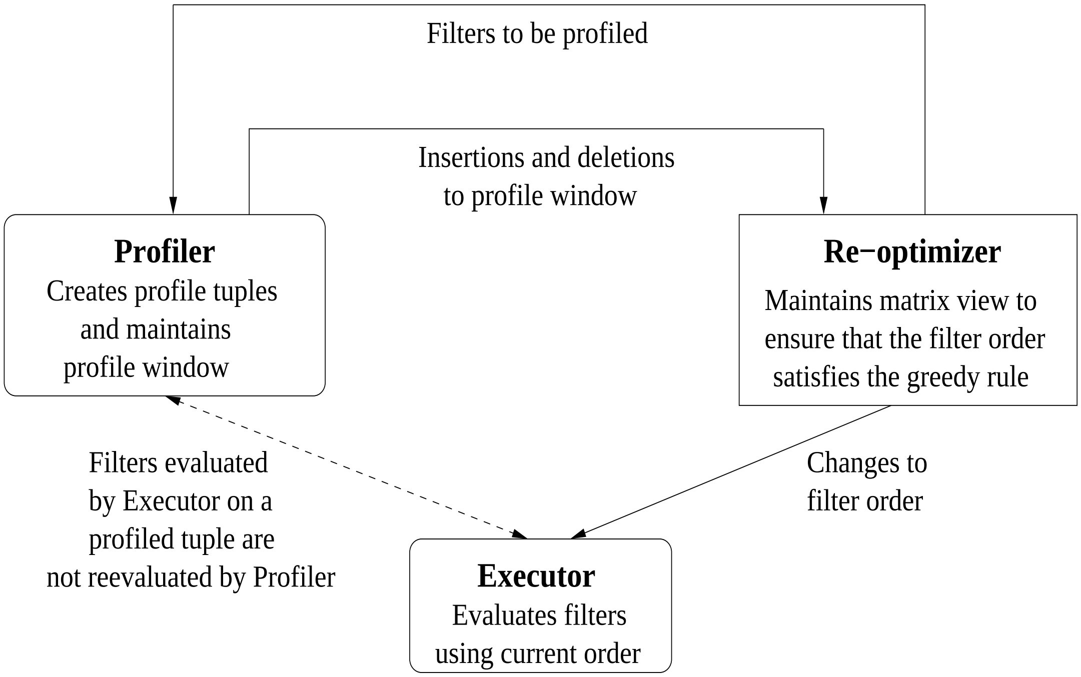

Filters to be profiled
Insertions and deletions to profile window
Profiler
Creates profile tuples and maintains profile window
Re−optimizer
Maintains matrix view to ensure that the filter order satisfies the greedy rule
Filters evaluated by Executor on a profiled tuple are not reevaluated by Profiler
Changes to filter order
Executor
Evaluates filters using current order

Figure 4.10: Architecture of the extended version of the `select` operator that uses A-Greedy to order filters

A query of this form is a conjunction on n filters $F_1, F_2, \ldots, F_n$ over stream S_0. In our experiments, when filter F_i is evaluated over a tuple $s \in S_0$, F_i returns true if and only if s is contained in a CQL relation R_i specific to F_i. We use a 10,000-tuple sliding window on stream S_i in our experiments as the relation R_i corresponding to filter F_i. Therefore, the pipelined filter queries that we consider can be expressed in CQL as:

```
Select  *
From    $S_0$
Where   $S_0$ in  (Select  *
                   From    $S_1$ [Rows 10,000])   and
        $S_0$ in  (Select  *
                   From    $S_2$ [Rows 10,000])   and
        ⋮
        $S_0$ in  (Select  *
                   From    $S_n$ [Rows 10,000])
```

In this representation, filter F_i corresponds to the CQL predicate "S_0 in (Select * From S_i [Rows 10,000])." The 10,000-tuple sliding window over each stream is maintained as a synopsis; recall Section 2.2.3. A hash index is maintained on each window as part of the synopsis so that the filters can search for matching tuples efficiently. All synopses used in the query plans are stored in memory. All experiments were done on a 700 MHz Linux machine with 1024 KB processor cache and 2 GB memory. The default values of all parameters used in the experiments are shown in Table 4.1. 95% confidence intervals are shown for measured values in the graphs in this chapter.

4.7.1 Summary of Experimental Results

1. Under stable stream and filter characteristics, A-Greedy's ordering is usually optimal, while Independent's ordering almost always has higher cost.

2. Sweep always converges on A-Greedy's ordering under stable stream and filter characteristics. LocalSwaps usually also finds this ordering.

3. As expected, Sweep, Independent, and LocalSwaps have lower run-time overhead than A-Greedy. As the number of filters increases, or as some filters become much more expensive than the others, the run-time overhead of both Independent and A-Greedy increases, while that of Sweep and LocalSwaps remains stable.

4. As expected, Independent and A-Greedy adapt faster to changes than Sweep or LocalSwaps. As the frequency of changes increases, the relative performance of A-Greedy over the other three algorithms improves because it adapts faster, usually finding the optimal ordering immediately.

4.7.2 Convergence Experiments

Our first set of experiments study the convergence behavior of our algorithms when stream and filter characteristics stabilize. The factors affecting performance during convergence are the number, selectivity, and cost of the filters, and the correlation among them. We use a relatively straightforward model to capture correlation among filters. The n filters

Parameter	Default Value
Drop-profiling probability	0.01 (A-Greedy), 0.005 (Independent)
Profile-window size	500 profile tuples (Sweep), 1000 profile tuples (others)
Thrash-avoidance parameter	$\alpha = 0.9$
Correlation factor	$\Gamma = 2$
Unconditional filter selectivity	50%
Filter processing cost	1 hash probe on 10,000-entry index

Table 4.1: Default values used in experiments

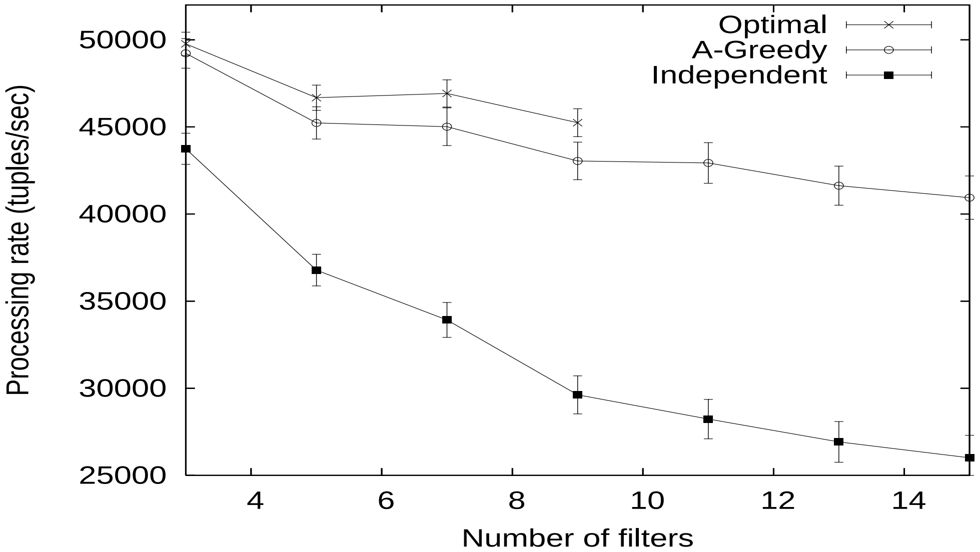


Figure 4.11: Effect of the number of filters under convergence

are divided into $\lceil n/\Gamma \rceil$ groups containing Γ filters each, where Γ is called the *correlation factor*. Two filters are independent if they belong to different groups, otherwise they are positively correlated such that they produce the same result on 80% of input tuples. The filters are completely independent when $\Gamma = 1$ and are most correlated when $\Gamma = n$. For each experiment, we fix Γ and an unconditional selectivity for each filter group, which implies the conditional selectivities based on Γ and our 80% rule.

Figure 4.11 shows the performance of A-Greedy and Independent for different values of n. (Sweep and LocalSwaps are considered later.) The y-axis shows the average processing rate in tuples per second, measured over a large interval after convergence. For small n,

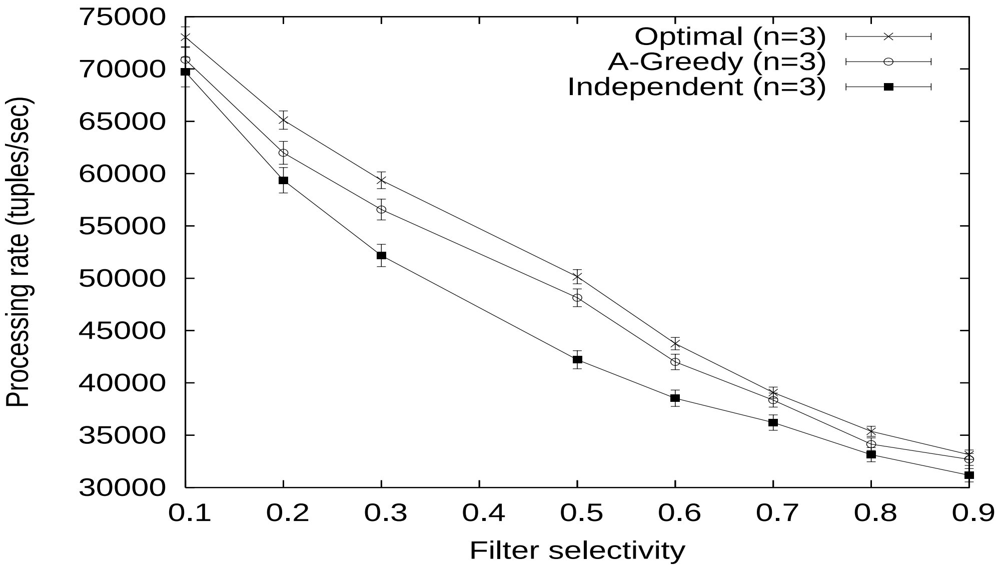


Figure 4.12: Effect of selectivity under convergence

Figure 4.11 also shows the performance of the *Optimal* algorithm, which uses the optimal ordering computed from the input statistics by an offline exhaustive search, thereby showing the maximum attainable performance. A-Greedy is better than Independent throughout, and this performance gap widens with n. A-Greedy's performance is near-optimal for small n, but it degrades as n increases. As we will see in Section 4.7.3, this degradation is because of A-Greedy's increasing run-time overhead.

Figure 4.12 shows the performance of A-Greedy and Independent for different filter selectivities. The relative performance of A-Greedy with respect to Independent is best for intermediate values of filter selectivities. For low selectivities, most of the tuples get dropped by the first filter, which is the same for both algorithms. For high selectivities, very few tuples get dropped, so costs do not vary significantly across orderings.

Figures 4.13 and 4.14 show the performance of A-Greedy and Independent for different values of the correlation factor Γ. As expected, the relative performance of A-Greedy initially improves with respect to Independent as the filters become more correlated. Because of our simple model of correlation, when Γ is close to n all the filters are similar and costs do not vary much across orderings, so the performance improvement of A-Greedy is less significant.

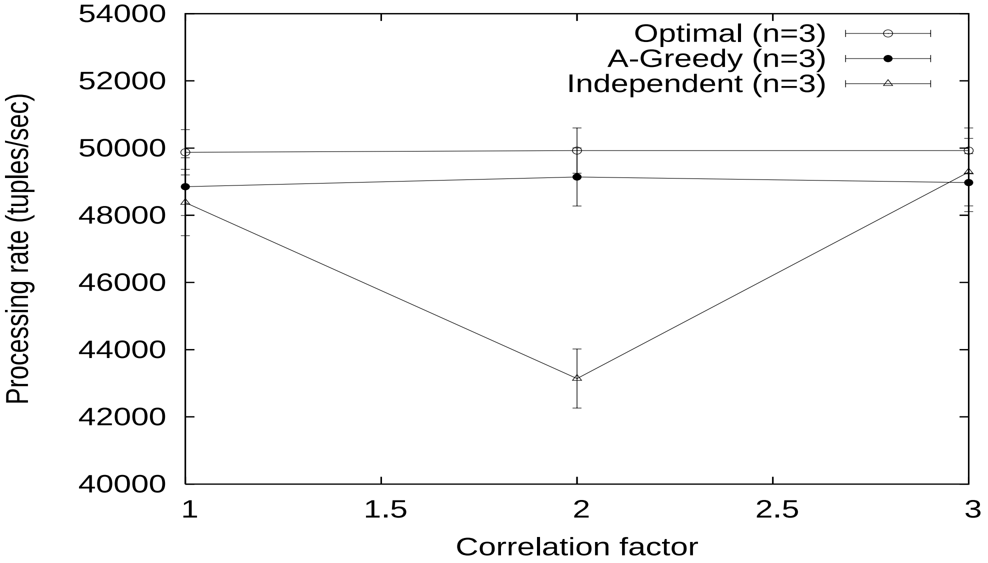


Figure 4.13: Effect of correlation under convergence for $n = 3$ filters

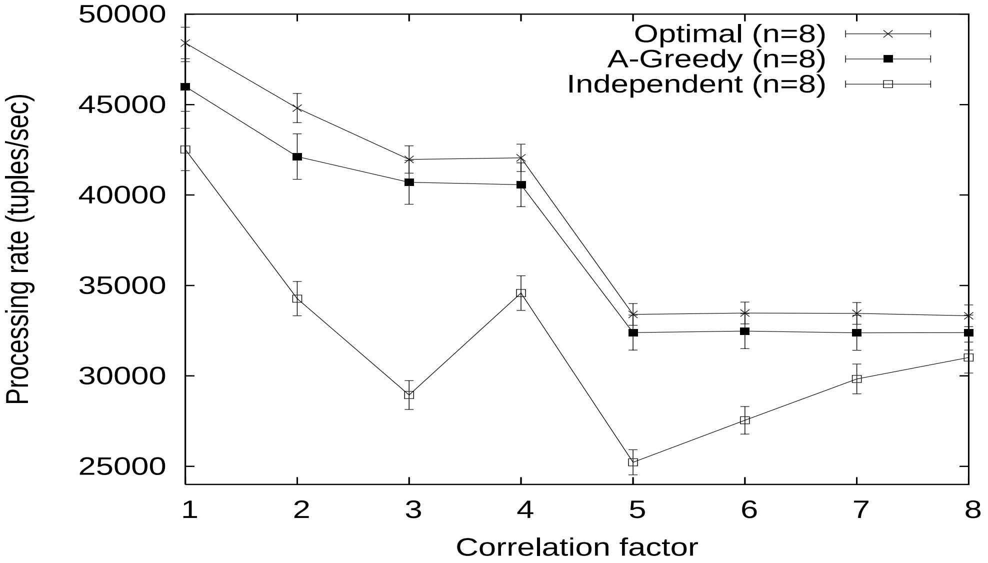


Figure 4.14: Effect of correlation under convergence for $n = 8$ filters

4.7.3 Run-time Overhead Experiments

Table 4.2 breaks down the time spent by A-Greedy for $n = 3$ and $n = 8$ under stable conditions. In each case we consider two values of the drop-profiling probability p: the default $p = 0.01$ and a higher $p = 0.05$. The listed tasks refer to Section 4.3.4. More

Component	n=3, p=0.01	3, 0.05	8, 0.01	8, 0.05
Tuple proc.	98.77	94.17	96.62	84.77
Profile-tuple	0.88	4.17	2.92	13.34
Profile-window	0.20	0.93	0.20	0.91
View update	0.15	0.73	0.22	0.98
View violations	0	0	0.04	0

Table 4.2: Run-time percentage breakdown for A-Greedy

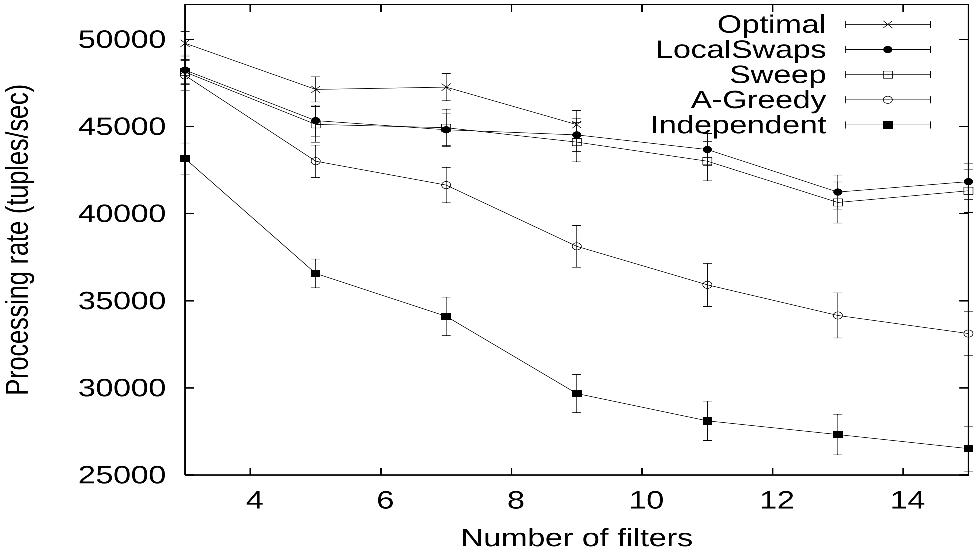


Figure 4.15: Performance of all algorithms

than 98% of A-Greedy's time is spent either performing useful tuple processing or creating profile tuples. The overhead of creating profile tuples increases both with n and with p, going from less than 1% for $n = 3$ and $p = 0.01$, to 13.34% for $n = 8$ and $p = 0.05$.

Figures 4.15 and 4.16 show the processing rate and run-time breakdown of all our adaptive algorithms under convergence, varying n. We use a drop-profiling probability of 0.05 in this experiment. Now we see the benefits of the lower run-time overhead of Sweep and LocalSwaps. Figure 4.15 shows that the gap between A-Greedy and Sweep and LocalSwaps increases with n. Figure 4.16 shows that this widening gap is solely because of A-Greedy's run-time overhead. Also note from Figure 4.16 that the time spent processing tuples is similar for Optimal, A-Greedy, Sweep, and LocalSwaps, indicating convergence to a similar ordering.

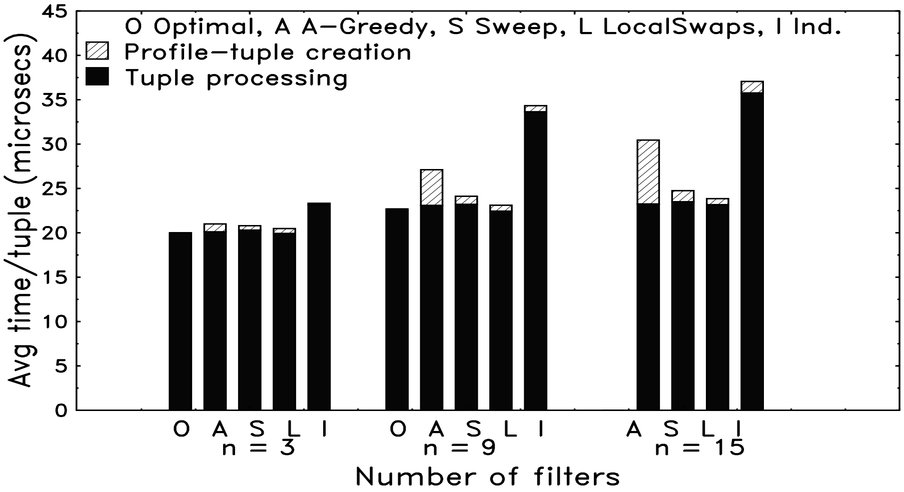


Figure 4.16: Run-time breakdown for Figure 4.15

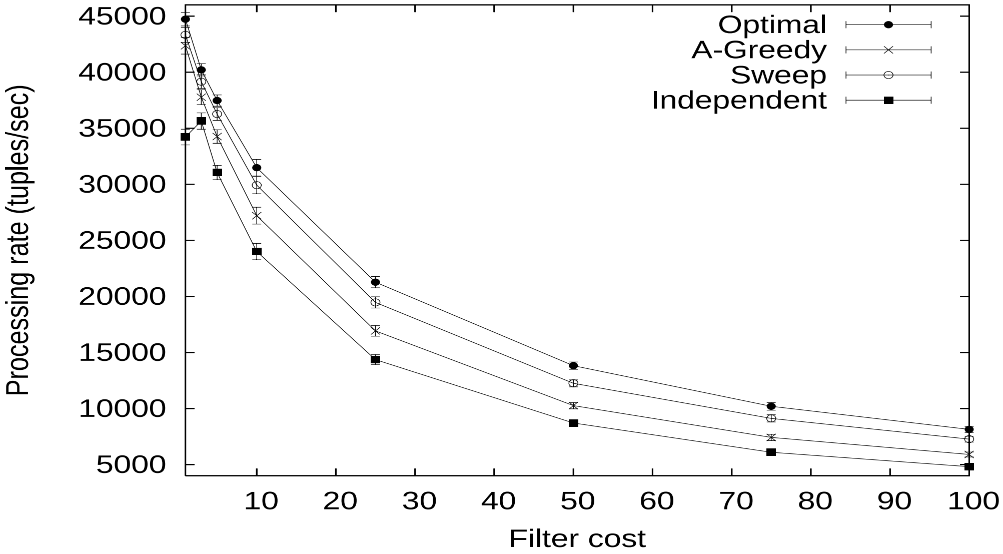


Figure 4.17: Effect of varying filter costs

Figures 4.17 and 4.18 show the processing rate and run-time breakdown of our adaptive algorithms for different values of the filter costs. We used $n = 8$, with four filters having cost one and the other four filters having cost c for $c \in \{1, 10, 50, 100\}$, where a filter with cost c performs c random probes on the memory-resident index per input tuple. Figure 4.18

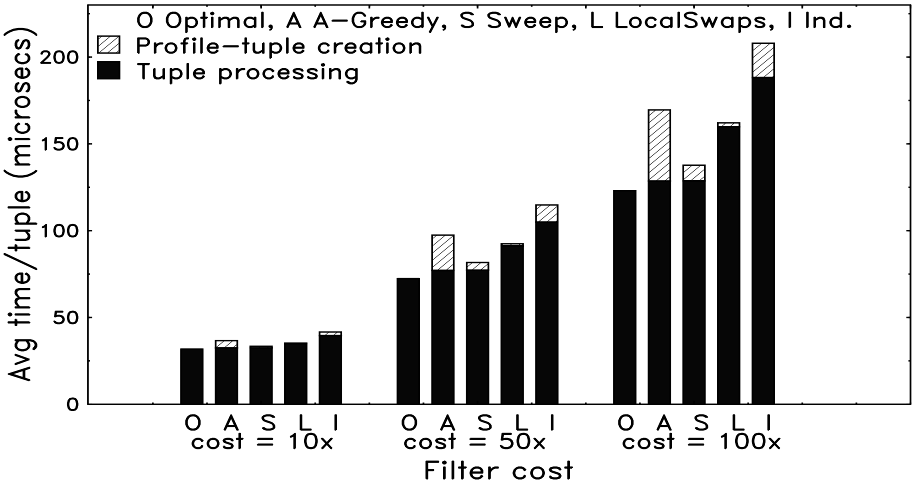


Figure 4.18: Run-time breakdown for Figure 4.17

shows that the percentage run-time overhead of A-Greedy is much higher than with uniform filter costs; around 25% for $c = 100$. Both Sweep and LocalSwaps perform better than A-Greedy because of their lower run-time overhead. Although LocalSwaps has slightly lower run-time overhead than Sweep, Sweep performs better because LocalSwaps does not find the optimal ordering for $c = 50$ and $c = 100$.

4.7.4 Adaptivity Experiments

Each of Figures 4.19–4.21 shows a timeline from experiments where we varied the stream and filter characteristics over time for $n = 8$. We use filters with different selectivities and periodically permute the filter selectivities to change filter characteristics. For example, Figure 4.19 shows that a change was made after the system had processed 600,000 input tuples. The y-axis in Figures 4.19–4.21 shows the number of filters evaluated per 2000 input tuples. As expected, Figures 4.19 and 4.20 show that A-Greedy converges to the new Greedy ordering much faster than Sweep or LocalSwaps. Also note that the cost of LocalSwaps's ordering changes more smoothly compared to Sweep. Figure 4.21 shows that Independent also adapts quickly, but its ordering is worse than A-Greedy's ordering both before and after the change.

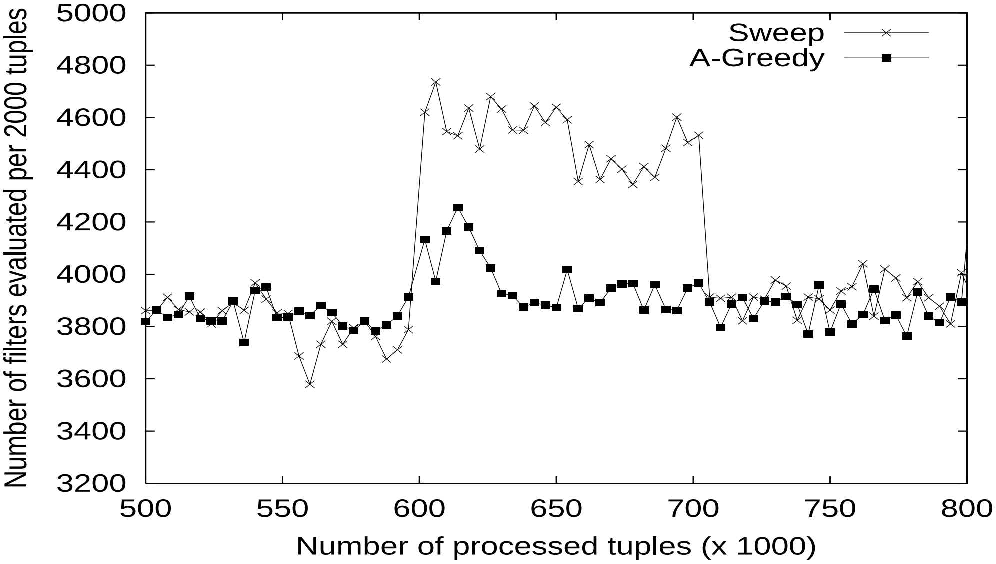


Figure 4.19: Adaptivity of A-Greedy versus Sweep

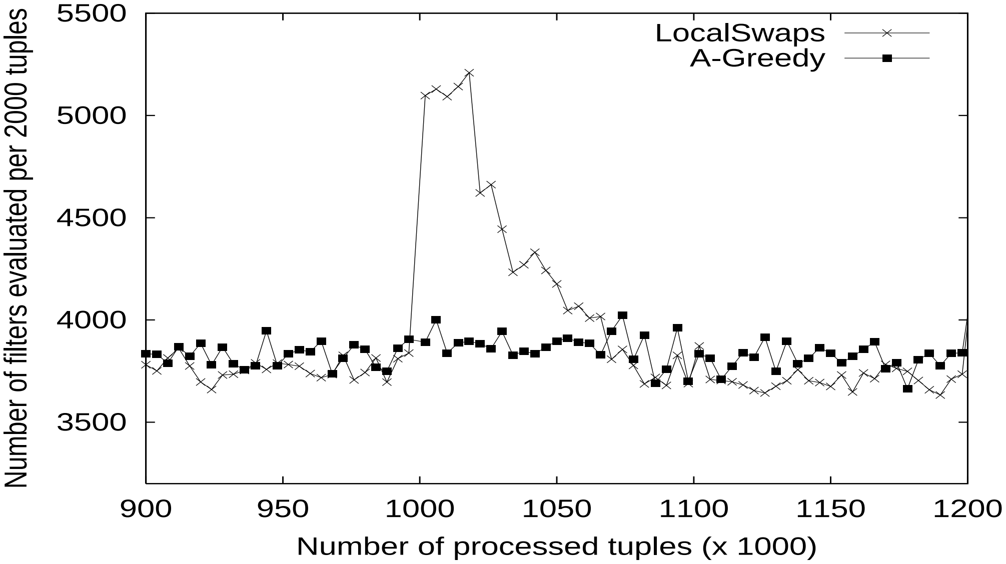


Figure 4.20: Adaptivity of A-Greedy versus LocalSwaps

In general, A-Greedy and Independent have higher run-time overhead but faster adaptivity to changes than Sweep or LocalSwaps. Figure 4.22 shows this tradeoff by plotting on the y-axis the total time to process a workload where filter characteristics are varied

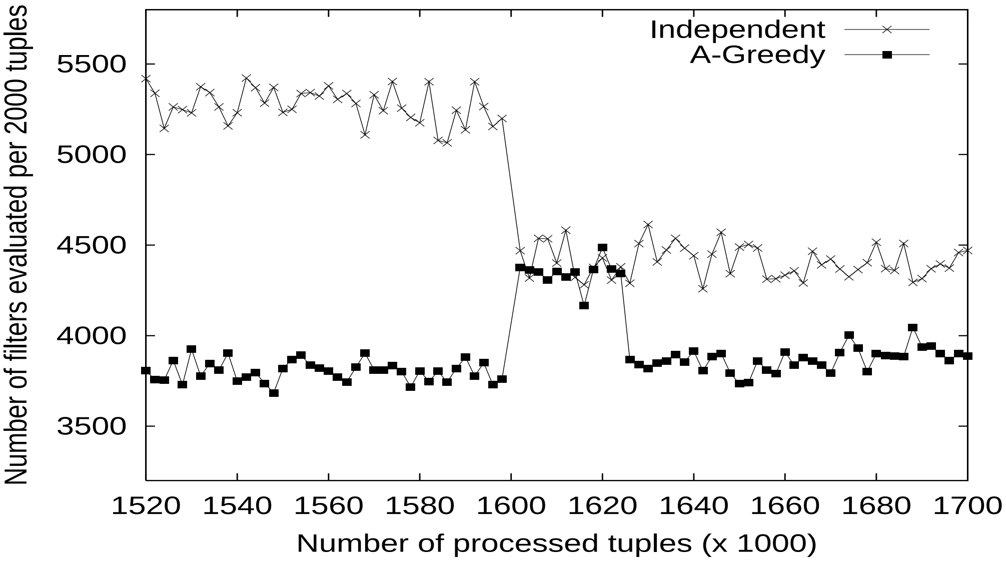


Figure 4.21: Adaptivity of A-Greedy versus Independent

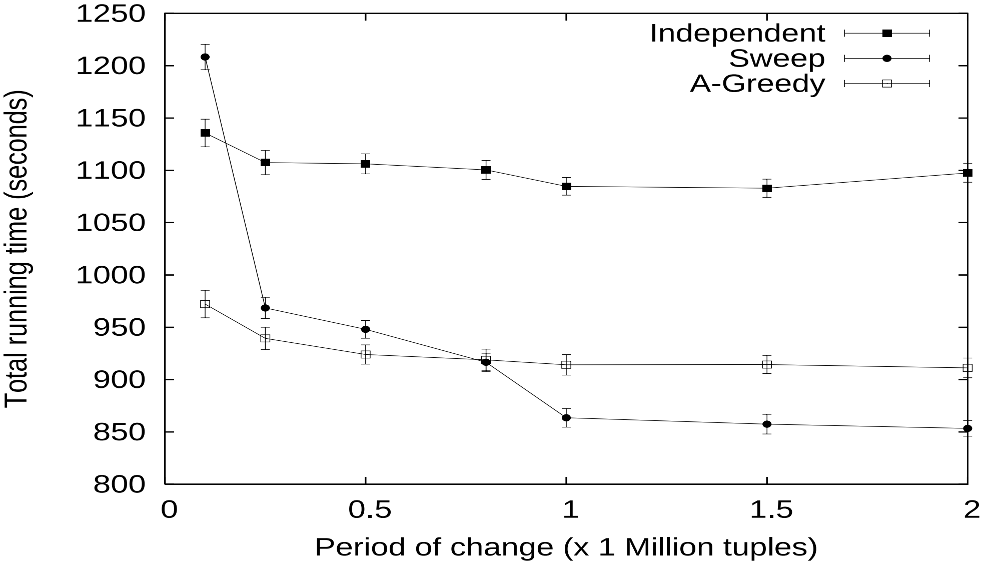


Figure 4.22: Varying the rate of change

periodically, with the period shown on the x-axis. The lower this period, the higher the rate of change. We used $n = 8$, with four filters having cost one and the other four having cost ten. Each change permutes the selectivities randomly. When the rate of change is high, the faster adaptivity of A-Greedy and Independent enable them to significantly outperform Sweep (and LocalSwaps also, which is not shown in Figure 4.22 to avoid clutter). This

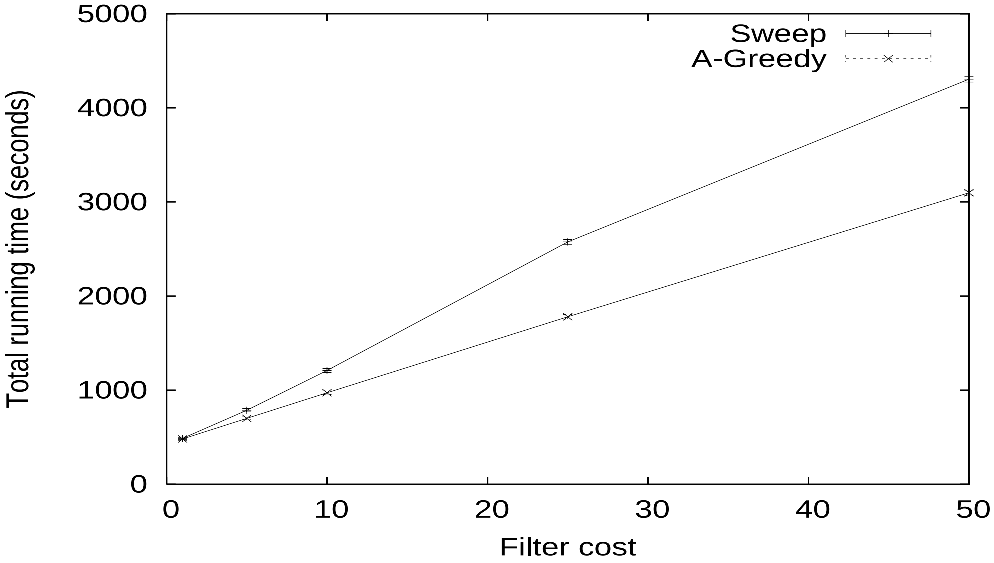


Figure 4.23: Varying filter cost at $x = 10^5$ from Figure 4.22

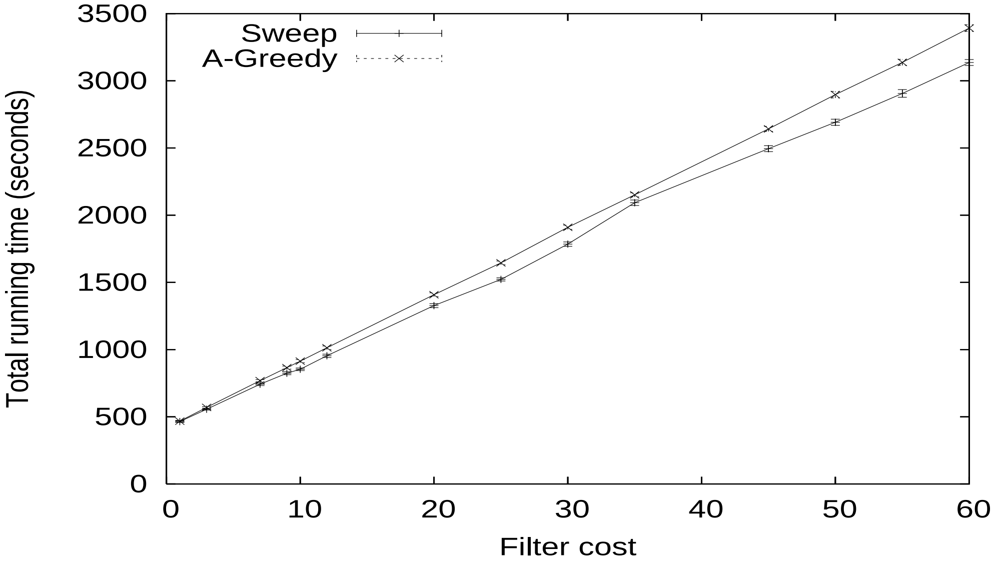


Figure 4.24: Varying filter cost at $x = 10^6$ from Figure 4.22

advantage diminishes as we reduce the rate of change, and the overall behavior approaches the convergence behavior of Figures 4.15 and 4.17.

Figures 4.23 and 4.24 further explore the tradeoff between run-time overhead and adaptivity using the same setup as in Figure 4.22. Figure 4.23 fixes the period of change at 100,000 tuples, where A-Greedy performs better than Sweep (Figure 4.22), and Figure 4.24 fixes the period of change at 1,000,000 tuples, where Sweep performs better. In each case

Algorithm	Convergence properties	Run-time overhead	Speed of Adaptivity
A-Greedy	4-approximation	High relative to others	Fast
Sweep	4-approximation	Low (less work per profiling step)	Slowest among the four algorithms
Independent	Misses correlations	Low profiling rate	Fast
LocalSwaps	May get caught in local optima	Low (less work per profiling step)	Slow

Table 4.3: Summary of performance of all algorithms

we vary on the x-axis the cost of the expensive filters, which varies the run-time overhead (Figure 4.18). In Figure 4.23 we see that even when the filter costs are high, so A-Greedy has high run-time overhead, A-Greedy continues to perform better than (actually improves over) Sweep. The reason is that as filter costs increase, the performance gaps between different orderings also increase, so Sweep gets penalized more for its slower adaptivity. Figure 4.24 shows that given sufficient time between changes, Sweep's performance with respect to A-Greedy improves as A-Greedy's percentage run-time overhead increases.

4.8 Summary of A-Greedy and its Variants

Table 4.3 summarizes the performance of A-Greedy, Sweep, Independent, and LocalSwaps with respect to the three metrics for adaptive query processing introduced in Section 3.1 in Chapter 3. A-Greedy has the best possible convergence properties for pipelined filters: the cost of the filter ordering satisfying the GI is at most four times the cost of the optimal filter ordering. Furthermore, any polynomial-time ordering algorithm for pipelined filters can at best provide a constant-factor approximation guarantee of four for this problem [89]. As we saw in the experiments in Section 4.7.4, because A-Greedy maintains the full matrix view, it can detect and correct violations in the GI quickly. Therefore, A-Greedy can adapt quickly to changes in stream and filter characteristics.

Sweep takes a stage-by-stage approach where in each stage it moves at most one filter to its appropriate place in the greedy order. Because of this property, if stream and filter characteristics stabilize, Sweep provably converges to an ordering that satisfies the GI. However, because the work done by Sweep's Profiler and Re-optimizer in each stage focuses on a single filter, Sweep adapts slowly compared to A-Greedy—e.g., see Figure 4.19—but the run-time overhead of Sweep is much lower than that of A-Greedy—e.g., see Figure 4.18.

Independent does not have the good convergence properties of A-Greedy and Sweep. In fact, if the filters are correlated, then the cost of the ordering that satisfies the Independent Invariant can be $O(n)$ times worse than that of the optimal ordering. However, like A-Greedy and unlike Sweep, Independent can adapt quickly when stream or filter characteristics change, e.g., see Figure 4.21. Furthermore, like Sweep, Independent has significantly lower run-time overhead than A-Greedy because of Independent's lower drop-profiling probability; see Figure 4.18. Note that if the filters are guaranteed to be independent, then Independent performs well on all metrics: it converges to the optimal ordering when statistics stabilize, adapts quickly to changes, and also has low run-time overhead.

Like Independent, LocalSwaps also does not have the good convergence properties of A-Greedy and Sweep. If stream and filter characteristics can change in arbitrary ways, then the cost of the ordering produced by LocalSwaps can be $O(n)$ times worse than that of the optimal ordering. Like Sweep, LocalSwaps may adapt slowly to changes in stream and filter characteristics, e.g., see Figure 4.20. However, like Sweep and Independent, LocalSwaps has significantly lower run-time overhead than A-Greedy, e.g., see Figure 4.18. Recall that LocalSwaps has low run-time overhead because its Profiler evaluates only one extra filter each time a tuple is profiled.

Figure 4.25 places our four adaptive algorithms on a three-dimensional plot where each axis corresponds to a performance metric for adaptive query processing: convergence properties under stable conditions, run-time overhead, and speed of adaptivity. Each axis has values "weak" and "strong", where "weak" represents weak performance on the corresponding metric, and "strong" represents good performance.

It can be observed from Figure 4.25 that each of our algorithms performs well on at most two metrics. For example, A-Greedy performs well on convergence properties and speed of adaptivity, but performs weakly on the run-time overhead metric. We conjecture that any algorithm for adaptive pipelined filters can perform well on two of the metrics only at the cost of not performing well on the third metric. That is, there is a three-way tradeoff in adaptive pipelined-filter processing among good convergence properties under stable conditions, run-time overhead, and speed of adaptivity; recall Section 3.1 in Chapter 3.

