Sliding Window Query Processing over Data Streams

by

Lukasz Golab

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2006

Abstract

Database management systems (DBMSs) have been used successfully in traditional business applications that require persistent data storage and an efficient querying mechanism. Typically, it is assumed that the data are static, unless explicitly modified or deleted by a user or application. Database queries are executed when issued and their answers reflect the current state of the data. However, emerging applications, such as sensor networks, real-time Internet traffic analysis, and on-line financial trading, require support for processing of unbounded data streams. The fundamental assumption of a data stream management system (DSMS) is that new data are generated continually, making it infeasible to store a stream in its entirety. At best, a sliding window of recently arrived data may be maintained, meaning that old data must be removed as time goes on. Furthermore, as the contents of the sliding windows evolve over time, it makes sense for users to ask a query once and receive updated answers over time.

This dissertation begins with the observation that the two fundamental requirements of a DSMS are dealing with transient (time-evolving) rather than static data and answering persistent rather than transient queries. One implication of the first requirement is that data maintenance costs have a significant effect on the performance of a DSMS. Additionally, traditional query processing algorithms must be re-engineered for the sliding window model because queries may need to re-process expired data and "undo" previously generated results. The second requirement suggests that a DSMS may execute a large number of persistent queries at the same time, therefore there exist opportunities for resource sharing among similar queries.

The purpose of this dissertation is to develop solutions for efficient query processing over sliding windows by focusing on these two fundamental properties. In terms of the transient nature of streaming data, this dissertation is based upon the following insight. Although the data keep changing over time as the windows slide forward, the changes are not random; on the contrary, the inputs and outputs of a DSMS exhibit patterns in the way the data are inserted and deleted. It will be shown that the knowledge of these patterns leads to an understanding of the semantics of persistent queries, lower window maintenance costs, as well as novel query processing, query optimization, and concurrency control strategies. In the context of the persistent nature of DSMS queries, the insight behind the proposed solution is that various queries may need to be refreshed at different times, therefore synchronizing the refresh schedules of similar queries creates more opportunities for resource sharing.

Acknowledgements

I would like to thank my supervisor, Prof. M. Tamer Özsu, for his guidance throughout my Ph.D. studies, and the remaining members of my examination committee for their time and interest in this dissertation. They are: Prof. Alejandro López-Ortiz, Prof. Kenneth M. Salem (departmental members), Prof. Paul A. S. Ward (internal-external examiner), and Prof. Jennifer Widom (external examiner).

Financial support for this dissertation has been provided by: Bell Canada, Communications and Information Technology Ontario (CITO), Natural Sciences and Engineering Research Council of Canada (NSERC), Sun Microsystems of Canada, and the University of Waterloo.

Implementation of the prototype system described in Chapter 4 was done by Piyush Prahladka. Implementation of the prototype system used in Chapters 7 and 8 was done by Kumar Gaurav Bijay. Some of the ideas behind the solution presented in Chapter 6 were developed in collaboration with David DeHaan.

Contents

1 Introduction **1**
- 1.1 Data Stream Management Systems . . . 1
- 1.2 Problem Statement . . . 5
- 1.3 Contributions, Scope, and Organization . . . 7

2 Survey of Data Stream Management **11**
- 2.1 Data Models and Query Languages for DSMSs . . . 12
  - 2.1.1 Data Models . . . 12
  - 2.1.2 Semantics of Persistent Queries . . . 13
  - 2.1.3 DSMS Query Algebras and Languages . . . 14
- 2.2 DSMS Query Operators . . . 17
  - 2.2.1 Query Operators over Unbounded Streams . . . 18
  - 2.2.2 Query Operators over Sliding Windows . . . 21
- 2.3 DSMS Query Processing . . . 23
  - 2.3.1 Queuing and Scheduling . . . 23
  - 2.3.2 Determining When Tuples Expire . . . 23
  - 2.3.3 Continuous Query Processing over Sliding Windows . . . 24
  - 2.3.4 Periodic Query Processing Over Sliding Windows . . . 26
- 2.4 DSMS Query Optimization . . . 29
  - 2.4.1 Cost Metrics and Statistics . . . 30
  - 2.4.2 Query Rewriting and Adaptive Query Optimization . . . 30
  - 2.4.3 Load Shedding and Approximation . . . 31
  - 2.4.4 Multi-Query Optimization . . . 32

3 Update-Pattern-Aware Modeling and Processing of Persistent Queries **33**
- 3.1 Introduction . . . 33
- 3.2 Update Patterns of Persistent Queries . . . 35
  - 3.2.1 Classification . . . 35
  - 3.2.2 Discussion . . . 36
- 3.3 Update-Pattern-Aware Semantics of Persistent Queries . . . 38
  - 3.3.1 Defining the Meaning of Relations in Persistent Queries . . . 38

3.3.2 Internal and External Windows 40
3.3.3 Windowing the Output Stream of a Persistent Query 42
3.3.4 Impact of Update Pattern Awareness on Periodic Query Processing 42
3.4 Making a DSMS Query Language Update-Pattern-Aware 44
3.5 Update-Pattern Aware Query Processing and Optimization 45
3.5.1 Update Pattern Propagation in Persistent Query Plans 45
3.5.2 Update-Pattern-Aware Physical Plan Generation 47
3.5.3 Update-Pattern-Aware Query Optimization 49
3.6 Experiments 50
3.6.1 Overview 50
3.6.2 Query 1 52
3.6.3 Query 2 52
3.6.4 Query 3 54
3.6.5 Query 4 54
3.6.6 Lessons Learned 55

4 Indexing Time-Evolving Data with Variable Lifetimes 57
4.1 Introduction 57
4.2 Assumptions and Motivation 58
4.3 Proposed Solution 59
4.3.1 Double Partitioning 59
4.3.2 Cost Analysis 63
4.3.3 Handling Fluctuating Stream Conditions 63
4.4 Experiments 64
4.4.1 Implementation Details 64
4.4.2 Optimal Values for G and E 65
4.4.3 Performance of Doubly Partitioned Indices 66
4.4.4 Fluctuating Stream Conditions 68
4.4.5 Scaling up to Large Index Sizes 68
4.4.6 Lessons Learned 69

5 Multi-Join Processing over Sliding Windows 71
5.1 Introduction 71
5.2 Sliding Window Join Algorithms 72
5.2.1 Eager Evaluation 72
5.2.2 Lazy Evaluation 74
5.2.3 Analysis of Algorithms 77
5.3 Join Ordering 79
5.3.1 Cost Formulae 80
5.3.2 Join Ordering Heuristic 81
5.4 Experimental Results 83

5.4.1 Experimental Setting . . . 83
5.4.2 Validation of Cost Model and Join Ordering Heuristic . . . 83
5.4.3 Relative Performance of Join Algorithms . . . 84
5.4.4 Lessons Learned . . . 85

6 Sliding Window Aggregation: Finding Frequent Items 87
6.1 Introduction . . . 87
6.2 Proposed Solution . . . 89
6.2.1 Algorithm Description . . . 89
6.2.2 Analysis . . . 91
6.3 Experimental Results . . . 92
6.3.1 Experimental Setup . . . 92
6.3.2 Accuracy . . . 92
6.3.3 Space Usage . . . 93
6.3.4 Precision . . . 93
6.3.5 Lessons Learned . . . 93

7 Concurrency Control in Periodic Queries over Sliding Windows 95
7.1 Introduction . . . 95
7.2 Motivation and Assumptions . . . 97
7.2.1 Data and Query Model . . . 97
7.2.2 Motivation for Study of Concurrency Control . . . 99
7.2.3 System Model . . . 100
7.3 Conflict Serializability of Sliding Window Queries . . . 101
7.3.1 Serializability and Serialization Orders . . . 101
7.3.2 Isolation Levels for Sliding Window Queries . . . 103
7.4 Transaction Scheduler Design . . . 104
7.4.1 Producing *LWS* Histories . . . 104
7.4.2 Optimal Ordering of Read Operations . . . 106
7.4.3 Note on Queries Accessing a Materialized Sub-Result . . . 109
7.5 Experiments . . . 110
7.5.1 Implementation Details and Experimental Procedure . . . 110
7.5.2 Percentage of Aborted Transactions . . . 111
7.5.3 Experiments with a Workload of Periodic Queries . . . 112
7.5.4 Experiments with a Mixed Workload of Periodic and One-Time Queries . . . 114
7.5.5 Lessons Learned . . . 115
7.6 Comparison with Related Work . . . 115

8 Multi-Query Optimization of Sliding Window Aggregates 119
8.1 Introduction . . . 119
8.2 Preliminaries . . . 121
8.3 Identifying and Processing Similar Queries . . . 122

8.3.1 Sliding Window Aggregates . . . 122
8.3.2 Aggregates over Joins . . . 123
8.3.3 Aggregates with Selection . . . 125
8.4 Multi-Query Scheduling . . . 127
8.4.1 Semantics of Periodic Query Re-Execution . . . 127
8.4.2 Earliest-Deadline-First Scheduling . . . 129
8.4.3 Scheduling Multiple Queries Together . . . 130
8.4.4 Hybrid Scheduling . . . 132
8.4.5 Additional Sharing during Overload . . . 133
8.5 Experimental Evaluation . . . 135
8.5.1 Setting . . . 135
8.5.2 Results with Low Data Rate . . . 136
8.5.3 Results with High Data Rate . . . 137
8.5.4 Lessons Learned . . . 137
8.6 Comparison with Related Work . . . 137

9 Conclusions and Future Work **141**
9.1 Summary of Contributions . . . 141
9.2 Directions for Future Research . . . 143

A Proof of Theorem 4.1 **163**

List of Tables

2.1 Summary of proposed data stream languages 18

4.1 Classification of previous work on maintenance of time-evolving data 58

5.1 Explanations of symbols used in this chapter 72
5.2 Probing orders given that the global join order is $S_1 \bowtie (S_2 \bowtie S_3)$ 73
5.3 Stream parameters in the initial heuristic example 81
5.4 Sizes of intermediate join results 81
5.5 Stream parameters in the augmented heuristic example 82
5.6 Stream parameters in the example with two fast streams and two slow streams . . 82

8.1 Abbreviations of scheduling techniques 136

List of Figures

1.1 Finding the largest element in a sliding window 3
1.2 Abstract reference architecture of a DSMS 5
1.3 Example query plan for Q_3 6
1.4 Possible execution timelines of two persistent queries 6
1.5 Dissertation coverage in terms of DSMS application requirements 9
1.6 Dissertation coverage in terms of query processing strategies 9

2.1 Examples of persistent query operators over data streams 19
2.2 Query execution using the negative tuple approach 25
2.3 Query execution using the direct approach 25
2.4 Sliding window implemented as a circular array of pointers to sub-windows 26
2.5 Examples of running, interval, and basic interval synopses 27
2.6 Two equivalent illustrations of a wave index 30

3.1 Update patterns of sliding window operators 37
3.2 Output of an aggregation query over four types of windows 41
3.3 Processing a join of internal windows using a materialized view 41
3.4 Drawbacks of periodic query re-evaluation 43
3.5 Two query plans (for the same query) annotated with update patterns 47
3.6 Storing the results of weak non-monotonic subqueries 48
3.7 Illustration of the first three query plans used in the experiments 51
3.8 Processing times of Query 1 using $protocol = ftp$ as the selection predicate 53
3.9 Processing times of Query 1 using *protocol=telnet* as the selection predicate 53
3.10 Processing times of Query 2 with duplicate elimination on the source IP address . 53
3.11 Processing times of Query 2 with dupl. elim. on src. and dest. IP addr. pairs . . . 53
3.12 Space consumption of Query 2 with duplicate elimination on the source IP address 54
3.13 Space consumption of Query 2 with dupl. elim. on src. and dest. IP addr. pairs . 54
3.14 Processing times of Query 3 55
3.15 Processing times of Query 4 55

4.1 Example of a doubly partitioned index, showing an update at time 18 (bottom) . . 60
4.2 Example of a round-robin doubly partitioned index 61

4.3 Structure of an individual sub-index 65
4.4 Relative performance of *Rebuild*4x4, *Rebuild*2x8, and *Rebuild*8x2 66
4.5 Update times of index partitioning techniques given a small window size 66
4.6 Probe times of index partitioning techniques given a small window size 67
4.7 Scan times of index partitioning techniques given a small window size 67
4.8 Effect of data rate fluctuations on index update performance. 68
4.9 Update times of index partitioning techniques given a large data set. 68
4.10 Probe times of index partitioning techniques given a large data set. 69
4.11 Scan times of index partitioning techniques given a large data set. 69

5.1 Join order $S_1 \bowtie (S_2 \bowtie (S_3 \bowtie S_4))$ expressed as a join tree and a series of for-loops . 72
5.2 Validation of join ordering heuristic 84
5.3 Cost of nested-loops algorithms 85
5.4 Cost of hash-based algorithms 85

6.1 A basic window synopsis storing the most popular URL in each sub-window 88
6.2 Example of a single execution of Algorithm 6 91
6.3 Performance analysis of Algorithm 6 94

7.1 Examples of query and window update sequences in a DSMS 96
7.2 Four techniques for computing sliding window sums 98
7.3 Assumed system architecture 100
7.4 Serialization graphs for H_a, H_b, H_c, and H_d 102
7.5 Differences in the results returned by T_{Q1} in H_a, H_b, H_c, and H_d 102
7.6 Update of a materialized sub-result having weak non-monotonic update patterns . 110
7.7 Staleness, response time, and inter-execution time of Q_2 111
7.8 Percentage of read-only transactions aborted by Algorithm 7 112
7.9 Comparison of query staleness for Serial, WS, LWS, and TTU 112
7.10 Comparison of query inter-execution times for Serial, WS, LWS, and TTU 113
7.11 Comparison of throughput of read-only transactions for LWS and "Others" 113
7.12 Comparison of one-time query response times for Serial, WS, LWS, and TTU . . . 114
7.13 Comparison of one-time query staleness for Serial, WS, LWS, and TTU 114
7.14 Comparison of periodic query staleness for Serial, WS, LWS, and TTU 115
7.15 Relationship between CPS, WS, and LWS 117

8.1 Two possible ways to schedule queries Q_1 and Q_2 120
8.2 Assumed query processing architecture 121
8.3 Examples of RJ and BIJ synopses 124
8.4 Example of a generalized BIJ synopsis 125
8.5 Semantics of periodic query re-execution 128
8.6 Execution schedules of queries in group G_1 using various scheduling techniques . . 132
8.7 Throughput measurements using a data rate of 1500 tuples per second 136

8.8 Average latency measurements using a data rate of 1500 tuples per second 136
8.9 Throughput measurements using a data rate of 3000 tuples per second 137
8.10 Average latency measurements using a data rate of 3000 tuples per second 137

A.1 Example of a doubly partitioned index, showing an update at time 18 (bottom) . . 164
A.2 Example of a round-robin doubly partitioned index 165

Chapter 1

Introduction

1.1 Data Stream Management Systems

A traditional relational database stores a collection of tables, which are inherently unordered and therefore viewed as sets. Data records are relatively static, and are assumed to be valid until explicitly modified or deleted by a user or application. Queries, typically assumed to occur more frequently than data modifications, are executed when posed and their answers reflect the current state of the database. The goal of a database management system (DBMS) is to provide persistent, consistent, and recoverable storage, as well as an efficient query answering mechanism.

The relational model has fulfilled the needs of traditional business applications, as evidenced by the commercial success of relational DBMSs. However, the requirements of a number of emerging applications do not fit the above description. One particularly interesting change is that data may be generated in real time, taking the form of an unbounded sequence (stream) of values. This shift is being instigated by the following trends and applications.

1. Networks of sensors with wireless communication capabilities are becoming increasingly ubiquitous [133, 231]. Sensor networks are used for environmental and geophysical monitoring [266], road traffic monitoring [13, 180], location tracking and surveillance [2, 259], and inventory and supply-chain analysis [92, 114, 259]. The measurements produced by sensors (e.g., temperature readings) may be modeled as continuous data streams.
2. The World Wide Web offers a multitude of on-line data feeds, such as news, sports, and financial tickers [52, 163]. Typically, a third-party service such as Traderbot[1] merges data from multiple sources and publishes a set of output streams to which users may subscribe. Example queries include moving averages of recent stock prices and finding correlations between the prices of several stocks.
3. An overwhelming amount of transaction log data is generated from telephone call records, point-of-sale purchase (e.g., credit card) transactions, and Web server logs [65, 105]. On-line

[1] www.traderbot.com

analysis of transaction logs can identify interesting customer spending patterns or possible credit card fraud.

4. In the networking community, there has been a great deal of recent interest in using a data management system for on-line monitoring and analysis of network traffic [66, 233]. Specific goals include tracking bandwidth usage statistics for the purposes of traffic engineering, routing system analysis and customer billing, as well as detecting suspicious activity such as equipment malfunctions or denial-of-service attacks [145]. In this context, a data stream is composed of IP packet headers.

A fundamental assumption of the data stream model is that new data are generated continually and in fixed order, though the arrival rates may vary across applications from millions of items per second (e.g., Internet traffic monitoring) down to several items per hour (e.g., temperature and humidity readings from a weather monitoring station). The ordering of streaming data may be implicit (by arrival time at the processing site) or explicit (by generation time, as indicated by a *timestamp* appended to each data item by the source). As a result of these assumptions, Data Stream Management Systems (DSMSs) face the following novel requirements.

1. Much of the computation performed by a DSMS is push-based, or data-driven. That is, newly arrived stream items are continually (or periodically) pushed into the system for processing. On the other hand, a DBMS employs a mostly pull-based, or query-driven computation model, where processing is initiated when a query is posed.
2. As a consequence of the above, DSMS queries are *persistent*[2] in that they are issued once, but remain active in the system for a possibly long period of time. This means that a stream of updated results must be produced as time goes on. In contrast, a DBMS deals with one-time queries (issued once and then "forgotten"), whose results are computed over the current state of the database.
3. The system conditions may not be stable during the *lifetime* of a persistent query. For example, the stream arrival rates may fluctuate and the query workload may change.
4. A data stream is assumed to have unbounded, or at least unknown, length. From the system's point of view, it is infeasible to store an entire stream in a DSMS. From the user's point of view, recently arrived data are likely to be more accurate or useful.
5. New data models, query semantics and query languages are needed for DSMSs in order to reflect the facts that streams are ordered and queries are persistent.