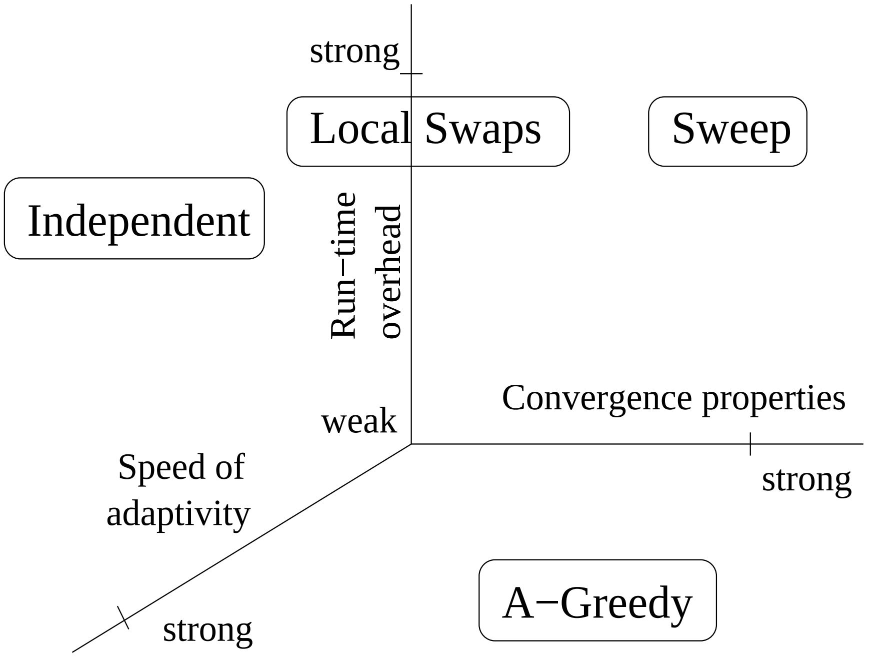


Figure 4.25: Tradeoff spectrum for adaptive processing of pipelined filters

4.9 Related Work

When filters are independent, the optimal ordering of a set of pipelined filters can be computed in polynomial time. Most previous work on pipelined filters and related problems makes the independence assumption and uses this ordering technique, e.g., [33, 63, 78]. Without the independence assumption, the problem is NP-hard. Previous work on ordering of correlated filters, e.g., [76, 95], either uses exhaustive search or proposes simple heuristics with no provable guarantees on the solution obtained. In [89] we introduce and study the *pipelined set cover* problem. Our analysis of the convergence properties of our adaptive algorithms is based on a mapping from pipelined filters to pipelined set cover. Pipelined set cover is NP-hard, so in [89] we develop a linear-programming framework to analyze several approximation algorithms for this problem. We also consider the online version of pipelined set cover in an adversarial setting.

As discussed in Section 1.5, previous work on adaptive query processing considers primarily traditional relational query processing [26, 70, 71, 72, 74, 105, 113]. Consequently, these algorithms do not extend to continuous queries and provide no guarantees on convergence.

Eddies [10] route each tuple adaptively across the operators that need to process it. Our approach is more coarse-grained than Eddies, since at a given point in time one ordering of the filters is followed by all tuples. Nevertheless, one of our algorithms (the *Independent* algorithm) is comparable to the original Eddies algorithm proposed in [10]. A more recent Eddies paper [41] reduces run-time overhead by routing tuples in batches. However, the effects on adaptivity are not studied, and the statistics that need to be monitored is exponential in the number of inputs and operators. Note that one application of our algorithms could be to provide efficient routing schemes for Eddies.

4.10 Conclusion

Pipelined filters, where a stream is processed by a set of commutative filters, are common in stream-based applications. We addressed the problem of ordering the filters adaptively to minimize processing cost in an environment where stream and system conditions vary unpredictably over time. Our core algorithm for adaptive pipelined filters, A-Greedy, follows the StreaMon framework. A-Greedy's Executor uses STREAM's `select` operator for processing filters. The Profiler samples tuples processed by the `select` operator, and maintains statistics about the evaluation of each filter on the sampled tuples. These statistics are input to A-Greedy's Re-optimizer, which orders the filters using a greedy algorithm based on the ratio of the conditional selectivities to tuple-processing times of the filters.

We analyzed the behavior of A-Greedy in terms of our three metrics for adaptive query processing, namely, convergence properties under stable conditions, run-time overhead, and speed of adaptivity. We observed that while A-Greedy has good convergence properties and speed of adaptivity, it suffers from relatively high run-time overhead. We developed variants of A-Greedy to address this problem. The Sweep variant, which profiles only one filter at a time, has good convergence properties and low run-time overhead, but may adapt slowly to changes. The Independent variant, which does not monitor conditional filter selectivities, has low run-time overhead and adapts quickly to changes, but lacks good convergence properties when filters are correlated. The LocalSwaps variant monitors only for local changes in the filter ordering when stream or filter characteristics change, so it lacks good convergence properties and may adapt slowly. However, LocalSwaps has much lower run-time overhead than A-Greedy.

Based on our analysis of A-Greedy and its variants, as well as the results of our experimental evaluation, we conjectured that there is a three-way tradeoff in adaptive pipelined-filter processing among convergence properties under stable conditions, run-time overhead, and speed of adaptivity. That is, any algorithm for adaptive pipelined filters can perform well on two of these metrics only at the cost of not performing so well on the third metric.

Finally, we presented a thorough performance evaluation of our algorithms based on an implementation in the STREAM prototype. One class of experiments studied the performance of A-Greedy and its variants in terms of our three metrics for adaptive query processing. We considered various steady-state settings of the input stream and filter characteristics, as well as dynamic variations of these characteristics. Our experimental results verified the observations that came from our analytical study of A-Greedy and its variants.

Another class of experiments studied the behavior of our algorithms under different rates at which input characteristics vary. For example, it is interesting to know how an algorithm like A-Greedy that adapts quickly, but has higher run-time overhead, compares to an algorithm like Sweep that has lower run-time overhead, but may adapt slowly to changes. Our experiments showed that for low rate of change, Sweep can be better than A-Greedy, but A-Greedy outperforms Sweep as the rate of change increases. A-Greedy performs well for a wide range of values of the rate of change, indicating that it is a very robust algorithm.

Chapter 5

Adaptive Processing of Windowed Stream Joins

The previous chapter described our instantiation of the generic StreaMon framework to process pipelined filter queries adaptively in environments where stream and system conditions can be unpredictable and volatile. We now consider the adaptive processing of a more complex class of continuous queries, which we call *windowed stream joins*. We first describe how the STREAM system processes windowed stream joins using fully-pipelined *MJoin* plans. An MJoin consists of one join pipeline per input stream, which joins updates to the window on that stream with the windows on the other streams. As illustrated in Section 2.2.4 in Chapter 2, MJoins are implemented by the `mjoin` operator in STREAM.

We show how the A-Greedy algorithm developed for pipelined filters in the previous chapter can handle adaptive ordering of join operators in MJoin pipelines. The theoretical guarantees provided by A-Greedy (Section 4.3.3) hold for a large class of windowed stream joins, but not for all of them. As it turns out, MJoins may not always be the most efficient way of processing windowed stream joins, due to a potential for excessive recomputation when multiple tuples in the stream have the same join values. We propose a new algorithm called *A-Caching* that addresses this problem by placing *caches* adaptively in MJoin pipelines. Caches store join results computed by operators in the pipeline in the recent past. The cached results may be reused when join values repeat among input stream tuples, avoiding the recomputation problem.

Caches are highly desirable from an adaptivity perspective: Caches can be added, populated incrementally, and dropped with little overhead. Memory can be allocated to caches and freed dynamically, without compromising the accuracy of the join result. In the STREAM system, caches are implemented as synopses managed by join operators in the MJoin pipelines. Recall from Section 2.2.3 in Chapter 2 that synopses are used by operators in STREAM query plans to store run-time state that may be required for future evaluation of that operator.

A-Caching enables our plans for windowed stream joins to adapt over the entire spectrum between subresult-free MJoins and conventional tree-shaped join plans with fully-materialized subresults at every intermediate node. Like A-Greedy, A-Caching follows the generic StreaMon framework from Chapter 3 for adaptive query processing. We describe how A-Caching instantiates the Executor, Profiler, and Re-optimizer components of StreaMon. Finally, we report a thorough experimental evaluation of A-Caching from an implementation in the STREAM DSMS.

This chapter proceeds as follows. Section 5.1 introduces windowed stream joins and MJoins. Section 5.2 describes how the A-Greedy algorithm from Chapter 4 can handle the adaptive ordering of join operators in MJoins. Section 5.3 illustrates how excessive recomputation of intermediate results may limit the performance of MJoins, and Section 5.4 shows how to address this problem by placing caches in MJoin pipelines. Section 5.5 introduces the A-Caching algorithm for adaptive placement of caches in MJoin pipelines, and describes A-Caching's Profiler and Re-optimizer. Section 5.6 presents A-Caching's dynamic memory allocation algorithm. In Section 5.7 we extend A-Caching by expanding the range of caches considered for placement in the MJoin pipelines. Section 5.8 presents experimental results from an implementation of our algorithms in the STREAM DSMS. Related work is discussed in Section 5.9, and we conclude in Section 5.10.

5.1 Stream Joins and MJoins

In Chapter 2 we gave the following example CQL query, which represents a common class of continuous queries called *windowed stream joins*:

```
Select    *
```

From S_1 [Rows 1000], S_2 [Range 2 Minutes]
Where $S_1.A = S_2.A$ and $S_1.A > 10$ and $S_1.B < 20$

This query, denoted Q, is a two-way windowed join of streams S_1 and S_2. A window is specified over each stream that limits the tuples in the stream that can contribute to new join results at any point in time. (Recall from Section 2.1.2 in Chapter 2 that windows in CQL can be tuple-based or time-based.) Q's result is a CQL relation. At any given time, this result relation contains the join (on attribute A, with $A > 10$ and $S_1.B < 20$) of the last 1000 tuples of S_1 with the tuples of S_2 that have arrived in the last 2 minutes.

In CQL, an n-way windowed stream join has the following general form:

Select *
From $S_1[W_1]$, $S_2[W_2]$, $\cdots$, $S_n[W_n]$
Where *join and filter predicates over* S_1–S_n

This query joins streams $S_1, S_2, \ldots, S_n$ to produce an n-way join result, which is a CQL relation. The window specification W_i over S_i limits the tuples in S_i that can contribute to new join results at any point in time. At any given time, the result relation contains the join of the last W_i worth of tuples in S_i, denoted window $S_i[W_i]$, across all streams. The tuples in the result relation must satisfy all the join and filter predicates specified in the query.

As time advances or tuples arrive in the input streams, new tuples enter the windows and old tuples expire from these windows. The insertion and deletion of tuples in the windows may cause the insertion and deletion of tuples in the join result. In Figure 2.2 in Chapter 2, we used a plan containing an MJoin operator (`mjoin`) to process our example windowed stream join. The multiway MJoin operator can process n-way windowed joins in general, as we show in Figure 5.1. σ_i denotes the set of filter predicates on stream S_i. For clarity, the pipelines containing the join operators in the MJoin have been enlarged in Figure 5.1. Each window in the plan is stored as a STREAM synopsis; recall Section 2.2.3 in Chapter 2.

An MJoin for an n-way windowed stream join contains n inputs. Each input corresponds to the insertion and deletion of tuples in the window over one of the streams, and is processed by one of the join pipelines in the MJoin. Without loss of generality, let us focus on the join pipeline for stream S_1 in Figure 5.1. When a tuple s_1 is inserted into the window

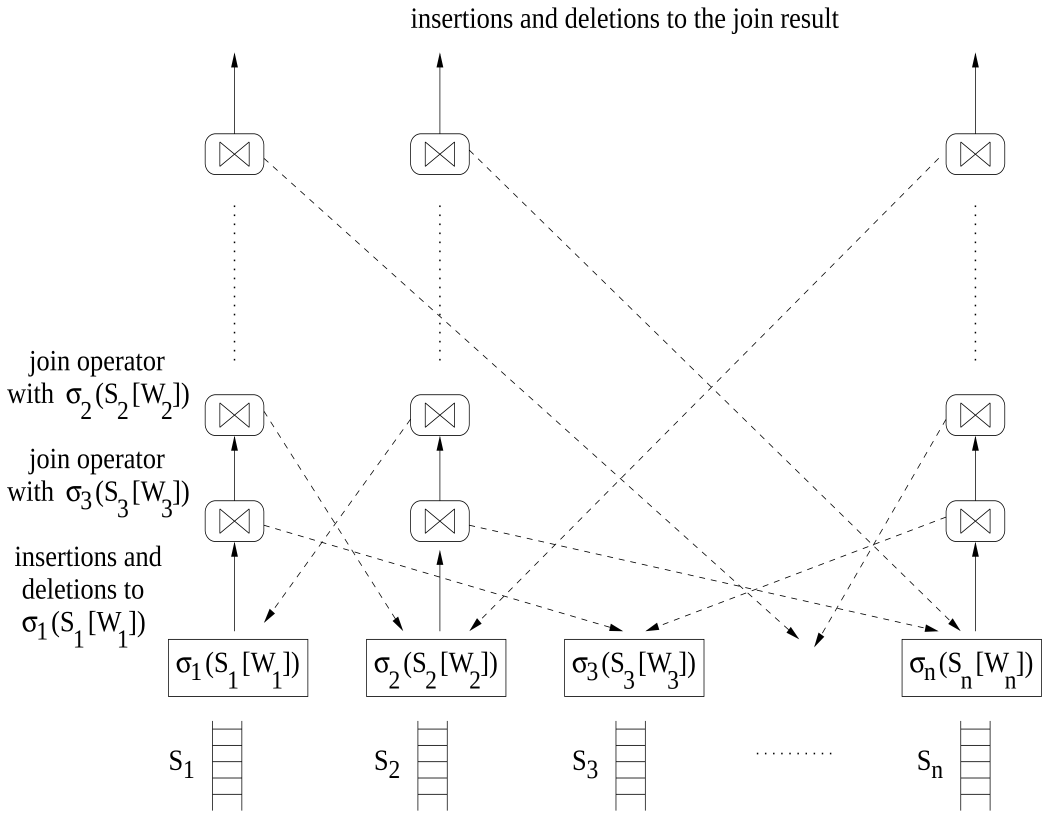

insertions and deletions to the join result
join operator with σ2(S2[W2])
join operator with σ3(S3[W3])
insertions and deletions to σ1(S1[W1])
σ1(S1[W1])
σ2(S2[W2])
σ3(S3[W3])
σn(Sn[Wn])
S1
S2
S3
Sn

Figure 5.1: MJoin for an n-way windowed stream join

$S_1[W_1]$, s_1 will be processed by this pipeline if s_1 satisfies the set of filter predicates σ_1 on S_1. From Figure 5.1, the first operator in this pipeline will join s_1 with the current (filtered) window $\sigma_3(S_3[W_3])$. For each $s_3 \in \sigma_3(S_3[W_3])$ that joins with s_1, the join operator generates intermediate joined tuples of the form $s_1 \cdot s_3$. These intermediate joined tuples are sent as insertions to the next operator in the pipeline, which in Figure 5.1 is a join with window $\sigma_2(S_2[W_2])$. This operator will join the $s_1 \cdot s_3$ tuples with tuples in $\sigma_2(S_2[W_2])$, and so on.

If s_1 has joining tuples in all of $\sigma_2(S_2[W_2])$–$\sigma_n(S_n[W_n])$, then the processing of s_1 by its join pipeline will generate all insertions to the result caused by the insertion of s_1 into $\sigma_1(S_1[W_1])$. Otherwise, processing on s_1 will stop at the first window that does not contain any joining tuples. Similar processing occurs when a tuple s_1, satisfying σ_1, is deleted from $S_1[W_1]$, i.e., when s_1 expires from the window. In this case, the processing of s_1 by its join pipeline will generate all deletions from the join result caused by the deletion of s_1 from $\sigma_1(S_1[W_1])$. Furthermore, similar processing occurs for insertions and deletions to the other windows, as shown in Figure 5.1. Note that each input stream in Figure 5.1 has its own order for probing the other windows.

Like all other operators in the STREAM system, an MJoin processes the insertions and deletions to the input windows in nondecreasing order of timestamp across all windows; recall Section 2.2.5 from Chapter 2. Each insertion or deletion to a window is processed to completion—which includes join processing as well as update of the data and indexes in the corresponding synopsis—before the next one is processed.

5.1.1 Notation

For exposition, we use a succinct algebraic notation in this chapter: A windowed stream join $\sigma_1(S_1[W_1]) \bowtie \sigma_2(S_2[W_2]) \cdots \bowtie \cdots \sigma_n(S_n[W_n])$ is a continuous n-way join over streams S_1–S_n, with respective CQL window specifications W_1–W_n and filter conditions σ_1–σ_n, whose general form in CQL was shown at the start of this section. Recall from Section 2.1.2 in Chapter 2 that $\sigma_i(S_i[W_i])$ is a time-varying CQL relation. For brevity, will denote $\sigma_i(S_i[W_i])$ as the window R_i, so the windowed stream join is simply $R_1 \bowtie R_2 \cdots \bowtie R_n$. For example, the arrival of an S_i tuple satisfying σ_i, and its insertion into the window $S_i[W_i]$, is now an insertion into the window R_i. Notation used in this chapter is summarized in Table 5.1. For clarity of presentation, we assume that all joins are equi-joins of the form $R_i.attr_j = R_k.attr_l$.

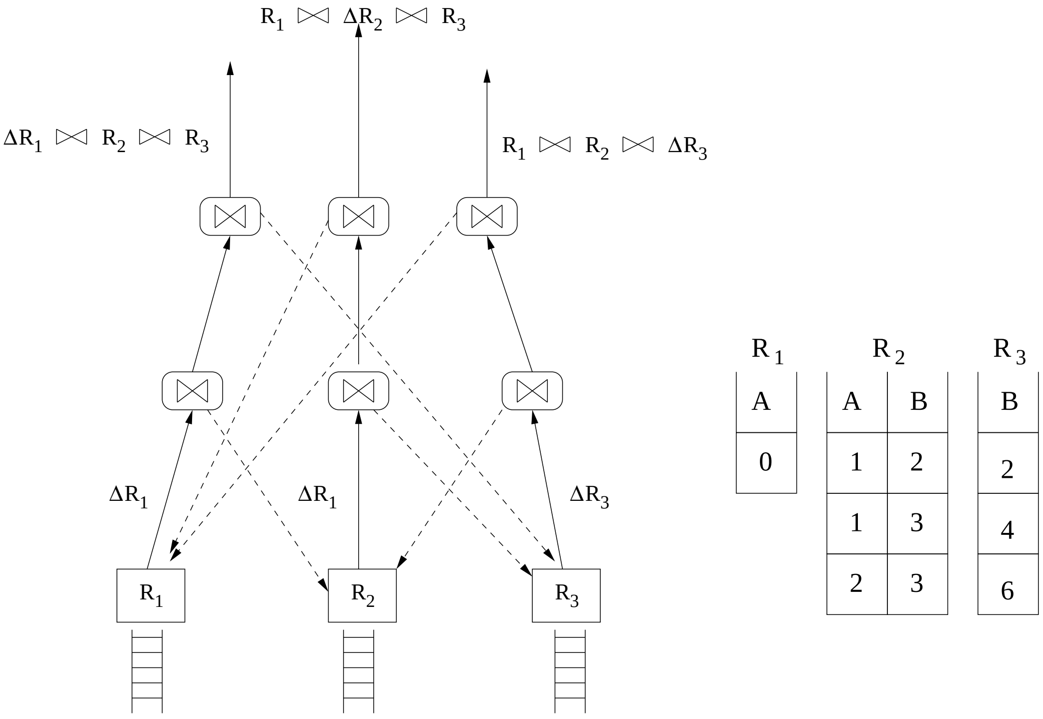
Recall from Section 2.2.5 in Chapter 2 that operators in STREAM query plans that produce a CQL time-varying relation as output actually produce a stream of insertions and deletions to the output relation. We use ΔR_i to denote the insertions and deletions to the window R_i, in nondecreasing order of timestamp. That is, the insertion of a tuple r into R_i at logical time τ generates an element $\langle r, \tau, + \rangle$ in ΔR_i. (Recall from Section 2.2.1 in Chapter 2 that *elements* are tuple-timestamp-flag triples, with the "+" flag denoting insertion and the "$-$" flag denoting deletion.) Similarly, the deletion of a tuple r from R_i at logical time τ generates an element $\langle r, \tau, - \rangle$ in ΔR_i. When we do not need to distinguish whether an element in ΔR_i is an insertion or a deletion, we simply refer to it as a tuple in ΔR_i. The result required for a windowed stream join is the stream of insertions and deletions to the n-way join result based on the tuples in ΔR_1, …, ΔR_n and the time-varying contents of windows R_1, R_2, …, R_n.

Notation	Description
n	Number of joining streams
S_i	ith joining stream
W_i	CQL window specification over S_i
σ_i	Filter conditions over S_i
R_i	Window R_i denotes $\sigma_i(S_i[W_i])$
ΔR_i	The insertions and deletions to R_i in nondecreasing order of timestamp
$\bowtie_{i_1}, \ldots, \bowtie_{i_{n-1}}$	ith pipeline processing ΔR_i; the join operator $\bowtie_{i_j}$ joins its input with R_{i_j}
C_{ijk}	Cache of segment $\bowtie_{i_j}, \bowtie_{i_{j+1}}, \ldots, \bowtie_{i_k}$
K_{ijk}	Cache key of C_{ijk}
benefit(C_{ijk})	Avg. processing cost per unit time when C_{ijk} is not used minus when it is used
cost(C_{ijk})	Avg. processing cost per unit time to maintain C_{ijk} on updates to $R_{i_j}, \ldots, R_{i_k}$
W	Our online estimate for any statistic is the avg. of its W most recent measurements

Table 5.1: Notation used in this chapter

In our algebraic notation, an MJoin to process an n-way windowed stream join consists of n pipelines, where the ith pipeline processes tuples in ΔR_i. To process a tuple r in ΔR_i, r is joined with the other $n-1$ windows in some order $R_{i_1}, \ldots, R_{i_{n-1}}$ to generate the corresponding insertions and deletions to the n-way join result. ΔR_i's pipeline is represented as $\bowtie_{i_1}, \bowtie_{i_2}, \ldots, \bowtie_{i_{n-1}}$ where $\bowtie_{i_j}$ denotes the join operator that performs the join with R_{i_j}. A tuple r input to $\bowtie_{i_j}$ is the concatenation of a tuple each from $R_i, R_{i_1}, \ldots, R_{i_{j-1}}$ such that r satisfies all the join predicates among $R_i, R_{i_1}, \ldots, R_{i_{j-1}}$. $\bowtie_{i_j}$ joins r with R_{i_j}, using indexes, enforcing all join predicates between R_{i_j} and $R_i, R_{i_1}, \ldots, R_{i_{j-1}}$. Recall from Section 2.2.3 in Chapter 2 that (hash) indexes are present on all join attributes in the window synopses. For each joining tuple $r_j \in R_{i_j}$, $\bowtie_{i_j}$ forwards $r \cdot r_j$ to the next operator in the pipeline.

Example 5.1.1 Consider a three-way windowed stream join $R_1 \bowtie_{R_1.A=R_2.A} R_2 \bowtie_{R_2.B=R_3.B} R_3$. Figure 5.2(a) shows an MJoin for this windowed stream join. Suppose the contents


$R_1 \bowtie \Delta R_2 \bowtie R_3$
$\Delta R_1 \bowtie R_2 \bowtie R_3$
$R_1 \bowtie R_2 \bowtie \Delta R_3$
ΔR_1
ΔR_1
ΔR_3
R_1
R_2
R_3

R_1

A
0

R_2

A	B
1	2
1	3
2	3

R_3

B
2
4
6

(a) MJoin for $R_1 \bowtie R_2 \bowtie R_3$ (b) Sample data

Figure 5.2: Plan and data used in examples

of the windows are as shown in Figure 5.2(b) and an insertion $\langle 1 \rangle$ on ΔR_1 is processed next. The first join operator in R_1's pipeline joins $\langle 1 \rangle$ with R_2, producing two intermediate tuples $\langle 1, 1, 2 \rangle$ and $\langle 1, 1, 3 \rangle$. These tuples are joined with R_3 by the second join operator, producing the output insertion $\langle 1, 1, 2, 2 \rangle$. Finally, $\langle 1 \rangle$ is inserted into R_1. □

5.2 A-Greedy for MJoins

An MJoin consists of pipelines containing commutative join operators. One option for ordering these pipelines adaptively is to use the A-Greedy algorithm from Chapter 4 for ordering pipelined filters. Unfortunately, A-Greedy does not apply here directly because, unlike a filter, a join operator may produce an arbitrary number of output tuples per input tuple. However, it turns out that an efficient way to execute the join pipelines for an input

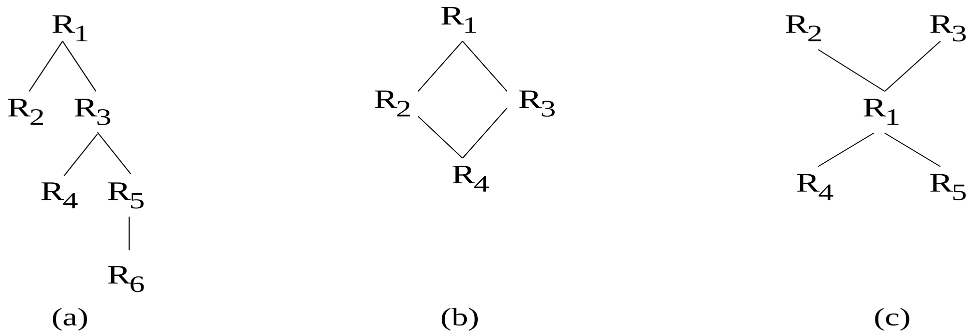


Figure 5.3: Acyclic, cyclic, and star joins

tuple r is to first have a low-overhead *probe phase* to see whether r will produce a join result at all. If it does, then a second *computation phase* performs the full join computation to produce all join results generated by r.

In the two-phase execution strategy for a join pipeline, the probe phase turns out to be an instance of pipelined filters. Also, we can show that the order used for evaluating the join operators in the computation phase does not affect the overall cost. Therefore, the two-phase execution strategy maps the problem of ordering join pipelines to the problem of ordering pipelined filters. With this mapping, the A-Greedy algorithm (and its variants) can be applied directly to order the join pipelines in an MJoin.

5.2.1 Note on Join Graphs

The details of our two-phase execution strategy for join pipelines depend on the *join graph* of the windowed stream join. We give a brief introduction to join graphs before describing our algorithms. The join graph of a windowed stream join $R_1 \bowtie R_2 \cdots \bowtie R_n$ is an undirected graph with the n windows R_1, R_2, ..., R_n as vertices, and an edge between R_i and R_j if and only if there is a join predicate between them. Join graphs can be *acyclic* as in Figure 5.3(a), or *cyclic* as in 5.3(b). We refer to windowed stream joins with acyclic (cyclic) join graphs as acyclic (cyclic) joins. A specific type of acyclic join seen frequently in practice is a *star join*, as shown in Figure 5.3(c). We begin by describing our algorithm to process star joins adaptively.

5.2.2 Adaptive Ordering of Star Joins

First consider processing updates in the *root* (center) window of the star join. (For example, R_1 is the root window of the star join shown in Figure 5.3(c).) Without loss of generality, let the root window be R_1, so we want to process tuples in ΔR_1. We denote ΔR_1's pipeline as O = $\bowtie_{i_1}, \bowtie_{i_2}, \ldots, \bowtie_{i_{n-1}}$. A tuple $r_1 \in \Delta R_1$ is processed in two phases—*drop probing* and *output generation*—applied sequentially:

- *Drop probing phase:* In the drop probing phase, each window R_j ($j \neq 1$) is probed with r_1 to find whether r_1 joins with a tuple in R_j or not. The probes are done independently, but in the order specified by O. If r_1 does not join with any tuple in R_j, i.e., R_j drops r_1, then we do not further process r_1. Otherwise, once we know that R_j does not drop r_1, we move on to probe the next window in the order specified by O, without generating any more matches in R_j. Note that drop probing is exactly pipelined filter processing, with the windows acting as filters. If none of R_2–R_n drop r_1, then r_1 goes into the output generation phase.

- *Output generation phase:* The output generation phase uses a multiway nested-loop join [57, 78], using indexes, to output the complete join result generated by r_1. This phase does not repeat the join computation done in the drop probing phase for r_1. For this purpose, if R_j does not drop r_1 during drop probing, then we save enough information so that the join with R_j can be continued during output generation from where it was left off during drop probing. For example, if the join of r_1 with R_j is based on a full scan of R_j, then we save the position of the last R_j tuple that was compared with r_1 during drop probing.

The multiway nested-loop join used during output generation does not materialize intermediate join results. Therefore, the unit-time cost of output generation in ΔR_1's pipeline is the sum of the cost of accessing joining tuples in R_2–R_n and the cost of creating new result tuples. Suppose p_i tuples in R_i are accessed on average per tuple in ΔR_1 during output generation. Then, the unit-time cost of accessing joining tuples is O(*rate*$(\Delta R_1) \times \Pi_{i=2}^{n} p_i$), assuming all join processing happens in memory. *rate*(ΔR_1) is the rate of insertions and deletions in R_1. The unit-time cost of creating new join result tuples is O(*rate*$(\Delta R_1) \times u$), where u is the average number of join results per ΔR_1 tuple. Note that both these costs are independent of the order of join operators used by the multiway nested-loop join for output generation.

The unit-time cost of processing updates in the root window R_1 consists of the cost of drop probing and the cost of output generation. Since the cost of output generation is independent of the order of operators in the join pipeline, only the cost of drop probing affects the relative performance of a given order. Furthermore, drop probing is an instance of pipelined filters. Therefore, the A-Greedy algorithm from Section 4.3 in Chapter 4 (and its variants) can be used directly to order the operators adaptively in the join pipeline. Because of the equivalence to pipelined filters, Theorem 4.3.1 from Chapter 4 applies in this case, ensuring the theoretical guarantees on convergence properties. Therefore, by using A-Greedy for ordering ΔR_1's pipeline, we can guarantee that the unit-time cost of the ordering produced when statistics stabilize is at most four times the unit-time cost of the optimal ordering.

The discussion so far focused on the processing of updates in the root window R_1 of the star join. We now consider nonroot windows. As a heuristic, we avoid using Cartesian products in the join pipelines. (Note that Cartesian products are inevitable if the join graph is not connected.) In our experiments using the STREAM system (Section 5.8), we have never come across a scenario where Cartesian products are part of the optimal MJoin. Furthermore, Cartesian products increase the space of possible plans and the number of statistics to be monitored, typically increasing run-time overhead considerably without any improvement in plan quality. The downside of avoiding Cartesian products is that we disallow certain filter (join) orderings in A-Greedy, so we no longer get the theoretical guarantees on convergence properties provided by A-Greedy in Theorem 4.3.1.

Let R_j be a nonroot window. For a tuple r_j in ΔR_j, the join with the root window R_1 will be done first to avoid a Cartesian product. If r_j does not join with any tuples in R_1, then we do not further process r_j. Otherwise, drop probing and output generation (if required) will be invoked for each tuple of $r_j \bowtie R_1$ to compute their join with R_2, $\ldots$, R_{j-1}, R_{j+1}, $\ldots$, R_n. This computation is similar to the processing of updates in the root window.

5.2.3 Adaptive Ordering of Acyclic and Cyclic Joins

Consider the acyclic join depicted in Figure 5.3(a). Because of our heuristic of avoiding Cartesian products, we can think of the edges in this join graph as specifying a set of precedence constraints on the join order. For example, the join with R_4 will be a Cartesian product in ΔR_1's pipeline unless the join with R_3 in the pipeline precedes the join with R_4. Therefore, O = $\bowtie_2$, $\bowtie_3$, $\bowtie_4$, $\bowtie_5$, $\bowtie_6$ is a possible pipeline for ΔR_1, but O' = $\bowtie_2$, $\bowtie_4$, $\bowtie_3$, $\bowtie_5$, $\bowtie_6$ is not. Specifically, $\bowtie_3$ must precede $\bowtie_4$ in ΔR_1's pipeline.

Note that the single path from R_1 to R_4 in Figure 5.3(a) goes through R_3. Consequently, R_4 will drop a tuple $r_1 \in \Delta R_1$ only if it drops every R_3 tuple in $r_1 \bowtie R_3$. We generalize this property to apply the two-phase execution strategy and the A-Greedy algorithm to acyclic joins. We treat each join operator $\bowtie_j$ in ΔR_i's pipeline as a *logical filter* corresponding to the join along the unique path from R_i to R_j in the join graph. We apply the A-Greedy algorithm from Chapter 4 directly to these logical filters, then generate the join ordering for ΔR_i's pipeline from A-Greedy's ordering of the logical filters. For example, the logical filters input to A-Greedy to order the join operators in ΔR_1's pipeline for the acyclic join graph in Figure 5.3(a) are: $F_1 = R_2$, $F_2 = R_3$, $F_3 = R_3 \bowtie R_4$, $F_4 = R_3 \bowtie R_5$, and $F_5 = R_3 \bowtie R_5 \bowtie R_6$. A-Greedy's ordering of these logical filters is mapped to an ordering of the join operators by enforcing the precedence constraints from the join graph. For example, if A-Greedy orders the filters as F_3, F_2, F_1, F_4, F_5, then ΔR_1's pipeline will be set to $\bowtie_3$, $\bowtie_4$, $\bowtie_2$, $\bowtie_5$, $\bowtie_6$.

The logical filters are profiled and A-Greedy's matrix view is maintained as described in Section 4.3 in Chapter 4. We use the above technique for acyclic joins to handle cyclic joins as well: We choose a spanning tree for the cyclic join graph, then treat the join as the corresponding acyclic join [10, 78, 94].

5.3 XJoins

An MJoin does not maintain any intermediate *join subresults*. For example, consider the MJoin in Figure 5.4 for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$. Each tuple $r_4 \in \Delta R_4$ will be joined first with R_2, the resulting tuples will be joined with R_1, and so on. The intermediate subresults

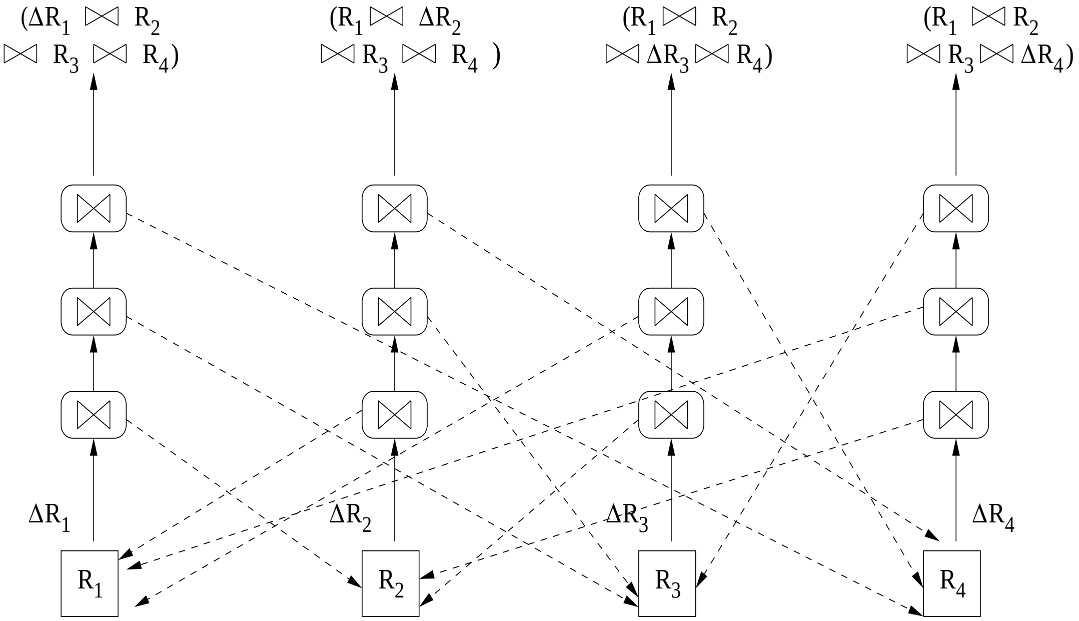


Figure 5.4: MJoin for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$

generated—$r_4 \bowtie R_2$, $r_4 \bowtie R_2 \bowtie R_1$, $r_4 \bowtie R_2 \bowtie R_1 \bowtie R_3$—are discarded after r_4 is processed. Not maintaining such intermediate results can limit performance in scenarios where values of join attributes repeat among tuples in ΔR_4. For example, if $r_4' \in \Delta R_4$ arrives after r_4 with the same join-attribute values as r_4, then the MJoin will repeat the computation it did for r_4, to generate the same join subresults again.

In contrast to MJoins, an *XJoin* [112], which is a tree of two-way joins, maintains a *fully-materialized* join subresult for each intermediate two-way join in the plan. XJoins are implemented in the STREAM system by the `binary-join` operator outlined in Table 2.1 in Chapter 2. Figure 5.5(a) shows an example XJoin for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$, which uses a left-deep tree and maintains two join subresults: $R_1 \bowtie R_2$ and $R_1 \bowtie R_2 \bowtie R_3$. Each new tuple $r_4 \in \Delta R_4$ can now be joined with $R_1 \bowtie R_2 \bowtie R_3$ directly. Consequently, intermediate join subresults in $R_1 \bowtie R_2 \bowtie R_3$ need not be computed again and again for tuples in ΔR_4 with repeated join values.

MJoins and XJoins actually lie at two extremes of a spectrum of plans for windowed stream joins. Figure 5.6 shows an example of an intermediate plan in this spectrum. This

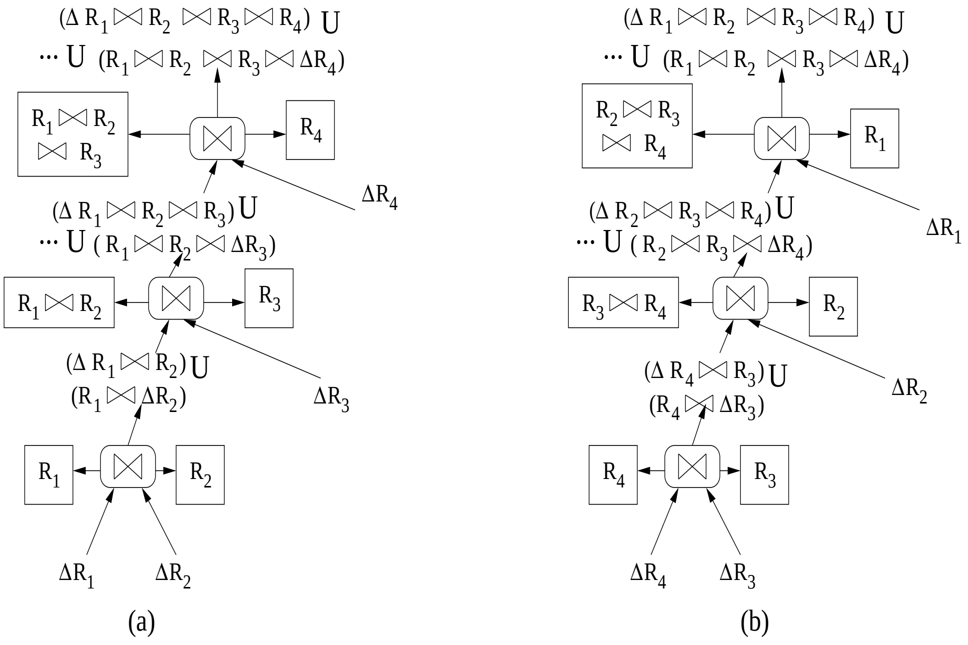

$(\Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4) \cup$
$\cdots \cup (R_1 \bowtie R_2 \bowtie R_3 \bowtie \Delta R_4)$
$R_1 \bowtie R_2 \bowtie R_3$
R_4
ΔR_4
$(\Delta R_1 \bowtie R_2 \bowtie R_3) \cup$
$\cdots \cup (R_1 \bowtie R_2 \bowtie \Delta R_3)$
$R_1 \bowtie R_2$
R_3
$(\Delta R_1 \bowtie R_2) \cup$
$(R_1 \bowtie \Delta R_2)$
ΔR_3
R_1
R_2
ΔR_1
ΔR_2
(a)
$(\Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4) \cup$
$\cdots \cup (R_1 \bowtie R_2 \bowtie R_3 \bowtie \Delta R_4)$
$R_2 \bowtie R_3 \bowtie R_4$
R_1
ΔR_1
$(\Delta R_2 \bowtie R_3 \bowtie R_4) \cup$
$\cdots \cup (R_2 \bowtie R_3 \bowtie \Delta R_4)$
$R_3 \bowtie R_4$
R_2
$(\Delta R_4 \bowtie R_3) \cup$
$(R_4 \bowtie \Delta R_3)$
ΔR_2
R_4
R_3
ΔR_4
ΔR_3
(b)

Figure 5.5: XJoins for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$

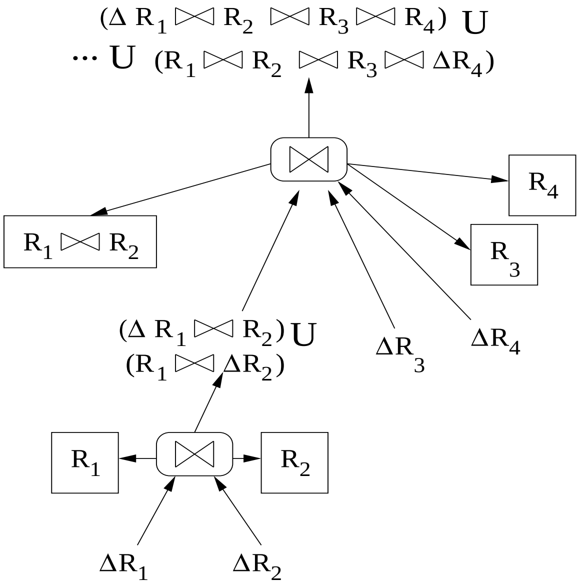

$(\Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4) \cup$
$\cdots \cup (R_1 \bowtie R_2 \bowtie R_3 \bowtie \Delta R_4)$
R_4
$R_1 \bowtie R_2$
R_3
$(\Delta R_1 \bowtie R_2) \cup$
$(R_1 \bowtie \Delta R_2)$
ΔR_3
ΔR_4
R_1
R_2
ΔR_1
ΔR_2

Figure 5.6: Tree of MJoins for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$

plan is a tree consisting of two MJoins: an MJoin of R_1 and R_2, followed by an MJoin of the resulting tuples with R_3 and R_4. It maintains the join subresult $R_1 \bowtie R_2$ only. (We will see later in this chapter that a large fraction of this spectrum consists of plans that cannot even be represented as a tree of MJoins.) Intermediate plans may be significantly more efficient than any MJoin or XJoin in some scenarios. For example, if the tuple-arrival rates in ΔR_3 and ΔR_4 are much higher than that in ΔR_1 and ΔR_2, then the plan in Figure 5.6 may outperform the MJoin in Figure 5.4, both the XJoins in Figure 5.5, and any other MJoin or XJoin. As usual, an MJoin may unnecessarily recompute the joining $R_1 \bowtie R_2$ tuples for incoming ΔR_3 and ΔR_4 tuples. An XJoin may incur high overhead to maintain a join subresult involving R_3, R_4, or both. This simple example motivates the need to consider the entire spectrum of plans when choosing the plan to execute a windowed stream join.