The first three requirements guide the design of a DSMS query engine. While superficially similar to relational query plans represented as trees of operators, persistent query plans make use of buffers, queues [2, 180, 190], and sophisticated scheduling algorithms [20, 46] in order to handle continuously incoming data. Moreover, a DSMS query plan must not include *blocking* operators

[2]Persistent queries are also referred to in the literature as continuous, long-running, or standing queries. This dissertation reserves the term *continuous query* for a persistent query that produces new answers continuously. In contrast, a *periodic query* is defined as a persistent query that returns new answers periodically.

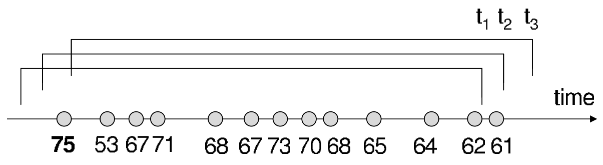


Figure 1.1: Finding the largest element in a sliding window

that consume the entire input before producing any results [157]. The third requirement implies that a DSMS must adapt to changes in system conditions, possibly by re-optimizing persistent queries over time [25] or initiating some form of load shedding [22, 204, 236, 238] (e.g., dropping a fraction of incoming data during periods of overload). The fourth requirement suggests that old data must be removed or at least archived for off-line processing, giving rise to various window models. In the simplest case, a fixed window model periodically clears the accumulated data. That is, a stream is divided into non-overlapping partitions and data are kept only for that part of a stream which falls within the current partition. A major disadvantage of this model is that the window size varies—the window begins with size zero and grows to some specified size, at which point it is reset back to size zero. In contrast, the sliding window model expires old items as new items arrive. Two common types of sliding windows are *count-based windows*, which store the N newest items, and *time-based windows*, which store only those items which have been generated or have arrived in the last T time units. Finally, in response to the fourth and fifth requirements, persistent queries over sliding windows must be supported by a DSMS.

To illustrate the challenges of maintaining and querying sliding windows, consider query Q_1, which may be written in CQL[3] as `SELECT MAX(a) FROM S`, where S is a stream of items having a numerical attribute a. At any time, Q_1 maintains the largest a-value over all the items seen thus far. To achieve this, Q_1 requires constant space and processing time per data item—it stores the current maximum value and compares it against the a-values of the incoming items on-the-fly. Now consider Q_2, which, at all times, maintains the largest a-value over a sliding window of one minute. In CQL, Q_2 may be written as `SELECT MAX(a) FROM S [RANGE 1 min]`. An example of finding the largest value in a sliding window is illustrated in Figure 1.1, with stream items and their a-values shown on a time axis. At time t_1, the window spans all but the youngest item (with $a = 61$), which has not yet arrived. The maximum at this time is 75. At time t_2, the youngest item arrives, but its value is smaller than the maximum, so the answer does not change. However, at time t_3, the window slides past the oldest item with $a = 75$ and a new maximum must be found. In the worst case, Q_2 needs to store and examine every data item inside the window in order to update the maximum (this occurs when the stream is sorted in decreasing order on a). Note that the sliding window version of `MAX` must examine newly arrived items and also react whenever an item expires from the window.

At this point, it is worth noting that instead of designing a DSMS, one may attempt to

[3]CQL is a stream query language with SQL-like syntax proposed in [12]; it will be described in more detail in Section 2.1.

employ existing data management technologies for data stream processing [231]. One possibility is to use a traditional DBMS augmented with an application layer that simulates sliding windows, persistent queries, and data-driven processing by continually inserting new data, removing expired data [210], and maintaining query results. Existing techniques for incremental maintenance of materialized views [38] may be helpful in order to avoid repeated re-computation of the entire result of a persistent query (some of the early work on persistent queries has focused on this fact [136, 239]). Nevertheless, the application-based approach is significantly less efficient than a DSMS, largely due to the semantic gap between the DBMS and the DSMS-like application [13].

Another alternative to designing a DSMS involves exploiting relevant features of the DBMS engine, among them triggers [129, 198, 257] and novel data types such as arrays [162], lists [176, 197, 232], sequences [202, 214, 215, 216], time series [208], and temporal data [268]. However, triggers are not sufficiently scalable and expressible for data stream applications[4] [13, 21, 231]. Furthermore, while the sequence, time series, and temporal extensions can potentially handle ordering and sliding windows, they are based upon the query-driven processing model and assume that all the data are already in the database.

The novel requirements of data stream processing have led to a number of recent DSMS proposals. Academic systems include Aurora [2, 30], Borealis [1], CAPE [206], Hi-Fi [92, 205], Nile [8, 127], PIPES [44, 153], PSoup [49], STREAM [12], and TelegraphCQ [48]. Additionally, Gigascope [62, 66], which is a DSMS for network monitoring, is used at AT&T. Moreover, the Aurora and Borealis academic systems have given rise to StreamBase Systems Inc.[5], whose product is targeted at financial trading applications, telecommunications systems monitoring, and government and military surveillance tasks.

This dissertation assumes DSMS-based processing of data streams using an abstract system architecture shown in Figure 1.2. Streaming inputs arrive for processing in real time. An input monitor regulates the input rates, perhaps by dropping some data if the system is unable to keep up. Conceptually, data are stored in two partitions: working storage (containing sliding windows) and local storage for metadata stored as relational tables (e.g., physical location of each data source or the schema of the stream). Users may update tables directly, but the working storage is maintained automatically by expiring items that fall out of their windows[6]. Persistent queries are registered in the query repository and possibly grouped for shared processing, though one-time queries over the current state of the inputs may also be issued. The query processor communicates with the input monitor and may re-optimize query plans in response to changes in the workload and the input rates. Results are streamed to users or materialized in the system. Users may then refine their queries based upon the latest results, or ask ad-hoc one-time queries in response to changes in the answers of persistent queries. The focus of this dissertation is on

[4]Publish-subscribe systems have been designed to extend the scalability of triggers, but the emphasis is on efficient monitoring of a very large number of simple conditions [87]. In contrast, the workload of a DSMS typically involves a somewhat smaller number of more complicated queries. However, there has been some recent work on extending the expressive power of triggers [242] and publish-subscribe systems [73, 259], and many novel solutions used therein are likely to be applicable to DSMSs.

[5]www.streambase.com

[6]Expired data may be discarded or archived for off-line processing.

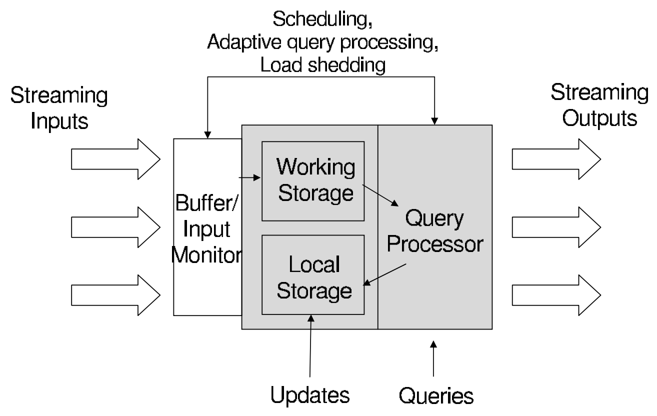


Figure 1.2: Abstract reference architecture of a DSMS

the shaded components, namely DSMS storage and query processing.

1.2 Problem Statement

The fundamental differences between a DSMS and a DBMS are the nature of the data and the nature of the queries. Specifically, a DBMS handles transient queries over persistent data, whereas a DSMS processes persistent queries over transient data. There are several important consequences of these differences. First, one of the implications of the time-evolving nature of data streams is that data maintenance costs have a significant effect on the performance of a DSMS. As a window slides forward, new data items must be inserted and old items evicted (or archived). If implemented without care, a DSMS may spend most of its time on window maintenance, leaving little time for query answering. Additionally, traditional relational operators must be re-engineered for the sliding window model because expired data items may need to be reprocessed in order to "undo" previously generated results. On the other hand, the persistent nature of queries over streams means that a DSMS may execute a large number of queries at the same time. As a result, multi-query optimization is a suitable approach for ensuring scalability. The purpose of this research is to develop solutions for processing persistent queries over sliding windows by focusing on these two differences.

In terms of the transient nature of streaming data, the insight offered in this dissertation is that although the data stored in a DSMS keep changing over time as the windows move forward, expirations from windows and output streams of queries are not random. On the contrary, the inputs and outputs of persistent queries exhibit patterns in the way the data are inserted and deleted. It will be shown that the notion of *update pattern awareness*, which aims at uncovering and exploiting these patterns, leads to an understanding of the semantics of persistent queries, lower window maintenance costs, novel query processing and optimization strategies, and new concurrency control models. For example, recall Figure 1.1 and observe that the next tuple to

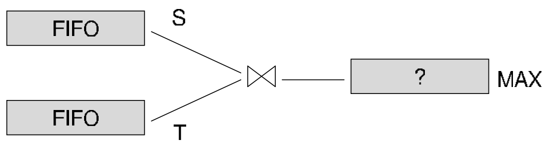


Figure 1.3: Example query plan for Q_3

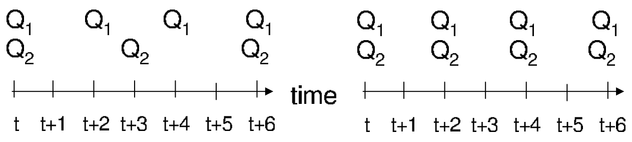


Figure 1.4: Possible execution timelines of two persistent queries

expire is always the oldest one. Hence, the expiration order from a sliding window is equivalent to the insertion order into the window, and therefore a sliding window is analogous to a first-in-first-out (FIFO) queue. In other words, FIFO-expiration is an example of an update pattern. One way of exploiting this pattern by query Q_2 is to implement its state buffer as a FIFO queue. This way, new tuples can be appended to the front of the queue and expired tuples are guaranteed to be found at the tail. However, consider Q_3 defined as follows.

```
SELECT MAX(a)
FROM S [RANGE 1 min], T [RANGE 1 min]
WHERE S.b = T.b
```

Q_3 maintains the largest a-value over an equi-join of two sliding windows (on the b-attribute). As illustrated in Figure 1.3, a FIFO queue may not be appropriate for storing the state needed by the `MAX` operator in Q_3. This is because the output of a sliding window join does not satisfy the FIFO property. For example, a newly arrived tuple may join with several tuples from the other window, of which some have arrived recently and some are about to expire.

In the context of the persistent nature of DSMS queries, the focus of this research is on periodic re-evaluation of persistent queries, as discussed in, e.g., [2, 45, 48, 52, 156, 167, 222, 276]. This may be done for performance reasons; additionally, users may find it easier to deal with periodic output rather than a continuous output stream [16, 49]. Suppose that the DSMS executes two queries, call them Q_1 and Q_2, with Q_1's answers refreshed every two time units and Q_2's answers refreshed every three time units. Furthermore, suppose that Q_1 and Q_2 can share computation. For example, the two queries could be identical except for the length of the window that they reference, with the query over the longer window re-using the computation used to refresh the query over the shorter window. Figure 1.4 illustrates two possible execution sequences of Q_1 and Q_2. On the left, computation sharing is exploited only once every six time units when both queries are due for a refresh. On the right, the refresh times of Q_2 are synchronized with those of Q_1. In this case, more computation is shared, but the savings in processing time may be defeated by the work expended on updating the answer of Q_2 more often than necessary. The novel problem in this context is to schedule the refresh times of a set of possibly similar queries in such a way as to minimize the total processing time.

1.3 Contributions, Scope, and Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents a survey of data stream management. Chapter 3 deals with the first fundamental property of a DSMS—handling time-evolving data—by proposing the idea of update pattern awareness. The particular contributions of Chapter 3 are as follows.

- The update patterns of persistent query operators and plans are classified into four types. The classification is then used to formulate the semantics of persistent queries over streams, sliding windows, and relations. Semantic issues that are clarified with the help of update pattern awareness include the difference between arbitrary updates of relational tables and insertions and deletions caused by the movement of the sliding windows, the relationship between sliding windows and monotonic queries, the difference between a window on the input stream and a window on the output stream, and the impact of periodic query processing on the completeness of the result.
- Incorporation of update-pattern-aware semantics into CQL is outlined.
- An update-pattern-aware query processing framework is introduced, where implementations of operators and intermediate state structures depend upon the update patterns of the inputs and outputs. The idea is to choose a query plan with the simplest update patterns in order to minimize state maintenance costs and simplify operator implementation. Update-pattern-aware query plans are shown to routinely outperform the existing techniques from the data stream literature by an order of magnitude.

Chapter 3 assumes on-line processing over memory-resident sliding windows. Chapter 4 extends update pattern awareness to accommodate stream warehousing for off-line mining and analysis. An update-pattern-aware index is proposed for secondary storage of time-evolving data, where each record has a specified lifetime in the system. The index is updated periodically by inserting a batch of new data and removing expired data. However, updates may incur high I/O costs if the entire data set must be loaded into memory to determine which items have expired and should be deleted. The insight behind the solution described in this dissertation is to partition the data such that only selected partitions are affected by updates. This technique is shown to perform updates over twice as fast as existing sliding window indices.

Chapters 5 and 6 take a closer look at two important sliding window operators: join and top-k aggregation. Chapter 5 contains two contributions. First, pipelined multi-window join algorithms are proposed for continual and periodic join evaluation, as well as continual and periodic expiration from the hash tables (using update-pattern-aware data structures that cluster the hash buckets according to expiration times of the data items). Second, a join ordering heuristic is developed by considering the stream arrival rates, window lengths, and the expected number of join results produced. Chapter 6 presents an algorithm for incrementally maintaining k most frequently occurring item types in a sliding window, with a focus on Internet traffic analysis.

After discussing query evaluation and window maintenance, this dissertation examines concurrency issues arising from simultaneous execution of queries and window-slides. In a DBMS, concurrency control manages the way in which many users access the same data. For example, reading a record in a table may be done at the same time by many users, but it does not make sense to let more than one person change the same record at the same time. In a DSMS, insertions and deletions can be thought of as happening automatically as the windows slide forward. Concurrency control is required because a window may slide forward while being accessed by a query. Chapter 7 shows that the traditional notion of conflict serializability [35] is insufficient in the DSMS environment and proposes stronger isolation levels that restrict the allowed serialization orders. An update-pattern-aware transaction scheduler is also designed and proven to minimize the number of aborted transactions while ensuring the desired isolation level. The insight behind the solution follows from the FIFO property of sliding windows—given that new data are added to the beginning and old data are removed from the tail, the "middle" of the window can be accessed by concurrent read-only transactions (queries).

Chapter 8 addresses the second fundamental property of a DSMS: handling a large number of persistent queries. As motivated in Figure 1.4, one of the novel challenges is to allow resource sharing among similar queries, even if they are re-evaluated with different frequencies and therefore get scheduled at different times. The particular contribution of Chapter 8 are as follows.

- An extensible set of rules for sharing state and/or computation is proposed for periodic queries with aggregation (e.g., aggregates over different window lengths can share a data structure for storing state).
- Persistent queries are modeled as periodic tasks that must be scheduled according to their deadlines and a query scheduler is designed on the basis of the traditional earliest-deadline-first (EDF) algorithm [229]. With the help of a cost model and according to a set of subexpression matching rules, the scheduler synchronizes the refresh times of similar queries in order to minimize the total query processing time and maximize system throughput.

Finally, Chapter 9 concludes this dissertation with a summary of contributions and suggestions for future work.

The scope of this dissertation may be illustrated in several ways. In terms of targeted applications, three points on a speed-versus-state-versus-query-complexity spectrum are labeled in Figure 1.5. On the far left are applications that process massive amounts of raw data in real-time, e.g., live network monitoring. Gigascope and, to some extent, Aurora are examples of DSMSs aimed at these types of applications. Very fast processing is needed in order to keep up with the data arrival rate. As a result, the amount of state available for queries may be restricted to a few variables, possibly stored in very fast CPU registers. Due to these extreme conditions, queries are typically stateless filters and simple aggregates over fixed windows. Furthermore, answers may be approximate, e.g., obtained by sampling. The middle point of the spectrum represents applications that require on-line processing, but whose data rates are tractable enough to allow complex queries over memory-resident sliding windows. Representative examples include analysis of pre-processed network data, financial tickers, and sensor networks; representative queries include joins and complex aggregation. In some cases, the entire window does not fit in memory and must be summarized in limited space, with the consequence that queries accessing the summary

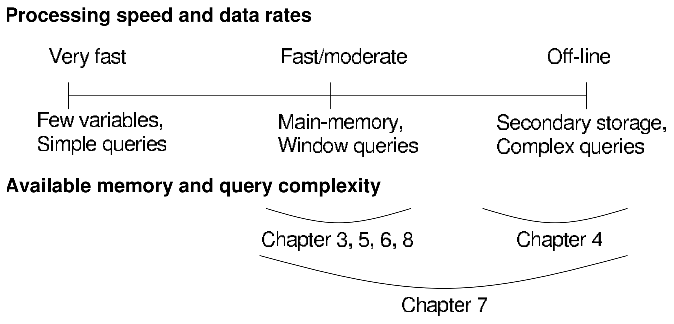

Processing speed and data rates
Very fast
Fast/moderate
Off-line
Few variables, Simple queries
Main-memory, Window queries
Secondary storage, Complex queries
Available memory and query complexity
Chapter 3, 5, 6, 8
Chapter 4
Chapter 7

Figure 1.5: Dissertation coverage in terms of DSMS application requirements

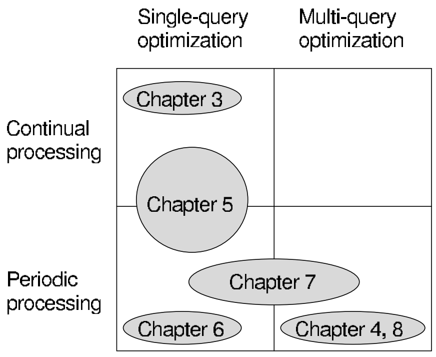


Figure 1.6: Dissertation coverage in terms of query processing strategies

can only return approximate answers. Nile and STREAM are two examples of all-purpose DSMSs aimed at applications in the middle of the spectrum. Finally, the right point describes applications where a large amount of streaming data is archived on disk and updated periodically, with complex analytical queries executed off-line over a sliding window residing in secondary storage[7]. Given the absence of real-time processing requirements, complex queries, including sophisticated data mining and pattern analysis, are possible. In particular, algorithms are allowed to make multiple passes over the data. As shown in Figure 1.5, this dissertation spans the middle and right endpoints of the spectrum.