Choosing an efficient join plan for the current stream and system conditions is a difficult problem as we have just motivated. In addition, it is important to adapt as these conditions change. For example, if R_1 has a long burst of updates, then we may want to switch to the XJoin in Figure 5.5(b) from the XJoin in Figure 5.5(a). Such plan switching can be expensive if plans carry a lot of run-time state like the fully-materialized join subresults in XJoins [121]: The state in the old plan has to be disposed (e.g., to reclaim memory) and the complete state in the new plan has to be generated and materialized before we can start processing new input tuples. In volatile stream and system conditions, it is important to keep the cost of switching between plans low, so as to avoid a significant pause in throughout when conditions change.

We propose a new approach for windowed stream joins that addresses the problems with MJoins and XJoins. Our approach starts with MJoins and adds *subresult caches* adaptively in the MJoin pipelines. This approach has many advantages. Our plan space captures the entire spectrum of join plans from MJoins to XJoins. Caches can be added, populated incrementally, and dropped with little overhead. Therefore, the cost of switching from one MJoin with caches to another is low, enabling fast adaptivity when stream or system conditions change. Finally, unlike XJoins, the performance of MJoins with caches degrades gracefully when memory availability decreases. Next we describe how caches can be used in MJoin pipelines, then describe our algorithm for adding and removing caches adaptively.

5.4 Caches in MJoin Pipelines

Consider an MJoin plan for a windowed stream join over windows $R_1, R_2, \ldots, R_n$. In this environment, a *cache* is an associative store used during join processing that corresponds to a contiguous segment of join operators in a pipeline. We use the notation C_{ijk} to represent a cache in R_i's pipeline corresponding to the operator segment $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$. Logically, C_{ijk} contains *key-value* pairs (u, v), where the key u is the set of join attributes, denoted K_{ijk}, between a window in $\{R_i, R_{i_1}, R_{i_2}, \ldots, R_{i_{j-1}}\}$ and a window in $\{R_{i_j}, \ldots, R_{i_k}\}$. In other words, K_{ijk} is the set of join attributes between the windows in the ith pipeline that appear before the cached segment and the windows in the cached segment. K_{ijk} is called the *cache key* for C_{ijk}. C_{ijk} is required to satisfy a *consistency invariant*, which guarantees that all cached entries are current and correct. However, we do not assume or make any guarantees on completeness, i.e., a cache may contain only a subset of the corresponding join subresult.

Definition 5.4.1 **(Consistency Invariant***) Cache C_{ijk} satisfies the consistency invariant if for all $(u, v) \in C_{ijk}$, $v = \sigma_{K_{ijk}=u}(R_{i_j} \bowtie \ldots \bowtie R_{i_k})$.* □

Each cache C_{ijk} supports the following operations:

- *create*(u, v) for a key-value pair (u, v), which adds the key-value pair to the cache.
- *probe*(u) for a key u, which returns a *hit* with value v if $(u, v) \in C_{ijk}$, and a *miss* otherwise.
- *insert*(u, r) and *delete*(u, r) for a key u and a tuple $r \in R_{i_j} \bowtie \cdots \bowtie R_{i_k}$. If $(u, v) \in C_{ijk}$, *insert*(u, r) adds r to set v, otherwise *insert*(u, r) is ignored. Similarly, if $(u, v) \in C_{ijk}$, *delete*(u, r) removes r from set v, otherwise *delete*(u, r) is ignored.

Intuitively, a cache C_{ijk} is placed in R_i's pipeline just before $\bowtie_{i_j}$. When a tuple r reaches $\bowtie_{i_j}$ during join processing, we first probe C_{ijk} with $\pi_{K_{ijk}}(r)$. If we get a hit, then we directly have the tuples in $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ that join with r, and we save the work that would otherwise have been performed to generate them. If we miss, then we continue regular join processing and add the computed result (which could be empty) to C_{ijk} for later probes. Because caches need not provide any guarantee on completeness, caches can be added at any time without stalling the pipelines, then populated incrementally. They can also be dropped at any time. In more general terms relating back to Section 5.3, plan-switching costs are negligible.

Specifically, to use a cache C_{ijk} during join processing, a *CacheLookup* operator L and a *CacheUpdate* operator U are placed in R_i's pipeline. L is placed just before $\bowtie_{i_j}$ and U is placed just after $\bowtie_{i_k}$. Recall that $\bowtie_{i_j}$ is the first operator in the segment corresponding to C_{ijk} and $\bowtie_{i_k}$ is the last operator in this segment. When L receives a (possibly composite) tuple r, it probes the cache with $u = \pi_{K_{ijk}}(r)$. If the cache returns a hit with a value v, then L bypasses operators $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$, and U, and forwards $r \cdot s$ for each $s \in v$ to the operator following U. If there is a cache miss, L sends r on to $\bowtie_{i_j}$ to continue with regular join processing through $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$, resulting in the computation of $v = \sigma_{K_{ijk}=u}(R_{i_j} \bowtie \cdots \bowtie R_{i_k})$, and U adds (u, v) to C_{ijk} using *create*(u, v).

Example 5.4.1 Consider the three-way join from Example 5.1.1 and the current contents of windows R_1, R_2, and R_3 in Figure 5.2(b). Figure 5.7 shows an MJoin with a cache, currently empty, for the R_2, R_3 segment in ΔR_1's pipeline. Suppose tuple $\langle 1 \rangle \in \Delta R_1$ is processed next, so the *CacheLookup* operator probes the cache with $\langle 1 \rangle$. This probe misses and $\langle 1 \rangle$ is forwarded to the join operators. The joining $\langle 1, 2, 2 \rangle$ tuple is inserted into the cache by the *CacheUpdate* operator in ΔR_1's pipeline. If the tuple processed next is also $\langle 1 \rangle \in \Delta R_1$, then the cache probe is a hit and the join result is output immediately. □

Next we describe the maintenance of C_{ijk} on updates to $R_{i_l} \in \{R_{i_j}, \ldots, R_{i_k}\}$. Suppose r is inserted into R_{i_l}. If C_{ijk} contains an entry (u, v) for $u \in \pi_{K_{ijk}}(R_{i_j} \bowtie \cdots R_{i_{l-1}} \bowtie r \bowtie R_{i_{l+1}} \bowtie \cdots \bowtie R_{i_k})$, then the tuples $\sigma_{K_{ijk}=u}(R_{i_j} \bowtie \cdots R_{i_{l-1}} \bowtie r \bowtie R_{i_{l+1}} \bowtie \cdots \bowtie R_{i_k})$ must be added to v to maintain the consistency invariant; similarly for a deletion. If C_{ijk} does not contain an entry (u, v) for a $u \in \pi_{K_{ijk}}(R_{i_j} \bowtie \cdots R_{i_{l-1}} \bowtie r \bowtie R_{i_{l+1}} \cdots \bowtie R_{i_k})$, then the consistency invariant is not affected and nothing needs to be done.

Example 5.4.2 We continue with Example 5.4.1. Recall that the cache currently contains one entry (u, v) with $u = \langle 1 \rangle$ and $v = \{\langle 1, 2, 2 \rangle\}$. Suppose tuple $\langle 3 \rangle$ is inserted into R_3 next. We must update (u, v) by adding $\sigma_{R_2.B=u}(R_2 \bowtie \{\langle 3 \rangle\}) = \langle 1, 3, 3 \rangle$ to v so that a new tuple $\langle 1 \rangle \in \Delta R_1$ will correctly produce two output tuples, $\langle 1, 1, 2, 2 \rangle$ and $\langle 1, 1, 3, 3 \rangle$. □

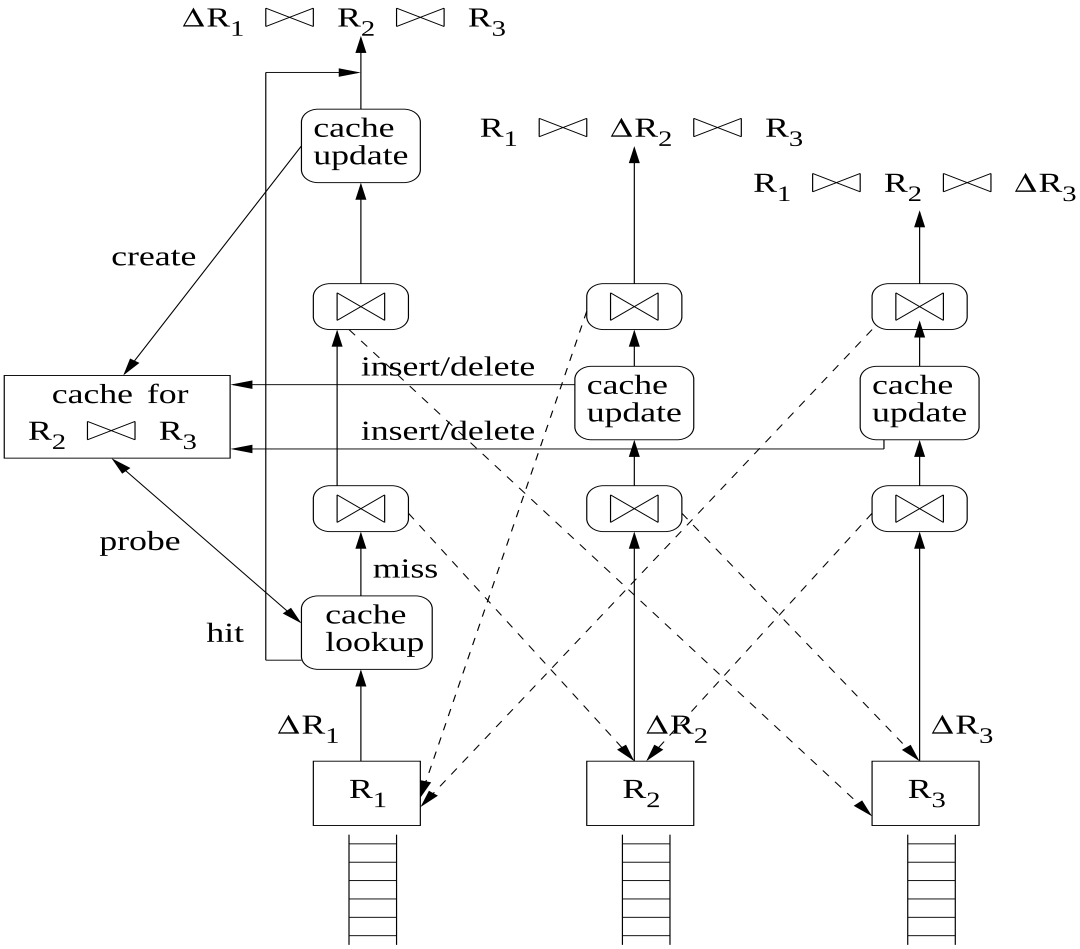


Figure 5.7: Plan with a $R_2 \bowtie R_3$ cache

To maintain C_{ijk}'s consistency when any of $R_{i_j}, \ldots, R_{i_k}$ is updated, we must compute the corresponding updates to $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$. If these updates are not computed as part of regular join processing, then we must compute them separately. We specify a *prefix invariant* (below) that is both necessary and sufficient to ensure that all updates to $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ are computed as part of regular join processing. We limit the discussion in the rest of this section to caches that satisfy the prefix invariant. This restriction is revisited in Section 5.5 and relaxed in Section 5.7.

Definition 5.4.2 *(***Prefix Invariant***) Cache C_{ijk} satisfies the prefix invariant if the first $k-j$ join operators in ΔR_{i_l}'s pipeline, for each $R_{i_l} \in \{R_{i_j}, \ldots, R_{i_k}\}$, correspond to joins with one of $\{R_{i_j}, \ldots, R_{i_k}\} - R_{i_l}$.* □

In other words, we require ΔR_{i_l}'s pipeline to join incoming tuples with the other windows in $R_{i_j}, \ldots, R_{i_k}$ in some order, before joining with the remaining windows in $R_1, \ldots, R_n$. Then, the tuples produced by the segment consisting of the first $k - j$ join operators in ΔR_{i_l}'s pipeline are the updates to $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ for each update to R_{i_l}.

Example 5.4.3 The plan in Figure 5.7 satisfies the prefix invariant for the cache corresponding to the R_2, R_3 segment in ΔR_1's pipeline: ΔR_2's pipeline contains R_3 for the first join, and vice-versa. However, a cache corresponding to the R_2, R_1 segment in ΔR_3's pipeline would not satisfy the prefix invariant because the join with R_1 is not the first join in ΔR_2's pipeline. □

To maintain the consistency of C_{ijk}, we place $k - j + 1$ *CacheUpdate* operators: U_l, $j \leq l \leq k$, is placed just before the $(k - j + 1)$st join operator in ΔR_{i_l}'s pipeline. Because of the prefix invariant, U_l has access to all updates to $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ caused by an update to R_{i_l}. U_l extracts the cache key from each tuple and updates C_{ijk} by making the required *insert* or *delete* call.

Example 5.4.4 Let us revisit Example 5.4.2. When the insertion $\langle 3 \rangle$ is processed by ΔR_3's pipeline, the intermediate tuples $\langle 1, 3, 3 \rangle$ and $\langle 2, 3, 3 \rangle$ pass through the *CacheUpdate* operator, which makes the corresponding *insert* calls to the cache. Since the cache key $\langle 1 \rangle$ for $\langle 1, 3, 3 \rangle$ is present, $\langle 1, 3, 3 \rangle$ is added to its associated value. Since the cache key $\langle 2 \rangle$ for $\langle 2, 3, 3 \rangle$ is not present, this insert call will be ignored by the cache, maintaining consistency.

□

5.4.1 Implementation of Caches in STREAM

Caches are implemented as synopses in STREAM. Each cache synopsis contains a hash index built on the cache key. The number of buckets in the hash table, which is constant, is chosen based on the expected cache size. Cache sizes are estimated online before we start using the caches, as described in Section 5.5.3.

Actual tuples are never copied into the caches. The cached values are always sets of references to joining tuples in the windows. Note that the windows involved in the query plan are themselves stored as STREAM synopses; recall Section 2.2.3 in Chapter 2. The memory for storing the cache synopses is allocated from a pool of memory reserved for caches in each MJoin. This memory is allocated dynamically to different caches, as described in Section 5.6.

Operations on caches are implemented in a straightforward manner using the hash index on the cache key. When an *insert*(u, r) is invoked on a cache that contains an entry (u, v), we add r to v. Deletes are the converse. *create*(u, v) adds a new entry, potentially replacing an existing entry. We use a simple *direct-mapped* cache replacement scheme to keep its runtime overhead low: If a new key hashes to a bucket that already contains another key (i.e., a *collision*), then we simply replace the existing entry with the new one, without violating consistency.

5.5 Adaptive Caching

The performance of our windowed stream join plans depends on join ordering as well as caching, where caching includes both cache selection and memory allocation to caches. Instead of optimizing both ordering and caching in concert, we take a modular approach as a first step in attacking this complex problem:

1. We first order the join operators in the MJoin pipelines adaptively using A-Greedy as described in Section 5.2.

2. For a given join ordering, we then focus on adaptive selection of caches to optimize performance for that ordering.

3. Given an ordering and caching scheme, we allocate memory adaptively to the selected caches.

Apart from making the overall problem easier to handle, our modular approach is motivated by the observation that the ordering and caching problems, which are NP-Hard individually, when considered separately permit efficient near-optimal approximation algorithms (Sections 5.2 and 5.5.4). Full treatment of the integrated ordering-caching problem is beyond the scope of this thesis.

Now let us focus on the problem of choosing caches adaptively for a given join ordering. We first restrict our plan space by considering only caches that satisfy the prefix invariant (Definition 5.4.2), and later drop this restriction in Section 5.7. There are compelling reasons to consider this restricted plan space:

1. The cache selection problem is NP-Hard even in this restricted plan space. However, the prefix invariant enables efficient algorithms to find solutions with strong quality guarantees (Section 5.5.4).

2. The prefix invariant ensures that all updates to caches are computed at no extra cost as part of regular join processing (Section 5.4).

3. This plan space subsumes the space of windowed stream join plans that are *trees of MJoins*. (See [114] for a detailed motivation of this plan space.) Plans from this space are very effective at improving basic MJoin performance (Section 5.8).

For presentation, we will first assume that we have enough memory for all selected caches. In Section 5.6 we describe our algorithm for allocating memory adaptively to caches.

5.5.1 Cache Cost Model

Intuitively, the *benefit* we get from using a cache C, denoted *benefit*(C), is the processing cost without the cache minus the processing cost with the cache. Recall that we consider processing costs in terms of the unit-time cost metric.

Let d_{ij} be the average number of tuples processed per unit time by the join operator $\bowtie_{i_j}$, and let c_{ij} be the average processing cost per tuple for $\bowtie_{i_j}$. When we do not use cache C_{ijk}, the average processing cost per unit time for the segment $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$ is $\sum_{l=j}^{k} d_{il}c_{il}$. When we use C_{ijk}, it is probed for each of the d_{ij} tuples that reach $\bowtie_{i_j}$. If there is a cache hit, the result of the join with $R_{i_j}, \ldots, R_{i_k}$ is available in the cache and no work needs to be done by $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$. In case of a cache miss, regular pipeline processing continues in $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$ and the joining tuples are inserted into the cache. Thus:

$$\begin{aligned}\textit{benefit}(C_{ijk}) &= \textstyle\sum_{l=j}^{k} d_{il}c_{il} - d_{ij} \times \textit{probe\_cost}(C_{ijk}) - \\ &\quad \textit{miss\_prob}(C_{ijk}) \times (\textstyle\sum_{l=j}^{k} d_{il}c_{il} + d_{i,k+1} \times \textit{update\_cost}(C_{ijk}))\end{aligned}$$

$d_{i,k+1}$ is the average number of tuples processed by the join operator following $\bowtie_{i_k}$, so it is the average number of tuples output by $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$ per unit time. (The $d_{i,k+1}$ notation is slightly different from the regular d_{ij} notation for clarity.) On average, *miss_prob*$(C_{ijk}) \times d_{i,k+1}$ tuples will need to be inserted into C_{ijk} because of cache misses.

The cost of using C_{ijk}, denoted *cost*(C_{ijk}), is the cost of maintaining C_{ijk} on updates to $R_{i_j}, \ldots, R_{i_k}$. Recall that because of the prefix invariant, the updates themselves are computed as part of the regular join processing. Thus:

$$\textit{cost}(C_{ijk}) = \textit{update\_cost}(C_{ijk}) \times \textstyle\sum_{l=j}^{k} d_{l,k-j+1}$$

5.5.2 Processing Windowed Stream Joins Adaptively

We are now ready to describe A-Caching—our adaptive algorithm for adding and dropping caches from MJoin pipelines. At any point in time we have a set of pipelines determined by a join ordering algorithm. From these pipelines, we identify a set of *candidate caches*, which are those that satisfy the prefix invariant (Definition 5.4.2). The goal of A-Caching is to ensure that the set of caches being used in the pipelines at any point of time is the subset X of *nonoverlapping* candidate caches that maximizes $\sum_{C \in X}$ *benefit*(C) − *cost*(C), where caches are nonoverlapping if they have no join operators in common. Although there may be certain situations where overlapping caches are beneficial, we do not consider them because they complicate cache management and optimization considerably.

A-Caching instantiates the three generic components of StreaMon. The Profiler maintains online estimates of the benefits and costs of candidate caches. The Re-optimizer ensures that the optimal set of caches is being used at any point of time. The third component is the Executor which executes the join operators in the pipelines. The Executor uses and maintains the caches chosen currently for use by the Re-optimizer.

In addressing re-optimization, we first consider the offline version of the cache selection problem, which is to find the optimal nonoverlapping subset of candidate caches given stable benefits and costs for all caches. We develop an *offline cache selection algorithm* for this problem, then use it to derive the adaptive algorithm employed by the Re-optimizer. Next, we describe how the Profiler performs online cache benefit and cost estimation (Section 5.5.3), then our offline cache selection algorithm (Section 5.5.4), and finally the overall adaptive algorithm (Section 5.5.5).

5.5.3 Estimating Cache Benefits and Costs Online

Let us consider how we can estimate the cost and benefit of a candidate cache online. Since caches can be added and dropped with little overhead, one way to compute *benefit*(C_{ijk}) and *cost*(C_{ijk}) is to force C_{ijk} to be used for some time and see how it performs. Alternatively, *benefit*(C_{ijk}) and *cost*(C_{ijk}) can be estimated from online estimates of the corresponding d_{ij}, c_{ij}, *probe_cost*(C_{ijk}), *update_cost*(C_{ijk}), and *miss_prob*(C_{ijk}). Our implementation of A-Caching currently uses the second technique because online estimation of these parameters can be combined with that for adaptive join ordering, as described next.

From the equations for *benefit*(C_{ijk}) and *cost*(C_{ijk}) in Section 5.5.1, it follows that they can be estimated from online estimates of the appropriate d_{ij}, c_{ij}, *probe_cost*(C_{ijk}), *update_cost*(C_{ijk}), and *miss_prob*(C_{ijk}). We discuss each of these in turn:

- $\mathbf{d_{ij}, c_{ij}}$: To maintain online estimates of d_{ij} and c_{ij}, we profile the complete processing of a fraction of tuples entering the ith pipeline, similar to the profiling for A-Greedy as described in Section 4.3.1 in Chapter 4. Before we start to process a tuple $r \in R_i$ using its pipeline $\ltimes_{i_1}, \ldots, \ltimes_{i_{n-1}}$, we will choose to profile it with a probability p_i. If we choose to profile r, then as r and the intermediate tuples generated by it are processed by $\ltimes_{i_1}, \ldots, \ltimes_{i_{n-1}}$, we maintain two values δ_j and τ_j per join operator $\ltimes_{i_j}$. δ_j is the number of tuples processed by $\ltimes_{i_j}$ as part of the processing for r, and τ_j is the corresponding processing time spent in $\ltimes_{i_j}$. Furthermore, we will not use any caches in this pipeline throughout the processing of r. Once the processing of r is complete, we insert δ_j and τ_j respectively into sliding windows that maintain the last W observations in each case. (Recall from Table 5.1 that our estimate for any statistic is the average of the corresponding W most recent observations.) We use the sum of values in these windows, denoted *sum*(δ_j) and *sum*(τ_j), to estimate d_{ij} and c_{ij} as: $d_{ij} = \textit{rate}(R_i) \times \text{sum}(\delta_j)/W, c_{ij} = \text{sum}(\tau_j)/\text{sum}(\delta_j)$. Note that we measure processing costs in terms of processing time. *rate*(R_i) is the number of R_i tuples processed per unit time, which is maintained in a straightforward manner.

- ***probe_cost***(C_{ijk}), ***update_cost***(C_{ijk}) : Based on our cache implementation described in Section 5.4.1, we can derive expressions for *probe_cost*(C_{ijk}) and *update_cost*(C_{ijk}) in terms of the cache key size, which is constant, and the average number of tuples in a cached entry. The details are straightforward. The average number of tuples in a C_{ijk} entry is $d_{i,k+1}/d_{ij}$.

- ***miss_prob***(C_{ijk}) : When C_{ijk} is being used, *miss_prob*(C_{ijk}) can be observed directly. When C_{ijk} is not being used and we need to estimate *miss_prob*(C_{ijk}), we place a *CacheLookup* operator L just before $\bowtie_{i_j}$ so that L can see the complete stream of tuples processed by $\bowtie_{i_j}$. For each nonoverlapping window of W_d tuples in this stream, L hashes each tuple on K_{ijk} into a *Bloom filter* [24] with αW_d, $\alpha \geq 1$, bits before forwarding the tuple. If b bits in the bloom filter are set after a sequence of W_d tuples have been processed, then an estimate of *miss_prob*(C_{ijk}) is b/W_d. Intuitively, b distinct keys are present among the W_d tuples, and each distinct key will cause a cache miss when it appears for the first time after which it will be cached. As usual, we maintain the last W such observations, and our online estimate of *miss_prob*(C_{ijk}) is the average of these W observations.

5.5.4 Offline Cache Selection Algorithm

Because of the prefix invariant, if two candidate caches in a pipeline overlap, then one of them is a superset of the other. Thus, overlapping caches in a pipeline have a hierarchical relationship, and they can be organized into a tree with each cache C having as its parent the smallest candidate cache in the pipeline that is a strict superset of C.

Example 5.5.1 Consider a six-way equi-join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_6$ on attribute A. Suppose the pipelines are ordered as follows:

1. ΔR_1's pipeline has the order R_2, R_3, R_4, R_5, R_6

2. ΔR_2's pipeline has the order R_1, R_3, R_5, R_4, R_6

3. ΔR_3's pipeline has the order R_2, R_1, R_4, R_5, R_6

4. ΔR_4's pipeline has the order R_5, R_1, R_2, R_3, R_6

5. ΔR_5's pipeline has the order R_4, R_2, R_3, R_1, R_6

6. ΔR_6's pipeline has the order R_2, R_1, R_4, R_5, R_3

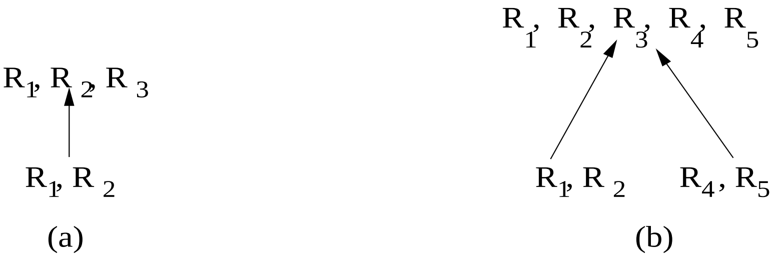


Figure 5.8: Cache hierarchy from Example 5.5.1

The prefix property holds for $\{R_1, R_2\}$, $\{R_4, R_5\}$, $\{R_1, R_2, R_3\}$, and $\{R_1, R_2, R_3, R_4, R_5\}$, which together give rise to the set of candidate caches with at least one join. For example, there are two candidate caches in ΔR_4's pipeline—one corresponding to the R_1, R_2 segment and the other corresponding to the overlapping R_1, R_2, R_3 segment—which can be organized into the tree in Figure 5.8(a). Similarly, there are three candidate caches in ΔR_6's pipeline, which can be organized into the tree in Figure 5.8(b). □

The following theorem states that the optimal set of nonoverlapping candidate caches can be chosen for a single pipeline in linear time.

Theorem 5.5.1 *Given m candidate caches on a single pipeline, the nonoverlapping subset X that maximizes $\sum_{C \in X} \text{benefit}(C) - \text{cost}(C)$ can be found in $O(m)$ time.*

Proof: Since the caches are on the same pipeline, they can be organized into a forest where each tree corresponds to a maximal overlapping subset of caches. The union of the optimal solutions over all trees in this forest gives the optimal solution for the forest. Within a tree, we compute the optimal solution recursively. We traverse the tree bottom-up and derive the optimal set of caches for each subtree T rooted at a cache C. The optimal solution for T is either C or the union of optimal solutions for C's children. This algorithm is $O(m)$. □

The maximum number of candidate caches in a single pipeline is linear in the number of joining windows n, so the above algorithm is $O(n)$. Unfortunately, Theorem 5.5.1 does not extend to candidate caches in multiple pipelines because caches can be *shared* across pipelines.

Definition 5.5.1 *(***Shared Caches***) Candidate caches C_{ijk} and C_{pqr} in different pipelines ($i \neq p$) are shared caches if $R_{i_j}, \ldots, R_{i_k}$ is some permutation of $R_{p_q}, \ldots, R_{p_r}$ and $K_{ijk} = K_{pqr}$.* □

That is, two caches are shared if they cache the same join and have the same key. Sharing is not restricted to pairs of caches—we can have larger groups of shared caches.

Example 5.5.2 Consider the six-way equi-join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_6$ from Example 5.5.1. The candidate cache corresponding to R_1, R_2 is shared in the pipelines for ΔR_3, ΔR_4, and ΔR_6. The candidate cache corresponding to R_1, R_2, R_3 is shared in the pipelines for ΔR_4 and ΔR_5. Note that the key for each cache is attribute A, which is equated across all windows in the join. □

The obvious advantage of using shared caches is that their maintenance costs can be shared. The total benefit of using a group of shared caches is the sum of their individual benefits, but the total cost is simply the cost of a single cache. Thus, from the perspective of a single cache, it is always advantageous to use it in multiple pipelines so that the benefits add up while the cost is fixed. However, the combination of sharing and the need to choose nonoverlapping caches makes the offline cache selection problem NP-Hard. The proof of the following theorem is given in Reference [89].

Theorem 5.5.2 *Given m candidate caches not all on the same pipeline, finding the nonoverlapping subset X that maximizes $\sum_{C \in X}$ benefit$(C) -$cost(C) is NP-Hard. However, if there are no shared caches, then the optimal solution can be found in $O(m)$ time.* □

Our objective of picking the optimal nonoverlapping subset X of candidate caches that maximizes $\sum_{C \in X}$ *benefit*(C) − *cost*(C) can be stated alternatively as picking the subset that minimizes $\sum_{C \in X}$ *proc*(C) + *cost*(C), where *proc*(C_{ijk}) is the average cost per unit time of using C_{ijk} in ΔR_i's pipeline. Note that *proc*(C_{ijk}) does not include the cost of maintaining C_{ijk} on updates to $R_{i_j}, \ldots, R_{i_k}$. From Section 5.5.1:

$$\begin{aligned} \textit{proc}(C_{ijk}) = d_{ij} \times \textit{probe\_cost}(C_{ijk}) + \textit{miss\_prob}(C_{ijk}) \\ \times (\textstyle\sum_{l=j}^{k} d_{il} c_{il} + d_{i,k+1} \times \textit{update\_cost}(C_{ijk})) \end{aligned}$$

In this alternative formulation, in addition to the set of candidate caches, we treat each operator $\bowtie_{i_j}$ as a cache of zero length with *cost*$(\bowtie_{i_j}) = 0$ and *proc*$(\bowtie_{i_j}) = d_{ij}c_{ij}$. We can state the following theorem in terms of this formulation. The proof and algorithms are given in Reference [89].

Theorem 5.5.3 *There exists a greedy polynomial-time algorithm for finding the nonoverlapping subset X minimizing $\sum_{C \in X} \text{proc}(C) + \text{cost}(C)$ that gives an $O(\log n)$ approximation, where n is the number of joining windows. Furthermore, there exists a randomized polynomial-time algorithm that is also an $O(\log n)$ approximation.* □

Our offline cache selection algorithm is the optimal recursive algorithm if there are no shared candidate caches, otherwise it is the greedy approximation algorithm from Theorem 5.5.3. While the total number of candidate caches $m = O(n^2)$, our experiments indicate that the overhead of exhaustively searching over the 2^m possible combinations of the candidate caches is typically negligible for $n \leq 6$, even in an adaptive setting.

5.5.5 Adaptive Cache Selection Algorithm

For presentation, we first describe a simplified version of our adaptive algorithm. We then identify the inefficiencies in this simplified version and describe how we address them. Recall that the goal of the adaptive algorithm is to ensure that the optimal nonoverlapping subset of candidate caches is being used as conditions change. A candidate cache C is in one of three states at any point in time:

- **Used**: C is being used in join processing as described in Section 5.4.
- **Profiled**: Although C is not being used in join processing, we want to estimate *benefit*(C) and *cost*(C) if it were it to be used. The estimation is performed as described in Section 5.5.3.
- **Unused**: C is neither being profiled nor chosen for use by our adaptive algorithm.

Figure 5.9 illustrates our complete adaptive cache selection algorithm in terms of how it manipulates the states of candidate caches. We begin by describing the simplified version of this algorithm:

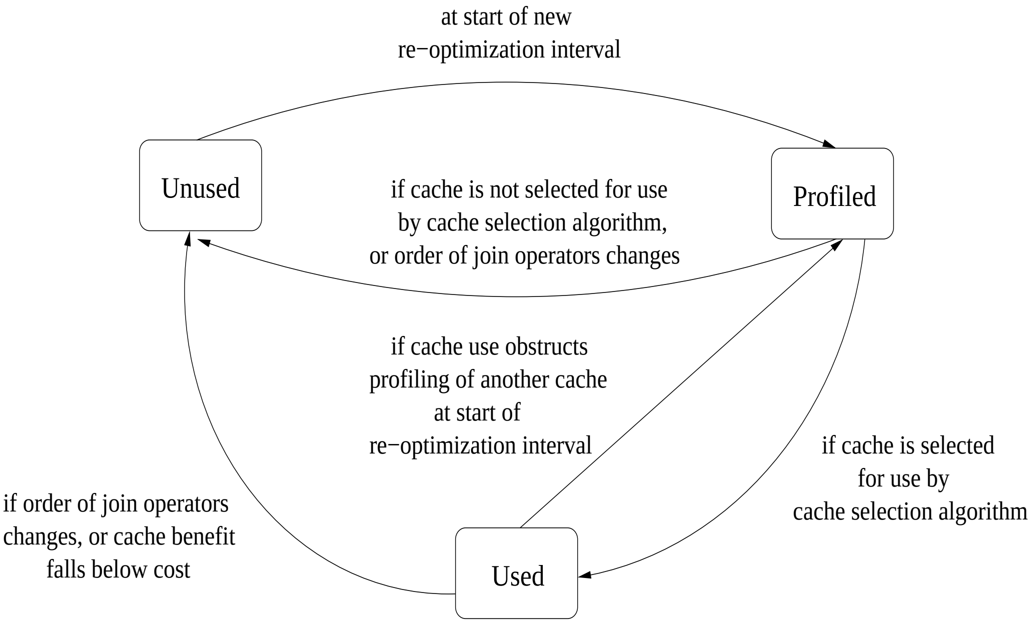

at start of new
re−optimization interval
Unused
Profiled
if cache is not selected for use
by cache selection algorithm,
or order of join operators changes
if cache use obstructs
profiling of another cache
at start of
re−optimization interval
if cache is selected
for use by
cache selection algorithm
if order of join operators
changes, or cache benefit
falls below cost
Used

Figure 5.9: State transition diagram for candidate caches

1. All candidate caches start out in the profiled state.

2. We proceed with regular join processing until each profiled cache has collected at least W observations for each estimated statistic. Recall from Section 5.5.3 that our online estimate of any statistic is the average over the W most recent observations.

3. We run our offline cache selection algorithm to choose the optimal set of caches to use among the profiled caches. The chosen caches are moved to the used state and the rest to the unused state. No new data updates are processed during this step. Each used cache is empty initially and populated incrementally as join processing continues.

4. After a *re-optimization interval* I, we move all candidate caches back to the profiled state and repeat Steps 2 and 3. The interval I can be specified in terms of time or number of tuples processed, and is typically much larger than the interval required to collect W observations for each estimated statistic.

5. If the ordering of join operators in a pipeline is changed at any point by the join ordering algorithm, we remove all caches used in that pipeline, recompute its candidate

caches, and start each new candidate cache in the profiled state. These caches will be considered for placement in the next re-optimization step.

Next we point out the inefficiencies in the simplified version and how we address them.

a. The simplified algorithm does not react to changes within the re-optimization interval I. This problem is fundamentally hard to solve because faster adaptivity requires higher run-time profiling overhead [18, 30]. A complete solution to this problem falls in the realm of *adapting adaptivity* [30], which is beyond the scope of this thesis. Our current approach is to react immediately to changes that make a used cache inefficient, but react gradually (at the next re-optimization step) to changes that make an unused cache useful. We monitor *benefit*(C) $-$ *cost*(C) continuously for all used caches, which can be done with little overhead, and move C immediately to the unused state if the difference becomes negative.

b. During re-optimization, the simplified algorithm always moves a used cache C to the profiled state to estimate its benefit and cost. In reality, we need move C only if it has an unused subset cache C' because C' cannot be profiled if C is being used. Specifically, if C is in the used state, then we cannot access the full probe stream for C', which is required to estimate *miss_prob*(C').

c. Even if statistics are stable, the simplified algorithm periodically invokes the offline algorithm. We reduce this overhead by invoking the offline algorithm only when the benefit or cost of at least one used or profiled cache has changed beyond a certain percentage. Our experiments indicate that a value of $p = 20\%$ is very effective at reducing run-time overhead without affecting adaptivity significantly.

5.5.6 Implementation of Adaptive Caching in STREAM

The full A-Caching algorithm has been implemented in the STREAM prototype. The basic approach used to implement this algorithm is similar to that used to implement A-Greedy and its variants for pipelined filters. Recall from Section 4.7 in Chapter 4 that we incorporated A-Greedy and its variants into STREAM's `select` operator which processes pipelined filters. In a similar fashion, we incorporated both A-Greedy and A-Caching into STREAM's `mjoin` operator. A-Greedy is used to order the join operators adaptively in the `mjoin` pipelines, while A-Caching is used to place caches adaptively in these pipelines.

Like the `select` operator, the `mjoin` operator is implemented as a combination of three interacting components: an Executor, a Profiler, and a Re-optimizer. These components of `mjoin` implement the functionality of the corresponding components of both A-Greedy and A-Caching. A-Greedy was incorporated into `mjoin`'s components in the same fashion as done for the `select` operator, which is illustrated in Figure 4.10 in Chapter 4. This implementation was possible because our extension of A-Greedy for windowed stream joins maps join operators to filters; recall Section 5.2.

In the rest of this section we describe how A-Caching was incorporated into the `mjoin` operator, as illustrated in Figure 5.10. `mjoin`'s Profiler collects estimates of the statistics of profiled and used caches at the beginning of each re-optimization interval. As explained in Section 5.5.3, these statistics for a cache C_{ijk} include the corresponding d_{ij}, c_{ij}, *probe_cost*(C_{ijk}), *update_cost*(C_{ijk}), and *miss_prob*(C_{ijk}). After W observations are collected for all of these statistics, the benefit and cost of each candidate cache is computed.

Once the benefit and cost of each candidate cache is computed at the beginning of a re-optimization interval, `mjoin`'s Re-optimizer first checks whether any of these values have changed from their previous value by more than a certain percentage. (Recall from Section 5.5.5 that our default value for this threshold is 20%.) If such a change has happpenned, then the Re-optimizer invokes the offline cache selection algorithm as a subroutine to compute the set of caches to use during the current re-optimization interval. `mjoin`'s Executor then starts using these caches in the join pipelines. The Profiler continues to track benefits and costs of used caches. If the benefit of a used cache falls below its cost, then the Executor stops using this cache.

5.6 Adaptive Memory Allocation to Caches

Since the memory in a DSMS must be partitioned among all active continuous queries [29, 88], it may be that we do not have sufficient memory to store all caches selected by our cache selection algorithm. Continuing with our modular approach, we first select caches assuming infinite memory, then allocate pages of memory dynamically to the chosen caches

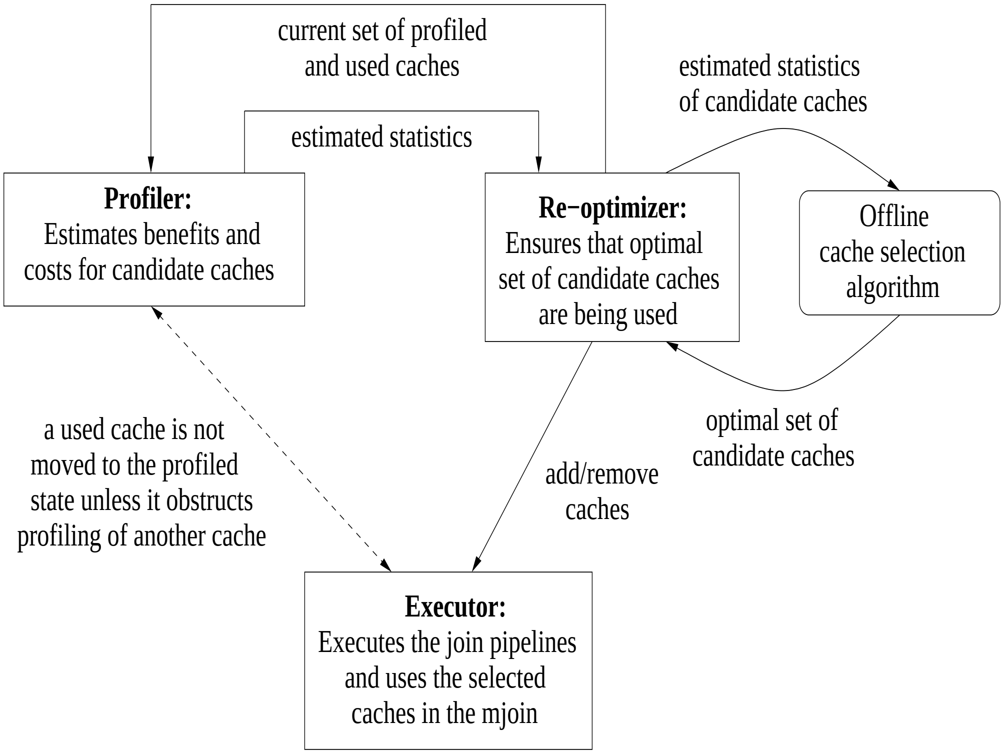

current set of profiled and used caches
estimated statistics
estimated statistics of candidate caches
Profiler: Estimates benefits and costs for candidate caches
Re−optimizer: Ensures that optimal set of candidate caches are being used
Offline cache selection algorithm
optimal set of candidate caches
a used cache is not moved to the profiled state unless it obstructs profiling of another cache
add/remove caches
Executor: Executes the join pipelines and uses the selected caches in the mjoin

Figure 5.10: Architecture of the `mjoin` operator that uses A-Caching for cache selection

so that we can adapt to changes in the amount of memory available. We use a greedy allocation scheme based on the *priority* of a cache C, which is defined as the ratio of *benefit*(C) − *cost*(C) to the expected memory requirement of C. Intuitively, the priority of a cache is its net benefit per unit memory used. The cache memory requirement is the product of the expected number of entries in the cache and the expected size of each cache entry, both of which are estimated before C is used (Section 5.5.3).

Once the offline cache selection algorithm chooses the set of caches to use at the beginning of each re-optimization interval, we compute the priorities of the chosen caches. An `mjoin` operator in STREAM is initially given a fixed-size pool of memory which the operator then allocates dynamically to the chosen caches based on their priorities. If a cache C needs an additional page of memory, then the `mjoin` checks whether it can get an unallocated page from the memory pool or deallocate a page from a cache with lower priority than C. If so, the obtained page is allocated to C. Otherwise, C continues operating with the current amount of memory allocated to it.

5.7 Globally-Consistent Caches

The biggest simplification of A-Caching as described so far is that it restricts candidate caches to those that satisfy the prefix invariant. We address this drawback by relaxing the consistency invariant to create a larger space of candidate caches—called *globally-consistent caches*—to choose from. The following example illustrates the basic idea of globally-consistent caches.

Example 5.7.1 Let the pipelines for a six-way join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_6$ be as given in Example 5.5.1, except suppose ΔR_6's pipeline is now R_2, R_3, R_4, R_5, R_1. Suppose we want to cache $R_2 \bowtie R_3$ in ΔR_6's pipeline. R_2, R_3 does not satisfy the prefix invariant because updates to $R_2 \bowtie R_3$ when R_2 changes are not computed as part of regular join processing. However, R_1, R_2, R_3 satisfies the prefix invariant. Therefore, if we were to cache $(R_2 \bowtie R_3) \ltimes R_1$ instead of $R_2 \bowtie R_3$, then all updates to this cache will be computed as part of regular join processing. Informally, $(R_2 \bowtie R_3) \ltimes R_1$ contains the subset of tuples in the $R_2 \bowtie R_3$ join subresult which have a joining tuple in R_1. □

Globally-consistent caches cache joins of the form $X \ltimes Y$, where X and Y are joins of disjoint subsets of $R_1, \ldots, R_n$, and $X \cup Y$ satisfies the prefix invariant. These caches satisfy the global-consistency invariant, which generalizes the consistency invariant.

Definition 5.7.1 *(***Global-Consistency Invariant***) Cache C_{ijk} satisfies the global-consistency invariant if for all $(u, v) \in C_{ijk}$, $(\sigma_{K_{ijk}=u}(R_{i_j} \bowtie \cdots \bowtie R_{i_k})) \ltimes (R_i \bowtie R_{i_1} \bowtie \cdots \bowtie R_{i_{j-1}} \bowtie R_{i_{k+1}} \bowtie \cdots \bowtie R_{i_{n-1}}) \subseteq v \subseteq \sigma_{K_{ijk}=u}(R_{i_j} \bowtie \cdots \bowtie R_{i_k})$.* □

Globally-consistent caches $X \ltimes Y$ actually capture a spectrum. Our original caches are at one end of this spectrum, where no windows are in Y. At the other end are caches where $X \cup Y = \{R_1, \ldots, R_n\}$, which can be used irrespective of the join ordering since the prefix invariant always holds for $R_1, \ldots, R_n$. In other words, we can always use a globally-consistent cache C_{ijk} based on $X \ltimes Y$ where $X = R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ and Y is the join of all windows not in X.

The use, maintenance, and online estimation algorithms for globally-consistent caches are similar to those outlined in Section 5.4. However, with globally-consistent caches, the offline cache selection problem becomes as hard as the NP-Hard *independent set* problem which cannot be approximated efficiently [67]. Without a good approximation algorithm, we must resort to exhaustive search, with pruning heuristics for efficiency. Our approach, described next, is to fix a maximum number m of candidate caches and to use exhaustive search over the 2^m possible combinations.

Let p be the number of caches that satisfy the prefix invariant. If $p \geq m$, then we ignore globally-consistent caches and use the cache selection algorithm from Section 5.5.4. If $p < m$, the m caches we consider are:

1. the p caches that satisfy the prefix invariant, and
2. $m - p$ additional caches: We start with up to n globally-consistent caches $X \ltimes Y$ where X is all but one window. Then, if we have not used up our quota, we add globally-consistent caches $X \ltimes Y$ where X is all but two windows, and so on. When the quota is reached within one of the iterations, we select from that group of caches arbitrarily.

To incorporate globally-consistent caches into the `mjoin` operator in the STREAM prototype, we require two extensions to the implementation technique described in Section 5.5.6. First, the candidate caches now include all caches that satisfy the global-consistency invariant, which subsumes the set of caches that satisfy the prefix invariant. Second, the offline cache selection algorithm now is the algorithm that we described above.

5.8 Experimental Evaluation

In Section 5.5.6 we gave the details of how A-Caching was implemented in the STREAM prototype DSMS. We now present experimental results based on this implementation.

5.8.1 Experimental Setup

The algorithms we described in this chapter consider the adaptive processing of windowed stream join queries. Our basic experimental setup consists of two such queries:

- Our first query, denoted $R \bowtie S \bowtie T$, is a three-way windowed join of synthetic streams $R'(A)$, $S'(A, B)$, and $T'(B)$. This query is represented in CQL as:

Select $*$
From R' [W_1], S' [W_2], T' [W_3]
Where $R'.A = S'.A$ and $S'.B = T'.B$

Here, W_1, W_2, and W_3 represent tuple-based sliding windows over streams R', S', and T' respectively. The sizes of these windows are set appropriately in our experiments to get desired join selectivities.

- Our second query, denoted $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$, is an n-way windowed join of synthetic streams S_1–S_n. This query is represented in CQL as:

  Select $*$
  From S_1 [W_1], S_2 [W_2], $\cdots$, S_n [W_n]
  Where $S_1.A = S_2.A$ and $S_2.A = S_3.A$ and $\cdots$ and $S_{n-1}.A = S_n.A$

  W_1–W_n represent tuple-based sliding windows over the streams S_1–S_n respectively. Similar to our first query, the sizes of these windows are set appropriately in our experiments to get desired join selectivities.

In our experiments, these queries are processed by the STREAM system using query plans like the one shown in Figure 2.2 in Chapter 2. The joins in these plans are processed by MJoins with or without caching (the `mjoin` operator in STREAM), or by XJoins (the `binary-join` operator in STREAM).

We used a synthetic data generator to generate input streams with specified data characteristics and arrival rates for executing our queries. In our experiments we report the maximum load the system can handle with the chosen query plan, in terms of the number of tuples processed per second, for each of the algorithms that we analyze. We have verified in each case that the bottleneck operators in the plan under peak load are the join operators. In our experiments, the join operators usually account for more than 95% of the total execution time among all plan operators.

Stream arrival rates are uniform by default. All input tuples are 32 bytes long. Default values of the re-optimization interval I (Section 5.5.5) and the window size W (Section 5.5.3) are 2 seconds and 10 respectively. All joins use hash indexes by default to probe the window synopses. All experiments were done on a 700 MHz Linux machine with 1024 KB processor cache and 2 GB memory. 95% confidence intervals are shown for measured values in the graphs in this chapter.

5.8.2 Summary of Experimental Results

The goals of our experimental study were to understand (i) how caches affect performance in MJoin pipelines, and (ii) the behavior of A-Caching with respect to the convergence properties, run-time overhead, and speed of adaptivity metrics. We summarize the results:

1. For a wide range of values of input parameters, e.g., join selectivity and stream arrival rate, caches improve MJoin performance significantly; up to 90% in some cases.

2. We studied the performance of four different algorithms for windowed stream joins under stable conditions: (i) MJoins (M), (ii) XJoins (X), (iii) A-Caching using prefix-invariant caches (P) (Section 5.5), and (iv) A-Caching using globally-consistent caches (G) (Section 5.7). We found that X, P, and G almost always outperform M. Furthermore, there are scenarios where X outperforms P significantly, and others where G outperforms X significantly.

3. When input conditions change, e.g., the rate of arrival of a stream increases abruptly, A-Caching adapts to the new conditions without any significant pause in throughput.

4. As expected, the performance of MJoins with A-Caching changes in a relatively smooth fashion when the memory for storing join subresults is increased or decreased. On the other hand, regular MJoins cannot make use of any such memory, while XJoins cannot execute unless some minimum amount of memory is available.

5.8.3 Performance of Caching

We begin by considering the performance of caching in MJoin pipelines with respect to a variety of input parameters such as join selectivity and stream rate. For all experiments in this section, we show an *absolute* graph and a *relative* graph. The absolute graph shows the average tuple-processing rates (average number of tuples processed per second) of the best plan with caches and the best MJoin plan without caches. The relative graph shows the ratio

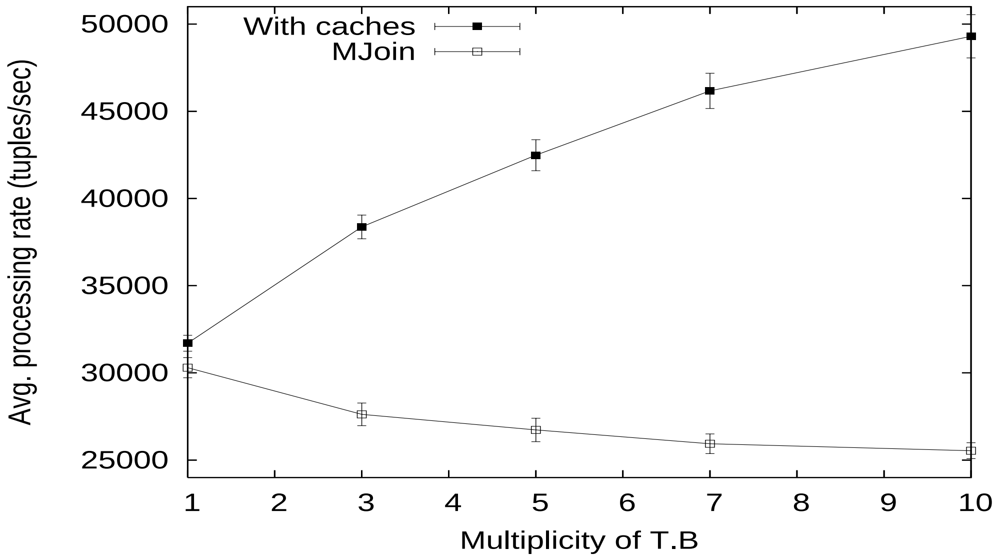


Figure 5.11: Varying cache hit probability (absolute graph)

of the average tuple-processing rate of MJoin to that of the plan with caches. For example, a y value of 0.6 on the relative graph means that if the plan with caches processes Y tuples per second, then the underlying MJoin processes $0.6Y$ tuples per second. Note that these numbers capture all types of overheads as well, including profiling and re-optimization overhead. Most experiments in this section use the $R \bowtie S \bowtie T$ query. The join attributes $R.A, S.A, S.B$, and $T.B$ draw values from the same domain in the same order. The multiplicity of these values is 1 in R and S and a variable r in T (default $r = 5$), and the rate of ΔT correspondingly is r times that of ΔR and ΔS. Globally-consistent caches were not used in the experiments reported here, but their performance is similar. Globally-consistent caches are considered later in this section.

Figures 5.11 and 5.12 compare plans with caching against MJoins when we vary the cache hit probability. These plans are similar to Figures 5.7 (plan with caches) and 5.2(a) (MJoin). There is only one candidate cache, which we force to be chosen. This cache is probed using $T.B$, so by varying the multiplicity of $T.B$ we vary the cache hit probability. As the hit probability increases, using the cache becomes significantly better than the best MJoin. Interestingly, using the cache is better even when all multiplicities are one. Because we are using sliding windows, every inserted tuple in the input stream subsequently appears

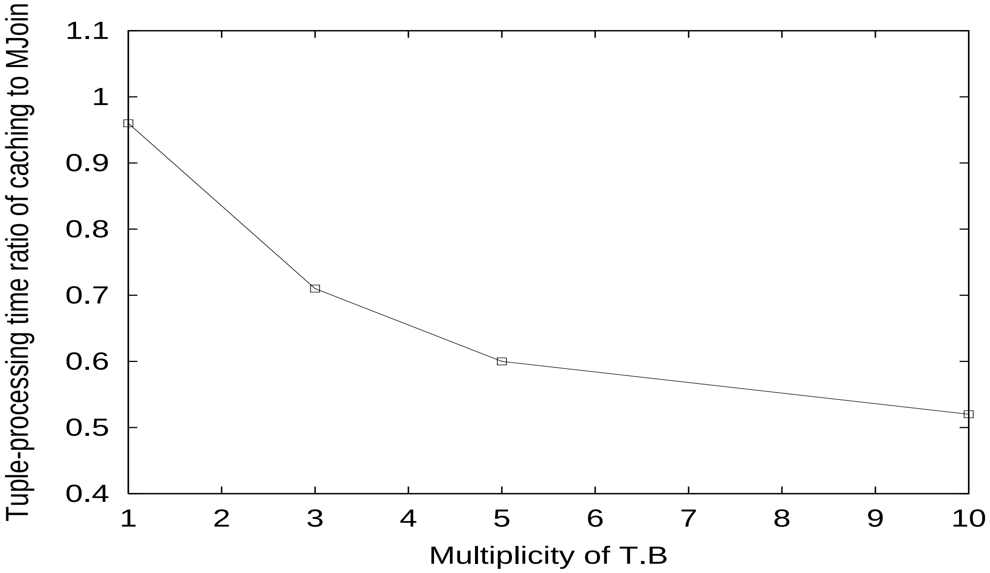


Figure 5.12: Varying cache hit probability (relative graph)

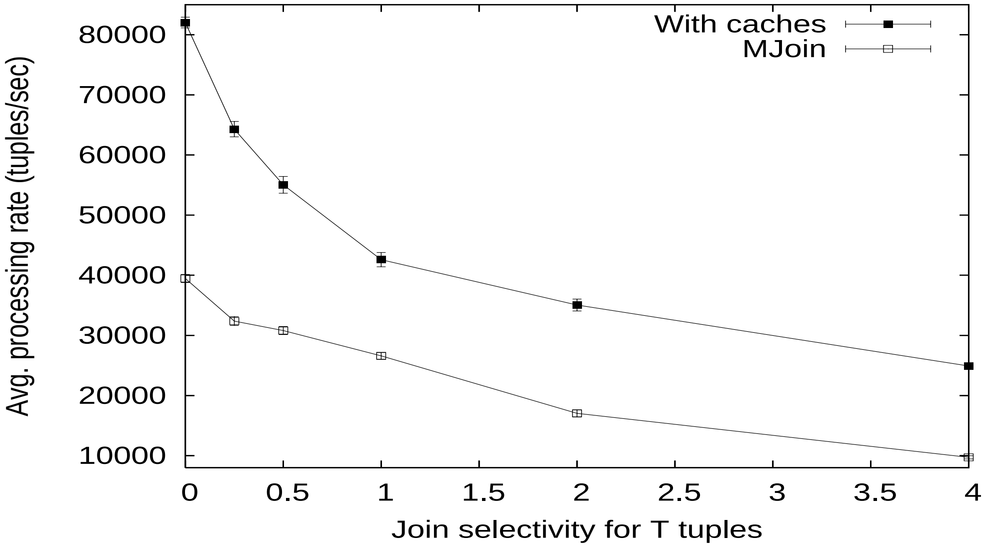


Figure 5.13: Varying join selectivity (absolute graph)

as a deleted tuple. Because of this property, we have at least one opportunity for a cache hit per distinct tuple.

Figures 5.13 and 5.14 vary join selectivity (or multiplicity) in ΔT's pipeline, that is, the number of $R \bowtie S$ tuples joining with ΔT tuples. Notice that caching improves performance over the entire join selectivity range. The relative improvement is least when

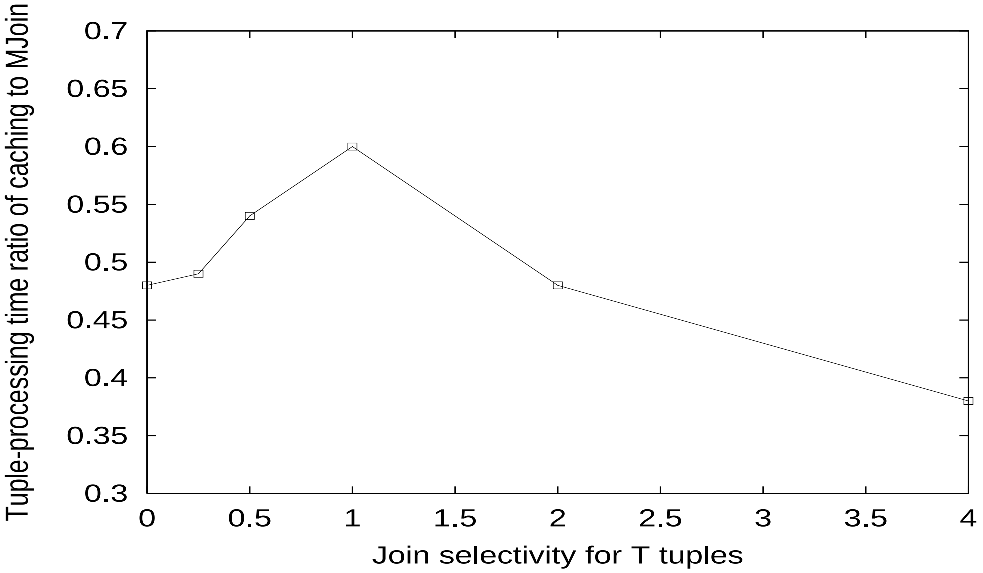


Figure 5.14: Varying join selectivity (relative graph)

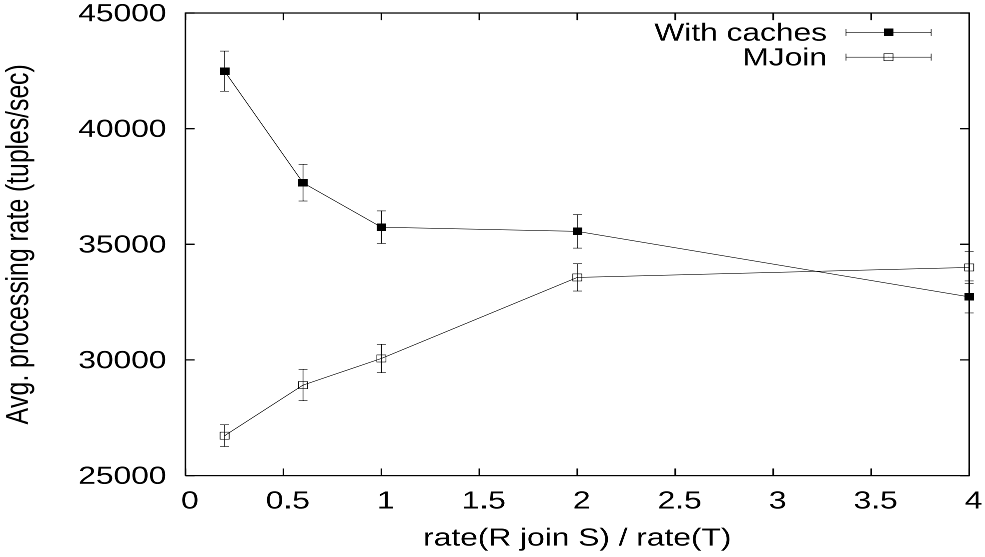


Figure 5.15: Varying update to probe ratio (absolute graph)

selectivities are close to 1, which is explained by the two opposing effects at work: As join selectivity increases, each cache hit saves more work, but each cache miss requires more work to update the cache.

Figures 5.15 and 5.16 vary the ratio of the cache update rate to the cache probe rate. Once again, we force the one candidate cache to be used. As expected, the performance

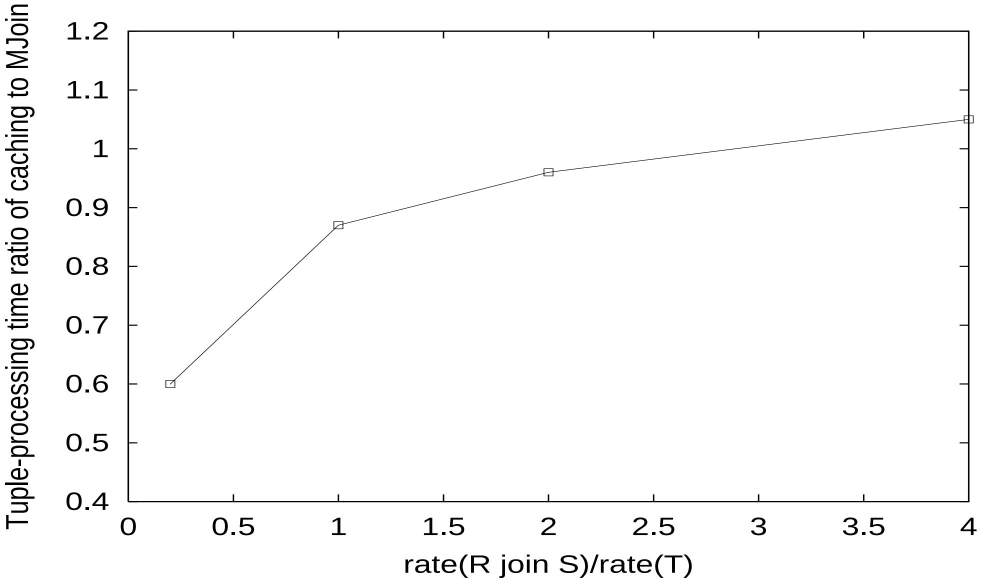


Figure 5.16: Varying update to probe ratio (relative graph)

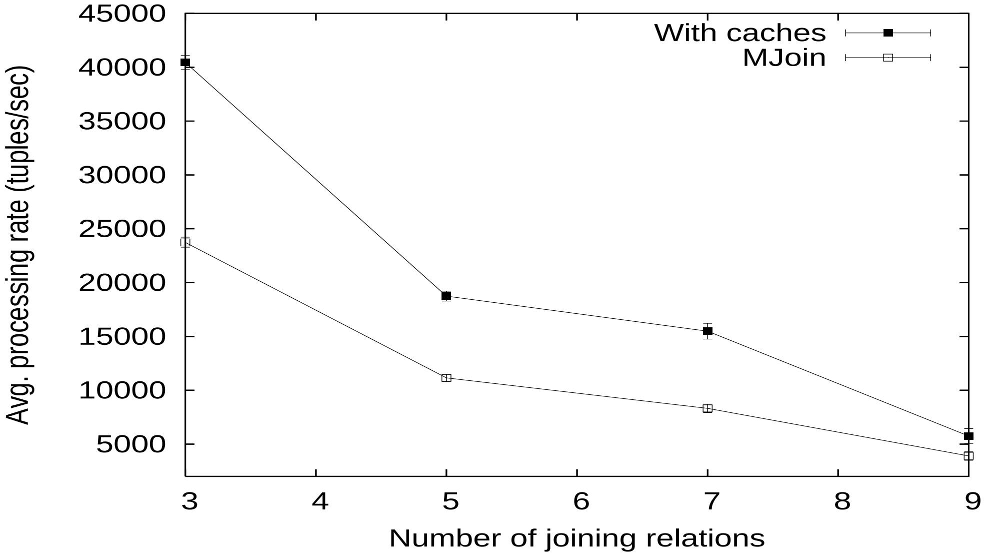


Figure 5.17: Varying number of joins (absolute graph)

of caching deteriorates with a higher update rate. However, the update cost of the cache is low with respect to the work saved per cache hit, so using the cache is better even when the update rate is higher than the probe rate.

Figures 5.17 and 5.18 consider the n-way join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$, varying n. The multiplicity of the join attributes is 1 for $\lfloor \frac{n}{2} \rfloor$ of the streams and 5 for the others. Notice

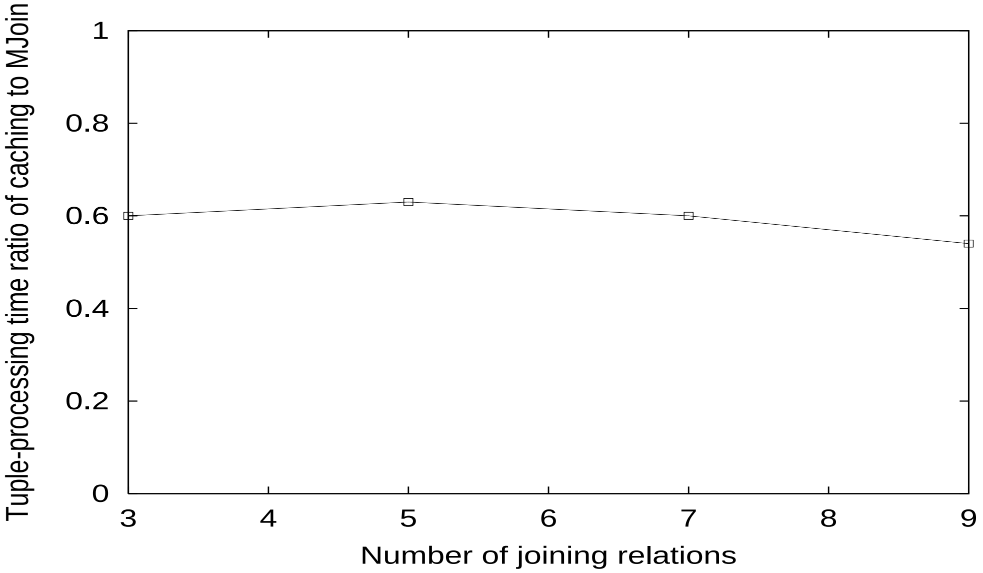


Figure 5.18: Varying number of joins (relative graph)

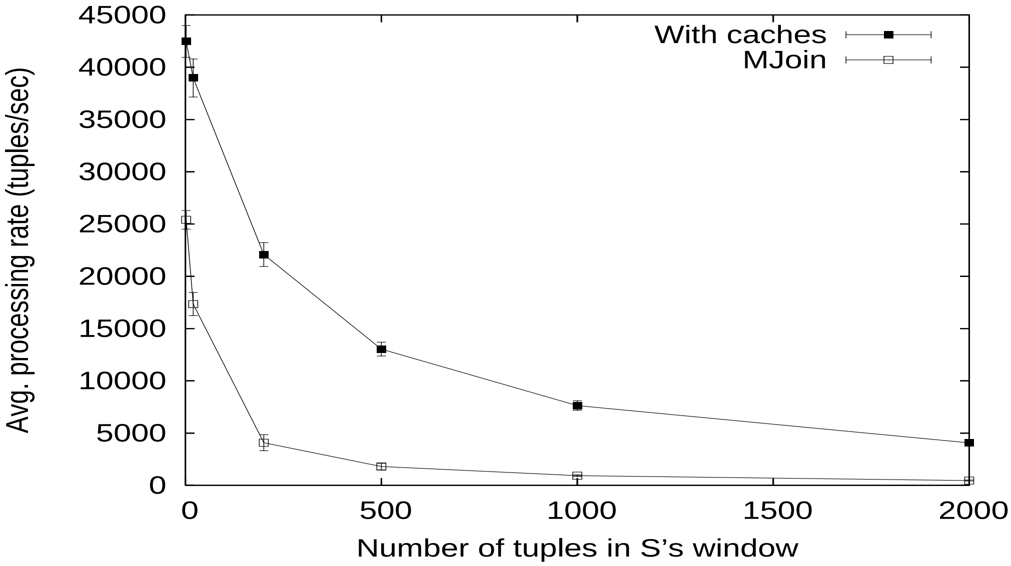


Figure 5.19: Varying join cost (absolute graph)

that the improved join performance using caches is maintained throughout the range. As an example, the 7-way join used 6 caches out of 15 available candidate caches.

In Figures 5.19 and 5.20, we consider varying join costs. Returning to the three-way join query, we drop the hash index on $S.B$ so that the join with S in ΔT's pipeline is forced to use a nested-loop join. The join cost is then proportional to the number of tuples in (the

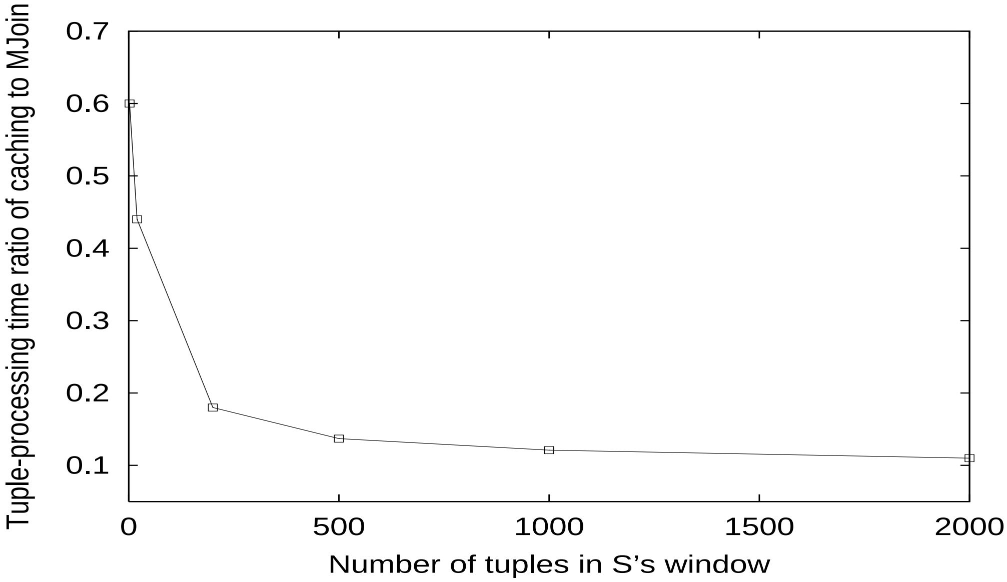


Figure 5.20: Varying join cost (relative graph)

window) S, which is varied in Figures 5.19 and 5.20. As expected, the relative performance of caching improves significantly with increasing join cost.

5.8.4 Performance of Plans for Windowed Stream Joins

We experimented extensively with different input stream characteristics, namely, update rates and join selectivities, to study the performance of the four different types of plans that we consider: (i) MJoins, (ii) XJoins, (iii) Caching-based plans from Section 5.5 satisfying the prefix invariant, and (iv) Caching-based plans from Section 5.7 which may include globally-consistent caches. Figure 5.21 summarizes eight of our experiments to illustrate the trends that we observed, described below.

Our experiments used the 4-way join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_4$. Update rates and join selectivities were varied in our experiments. Figure 5.21 shows plan performance for eight sample points D_1–D_8 from this spectrum of input characteristics. The relative stream arrival rates and pairwise join selectivities at these sample points are shown in Table 5.2. As usual, the numbers in Figure 5.21 represent the maximum input load the system can handle in each case, so they include all types of overheads as well. For each point, we report the

Sample Point	Arrival Rates				Pairwise Join Selectivities					
	R	S	T	U	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$
D_1	10	1	1	1	0.004	0.005	0.005	0.007	0.0045	0.005
D_2	8	1	1	8	0.004	0.005	0.005	0.007	0.0045	0.005
D_3	10	15	1	5	0.003	0.005	0.007	0.0045	0.006	0.008
D_4	1	1	1	1	0.003	0.004	0.0067	0.002	0.0023	0.0027
D_5	4	1	1	4	0.005	0.007	0.005	0.006	0.005	0.002
D_6	1	1	1	1	0.005	0.0033	0.0025	0.0067	0.005	0.0075
D_7	1	1	1	1	0	0	0	0	0	0
D_8	1	1	1	1	0.001	0.001	0.001	0.001	0.001	0.001

Table 5.2: Relative stream arrival rates and pairwise join selectivities for the eight sample points in Figure 5.21. The stream arrival rates are relative to the rate of stream T

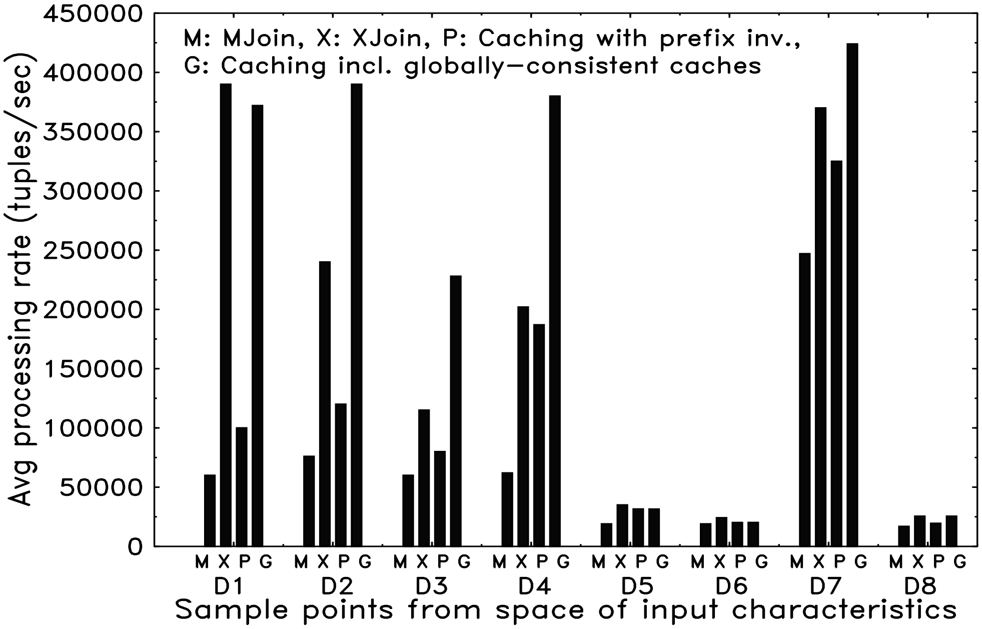


Figure 5.21: Performance of join plans

performance of the best MJoin (M), the best XJoin (X), the best caching-based plan with the prefix invariant (P), and the best caching-based plan with globally-consistent caches (G). M is chosen using the A-Greedy algorithm from Chapter 4, which is the adaptive join ordering algorithm used in caching-based plans as well. X is chosen by exhaustive search. For uniformity, both P and G are chosen by exhaustive search over the set of candidate caches, with $m = 6$ in the algorithm in Section 5.7. The overhead of optimization accounts for a negligible fraction of the total processing time in all cases in Figure 5.21. All plans were given as much memory as they required to maintain join subresults; memory constraints are considered in Section 5.8.5.

Our experiments showed the following trends:

1. X, P, and G almost always outperform M; see D_1–D_8 in Figure 5.21. (The experiments in [42] seem to indicate a similar trend between X and M in Eddies.) This trend is aided by our efficient implementation of join subresult materialization and caching (Section 5.4.1), especially minimizing memory-to-memory copying and using a low-overhead cache management scheme.

2. P usually significantly outperforms M, showing the benefit of adding caches to MJoins; see D_1–D_8.

3. There are cases where X significantly outperforms P; see D_1, D_2, and D_3. The best MJoin ordering for the input characteristics in these experiments was such that the prefix invariant was not satisfied for one or more high-benefit caches. This problem is alleviated when our algorithm considers globally-consistent caches. That is, G is almost always as good as X. When X, P, and G all maintain the same join subresults, X may be slightly better than P and G because a probe into a (fully materialized) join subresult in an XJoin never "misses": If no joining tuples are found, then no joining tuples exist.

4. G can significantly outperform X; see D_2, D_3, D_4, and D_7. Any XJoin for a 4-way join can materialize at most two join subresults, with at most one 3-way join subresult. G removes this restriction by considering plans from the spectrum between MJoins and XJoins. G outperforms X at D_2, D_3, D_4, and D_7 by caching more than one 3-way join subresult.

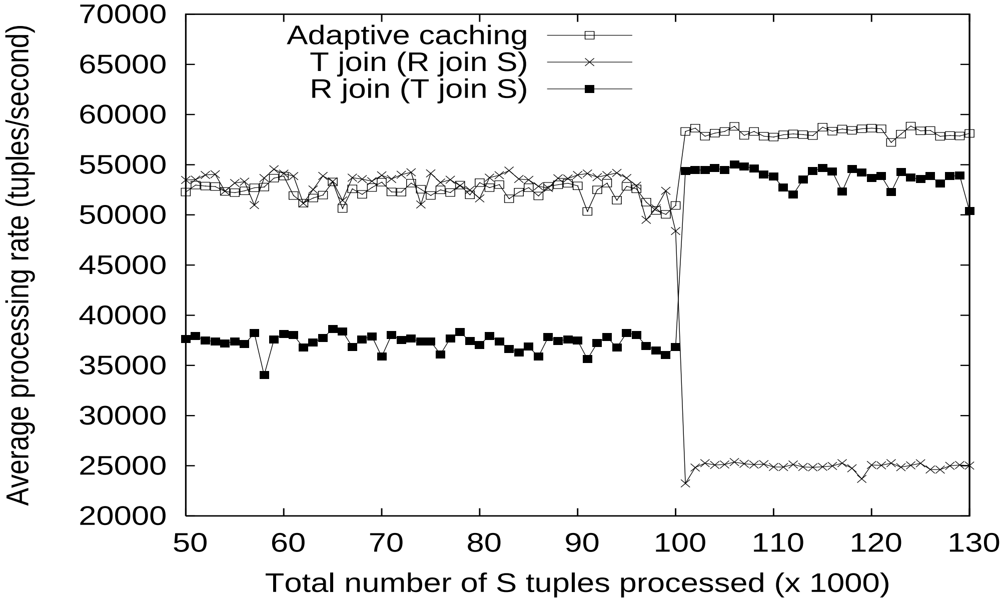


Figure 5.22: Adaptivity to changing stream rate

5.8.5 Adaptivity Experiments

Next we consider adaptivity to changing input characteristics. We use the 3-way join query and consider the case where ΔR is bursty. The rate of ΔR during a burst is 20 times its normal rate. The re-optimization interval I was set to 10,000 tuples.

In Figure 5.22 we consider three plans for the 3-way join query: a static plan, denoted $T \bowtie (R \bowtie S)$, which always uses a single $R \bowtie S$ cache in ΔT's pipeline, another static plan, denoted $R \bowtie (T \bowtie S)$, which always uses a $T \bowtie S$ cache in ΔR's pipeline, and our adaptive cache selection algorithm from Section 5.7. The x-axis in Figure 5.22 shows the progress of time in terms of the number of ΔS tuples that have arrived so far. The y-axis shows the current tuple-processing rate (average number of tuples processed per unit time). The initial stream characteristics correspond to our default experimental setup from Section 5.8.3, so that using the $R \bowtie S$ cache in ΔT's pipeline is optimal as evident from the performance of the static plan $T \bowtie (R \bowtie S)$. We see that our adaptive algorithm converges to this plan. Also note that the performance of our adaptive algorithm is almost equal to that of the static plan, indicating very low run-time overhead for adaptivity. A

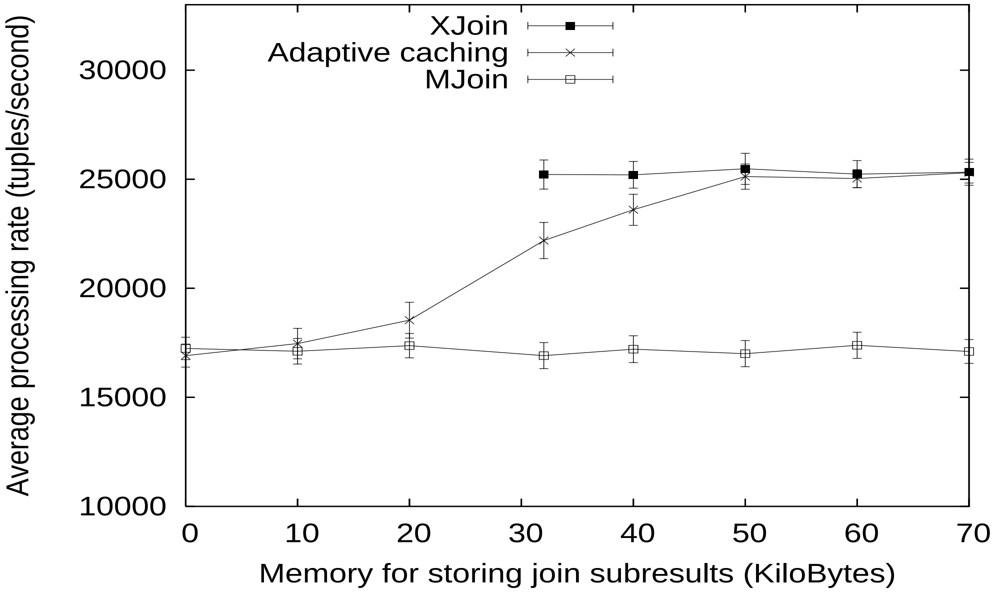


Figure 5.23: Adaptivity to memory availability

burst in ΔR starts when 100,000 ΔS tuples have arrived and continues through the rest of the run. As expected, $T \bowtie (R \bowtie S)$ performs poorly during the burst, while static $R \bowtie (T \bowtie S)$ becomes the high-performer. Our adaptive algorithm converges quickly to the new best plan which uses a globally-consistent $(T \bowtie S) \rhd\!\!< R$ cache in ΔR's pipeline.

Figure 5.23 shows how our caching-based plans adapt smoothly to the amount of memory available for storing join subresults. This experiment used the same setup as in point D_8 in Figure 5.21 (Section 5.8.4), except now we vary on the x-axis the amount of memory available for storing join subresults. MJoins are insensitive to extra memory since they do not store join subresults. For the input that we used, any XJoin should require at least 25.6 KB of memory to store its join subresults, so the region before $x =$ 25.6 KB is infeasible for XJoins. The XJoin curve following this infeasible region should actually be a step function, with the steps corresponding to optimal plans for different memory availability. Since our XJoin optimizer does not take the amount of memory available as a parameter, Figure 5.23 reports only the best XJoin, which requires roughly 32 KB memory to store its join subresults in our system.

5.9 Related Work

Although multiway windowed stream joins have received a great deal of attention recently [18, 56, 114], no previous work has considered adaptive caching in this environment to the best of our knowledge. Reference [114] studies the advantages of MJoins over XJoins. Reference [56] considers lazy window maintenance, [11] considers plan selection under resource constraints, and [121] considers plan switching between XJoins.