The quad-chart in Figure 1.6 organizes the contributions of this dissertation according to the query processing strategies. Chapter 3 introduces update pattern awareness in the context of a single query using data-driven (continual) processing; however, the impact of update pattern awareness on the semantics of periodically refreshed queries will be explained in Section 3.3.4. Chapter 5 shows that some operators, namely the sliding window join, can process new tuples continually or periodically, whereas Chapters 6 and 8 concentrate on periodic processing of complex aggregates over sliding windows. Multi-query optimization and scalability with respect to the query workload are covered in Chapter 8, though the indexing techniques from Chapter 4 and concurrency control solutions from Chapter 7 have also been designed with multi-query processing in mind.

Finally, in the context of centralized versus distributed DSMS architectures, the scope of this dissertation is limited to the centralized scenario.

[7]Applications occupying the right point of the spectrum are similar to existing data warehousing and on-line analytical processing (OLAP) applications, with an emphasis on the maintenance of disk-resident sliding windows.

Chapter 2

Survey of Data Stream Management

This chapter reviews recent work on data stream processing. Related surveys include an earlier and shorter version of this chapter [110], a discussion of issues in data stream processing in the context of the STREAM system [21], and a review of query processing over relational and XML streams [152]. In keeping with the focus of this dissertation, the emphasis is on centralized processing of relational-like queries over sliding windows. For brevity, a discussion of the following related topics has been omitted.

- Early work on stream and workflow processing in the programming languages community. See [230] for a survey.
- On-line data mining and time series analysis over streams and sliding windows. See [88, 93, 97] for an overview and [43, 47, 169, 177] for recent system demonstrations.
- Application-specific DSMS issues, such as query processing in sensor networks. See, for example, the following representative papers [78, 118, 133, 139, 181, 182, 184, 193, 223, 224, 226, 243, 266].
- Distributed stream processing. See, for example,
  - recent work on the Borealis [1, 6, 31, 32, 54, 134, 237, 263, 264], D-CAPE [172] and SPC [138] distributed DSMSs,
  - extensions of the TelegraphCQ system for parallel dataflow processing [218, 219],
  - recent work on distributed query processing over streams [98, 217, 269, 273],
  - recent work on distributed tracking of approximate stream statistics [53, 60, 61, 64, 67, 74, 103, 104, 137, 148, 158, 185, 194, 221].

The remainder of this chapter surveys DSMS data models and query languages (Section 2.1), operator implementation (Section 2.2), query processing (Section 2.3), and query optimization (Section 2.4).

2.1 Data Models and Query Languages for DSMSs

2.1.1 Data Models

A data stream is an append-only[1] sequence of timestamped items[2] that arrive in some order [119]. Since items may arrive in bursts, a stream may instead be modeled as a sequence of sets (or bags) of elements [245], with each set storing elements that have arrived during the same unit of time (no order is specified among tuples that have arrived at the same time). In relation-based stream models (e.g., STREAM [12]), individual items take the form of relational tuples such that all tuples arriving on the same stream have the same schema. In object-based models (e.g., COUGAR [39] and Tribeca [233]), sources and item types may be instantiations of (hierarchical) data types with associated methods. As introduced in Section 1.1, stream items may contain explicit source-assigned timestamps or implicit timestamps assigned by the DSMS upon arrival. In either case, the timestamp attribute may or may not be part of the stream schema, and therefore may or may not be visible to users.

Stream items may arrive out of order (if explicit timestamps are used) and/or in pre-processed form. For instance, rather than propagating the header of each IP packet, one value (or several partially pre-aggregated values) may be produced to summarize the length of a connection between two IP addresses and the number of bytes transmitted [66, 271]. This gives rise to the following list of possible models [105]:

1. *Unordered cash register*: Individual items from various domains arrive in no particular order and without any pre-processing. This is the most general model.
2. *Ordered cash register*: Individual items from various domains are not pre-processed but arrive in some known order, e.g., timestamp order.
3. *Unordered aggregate*: Individual items from the same domain are pre-processed and only one item per domain arrives in no particular order, e.g., one packet per TCP connection.
4. *Ordered aggregate*: Individual items from the same domain are pre-processed and one item per domain arrives in some known order, e.g., one packet per TCP connection in increasing order of the connection end-times.

As discussed in Section 1.1, unbounded streams cannot be stored locally in a DSMS, and only a recent excerpt of a stream is usually of interest at any given time. In general, this may be accomplished using a *time-decay model* [57, 58, 85], also referred to as an *amnesic* [196] or *fading* [5] model. Time-decay models discount each item in the stream by a scaling factor that is non-decreasing with time. Exponential and polynomial decay are two examples, as are window models where items within the window are given full consideration and items outside the window are ignored. Windows may be classified according the the following criteria.

[1]While the majority of DSMS research assumes that data streams are append-only, there has been some recent work on processing *revision tuples*, which are understood to replace previously reported (presumably erroneous) data [207].

[2]Alternatively, from the point of view of publish-subscribe systems, a data stream may be thought of as a sequence of events that are being reported continually [259].

1. *Direction of movement of the endpoints:* Two fixed endpoints define a *fixed window*, two sliding endpoints (either forward or backward, replacing old items as new items arrive) define a *sliding window*, and one fixed endpoint and one moving endpoint (forward or backward) define a *landmark window*. There are a total of nine possibilities as each of the two endpoints could be fixed, moving forward, or moving backward.
2. *Definition of window size:* Logical, or *time-based* windows are defined in terms of a time interval, whereas physical, (also known as *count-based* or *tuple-based*) windows are defined in terms of the number of tuples. Moreover, *partitioned windows* may be defined by splitting a sliding window into groups and defining a separate count-based window on each group [12]. The most general type is a *predicate window*, in which an arbitrary predicate specifies the contents of the window; e.g., all the packets from TCP connections that are currently open [100]. A predicate window is analogous to a materialized view.
3. *Windows within windows:* In the *elastic window model*, the maximum window size is given, but queries may need to run over any smaller window within the boundaries of the maximum window [277]. In the *n-of-N window model*, the maximum window size is N tuples or time units, but any smaller window of size n and with one endpoint in common with the larger window is also of interest [170].
4. *Window update interval:* Eager updating advances the window upon arrival of each new tuple or expiration of an old tuple, but batch processing (lazy updating) induces a *jumping window*. Note that a count-based window may be updated periodically and a time-based window may be updated after some number of new tuples have arrived; these are referred to as *mixed jumping windows* [179]. If the update interval is larger than the window size, then the result is a series of non-overlapping *tumbling windows* [2].

As a consequence of the unbounded nature of data streams, DSMS data models may include some notion of change or drift in the underlying distribution that is assumed to generate the attribute values of stream items (for details, see, e.g., [34, 69, 274]). Additionally, it has been observed that in many practical scenarios, the stream arrival rates and distributions of values tend to be bursty or skewed [149, 151, 161, 200, 277].

2.1.2 Semantics of Persistent Queries

Let $Q(\tau)$ be the answer of a persistent query Q at time τ. Q is monotonic if $Q(\tau) \subseteq Q(\tau')$ for all $\tau \leq \tau'$[3]. For example, queries containing simple selection predicates are monotonic over an append-only stream, as are joins of append-only streams. To see this, note that when a new tuple arrives, it either satisfies the (selection or join) predicate or it does not and the satisfaction condition does not change over time. If Q is monotonic, then its semantics may be defined as follows (assuming that time is represented as a set of natural numbers).

[3]In [157], it is proven that a persistent query is monotonic (with respect to its output sequence) if and only if it is non-blocking, i.e., if it does not need to wait until the end-of-output marker before producing results.

$$Q(\tau) = \bigcup_{t=1}^{\tau}(Q(t) - Q(t-1)) \cup Q(0)$$

That is, it suffices to re-evaluate the query over newly arrived items and append qualifying tuples to the result [12]. Consequently, the answer of a monotonic persistent query is a continuous, append-only stream of results. Optionally, the output may be updated periodically by appending a batch of new results.

Non-monotonic queries may produce results that cease to be valid as new data are added and existing data changed (or deleted). In the context of one-time queries over the current state of a DBMS, non-monotonic queries (e.g., negation) are blocking as they must read the entire data set before returning any results. In the context of DSMSs and persistent queries, negation is non-monotonic, even if issued over an append-only stream (e.g., "select from a stream of e-mail messages all those messages that have not yet received a reply"). The semantics of non-monotonic persistent queries are as follows [125, 154].

Definition 2.1 *At any time τ, $Q(\tau)$ must be equal to the output of a corresponding one-time relational query whose inputs are the current states of the streams, sliding windows, and relations referenced in Q. If results are updated periodically, then $Q(\tau)$ must reflect the state of the inputs as of the most recent update.*

Conceptually, there are three ways of representing the answer of a non-monotonic persistent query. First, the query could be re-executed from scratch and return a complete answer at every time instant (or periodically). Second, the result may be a materialized view that incurs insertions, deletions, and updates over time. Third, the answer may be a continuous stream that contains new results as well as *negative tuples* [125] that correspond to deletions from the result set.

Note that according to Definition 2.1, queries over sliding windows are non-monotonic; when an input tuple t expires from its window, all the results that were generated using t must be removed from the answer set (e.g., by way of negative tuples). However, [157] presents a different definition of monotonicity, which relates to non-blocking computation and is independent of the expiration mechanism. According to this definition, queries that are non-blocking (and therefore monotonic) over unbounded streams are monotonic over sliding windows. These two definitions will be unified in Chapter 3.

2.1.3 DSMS Query Algebras and Languages

Three querying paradigms for streaming data have been proposed in the literature. Declarative languages have SQL-like syntax, but stream-specific semantics, as described above. Similarly, object-based languages resemble SQL in syntax, but employ DSMS-specific constructs and semantics, and may include support for streaming abstract data types (ADTs) and associated methods. Finally, procedural systems construct queries by defining data flow through various operators.

Declarative Languages

The proposed declarative languages are CQL [12, 15], GSQL [66], and StreaQuel [48].

The Continuous Query Language (CQL) is used in the STREAM DSMS and includes three types of operators: relation-to-relation (corresponding to standard relational algebraic operators), stream-to-relation (sliding windows), and relation-to-stream. Conceptually, unbounded streams are converted to relations by way of sliding windows, the query is computed over the current state of the sliding windows as if it were a traditional SQL query, and the output is converted back to a stream. There are three relation-to-stream operators—`Istream`, `Dstream`, and `Rstream`—which specify the nature of the output. The `Istream` operator returns a stream of all those tuples which exist in a relation at the current time, but did not exist at the current time minus one. Thus, `Istream` suggests incremental evaluation of monotonic queries. `Dstream` returns a stream of tuples that existed in the given relation in the previous time unit, but not at the current time. Conceptually, `Dstream` is analogous to generating negative tuples for non-monotonic queries. Finally, the `Rstream` operator streams the contents of the entire output relation at the current time and corresponds to generating the complete answer of a non-monotonic query. The `Rstream` operator may also be used in periodic query evaluation (recall lazy updating and jumping windows from Section 2.1) to produce an output stream consisting of a sequence of relations, each corresponding to the answer at a different point in time. An example query, computing a join of two time-based windows of size one minute each, is shown below (the `RANGE` keyword following the name of the input stream specifies a time-based sliding window on that stream, whereas the `ROWS` keyword may be used to define count-based sliding windows).

```
SELECT Rstream(*)
FROM S1 [RANGE 1 min], S2 [RANGE 1 min]
WHERE S1.a = S2.a
```

GSQL is used in Gigascope, a stream database for network monitoring and analysis. The input and output of each operator is a stream for reasons of composability. Each stream is required to have an ordering attribute, such as timestamp or packet sequence number. GSQL includes a subset of the operators found in SQL, namely selection, aggregation with group-by, and join of two streams, whose predicate must include ordering attributes that form a join window. The *stream merge* operator, not found in standard SQL, is included and works as an order-preserving union of ordered streams. This operator is useful in network traffic analysis, where flows from multiple links need to be merged for analysis. Only landmark windows are supported directly, but sliding windows may be simulated via user-defined functions.

StreaQuel is used in the TelegraphCQ system and is noteworthy for its windowing capabilities. Each query, expressed in SQL syntax and constructed from SQL's set of relational operators, is followed by a for-loop construct with a variable `t` that iterates over time. The loop contains a `WindowIs` statement that specifies the type and size of the window. Let `S` be a stream and let `ST` be the start time of a query. To specify a sliding window over `S` with size five that should run for fifty time units, the following for-loop may be appended to the query.

```
for(t=ST; t<ST+50; t++)
   WindowIs(S, t-4, t)
```

Changing to a landmark window can be done by replacing `t-4` with some constant in the `WindowIs` statement. Changing the for-loop increment condition to `t=t+5` would cause the query to re-execute every five time units. The output of a StreaQuel query consists of a time sequence of sets, each set corresponding to the answer set of the query at that time (cf. `Rstream`).

Object-Based Languages

One approach to object-oriented stream modeling is to classify stream contents according to a type hierarchy. This method is used in the Tribeca network monitoring system, which implements Internet protocol layers as hierarchical data types [233]. The query language used in Tribeca has SQL-like syntax, but accepts a single stream as input, and returns one or more output streams. Supported operators are limited to projection, selection, aggregation over the entire input stream or over a sliding window, multiplex and demultiplex (corresponding to union and group-by respectively, except that different sets of operators may be applied on each of the demultiplexed sub-streams), as well as a join of the input stream with a fixed window.

Another object-based possibility is to model the sources as ADTs, as in the COUGAR system for managing sensor data [39]. Each type of sensor is modeled by an ADT, whose interface consists of the supported signal processing methods. The proposed query language has SQL-like syntax and also includes a `$every()` clause that indicates the query re-execution frequency. However, few details on the language are available in the published literature and therefore it is not included in Table 2.1. For a simple example, a query that runs every sixty seconds and returns temperature readings from all sensors on the third floor of a building may be specified as follows.

```
SELECT R.s.getTemperature()
FROM R
WHERE R.floor = 3 AND $every(60)
```

Procedural Languages

An alternative to declarative query languages is to let the user specify how the data should flow through the system. In the Aurora DSMS [2], users construct query plans via a graphical interface by arranging boxes, corresponding to query operators, and joining them with directed arcs to specify data flow, though the system may later re-arrange, add, or remove operators in the optimization phase. SQuAl is the boxes-and-arrows query language used in Aurora, which accepts streams as inputs and returns streams as output (however, static data sets may be incorporated into query plans via *connection points* [2]). There are a total of seven operators in the SQuAl algebra, four of them order-sensitive. The three order-insensitive operators are projection, union, and `map`, the last applying an arbitrary function to each of the tuples in the stream or a window thereof. The other four operators require an order specification, which includes the ordered field and a slack parameter. The latter defines the maximum disorder in the stream, e.g., a slack of two means that each tuple in the stream is either in sorted order, or at most two positions or two time units away from being in sorted order. The four order-sensitive operators are buffered sort (which takes an almost-sorted stream and the slack parameter, and outputs the stream in sorted order), windowed aggregates (in which the user can specify how often to advance the window and re-evaluate the aggregate), binary band join (which joins tuples whose timestamps are at most t units apart), and resample (which generates missing stream values by interpolation, e.g., given tuples with timestamps 1 and 3, a new tuple with timestamp 2 can be generated with an attribute value that is an average[4] of the other two tuples' values).

Summary of DSMS Query Languages

A summary of the proposed DSMS query languages is provided in Table 2.1 with respect to the allowed inputs and outputs (streams and/or relations), novel operators, supported window types (fixed, landmark or sliding), and supported query re-execution frequency (continuous and/or periodic). With the exception of SQuAl, the surface syntax of DSMS query languages is similar to SQL, but their semantics are considerably different. CQL allows the widest range of semantics with its relation-to-stream operators; note that CQL re-uses the semantics of SQL during its relation-to-relation phase and incorporates streaming semantics in the stream-to-relation and relation-to-stream components. On the other hand, GSQL, SQuAL, and Tribeca only allow streaming output, whereas StreaQuel continually (or periodically) outputs the entire answer set. In terms of expressive power, CQL closely mirrors SQL as CQL's core set of operators is identical to that of SQL. Additionally, StreaQuel can express a wider range of windows than CQL. GSQL, SQuAl, and Tribeca, which operate in the stream-in-stream-out mode, may be thought of as restrictions of SQL as they focus on incremental, non-blocking computation. In particular, GSQL and Tribeca are application-specific (network monitoring) and have been designed for very fast implementation [66]. However, although SQuAl and GSQL are stream-in-stream-out languages and as a result may have lost some expressive power as compared to SQL, they are extensible via user-defined functions. Moreover, SQuAl is noteworthy for its attention to issues related to real-time processing such as buffering, out-of-order arrivals and timeouts.

2.2 DSMS Query Operators

Recall that some relational operators are blocking. For instance, prior to returning the next result, the Nested Loops Join (NLJ) may potentially scan the entire inner table and compare each tuple therein with the current outer tuple. Given sufficient memory, any pipelined and non-blocking (or equivalently, monotonic [157]) query may be executed over data streams. Conceptually, an operator may be thought of as a function that consumes one or more inputs streams, stores some state, performs some computation in response to new data, and outputs a stream

[4]Other resampling functions are also possible, e.g., the maximum, minimum, or weighted average of the two neighbouring data values.

Language/ system	Allowed inputs	Allowed outputs	Novel operators	Supported windows	Execution frequency
CQL/ STREAM	streams and relations	streams and relations	relation-to-stream, stream-to-relation	sliding	continuous or periodic
GSQL/ Gigascope	streams	streams	order-preserving union	landmark	periodic
SQuAl/ Aurora	streams and relations	streams	resample, `map`, buffered sort	fixed, landmark, sliding	continuous or periodic
StreaQuel/ TelegraphCQ	streams and relations	sequences of relations	`WindowIs`	fixed, landmark, sliding	continuous or periodic
Tribeca	single stream	streams	multiplex, demultiplex	fixed, landmark, sliding	continuous

Table 2.1: Summary of proposed data stream languages

of results. As discussed below, although standard relational operators are supported by DSMSs, their implementation is considerably different.