Reference [42] shows the performance benefits of XJoins (*STAIRs*) over MJoins (*SteMs* [94]) in Eddies [10], and studies the interaction between join subresults and adaptivity. Reference [42] does not consider caching or the spectrum between MJoins and XJoins.

Caches are used widely in database systems, e.g., to avoid recomputing expensive predicates [65], as join indexes [119], view indexes [97], and view caches [98] to balance update costs and query speedup, and to make views self-maintainable [93]. Previous cache selection algorithms do not address one or more issues relevant in the context of data streams, such as adaptivity, plan switching, cache sharing, and ease of collecting statistics during query execution. Previous work on optimizing incremental view maintenance plans [80, 96, 115] does not consider the spectrum between MJoins and XJoins, and is non-adaptive.

5.10 Conclusion

In this chapter we showed how the A-Greedy algorithm for pipelined filters can handle adaptive ordering of join operators in MJoins for windowed stream joins. However, MJoins may not be the most efficient way of processing windowed stream joins under all scenarios. We proposed the A-Caching algorithm that addresses this problem by placing subresult caches adaptively in MJoin pipelines, improving overall performance and adaptivity over both MJoins and XJoins.

Our experiments indicate the importance of considering the spectrum between stateless MJoins and cache-rich XJoins, and adapting as stream and system conditions change. Caches improve the performance of MJoins significantly for a wide range of values of input parameters such as join selectivity and stream arrival rate; up to 90% as shown in Figure 5.20. A-Caching using globally-consistent caches outperforms both MJoins and

XJoins significantly in many cases, and was never found to be worse than optimal by a large margin; see Figure 5.21. A-Caching enables MJoins to adapt quickly and smoothly to changes in input stream characteristics, e.g., to bursty stream arrival (Figure 5.22), and to changes in system conditions, e.g., to the amount of memory available for storing join subresults (Figure 5.23).

The three-way tradeoff among the convergence properties, run-time overhead, and speed of adaptivity metrics does apply in the case of windowed stream joins, although the effect is not as evident as it was for pipelined filters in Chapter 4. For example, as we go from MJoins without caches to A-Caching using prefix-invariant caches to A-Caching using globally-consistent caches, the quality of plans produced under stable conditions improves significantly. However, the run-time overhead increases considerably, while the speed of adaptivity is little affected.

Chapter 6

Exploiting k-Constraints to Reduce Memory Overhead for Windowed Stream Joins

In the previous chapter we considered query execution plans for multiway windowed stream joins. These plans often have large memory requirements because they need to maintain a significant amount of run-time state. For example, all the plans we considered for windowed stream joins in the previous chapter—MJoins with and without subresult caches, and XJoins—need to store all tuples in the current windows over the streams. Unless the windows are stored and maintained completely at all points in time, these plans cannot guarantee correct join results. If tuples arrive rapidly in the input streams, then these windows can potentially get very large in size, leading to large memory requirements. For example, if a stream of 32-byte packet headers arrives at the rate of 10^5 tuples per second into a 10-minute window, then the average window size would be almost 2 gigabytes.

However, streams in real applications often exhibit certain data and arrival patterns, which we refer to as *constraints*. For example, as we saw in the network monitoring application in Chapter 1, tuples in different streams that match in a join may usually arrive close together in time. The same application also had multiple instances of *many-one join* constraints, where a tuple in a stream joins with at most one tuple in another stream. In this chapter we show how such constraints can be exploited to reduce considerably run-time state in windowed stream join plans, without compromising the accuracy of the join result. We discuss various types of stream-based constraints and show how such constraints can be detected and exploited in a DSMS.

It is unrealistic to expect constraints to be satisfied precisely in a data streams environment, e.g., due to variability in data generation, network load, and scheduling. But streams may frequently come close to satisfying constraints, and we want to capture and take advantage of these situations. Therefore, we consider k*-constraints*, where k is an *adherence parameter* specifying how closely a stream adheres to the constraint. $k = 0$ denotes the strict version of the constraint. (Smaller k's are closer to strict adherence and offer better state reduction.) For example, *strict referential integrity* on a many-one join from stream S_1 to stream S_2 requires that each tuple $s_2 \in S_2$ must arrive before any tuple $s_1 \in S_1$ that joins with s_2. The relaxed k-constraint version only requires s_2 to arrive within k tuple arrivals on S_2 after s_1 arrives. In the join from stream C to stream B in our example network monitoring query from Section 1.3 in Chapter 1, a referential integrity constraint holds with $k = d_{cb} \cdot rate(B)$. Recall that d_{cb} is the latency bound between links C and B, and $rate(B)$ is the arrival rate of stream B.

As we described in Chapter 2, operators in STREAM query plans process input tuples in nondecreasing order of timestamp. Therefore, input streams from external sources are processed first by STREAM's input manager to reorder tuples that are not in timestamp order (recall Section 2.2.5 in Chapter 2). The algorithms for reordering are based on stream properties like the degree of out-of-order arrival within a stream and the timestamp skew in the arrival of different streams. These stream properties are effectively k-constraints. The query processor uses k-constraints to reduce run-time state in plans for windowed stream joins, while the input manager uses similar stream properties to put tuples in timestamp order; a full comparison between the two approaches is given in Section 6.1.4.

Adherence of streams to constraints can change during the lifetime of long-running continuous queries. For example, if $rate(B)$ increases in our network monitoring example, then k will increase for the referential integrity constraint on the join from C to B. To address this challenge, we propose a query processing architecture called k*-Mon*. k-Mon can detect useful k-constraints automatically in streams and exploit these constraints to reduce run-time state in plans for windowed stream joins. Furthermore, k-Mon can adapt to changes in stream constraints while join queries are running. Like the A-Greedy and A-Caching algorithms described in the preceding chapters, k-Mon follows the generic StreaMon framework from Chapter 3 for adaptive query processing. We describe how k-Mon instantiates the Executor, Profiler, and Re-optimizer components of StreaMon. We also report a thorough experimental evaluation of k-Mon, showing dramatic state reduction in windowed stream join plans, while only modest computational overhead is incurred for the constraint monitoring and query execution algorithms.

This chapter proceeds as follows. Section 6.1 illustrates the problem of detecting and exploiting stream constraints to reduce run-time state in query plans. Section 6.2 establishes notation and definitions for this chapter. Section 6.3 describes our basic query execution algorithm. Section 6.4 gives an overview of our implementation of k-Mon in a prototype continuous query processor, while Section 6.5 describes our experimental setup. Sections 6.6–6.8 formalize the three specific k-constraint types we consider, incorporate them into our basic execution algorithm, present monitoring algorithms for them, and include experimental results for each k-constraint type. Section 6.9 measures the computational overhead of detecting and exploiting k-constraints during query processing. Section 6.10 summarizes our complete approach. Section 6.11 provides examples and experiments from the *Linear Road Benchmark* [6] for DSMSs. We discuss work related to k-constraints in Section 6.12, and conclude in Section 6.13.

6.1 Introduction

We begin with an example that illustrates the overall problem we consider in this chapter:

6.1.1 Example

Consider the example network monitoring query from Section 1.3 in Chapter 1. This query tracks the total traffic from a customer network that went through a specific set of links in an ISP's network within the last 10 minutes. The CQL representation of this query is reproduced here for convenience:

Select sum(C.size)

From C [Range 10 minutes], B [Range 10 minutes], O [Range 10 minutes]
Where C.`pid` = B.`pid` and B.`pid` = O.`pid`

This continuous query is a three-way windowed equi-join of streams C, B, and O on attribute `pid`. A 10-minute tuple-based sliding window is specified on each stream. The join result is aggregated continuously to compute the total common traffic across links C, B, and O in the last 10 minutes.

An MJoin-based plan to execute this query is shown in Figure 6.1. The run-time state maintained by this plan consists of the three synopses corresponding to the windows over the streams. In addition to the tuples in the windows, the synopses also contain indexes to enable the join operators to probe for joining tuples efficiently. Therefore, the total memory requirement for this plan is roughly the sum of the sizes of the three windows, plus some extra memory for the (hash) indexes on the windows. Assuming 10-byte tuples and tuple rates of 10^3, 10^4, and 10^3 per second in streams C, B, and O respectively, the total memory requirement for the window synopses in Figure 6.1 is at least 72 MB, which is relatively high for a single query.

Recall from Section 1.3 in Chapter 1 that the streams in this application exhibit some interesting properties:

- The packets we are monitoring flow through link C to link B to link O. Thus, a tuple corresponding to a specific `pid` appears in stream C first, then a joining tuple may appear in stream B, and lastly in stream O.

- If the latency of the network between links C and B and between links B and O is bounded by d_{cb} and d_{bo} respectively, then a packet that flows through links C, B, and O will appear in stream B no later than d_{cb} time units after it appears in stream C, and in stream O no later than d_{bo} time units after it appears in stream B.

Both of these properties can be exploited to reduce the number of tuples that need to be stored in the windows at any point in time:

- When a tuple t arrives in stream B and no joining tuple exists in the window on C, t can be discarded immediately because a tuple in C joining with t should have arrived before t.

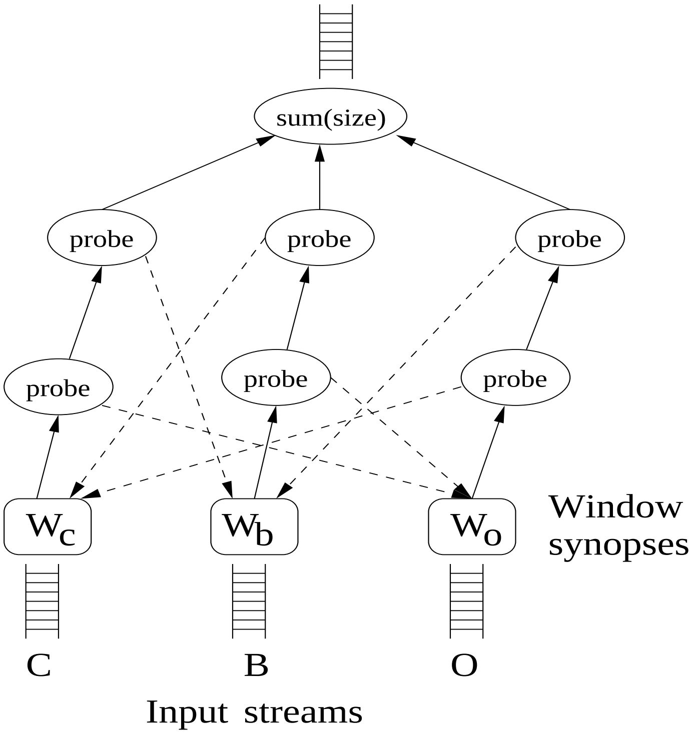


Figure 6.1: Query execution plan for example windowed equi-join query

- The query processor can discard a tuple t with timestamp ts from the window on C when a tuple with timestamp $> ts + d_{cb}$ arrives on B and no tuple joining with t has arrived so far on B.

Similar memory reductions can be applied to the windows on B and O. By reducing the number of tuples that need to be stored in the windows at any point in time, the overall memory requirement for the window synopses in Figure 6.1 can be reduced significantly.

To appreciate the scale of memory reduction possible by exploiting the stream properties, let us assume that approximately 10% of the packets on link C go on to link B, and independently 10% of the packets on B go on to O. Then, the total memory requirement for the window synopses in Figure 6.1 can be reduced to around 0.18 MB if the input stream properties are detected and exploited; a reduction by two orders of magnitude.

6.1.2 Challenges in Exploiting Stream Properties

The example in the previous section illustrates how the memory requirement for a query plan can be reduced by orders of magnitude if input stream properties are exploited during query processing. Three challenges arise in this context:

1. The stream properties used in our example seem application-specific. Is there a set of properties that are useful across a wide variety of applications and continuous queries?

2. The query processor has little control over the data and arrival patterns of streams from external sources [14, 55].

3. Stream properties can change during the lifetime of a long-running continuous query [52, 83, 103]. For example, the latency bound d_{cb} may change based on congestion in the network.

To address the first challenge we studied several data stream applications and identified a set of basic *constraints* that individually or in combination capture the majority of properties useful for memory reduction in continuous queries [102]. The basic constraints we identified are *many-one joins*, *stream-based referential integrity*, *ordering*, and *clustering*.

To address the second challenge we introduce the notion of *k-constraints*: $k \geq 0$ is an *adherence parameter* capturing the degree to which a stream or joining pair of streams adheres to the strict interpretation of the constraint. The constraint holds with its strict interpretation when $k = 0$. The concept of k-constraints is very important in the stream context since it is unreasonable to expect streams to satisfy stringent constraints at all times, e.g., due to variability in data generation, network load, and scheduling. But streams may frequently come close to satisfying constraints, and k-constraints enable us to capture and take advantage of these situations.

To address the third challenge we developed an architecture where the query processor continuously monitors input streams to detect potentially useful k-constraints. This approach adapts to changes in stream constraints and enables the query processor to give the best memory reduction based on the current set of constraints. It frees users and system administrators from having to keep track of stream constraints, thereby improving system manageability. As we will see, only modest computational overhead is incurred for constraint monitoring and for constraint-aware query processing.

6.1.3 Overview of Stream Constraints

Next we informally describe in a bit more detail the constraint types and adherence parameters we consider. We continue considering the network monitoring application from Section 6.1.1 and give detailed examples of some k-constraints that arise in this setting. Section 6.11 provides more examples of k-constraints from the Linear Road Benchmark, a sensor-based application developed as a benchmark for data stream systems [4, 6].

Ordered-Arrival Constraints

Streams often arrive roughly in order according to one of their attributes, such as a time-related or a counter attribute [103]. We define an *ordered-arrival constraint* on a stream attribute $S.A$ with adherence parameter k, denoted OA(k), as follows: For any tuple s in stream S, all S tuples that arrive at least $k + 1$ tuples after s have an A value $\geq s.A$. That is, any two tuples that arrive out of order are within k tuples of each other. Note that $k = 0$ captures a strictly nondecreasing attribute.

Example 6.1.1 Some routers on the network are usually configured to report traffic statistics for recently expired flows [90]. (Here a flow denotes the collection of packets sent in one TCP connection from a source to a destination.) A typical setting expires a flow and outputs a flow record when a “finish” packet arrives in the flow (voluntary closure) or when no packets arrive in the flow for a timeout interval of 15 seconds (forced closure). Consider the `stop_time` attribute of the resulting flow record stream, which denotes the arrival time of the last packet in the flow. The values of this attribute are nondecreasing over voluntarily closed flows, but forced closures create a scrambling of `stop_time` values in the stream as a whole. If the router reports at most n flows every second to limit the bandwidth consumed by monitoring applications, any two `stop_time` values that are out of order in the flow record stream will be at most $15n$ tuples apart, so the flow record stream satisfies OA($15n$) over the `stop_time` attribute. □

Clustered-Arrival Constraints

Even when stream tuples are not ordered, they may be roughly clustered on an attribute. For example, if we consider the union of streams C, B, and O in our example network monitoring query, then all tuples for a particular `pid` will be relatively close together in the stream. In the Linear Road Benchmark, the incoming sensor stream is approximately clustered on a combination of *car* and *segment* identifiers [6]; see Section 6.11.

We define a *clustered-arrival constraint* on stream attribute $S.A$ with adherence parameter k, denoted CA(k), as follows: If two tuples in stream S have the same value v for A, then at most k tuples with non-v values for A occur on S between them.

Example 6.1.2 An interesting continuous query in network monitoring, termed *trajectory sampling*, maintains a summary of routes taken by packets through the network [46]. To support this query, devices on links across the network sample packets continuously with the property that a packet chosen by any one device will be chosen by all other devices that observe the packet [46]. Consider the resulting merged stream of tuples with schema (`pkt_id`, `link_id`) sent by these devices. `pkt_id` is a unique identifier for a packet and `link_id` represents a link where the packet was observed [46]. If there are m devices sampling at the rate of s packets per second, and d represents the maximum delay of packets through the network, then any two tuples in the stream with the same value of `pkt_id` are separated by no more than $m \times s \times d$ tuples with a different value of `pkt_id`, creating a CA($m \times s \times d$) constraint on the `pkt_id` attribute. □

Join Constraints

In our example network monitoring query from Section 6.1.1, the join between each pair of streams is a *one-one join*. One-one joins are a special case of *many-one joins*, which are very common in practice [102]. (As we will see in Section 6.11, most joins in the Linear Road Benchmark queries are many-one joins.) In this chapter we will assume that all joins in our queries are many-one joins. Our overall approach, theorems, and algorithms are fairly independent of this assumption, but the benefit of our algorithms is reduced in the presence of many-many joins.

An additional join constraint that we saw in the network monitoring application bounded the delay between the arrival of a tuple on one stream and the arrival of its joining tuple on the other stream. We define a *referential integrity over data streams* constraint on a many-one join from stream S_1 to stream S_2 with adherence parameter k, denoted RIDS(k), as follows: For a tuple $s_1 \in S_1$ and its unique joining tuple $s_2 \in S_2$, s_2 will arrive within k tuple arrivals on S_2 after s_1 arrives. For the special case of $k = 0$ for this constraint, termed *strict referential integrity*, s_2 will always arrive before s_1.

Example 6.1.3 If the amount of traffic destined to a *peer* ISP on a network link L exceeds a certain threshold, a network analyst might want to drill down into a sample of this traffic. A continuous query for this purpose joins two streams: S_1(`pkt_hdr`, `peer_id`, `interval`) and S_2(`peer_id`, `num_bytes`, `interval`). S_1 is a stream of packets sampled from L containing the packet header (`pkt_hdr`), the identifier of the destination peer ISP (`peer_id`), and the packet capture time on L at the granularity of 5-minute intervals (`interval`). S_2 is a stream of measurements containing the total observed traffic (`num_bytes`) on L destined to peer ISP (`peer_id`) for each 5-minute interval (`interval`). Clearly each packet on S_1 is destined to a unique peer and arrives at a unique 5-minute interval, making the equi-join from S_1 to S_2 a many-one join. Furthermore, if the number of peer ISPs is less than 25, ignoring the effects of computational and network latency for simplicity, the unique joining S_2 tuple of any tuple $s_1 \in S_1$ will have arrived within 25 S_2 tuples that arrive after s_1, providing the basis for a RIDS(k) constraint over the many-one join from S_1 to S_2. □

Note that we have chosen to use tuple-based constraints in our work, but time-based constraints also can be used without affecting our basic approach. The details are beyond the scope of this thesis.

6.1.4 Using Stream Properties for Reordering Tuples

As discussed in Section 2.2.5 in Chapter 2, the STREAM system includes an input manager that reorders tuples that arrive out of timestamp order, by exploiting stream properties like the degree of out-of-order arrival within a stream and the timestamp skew in the arrival of different streams. The techniques used by the input manager for reordering are not part of this thesis, and are described in [103]. In this section we describe how the stream properties and reordering techniques from [103] relate to our work on k-constraints. For clarity, we present a simplified version of the work in [103], while preserving all aspects of this work related to k-constraints.

The main stream property considered in [103] is an *interstream skew bound* that captures the lag between timestamps generated simultaneously by a pair of stream sources. Specifically, an interstream skew bound δ_{ij} between the streams generated by sources ϕ_i and ϕ_j is interpreted as follows: if at a point in time t, ϕ_i generates a tuple with timestamp τ, then the tuples generated by ϕ_j at any time $> t$ will have timestamp $> \tau - \delta_{ij}$. Intuitively, an interstream skew bound δ_{ij} resembles a RIDS(k) constraint between the streams generated by ϕ_i and ϕ_j, with timestamp as the join attribute. Note that while δ_{ij} can hold only on attributes from ordered domains (e.g., timestamps), RIDS(k) can hold on attributes from both ordered and unordered domains.

A skew bound can also be a property of a single stream, thereby capturing by how much tuples in the stream can arrive out of timestamp order. An *intrastream skew bound* δ_{ii} for the stream generated by a source ϕ_i is interpreted as follows: if at a point in time t, ϕ_i generates a tuple with timestamp τ, then the tuples generated by ϕ_i at any time $> t$ will have timestamp $> \tau - \delta_{ii}$. That is, the reordering of timestamps in the stream generated by S_i is bounded by δ_{ii}. Therefore, an intrastream skew bound is similar to an OA(k) constraint on the timestamp, which requires out of order tuples in the stream to be within k tuples of each other. There is no analogy in [103] to the CA(k) constraints used in this chapter.

The STREAM input manager's basic technique for reordering involves buffering an incoming tuple with timestamp τ, and sending the tuple to the query processor only when it can be guaranteed that no buffered tuple or future incoming tuple will have a timestamp $< \tau$. Skew bounds enable the input manager to provide such guarantees. For example, suppose the single input stream S to the DSMS has an intrastream skew bound δ. When a tuple $s \in S$ arrives with timestamp τ, the input manager can guarantee that no future incoming tuple will have a timestamp $< \tau - \delta$. At this point in time, all buffered tuples with timestamp $< \tau - \delta$ can be sent to the query processor.

Our focus is on using k-constraints to reduce run-time state in plans for windowed stream joins. It turns out that our algorithm for using OA(k) constraints for state reduction generates guarantees on tuple arrival similar to those generated when skew bounds are used for reordering tuples. However, our algorithms for using RIDS(k) and CA(k) constraints do not have similar mappings to algorithms for reordering tuples.

Finally, consider a stream S that satisfies an OA(k) constraint on attribute A. We could consider an approach where incoming S tuples are first buffered and sorted on A using the input manager's reordering techniques, so that the stream of tuples input to the query processor has $k = 0$ for the constraint on A. However, this approach has some disadvantages. First, k-constraints may hold on multiple attributes in S, so we cannot generate one ordering of S tuples where $k = 0$ for all constraints. Second, reordering based on A may destroy the timestamp-based ordering of S tuples, which is required by the STREAM query processor.

6.1.5 Queries and Execution Overview

We introduced n-way windowed stream join queries in the previous chapter. We consider the same class of queries in this chapter, like the following example CQL query that joins streams S_1 and S_2 with time-based sliding windows on each stream:

Select *
From S_1 [Range 5 Minutes], S_2 [Range 10 Minutes]
Where $S_1.A = S_2.A$

Figure 6.2 shows a plan to execute this query in the STREAM DSMS. The `seq-window` operators maintain the windows over the streams, and the `mjoin` operator processes the equi-join. Recall that each window is maintained as a synopsis in STREAM query plans. The window synopsis for stream S_1 will contain all S_1 tuples in the last 5 minutes, and the window synopsis for stream S_2 will contain all S_2 tuples in the last 10 minutes. The synopses also contain hash indexes that are used by the `mjoin` operator when it probes for joining tuples in the windows. Each window synopsis is shared between the `mjoin` operator and the corresponding `seq-window` operator. Recall from Section 2.3.1 in Chapter 2

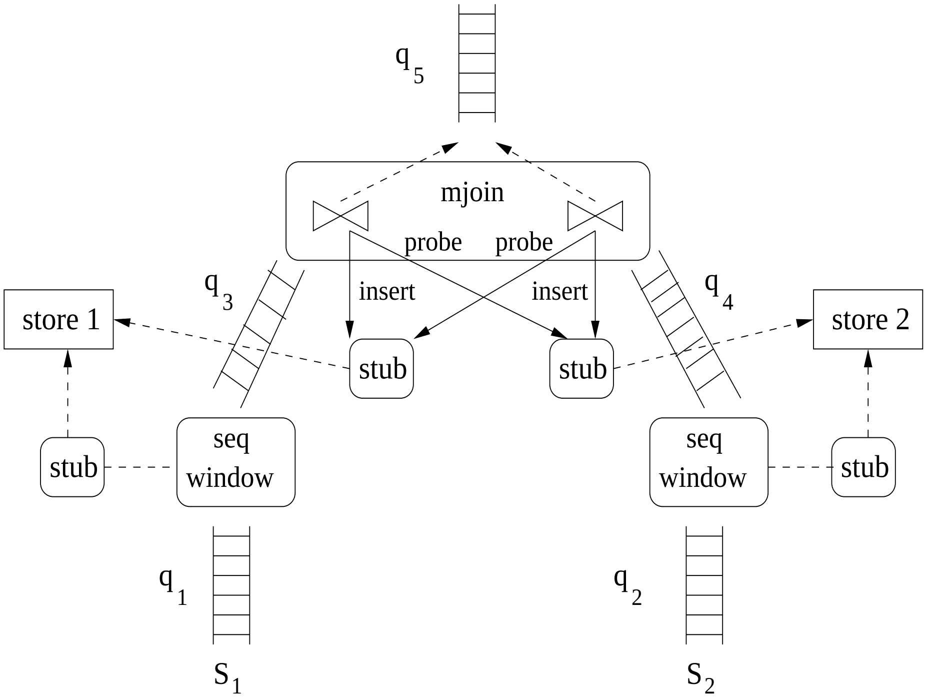


Figure 6.2: Query plan for example windowed join query

that when a set of synopses are shared, we allocate one store to hold the actual tuples, and one stub per shared synopsis to access the subset of tuples in the store that belongs to the synopsis.

The algorithm used by the `mjoin` operator in our example to process the join is an extended version of the MJoin algorithm from Chapter 5. We refer to this extended algorithm as a *sliding-window join*, or *SWJ*. SWJs are similar to MJoins in all aspects except that SWJs perform extra computation to minimize synopsis sizes in the plan, but (so far) without any knowledge or exploitation of many-one joins or k-constraints. This extra computation may increase the unit-time cost of SWJs over MJoins. However, since the focus of this chapter is on minimizing memory requirements, SWJs give us a good benchmark for comparing against our constraint-based algorithms. By comparing our constraint-based algorithms against SWJ, we can identify exactly the memory savings due to exploiting constraints.

We will work through two examples to illustrate the basic technique used by SWJs to reduce synopsis sizes. SWJs do not exploit many-one joins or k-constraints, instead they use the filter predicates in the query to identify and eliminate tuples in windows that will not contribute to join results.

Example 6.1.4 Consider the example windowed join query over streams S_1 and S_2 introduced earlier in this section. Suppose this query includes an additional filter predicate $S_1.B > 10$. Then, S_1's synopsis in an SWJ for the join will not store any tuple with $S_1.B \leq 10$, because these tuples will not contribute to join results. □

Example 6.1.5 Unlike time-based windows, tuple-based windows do not commute with filter predicates in windowed stream joins. Therefore, an SWJ is more complex when the join query has filter predicates on streams with tuple-based windows. Consider the example windowed join query over streams S_1 and S_2, with the time-based window on S_1 replaced by a "[Rows 50,000]" tuple-based window, and also containing the filter predicate $S_1.B > 10$. The filter predicate cannot be applied independently before the join since S_1's tuple-based window must be based on all tuples in S_1. However, we can discard S_1 tuples that fail the filter predicate, provided we keep track of the arrival order of the discarded tuples so that S_1's window can be maintained correctly. As an example, if the filter predicate's selectivity is 50%, then SWJ's S_1 synopsis would now contain 25,000 tuples on average (and 25,000 placeholders), instead of 50,000. □

The two examples above illustrate how SWJs reduce synopsis sizes without exploiting many-one joins or k-constraints. Now let us consider how constraints help to reduce synopsis sizes further. Suppose the join is many-one from S_1 to S_2. We can immediately eliminate any tuple in S_1's synopsis once it joins with a tuple in S_2, often reducing S_1's synopsis size considerably. If strict referential integrity (RIDS(0)) holds over this join, then we need no synopsis at all for S_1, since for a tuple $s_1 \in S_1$ and its unique joining tuple $s_2 \in S_2$, when s_1 arrives, s_2 must either appear in S_2's synopsis or it has dropped out of S_2's window. If we have referential integrity with adherence parameter k (RIDS(k)), then a tuple $s_1 \in S_1$ must be saved for at most k arrivals on S_2 after the arrival of s_1. Furthermore, if S_2 satisfies k-ordered-arrival (OA(k)) on $S_2.A$, then a tuple $s_1 \in S_1$ must be saved for at most k arrivals on S_2 following any $S_2.A$ value greater than $s_1.A$.

6.1.6 k-Mon: An Architecture for Exploiting k-Constraints

The example in the previous section illustrates how k-constraints can be used during query execution to reduce synopsis sizes considerably. We now discuss our overall query processing architecture, called k*-Mon*, that detects and exploits different types of k-constraints automatically to reduce the memory requirement for windowed stream joins. k-Mon integrates algorithms for monitoring k-constraints and exploiting them during query execution. The basic structure of k-Mon is shown in Figure 6.3.

When a windowed stream join query is registered, the Re-optimizer generates an initial query plan based on k-constraints already being tracked for previously-registered queries, if any. The Executor begins executing this plan, using the current k-constraints for memory reduction. At the same time, the Re-optimizer informs the Profiler about all additional constraints that may be used to reduce the memory requirement for this query. Identifying potentially-useful constraints for windowed stream joins is straightforward, as will be seen when our query execution algorithm is presented in Section 6.3.

The Profiler monitors input streams continuously and informs the Re-optimizer whenever k values for potentially-useful constraints change. Actually, we combine constraint monitoring with query execution whenever possible to reduce the monitoring overhead. The Re-optimizer adapts to the changes detected by the Profiler by adding or dropping k-constraints from the set of constraints being used by the Executor for memory reduction, or by adjusting their k values.

Obviously if a k value is very high (e.g., when a constraint does not hold at all, k may grow without bound), the memory reduction obtained from using the constraint may not be large enough to justify the extra computational cost. The decision of when to exploit constraints and when not to is part of a larger cost-based query optimization framework which is beyond the scope of this thesis. In this chapter, we simply assume that the Executor uses only constraints with k values lower than a specified threshold. While a query is running, the Executor will start using an unused k-constraint if its k value drops below the threshold, and will stop using the constraint if k rises above the threshold. Therefore, k-constraints that are not useful when a join query is registered, may get used later while the query is running.

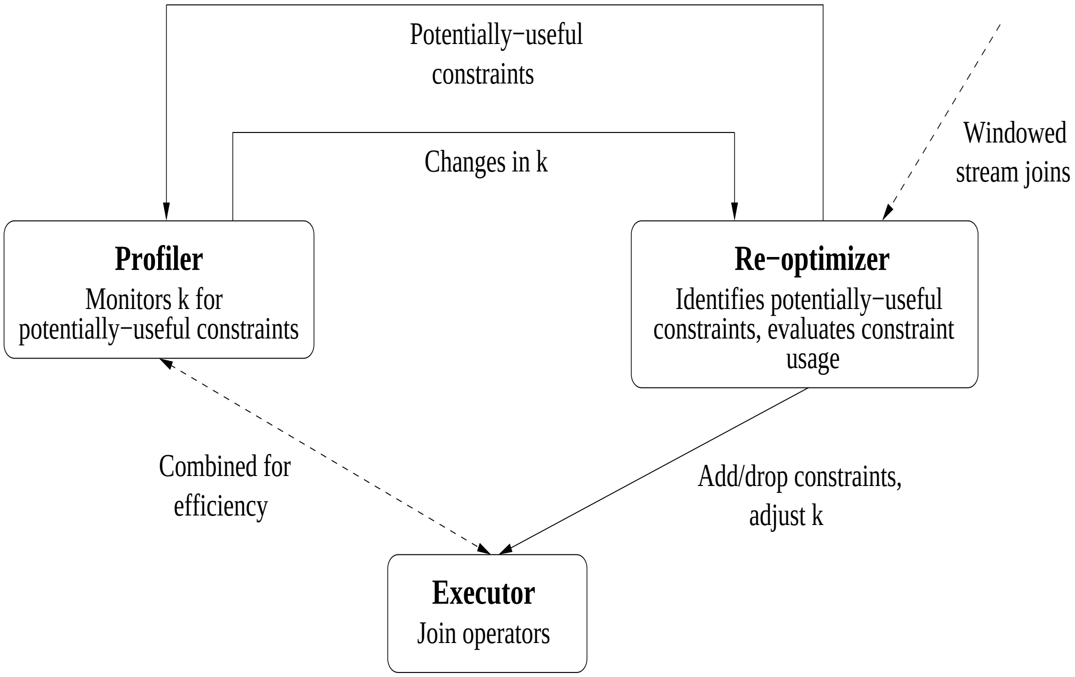

Potentially−useful constraints
Changes in k
Windowed stream joins
Profiler
Monitors k for potentially−useful constraints
Re−optimizer
Identifies potentially−useful constraints, evaluates constraint usage
Combined for efficiency
Add/drop constraints, adjust k
Executor
Join operators

Figure 6.3: k-Mon

Our query execution algorithm assumes adherence to k-constraints within the values for k given by the Profiler. Specifically, during query execution, some state may be discarded that would otherwise be saved if the constraints did not hold or if k values were higher (indicating less adherence). If our monitoring algorithms underestimate k, particularly if k increases rapidly, then for the windowed join queries we consider, *false negatives* (missing tuples) occur in query results. In many stream applications, modest inaccuracy in query results is an acceptable tradeoff for more efficient execution, especially if the inaccuracy persists for only short periods [44]. The example query in Section 6.1.1 clearly has this property. If false negatives cannot be tolerated under any circumstance, then our approach can still be used, pushing "probably unnecessary" state to disk instead of discarding it entirely. Potential joins between tuples on disk and those in memory can be detected using one of two common approaches: Join keys of tuples on disk can be retained in main-memory indexes or these join keys can be hashed into in-memory Bloom filters [24].

In this chapter we instantiate the k-Mon architecture for the referential-integrity, ordering, and clustering constraints outlined in Section 6.1.3.

6.2 Preliminaries

The queries we consider in this chapter are n-way windowed stream joins. Recall from Chapter 5 that n-way windowed stream joins in CQL have the following general form:

Select *
From S_1 [W_1], S_2 [W_2], $\cdots$, S_n [W_n]
Where *join and filter predicates over* S_1–S_n

This query joins streams S_1–S_n with window specifications W_1–W_n respectively over the streams. For exposition, we first describe our algorithm for processing joins with unbounded windows over the streams, and later extend this algorithm to handle sliding windows over the streams. That is, our initial query class consists of n-way windowed stream joins where all the window specifications W_1–W_n are equal to CQL's "`[Rows Unbounded]`" clause, represented in CQL as:

Select *
From S_1 [Rows Unbounded], S_2 [Rows Unbounded], $\cdots$, S_n [Rows Unbounded]
Where *join and filter predicates over* S_1–S_n

Extending our techniques to work with windows is straightforward and is described in Section 6.3.2.

The selection conditions we consider are conjunctions of any number of *filter predicates* over single streams along with any number of *join predicates* over pairs of streams. For clarity of presentation we assume that all join predicates in our queries represent equi-joins (e.g., $S_1.A = S_2.A$), and that the predicates are closed under implication. Note that the result of an n-way windowed stream join in CQL is a relation. Since the size of the result relation is independent of the techniques used to process the join, we do not account for the cost of storing join results.

As mentioned earlier, we assume that all joins in queries are many-one joins. That is, if a windowed stream join query Q contains one or more join predicates between streams S_1 and S_2, then we are guaranteed that each tuple on stream S_1 joins with at most one tuple on S_2 (e.g., if Q contains $S_1.A = S_2.B$ and $S_2.B$ is a key), or vice-versa. We denote a many-one join from S_1 to S_2 as $S_1 \rightarrow S_2$, and we can thus construct a *directed join graph* $G(Q)$ for any join query Q we consider. Each stream $S \in Q$ along with any filter predicates over S produces a vertex in $G(Q)$, and each join $S_1 \rightarrow S_2$ produces an edge from S_1 to S_2. We assume that the directed join graph is connected, that is, there is an undirected path between each pair of vertices in the graph. Note that directed join graphs are different from the undirected join graphs that we considered in Section 5.2.1 in Chapter 5. We use the following technical definitions:

- Given $S_1 \rightarrow S_2$, S_1 is the *parent stream* and S_2 is the *child stream*. In a directed join graph $G(Q)$, $Children(S)$ denotes the set of child streams of S and $Parents(S)$ denotes the set of parent streams of S. A stream with no parents is called a *root stream*.

- Given $S_1 \rightarrow S_2$ with joining tuples $s_1 \in S_1$ and $s_2 \in S_2$, s_2 is the unique *child tuple of* s_1, and s_1 is a *parent tuple of* s_2.

- In a directed join graph $G(Q)$ containing a stream S, $G_S(Q)$ denotes the directed subgraph of $G(Q)$ containing S, all streams reachable from S by following directed edges, the filter predicates over these streams, and all induced edges. We abuse notation and sometimes use $G_S(Q)$ to denote the result of the query corresponding to the join (sub)graph $G_S(Q)$.

- A set ρ of streams in $G(Q)$ is a *cover* of $G(Q)$ if every stream in $G(Q)$ is reachable from some stream in ρ by following directed edges. ρ is a *minimal cover* if no proper subset of ρ is a cover, and we use $MinCover(G(Q))$ to denote the set of minimal covers of $G(Q)$.

- $G(Q)$ is *directed-tree-shaped* (*DT-shaped*) if there are no cycles in the undirected version of the graph. (Recall that we assume join graphs are connected.) We only consider DT-shaped graphs in this chapter. Algorithms for *DAG-shaped* and *cyclic* join graphs are presented in [20].

Recall from Section 2.1.1 in Chapter 2 that timestamps of stream tuples are taken from a discrete ordered domain. Therefore, we consider a logical interleaving of all the tuples in the input streams based on timestamp, where timestamp ties are broken arbitrarily. We denote this totally ordered sequence as Σ. Each tuple $s \in \Sigma$ is logically tagged with its *sequence number* in Σ, denoted $\Sigma(s)$. We use the following metrics for measuring the distance between two tuples in Σ:

- **Clustering Distance:** For a pair of tuples $s_1, s_2 \in S$ with $s_1.A = s_2.A$, their clustering distance over attribute A is defined as the number of tuples $s \in S$ with $\Sigma(s_1) < \Sigma(s) < \Sigma(s_2)$ and $s.A \neq s_1.A$.

- **Scrambling Distance:** For a pair of tuples $s_1, s_2 \in S$ with $s_1.A > s_2.A$ and $\Sigma(s_1) < \Sigma(s_2)$, their scrambling distance over attribute A is defined as the number of S tuples that arrive after s_1 and up to s_2 (including s_2).

- **Join distance:** For a join $S_1 \rightarrow S_2$, the join distance for a pair of joining tuples $s_1 \in S_1$ and $s_2 \in S_2$ is defined as follows: if $\Sigma(s_1) < \Sigma(s_2)$, it is the number of S_2 tuples arriving after s_1 and up to s_2 (including s_2), otherwise it is 0.

6.3 Basic Query Execution Algorithm

In this section we define a query execution algorithm that we will use as a basis for our constraint-specific memory reduction techniques in subsequent sections. For exposition, we first describe our algorithm for processing joins with unbounded windows over the streams. In Section 6.3.2 we will extend this algorithm to handle sliding windows over the streams. We separate two aspects of processing a windowed stream join query:

1. Maintaining the synopses in the plan as new tuples arrive in the streams (*synopsis maintenance*)

2. Generating new query result tuples as they become available (*result generation*)

Consider a directed join graph $G(Q)$ for a windowed stream join query Q. Recall from the query plans in Figures 6.1 and 6.2 that we maintain a synopsis for each stream S in Q. The synopsis for S, denoted $\mathcal{S}(S)$, is partitioned logically into three components defined formally as follows.

Definition 6.3.1 *(***Synopsis Components***) Consider a time τ and a tuple $s \in S(\tau)$, where $S(\tau)$ denotes the tuples in S with timestamp $\leq \tau$. Tuple s belongs to one of three partitions of $\mathcal{S}(S)$:*

1. *$s \in \mathcal{S}(S).Yes$ at time τ if $s \ltimes G_S(Q)$ is nonempty at time τ. (Note that due to monotonicity of $G_S(Q)$, $s \ltimes G_S(Q)$ will remain nonempty for all times after τ if $s \ltimes G_S(Q)$ is nonempty at time τ.)*
2. *$s \in \mathcal{S}(S).No$ at time τ if $s \ltimes G_S(Q)$ is empty at time τ and is guaranteed to remain empty at all future times.*
3. *$s \in \mathcal{S}(S).Unknown$ at time τ if $s \notin \mathcal{S}(S).Yes$ and $s \notin \mathcal{S}(S).No$ at time τ.* □

Informally, Yes contains tuples that may contribute to a query result, No contains tuples that cannot contribute, and $Unknown$ contains tuples we cannot (yet) distinguish. This logical partitioning is required by our basic query execution algorithm to identify tuples that can be discarded to reduce synopsis sizes, without affecting the accuracy of the join result. While this partitioning could also have been applied to the synopses we used for windows and caches earlier in the thesis, reducing synopsis sizes was not our objective then. Effectively, we treated all synopsis tuples as *Unknown*.

Before we describe a method for synopsis maintenance, we state some useful lemmas.

Lemma 6.3.1 *A tuple $s \in S$ joins with at most one tuple in each stream $R \in G_S(Q)$, $R \neq S$.*

Proof: We use induction on the length of the unique directed path from S to R, denoted $l_{S \to R}$. Clearly, $l_{S \to R} \geq 1$. If $l_{S \to R} = 1$, then R is a child of S, and the claim holds because of the many-one join from S to R. This step forms the basis of the induction. As the induction hypothesis, suppose the claim holds whenever $l_{S \to R} < n$. If $l_{S \to R} = n$, $n > 1$, consider the first stream T in the directed path from S to R. A tuple $s \in S$ can join with at most one tuple $t \in T$. Since $l_{T \to R} < n$, by the induction hypothesis, t joins with at most one tuple in R. By transitivity, s can join with at most one tuple in R. Hence, the claim holds for $l_{S \to R} = n$. □

Lemma 6.3.2 *If a tuple $s \in S$ is part of a query result tuple, then $s \in \mathcal{S}(S).Yes$.*

Proof: Let t be the query result tuple that s is part of. Consider the projection of t onto the streams in $G_S(Q)$, denoted α_s. The existence of α_s shows that $s \ltimes G_S(Q) \neq \phi$, which means that $s \in \mathcal{S}(S).Yes$ by Definition 6.3.1. □

Lemma 6.3.3 *Consider a stream R that is reachable from a stream S in $G(Q)$ by following directed edges. The insertion of a tuple $s \in S$ into $\mathcal{S}(S).Yes$ cannot happen before the insertion of its unique joining tuple $r \in R$ into $\mathcal{S}(R).Yes$. (We say an event e_1 happens before an event e_2 if the set of tuples that have been processed completely when e_1 happens is a strict subset of the set of tuples that have been processed completely when e_2 happens.)*

Proof: The proof follows from Definition 6.3.1 of Yes synopsis components. □

The following theorems are based on Definition 6.3.1 of synopsis components, and they suggest a method for synopsis maintenance.