2.2.1 Query Operators over Unbounded Streams

Operator Implementation

Duplicate-preserving projection, selection, and union are stateless operators that process new tuples on-the-fly, either by discarding unwanted attributes (projection) or dropping tuples that do not satisfy the selection condition. An example of selection over stream S_1 is shown in Figure 2.1(a). Note that only non-blocking merge union (on the timestamp column) is allowed in order to ensure that output is produced in timestamp order[5] [157].

A non-blocking pipelined join is shown in Figure 2.1(b) [82, 121, 123, 178, 189, 235, 246, 250, 258]. It stores the input streams (S_1 and S_2), possibly in the form of hash tables, and for each arrival on one of the inputs, the state of the other input is probed to generate new results. Joins of more than two streams and joins of streams with static relations are straightforward extensions. In the former, for each arrival on one input, the states of the other inputs are probed [250]. In the latter, new arrivals on the stream trigger the probing of the relation.

Duplicate elimination, illustrated in Figure 2.1(c), maintains a list of distinct values already seen and filters out duplicates from the output stream. As shown, when a new tuple with value b arrives, the operator probes its output list, and drops the new tuple because a tuple with value b has already been seen before and appended to the output stream.

Non-blocking aggregation is shown in Figure 2.1(d) [132, 157, 252]. When a new tuple arrives, a new result is appended to the output stream if the aggregate value has changed. The new result is understood to replace previously reported results. `GROUP BY` may be thought of as a general

[5]Technically, this means that duplicate-preserving union is not a stateless operator. However, it is required to store only out-of-order tuples, not the entire input.

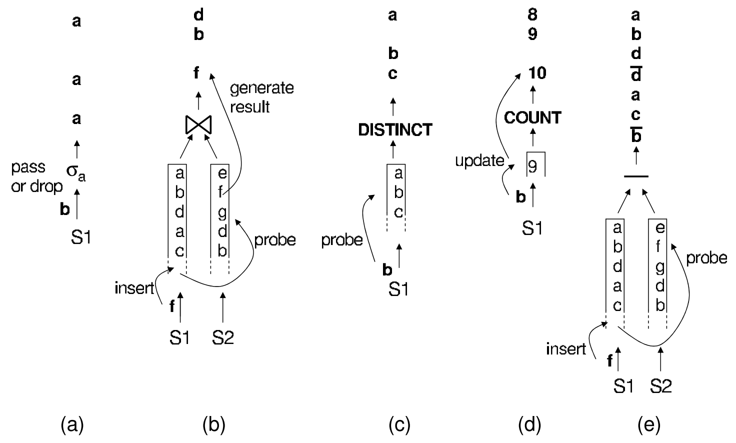


Figure 2.1: Examples of persistent query operators over data streams

case of aggregation, where a newly arrived tuple may produce new output if the aggregate value for its group has changed. The time and space requirements of aggregation depend upon the type of function being computed [116]. An aggregate f is *distributive* if, for two disjoint multi-sets X and Y, $f(X \cup Y) = f(X) \cup f(Y)$. Distributive aggregates, such as `COUNT`, `SUM`, `MAX` and `MIN`, may be computed incrementally using constant space and time (per tuple). For instance, `SUM` is evaluated by storing the current sum and continually adding to it the values of new tuples as they arrive. Moreover, f is *algebraic* if it can be computed using the values of two or more distributive aggregates using constant space and time (e.g., `AVG` is algebraic because `AVG = SUM / COUNT`). Algebraic aggregates are also incrementally computable using constant space and time. On the other hand, f is *holistic* if, for two multi-sets X and Y, computing $f(X \cup Y)$ requires space proportional to the size of $X \cup Y$. Examples of holistic aggregates include `TOP-k`, `QUANTILE`, and `COUNT DISTINCT`. For instance, multiplicities of each distinct value seen so far may have to be maintained in order to identify the k most frequent item types at any point in time. This requires $\Omega(n)$ space, where n is the number of stream tuples seen so far—consider a stream with $n-1$ unique values and one of the values occurring twice.

Non-monotonic queries over unbounded streams are possible if previously reported results can be removed if they cease to satisfy the query. This can be done by appending corresponding *negative tuples* to the output stream [12, 125]. This way, negation over two streams, $S_1 - S_2$, may produce results that are valid at a given time and possibly invalidate them later. An example is shown in Figure 2.1(e), where a tuple with value d was appended to the output because there did not exist any matching S_2-tuples at that time. However, a negative tuple (denoted by $\bar{d}$) was generated on the output stream upon subsequent arrival of an S_2-tuple with value d.

Memory Requirements

Joins, complex aggregation, and negation may require unbounded memory when executed over streams. Computing the memory requirements of continuous queries has been studied in [11] and, in an earlier context of enforcing temporal integrity constraints, in [55, 56]. Consider two unbounded streams: $S(A, B, C)$ and $T(D, E)$. The query $\pi_A(\sigma_{A=D \wedge A>10 \wedge D<20}(S \times T))$ may be evaluated in bounded memory whether or not the projection preserves duplicates. To preserve duplicates, for each integer i between 11 and 19, it suffices to maintain the count of tuples in S such that $A = i$ and the count of tuples in T such that $D = i$. To remove duplicates, it is necessary to store flags indicating which tuples have occurred such that $S.A = i$ and $T.D = i$ for $i \in [11, 19]$. Conversely, the query $\pi_A(\sigma_{A=D}(S \times T))$ is not computable in finite memory either with or without duplicates. Interestingly, $\pi_A(\sigma_{A>10}S)$ is computable in finite memory only if duplicates are preserved; any tuple in S with $A > 10$ is added to the answer as soon as it arrives. On the other hand, the query $\pi_A(\sigma_{B<D \wedge A>10 \wedge A<20}(S \times T))$ is computable in bounded memory only if duplicates are removed: for each integer i between 11 and 19, it suffices to maintain the current minimum value of B among all the tuples in S such that $A = i$ and the current maximum value of D over all tuples in T.

Exploiting Stream Constraints

One way to reduce the state requirements and unblock some queries is to exploit stream constraints. Constraints may take the form of control packets inserted into a stream, called *punctuations* [245]. For instance, a punctuation may arrive asserting that all the items henceforth have the a-attribute value larger than ten. This punctuation could be used to partially unblock a group-by query on a since all the groups where $a \leq 10$ are guaranteed not to change for the remainder of the stream's lifetime, or until another punctuation arrives and specifies otherwise. Punctuations may also be used to reduce the state required for operators such as joins [80, 81, 165] and aggregation [144, 167], and to synchronize multiple streams in that a source may send a punctuation asserting that it will not produce any more tuples with timestamp smaller than τ [12]. The latter are called *heartbeats* [144, 167, 227]. Note that constraints may be useful even if they are not precisely adhered to at all times [28]. For example, if two streams have nearly synchronized timestamps, then an equi-join on the timestamp attribute can be performed using very little space.

Approximating Unbounded Streams in Limited Space

Recall that holistic aggregates may require unbounded memory in order to return exact results. An alternate solution is to maintain a (possibly non-uniform) sample of the stream and compute approximate aggregates over the sample. Sampling is used in a number of algorithms for approximating `COUNT DISTINCT` [101, 199], `QUANTILE` [186] and `TOP k` [72, 86, 102, 187] queries. Another possibility is to avoid storing frequency counters for each distinct value seen so far by periodically evicting counters having low values [174, 187, 188]. This is a possible approach for computing `TOP k` queries, so long as frequently occurring values are not missed by repeatedly deleting and re-starting counters. A related space-reduction technique may also be used for approximate quantile computation, where the rank of a subset of values is stored along with corresponding error bounds (rather than storing a sorted list of all the frequency counters for exact quantile calculation) [106, 117]. Finally, hashing is another way of reducing the number of counters that need to be maintained. Stream summaries created using hashing are often referred to as *sketches*. Examples include the following.

- A Flajolet-Martin (FM) sketch [9, 90] is used for `COUNT DISTINCT` queries. It uses a set of hash functions h_j that map each value v onto the integral range $[1, \ldots, \log U]$ with probability $\mathbf{PR}[h_j(v)=l] = 2^{-l}$, where U is the upper bound on the number of possible distinct values. Given d distinct items in the stream, each hash function is expected to map $d/2$ items to bucket 1, $d/4$ items to bucket 2, and so on with all buckets above $\log d$ expected to be empty. Thus, the highest numbered non-empty bucket is an estimate for the value $\log d$. The FM sketch approximates d by averaging the estimate of $\log d$ (i.e. the largest non-zero bit) from each hash function.
- A Count-Min (CM) sketch [63] is a two-dimensional array of counters with dimensions d by w. There are d associated hash functions, call them h_1 through h_d, each mapping stream items onto the integral range $[1, \ldots, w]$. When a new tuple arrives with value v, the following cells in the array are incremented: $[i, h_i(v)]$ for $= 1$ to d. An estimate of the count of tuples with value v can be obtained by taking the minimum of values found in cells $[i, h_i(v)]$ for $i = 1$ to d. The approximate counts can then be used to compute `TOP k` queries.

A more detailed survey of stream summarization, approximate histograms, and approximate stream algorithms is beyond the scope of this dissertation. Further information on these topics may be found in recent surveys and tutorials [97, 120, 192]. Moreover, implementation of approximate algorithms in the Gigascope DSMS is described in [62, 143].

2.2.2 Query Operators over Sliding Windows

Sliding window operators process two types of events: arrivals of new tuples and expirations of old tuples; the orthogonal problem of determining when tuples expire will be discussed in Section 2.3.2. The actions taken upon arrival and expiration vary across operators [125, 251]. A new tuple may generate new results (e.g., join) or remove previously generated results (e.g., negation). Furthermore, an expired tuple may cause a removal of one or more tuples from the result (e.g., aggregation) or an addition of new tuples to the result (e.g., duplicate elimination and negation). Moreover, operators that must explicitly react to expired tuples (by producing new results or invalidating existing results) perform state purging eagerly (e.g., duplicate elimination, aggregation, and negation), whereas others may do so eagerly or lazily (e.g., join).

In a sliding window join, newly arrived tuples on one of the inputs probe the state of the other input, as in a join of unbounded streams. Additionally, expired tuples are removed from the state [111, 123, 124, 147, 254]. Expiration can be done periodically (lazily), so long as old tuples can be identified and skipped during processing.

Aggregation over a sliding window updates its result when new tuples arrive and when old tuples expire[6] (recall Figure 1.1). In many cases, the entire window needs to be stored in order to account for expired tuples, though selected tuples may sometimes be removed early if their expiration is guaranteed not to influence the result. For example, when computing `MAX`, tuples with value v need not be stored if there is another tuple in the window with value greater than v and a younger timestamp (see, e.g., [171, 234] for additional examples of reducing memory usage in the context of skyline queries and [191] in the context of top-k queries). Additionally, in order to enable incremental computation, the aggregation operator stores the current answer (for distributive and algebraic aggregates) or frequency counters of the distinct values present in the window (for holistic aggregates). For instance, computing `COUNT` entails storing the current count, incrementing it when a new tuple arrives, and decrementing it when a tuple expires. Note that, in contrast to the join operator, expirations must be dealt with immediately so that an up-to-date aggregate value can be returned right away. For example, recall Figure 1.1 and observe that if the expiration of the tuple with value 75 is delayed, then the answer will be temporarily incorrect—as soon as the tuple with value 75 expires, the answer must be changed to 73.

Duplicate elimination over a sliding window may also produce new output when an input tuple expires. This occurs if a tuple with value v was produced on the output stream and later expires from its window, yet there are other tuples with value v still present in the window [125]. Alternatively, as is the case in the STREAM DSMS, duplicate elimination may produce a single result tuple with a particular value v and retain it on the output stream so long as there is at least one tuple with value v present in the window; these two alternatives will be analyzed in more detail in Chapter 3. In both cases, expirations must be handled eagerly so that the correct result is maintained at all times.

Finally, negation of two sliding windows, $W_1 - W_2$, may produce negative tuples (e.g., arrival of a W_2-tuple with value v causes the deletion of a previously reported result with value v), but may also produce new results upon expiration of tuples from W_2 (e.g., if a tuple with value v expires from W_2, then a W_1-tuple with value v may need to be appended to the output stream [125]). One way of implementing duplicate-preserving negation is as follows. The left and right inputs are stored along with multiplicities of the distinct values occurring within. Let v_1 and v_2 be the number of tuples with value v in W_1 and W_2, respectively. For each distinct value v present in W_1, the output of the negation operator consists of v_3 tuples from W_1 such that

$$v_3 = \begin{cases} v_1 - v_2 & \text{if } v_1 > v_2 \\ 0 & \text{otherwise.} \end{cases}$$

A new arrival on W_1 with value v is inserted into its state buffer and the corresponding counter (v_1) is incremented. If $v_1 > v_2$, then the new tuple is appended to the output stream. Expiration from W_1 is handled (eagerly) by removing the old tuple from the W_1 state and decrementing v_1. An arrival on W_2 with value v is inserted into its state buffer and increments v_2. If $v_2 \leq v_1$, then

[6]See [252] for implementation of sliding window aggregates as SQL user-defined functions.

one result tuple with value v (say the oldest) must be deleted from the answer set to satisfy the negation condition in Equation 13. As in the case in negation over unbounded streams, these explicit deletions are represented as negative tuples. Finally, if a tuple with value v expires from W_2, then v_2 is decremented and if $v_1 \geq v_2$, then W_1 is probed and a tuple from W_1 with value v (say the youngest) is appended to the output stream.

2.3 DSMS Query Processing

Having discussed the implementation of individual operators, this section outlines DSMS query processing. As in a DBMS, declarative queries are translated into execution plans that map logical operators specified in the query into physical implementations. For now, the inputs and operator state are assumed to fit in main memory; disk-based processing will be discussed in Section 2.3.4.

2.3.1 Queuing and Scheduling

DBMS operators are pull-based, whereas DSMS operators consume data pushed into the plan by the sources. Queues allow sources to push data into the query plan and operators to retrieve data as needed [2, 17, 12, 180, 183]; see [141] for a discussion on calculating queue sizes of streaming relational operators using classical queueing theory. A simple scheduling strategy allocates a time slice to each operator, during which the operator extracts tuples from its input queue(s), processes them in timestamp order, and deposits output tuples into the next operator's input queue. The time slice may be fixed or dynamically calculated based upon the size of an operator's input queue and/or processing speed. A possible improvement could be to schedule one or more tuples to be processed by multiple operators at once. In general, there are several possibly conflicting criteria involved in choosing a scheduling strategy, among them queue sizes in the presence of bursty stream arrival patterns [20], average or maximum latency of output tuples [46, 142, 195], and average or maximum delay in reporting the answer relative to the arrival of new data [220].

2.3.2 Determining When Tuples Expire

In addition to dequeuing and processing new tuples, sliding window operators must remove old tuples from their state buffers and possibly update their answers, as discussed in Section 2.2.2. Expiration from an individual time-based window is simple: a tuple expires if its timestamp falls out of the range of the window. That is, when a new tuple with timestamp ts arrives, it receives another timestamp, call it exp, that denotes its expiration time as ts plus the window length. In effect, every tuple in the window may be associated with a lifetime interval of length equal to the window size [154]. Now, if this tuple joins with a tuple from another window, whose insertion and expiration timestamps are ts' and exp', respectively, then the expiration timestamp of the result tuple is set to $\min(exp, exp')$. That is, a composite result tuple expires if at least one of its constituent tuples expires from its windows (recall Definition 2.1). This means that various join results may have different lifetime lengths and furthermore, the lifetime of a join result may have a lifetime that is shorter than the window size [45]. Moreover, as discussed in Section 2.2.2, the negation operator may force some result tuples to expire earlier than their *exp* timestamps by generating negative tuples. Finally, if a stream is not bounded by a sliding window, then the expiration time of each tuple is infinity [154].

In a count-based window, the number of tuples remains constant over time. Therefore, expiration can be implemented by overwriting the oldest tuple with a newly arrived tuple. However, if an operator stores state corresponding to the output of a count-based window join, then the number of tuples in the state may change, depending upon the join attribute values of new tuples. In this case, expirations must be signaled explicitly using negative tuples.

2.3.3 Continuous Query Processing over Sliding Windows

There are two techniques for sliding window query processing and state maintenance: the negative tuple approach [12, 125, 127] and the direct approach [125, 127].

Negative Tuple Approach

In the negative tuple approach, each window referenced in the query is assigned an operator that explicitly generates a negative tuple for every expiration, in addition to pushing newly arrived tuples into the query plan. Thus, each window must be materialized so that the appropriate negative tuples are produced. This approach generalizes the purpose of negative tuples, which are now used to signal all expirations explicitly, rather than only being produced by the negation operator if a result tuple expires because it no longer satisfies the negation condition. Negative tuples propagate through the query plan and are processed by operators in a similar way as regular tuples, but they also cause operators to remove corresponding "real" tuples from their state[7]. This is illustrated in Figure 2.2, showing how aggregation over a sliding window join processes a particular negative tuple generated by an expiration from the window over Stream 1 (expirations from the other window are treated similarly and are not shown for clarity). Observe that the negative tuple is processed by all the operators in the pipeline, therefore the aggregation operator may eventually receive a number of negative tuples corresponding to all the join results in which the original negative tuple participated.

The negative tuple approach can be implemented efficiently using hash tables as operator state so that expired tuples can be looked up quickly in response to negative tuples. Conceptually, this is similar to a DBMS indexing a table or materialized view on the primary key in order to speed up insertions and deletions. However, the downside is that twice as many tuples must be processed by the query because every tuple eventually expires from its window and generates a corresponding negative tuple. Furthermore, additional operators (labeled window state in Figure 2.2) must be present in the plan to generate negative tuples as the window slides forward.

[7]Note that rather than storing a local copy of each tuple, operator state may consist of pointers to tuples.