Theorem 6.3.1 *Consider any stream S, time τ, and tuple $s \in S(\tau)$ such that s satisfies all filter predicates on S. If $Children(S) = \phi$, or if for all streams $S' \in Children(S)$, $\mathcal{S}(S').Yes$ contains the child tuple of s in S', then $s \in \mathcal{S}(S).Yes$ at time τ.*

Proof: Definition 6.3.1 says that tuple $s \in \mathcal{S}(S).Yes$ for a stream S if $s \ltimes G_S(Q) \neq \phi$. Theorem 6.3.1 says that $s \in \mathcal{S}(S).Yes$ for a DT-shaped join graph $G_S(Q)$ if s satisfies all filter predicates on S, and all children of s are in the respective Yes components. We have to prove that the statements in Definition 6.3.1 and those in Theorem 6.3.1 are equivalent for DT-shaped join graphs.

We will first prove the forward direction: If $s \ltimes G_S(Q) \neq \phi$, then s satisfies all filter predicates on S, and all child tuples of s are in the respective Yes components. If $s \ltimes G_S(Q) \neq \phi$, then clearly s satisfies all filter predicates on S. The rest of the proof assumes this fact. The proof is by induction on the length of the longest directed path starting at S, denoted l_S. If $l_S = 0$, then S has no children, and the claim holds. This step forms the basis of the induction. As the induction hypothesis, let the claim hold whenever $l_S < n$. We now consider a stream S with $l_S = n$, and a tuple $s \in S$ such that $s \ltimes G_S(Q) \neq \phi$. By Lemma 6.3.1, s joins with a unique tuple in each stream $R \in G_S(Q)$, $R \neq S$. Therefore, s must generate a unique result tuple in $G_S(Q)$, denoted α_s (α_s is a joined tuple containing one tuple each from all streams in $G_S(Q)$). Now consider stream $R \in \mathit{Children}(S)$. If $r \in R$ is the unique child tuple of s, then α_s must contain r as its component tuple from R. We claim $r \ltimes G_R(Q) \neq \phi$. The proof is straightforward. By definition, $G_R(Q) \subset G_S(Q)$ for DT-shaped graphs. Thus, r will join with the same tuples in streams $T \in G_R(Q)$, that are contained in α_s. Also, $l_R < n$ by property of DT-shaped graphs. Given $r \ltimes G_R(Q) \neq \phi$ and $l_R < n$, by the induction hypothesis we know that all child tuples of R are in the respective Yes components, which puts $r \in \mathcal{S}(R).\mathit{Yes}$. Thus, we have proved that all child tuples of s are in the respective Yes components.

We will now prove the reverse direction: If s satisfies all filter predicates on S, and all child tuples of s are in the respective Yes components, then $s \ltimes G_S(Q) \neq \phi$. Again the proof is by induction on the length of the longest directed path starting at S, denoted l_S. If $l_S = 0$ the claim clearly holds. This step forms the basis of the induction. As the induction hypothesis, let the claim hold whenever $l_S < n$. We now consider a stream S with $l_S = n$. Let $R_1, R_2, \ldots, R_m$ be the children of S. Consider a tuple $s \in S$ that satisfies all filter predicates on S, and all child tuples of s are in the respective Yes components. For any child tuple $r_i \in R_i$ of s, $1 \leq i \leq m$, $r_i \in \mathcal{S}(R_i).\mathit{Yes}$ means that all child tuples of r_i are their Yes components. Since $l_{R_i} < n$ by property of DT-shaped graphs, by the induction hypothesis we know $r_i \ltimes G_{R_i}(Q) \neq \phi$. Consider the joined tuple t consisting of $s, \alpha_{r_1}, \alpha_{r_2}, \ldots, \alpha_{r_m}$, where α_{r_i} is the unique result tuple generated by r_i in $G_{R_i}(Q)$ (recall Lemma 6.3.1). We claim that t is a result tuple of $G_S(Q)$. The proof is straightforward. By property of DT-shaped graphs, no stream is common between $G_{R_i}(Q)$ and $G_{R_j}(Q)$, for $1 \leq i \leq m$, $1 \leq j \leq m$, and $i \neq j$, and there are no join predicates involving a stream in $G_{R_i}(Q)$ and a stream in $G_{R_j}(Q)$. Also, the union of all streams in $G_{R_i}(Q)$, $1 \leq i \leq m$, and S together constitute all streams in $G_S(Q)$. Since $r_i \ltimes G_{R_i}(Q) \neq \phi$, $1 \leq i \leq m$, we know that α_{r_i} satisfies the filter and join conditions over streams in $G_{R_i}(Q)$. The remaining filter predicates in $G_S(Q)$ are those over S, which are given to be satisfied by s. The remaining join predicates in $G_S(Q)$ are those involving S and one of its children, all of which are satisfied by $s, r_1, r_2, \ldots, r_m$ since $r_1, r_2, \ldots, r_m$ are the child tuples of s. Thus, t is a result tuple of $G_S(Q)$ which implies $s \ltimes G_S(Q) \neq \phi$. □

Theorem 6.3.2 *Consider any stream S, time τ, and tuple $s \in S(\tau)$. If s fails a filter predicate on S, or if for some stream $S' \in \mathit{Children}(S)$, $\mathcal{S}(S').\mathit{No}$ contains the child tuple of s, then $s \in \mathcal{S}(S).\mathit{No}$ at time τ.*

Proof: We will prove that for a tuple $s \in S(\tau)$, if s fails a filter predicate on S or if a child tuple of s is in the respective No component (i.e., if Theorem 6.3.2 adds s to $\mathcal{S}(S).\mathit{No}$), then $s \ltimes G_S(Q) = \phi$ at all times $\geq \tau$. Clearly, if s fails a filter predicate on S, $s \ltimes G_S(Q) = \phi$. We will assume this fact in the rest of the proof.

The proof is by induction on the length of the longest directed path starting at S, denoted l_S. If $l_S = 0$ the claim clearly holds. This step forms the basis of the induction. As the induction hypothesis, let the claim hold whenever $l_S < n$. We now consider a stream S with $l_S = n$. Let $r \in R(\tau)$ be the child tuple of s such that $r \in \mathcal{S}(R).\mathit{No}$ at time τ either because r fails a filter predicate on R or because a child tuple of r is in the respective No component. Since $l_R < n$ by property of DT-shaped graphs, the claim holds for $r \in R$. Thus, $r \ltimes G_R(Q) = \phi$ for all times $\geq \tau$. We will prove by contradiction that $s \ltimes G_S(Q) = \phi$ at all times $\geq \tau$. Suppose $s \ltimes G_S(Q) \neq \phi$ at time $\tau' \geq \tau$. Let α_s be the unique result tuple that s generates in $G_S(Q)$ at time τ' (recall that α_s is a joined tuple containing one tuple each from all streams in $G_S(Q)$). By the property of DT-shaped graphs, $G_R(Q) \subset G_S(Q)$. Thus, the existence of α_s implies the existence of α_r, which will be the projection of tuple α_s on to the streams in $G_R(Q)$. The existence of α_r contradicts the fact that $r \ltimes G_R(Q) = \phi$. Thus, we have shown by contradiction that $s \ltimes G_S(Q) = \phi$ at all times $\geq \tau$, which completes the proof. □

A recursive algorithm for maintaining synopsis components (i.e., inserting and deleting synopsis tuples) as stream tuples arrive follows from Theorems 6.3.1 and 6.3.2. A procedural description of this algorithm is given in Figures 6.4–6.7. The algorithm has been simplified somewhat for clarity of presentation and it is written in an object-oriented style, with

```
/* Insert tuple $s$ into the synopsis of stream $S$ */
Procedure $\mathcal{S}(S).\mathit{InsertTuple}(s)$ {
    if ($s$ fails a filter predicate on $S$) {
        /* Add $s$ to $\mathcal{S}(S).\mathit{No}$ */
        $\mathcal{S}(S).\mathit{No}.\mathit{InsertTuple}(s)$; return; }
    For each stream $R \in \mathit{Children}(S)$ {
        /* If the child tuple of $s$ in $R$ is present in $\mathcal{S}(R).\mathit{No}$, then add $s$ to $\mathcal{S}(S).\mathit{No}$ */
        if ($(s \rightarrow \mathcal{S}(R).\mathit{No}) \neq \phi$) {
            $\mathcal{S}(S).\mathit{No}.\mathit{InsertTuple}(s)$; return; }}
    For each stream $R \in \mathit{Children}(S)$ {
        /* If the child tuple of $s$ in $R$ is in $\mathcal{S}(R).\mathit{Unknown}$, add $s$ to $\mathcal{S}(S).\mathit{Unknown}$ */
        if ($(s \rightarrow \mathcal{S}(R).\mathit{Unknown}) \neq \phi$) {
            $\mathcal{S}(S).\mathit{Unknown}.\mathit{InsertTuple}(s)$; return; }
        /* Further, if the child tuple of $s$ in $R$ is not present in $\mathcal{S}(R).\mathit{Yes}$, then the
            child tuple has not yet arrived in $R$. Add $s$ to $\mathcal{S}(S).\mathit{Unknown}$ */
        if ($(s \rightarrow \mathcal{S}(R).\mathit{Yes}) = \phi$) {
            $\mathcal{S}(S).\mathit{Unknown}.\mathit{InsertTuple}(s)$; return; }}
    /* Otherwise, $S$ has no children, or all child tuples of $s$ are present in the respective
        $\mathit{Yes}$ components. Add $s$ to $\mathcal{S}(S).\mathit{Yes}$ */
    $\mathcal{S}(S).\mathit{Yes}.\mathit{InsertTuple}(s)$; }
```

Figure 6.4: Procedure invoked when a tuple s arrives in stream S

the stream synopses and their components as the objects. Procedure $\mathcal{S}(S).\mathit{InsertTuple}(s)$ in Figure 6.4 is invoked when a new tuple s arrives in input stream S. This procedure applies the criteria from Theorems 6.3.1 and 6.3.2 to determine whether s should be inserted in $\mathcal{S}(S).\mathit{Yes}$, $\mathcal{S}(S).\mathit{No}$, or $\mathcal{S}(S).\mathit{Unknown}$, and invokes $\mathcal{S}(S).\mathit{Yes}.\mathit{InsertTuple}(s)$ (Figure 6.5), $\mathcal{S}(S).\mathit{No}.\mathit{InsertTuple}(s)$ (Figure 6.6), or $\mathcal{S}(S).\mathit{Unknown}.\mathit{InsertTuple}(s)$ (Figure 6.7) appropriately. (As we will describe momentarily, if S is a root stream, then Procedure $\mathcal{S}(S).\mathit{Yes}.\mathit{InsertTuple}(s)$ joins s with the Yes components of all other streams to produce the new tuples that are generated by the arrival of s in the query result.)

In Figures 6.4–6.7 we use the notation $s \rightarrow \mathcal{S}(R).\mathit{Yes}$ ($\mathcal{S}(R).\mathit{No}$, $\mathcal{S}(R).\mathit{Unknown}$) to denote the join of tuple $s \in S$ with the Yes (No, $\mathit{Unknown}$) synopsis component of stream $R \in \mathit{Children}(S)$. Note that s will join with at most one tuple in the synopsis maintained for R. The statement $(s \rightarrow \mathcal{S}(R).\mathit{No}) \neq \phi$ in Figure 6.4, where $R \in \mathit{Children}(S)$,

```
/* Insert tuple $s$ into the $Yes$ synopsis component of stream $S$ */
Procedure $\mathcal{S}(S).Yes.InsertTuple(s)$ {
    Insert $s$ into $\mathcal{S}(S).Yes$;
    /* Propagate the effects of the insertion */
    if ($S$ is a root stream) {
        Join $s$ with the $Yes$ components of other streams to produce the new tuples in
           the query result; }
    else {
        For each stream $R \in Parents(S)$ {
        /* Reevaluate the criteria for each tuple $r$ in the $Unknown$ component of parent
            stream $R$ that joins with $s$ */
            For each tuple $r \in \mathcal{S}(R).Unknown$ {
                if (($r \rightarrow s) \neq \phi$) {
                    Delete tuple $r$ from $\mathcal{S}(R).Unknown$; $\mathcal{S}(R).InsertTuple(r)$; }}}}}
```

Figure 6.5: Procedure to insert s into $\mathcal{S}(S).Yes$

```
/* Insert tuple $s$ into the $No$ synopsis component of stream $S$ */
Procedure $\mathcal{S}(S).No.InsertTuple(s)$ {
    Insert $s$ into $\mathcal{S}(S).No$;
    /* Propagate the effects of the insertion */
    For each stream $R \in Parents(S)$ {
        /* Each tuple $r$ in parent $R$ joining with $s$ goes to $\mathcal{S}(R).No$ */
        For each tuple $r \in (\mathcal{S}(R).Yes \cup \mathcal{S}(R).Unknown)$ {
            if (($r \rightarrow s) \neq \phi$) {
                Delete tuple $r$ from its current component;
                $\mathcal{S}(R).No.InsertTuple(r)$; }}}}
```

Figure 6.6: Procedure to insert s into $\mathcal{S}(S).No$

therefore means that the child tuple in R of tuple $s \in S$ is present in $\mathcal{S}(R).No$; likewise for $\mathcal{S}(R).Yes$ and $\mathcal{S}(R).Unknown$.

Now consider result generation. By Definition 6.3.1, all tuples in the result of Q can be generated from the Yes synopsis components of the streams in $G(Q)$. We exploit the following two theorems.

Theorem 6.3.3 *New tuples are generated in the result of Q only when a tuple is inserted into the Yes synopsis component of a stream $S \in G(Q)$ where $S \in \rho \in MinCover(G(Q))$.*

```
/* Insert tuple $s$ into the $Unknown$ synopsis component of stream $S$ */
Procedure $\mathcal{S}(S).Unknown.InsertTuple(s)$ {
    Insert $s$ into $\mathcal{S}(S).Unknown$; }
```

Figure 6.7: Procedure to insert s into $\mathcal{S}(S).Unknown$

Proof: The proof is by contradiction. Suppose a query result tuple t is generated when a tuple s is inserted into the Yes synopsis component of a stream S such that S is not part of any minimal cover. Consider a minimal cover of Q, denoted ρ. Let R be a stream in ρ such that S is reachable from R. (R is not reachable from S. Otherwise, $\rho - \{R\} \cup \{S\}$ would be a minimal cover, which would give a contradiction.) Let r and s be the component tuples in t from streams R and S respectively. By Lemma 6.3.2, $r \in \mathcal{S}(R).Yes$ and $s \in \mathcal{S}(S).Yes$. By Lemma 6.3.3, we know that the insertion of r into $\mathcal{S}(R).Yes$ cannot happen before the insertion of s into $\mathcal{S}(S).Yes$ which contradicts the fact that t is generated when s is inserted. (Given our definition of "happens before" in Lemma 6.3.3, it is possible that neither the insertion of r into $\mathcal{S}(R).Yes$ nor the insertion of s into $\mathcal{S}(S).Yes$ happens before the other. However, since R is not reachable from S, we will always have to infer $s \in \mathcal{S}(S).Yes$ before we can infer $r \in \mathcal{S}(R).Yes$.) □

Theorem 6.3.4 *The set of root streams is the only minimal cover in a DT-shaped join graph.*

Proof: Theorem 6.3.4 has a straightforward proof from graph theory. □

Thus, new result tuples are generated only when a tuple s is inserted into the Yes synopsis component of a root stream in $G(Q)$. Tuple s is joined with the Yes synopsis components of all other streams to produce the new tuples in the result, as illustrated in Figure 6.5. Let us work through two examples to illustrate our algorithm so far. For presentation, all join graphs in our examples contain natural joins only.

Example 6.3.1 Consider a windowed stream join Q having the directed join graph in Figure 6.8(a). Q contains two many-one joins, $S_1 \rightarrow S_2$ ($S_1.A = S_2.A$) and $S_1 \rightarrow S_3$ ($S_1.B = S_3.B$), and a filter predicate $D < 8$ on stream S_3. A state of the synopses

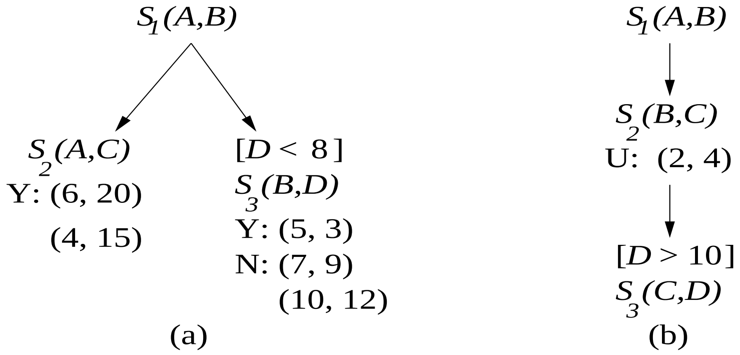


Figure 6.8: Directed join graphs used in the examples

also is shown in Figure 6.8(a): $\mathcal{S}(S_2).\mathit{Yes} = \{(6,20),(4,15)\}$, $\mathcal{S}(S_3).\mathit{Yes} = \{(5,3)\}$, $\mathcal{S}(S_3).\mathit{No} = \{(7,9),(10,12)\}$, and all other synopsis components are empty. Suppose tuple $s = (6,5)$ arrives next in S_1. Since the child tuples of s in both S_2 and S_3 are in Yes, s is added to $\mathcal{S}(S_1).\mathit{Yes}$ and result tuple $(6,5,20,3)$ is emitted. Next suppose tuple $s' = (8,10)$ arrives in S_1. The child tuple of s' in S_2 has not arrived yet. However, since the child tuple of s' in S_3 is in No, s' is added to $\mathcal{S}(S_1).\mathit{No}$. □

Example 6.3.2 Consider the directed join graph and synopses in Figure 6.8(b). $s_2 = (2,4)$ is in $\mathcal{S}(S_2).\mathit{Unknown}$ since its child tuple in S_3 has not arrived yet. Suppose tuple $s_1 = (1,2)$ arrives in S_1. Since the child tuple of s_1 in S_2 belongs to $\mathcal{S}(S_2).\mathit{Unknown}$, s_1 is added to $\mathcal{S}(S_1).\mathit{Unknown}$. Next suppose $s_3 = (4,12)$ arrives in S_3. Since s_3 satisfies the filter predicate on S_3, it is added to $\mathcal{S}(S_3).\mathit{Yes}$. As a result, s_2 is moved to $\mathcal{S}(S_2).\mathit{Yes}$, which further results in s_1 being moved to $\mathcal{S}(S_1).\mathit{Yes}$, and result tuple $(1,2,4,12)$ is emitted. □

6.3.1 Synopsis Reduction

In our basic query execution algorithm, the synopsis for a stream S simply contains each tuple of S in either Yes, No, or $\mathit{Unknown}$, thus the synopsis is no smaller than S itself. In this section we show how, even without k-constraints, we can reduce synopsis sizes under some circumstances. We present techniques to eliminate tuples from synopses as well as techniques to eliminate columns.

Eliminating Tuples

Our first technique is based on Theorem 6.3.5.

Theorem 6.3.5 *Consider a directed join graph $G(Q)$. If a stream S forms a minimal cover for $G(Q)$, i.e., $\{S\} \in MinCover(G(Q))$, then a tuple $s \in S$ inserted into $\mathcal{S}(S).Yes$ by our algorithm will not join with any future tuples to produce additional results.*

Proof: The proof follows from Lemma 6.3.1. □

By this theorem, all result tuples using s can be generated when s is (logically) inserted into $\mathcal{S}(S).Yes$, so we need not create $\mathcal{S}(S).Yes$ at all. A common case is when the join graph has a single root stream S, since for DT-shaped join graphs $\{S\}$ is a minimal cover.

Now consider No and $Unknown$ components. Informally, No components contain tuples that will never contribute to a query result, while $Unknown$ components contain tuples for which we do not yet know whether they may or may not contribute. As one simple reduction technique we can always eliminate the No component for root stream synopses. In fact we can always eliminate all No components without compromising query result accuracy, but it may not be beneficial to do so—eliminating any No tuple from nonroot streams may have the effect of leaving some tuples in parent and ancestor $Unknown$ components that may otherwise be moved to No components. If moved to No components these tuples might be discarded (if at a root) or might cause other root tuples to move to No and be discarded. The following example illustrates this behavior.

Example 6.3.3 Consider the directed join graph from Figure 6.8(b). Suppose tuple $s_3 = (6, 8)$ arrives on S_3. Since s_3 fails the filter predicate over S_3, s_3 will be added to $\mathcal{S}(S_3).No$. Suppose our policy is to discard nonroot No tuples, so s_3 is discarded. A new tuple $(3, 6)$ arrives next on S_2. Also, many tuples of the form $(x, 3)$, for arbitrary values of x, arrive on S_1. These new tuples in S_1 and S_2 will not find a joining tuple in S_3's synopsis, so they will all be stored in the respective $Unknown$ components.

Now consider the alternative policy of storing nonroot No tuples, so $s_3 \in \mathcal{S}(S_3).No$ will not be discarded. Then, by our query processing algorithm, all the new S_1 and S_2 tuples will join with $s_3 \in \mathcal{S}(S_3).No$, and go to the respective No components. Since S_1

is a root stream, we can discard all tuples in $\mathcal{S}(S_1).No$, reducing the memory requirement significantly. □

Formal modeling of the tradeoff between keeping nonroot No components or eliminating them is beyond the scope of this thesis. The presence of k-constraints and windows further complicates the tradeoff, although often k-constraints can be used to eliminate nonroot No components without any detrimental effect, as we will see in Section 6.6.1. Hereafter we assume as a default that tuples in nonroot No components are stored unless these tuples have expired from windows (Section 6.3.2) or they are eliminated by our k-constraint-based techniques.

Eliminating Columns

Handling queries where only a subset of the attributes are projected in the result does not change our basic query execution algorithm. However, such projection helps us eliminate columns from synopses. Specifically, in the synopsis of a stream S we need only store those attributes of S that are involved in joins with other streams, or that are projected in the result of the query. A second column elimination technique specific to No synopsis components is that in $\mathcal{S}(S).No$ we need only store attributes involved in joins with $Parents(S)$.

6.3.2 Handling Windows over Streams

So far in this section we restricted our query class to windowed stream join queries where each window is specified by CQL's "Rows Unbounded" clause. We now drop this restriction and explain how our basic query execution algorithm and synopsis reduction techniques are extended to handle sliding windows over streams in join queries. Specifically, we now need to handle the deletion (or expiry) of tuples from windows, which our algorithms did not have to consider so far. When a tuple s is deleted from a window, we need to take two actions:

- Maintain all synopsis components appropriately
- Generate the deletions to the join result caused by the deletion of s

Once s drops out of the window over stream S, s will no longer contribute to any new results for the join query. Therefore, by Definition 6.3.1, s should be moved to $\mathcal{S}(S).No$. The recursive algorithm from Figure 6.6 will be invoked to maintain all synopses correctly when s is inserted into $\mathcal{S}(S).No$. For example, this insertion may cause tuples from S's ancestors in the join graph to be moved to their corresponding No components. Our current implementation uses a periodic *garbage collection phase* to discard No tuples generated by the expiry of tuples from windows.

Now consider result generation when a tuple s expires from the window over stream S in a windowed stream join Q. We need to generate all tuples deleted from the join result because of the deletion of s. We saw in Section 6.3 that tuples in the join result are generated by tuples in the Yes synopsis components of the streams. Therefore, if $s \notin \mathcal{S}(S).Yes$, then we do not need to process s for result generation. Furthermore, tuples will be deleted from the result of Q only if $S \in \rho \in MinCover(G(Q))$. (The proof follows from the proof of Theorem 6.3.3.) Since the set of root streams is the only minimal cover in a DT-shaped join graph, tuples are deleted from the join result only if S is a root stream in $G(Q)$. Furthermore, the tuples deleted from the join result will be the tuples in the join of s with the Yes components of the other streams in $G(Q)$. (Recall from Figure 6.5 that new tuples are inserted in the join result when a tuple is inserted in the Yes component of a root stream in $G(Q)$.) We work through an example to illustrate the algorithm.

Example 6.3.4 Consider the directed join graph in Figure 6.8(b). Suppose a tuple-based window of size 1 is specified over stream S_3 as part of this join query. That is, the query in CQL is:

Select *
From S_1 [Rows Unbounded], S_2 [Rows Unbounded], S_3 [Rows 1]
Where $S_1.B = S_2.B$ and $S_2.C = S_3.C$ and $S_3.D > 10$

Let the current state of the synopses be: $s_1 = (1, 2) \in \mathcal{S}(S_1).Yes$, $s_2 = (2, 4) \in \mathcal{S}(S_2).Yes$, and $s_3 = (4, 12) \in \mathcal{S}(S_3).Yes$. All other synopsis components are empty. s_3 is the current tuple in the window over S_3. Now suppose s_3 expires from the one-tuple window because of the arrival of a new tuple in S_3. s_3 will be moved to $\mathcal{S}(S_3).No$, which will cause s_2 to be moved to $\mathcal{S}(S_2).No$, which further results in s_1 being moved from $\mathcal{S}(S_1).Yes$ to $\mathcal{S}(S_1).No$ (recall Figure 6.6). Since S_1 is a root stream, result generation will be invoked, and tuple $(1, 2, 4, 12)$ will be deleted from the join result. □

In this section we described how our algorithms can be extended to handle windows over streams. Hereafter, our query class consists of the full range of windowed stream join queries. Next we describe how we implemented our algorithms in a prototype continuous query processor.

6.4 Overview of k-Mon's Implementation

The STREAM prototype was in its early stages of development when most of the work described in this chapter was done. Therefore, we first prototyped the k-Mon architecture and our algorithms in a different continuous-query processor, called the *k-Mon system*, which is a simpler scaled-down version of the STREAM DSMS. The k-Mon system supports only windowed stream join queries in CQL. The basic query execution architecture of the k-Mon system is very similar to that of the STREAM DSMS. However, unlike STREAM, the k-Mon system can process only one continuous query at a time.

Like STREAM, query plans in the k-Mon system are composed of operators, queues, and synopses. The operators supported by the k-Mon system are a strict subset of STREAM's operators, and they perform the same functions. (Table 2.1 in Chapter 2 lists the operators supported by STREAM.) The operators in the k-Mon system include `select`, `project`, `mjoin`, and `seq-window` (only tuple-based and time-based windows are supported). The operators in a query plan are scheduled for execution by a simple FIFO scheduler; recall Section 2.3.2 in Chapter 2.

The k-Mon system supports synopses that store relations corresponding to sliding windows in query plans. Window synopses are always shared between the `mjoin` operator and the corresponding `seq-window` operator. The window synopses maintain hash indexes for efficient equi-join processing.

We implemented the k-Mon architecture in the k-Mon system as described in Section 6.1.6 and illustrated in Figure 6.3. Our basic approach was similar to that used to implement A-Greedy and its variants (Section 4.7 in Chapter 4) as well as A-Caching (Section 5.5.6 in Chapter 5). We incorporated the k-Mon architecture into the `mjoin` operator by implementing `mjoin` as a combination of three interacting components: an Executor, a Profiler, and a Re-optimizer. These components of `mjoin` implement the functionality of the corresponding components of k-Mon shown in Figure 6.3.

When a windowed stream join query is registered with the k-Mon system, a straightforward plan is generated that contains an `mjoin` operator to process the join. `mjoin`'s Executor begins executing the join using the basic algorithm from Section 6.3. The Executor uses k-constraints on the input streams for memory reduction. The algorithms for exploiting different types of k-constraints will be described in Sections 6.6.1, 6.7.1, 6.8.1, and 6.8.5.

When query execution begins, the `mjoin`'s Re-optimizer tells the Profiler which constraints can be used potentially to reduce the memory requirement for the join. The Profiler then monitors input streams continuously and informs the Re-optimizer whenever k values for any of these constraints change. Algorithms to monitor different types of k-constraints in input streams, as well as techniques to combine constraint monitoring with regular query execution, will be described in Sections 6.6.3, 6.7.3, and 6.8.3. Based on the changes in the monitored k values, the Re-optimizer adjusts the k values used by the Executor for memory reduction. Recall that constraints with k values higher than a specified threshold are not used for memory reduction.

6.5 Experimental Setup

In Sections 6.6–6.10, we will present our first set of experiments with the k-Mon system using n-way windowed join queries over synthetic streams. Experiments with data and queries from the Linear Road Benchmark are reported in Section 6.11. By default, we use tuple-based sliding windows of size 50,000 tuples on all streams. The synthetic stream generator from Section 5.8 in Chapter 5 was used to generate input streams.

Stream characteristics relevant to our experiments include the multiplicity of tuples in joins and the selectivity of filter predicates. Multiplicity of a tuple $s_2 \in S_2$ in a join $S_1 \to S_2$ is the number of S_1 tuples that join with s_2. The definition is analogous for a tuple $s_1 \in S_1$, although in this case the multiplicity has to be either 0 or 1. By default, all many-one joins $S_1 \rightarrow S_2$ in our experiments have an average multiplicity of 5 for tuples in S_2, and a multiplicity of 1 for tuples in S_1. The selectivity of a filter predicate on a stream S is the percentage of tuples in S satisfying the predicate. By default, all filter predicates in our experiments have an average selectivity of 50%.

In our experiments, we compare the performance of our constraint-based algorithms against SWJ. Recall from Section 6.1.5 that our SWJ implementation is optimized to reduce synopsis sizes as much as possible, but without any knowledge or exploitation of many-one joins or k-constraints. Comparing our constraint-based algorithms against SWJ identifies exactly the memory savings due to exploiting constraints. Furthermore, in Section 6.9 we compare the tuple-processing times of our algorithms with that of SWJ to quantify the extra computational overhead imposed by our algorithms. As we will see in Section 6.11, the scale of memory reduction enabled by our algorithms for queries in the Linear Road Benchmark improves the overall computational performance as well. All experiments were run on a 700 MHz Linux machine with 1024 KB processor cache and 2 GB memory.

6.6 Referential Integrity Constraints

In the next three sections we will consider the three different k-constraint types in turn. For each type we provide:

- The formal definition of the k-constraint type.
- Algorithms for memory reduction enabled by constraints of that type.
- Algorithms to detect and to monitor constraints of that type.
- How these algorithms are implemented in the `mjoin` operator in the k-Mon system.
- Experimental results from the k-Mon system, demonstrating memory reduction, accuracy of monitoring, and the false-negative rate when adherence to the constraint varies over time. Experimental results evaluating the computational overhead of each constraint type are presented in Section 6.9.

We begin with the data stream equivalent of standard relational *referential integrity*. Referential integrity on a many-one join from relation R_1 to relation R_2 states that for each R_1 tuple there is a joining R_2 tuple. The definition translates to streams S_1 and S_2 with a slight twist. In its strictest form, referential integrity over data streams (hereafter RIDS) on a many-one join $S_1 \rightarrow S_2$ states that the joining (child) tuple $s_2 \in S_2$ of any tuple $s_1 \in S_1$ must arrive before s_1. Unlike relational referential integrity, RIDS does not require that a joining tuple exist in S_2 for each tuple in S_1. RIDS only requires that if a joining tuple $s_2 \in S_2$ exists for a tuple $s_1 \in S_1$, then s_2 must arrive before s_1. The more relaxed *k-constraint* version states that when a tuple s_1 arrives on S_1, its joining tuple $s_2 \in S_2$ has already arrived or s_2 will arrive within k tuple arrivals on S_2. (When $k = 0$ we have the strictest form described above.)

Definition 6.6.1 *(***RIDS(k)***) Constraint RIDS(k) holds on join $S_1 \rightarrow S_2$ if, for every tuple $s_1 \in S_1$, assuming S_2 produces a tuple s_2 joining with s_1, the join distance (Section 6.2) between s_1 and s_2 is $\leq k$.* □

6.6.1 Modified Algorithm to Exploit RIDS(*k*)

Consider a directed join graph $G(Q)$. In Section 6.3.1 we discussed that No synopsis components are not strictly necessary, but eliminating No components runs the risk of leaving tuples in parent and ancestor $Unknown$ components until they drop out of their windows. RIDS constraints allow us to eliminate No components without this risk, using the following technique.

Consider a stream $S \in G(Q)$ and suppose for each stream $S' \in Parents(S)$ we have RIDS(k) on $S' \rightarrow S$, where the k values can differ across parents. We eliminate $\mathcal{S}(S).No$ entirely. Recall from Theorem 6.3.2 that our basic query execution algorithm uses $\mathcal{S}(S).No$ to determine whether a parent tuple $s' \in S'$ belongs in $\mathcal{S}(S').No$. If RIDS(k) holds with $k = 0$, then when s' arrives, its child tuple $s \in S$ must already have arrived, otherwise s' has no child tuple in S. If $s \notin \mathcal{S}(S).(Yes \cup Unknown)$ when s' arrives, then we can infer that either $s \notin S$, or s was discarded either because it belonged to $\mathcal{S}(S).No$ (which we do not keep); s' will not contribute to any result tuple so we insert s' into $\mathcal{S}(S').No$ and proceed accordingly. (Recall from Section 6.3.2 that tuples that drop out of the window over S are moved to $\mathcal{S}(S).No$.) If $k > 0$ and child tuple $s \notin \mathcal{S}(S).(\mathit{Yes} \cup \mathit{Unknown})$ when s' arrives, then $s \notin S$, or s has not arrived yet, or s arrived and was discarded for the same reasons as before. We place s' in $\mathcal{S}(S').\mathit{Unknown}$. If k more tuples arrive on S without arrival of the child tuple s, we can infer that s will not arrive in future; we move s' to $\mathcal{S}(S').No$ and proceed accordingly.

Example 6.6.1 Consider the join graph and synopses shown in Figure 6.8(a). Suppose RIDS(1) holds on $S_1 \rightarrow S_3$, so we eliminate $\mathcal{S}(S_3).No$. Now suppose $s_1 = (4, 10)$ arrives on S_1. (s_1's child tuple $(10, 12) \in \mathcal{S}(S_3).No$ had arrived earlier and was discarded.) RIDS(1) specifies that the first S_3 tuple arriving after s_1 will be s_1's child tuple, or else either s_1 has no child tuple in S_3 or the child tuple must have arrived before s_1. Hence, s_1 can be moved to $\mathcal{S}(S_1).No$ and thus dropped (recall Section 6.3.1) as soon as the next tuple arrives in S_3. □

6.6.2 Implementing RIDS(k) Usage

We have implemented our algorithm for exploiting RIDS(k) constraints in the `mjoin` operator in the k-Mon system. To exploit RIDS(k) for $k = k_u$ over $S' \rightarrow S$, we maintain a counter C_S of tuples that have arrived on S, and an extra sequence-number attribute in $\mathcal{S}(S').\mathit{Unknown}$, denoted $C_{S' \rightarrow S}$, along with an index on this attribute that enables range scans. When a tuple $s' \in S'$ is inserted on arrival into $\mathcal{S}(S').\mathit{Unknown}$ because (possibly among other factors) its child tuple $s \in S$ is not present in $\mathcal{S}(S).\mathit{Yes} \cup \mathcal{S}(S).\mathit{Unknown}$, we set $s'.C_{S' \rightarrow S} = C_S$ and insert an entry for s' into the index on $C_{S' \rightarrow S}$. For each $s' \in \mathcal{S}(S').\mathit{Unknown}$ that joins with a newly arriving tuple $s \in S$, we delete the index entry corresponding to $s'.C_{S' \rightarrow S}$. (The join distance between s' and s is $C_S - s'.C_{S' \rightarrow S}$, which is used by the monitoring algorithm in Section 6.6.3.)

A periodic garbage collection phase uses the index on $C_{S' \rightarrow S}$ to retrieve tuples $s' \in \mathcal{S}(S').\mathit{Unknown}$ that have $s'.C_{S' \rightarrow S} + k_u \leq C_S$. Because of RIDS($k_u$) on $S' \rightarrow S$, $s'.C_{S' \rightarrow S} + k_u \leq C_S$ guarantees that the child tuple $s \in S$ of s' will not arrive in the future. Thus, we can infer that s' will not contribute to any future result tuple, we move s' to $\mathcal{S}(S').No$, and propagate the effects of this insertion in the usual manner. We also delete the index entry corresponding to $s'.C_{S' \rightarrow S}$.

6.6.3 Monitoring RIDS(*k*)

Our general goals for constraint monitoring by the Profiler are to inform the Re-optimizer, and thereby the Executor, about changes in k for relevant constraints (recall Figure 6.3). We should not incur too much memory or computational overhead in the monitoring process while still maintaining good estimates. If our estimate for k is higher than the actual value exhibited in the data, then our algorithm always produces correct answers, but will not be as memory-efficient as possible. However, if we underestimate k, then false negatives may be introduced, as discussed in Section 6.1.6. In addition to maintaining good estimates efficiently, we also do not want to react too quickly to changes observed in the data, since the changes may be transient and it may not be worthwhile changing query execution strategies for short-lived upward or downward "spikes."

We now describe how k-Mon's Profiler, as implemented in the `mjoin` operator, estimates k for a RIDS constraint on join $S' \rightarrow S$. As we will see, detecting decreases in k is easy, while detecting increases poses our real challenge. Let:

- k_u denote the current value of k used by the Executor. Initially $k_u = \infty$.
- $k_e = c \cdot k_u$ for $c \geq 1$ denote the largest increase to k that the Profiler is guaranteed to detect. c is a configuration parameter: a large c requires more memory but can provide more accurate results.
- p denote the probability that an additional tuple is kept to detect k values even higher than k_e.
- W denote a window over which observed values are taken into account for adjustments to k. W is a configuration parameter that controls how quickly the Profiler reacts to changes.

Our algorithm proceeds as follows. Logically, the Profiler "mirrors" the RIDS-based join algorithm of Section 6.6.1, but using $k_e \geq k_u$ instead of k_u. In reality (and in our implementation), monitoring is integrated into query execution so we don't duplicate state or computation, but for presentation purposes let us assume they are separate. For each newly arriving tuple $s \in S$, we compute the maximum join distance over all parent tuples of s

in $\mathcal{S}(S').Unknown$ as described in Section 6.6.2. If the maximum observed join distance for tuples in S is $k' < k_u$ for the last W tuple arrivals in S, then we set $k_u = k'$ (and consequently $k_e = c{\cdot}k'$) and notify the Re-optimizer and Executor accordingly.

Increases in k are more difficult for two reasons: (1) In order to detect increases, we need more data than would otherwise be kept for query execution. (2) Unlike decreases, increases introduce false negatives. As part of (1), we ensure that any tuple in $\mathcal{S}(S').Unknown$ that is moved to $\mathcal{S}(S').No$ by the execution algorithm because of RIDS(k_u) is logically retained in $\mathcal{S}(S').Unknown$ until the tuple can be moved because of RIDS($k_e = c{\cdot}k_u$), $c \geq 1$. This step ensures that an increase in k up to k_e will be detected at the potential cost of lower memory reduction than permitted by k_u. In addition, each tuple $s' \in \mathcal{S}(S').Unknown$ is, with probability p, retained until s' drops out of S''s window specified in the query, if s' would otherwise be discarded because of RIDS(k_u). Effectively, we are sampling in order to detect increases in k to values even higher than k_e. To address issue (2), as soon as an increase in k is detected, we conservatively set $k_u = \infty$ and notify the Re-optimizer and Executor, so the Executor stops using the constraint and possibly generating additional false negatives. (Recall from Section 6.1.6 that the Executor ignores constraints with k_u values higher than some threshold.) The value of k_u will be reset by decrease detection after W more tuples have arrived on S. We set W conservatively to a large value in order to reduce oscillations in k_u, thereby reducing the chances of generating false negatives.

We have taken a conservative approach to detecting and handling increases in k, in order to ensure that we retain high query result accuracy. A potential memory-accuracy tradeoff exists in this context: we can be less conservative and lower the memory requirement if the application is willing to accept the possibility of some (temporary) query result inaccuracy. A promising step, which is beyond the scope of this thesis, is to extend our algorithms to exploit this tradeoff.