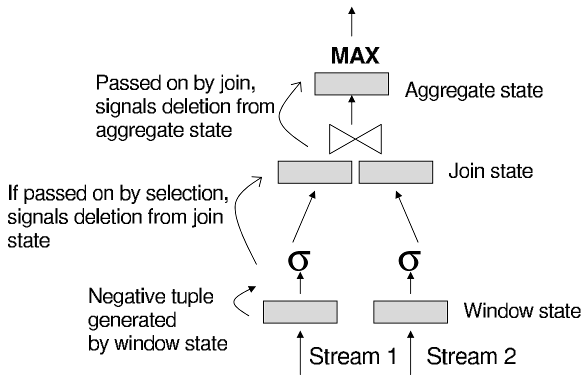

MAX
Passed on by join, signals deletion from aggregate state
Aggregate state
Join state
If passed on by selection, signals deletion from join state
σ
σ
Negative tuple generated by window state
Window state
Stream 1
Stream 2

Figure 2.2: Query execution using the negative tuple approach

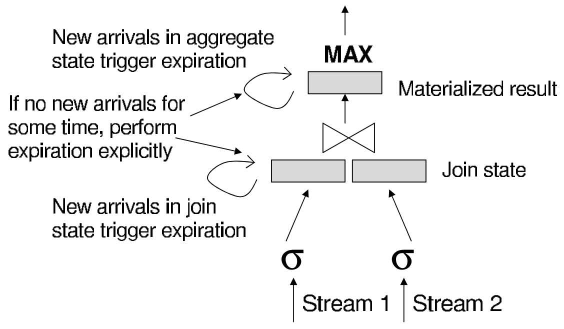

New arrivals in aggregate state trigger expiration
MAX
Materialized result
If no new arrivals for some time, perform expiration explicitly
Join state
New arrivals in join state trigger expiration
σ
σ
Stream 1
Stream 2

Figure 2.3: Query execution using the direct approach

Direct Approach

Negation-free queries over time-based windows have the property that the expiration times of base tuples and intermediate results can be determined via their *exp* timestamps, as explained in Section 2.3.2. Hence, operators can access their state directly and find expired tuples without the need for negative tuples. The direct approach is illustrated in Figure 2.3 for the same query as in Figure 2.2; again, only deletions from the window over Stream 1 are illustrated. For every new arrival into one of the join state buffers, expiration is performed at the same time as the processing of the new tuple. However, if there are no arrivals for some time (this interval may be specified by the user as the maximum delay in reporting new answers), then each operator that stores state initiates expiration from its state buffer.

The direct approach does not incur the overhead of negative tuples and does not have to store the base windows referenced in the query. However, it may be slower than the negative tuple approach for queries over multiple windows [125]. This is because straightforward implementations of state buffers may require a sequential scan during insertions or deletions. For example, if the state buffer is sorted by tuple arrival time, then insertions are simple, but deletions require a sequential scan of the buffer. On the other hand, sorting the buffer by expiration time simplifies deletions, but insertions may require a sequential scan to ensure that the new tuple is ordered correctly, unless the insertion order is the same as the expiration order. This issue will be dealt with in Chapter 3.

A technical issue with the direct approach is that newly arrived tuples may not be processed immediately by all the operators in the pipeline, therefore the state of intermediate results may be delayed with respect to the inputs. For example, in Figure 2.3, tuples with timestamps of up to 100 may have arrived on the input streams, but the `MAX` operator may have only processed tuples with timestamps up to 98 (with the remaining tuples "stuck" in one of the operator queues along the pipeline). One solution to guarantee correct results maintains a local clock at each operator,

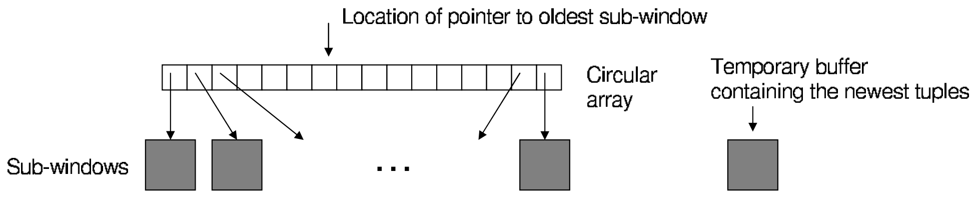

Location of pointer to oldest sub-window
Circular array
Temporary buffer containing the newest tuples
Sub-windows
...

Figure 2.4: Sliding window implemented as a circular array of pointers to sub-windows

corresponding to the timestamp of the tuple most recently processed by its parent [125]. This way, the local clock of the `MAX` operator in Figure 2.3 is 98 and it will not expire tuples out of its state prematurely by assuming that the current time is 100.

A related problem appears when newly arrived tuples are dropped early on during processing because they do not satisfy the query's selection predicate. In this case, the `MAX` operator in Figure 2.3 would not receive any new input, yet it should check for old tuples because the aggregate value may have changed as a result of an expiration. The local clock approach can handle this case as well—the `MAX` operator requests the value of the local clock of the join, which then requests the value of the local clock of the selection operator. Even though the selection operator may not have passed any tuples forward, its local clock will correspond to the timestamp of the latest tuple that it has seen. An alternate solution is for the selection operator to periodically propagate a punctuation (heartbeat) into the plan if it has not passed on any "regular" tuples that match the selection predicate. The punctuation contains the timestamp of the most recently processed tuple, even if that tuple did not satisfy the selection condition.

2.3.4 Periodic Query Processing Over Sliding Windows

Query Processing over Windows Stored in Memory

For reasons of efficiency (reduced expiration and query processing costs) and user preference (users may find it easier to deal with periodic output rather than a continuous output stream [16, 49]), sliding windows may be advanced and queries re-evaluated periodically with a specified frequency[8] [2, 48, 52, 109, 156, 167, 175, 222, 270]. As illustrated in Figure 2.4, a periodically-sliding window can be modeled as a circular array of *sub-windows*, each spanning an equal time interval for time-based windows (e.g., a ten-minute window that slides every minute) or an equal number of tuples for tuple-based windows (e.g., a 100-tuple window that slides every ten tuples). Let a *window update* denote the process of replacing the oldest sub-window with newly arrived data (accumulated in a buffer), thereby sliding the window forward by one sub-window. There has been recent work on periodic evaluation of window aggregates, with a secondary goal of sharing state among similar aggregates over different window sizes, as summarized below.

[8]Alternatively, queries may be re-evaluated on-demand, as suggested in [210], in which case a materialized query result is required to contain the correct results only when probed by a query.

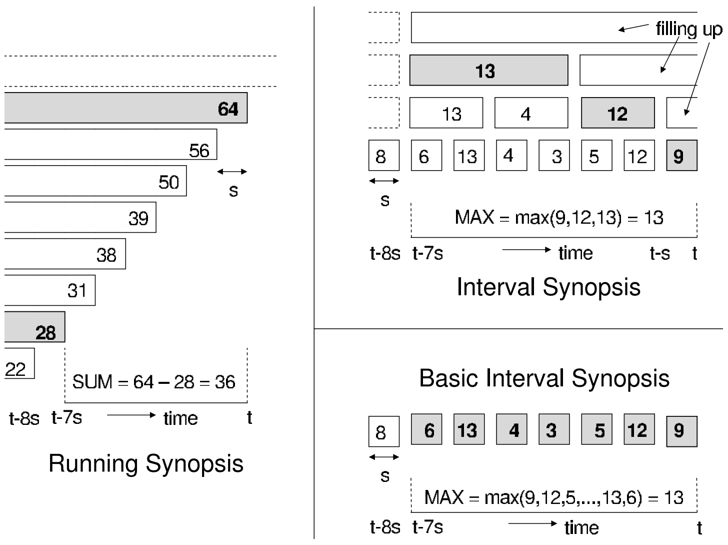


Figure 2.5: Examples of running, interval, and basic interval synopses

Rather than storing the entire window and re-computing an aggregate after every new tuple arrives or an old tuple expires, a synopsis can be stored that pre-aggregates each sub-window and reports updated answers whenever the window slides forward by one sub-window. First, a *running synopsis* [16] is used for subtractable aggregates [59] such as `SUM` and `COUNT`. An aggregate f is subtractable if, for two multi-sets X and Y such that $X \supseteq Y$, $f(X - Y) = f(X) - f(Y)$. A running synopsis is associated with three parameters: s, which is the time between updates (i.e., the sub-window size), b, which defines the longest window covered by the synopsis as bs, and f, which is the type of aggregate function used to create the synopsis. Let t be time of the last update of the sliding window. The synopsis stores aggregate values over b running intervals: $f([1,t]), f([1,t-s]), f([1,t-2s]), \ldots, f([1,t-bs])$. To compute f over a window of size ns (where $n < b$) at time t , i.e., $f((t-ns,t])$, it suffices to calculate $f([1,t]) - f([1,t-ns])$. An example is shown in Figure 2.5, computing `SUM` over a window of size $7s$ using a running synopsis with $b = 8$ and $f =$ `SUM`. The next synopsis update, which takes place at time $t+s$, replaces $f([1,t-8s])$ with $f([1,t+s])$, where $f([1,t+s])$ can be computed as $f([1,t]) + f((t,t+s])$. This can be done efficiently by having the buffer pre-compute $f((t,t+s])$ incrementally as new tuples arrive.

An *interval synopsis* applies to distributive aggregates (recall Section 2.2.1) that are not subtractable, such as `MIN` and `MAX`. An interval synopsis with parameters s, b, and f (as defined above; additionally, assume that b is a power of two) stores the values of f over $2b$ intervals. In particular, there are b disjoint intervals of length s, $\frac{b}{2}$ disjoint intervals of length $2s$, and so on up to one interval of length bs. The example in Figure 2.5 uses an interval synopsis with $b = 8$ and $f =$ `MAX` to compute `MAX` over a window of size $7s$. To do this, the maximum of the values stored in three disjoint intervals is taken; in general, access to $\log b$ intervals is required. During the next update at time $t+s$, $f((t-8s,t-7s])$ may be dropped because this interval now references expired tuples. Furthermore, $f((t, t+s])$ can now be inserted into the synopsis. As before, $f((t, t+s])$ may be pre-computed in the buffer. Moreover, interval $(t-s, t+s]$ is now full and its value may be computed as $\max(f((t-s, t]), f((t, t+s]))$. Variations of the interval synopsis have been proposed in [16, 41, 42, 272, 277].

Note that algebraic aggregates may be computed using the synopses of appropriate distributive aggregates. For instance, since `AVG = SUM / COUNT`, a query computing the average may use `SUM` and `COUNT` synopses, provided that they cover the appropriate window length.

Holistic aggregates may use an interval synopsis if it stores additional information per interval. For instance, storing the frequency counts of values occurring in each interval may be used for `QUANTILE`, `TOP k`, and `COUNT DISTINCT` queries In this case, the buffer pre-aggregates the newest interval by maintaining frequency counts of new tuples. Alternatively, a sample or sketch may be stored in each interval [14, 160] (recall Section 2.2.1). Merging individual sketches to obtain an approximation over the entire window is straightforward; for instance, the corresponding entries in the CM sketch arrays can be added.

Finally, space usage and synopsis update times may be reduced by storing only the b short intervals "at the bottom" of an interval synopsis. The tradeoff is that up to b intervals must now be accessed during probing, up from $\log b$. The resulting structure will be referred to as a *basic interval synopsis*. An example is shown in Figure 2.5 for $b = 8$ and a `MAX` query over a window of length $7s$. Variations of the basic interval synopsis have been proposed in [109, 156, 166, 276].

Note that frequency counts and sketches could be stored in a running synopsis instead of an interval or basic interval synopsis, and appropriate intervals could be subtracted to compute answers over various window lengths. The problem with storing counters or sketches in this way is that each running interval summarizes the entire stream up to a certain point. Therefore, the synopsis may store counts of values that appeared a long time ago and may never be seen again. This leads to excessive space usage and increased computation costs. Additionally, given two sketches of the same size, one summarizing the interval $[1, t+s]$ and another summarizing $(t, t+s]$, the latter is expected to yield better accuracy since it does not have to approximate the entire distribution of the stream. As a result, complex aggregates are assumed to use an interval or basic interval synopsis.

A disadvantage of periodic query evaluation is that results may be stale. One way to stream new results after each new item arrives is to bound the error caused by delayed expiration of tuples in the oldest sub-window. In [70], it is shown that restricting the sizes of the sub-windows (in terms of the number of tuples) to powers of two and imposing a limit on the number of sub-windows of each size yields a space-optimal algorithm (called *exponential histogram*, or EH) that approximates simple aggregates to within ϵ using logarithmic space (with respect to the sliding window size). Variations of the EH algorithm have been used to approximately compute the sum [70, 104], variance and k-medians clustering [23], windowed histograms [201], and order statistics [170, 265]. Extensions of the EH algorithm to time-based windows are given in [58].

Query Processing over Windows Stored on Disk

In traditional database applications that use secondary storage, performance may be improved if appropriate indices are built. Consider maintaining an index over a periodically-sliding window stored on disk, e.g., in a data warehousing scenario where new data arrive periodically and decision support queries are executed (off-line) over the latest portion of the data. In order to reduce the index maintenance costs, it is desirable to avoid bringing the entire window into memory during every update. This can be done by partitioning the data so as to localize updates (i.e., insertions of newly arrived data and deletion of tuples that have expired from the window) to a small number of disk pages. For example, if an index over a sliding window is partitioned chronologically [91, 222], then only the youngest partition incurs insertions, while only the oldest partition needs to be checked for expirations (the remaining partitions "in the middle" are not accessed). A similar idea of grouping objects by expiration time appears in [85] in the context of clustering large file systems, where every file has an associated lifetime. However, the disadvantage of chronological clustering is that records with the same search key may be scattered across a very large number of disk pages, causing index probes to incur prohibitively many disk I/Os.

One way to reduce index access costs is to store a reduced (summarized) version of the data that fits on fewer disk pages [50], but this does not necessarily improve index update times. In order to balance the access and update times, a *wave index* has been proposed that chronologically divides a sliding window into n equal partitions, each of which is separately indexed and clustered by search key for efficient data retrieval [222]. An example is shown in Figure 2.6, where a window of size 16 minutes that is updated every 2 minutes is split into four sub-indices: I_1, I_2, I_3, and I_4. Triangles indicate index directories—each associated with a single sub-index—which could be B+-trees, R-trees, or any other data structure as appropriate. Rectangles represent data records, which are stored on disk. On the left, the window is partitioned by insertion time. On the right, an equivalent partitioning is shown by expiration time; the window size of 16 is added to each item's insertion time to determine the expiration time (recall Section 2.3.2). As illustrated, an update at time 18 inserts newly arrived tuples between times 17 and 18 (which will expire between times 33 and 34) into I_1, at the same time deleting tuples which have arrived between times one and 2 (or which have expired between times 17 and 18). The advantage of this approach is that only one sub-index is affected by any given update; for instance, only I_1 changes at times 18 and 20, only I_2 will change at times 22 and 24, and so on. The tradeoff is that access times are slower because multiple sub-indices are probed to obtain the answer. Disk-based indexing of time-evolving data will be revisited in Chapter 4.

2.4 DSMS Query Optimization

It is usually the case that a query may be executed in a number of different ways. A DBMS query optimizer is responsible for enumerating (some or all of) the possible query execution strategies and choosing an efficient one using a cost model and/or a set of transformation rules. A DSMS query optimizer has the same responsibility, but it must use an appropriate cost model and rewrite

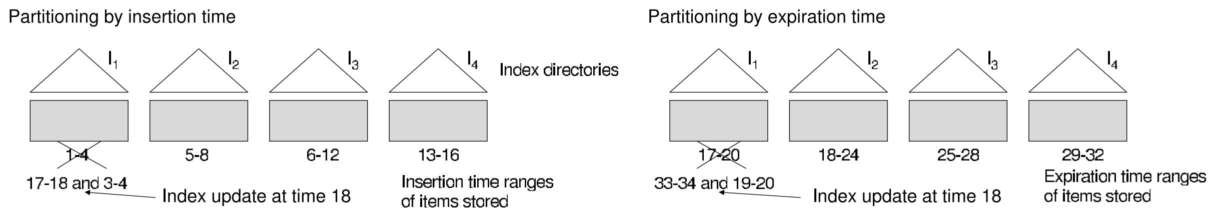

Partitioning by insertion time
I1
I2
I3
I4
Index directories
1-4
5-8
6-12
13-16
17-18 and 3-4
Index update at time 18
Insertion time ranges of items stored
Partitioning by expiration time
17-20
18-24
25-28
29-32
33-34 and 19-20
Index update at time 18
Expiration time ranges of items stored

Figure 2.6: Two equivalent illustrations of a wave index

rules. Additionally, DSMS query optimization involves adaptivity, load shedding, and resource sharing among similar queries running in parallel, as summarized below.

2.4.1 Cost Metrics and Statistics

Traditional DBMSs use selectivity information and available indices to choose efficient query plans (e.g., those which require the fewest disk accesses). However, this cost metric does not apply to (possibly approximate) persistent queries, where processing cost per-unit-time is more appropriate [147]. Alternatively, if the stream arrival rates and output rates of query operators are known, then it may be possible to optimize for the highest output rate or to find a plan that takes the least time to output a given number of tuples [235, 247, 249]. Finally, quality-of-service metrics such as response time may also be used in DSMS query optimization [2, 36, 211, 212].

2.4.2 Query Rewriting and Adaptive Query Optimization

Some of the DSMS query languages discussed in Section 2.1.3 introduce rewritings for new operators, e.g., selections and time-based sliding windows commute, but not selections and count-based windows [12]. Other rewritings are similar to those used in relational databases, e.g., re-ordering a sequence of binary joins in order to minimize a particular cost metric. There has been some work in join ordering for data streams in the context of the rate-based model [249, 250]. Furthermore, adaptive re-ordering of pipelined stream filters is studied in [26] and adaptive materialization of intermediate join results is considered in [27].

Note the prevalence of the notion of adaptivity in query rewriting; operators may need to be re-ordered on-the-fly in response to changes in system conditions. In particular, the cost of a query plan may change for three reasons: change in the processing time of an operator, change in the selectivity of a predicate, and change in the arrival rate of a stream [17]. Initial efforts on adaptive query plans include mid-query re-optimization [146] and query scrambling, where the objective was to pre-empt any operators that become blocked and schedule other operators instead [10, 248]. To further increase adaptivity, instead of maintaining a rigid tree-structured query plan, the Eddies approach (introduced in [17], evaluated in [77], extended to multi-way joins in [241, 203], applied to continuous queries in [49, 183], and currently being extended to consider semantics information such as attribute correlations during routing [37]), performs scheduling of each tuple separately by routing it through the operators that make up the query plan. In effect, the query plan is dynamically re-ordered to match current system conditions. This is accomplished by tuple routing policies that attempt to discover which operators are fast and selective, and those operators are scheduled first. A recent extension adds queue length as the third factor for tuple routing strategies in the presence of multiple distributed Eddies [240]. There is, however, an important trade-off between the resulting adaptivity and the overhead required to route each tuple separately. More details on adaptive query processing may be found in [25, 29, 115].

Adaptivity involves on-line reordering of a query plan and may therefore require that the internal state stored by some operators be migrated over to the new query plan consisting of a different arrangement of operators. The issue of state migration across query plans has recently been studied in [79, 275].

2.4.3 Load Shedding and Approximation

The stream arrival rates may be so high that not all tuples can be processed, regardless of the (static or run-time) optimization techniques used. In this case, two types of load shedding may be applied—random or semantic—with the latter making use of stream properties or quality-of-service parameters to drop tuples believed to be less significant than others [236]. For an example of semantic load shedding, consider performing an approximate sliding window join with the objective of attaining the maximum result size. The idea is that tuples that are about to expire or tuples that are not expected to produce many join results should be dropped (in case of memory limitations [68, 164, 262]), or inserted into the join state but ignored during the probing step (in case of CPU limitations [19, 99, 128]). Note that other objectives are possible, such as obtaining a random sample of the join result [228].

In general, it is desirable to shed load in such a way as to minimize the drop in accuracy. This problem becomes more difficult when multiple queries with many operators are involved, as it must be decided where in the query plan the tuples should be dropped. Clearly, dropping tuples early in the plan is effective because all of the subsequent operators enjoy reduced load. However, this strategy may adversely affect the accuracy of many queries if parts of the plan are shared. On the other hand, load shedding later in the plan, after the shared sub-plans have been evaluated and the only remaining operators are specific to individual queries, may have little or no effect in reducing the overall system load. Results on the problem of optimal placement of sampling operators in multi-query plans may be found in [22] for the special case of random load shedding for windowed aggregates and in [238] for quality-of-service-driven load shedding for windowed aggregates. Moreover, approximate evaluation of expensive user-defined functions over streams is addressed in [76]. Finally, a load shedding approach employing a feedback loop to monitor the queue lengths is discussed in [244].

One issue that arises in the context of load shedding and query plan generation is whether an optimal plan chosen without load shedding is still optimal if load shedding is used. It is shown in [18] that this is indeed the case for sliding window aggregates, but not for queries involving sliding window joins.

Note that instead of dropping tuples during periods of high load, it is also possible to put them aside (e.g., spill to disk) and process them when the load has subsided [173, 204]. Finally, note that in the case of periodic re-execution of persistent queries, increasing the re-execution interval may be thought of as a form of load shedding [21, 45, 261].

2.4.4 Multi-Query Optimization

As seen in Section 2.3.4, memory usage may be reduced by sharing internal data structures that store operator state [75, 84, 271]. Additionally, in the context of complex queries containing stateful operators such as joins, computation may be shared by building a common query plan [52]. For example, queries belonging to the same group may share a plan, which produces the union of the results needed by the individual queries. A final selection is then applied to the shared result set and new answers are routed to the appropriate queries. An interesting trade-off appears between doing similar work multiple times and doing too much unnecessary work; techniques that balance this trade-off are presented in [51, 155, 253]. For example, suppose that the workload includes several queries referencing a join of the same windows, but having a different selection predicate. If a shared query plan performs the join first and then routes the output to appropriate queries, then too much work is being done because some of the joined tuples may not satisfy any selection predicate (unnecessary tuples are being generated). On the other hand, if each query performs its selection first and then joins the surviving tuples, then the join operator cannot be shared and the same tuples will be probed many times. Finally, sharing a single join operator among queries referencing different window sizes is discussed in [126].

For selection queries, a possible multi-query optimization is to index the query predicates and store auxiliary information in each tuple that identifies which queries it satisfies [49, 73, 129, 156, 168, 183, 260]. When a new tuple arrives for processing, its attribute values are extracted and matched against the query index to see which queries are satisfied by this tuple. Data and queries may be thought of as duals, in some cases reducing query processing to a multi-way join of the query predicate index and the data tables [49, 168]. Indexing range predicates is discussed in [168, 260], whereas a predicate index on multiple attributes is presented in [159, 168].

Chapter 3

Update-Pattern-Aware Modeling and Processing of Persistent Queries

3.1 Introduction

As motivated in Section 1.2, a defining characteristic of persistent queries over streams and windows is the potentially unbounded and time-evolving nature of their inputs and outputs. New answers are produced in response to the arrival of new data and older data expire as the windows slide forward. Furthermore, previously reported answers may cease to satisfy the query at some point. An update pattern of a persistent query plan is said to be the order in which its results are produced and deleted over time. This chapter analyzes the update patterns of persistent query plans, and presents update-pattern-aware query semantics and processing strategies [112].

As discussed in Section 2.1.2, previous work on update patterns of persistent queries distinguishes between monotonic and non-monotonic queries, with the conclusion that only the former are feasible over unbounded streams [157, 239]. The update patterns of monotonic queries are simple because results never expire. Hence, this classification is not sufficiently precise as it fails to sort non-monotonic queries according to the pattern of deletions from their answer sets. To motivate the need for a study of update patterns of persistent queries and a more precise definition of their semantics, note that Definition 2.1 from Section 2.1.2 does not distinguish between the time-evolving state of the data streams versus relational tables. Existing research either disallows relations in continuous query plans [18], assumes that relations are static, at least throughout the lifetime of a persistent query [2, 21, 125, 180], or allows arbitrary updates [30, 190]. If updates of tables are allowed, then they are likely to be semantically different from changes caused by the movement of the sliding windows. Furthermore, as discussed in Section 2.1.2, existing research offers two conflicting viewpoints on the nature of sliding window queries. According to Definition 2.1 and [125, 154], queries over sliding windows are non-monotonic because all of their results eventually expire as the windows slide forward. However, [2, 12, 48, 147] treat some sliding window operators (e.g., join and aggregation) as monotonic. Similarly, [157] considers on-line incremental aggregation over an unbounded stream to be monotonic, thereby assuming that out-of-date answers are not deleted from the result.

Another issue that depends upon the knowledge of update patterns is the maintenance of operator state. Recall the two approaches to sliding window query processing from Section 2.3.3. The direct approach eliminates the need for generating and propagating negative tuples, but may be less efficient than the negative tuple approach if the state buffers must be scanned sequentially for every expiration. If the order in which tuples expire is known, then suitable data structures may be designed to reduce the state maintenance overhead.

The contributions of this chapter consist of the classification of update patterns of persistent queries and two applications of this classification, namely definition of precise query semantics and introduction and evaluation of update-pattern-aware query processing techniques. In particular:

- A classification of update patterns of persistent query plans is presented that divides non-monotonic plans into three types in order to highlight the differences in their expiration patterns.
- The classification is used to formulate the semantics of persistent queries. The update-pattern-aware definition of query semantics addresses the following issues. First, updates of relations are treated separately from expirations from sliding windows. Second, the two conflicting viewpoints regarding the monotonicity of sliding window queries are reconciled. Third, the difference between a window on the input stream versus a window on the output stream of a query is analyzed. Fourth, it is shown that for some types of queries, periodic re-evaluation may produce fewer result tuples than continuous execution.
- Update-pattern-aware semantics of persistent queries are incorporated into CQL.
- An update-pattern-aware query processor is developed, where each branch in the query plan is annotated with its update patterns and physical operator implementations vary according to the nature of their inputs. In particular, operators use update-pattern-aware data structures for intermediate state maintenance.
- An update-pattern-aware optimization framework is presented, with the goal of choosing a query plan with the simplest update patterns in order to minimize state maintenance costs and simplify operator implementation. When tested on IP traffic logs, update-pattern-aware query plans significantly outperform existing data stream processing techniques.

In the remainder of this chapter, Section 3.2 presents a classification of update patterns of persistent queries, Section 3.3 uses the classification to define persistent query semantics, Section 3.4 incorporates update-pattern-aware semantics into CQL, Section 3.5 develops update-pattern-aware processing and optimization strategies, and Section 3.6 presents experimental results.

3.2 Update Patterns of Persistent Queries

3.2.1 Classification

Recall from Section 2.3.2 that a monotonic query plan produces results that never expire. On the other hand, results of non-monotonic query plans have finite lifetimes. The purpose of this section is to analyze the nature of the lifetimes of the results of persistent query plans in order to identify their update patterns. Assume (for the remainder of this dissertation) that data streams consist of relational tuples with a fixed schema arriving in non-decreasing timestamp order[1]. Additionally, in order to simplify the presentation of the forthcoming classification, assume that each tuple is processed instantaneously as soon as it is generated. That is, if a tuple is generated at time ts, it is assumed that all results that it produces, possibly by joining with other tuples which have arrived previously and have not yet expired from their windows, are also generated at time ts.

Let Q be an arbitrary query plan that satisfies the semantics of Definition 2.1 and let τ be some point in time after all the sliding windows referenced by Q have filled up. Recall from Section 2.1.2 that $Q(\tau)$ is defined to be the answer set of Q at time τ. Now, assume that time is a set of natural numbers and that zero, one, or more tuples may arrive at one clock tick. Let $S(\tau)$ be the multi-set of tuples generated on the input stream(s) of Q at time τ and let $S(0, \tau)$ be the multi-set of tuples generated on the input(s) at times up to and including τ. S may contain an arbitrary (but finite) number of tuples having arbitrary attribute values, so long as the values are chosen from the specified attribute domains. Furthermore, let $P_S(\tau)$ be the multi-set of result tuples produced at time τ and let $E_S(\tau)$ be the multi-set of result tuples that expire at time τ, given an input set S; let $P_S(0, \tau)$ and $E_S(0, \tau)$ be all the tuples produced and expired up to and including time τ, respectively. Note that tuples in P and E may have a different schema than those in S, depending upon the set of attributes (and aggregate functions) included in the `SELECT` clause of the query. Given the above definitions and assumptions, the following relationship abstractly defines the evolution of $Q(\tau)$.

$$\forall\tau\ Q(\tau + 1) = Q(\tau) \cup P_S(\tau + 1) - E_S(\tau + 1)$$

That is, the updated answer of Q is obtained from the previous answer by adding the new result tuples produced and subtracting expired tuples. Using this relationship, individual persistent query operators may be divided into the following four types (determining the update patterns of complete query plans will be discussed in Section 3.5.1).

- An operator is monotonic if $\forall\tau\forall S\ E_S(\tau) = \emptyset$. That is, every result produced has a lifetime with unbounded length, regardless of the contents of the input stream(s).
- An operator is *weakest non-monotonic* if $\forall\tau\forall S\ \exists c \in \mathbb{N}$ such that $E_S(\tau) = P_S(\tau - c)$. That is, the expired results are exactly those which have been produced c time-ticks ago,

[1]That is, either non-decreasing timestamps are implicitly assigned by the system upon arrival or tuples are buffered and pushed into query plans in timestamp order.

irrespective of the input S. Hence, the lifetime of each result is known at generation time and furthermore, all the lifetimes have the same finite length c.

- An operator is *weak non-monotonic* if $\forall\tau$ and $\forall S, S'$ such that $S(0,\tau) = S'(0,\tau)$, it is true that $\forall t \in P_S(0,\tau)\ \exists r$ such that $t \in E_S(r) \wedge t \in E_{S'}(r)$. That is, every result tuple eventually expires and the expiration times of tuples that have already arrived do not depend upon tuples that will arrive in the future. In other words, the lifetime of each result tuple is known at generation time, but the lifetimes do not necessarily have the same length.
- An operator is *strict non-monotonic* if $\exists\tau\exists S, S'$ such that $S(0,\tau) = S'(0,\tau)$ and $\exists r \exists t \in P_S(0,\tau)$ such that $t \in E_S(r) \wedge t \notin E_{S'}(r)$. That is, at least some results have expiration times that depend upon future inputs and therefore cannot be predicted at generation time.

3.2.2 Discussion

The above classification identifies three types of non-monotonic operators according to their update patterns, with weakest non-monotonic operators having the simplest patterns, followed by weak and strict non-monotonic operators, respectively. Weakest non-monotonic operators produce results that expire at predictable times and in the same order in which they were generated, i.e., first-in-first-out (FIFO). Projection and selection over a time-based window are weakest non-monotonic, as is a merge-union of two time-based windows. For example, selection (see Figure 3.1(a)) drops tuples that do not satisfy the predicate, but does not alter the lifetimes of surviving tuples, which are initially set to one window length (recall Section 2.3.2). When the window slides forward and the oldest tuple with value a expires from its window, it also expires from the output stream in the same order.

Weak non-monotonic operators may not necessarily exhibit FIFO behaviour, but their results have predictable expiration times. A join of two time-based windows, as illustrated in Figure 3.1(b), is one example. A newly arrived tuple on one stream may join with many tuples from the other window, some of which have just arrived and others are about to expire. For instance, when the old S_2-tuple with value c expires, it causes the expiration of a recently produced output tuple with value c. Although the expiration order is not necessarily FIFO, the expiration time of each join result is the minimum of the expiration times of the individual tuples participating in the result (recall the discussion in Section 2.3.2).

The significance of weakest and weak non-monotonic operators is that their results have predictable expiration times and therefore only these types of operators are compatible with the direct approach to state maintenance from Section 2.3.3. On the other hand, strict non-monotonic operators produce at least some results with unpredictable expiration times and therefore negative tuples must be propagated to explicitly announce these expirations. For example, as discussed in Section 2.2.2, negation produces negative tuples to indicate that previously reported results no longer satisfy the query. These expirations are not caused by the movement of the sliding windows, as was the case with weakest and weak non-monotonic operators, but rather by the

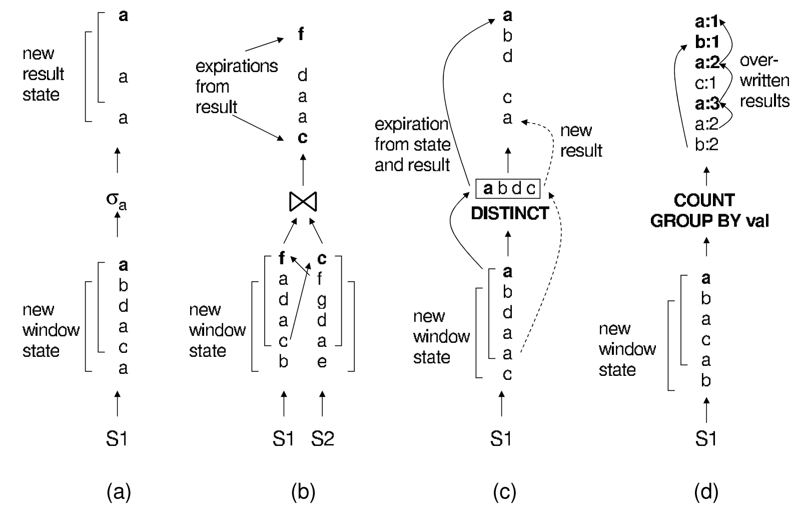


Figure 3.1: Update patterns of sliding window operators

semantics of the negation operator[2]. Thus, it may not be possible to determine when a result tuple will expire without knowing which stream tuples will arrive in the future. Note that some results of negation over time-based windows expire "naturally" when they fall out of the window and have not been previously invalidated by a negative tuple.

Duplicate elimination over a time-based window may be implemented as weak or strict non-monotonic. First, consider the example shown in Figure 3.1(c), which illustrates a weak non-monotonic implementation. A tuple with value a was produced on the output stream and now expires from its window. In order to guarantee weak non-monotonicity, this tuple must now expire from the result as well, i.e., its lifetime must end as predicted by its expiration timestamp. However, there exist other tuples with value a in the window. Therefore, in order to maintain the correct answer, one of these tuples, say the youngest, must now be appended to the output and will expire when its lifetime ends. Note that the insertion order into the result is different from the arrival order, as evidenced by the two newest result tuples with values c and a. This is because some tuples may not be appended to the output stream immediately after they arrive, therefore their lifetimes in the output may have different lengths.

A straightforward implementation of weak non-monotonic duplicate elimination stores its output in order to filter out duplicates, as well as the entire input—for instance, as described above, when the result tuple with value a expires, the input must be accessed in order to determine if any other tuples with value a are present in the window [125]. However, as will be shown in Section 3.5.2, it suffices to store only a certain subset of the input, no larger than the size of the stored output, and still guarantee weak non-monotonicity, provided that the inputs to the

[2]This implies that strict non-monotonic operators over time-based windows must also be strict non-monotonic over unbounded streams because time-based windowing alone does not produce premature expirations.

duplicate elimination operator are not strict non-monotonic.

Moreover, rather than storing a subset of the input, it is possible to track only the multiplicities of the distinct values present in the input. However, this type of implementation requires negative tuples on the input stream (or the storage of the entire input, as in the straightforward weak non-monotonic implementation), which means that the underlying window must be stored by the query plan anyway (recall Section 2.3.3). Furthermore, negative tuples are produced on the output stream, meaning that the update patterns of this implementation of duplicate elimination are strict non-monotonic. Such an implementation produces new results whenever new tuples arrive with never-before-seen values, as before. Additionally, duplicate tuples cause the corresponding counters to be incremented, whereas expired tuples decrement the appropriate counters. Finally, a negative tuple is appended to the output stream when a tuple expires from the input and causes the corresponding counter to become zero. Hence, a result tuple does not expire unless there are no other tuples with the same value in the current window. This means that the lifetimes of result tuples cannot be predicted ahead of time, which is why negative tuples must be produced on the output. The strict non-monotonic implementation of duplicate elimination is used in the STREAM DSMS, which employs the negative tuple approach to query processing and expiration.

Finally, note that group-by with aggregation as well as all operators over count-based windows considered in this chapter are strict non-monotonic. The former, illustrated in Figure 3.1(d), produces updated aggregate values as new tuples arrive and old tuples expire. However, it is unknown if or when a new value will arrive on the stream that will change the value of the aggregate. Consequently, it is not possible to predict the expiration time of the current aggregate value. In terms of count-based windows, note that the oldest tuple expires when the first new tuple arrives on the stream. Hence, expiration times depend upon the arrival times of subsequent tuples and therefore negative tuples are needed to signal expirations.