6.6.4 Experimental Analysis for RIDS(*k*)

We have implemented our algorithms to use and to monitor RIDS(k) constraints in the `mjoin` operator in the k-Mon system. We now report results from an experimental study

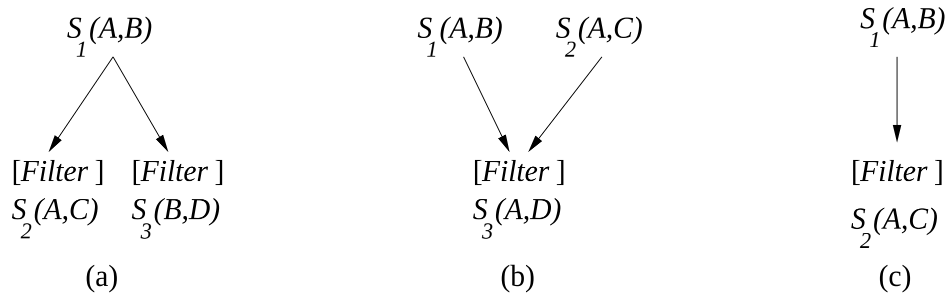


Figure 6.9: Directed join graphs used in the experiments

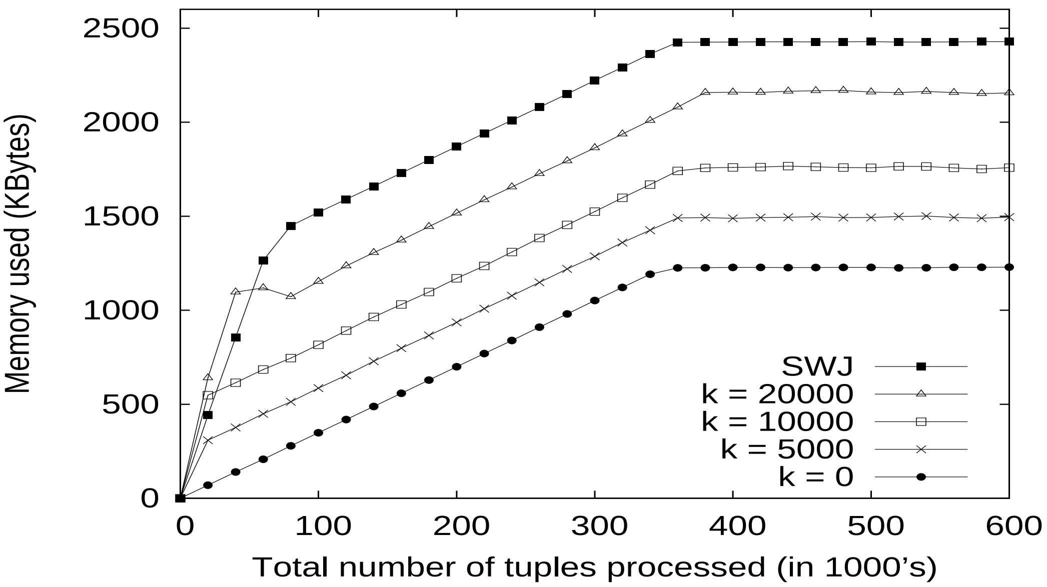


Figure 6.10: Memory reduction using RIDS(k)

of our algorithms based on this implementation. For our RIDS experiments we used the join graph in Figure 6.9(a). Figure 6.10 shows the memory reduction achieved by our query execution algorithm for different values of k. The x-axis shows the progress of time in terms of the total number of tuples processed so far across all streams. The y-axis shows the total memory used, including synopsis size and monitoring overhead. We show plots for $k \in \{0, 5000, 10000, 20000\}$ and for SWJ. For each $k = k'$, we generated synthetic data for streams S_1, S_2, and S_3 with join distances distributed uniformly in $[0, \ldots, k']$ so that RIDS(k') always holds and RIDS(k'') does not hold for any $k'' < k'$. Note that the adherence is not varied over time in this experiment. All tuple sizes are 24 bytes each in this experiment and all subsequent experiments in this chapter.

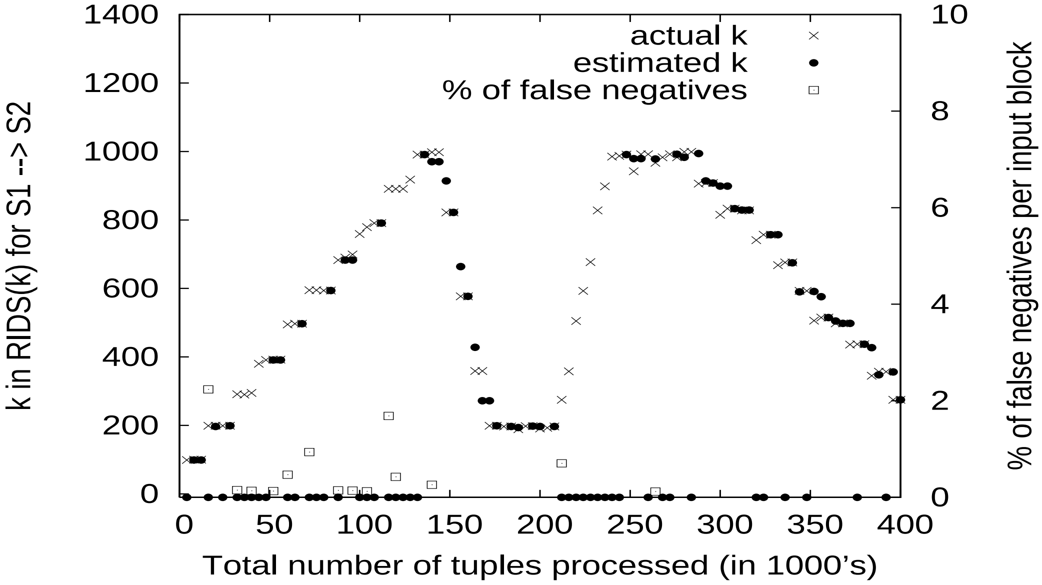


Figure 6.11: Monitoring RIDS(k)

Recall that RIDS(k) on $S_1 \rightarrow S_2$ and $S_1 \rightarrow S_3$ eliminates $\mathcal{S}(S_2).No$ and $\mathcal{S}(S_3).No$, and prevents tuples from accumulating in $\mathcal{S}(S_1).Unknown$. $\mathcal{S}(S_1).Yes$ and $\mathcal{S}(S_1).No$ are eliminated by default (Section 6.3.1). On the other hand, SWJ stores a full window of tuples for S_1, and all tuples in the windows over S_2 and S_3 that pass the respective filter predicates. The total synopsis size stabilizes around 350,000 tuples once all windows get filled so that each newly arriving tuple will displace the oldest tuple in the respective window. (S_1's window fills up around 70,000 tuples.) Figure 6.10 shows the increase in memory overhead as the adherence to RIDS decreases, i.e., as k increases.

Figure 6.11 shows the performance of the complete k-Mon framework using RIDS when k varies over time. The left y-axis shows the value of k in RIDS(k) and the right y-axis shows the percentage of false negatives per block of 4000 input stream tuples. Parameters c, p, and W for the monitoring algorithm were set to 1, 0.01, and 500 respectively. The two plots using the left y-axis show that the k estimated by our monitoring algorithm tracks the actual k in the data very closely. Five different types of variation in k are shown in Figure 6.11: no variation, gradual increase, gradual drop, quick increase, and quick drop. Points of the "estimated k" plot on the x-axis itself indicate periods when $k_u = \infty$ and the constraint is not being used. Note that the percentage of false negatives remains close to zero except during periods of increase in k, and even then it remains reasonably low ($< 2\%$). (For clarity, only the nonzero false negative percentages are shown here and in subsequent experiments.)

6.7 Clustered-Arrival Constraints

In its strictest form, a clustered-arrival constraint on attribute A of a stream S specifies that tuples having duplicate values for A arrive at successive positions in S. The relaxed k-constraint version (hereafter CA(k)) specifies that the number of S tuples with non-v values for attribute A between any two S tuples with A equal to v is no greater than k. As always, $k = 0$ yields the strictest form of the constraint. Note that CA(k) holds over a single stream, in contrast to RIDS(k) which holds over a join of two streams.

Definition 6.7.1 *(***CA(k)***) Constraint CA(k) holds on attribute A in stream S if, for every pair of tuples $s_1, s_2 \in S$ with $s_1.A = s_2.A$, the clustering distance over A between s_1 and s_2 (Section 6.2) is no greater than k.* □

6.7.1 Modified Algorithm to Exploit CA(*k*)

The benefits of RIDS(k) constraints were focused on the reduction or elimination of No and $Unknown$ synopsis components. CA(k) constraints help eliminate tuples from all three components. Elimination of tuples from Yes and $Unknown$ components is based on the following theorem.

Theorem 6.7.1 *Let S be a stream in a join graph $G(Q)$ with $Parents(S) = \{S_1, S_2, \ldots, S_n\}$. A tuple $s \in S$ will not join with any future tuples to produce result tuples if the following conditions are satisfied for some $\rho \subseteq \{S_1, S_2, \ldots, S_n\}$:*

C1: $\rho \in MinCover(G(Q))$.

C2: For all $S_i \in \rho$, no tuple in the current $\mathcal{S}(S_i).Unknown$ component joins with s.

C3: For all $S_i \in \rho$, no future tuple in S_i can join with s.

Proof: We will prove that if the three conditions in Theorem 6.7.1 are satisfied for a tuple $s \in S$, then no future result tuple can have s as its component tuple from S. The proof is by contradiction. Assume that a future result tuple t has s as its component tuple from S. Since t is a future result tuple, t must contain at least one tuple that arrived after the conditions in Theorem 6.7.1 were satisfied. Without loss of generality, let this tuple be $r \in R$. By Lemma 6.3.2, all component tuples of t belong to the respective Yes components of the streams, including all component tuples of t from streams in the minimal cover ρ in Theorem 6.7.1. From Condition C3 in Theorem 6.7.1, we can infer that $R \notin \rho$. Since ρ is a cover of $G(Q)$, there exists a stream $U \in \rho$ such that R is reachable from U. Let tuple $u \in U$ be the component tuple of t from U. From Conditions C2 and C3 in Theorem 6.7.1 we can infer that u was inserted into $\mathcal{S}(U).Yes$ before the arrival of r which contradicts Lemma 6.3.3.

Note that the proof did not use the fact that $\rho \subseteq Parents(S)$. Although this condition is not necessary for Theorem 6.7.1 to hold, it gives an efficient way to evaluate Condition C3 using CA(k) or OAP(k) constraints on attributes in $Parents(S)$ joining with S. □

Each $\rho \subseteq \{S_1, \ldots, S_n\}$ that forms a minimal cover of $G(Q)$ can be identified at query compilation time. For each such ρ, condition C2 in Theorem 6.7.1 can be evaluated at a given time by joining s with the contents of $\mathcal{S}(S_i).Unknown$. A CA(k) constraint on any one of S_i's join attributes in $S_i \rightarrow S$ for each $S_i \in \rho$ is sufficient to evaluate condition C3, as follows. Let $S_i.A = S.B$ be a predicate in the $S_i \rightarrow S$ join, with CA(k) on $S_i.A$. Once tuple s_1 arrives on S_i with $s_1.A = v$, after $k+1$ new tuples with $A \neq v$ arrive on S_i, no future S_i tuple can have $A = v$. That is, no future tuple will join with a tuple $s \in S$ with $s.B = v$.

When we determine that a tuple $s \in \mathcal{S}(S)$ satisfies conditions C1–C3 in Theorem 6.7.1, s can be eliminated. Also, any tuple in $\{S_1, \ldots, S_n\}$ that joins with s can be eliminated from whatever synopsis component it resides in. Recall from Section 6.3.1 that tuples in the No synopsis component of a stream S are used only by parents of S to move tuples from $Unknown$ to No. Therefore, a tuple $s \in \mathcal{S}(S).No$ can be removed if no future tuple in any stream $S' \in Parents(S)$ can join with s. CA(k) constraints can be used to identify such tuples as explained above.

Example 6.7.1 Consider again the join graph and synopses in Figure 6.8(a). Suppose CA(1) holds on attribute $S_1.B$ and consider the following sequence of tuple arrivals in S_1: $(6,5)$, $(8,8)$, $(4,5)$, $(11,10)$. After these arrivals, logically $\mathcal{S}(S_1).Yes = \{(6,5),$ $(4,5)\}$, logically $\mathcal{S}(S_1).No = \{(11,10)\}$, $\mathcal{S}(S_1).Unknown = \{(8,8)\}$, and result tuples $(6,5,20,3)$ and $(4,5,15,3)$ are emitted (recall we do not store $\mathcal{S}(S_1).Yes$ or $\mathcal{S}(S_1).No$ in this case). On $S_1.B$ two non-5 values have appeared after the first 5, so by the CA(1) constraint no future tuple $s \in S_1$ will have $s.B = 5$. Furthermore, since no tuple in $\mathcal{S}(S_1).Unknown$ has $B = 5$, the tuple $(5,3) \in \mathcal{S}(S_3).Yes$ cannot contribute to any future result tuples and can be eliminated. □

6.7.2 Implementing CA(*k*) Usage

We have implemented our algorithm for exploiting CA(k) constraints in the `mjoin` operator in the k-Mon system. We use the criteria in Theorem 6.7.1 to delete tuples from the synopsis components of a stream S if some $\rho \subseteq Parents(S)$ is a minimal cover and, for each $S' \in \rho$, we have a CA(k) constraint on any one of S''s join attributes in $S' \to S$. For each $S' \in \rho$ we maintain an auxiliary data structure, denoted *CA-Aux*$(S'.A)$, where $S'.A$ is a join attribute in $S' \to S$ on which CA(k) holds with $k = k_u$. We also maintain a counter $C_{S'}$ of tuples that have arrived on S'. Furthermore, we maintain a bitmap of size $|\rho|$ per tuple $s \in \mathcal{S}(S)$, with one bit per $S' \in \rho$ indicating whether s satisfies Conditions C2 and C3 in Theorem 6.7.1 for S'.

CA-Aux$(S'.A)$ contains elements (v, C_v), where v is an A value that arrived in S' and C_v is $C_{S'}$ minus the number of tuples with non-v values of A that arrived in S' after the very first tuple in S' with $A = v$. A hash index is maintained on the A values in *CA-Aux*$(S'.A)$. Also, the elements in *CA-Aux*$(S'.A)$ are linked together in sorted order of C_v values using a doubly-linked list.

When a tuple $s' \in S'$ arrives, the value of $s'.A$ is looked up in the hash index on *CA-Aux*$(S'.A)$. If an element $(v = s'.A, C_v)$ is present in *CA-Aux*$(S'.A)$, then we increment the corresponding C_v value by 1. (The maximum clustering distance so far over $S'.A$ between any two tuples with $S'.A = v$ is $C_{S'} - C_v$, which is used by the CA(k) monitoring algorithm in Section 6.7.3.) Otherwise, we insert the element ($v = s'.A, C_{S'}$) into *CA-Aux*($S'.A$). Both steps require very limited maintenance of the doubly-linked list linking the elements in sorted order of C_v values.

A periodic garbage collection phase uses the doubly-linked list to retrieve the elements (v, C_v) with $C_v < C_{S'} - k_u$. For these elements CA(k_u) guarantees that no future tuple in S' will have $s'.A = v$. We look up $\mathcal{S}(S').Unknown$ to determine whether any tuple $s'' \in \mathcal{S}(S').Unknown$ has $s'' = v$. If so, we skip v as per Condition C2 in Theorem 6.7.1. Otherwise, we look up $\mathcal{S}(S)$ to find whether any tuple $s \in \mathcal{S}(S)$ has $s = v$. If not, we delete (v, C_v) from *CA-Aux*($S'.A$). Otherwise, we set the bit (initially false) corresponding to S' in s's bitmap to indicate that s satisfies Conditions C2 and C3 in Theorem 6.7.1 for S'. If the bits corresponding to all streams in ρ are set in s, we delete s and all tuples in parent and ancestor streams of S that join with s. We delete (v, C_v) from *CA-Aux*($S'.A$).

6.7.3 Monitoring CA(*k*)

Monitoring CA(k) can be done very similarly to monitoring RIDS(k) as described in Section 6.6.3, except now we track clustering distances between tuples in the same stream instead of join distances across streams as in RIDS(k). With reference to Theorem 6.7.1, suppose we are monitoring CA(k) on join attribute A in stream $S_i \in Parents(S)$. As with RIDS, our monitoring algorithm mirrors query execution using $k_e = c \cdot k_u$. In reality, the two are combined. Clustering distances can be tracked during query execution as described in Section 6.7.2. If the maximum clustering distance over $S_i.A$ is observed as $k' < k_u$ for the last W tuple arrivals in S_i, then we set $k_u = k'$. We ensure that the *CA-Aux*($S_i.A$) entry corresponding to a tuple $s \in S$ that would normally be discarded because of CA(k_u) on $S_i.A$ is retained until s can be discarded because of CA(k_e). As with RIDS, this step guarantees detection of increases in k within k_e. For detecting increases beyond k_e, with probability p we retain the *CA-Aux*($S_i.A$) entry corresponding to a tuple $s \in S$, which would normally be discarded because of CA(k_u), until s logically drops out of S's window specified in the query. As with RIDS, we conservatively set $k_u = \infty$ on increase detection and the value is reset by decrease detection after W more arrivals.

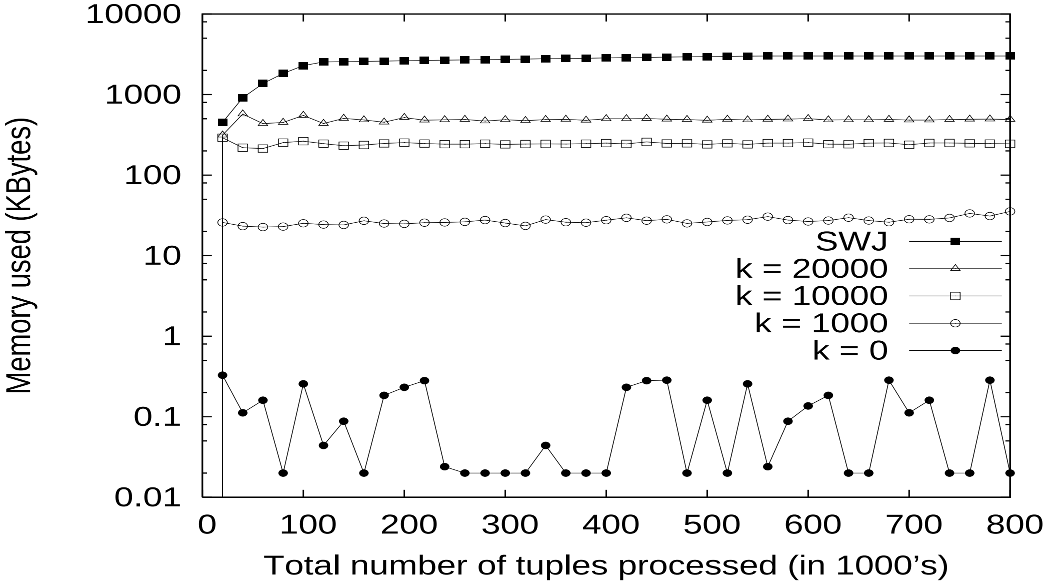


Figure 6.12: Memory reduction using CA(k)

6.7.4 Experimental Analysis for CA(*k*)

We have implemented our algorithms to use and to monitor CA(k) constraints in the `mjoin` operator in the k-Mon system. We now report results from an experimental study of our algorithms based on this implementation. For the CA experiments we used the join graph shown in Figure 6.9(b). Figure 6.12 shows the memory reduction achieved by our query execution algorithm for different values of k. (Note the log scale on the y-axis in Figure 6.12.) We generated synthetic data for streams S_1, S_2, and S_3 with different arrival orders conforming to CA(k) on both $S_1.A$ and $S_2.A$. Maximum clustering distances for distinct values of $S_1.A$ and $S_2.A$ are distributed uniformly in $[0, \ldots, k]$. The adherence is not varied over time in this experiment. To isolate the effect of the CA(k) constraints, we generated the arrival order of tuples in S_3 to satisfy RIDS(0) on $S_1 \rightarrow S_3$ and $S_2 \rightarrow S_3$. However, the RIDS constraints are not used explicitly to reduce synopsis sizes. CA(k) on the join attributes in S_1 and S_2 enables the removal of tuples from $\mathcal{S}(S_3).Yes$, $\mathcal{S}(S_3).No$, $\mathcal{S}(S_1).Yes$, and $\mathcal{S}(S_2).Yes$. Although RIDS(0) is not used, its presence in the input streams keeps the $Unknown$ components empty. Hence the total memory overhead for the CA(k) algorithm reaches its peak much before all windows fill up at around 550,000 tuples when

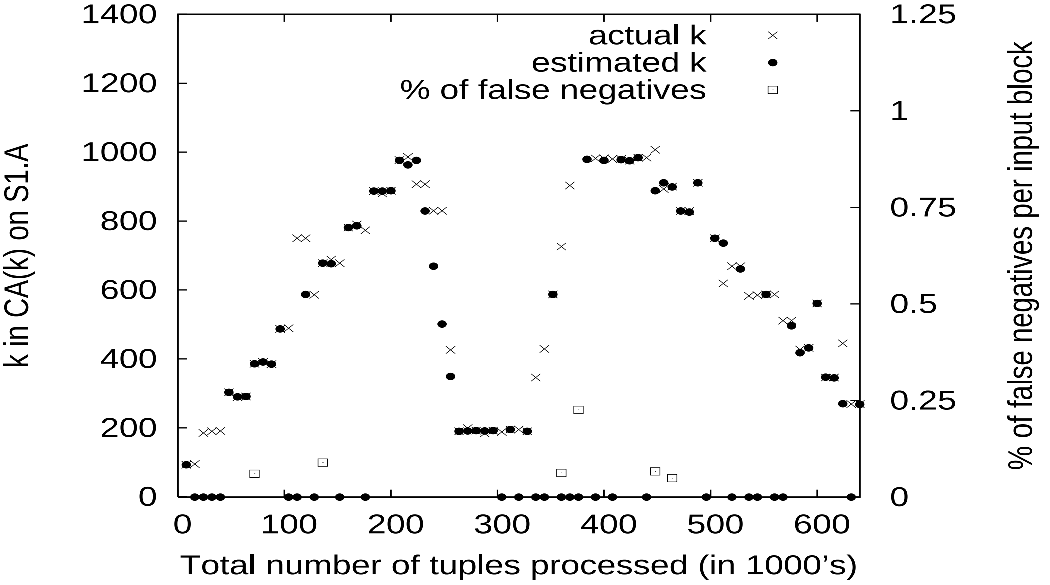


Figure 6.13: Monitoring CA(k)

the memory overhead of SWJ stabilizes. (Windows over S_1 and S_2 fill up around 110,000 tuples.)

Figure 6.13 shows the performance of k-Mon using CA when k varies over time. For this experiment, parameters c, p, and W for the monitoring algorithm were set to 1.2, 0.01, and 1000 respectively. Notice again that the k estimated by our monitoring algorithm tracks the actual k closely so the number of false negatives produced by our execution component remains close to zero. Recall from Section 6.6.4 that points of the "estimated k" plot on the x-axis indicate periods when $k_u = \infty$ and the constraint is not being used.

6.8 Ordered-Arrival Constraints

In its strictest form, an ordered-arrival constraint on attribute A of a stream S specifies that the value of A in any tuple $s \in S$ will be no less than the value of A in any tuple that arrived before s, i.e., the stream is sorted by A. (We assume ascending order; obviously descending order is symmetric.) The relaxed k-constraint version (hereafter OA(k)) specifies that for any tuple $s \in S$, S tuples that arrive at least $k + 1$ tuples after s will have a value of A that is no less than $s.A$. As always, $k = 0$ is the strictest form, and like CA(k), an OA(k) constraint holds over a single stream.

Definition 6.8.1 *(**OA(k)**) Constraint OA(k) holds on attribute A in stream S if for every pair of tuples $s_1, s_2 \in S$ with $\Sigma(s_1) < \Sigma(s_2)$ and $s_1.A > s_2.A$, the scrambling distance between s_1 and s_2 (Section 6.2) is no greater than k.* □

OA(k) is useful on join attributes, and we use it differently depending whether the constraint is on the parent stream or the child stream in a many-one join. Thus, we distinguish two classes of OA(k): *ordered-arrival of parent stream* (hereafter OAP(k)) and *ordered-arrival of child stream* (hereafter OAC(k)). The constraint monitoring algorithm is the same for both classes.

6.8.1 Modified Algorithm to Exploit OAP(*k*)

Like CA(k), OAP(k) constraints on the join attributes in streams $\{S_1, S_2, \ldots, S_n\}$ can be used to evaluate condition C3 in Theorem 6.7.1. Let $S_i.A = S.B$ be a predicate in the $S_i \to S$ join. If OAP(k) holds on $S_i.A$, once k S_i tuples have arrived after a tuple $s_i \in S_i$, no future S_i tuple can have $A < s_i.A$. That is, no future tuple will join with tuple $s \in S$ if $s.B < s_i.A$. Hence, an OAP(k) constraint on any one of S_i's join attributes in $S_i \to S$ for each $S_i \in \rho$ is sufficient to evaluate condition C3 in Theorem 6.7.1. Note an advantage of OAP(k) constraints over CA(k) constraints: in the absence of RIDS, OAP(k) constraints can always eliminate dangling tuples in S (tuples that never join), while CA(k) cannot. The algorithm can be extended in a straightforward manner to the case where a mix of CA(k) and OAP(k) constraints hold over streams in ρ in Theorem 6.7.1.

6.8.2 Implementing OAP(k) Usage

We have incorporated our algorithm for exploiting RIDS(k) constraints into the `mjoin` operator in the k-Mon system. We use the criteria in Theorem 6.7.1 to delete tuples from the synopsis components of a stream S if some $\rho \subseteq Parents(S)$ is a minimal cover and, for each $S' \in \rho$, we have an OAP(k) constraint on one of S''s join attributes in $S' \to S$. Let $S' \in \rho$ and let $S'.A$ be a join attribute in $S' \to S$ on which OAP(k) holds with $k = k_u$. Also, let max denote the maximum value of A seen so far on S'. We maintain a sliding window $[max_1, \ldots, max_{k_u+1}]$ containing the values of max after each of the last $k_u + 1$

arrivals in S', with max_1 being the most recent value. OAP(k_u) guarantees that no future tuple $s' \in S'$ will have $s'.A < max_{k_u+1}$.

In addition, for each $S' \in \rho$ we maintain an equi-width histogram, denoted *hist*$(S'.A)$, on the values of $S'.A$ in $\mathcal{S}(S').Unknown$. The histogram is implemented as a circular buffer that can grow and shrink dynamically. Whenever a tuple $s' \in S'$ is inserted into or deleted from $\mathcal{S}(S').Unknown$, the count of the bucket in *hist*$(S'.A)$ containing $s'.A$ is incremented or decremented, respectively. Whenever the count of the first bucket in *hist*$(S'.A)$, i.e., the bucket corresponding to the smallest values, drops to 0, we delete the bucket if its upper bound is $< max_{k_u+1}$. Notice that any tuple $s' \in S'$ inserted into $\mathcal{S}(S').Unknown$ will have $s'.A \geq max_{k_u+1}$.

A periodic garbage collection phase retrieves the lower bound of the first bucket in *hist*$(S'.A)$, denoted A_{lo}. If $S.B$ is an attribute in S involved in a join with $S'.A$, then any tuple $s \in S$ with $s.B < A_{lo}$ will not join with any tuple $s' \in \mathcal{S}(S').Unknown$. Thus, s satisfies Condition C2 in Theorem 6.7.1. Also, if $s.B < max_{k_u+1}$, then s will not join with any future tuple in S', satisfying Condition C3 in Theorem 6.7.1. We use an index that enables range scans on $S.B$ in $\mathcal{S}(S)$ to retrieve tuples $s \in S$ that have $s.B$ less than the minimum of A_{lo} and max_{k_u+1}. For each retrieved tuple s, we set the bit corresponding to S' in a bitmap maintained with s (similar to CA(k) usage in Section 6.7.2) to indicate that s satisfies Conditions C2 and C3 in Theorem 6.7.1 for S'. (We use the index to scan $\mathcal{S}(S)$ in non-increasing order of $S.B$ values so that we do not access tuples that were already marked in an earlier garbage collection step.) If the bits corresponding to all streams in ρ are set in s, we delete s and all tuples in parent and ancestor streams of S that join with s.

We have also experimented with other ways of implementing OAP(k) usage. The technique described here gave us the best tradeoff between memory reduction and computation time.

6.8.3 Monitoring OA(*k*)

Consider monitoring k for OA on attribute A in stream S. We use a different technique than that used for RIDS and CA, although we still integrate monitoring with query execution to avoid duplicating state and computation. As mentioned in Section 6.8.2, we maintain a sliding window $[max_1, \ldots, max_{k_u+1}]$ containing the maximum value of A after each of the last $k_u + 1$ arrivals, with max_1 being the most recent value. When a tuple $s \in S$ arrives, we compute the current maximum scrambling distance d_s involving tuple s as follows. If $s.A \geq max_1$, then $d_s = 0$ since $s.A \geq$ all values seen so far. Otherwise, we perform a binary search on the window of max values to find i such that $max_{i+1} \leq s.A < max_i$. If such an i exists, then $d_s = i \leq k_u$, otherwise $d_s > k_u$.

Consider decreases to k first. If there is a $k' < k_u$ such that all d_s values are $\leq k'$ over the last W tuple arrivals in S, then we set $k_u = k'$ and notify the Executor. We have an increase when $d_s > k_u$. As with RIDS and CA, we set $k_u = \infty$, notify the Executor, and allow k_u to be reset by decrease detection. Note that when $k_u = \infty$, the window of max values grows in size, but it can only grow indefinitely if k values increase indefinitely as well. (In practice we do not let the window grow beyond a threshold.) Finally, if we wish to speed up “convergence” of the new k value after an increase, we can maintain $k_e = c{\cdot}k_u$ elements in our window of max values for some $c > 1$.

6.8.4 Experimental Analysis for OAP(*k*)

We have implemented our algorithms to use and to monitor OAP(k) constraints in the `mjoin` operator in the k-Mon system. We now report results from an experimental study of our algorithms based on this implementation. For the OAP experiments we used the same join graph as for CA (Figure 6.9(b)). Figure 6.14 shows the memory reduction achieved by the Executor for different values of k. The data generation was similar to that for CA except here we adhere to OAP(k) on $S_1.A$ and $S_2.A$. Maximum scrambling distances for distinct values of $S_1.A$ and $S_2.A$ are distributed uniformly in $[0, \ldots, k]$. In Figure 6.14, the total memory requirement for each value of k varies around some fairly fixed value. The scale of variation is determined by the degree of out-of-order arrival in the streams, which in turn is proportional to k. Hence higher values of k cause larger variation. (Note the log scale on the y-axis in Figure 6.14.) As adherence to OAP decreases, i.e., as k increases, the peak memory overhead increases.

Figure 6.15 shows the performance of k-Mon using OAP when k varies over time. Parameters c and W for the monitoring algorithm were set to 1.2 and 1000 respectively.

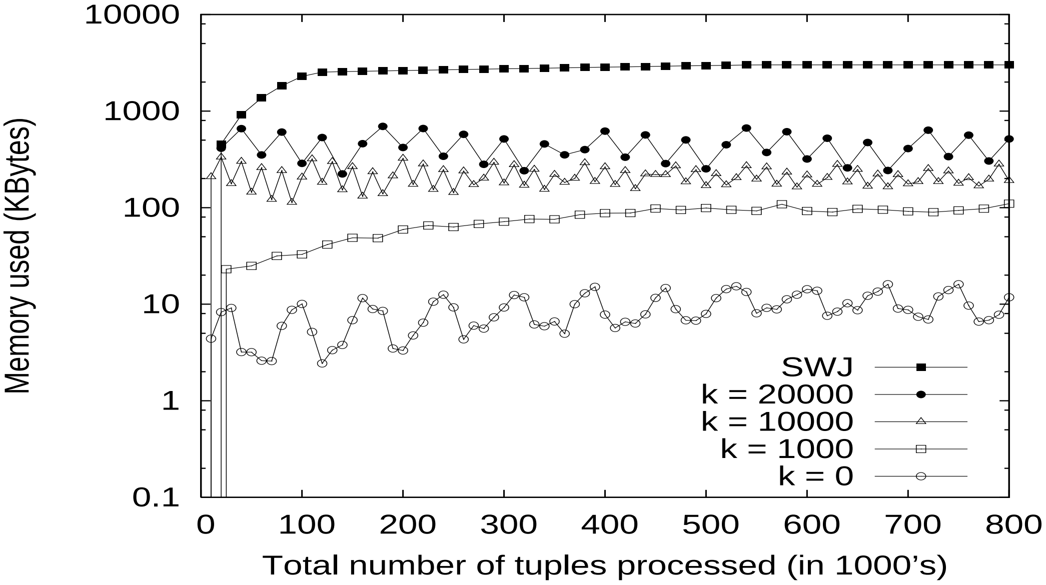


Figure 6.14: Memory reduction using OAP(k)

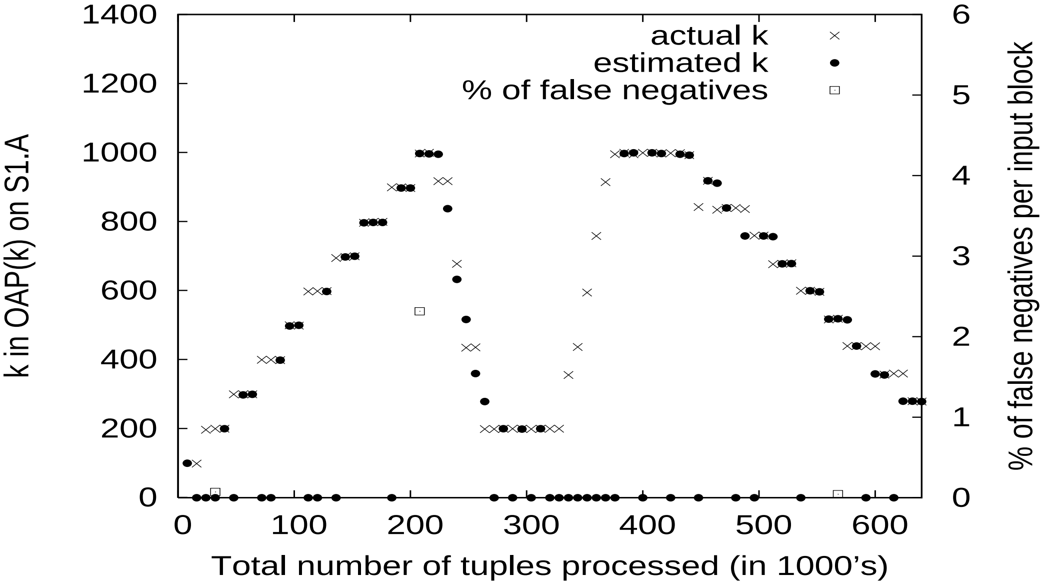


Figure 6.15: Monitoring OAP(k)

The number of false negatives produced remains close to zero except during one period of increasing k where the percentage of false negatives goes up to 2.3%.

6.8.5 Modified Algorithm to Exploit OAC(*k*)

Recall that an OAC(k) constraint is an OA(k) constraint holding on a join attribute in a child stream in a many-one join. The treatment of OAC(k) is similar to RIDS(k). OAC(k) constraints allow us to eliminate No components without running the risk of leaving tuples in parent or ancestor $Unknown$ components until they drop out of their windows (Section 6.3.1). Recall from Section 6.6 that RIDS(k) constraints are used for the same purpose.

Consider a join graph $G(Q)$ and a stream $S \in G(Q)$. Suppose for each stream $S' \in Parents(S)$ we have OAC(k) on $S.A$, where $S'.B = S.A$ is a predicate in the $S' \rightarrow S$ join. Then we can eliminate $\mathcal{S}(S).No$ entirely. Recall that our basic query processing algorithm uses $\mathcal{S}(S).No$ to determine whether a parent tuple $s' \in S'$ belongs to $\mathcal{S}(S').No$. With an OA(k) constraint on $S.A$, we can continuously maintain a value $S.A_{lo}$ such that no future tuple $s \in S$ will have $s.A < S.A_{lo}$. For a tuple $s' \in S'$ with $s'.B < S.A_{lo}$, either s''s child tuple $s \in S$ has arrived, or it will never arrive. Hence, the absence of $\mathcal{S}(S).No$ will not leave tuples blocked in $\mathcal{S}(S').Unknown$ indefinitely.

Example 6.8.1 Consider the join graph and synopses in Figure 6.8(a). Suppose OAC(2) holds on $S_3.B$, so we eliminate $\mathcal{S}(S_3).No$, and suppose the S_3 tuples shown in the figure arrived in the order $(7, 9), (5, 3), (10, 12)$. By OAC(2), $S_3.B_{lo} = 7$. Suppose a tuple $s_1 = (6, 4)$ arrives in S_1. Since $s_1.B < S_3.B_{lo}$ and $\mathcal{S}(S_3).(Yes \cup Unknown)$ does not contain s_1's child tuple in S_3, either s_1's child tuple was eliminated as part of $\mathcal{S}(S_3).No$ or s_1 is a dangling tuple. In either case, logically $s_1 \in \mathcal{S}(S_1).No$ and it can be eliminated. □

6.8.6 Implementing OAC(*k*) Usage

We have incorporated our algorithm for exploiting OAC(k) constraints into the `mjoin` operator in the k-Mon system. We exploit OAC(k) constraints to eliminate $\mathcal{S}(S).No$ if for each stream $S' \in Parents(S)$ we have OAC(k) on $S.A$, where $S.A$ is an attribute in the $S' \rightarrow S$ join. For simplicity, let us assume that all streams $S' \in Parents(S)$ are involved in a join with S on the same attribute $S.A$, and OAC(k) holds on $S.A$ for $k = k'$. It is easy to extend to the case when more than one attribute in S is involved in joins with the parent streams, and OAC(k) constraints hold on these attributes. We maintain a sliding window containing the values of $S.A$ in the last $k' + 1$ tuples in S. If we denote the values in the window as $W[0], W[1], \ldots, W[k']$, with $W[k']$ being the most recent value, OAC(k') guarantees that no future tuple $s \in S$ will have $s.A < W[0]$.

For each stream $S' \in \mathit{Parents}(S)$, we maintain an index enabling range scans on the attribute $S'.B$ involved in a join with $S.A$. During each garbage collection phase, we use this index to retrieve tuples $s' \in S'$ with $s'.B < W[0]$, which guarantees that the child tuple of s' in S will not arrive in the future. After retrieving s', we delete the entry corresponding to s' from this index. We then join s' with $\mathcal{S}(S).\mathit{Yes} \cup \mathcal{S}(S).\mathit{Unknown}$. (This join is a lookup on the hash index on $S.A$ that is used for regular join processing.) If the child tuple is not found, we move s' to $\mathcal{S}(S').\mathit{No}$ and propagate the effects of this insertion as listed in Procedure $\mathcal{S}(S).\mathit{No}.\mathit{InsertTuple}(s)$ (Figure 6.6). If the tuple is found, nothing needs to be done.

6.8.7 Experimental Analysis for OAC(*k*)

We have implemented our algorithms to use and to monitor OAC(k) constraints in the `mjoin` operator in the k-Mon system. We now report results from an experimental study of our algorithms based on this implementation. For the OAC experiments we used the join graph shown in Figure 6.9(c) and the results are shown in Figure 6.16. Streams S_1 and S_2 were generated with different arrival orders conforming to OAC(k) on $S_2.A$ for varying values of k. Maximum scrambling distances for distinct values of $S_2.A$ are distributed uniformly in $[0, \ldots, k]$. OAC(k) on $S_2.A$ eliminates $\mathcal{S}(S_2).\mathit{No}$ completely. The sharp drop in synopsis size for k =10,000 in Figure 6.16 around 60,000 tuples is because the total number of tuples in S_2 crosses 10,000 at this point and the system starts eliminating tuples from S_1 that arrived after their child tuple was dropped from $\mathcal{S}(S_2).\mathit{No}$. The corresponding drop for k = 20,000 is less dramatic because many of the tuples that could have been dropped have already dropped out of the window over S_1 (recall from 6.3.2 that we discard tuples that drop out of their respective windows). Figure 6.16 shows the increase in memory overhead as the adherence to OAC decreases, i.e., as k increases.

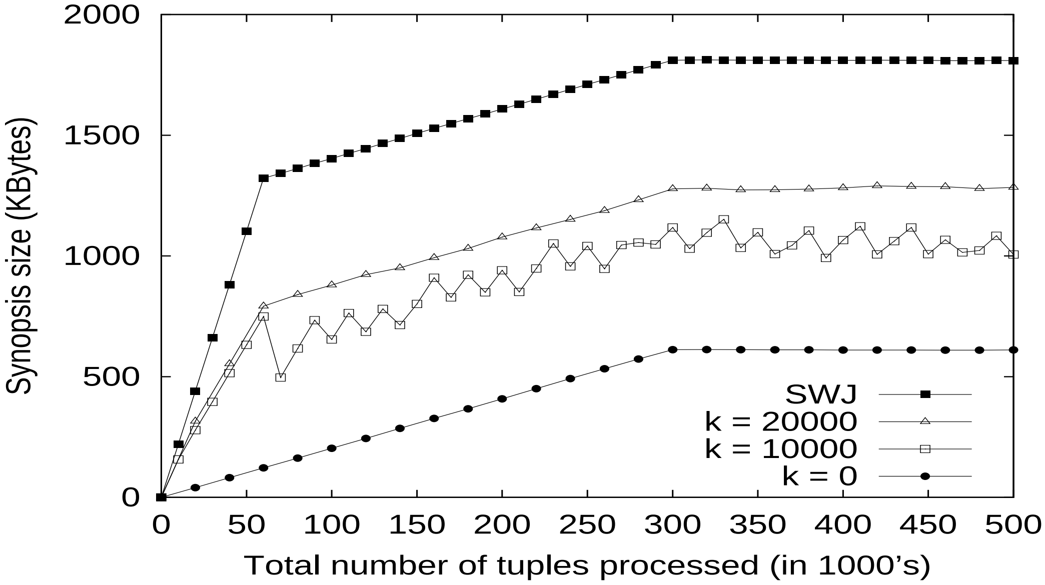


Figure 6.16: Memory reduction using OAC(*k*)

6.9 Computational Overhead

The experiments in Sections 6.6.4, 6.7.4, and 6.8.4 demonstrate the effectiveness of our k-constraint approach in reducing the memory requirement compared to SWJ. In Table 6.1 we show the per-tuple processing time for each of our algorithms for different values of k, along with SWJ which has no computational overhead apart from evaluating the join itself. Each entry in Table 6.1 is of the form $\frac{t_1(d_1)}{t_2(d_2)}$, where t_1 is the average per-tuple processing time for k-Mon, which includes monitoring and all other overhead specific to k-Mon. d_1 is half the width of the 95% confidence interval on t_1. t_2 and d_2 are the corresponding times for SWJ. These values were computed from the experiments in Figures 6.10, 6.12, and 6.14 based on the total time to process a million tuples after the system had stabilized. All times are in microseconds. The throughput achieved in our experiments was on the order of 20,000–50,000 tuples per second. Recall that all experiments were run on a 700 MHz Linux machine with 1024 KB processor cache and 2 GB memory.