3.3 Update-Pattern-Aware Semantics of Persistent Queries

This section defines the semantics of persistent queries with the help of the proposed update pattern classification. Starting with Definition 2.1, the meaning of relations in persistent queries is clarified in Section 3.3.1. Next, Section 3.3.2, divides sliding windows into two types—external and internal—with the latter used when the desired behaviour of the query is monotonic, but the system defines windows on the inputs due to memory constraints. Section 3.3.3 motivates and proposes an alternate definition of the current result of a query, which defines a window on the output stream. Finally, Section 3.3.4 shows that weak and strict non-monotonic queries do not always produce a full answer set if they are re-evaluated periodically.

3.3.1 Defining the Meaning of Relations in Persistent Queries

In a DBMS, a relation is an unordered multi-set of tuples with the same schema that supports arbitrary insertions, deletions, and updates. In a DSMS, a relation may be referenced in a query by being joined with a stream or a sliding window. However, unless relations are assumed to be static, the possibility of arbitrary updates means that finite relations appear more difficult to deal with than (time-based) sliding windows. To see this, consider an operator that joins a relation with a time-based window, call it $\bowtie^R$. According to Definition 2.1, an insertion into the table requires a window scan in order to produce any new join results. This is because the current output must correspond to the current state of the input streams and relations at all times. Similarly, a deletion from the table requires a window scan in order to undo previously reported join results containing the deleted tuple, if any. That is, negative tuples must be produced on the output stream so that the result corresponds to the current state of the relation. As a result, $\bowtie^R$ is strict non-monotonic, whereas a join of time-based windows is only weak non-monotonic.

According to the update pattern classification, any solution that considers updates of relations to be "easier" than insertions and expirations from sliding windows must treat $\bowtie^R$ as weakest non-monotonic. Given this constraint, the solution presented here defines a *non-retroactive relation (NRR)* as a table that allows arbitrary updates, but has the following semantics: updates of *NRR*s do not affect previously arrived stream tuples. Consequently, a join of a sliding window and a *NRR*, denoted $\bowtie^{NRR}$, does not need to scan the window when processing an update of the *NRR*; only the incoming stream tuples trigger the probing of the *NRR* and generation of new results. Thus, the streaming input does not have to be stored, and furthermore, $\bowtie^{NRR}$ is monotonic if the second input is a stream and weakest non-monotonic if it is a time-based window.

Aside from being simpler to implement, the definition of *NRR*s is intuitive based upon the nature of some of the data stored by DSMSs in relations, namely metadata. For example, an on-line financial ticker may store a table with mappings between stock symbols and company names. In this case, when a financial ticker updates its table of stock symbols and company names by deleting a row corresponding to a company that is no longer traded, all the previously returned stock quotes for this company need not be deleted. Similarly, adding a new stock symbol for a new company should not involve attempting to join this stock symbol with any previously arrived stream tuples, because there are no prior stock quotes for this new company. Formally, an update of a *NRR* at time τ affects only those stream tuples which arrive after time τ.

One restriction that must be made in the context of *NRR*s involves deleting a row of a table and subsequently adding a row having the same key. For example, if a company is removed from the *NRR* but its stock quotes are not deleted from the result, then an insertion of a different company with the same key makes it appear that the old stock quotes refer to the new company. One way to solve this problem involves banning the re-use of keys in an *NRR*.

Note that the difference between streams, *NRR*s, and relations is strictly semantic. It is possible to treat metadata as a stream if insertions are to be retroactive to previously arrived tuples, or as a traditional relation if arbitrary insertions, deletions, and updates are to affect previously arrived stream tuples. However, as explained above, the update patterns and implementations of operators that allow retroactive updates are more complicated.

The revised definition of the semantics of persistent queries is as follows. A persistent query Q references one or more streams (possibly bounded by sliding windows), zero or more *NRR*s, and zero or more relations, runs over a period of time, and produces the following output.

Definition 3.1 *Let $Q(\tau)$ be the answer set of Q at time τ and let $\{\mathrm{NRR}_1(\tau), \mathrm{NRR}_2(\tau), \ldots, \mathrm{NRR}_k(\tau)\}$ be the state of each of the k NRRs referenced in Q at time τ. Let t.ts be the timestamp of a result tuple t. If Q does not reference any NRRs, i.e., $k = 0$, then $Q(\tau)$ is equivalent to the output of a corresponding relational query Q' whose inputs are the current states of the streams, sliding windows, and relations referenced in Q. If $k > 0$, then in addition to the above, each result tuple t in $Q(\tau)$ must reflect the following state of the NRRs referenced in Q: $\{\mathrm{NRR}_1(t.ts), \mathrm{NRR}_2(t.ts), \ldots, \mathrm{NRR}_k(t.ts)\}$.*

Note that Definition 3.1 includes situations in which a query arrives for processing, but the system has not been maintaining the window of interest to the query (either because no other queries reference this stream or all queries referencing this stream specify shorter windows). Then, the "current state" of a sliding window may be initially empty, until it grows to the required size and begins to slide forward. However, if an appropriate window is maintained by the system, then Definition 3.1 assumes that the query can start producing answers right away over the window, in effect having access to "historical" data that has arrived before the query was registered. A similar assumption is made in [2, 49], but not in [183], where new persistent queries can only reference future tuples.

3.3.2 Internal and External Windows

Definition 3.1 implies that all sliding window operators are non-monotonic. This may be too restrictive if windows are used only to manage the "unboundedness" of the input streams. In this case, the fact that a tuple expires from its window does not necessarily imply that it also expires from the output. This issue may be resolved by defining the following two types of sliding windows (both can be time-based or count-based).

- *External* windows are used when the output is meant to conform to Definition 2.1, e.g., if the user or the application semantics specifically request that old results must be removed from the result as the windows slide forward. All operators over external windows are (weakest, weak, or strict) non-monotonic.
- *Internal* windows are used when the output is meant to be monotonic. The semantics are such that an operator over internal windows produces the same results in the same order as in the case of external windows, but the output stream is assumed to be append-only. That is, the lifetimes of the results have unbounded length.

One example of internal windows involves a join of two streams. As compared to the unbounded stream join from Figure 2.1(b), internal windows allow the join to be executed in finite memory by producing only those results where the matching tuples are within one window size of one another (i.e., expired tuples must be removed from the inputs, as before). Note that a join of internal windows produces the same output in the same order as the join of external windows illustrated in Figure 3.1(b), but its results do not expire. Furthermore, a join of n time-based windows is weakest non-monotonic if at most one of them is external. If at least two windows

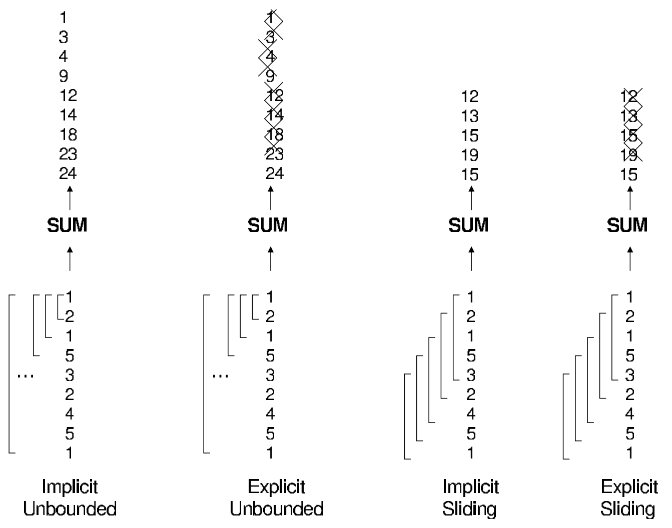


Figure 3.2: Output of an aggregation query over four types of windows

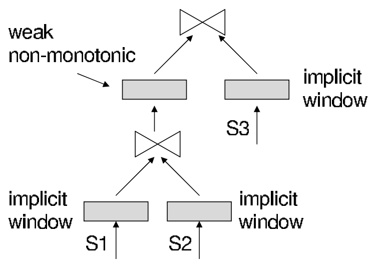


Figure 3.3: Processing a join of internal windows using a materialized view

are (time-based and) external, then the join is weak non-monotonic. If the join references only unbounded streams and internal time-based windows, then it is monotonic.

Internal and external windows may also be used with grouping and aggregation, with internal windows producing an append-only output stream. For instance, the outputs in Figure 3.1(d) would not expire if internal windows were used. Moreover, unbounded external and internal windows may be defined for aggregation over streams, giving rise to four types of windows: internal sliding, external sliding, internal unbounded, and external unbounded. In turn, this defines four types of semantics of aggregate queries, as shown in Figure 3.2; the query is a `SUM` over an unbounded stream or over a count-based window of size five. Note that only external windowing causes a previously reported aggregate value to be removed (crossed out) from the result set when a new value is computed.

Note that internal windowing does not apply to duplicate elimination and negation because forcing these operators into monotonic form violates their semantics. For example, if results are not deleted when they expire from their windows, then the output stream may contain duplicates or results that do not satisfy the negation predicate.

The last issue regarding internal windows discussed here deals with composability of operators. Recall that an operator over internal windows must produce the same result stream as an equivalent operator over external windows. Therefore, even though the output of an operator over internal windows is monotonic, its state must be maintained as if external windows were used. That is, the state must be maintained continually by removing tuples that have expired from their windows. For example, Figure 3.3 illustrates that evaluating a three-window join by materializing the join of the first two means that expirations from these two windows must be reflected in the materialized sub-result (i.e., weak non-monotonic update patterns, assuming that the windows are time-based). Otherwise, new S_3-tuples could join with old $S_1 \bowtie S_2$ results.

3.3.3 Windowing the Output Stream of a Persistent Query

Similar to using internal windows over the inputs to bound the memory requirements of query operators, the size of the result may be bounded by windowing the output stream. The semantics of windowing the output of a persistent query are as follows. For a count-based output window of size N, the answer consists of the N most recently generated results[3]; for a time-based output window of size T, the answer consists of all the results generated in the last T time units.

Note that the size of the input window(s) is independent of the size of the output window. For example, a one-minute window may be defined over the result stream of a join of five-minute windows. In fact, a count-based output window may be defined over the result of a query that references time-based windows, and vice versa [179]. For instance, a count-based output window may be used if a query is interested in recent values produced by an aggregate query over an internal time-based window (recall Figure 3.2).

In principle, the output stream of any type of query may be windowed. However, it may not make sense to do this for negation and duplicate elimination for the same reason why these operators are incompatible with internal windows (recall Section 3.3.2). That is, the windowed output may contain tuples that violate the operator semantics. For example, the output stream of the `DISTINCT` operator in Figure 3.1(c) contains two a-tuples, one of which has expired; however, a sufficiently large window on the output stream could span both of these a-tuples. Moreover, the answer produced by windowing the output typically does not conform to Definition 3.1, except for weakest non-monotonic operators. To see this, recall that result lifetimes of weak non-monotonic operators all have length T, which is equivalent to placing a time-based window of size T on the output stream (see Figure 3.1(a)). In contrast, results of weak non-monotonic operators remain in the answer set for a time of at most T, therefore an output window of size T yields a superset of the results produced in accordance with Definition 3.1.

As in Section 3.3.2, note that only the output of an operator may be windowed and intermediate state must be maintained as per Definition 3.1. In terms of Figure 3.3, this means that the materialized result of $S_1 \bowtie S_2$ must always reflect the current state of the two input windows (i.e., it is not correct to window its output stream), even if the final result stream is to be windowed.

3.3.4 Impact of Update Pattern Awareness on Periodic Query Processing

Suppose that Definition 3.1 is modified such that the result of a persistent query is updated periodically rather than reflecting the current state of the inputs at all times. Assume that the interval between two query re-executions, call it Δ, is smaller than the size of the window(s) that the query references (i.e., two consecutive re-executions reflect overlapping window states). Periodic evaluation is likely to be more efficient, but has the following drawbacks.

[3]If more than N results are generated at the same time, then ties are assumed to be broken arbitrarily. Therefore, the semantics of a count-based output window may be non-deterministic.

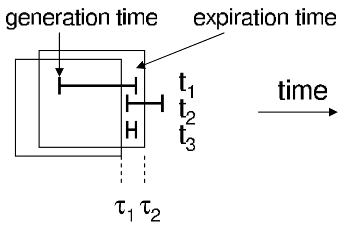


Figure 3.4: Drawbacks of periodic query re-evaluation

- If a tuple arrives during the period between two re-evaluations, then any results that it generates are not reported until the next re-evaluation.
- If a tuple expires during the period between two re-evaluations, then its expiration is not reflected in the answer set until the next re-evaluation.
- If a result tuple is produced between two re-evaluations and its lifetime is shorter than Δ, then it will never be reported in the answer set.

These effects are illustrated in Figure 3.4, with results of a persistent query represented as intervals corresponding to their lifetimes. Suppose that two consecutive re-evaluations of the query occur at times τ_1 and τ_2 (that is, $\Delta = \tau_2 - \tau_1$). At time τ_1, the output contains only tuple t_1 as the other two have not yet arrived. The arrival of t_2 is not reported until time τ_2, when the next re-evaluation occurs. Similarly, the expiration of t_1 is not reflected in the answer until time τ_2. The short-lived tuple t_3 is not in the result set at time τ_1 (it has not arrived yet) or at time τ_2 (by then, it has expired). Notably, only weak and strict non-monotonic operators suffer from the missing result problem as the lifetimes of the results of weakest non-monotonic operators are all equal to the window size, which, as assumed above, is longer than Δ.

Short-lived results are generated by the sliding window join when a newly arrived tuple joins with a tuple that is about to expire. The same occurs in duplicate elimination if a result tuple with value v expires and there is only one other v-tuple in the window that will expire soon thereafter. In the context of aggregation, missing results are the unreported values of the aggregate between re-evaluations[4]. Finally, sliding window negation, $W_1 - W_2$, can introduce short-lived results in two ways. First, a new result with value v may be generated and invalidated shortly afterwards if a tuple with value v appears in W_2. Second, a W_2-tuple with value v may expire and cause an old W_1-tuple (which is about to expire) with value v to be added to the result.

It may be argued that delayed updating of the result is an expected and reasonable side-effect of periodic re-evaluation, but missing answers are more serious. This leads to two types of semantics of periodically re-evaluated queries. *No-restore* semantics ignore missing results and are equivalent to Definition 3.1, in which case each re-evaluation appends all the newly generated

[4]There may be a very large number of intermediate aggregate values that are never reported—the value may change whenever a new tuple arrives or an old tuple expires from its window. This is the main reason why sliding window aggregation is much more efficient when executed periodically, as outlined in Section 2.3.4.

results to the output stream. *Restore* semantics force the output stream to include all the results that would be produced by continual re-evaluation. Thus, in addition to appending new results to the output stream upon re-evaluation, all the short-lived results that were generated since the last re-evaluation (and have already expired) are reported during the subsequent re-evaluation. For instance, in Figure 3.4, the short-lived result would be reported at time τ_2. Join processing algorithms for restore and no-restore semantics will be presented in Chapter 5.

3.4 Making a DSMS Query Language Update-Pattern-Aware

Having proposed several clarifications of persistent query semantics—*NRR*s, internal windows, output windows, and restoring lost answers in periodic re-evaluation—this section incorporates these concepts into a continuous query language. To be specific, CQL [12] was chosen as the base language, but the extensions may be applied to other languages as well.

Internal and external windows can be represented in CQL as two types of stream-to-relation operators. In fact, the notion of internal windows generalizes other stream-to-relation functions that are meant to produce monotonic output; k-constraints [28] and punctuations [245] are two examples. Internal windows may be denoted with the keyword `INTERNAL`, for example, `[RANGE 1 min INTERNAL]`. As discussed, a join of internal windows is monotonic, therefore the `Istream` operator may be omitted if the `INTERNAL` keyword is used in the window specification.

Windowing of the output stream may be modeled as a new relation-to-stream function, call it `Wstream`. For example, a query that maintains the last ten results of a time-based window join could be specified as follows (analogous to the `RANGE` keyword, the `ROWS` clause is used to specify count-based windows).

```
SELECT Wstream(*)[ROWS 10]
FROM Stream1[RANGE 1 min], Stream2[RANGE 1 min]
WHERE S1.a = S2.a
```

Note that the windows do not need to be declared as `INTERNAL` because `Wstream` assumes that the output is monotonic and may be windowed.

One advantage of the `Wstream` operator is that it facilitates an optimization strategy for periodically re-evaluated distributive aggregates [167] without using nested sub-queries. For instance, to compute a sum of attribute a over a one-minute sliding window that is re-evaluated every six seconds, it suffices to split the stream into non-overlapping six-second chunks, compute the sum over each chunk, and add up the sum of the last ten chunks to get the final answer. Without `Wstream`, this query requires a nested sub-query and may be posed as follows (assume that the final output is non-monotonic, meaning that only the latest value of the aggregate ought to be returned; the `Rstream` keyword may be omitted because the count-based window on the output is not declared as internal).

```
SELECT SUM(a)
FROM (SELECT SUM(a) FROM Stream1[RANGE 6 sec SLIDE 6 sec INTERNAL]) [ROWS 10]
```

That is, each six-second interval in the inner sub-query is summed up, which produces a monotonic output stream because it operates over an internal window. Next, the outer query computes the sum over a count-based window of size ten defined on the output of the inner sub-query; the outer window is not internal, therefore the final output is non-monotonic. Note that specifying an external window in the inner sub-query would give a syntax error as it is not possible to window a materialized output that always consists of only one (latest) aggregate value. Using `Wstream`, the above query simplifies to the following.

```
SELECT SUM(Wstream(SUM(a))[ROWS 10])
FROM Stream1[RANGE 6 sec SLIDE 6 sec]
```

Again, the type of window does not have to be specified in the `FROM` clause; using the `Wstream` operator implies that the input window is internal.

CQL allows periodic query re-execution using the `SLIDE` clause in the input window specification, e.g., `[RANGE 1 min SLIDE 10 sec]`. To incorporate restore semantics, two relation-to-stream functions may be added: `Istream-restore` and `Rstream-restore`. For example, a periodically re-evaluated join of external windows with restore semantics could be specified as follows (assuming that the `SLIDE` interval is the same for all windows referenced in a query).

```
SELECT Rstream-restore(*)
FROM Stream1[RANGE 1 min SLIDE 10 sec],
     Stream2[RANGE 1 min SLIDE 10 sec]
WHERE S1.a = S2.a
```

Note that `Wstream-restore` is not defined because windowing of the output stream is not compatible with periodic re-evaluation. For instance, if a count-based output window of size N is used and the next periodic re-evaluation produces more than N results, then it is not clear which N newly generated results should be included in the output window at this time.