The computational overhead of our approach when compared to SWJ is low for the CA, OAP, and OAC algorithms, and it remains fairly stable as k increases. However, the overhead for RIDS increases with k, going to about 64% at k =20,000. Although 64%

Alg.	k=0	k=1000	k=5000	k=10000	k=20000
RIDS	$\frac{20.1(0.84)}{24.24(0.66)}$	$\frac{40.46(1.72)}{26.17(0.94)}$	$\frac{42.86(3.62)}{28.33(1.18)}$	$\frac{45.11(3.96)}{28.71(1.68)}$	$\frac{46.01(4.84)}{28.84(2.12)}$
CA	$\frac{22.18(1.14)}{20.71(0.44)}$	$\frac{24.7(1.32)}{21.62(0.62)}$	$\frac{25.13(1.26)}{23.14(1.12)}$	$\frac{27.04(1.78)}{23.88(1.3)}$	$\frac{28.32(2.56)}{24.24(1.36)}$
OAP	$\frac{21.02(1.12)}{21.13(0.86)}$	$\frac{23.1(1.56)}{21.76(1.14)}$	$\frac{24.42(1.8)}{22.42(1.36)}$	$\frac{25.96(2.14)}{23.91(1.58)}$	$\frac{26.94(2.98)}{25.3(2.14)}$
OAC	$\frac{20.11(1.12)}{18.13(0.74)}$	$\frac{20.94(1.96)}{18.36(0.62)}$	$\frac{20.86(2.74)}{18.28(1.76)}$	$\frac{21.72(3.14)}{18.66(2.14)}$	$\frac{22.36(2.92)}{18.91(1.98)}$

Table 6.1: Tuple-processing times (microseconds) for different *k*

additional overhead per tuple may sound excessive, it can still be a viable approach if the data stream system has excess processor cycles but not enough memory to support its workload [39, 75].

6.10 Constraint Combination

In Sections 6.6–6.8 we discussed constraint types RIDS, CA, OAP, and OAC, in each case exploiting constraints of that type without considering the simultaneous presence of constraints of another type. In this section we briefly explore the interaction of multiple simultaneous constraints of different types. To begin, we review the synopsis components that may be reduced or eliminated by the four constraint types independently, summarized in Table 6.2.

It is never the case that combining constraints of different types results in a situation where we can eliminate fewer synopsis tuples than the union of the tuples eliminated by considering the constraints independently. Furthermore, in some cases combining constraints allows us to eliminate more tuples, as seen in the following example.

Example 6.10.1 Consider the join graph and synopses in Figure 6.8(a). Suppose CA(0) holds on $S_1.A$ and OAC(0) holds on $S_3.B$. Consider the following sequence of tuple arrivals in S_1: $(4,5)$, $(6,8)$, $(3,13)$. Let us consider three different situations: (i) only the CA constraint is used; (ii) only the OAC constraint is used; (iii) both constraints are used simultaneously. All three situations infer $(4,5)$ to be in $\mathcal{S}(S_1).Yes$ and drop

k-constraint for $S_1 \to S_2$	Can reduce or eliminate
Default	$\mathcal{S}(S_1).\mathit{Yes}$ if $\{S_1\}$ is a cover, $\mathcal{S}(S_1).\mathit{No}$ if S_1 is a root stream
RIDS	$\mathcal{S}(S_2).\mathit{No}$, $\mathcal{S}(S_1).\mathit{Unknown}$
CA on $S_1.A$	$\mathcal{S}(S_1).\mathit{Yes}$, non-dangling tuples in $\mathcal{S}(S_2).(\mathit{Yes} \cup \mathit{No}) \cup \mathcal{S}(S_2).\mathit{Unknown}$
OAP on $S_1.A$	$\mathcal{S}(S_1).\mathit{Yes}$, $\mathcal{S}(S_2).(\mathit{Yes} \cup \mathit{No} \cup \mathit{Unknown})$
OAC on $S_2.A$	$\mathcal{S}(S_2).\mathit{No}$, $\mathcal{S}(S_1).\mathit{Unknown}$

Table 6.2: Summary of synopsis reductions

it after result tuple $(4, 5, 15, 3)$ is emitted. When only CA(0) on $S_1.A$ is used, $(6, 8)$ ends up in $\mathcal{S}(S_1).\mathit{Unknown}$ since its child tuple in S_3 has not arrived. CA(0) infers that $(4, 15) \in \mathcal{S}(S_2).\mathit{Yes}$ will not produce any future result tuples and eliminates it. But it is unable to eliminate $(6, 20) \in \mathcal{S}(S_2).\mathit{Yes}$ because parent tuple $(6, 8)$ is in $\mathcal{S}(S_1).\mathit{Unknown}$. OAC(0) (which eliminates $\mathcal{S}(S_3).\mathit{No}$) infers $(6, 8)$ to be in $\mathcal{S}(S_1).\mathit{No}$ since a value 10 has arrived in $S_3.B$ and no tuple in $\mathcal{S}(S_3).(\mathit{Yes} \cup \mathit{Unknown})$ has $B = 8$, and eliminates $(6, 8)$. But OAC(0) on $S_3.B$ cannot eliminate any tuple in $\mathcal{S}(S_2).\mathit{Yes}$. Now consider what happens when both constraints are used simultaneously. Independently, OAC(0) will eliminate $(6, 8) \in S_1$, and CA(0) will eliminate $(4, 15) \in S_2$, as explained above. Additionally, since no tuple in $\mathcal{S}(S_1).\mathit{Unknown}$ has $A = 6$, CA(0) eliminates $(6, 20) \in \mathcal{S}(S_2).\mathit{Yes}$, which it was unable to eliminate earlier. Using both constraints simultaneously thus gives better synopsis reduction than the union of their independent reductions. □

In Figure 6.17 we report an experimental result showing the effect of combining CA and OAC constraints for the join graph in Figure 6.9(a). We generated synthetic streams S_1, S_2, and S_3 with CA(0) on $S_1.A$ and OAC(5000) on $S_3.B$. On average, 25% of the tuples in S_1 have no joining (child) tuple in S_3. Using both constraints simultaneously gives the best memory reduction in Figure 6.17. In terms of computational overhead, the per-tuple processing time is $23.14\mu s$ when OAC alone is used, $25.2\mu s$ when CA alone is used, and $28.22\mu s$ when both constraints are used simultaneously.

However, there is an interesting subtlety when we mix multiple constraint types. Although exploiting multiple constraints will never decrease the number of tuples that can be eliminated from synopses, in certain cases it can increase the length of time that tuples remain in synopses before they are eliminated, as seen in the following example.

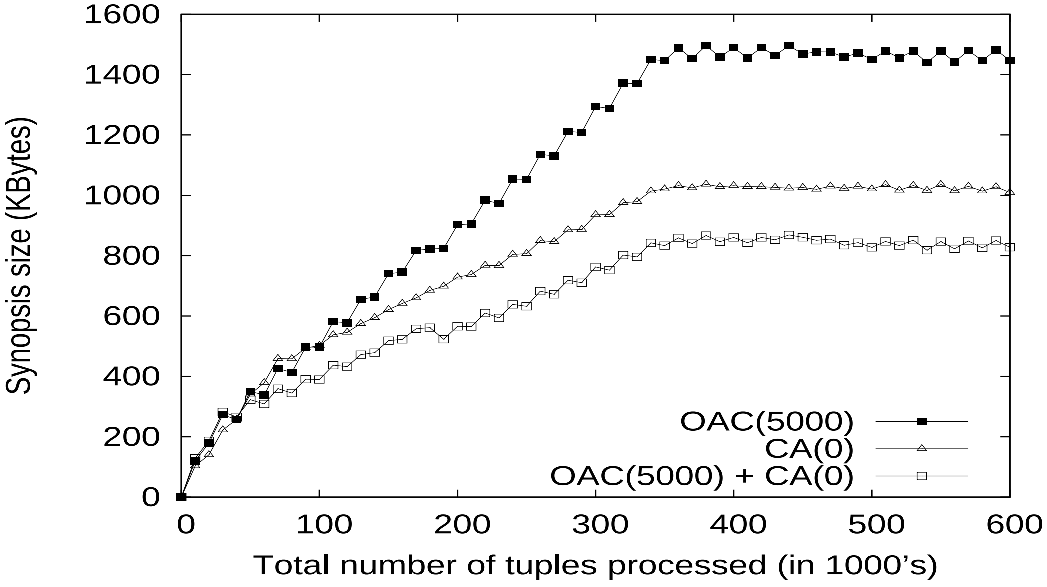


Figure 6.17: Effect of combining CA and OAC

Example 6.10.2 Consider the join graph and synopses in Figure 6.8(a). Suppose CA(0) holds on $S_1.B$ and RIDS(3) holds on the $S_1 \rightarrow S_3$ join. Let us consider two different situations: (i) only the CA constraint is used; (ii) the CA and RIDS constraints are used simultaneously. Consider the following sequence of tuple arrivals in S_1: $(6, 10), (4, 10), (8, 8)$. When only CA(0) on $S_1.B$ is used, $(6, 10)$ and $(4, 10)$ join with their child tuple $(10, 12) \in \mathcal{S}(S_3).No$ and get dropped. Also, $(10, 12) \in \mathcal{S}(S_3).No$ is eliminated since CA(0) infers that no future tuple in S_1 will join with it. If RIDS(3) is also used, $(10, 12) \in S_3$ would have been dropped on arrival since $\mathcal{S}(S_3).No$ is not stored. Thus, $(6, 10)$ and $(4, 10)$ end up in $\mathcal{S}(S_1).Unknown$ on arrival. They are dropped only after three additional tuples arrive in S_3, and hence remain in $\mathcal{S}(S_1)$ longer than when CA(0) alone is used. □

In Figure 6.18 we report an experimental result illustrating the effect. We used the join graph shown in Figure 6.9(c) for this experiment, with the filter predicate having 10% selectivity. We generated synthetic streams S_1 and S_2 with CA(5000) on $S_1.A$ and RIDS(k) on the join, varying the RIDS adherence parameter k in the experiment. $S_1 \rightarrow S_2$ has an average multiplicity of 2 for tuples in S_2 and a multiplicity of 1 for tuples in S_1. The y-axis in Figure 6.18 shows the total memory in use after 600,000 tuples have been processed. Once k increases beyond 2000 (roughly), the simultaneous use of both constraints performs

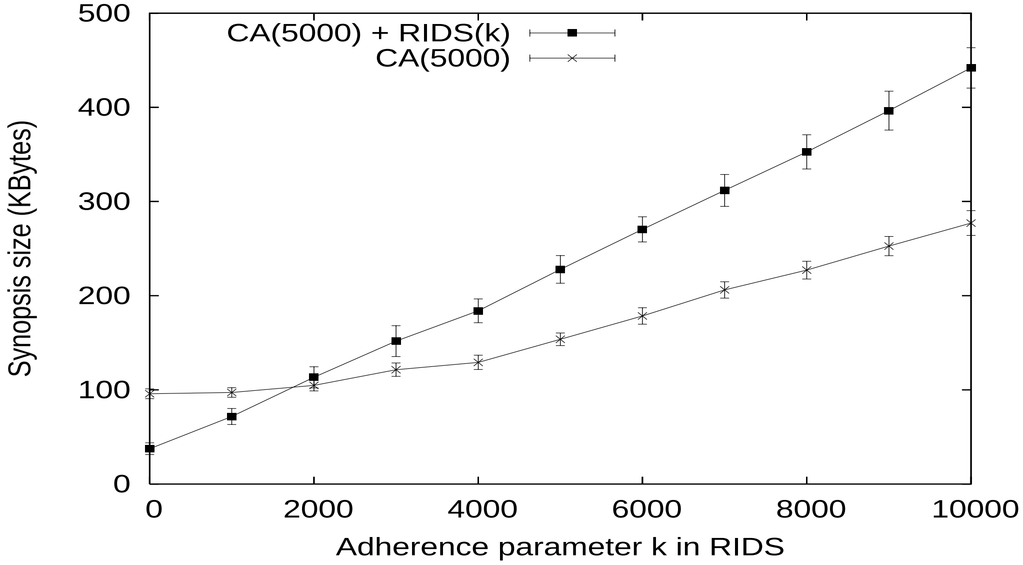


Figure 6.18: Effect of combining CA and RIDS

worse because of the extra time RIDS requires to eliminate tuples in $\mathcal{S}(S_1).Unknown$ that arrived after their child tuple was dropped from $\mathcal{S}(S_2).No$.

Based on the observations in this section, if we are interested in minimizing the *time-averaged* total synopsis size, then we are faced with the problem of selecting which constraints to exploit and which to ignore. For a complex join graph with numerous interacting constraints of different types, this *constraint selection problem* is quite difficult. Our current implementation of k-Mon uses all k-constraints with k lower than a user-specified threshold, and no others.

6.11 *k*-Constraints in the Linear Road Queries

The experiments reported so far in this chapter were done using synthetic data streams on the k-Mon system. We now show how queries in the Linear Road Benchmark, a benchmark developed for DSMSs [6], benefit from k-constraints. For this purpose we implemented our algorithms for monitoring and exploiting join constraints in the STREAM prototype DSMS. Specifically, we extended STREAM's `mjoin` operator to use many-one join constraints and RIDS(k) constraints for memory reduction, and to monitor RIDS(k) constraints automatically. (CA(k) and OA(k) constraints were not implemented in STREAM.)

```
Select distinct sid From
 (Select cid, sid From
    (CarStr [Partition By cid Rows 1]) as LastRep,
    (Select distinct cid From
     CarStr [Range 30 seconds]) as CurActiveCars
  Where LastRep.cid = CurActiveCars.cid)
  as CurCarSeg,
 (Select cid
  From CarStr [Partition By cid Rows 4]
  Group By cid
  Having count (distinct xpos) = 1 and count(*) = 4)
  as AccCars
Where CurCarSeg.cid = AccCars.cid
```

Figure 6.19: *AccSeg* query from Linear Road

Many-one join constraints are registered at the DSMS when the streams are registered. Our implementation of the algorithms for k-constraints in STREAM follows the same steps as our implementation of the corresponding algorithms in the k-Mon system; see Section 6.5.

We consider one Linear Road query in detail, then summarize our results. For presentation we simplify the main input stream of the Linear Road application to:

CarStr(cid, xpos, sid)

Each tuple in *CarStr* is a report from a sensor in a car identified by *cid*. The tuple indicates that the car was at position *xpos* in the expressway segment *sid* when the report was generated. For details see [4, 6].

One of the Linear Road queries, referred to as *AccSeg* in [4], tracks segments where accidents may have occurred. A possible accident is identified when the last four reports from a car have the same *xpos*. (*xpos* is global, not relative to segments.) The query is specified in CQL in Figure 6.19. This query uses partitioned sliding windows on *CarStr* which contain the last N ($N = 1, 4$) tuples in *CarStr* for each unique *cid*. Note that this query could have been written in a slightly simpler form by exploiting the fact that *sid* is functionally determined by *xpos*, but the more complex form is useful anyway for illustrative purposes.

Query (from [4])	Constraints	Memory used (ratio)	Tuple proc.time (ratio)
CurCarSeg	Many-one	0.09	0.65
AccSeg	RIDS	0.13	0.99
CarExitStr	RIDS	0.10	0.49
NegTollStr	RIDS	0.13	0.62

Table 6.3: Results for Linear Road queries

LastRep tracks the most recent report from each car. *CurActiveCars* tracks cars that have reported within the last 30 seconds, which are the cars active currently. *CurCarSeg* is the join of *LastRep* and *CurActiveCars*, tracking the current segment for each active car. *AccCars* tracks cars involved in recent possible accidents, and its join with *CurCarSeg* locates the segments where these cars reported from.

Linear Road simulates approximately 1 million cars [6]. Thus, joins in *AccSeg* require large synopses, e.g., the synopsis for *LastRep* can occupy around 8 Megabytes of memory. k-Mon identifies and exploits three constraints in *AccSeg*, reducing the memory requirement substantially as shown in Table 6.3. The join from *LastRep* to *CurActiveCars* (producing *CurCarSeg*) and the join from *CurCarSeg* to *AccCars* (producing *AccSeg*) are both many-one. Furthermore, RIDS(k) holds on the join from *CurCarSeg* to *AccCars* for a small value of k that is data-dependent but easily tracked through monitoring.

Eighteen single-block queries are used to express the Linear Road continuous queries in CQL [4]. Twelve of them have joins, of which seven are many-one joins. (Four out of the remaining five are a special type of spatial join.) Six out of the seven single-block queries with many-one joins benefit substantially from our technique. The constraints that apply, the memory reduction achieved by k-Mon in steady state, and the tuple processing time are given for four of these six single-block queries in Table 6.3. The remaining two queries which benefit from our technique use the same joins as one of the four queries reported here, and thus the performance improvements are identical.

The memory used and tuple processing times in Table 6.3 are ratios of the form X/Y, where X and Y are the measured steady-state values with and without using constraints, respectively. For these experiments we used a dataset provided by the authors of the Linear Road benchmark in June 2003. For the queries listed in Table 6.3, k-Mon reduces the memory requirement by nearly an order of magnitude. The scale of memory reduction enables k-Mon to reduce tuple-processing times as well. (All joins used hash indexes on *cid*.) Furthermore, k-Mon produces accurate results for all of these queries.

6.12 Related Work

Most current work on processing join queries over streams, e.g., [31, 39, 56, 61, 75], requires finite windows to be specified on all streams. Window sizes are often set conservatively in such cases to ensure with high probability that joining tuples do fall into concurrent windows, since properties of streams may not be known. The SWJ algorithm (Section 6.1.5) may waste an excessive amount of memory in this situation, while k-Mon reduces synopses to contain only the data actually needed. Furthermore, window sizes may be dictated by application-specific requirements unrelated to the properties of input streams, as in our example network monitoring query. The window sizes in this query were set to 10 minutes because the application needed to track the total traffic from a customer network that went through a specific set of links in an ISP's network within the last 10 minutes. Based on the window specifications alone, it is not possible for the query processor to identify that the synopsis sizes can be reduced considerably, without compromising result accuracy. Finally, our approach permits users to omit window specifications entirely (with the default of an unbounded window), since we use k-constraints to effectively impose the appropriate windows based on properties of the data.

The work most closely approaching ours is *punctuated data streams* [111]. Punctuations are assertions inserted into a stream to convey information on what can or cannot appear in the remainder of the stream. The query processor can use this information to reduce memory overhead for joins and aggregation and to know when results of blocking operators can be streamed. However, [111] does not address constraints over multiple streams, adherence parameters, or constraint monitoring. Reference [81] mentions how partial out-of-order arrival of streams (e.g., OA(k) on the timestamp) can be conveyed using punctuations from external stream sources to the DSMS. However, [81] does not address adaptive monitoring of these constraints at the DSMS or using constraints to reduce run-time state in stream join plans.

W-join, a multi-way windowed stream join operator supporting many types of sliding window specifications and algorithms to reduce stored data based on these specifications, is proposed in [61]. W-join does not address other types of constraints, adherence parameters, or constraint monitoring. Other techniques for controlling memory overhead in continuous query environments include using disk to buffer data for memory overflows [29, 71, 113], grouping queries or operators to minimize memory usage [34, 84], a wide variety of memory-efficient approximation techniques [44, 49, 54, 104], and run-time load shedding [15, 107]. None of these approaches consider constraints on streams as a method of reducing the data that must be stored during query processing. Reference [103] proposes techniques for reordering stream tuples when they do not arrive in timestamp order at the DSMS, as discussed in Section 6.1.4.

Reference [50] presents a language for expressing constraints over relations and views and develops algorithms to exploit the constraints for deleting data no longer needed for maintaining materialized views. However, the language and algorithms in [50] are inadequate to support constraints over streams (as opposed to relations) because streams have arrival characteristics in addition to data characteristics. Reference [66] exploits clustering based on the time of data creation to use SWJ-like techniques for joins over regular relations.

Algorithms to detect strict stream ordering or clustering with low space and time overhead are presented in [48], and [2] proposes algorithms to count the number of out-of-order pairs of stream elements. These algorithms do not address constraints over multiple streams, adherence parameters, or query processing.

6.13 Conclusion

Query plans in DSMSs often require significant amounts of run-time state to be stored and maintained while processing windowed stream joins, leading to large memory requirements. To address this problem, we proposed the concept of k-constraints: “relaxed” versions of strict constraints like referential integrity, ordering, and clustering, that are more likely to hold in data stream environments than their strict counterparts. The adherence parameter k captures how closely streams adhere to the constraint. k-constraints can be exploited to reduce run-time state in windowed stream join plans considerably without compromising the accuracy of the join result.

We considered three common types of k-constraints in a DSMS: Referential Integrity over Data Streams (RIDS(k)), Clustered Arrival (CA(k)), and Ordered Arrival (OAP(k) and OAC(k)). For each constraint type:

- We developed algorithms that can be incorporated into a plan for windowed stream joins to reduce the run-time state that needs to be stored and maintained by the plan.
- We developed algorithms for monitoring the value of k for these constraints in input streams. Our monitoring algorithms can be combined with the query execution algorithms to reduce their run-time overhead.

We reported experimental results based on an implementation of our algorithms in the k-Mon system for processing windowed joins over data streams. For a wide range of values of k, our algorithms are effective at reducing run-time memory requirements for windowed stream join plans; see Figures 6.10, 6.12, 6.14, and 6.16. Note that low to medium values of k capture situations where streams come close to satisfying constraints, but they do not satisfy strict constraints at all points in time. Our experiments also showed that k-constraints can be monitored and incorporated into query processing with low computational overhead and with minimal reduction in the accuracy of query results; see Figures 6.11, 6.13, and 6.15, and Table 6.1. We studied the interaction among different k-constraints briefly. Our results showed that using all available k-constraints may not be the best choice if we are interested in minimizing the time-averaged total synopsis size; e.g., see Figure 6.18.

Finally, we showed how queries in the Linear Road Benchmark for DSMSs benefit significantly from k-constraints. For some of these queries, our algorithms reduced the overall memory requirement by almost an order of magnitude; see Table 6.3. This scale of memory reduction also enabled our algorithms to reduce the latency of query results, up to 50% in some cases.

The Linear Road Benchmark highlights the ability of our approach to achieve good memory reduction in a complex application. The user simply provides declarative query specifications and is freed from any concern over stream properties or special execution strategies. The system detects automatically those properties of the data and queries that can be exploited to reduce the ongoing memory requirement during continuous query processing.

Chapter 7

Future Directions in Adaptive Query Processing

In this thesis we addressed the need for adaptive processing of continuous queries over data streams in a DSMS. Adaptive query processing (AQP) is required in a DSMS because stream and system conditions may change during the lifetime of long-running continuous queries. In Chapter 3 we proposed the generic StreaMon framework for AQP in a DSMS. In the subsequent chapters we described three instantiations of StreaMon:

1. The A-Greedy algorithm and its variants for adaptive processing of pipelined filter queries (Chapter 4)

2. The A-Caching algorithm for adaptive placement of subresult caches in pipelined plans for windowed stream joins (Chapter 5)

3. The k-Mon approach for detecting relaxed constraints in input streams and exploiting these constraints to reduce memory requirements in query plans for windowed stream joins (Chapter 6)

For adaptive processing of continuous queries that are more complex than those considered in this thesis, StreaMon also becomes more complex, and may incur high run-time overhead. For example, recalling StreaMon's overall architecture (Section 3.1 in Chapter 3), the Re-optimizer must search through larger plan spaces to find good plans, the

Profiler needs to monitor more statistics, and plan-switching costs in the Executor increase because of increased plan state. In Section 7.1 of this chapter we briefly present a new technique, called *plan logging*, that should enable StreaMon to handle much more complex queries adaptively.

While our focus in this thesis is on applying AQP in a DSMS, AQP is also useful in conventional DBMSs, addressing the issue that optimizers sometimes pick plans that perform poorly compared to the actual best plan. We describe how previous work has used AQP to detect and correct optimizer mistakes in DBMSs. However, previous approaches have some significant shortcomings. In Section 7.2 of this chapter we present a new technique for DBMSs, called *proactive re-optimization*, that addresses shortcomings of previous work.

7.1 Plan Logging

StreaMon currently supports AQP for relatively simple classes of continuous queries, e.g., pipelined filters and windowed stream joins. A next step is for StreaMon to handle more complex queries and larger plan spaces adaptively: to simultaneously consider access methods and plan shapes, memory allocation to operators, parallelism, and sharing of synopses like windows and caches within a query plan. However, run-time overhead to support AQP increases significantly as plan spaces expand: the Re-optimizer must search through the larger plan space, more statistics need to be tracked, and plan-switching costs increase because of increased plan state. Furthermore, larger plan-switching costs exaggerate the effect of thrashing if the system reacts too rapidly to changes. In this section, we outline *plan logging*, a promising extension to StreaMon to attack these problems.

With plan logging for a continuous query Q, the Profiler continuously logs the statistics relevant to Q that it collects, and the Re-optimizer logs the corresponding plan that it picked for Q based on these statistics. For example, entries in the log for query Q may have the form $\langle t, S, P\rangle$ indicating that the Re-optimizer picked plan P for query Q based on statistics S observed at time t.

Over time, the entries for query Q in the log effectively capture the path traversed by the system in the space of statistics relevant to Q. That is, we can consider a high-dimensional space containing a dimension for each independent input statistic relevant to Q, e.g., input stream arrival rates, filter selectivities, and join selectivities between streams. For each point S in the space of statistics relevant to Q, there is a plan P that is optimal for Q with respect to the desired cost metric, e.g., the unit-time cost metric introduced in Section 3.1 in Chapter 3. (If the properties of the input are exactly as specified by S, then plan P indeed has the minimum cost among all plans for Q.) The information in the log for Q, which captures the space of statistics relevant to Q, can be used in StreaMon as follows:

- By grouping together log entries that contain the same plan P for query Q, the system can identify regions in the space of statistics relevant to Q where P is optimal. This information can be used to pick plans without full optimization, and thus with lower overhead, when Q is re-optimized under statistical conditions that fall in regions seen previously.
- The system can identify those statistics whose changes most contribute to significant changes in the selected plan. This information can be used to reduce run-time overhead by reducing the tracking of "unimportant" statistics.
- The history captured by the log can be used to do an online "what-if" analysis. For example, the system can identify the performance that it could have achieved without adaptivity, e.g., the average performance of the single best plan over a time interval. This information can be used to track the "return-of-investment" on adaptive processing, so the system can control the run-time resources allocated to maintain adaptivity.
- The system can identify statistics that are prone to transient changes, so that it can reduce chances of thrashing on such changes.

Intuitively, by keeping track of execution history, plan logging enables StreaMon to potentially reduce the run-time overhead of AQP and to improve the speed of adaptivity over time. Therefore, plan logging is one way of overcoming the three-way tradeoff in AQP among the run-time overhead, convergence properties, and speed of adaptivity metrics, as described in Section 3.1 in Chapter 3. However, note that the benefits of plan logging are obtained only if statistical conditions repeat over time.

An approach similar to plan logging has been proposed recently in the context of conventional DBMSs. In this approach, the statistics observed during query execution, e.g., sizes of intermediate query results, are logged over time. The log is analyzed periodically, e.g., to identify correlated attributes and to build accurate histograms [105].

7.2 AQP in Conventional DBMSs

Recall from Section 1.1 in Chapter 1 that conventional DBMSs use a plan-first execute-next approach to process a query Q: the optimizer enumerates candidate query plans for Q from the many different possible plans, estimates the cost of executing each candidate plan based on statistics about data and system conditions, picks the plan with the lowest estimated cost, and runs the plan until all query results are produced. Conventional optimizers sometimes pick plans that perform significantly worse than the actual best plan. The usual cause of such optimizer mistakes is the unavailability of statistics about attribute correlations in the data [105]. For example, DBMSs typically estimate the overall selectivity of a conjunction of filter predicates involving multiple attributes (e.g., "$A > 10$ and $B < 20$ and $C > 30$") assuming that the attributes are all independent. If the attributes are actually correlated (e.g., if there is functional dependency from A to B [51]), then the selectivity may be underestimated, which could cause the optimizer to pick an index-based plan when a full scan is more efficient [74, 86]. Another cause of optimizer mistakes is the fact that statistics may not be kept up-to-date because of the high maintenance overhead involved [105].

AQP is a promising technique to address the problem of optimizer mistakes in DBMSs. Recall that with AQP the optimization and execution stages of processing a query are interleaved, possibly multiple times, over the running time of the query. In a conventional DBMS, during each execution stage statistics are collected about the intermediate results generated by the current plan (roughly analogous to the Profiler in StreaMon). If the statistics collected during execution differ significantly from the corresponding estimates made by the optimizer when it chose the current plan, then the query is re-optimized. During re-optimization, execution of the current plan is stopped, and the query is optimized again using the collected statistics. A new plan will be chosen and query execution is restarted. As an example, if the filter predicate over a table is less selective than originally estimated, then re-optimization may be invoked, and execution may switch from using an index on the table to a full scan [74, 86]. The severity of optimizer mistakes and the usefulness of AQP in solving this problem are demonstrated in [70, 71, 72, 74, 86].

However, the above approach to incorporate AQP in DBMSs is *reactive*. Specifically, this approach uses a traditional optimizer to generate a plan during each optimization stage, then reacts to estimation errors detected in the plan during the subsequent execution stage. Reactive re-optimization is limited by its use of a traditional optimizer that does not incorporate issues affecting AQP, resulting in three significant shortcomings:

- *Choosing risky plans:* Plans may be selected whose performance depends heavily on uncertain statistics, making re-optimization very likely.

- *High potential for loss of work:* Potential for reusing previous work is not considered. For example, it is hard to keep track of and reuse the partial work done by an operator pipeline if the query needs to be re-optimized while the pipeline is in execution. On the other hand, it is easy to reuse an intermediate result that is materialized completely, e.g., a hash table built by a hash join operator [17, 86].

- *Slow convergence to good plans:* No special techniques are used to collect statistics. Consequently, when re-optimization is triggered, the re-optimizer may pick a suboptimal plan again if some statistics are still uncertain.

To address the above problems, we propose *proactive re-optimization,* a new approach based on three techniques:

1. Computing the uncertainty in estimates, e.g., by computing confidence intervals.

2. Using both the estimated statistics and their uncertainty to pick plans whose performance is robust to deviations of actual values of statistics from estimates.

3. Estimating uncertain statistics quickly, accurately, and efficiently during query execution.

In Reference [17] we describe a prototype proactive re-optimizer called *Rio*. Our initial experimental results with Rio are very encouraging: When statistics are uncertain, Rio outperforms current reactive re-optimizers by up to a factor of three.

In a more general setting, with the growing popularity of *web services* it has become much easier to make any data source available online [120]. Now, a DBMS may have to execute queries involving some sources for which few or no statistics are available initially [70, 71, 72]. A plan-first execute-next approach is of limited use in this environment because estimating plan costs based on few statistics is extremely unreliable. AQP with proactive re-optimization is a promising approach to tackle this problem: by accounting for uncertainty or absence of statistics during plan selection, and by collecting statistics during query execution, a proactive approach will enable rapid convergence to good plans.

7.3 Conclusion

As continuous queries get more complex, the complexity and run-time overhead of AQP in StreaMon increases significantly: more plans must be considered, more statistics need to be monitored, and plan-switching costs increase. We proposed the plan logging technique to address this problem. With plan logging for a continuous query Q, the Profiler continuously logs the statistics relevant to Q that it collects, and the Re-optimizer logs the corresponding plan that it picked for Q based on these statistics. Over time, the logged entries contain information that can be used to reduce the run-time overhead of AQP significantly.

We also showed how AQP can be used in conventional DBMSs to detect and correct optimizer mistakes due to uncertain or stale statistics. However, current approaches to incorporate AQP in DBMSs suffer from three significant shortcomings: risky plan choices that make re-optimization highly likely, high potential for loss of work when re-optimization is required, and slow convergence to good plans. We proposed proactive re-optimization to address these shortcomings. Proactive re-optimizers compute the uncertainty in estimates of statistics, pick plans that are robust to deviations of actual values from estimates, and collect statistics quickly, accurately, and efficiently during query execution. Proactive re-optimization is also very promising for processing queries over web services and other autonomous data sources for which statistics are usually unavailable.

In general, plan logging and proactive re-optimization are orthogonal and compatible ideas. For example, when statistics are very uncertain, plan logging enables the collection of accurate statistics over time, while proactive re-optimization makes the performance of each query robust to estimation errors in statistics that are currently uncertain. In this manner, the inter-query benefits of plan logging and the intra-query benefits of proactive re-optimization can complement each other.

Finally, AQP is emerging as a core component of *autonomic* DBMSs that run efficiently with little human effort [69]. The low system administration cost of an autonomic DBMS makes it appealing to modern enterprises where the cost of administering a database can comprise up to 80% of the total cost incurred for the database in its lifetime [87]. In autonomic DBMSs, AQP can automatically ensure that query performance does not degrade due to missing statistics, missing indexes or materialized views, incorrectly set database configuration parameters, or changes in workload [17, 86, 105].

Bibliography

[1] C. Aggarwal, J. Han, J. Wang, and P. Yu. On demand classification of data streams. In *Proc. of the 2004 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 503–508, August 2004.

[2] M. Ajtai, T. Jayram, R. Kumar, and D. Sivakumar. Counting inversions in a data stream. In *Proc. of the 2002 Annual ACM Symp. on Theory of Computing*, pages 370–379, 2002.

[3] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 53–64, September 2000.

[4] A. Arasu. CQL specification of the Linear Road Benchmark. http://www-db.stanford.edu/stream/cql-benchmark.html, 2003.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB Journal*, 2005. (To appear).

[6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 480–491, September 2004.

[7] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *Proc. of the 2004 ACM Symp. on Principles of Database Systems*, pages 286–296, June 2004.

[8] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, pages 336–347, September 2004.

[9] R. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. on Computer Systems*, 21(1):36–86, 2003.

[10] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[11] A. Ayad and J. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 419–430, June 2004.

[12] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 253–264, June 2003.

[13] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *VLDB Journal*, 13(4):333–353, 2004.

[14] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 1–16, June 2002.

[15] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of the 20th Intl. Conf. on Data Engineering*, pages 350–361, March 2004.

[16] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *Proc. Second Biennial Conf. on Innovative Data Systems Research*, pages 238–249, January 2005.

[17] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, June 2005.

[18] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 407–418, June 2004.

[19] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Proc. of the 2005 Intl. Conf. on Data Engineering*, pages 118–129, April 2005.

[20] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. on Database Systems*, 29(3):545–580, 2004.

[21] S. Babu and J. Widom. StreaMon: An adaptive engine for stream query processing. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 931–932, June 2004. Demonstration proposal.

[22] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proc. of the 2004 Networked Systems Design and Implementation*, pages 197–210, March 2004.

[23] J. Beale. *Snort 2.1 Intrusion Detection*. Syngress Publishing, 2004.

[24] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[25] Y. Breitbart, M. Garofalakis, C. Martin, R. Rastogi, S. Seshadri, and A. Silberschatz. Topology discovery in heterogeneous IP networks. In *Proc. of the 2000 IEEE INFOCOMM*, pages 265–274, March 2000.

[26] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 263–274, June 2002.

[27] R. Caceres et al. Measurement and analysis of IP network usage and behavior. *IEEE Communications Magazine*, 38(5):144–151, 2000.

[28] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 838–849, September 2003.

[29] D. Carney et al. Monitoring streams–A new class of data management applications. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, pages 215–226, August 2002.

[30] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. First Biennial Conf. on Innovative Data Systems Research*, January 2003.

[31] S. Chandrasekaran and M. Franklin. PSoup: A system for streaming queries over streaming data. *VLDB Journal*, 12(2):140–156, 2003.

[32] S. Chandrasekaran and M. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, pages 384–359, September 2004.

[33] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. on Database Systems*, 24(2):177–228, 1999.

[34] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[35] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. on Database Systems*, 9(2):163–186, 1984.

[36] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic udafs at streaming speeds. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 35–46, June 2004.

[37] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 464–475, September 2003.

[38] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 647–651, June 2003.

[39] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 40–51, June 2003.

[40] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 635–644, January 2002.

[41] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49, March 2004.

[42] A. Deshpande and J. Hellerstein. Lifting the burden of history from adpative query processing. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, pages 948–959, September 2004.

[43] Y. Diao and M. Franklin. Query processing for high-volume xml message brokering. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 261–272, September 2003.

[44] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 61–72, June 2002.

[45] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. of the 2000 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 71–80, August 2000.

[46] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. of the 2000 ACM SIGCOMM*, pages 271–284, September 2000.

[47] U. Feige, L. Lovasz, and P. Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004.

[48] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. Testing and spot checking of data streams. In *Proc. of the 2000 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 165–174, 2000.

[49] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing data-stream join aggregates using skimmed sketches. In *Proc. of the 9th Intl. Conf. on Extending Database Technology*, pages 569–586, March 2004.

[50] H. Garcia-Molina, W. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 500–511, August 1998.

[51] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey, 2001.

[52] J. Gehrke. Special issue on data stream processing. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[53] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 13–24, May 2001.

[54] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 79–88, September 2001.

[55] L. Golab and T. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, June 2003.

[56] L. Golab and T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 500–511, September 2003.

[57] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[58] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. of the 2000 Annual Symp. on Foundations of Computer Science*, pages 359–366, November 2000.

[59] A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, June 1995.

[60] Hammad et al. Nile: A query processing engine for data streams. In *Proc. of the 2004 Intl. Conf. on Data Engineering*, page 851, March 2004.

[61] M. Hammad, W. Aref, and A. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proc. of the 2003 Intl. Conf. on Scientific and Statistical Database Management*, pages 75–84, June 2003.

[62] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 297–308, September 2003.

[63] J. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. on Database Systems*, 23(2):113–157, 1998.

[64] J. Hellerstein, M. Franklin, et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[65] J. Hellerstein and J. Naughton. Query execution techniques for caching expensive methods. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pages 423–434, June 1996.

[66] S. Helmer, T. Westmann, and G. Moerkotte. Diag-join: An opportunistic join algorithm for 1:n relationships. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 98–109, August 1998.

[67] D. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.

[68] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proc. of the 2001 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, August 2001.

[69] *Autonomic Computing*. http://www.research.ibm.com/autonomic/.

[70] Z. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, Seattle, WA, USA, August 2002.

[71] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

[72] Z. Ives, A. Halevy, and D. Weld. Adapting to source properties in processing data integration queries. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 395 – 406, 2004.

[73] H. V. Jagadish, I. Mumick, and A. Silberschatz. View maintenance issues for the Chronicle data model. In *Proc. of the 1995 ACM Symp. on Principles of Database Systems*, pages 113–124, May 1995.

[74] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 106–117, June 1998.

[75] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, March 2003.

[76] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object-bases. In *Proc. of the 1992 Intl. Conf. on Very Large Data Bases*, pages 79–90, August 1992.

[77] N. Koudas, B. Ooi, K. Tan, and R. Zhang. Approximate NN queries on streams with guaranteed error/performance bounds. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, pages 804–815, September 2004.

[78] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. of the 1986 Intl. Conf. on Very Large Data Bases*, pages 128–137, August 1986.

[79] S. Krishnamurthy et al. TelegraphCQ: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, March 2003.

[80] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 461–472, August 2000.

[81] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pages 311–322, June 2005.

[82] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4):583–590, August 1999.

[83] D. Lomet and A. Levy. Special issue on adaptive query processing. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[84] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, June 2002.

[85] V. Markl, V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 659–670, June 2004.

[86] V. Markl, V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 659–670, June 2004.

[87] *Fighting Complexity in IT*. McKinsey and Company. http://www.forbes.com/technology/2003/03/04/cx_0304mckinsey.html.

[88] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2003.

[89] K. Munagala, S. Babu, J. Widom, and R. Motwani. The pipelined set cover problem. In *Proc. of the 2005 Intl. Conf. on Database Theory*, pages 83–98, January 2005.

[90] *Netflow Services and Applications*. Cisco Systems. http://www.cisco.com/warp/public/732/netflow.

[91] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 437–448, May 2001.

[92] *The PostgreSQL object-relational database management system*. http://www.postgresql.org.

[93] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of the 1996 Intl. Conf. on Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[94] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, pages 353–364, March 2003.

[95] K. Ross. Conjunctive selection conditions in main memory. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 109–120, June 2002.

[96] K. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pages 447–458, June 1996.

[97] N. Roussopoulos. View indexing in relational databases. *ACM Trans. on Database Systems*, 7(2):258–290, 1982.

[98] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *ACM Trans. on Database Systems*, 16(3):535–563, 1991.

[99] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. of the 1991 Intl. Conf. on Very Large Data Bases*, pages 469–478, September 1991.

[100] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, June 1979.

[101] *Snort: The Open Source Network Intrusion Detection System*. http://www.snort.org.

[102] SQR – A Stream Query Repository. http://www-db.stanford.edu/stream/sqr.

[103] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of the 2004 ACM Symp. on Principles of Database Systems*, pages 263–274, June 2004.

[104] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, pages 324–335, September 2004.

[105] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 9–28, September 2001.

[106] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, page 594, September 1996.

[107] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniak, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, pages 309–320, September 2003.

[108] D. Terry, D. Goldberg, D. Nichols, and B.Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, June 1992.

[109] D. Thomas and R. Motwani. Caching queues in memory buffers. In *Proc. of the 15th Annual ACM-SIAM Symp. on Discrete Algorithms*, January 2004.

[110] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, September 2003.

[111] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowledge and Data Engineering*, 15(3):555–568, May 2003.

[112] T. Urhan and M. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[113] T. Urhan, M. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 130–141, June 1998.

[114] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-join queries over streaming information sources. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 285–296, September 2003.

[115] D. Vista. Integration of incremental view maintenance into query optimizers. In *Proc. of the 1998 Intl. Conf. on Extending Database Technology*, pages 374–388, March 1998.

[116] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, March 1985.

[117] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, California, 1996.

[118] W. Willinger, V. Paxson, R. Riedi, and M. Taqqu. Long-range dependence and data network traffic. In *Long-range Dependence: Theory and Applications, P. Doukhan, G. Oppenheim and M. Taqqu, eds., Birkhauser*, 2002.

[119] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proc. of the 1994 Intl. Conf. on Very Large Data Bases*, pages 522–533, August 1994.

[120] V. Zadorozhny, L. Raschid, M. Vidal, T. Urhan, and L. Bright. Efficient evaluation of queries in a mediator for websources. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 85–96, 2002.

[121] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 431–442, June 2004.