3.5 Update-Pattern Aware Query Processing and Optimization

This section presents an update-pattern-aware query processor and optimizer. The goal is to decrease processing times and reduce the state maintenance overhead. Existing techniques only satisfy one of these requirements: the negative tuple approach performs state maintenance efficiently, but performs twice as much processing, whereas the direct approach does not incur processing overhead, but performs state maintenance inefficiently. The technique described in this section satisfies both goals by exploiting the update patterns of query operators. In the remainder of this chapter, all windows and operator state are assumed to fit in main memory.

3.5.1 Update Pattern Propagation in Persistent Query Plans

The first step is to define the update patterns of persistent query plans based upon the update characteristics of individual operators. Let *WKS*, *WK*, and *STR* denote weakest, weak, and strict non-monotonic update patterns, respectively. The edges of a query plan are labeled with update pattern aware information as follows. The edges originating at the leaf nodes (i.e., base windows) are *WKS* if the window is time-based and *STR* if the window is count-based. The remaining edges are labeled using the following five rules (the update pattern of the complete query plan is the label of its final output edge).

1. The output type of unary weakest non-monotonic operators and $\bowtie^{NRR}$ is the same as the input type.
2. The output of binary weakest non-monotonic operators is *STR* if at least one if their inputs is *STR*, *WK* if the inputs are either *WKS* or *WK*, and *WKS* if both inputs are *WKS*.
3. The output of weak non-monotonic operators is *STR* if at least one of their inputs is *STR*. Otherwise, the output is *WK*.
4. The output of strict non-monotonic operators and $\bowtie^{R}$ is always *STR*.

Rule 1 follows from the fact that weakest non-monotonic operators do not interfere with the order of incoming tuples. Rule 2 applies to merge-union, which also does not reorder incoming tuples, therefore the output patterns correspond to whichever input patterns are more complex. Rule 3 states that *WK* operators produce update patterns at least as complex as *WK*, and possibly *STR* if the inputs already contain premature expirations. Finally, Rule 4 follows from the fact that count-based windows, *STR* operators, as well as relations whose updates are retroactive generate results with unpredictable expiration times.

An example of an annotated query plan containing selections, joins, and negation over time-based windows is shown in Figure 3.5. A possible CQL query that may produce these plans may have the following structure (selection conditions and window size specifications have been omitted for clarity; the `Rstream` operator is used to maintain a materialized view of the result).

```
SELECT Rstream(*)
FROM S1, S2
WHERE S1.a = S2.a
AND NOT EXISTS (
   SELECT Rstream(*)
   FROM S3
   WHERE S3.a = S1.a)
```

Two equivalent rewritings of the query are depicted. Observe that the two rewritings result in different update patterns on some of the edges; this issue will be revisited in the context of query optimization in Section 3.5.3. Moreover, note that an update pattern is a property of a particular query plan rather than the query itself. For instance, if it is possible to rewrite the plans from Figure 3.5 in such a way as to optimize away the negation operation, then their update patterns may no longer be strict non-monotonic.

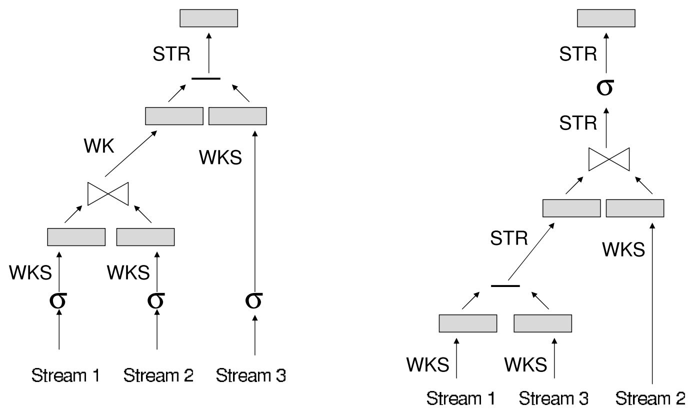


Figure 3.5: Two query plans (for the same query) annotated with update patterns

3.5.2 Update-Pattern-Aware Physical Plan Generation

Given a query plan annotated with update patterns, two strategies are used in the generation of a physical query plan: operator implementations that depend upon the update patterns of their inputs, and update-pattern-aware data structures for maintaining operator state.

Operator Implementation

Recall from Section 3.2.2 that a straightforward implementation of weak non-monotonic duplicate elimination stores both its input and its output [125]. If the update patterns of the input are *WKS* or *WK*, then a more efficient implementation (both in terms of time and space complexity) is possible. The idea is to avoid storing the entire input if premature expirations are guaranteed not to occur. Instead, for each tuple in the output state, it suffices to additionally store the youngest tuple with the same distinct value (if any); this additional state will be referred to as *auxiliary output state*. When a new tuple arrives and does not match any tuples in the stored output, it is inserted into the output state and appended to the output stream, as before. However, if the new tuple is a duplicate, it means that it is the youngest tuple with its particular distinct value, and is added to the auxiliary output state. When an output tuple expires, the auxiliary output state is probed and the youngest tuple with the same distinct value is appended to the output, if one exists. Thus, instead of storing both the input and the output, the space requirement of the improved operator, call it δ^*, is at most twice the size of the output. Since duplicate elimination never produces an output whose size is larger than the input, δ^* is more space-efficient than the naive weak non-monotonic implementation. Moreover, the time overhead of inserting and expiring tuples is lower because the entire input is never scanned.

The physical operator δ^* does not work with *STR* update patterns on its input because tuples stored in the auxiliary state may expire early and therefore it may not be possible to determine which distinct values are still present in the window without access to the whole input. In

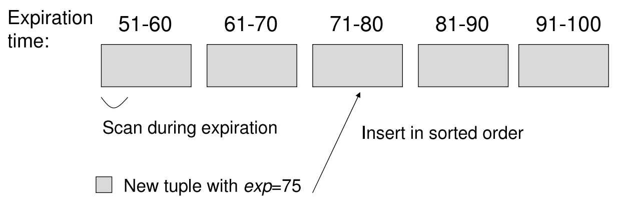


Figure 3.6: Storing the results of weak non-monotonic subqueries

this case, the two choices are to use the naive implementation of weak non-monotonic duplicate elimination or the strict non-monotonic implementation outlined in Section 3.2.2. The latter produces fewer "positive" tuples (because a result does not expire unless there are no more tuples with the same distinct value present in the input), but must also generate negative tuples on the output because its results expire at unpredictable times[5]. Note that if the input to the duplicate elimination operator is already *STR*[6], then all of the subsequent operators must be prepared to deal with negative tuples anyway. Therefore, a reasonable heuristic is to employ the strict non-monotonic version in this case.

Data Structures for Storing Operator State

The second update-pattern-aware physical strategy involves using suitable data structures for maintaining state. In the simplest case of *WKS* update patterns, the state buffer is a FIFO queue (a count-based window or the result of a selection over a count-based window may also be stored this way). However, if the input is *WK*, then the insertion order is different from the expiration order. Observe that if the state buffer is sorted by insertion time, then deletions are inefficient (the entire buffer must be scanned in order to find expired tuples). On the other hand, sorting by expiration time means that insertions require a sequential scan of the state buffer. One solution is to partition the state buffer by expiration time, effectively forming a calendar queue [40] if stream tuples are thought of as events scheduled according to their expiration times. Individual partitions (i.e., the "days" in the calendar queue) can then be sorted by expiration time for operators that must expire results eagerly, or by insertion time for operators with lazy expiration. An example is illustrated in Figure 3.6 for five partitions (with each partition sorted by expiration time), assuming that the current time is 50 and the window size is 50. This version of the calendar queue is a circular array of partitions, therefore at time 60, the left-most partition will contain tuples with expiration times of 101 through 110. The calendar queue may be used, for example, to store the left input to the negation operator on the left of Figure 3.5.

[5]The total number of result tuples (positive and negative) produced by the two variants of duplicate elimination depends upon the distribution of attribute values in the stream and the sizes of the windows. For example, if the stream contains few distinct values that are always present in the window, then the strict non-monotonic implementation is likely to produce a set of initial results, which will not be invalidated for a long time.

[6]That is, a strict non-monotonic operator, such as negation or a count-based window, is present in the query plan ahead of duplicate elimination.

Maintaining the results of (sub)plans with *STR* patterns is difficult because some result tuples may expire at unpredictable times. If premature expirations are rare, then the calendar queue may be used, with the understanding that occasional premature expirations triggered by negative tuples will require a sequential scan of all the partitions. Otherwise, if the majority of expirations are expected to occur via negative tuples, then it may be beneficial to employ the negative tuple approach and implement state buffers as hash tables on the primary key of the stream schema. That is, negative tuples are generated for every expiration. The intuition is that if most of the results expire prematurely, then the system may as well expire all the results via negative tuples and use a data structure that makes it easy to do so. The choice between these two techniques depends upon the frequency of premature expiration, which, in turn, depends upon the distribution of attribute values in the inputs. For example, if the two inputs to a negation operator have different sets of values of the negation attribute, then premature expirations never happen. To see this, note that negative tuples are produced by the negation operator only if both inputs contain at least one tuple each with a common attribute value (recall Section 2.2.2). On the other hand, if the plan references one or more count-based windows, then all expirations must be signaled with negative tuples and therefore the negative tuple approach is used exclusively.

One exception to the above rule involves aggregation and group-by. In this case, the result consists of aggregate values for each group and may be stored as an array indexed by group label. Moreover, negative tuples are not necessary. Instead, when a new aggregate value for some group is produced, it is understood to replace the old aggregate value for this group.

3.5.3 Update-Pattern-Aware Query Optimization

As in traditional DBMSs, there may be multiple ways of evaluating a given persistent query, and an efficient plan may be chosen by estimating and comparing the costs of a set of candidate plans (using a DSMS-specific per-unit-time cost model that accounts for operator processing, state maintenance, and negative tuple processing, if applicable [147]). Candidate plans may be derived via DBMS-style algebraic rewritings, e.g., selection push-down, duplicate elimination push-down, and join re-ordering. However, one constraint is that the input of $\bowtie^R$ and $\bowtie^{NRR}$ cannot be strict non-monotonic, therefore it is not possible to push these through a negation. This is because a join involving a relation or a *NRR* is incapable of processing negative tuples—the "real" tuple that corresponds to the negative tuple may have been deleted or updated in the relation, therefore it may not be possible to reproduce the join results involving the negative tuple.

In addition to the above, a novel set of rewritings is employed: update pattern simplification. The idea is to push down operators with simple (weakest non-monotonic) update patterns and pulls up those with more complicated update patterns (particularly strict non-monotonic). This is done to minimize the number of operators adversely affected by negative tuples and more generally, to reduce the update pattern complexity in the largest possible sub-tree of the plan. Other benefits of update pattern simplification include being able to use δ^* more often and greater flexibility in reordering $\bowtie^R$ and $\bowtie^{NRR}$.

For the most part, update pattern simplification is consistent with relational optimization rules. For example, pushing down weakest non-monotonic operators coincides with predicate push-down. This similarity suggests that update pattern awareness can be easily incorporated into relational optimizers. One difference is that relational optimizers typically push down the negation operator if the negation condition is a simple predicate or if it reduces the cardinality of intermediate results. However, in the context of update pattern awareness, it may, in some cases, be cheaper to pull up the negation operator in order to decrease the burden of processing negative tuples. For example, the plan on the left of Figure 3.5 may be more efficient than the one on the right because only the final materialized result needs to deal with negative tuples (the remainder of the plan uses the direct expiration approach using update pattern aware data structures from Section 21). Of course, if the join generates a large number of results, but the negation predicate reduces the cardinality of intermediate results, then the plan on the right may turn out to be less costly, even though there are more negative tuples flowing through more operators. Hence, an update pattern aware optimizer should consider both possibilities.

3.6 Experiments

3.6.1 Overview

An update-pattern-aware query processor, abbreviated as *UPA*, was implemented in Java. For comparison, the negative tuple and direct approaches from [125, 127] were also implemented, and are referred to as *NT* and *DIRECT*, respectively. Sliding windows and state buffers are implemented as linked lists (FIFO queues) or circular arrays of linked lists (calendar queues). Testing was performed on a Windows XP machine with a Pentium IV 1.8 Ghz processor and 512 Mb of RAM. Query inputs consist of network traffic data obtained from the Internet Traffic Archive (`http://ita.ee.lbl.gov`). The packet trace used in the experiments contains all wide-area TCP connections between the Lawrence Berkeley Laboratory and the rest of the world between September 16, 1993 and October 15, 1993 [200]. Each tuple in the trace consists of the following fields: system-assigned timestamp *ts*, expiration timestamp *exp* (recall Section 2.3.2), session duration, protocol type, payload size, source IP address, and destination IP address. Furthermore, negative tuples contain a special flag. Although the trace may be thought of as a single stream, it is broken up into several logical streams based upon the destination IP addresses. This simulates different outgoing links and is used in queries containing joins.

Four types of query plans are tested, the first three of which are illustrated in Figure 3.7 and the last is as shown in Figure 3.5. Query 1 joins tuples from two outgoing links on the source IP address, with the selection predicate being either *protocol=ftp* or *protocol=telnet*. The former is a selective predicate (the result size is approximately equal to the size of the inputs), whereas the latter produces ten times as many results (telnet is a more popular protocol type in the trace). Query 1 tests the performance of the calendar queue and may be posed as follows in CQL (the window lengths vary throughout the experiments).

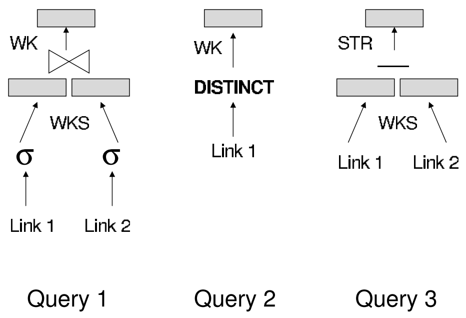


Figure 3.7: Illustration of the first three query plans used in the experiments

```
SELECT Rstream(*)
FROM Link1 [RANGE ...], Link2 [RANGE ...]
WHERE Link1.source_IP_address = Link2.source_IP_address
AND Link1.protocol = ftp
AND Link2.protocol = ftp
```

Query 2 selects the distinct source IP addresses (or the distinct source-destination IP pairs) on an outgoing link and is used to test the δ^* operator as well as the calendar queue. In CQL, it may be posed as follows (again, several different window lengths are tested).

```
SELECT Rstream(DISTINCT source_IP_address)
FROM Link1 [RANGE ...]
```

Query 3 performs a negation of two outgoing links on the source IP address and tests the two possible choices for storing the results of strict non-monotonic queries: using the calendar queue or using the negative tuple approach (recall Section 21).

```
SELECT Rstream(*)
FROM Link1 [RANGE ...]
WHERE NOT EXISTS (
   SELECT Rstream(*)
   FROM Link2 [RANGE ...]
   WHERE Link2.source_IP_address = Link1.source_IP_address)
```

Finally, Query 4 performs a negation of two outgoing links on the source IP address and joins a third link on the source IP address having *protocol* $= ftp$. That is, Query 4 is essentially a composition of queries 1 and 3. Both rewritings of Query 4 illustrated in Figure 3.5 are tested in order to show that negation pull-up may be efficient in some situations.

For simplicity of implementation, each incoming tuple is fully processed before the next tuple is scheduled for processing. As a result, the stream arrival rates are fixed and queuing delays caused by bursts of tuples arriving at the same time are ignored; this has been discussed in the context of data stream scheduling [20, 46, 142] and is an orthogonal issue.

There are four experimental parameters: sliding window size, lazy expiration interval (for operators that maintain state lazily), eager expiration interval (for operators such as grouping, duplicate elimination, and negation, which must react to expirations immediately), and the number of partitions in the calendar queues and hash tables. Depending upon the query, the window size varies anywhere from 100 Kilobytes to over 10 Megabytes. In terms of time, this corresponds to a range of 2000 to 200000 time units, with an average of one tuple arriving on each link during one time unit. For simplicity, the lazy expiration interval is set to five percent of the window size. Increasing this interval gives slightly better performance and is not discussed further. Furthermore, due to the fixed stream arrival rates, the eager expiration interval is set to the tuple inter-arrival time[7]. In *NT*, this means that each new arrival into one of the input windows triggers a window scan to determine if any negative tuples must be generated. In *DIRECT* and *UPA*, each new arrival causes a probe of the state of each operator that must immediately react to expirations. Finally, the number of state buffer partitions is set to 10, unless otherwise noted. The reported performance figures correspond to the average overall query execution times (including processing, tuple insertion, and expiration) per 1000 tuples processed after all the windows have filled up.

3.6.2 Query 1

Two variants of Query 1 are tested first. Figure 3.8 illustrates the performance of the first variant, which uses *protocol=ftp* as the selection predicate (recall that this is the more selective predicate that produces fewer results). As the window size grows, *UPA* is nearly twice as fast as the other two. *DIRECT* outperforms *NT* because the result size is relatively small and therefore the cost of scanning the entire result set during updates is not as great as the overhead of negative tuples. However, the performance of *DIRECT* degrades as the window size grows. The second variant of Query 1 is analyzed in Figure 3.9 and uses *protocol=telnet* as the selection predicate. The result size is approximately ten times as large as in the first variant. In this case, *DIRECT* is by far the slowest because it is very expensive to scan the large result when performing expiration. The update-pattern-aware approach with ten partitions (denoted by *UPA(10)*) initially performs well, but becomes very slow as the window size, and the result size, grows. However, increasing the number of partitions to fifty (denoted by *UPA(50)*) yields processing times that are up to one order of magnitude faster than *NT* for large window sizes.

3.6.3 Query 2

Figure 3.10 illustrates the processing times of duplicate elimination on the source IP address, whereas Figure 3.11 graphs the processing times of duplicate elimination on source and destination

[7]Technically, the expiration procedure does not actually delete old tuples as this is under the control of Java's garbage-collection mechanism. Experiments were performed with Java's System.gc() method, which acts as a suggestion for Java to perform garbage collection. As expected, the processing times of each technique tested here increased when this method was called frequently. Garbage collection is not discussed further as it is an implementation-specific issue.

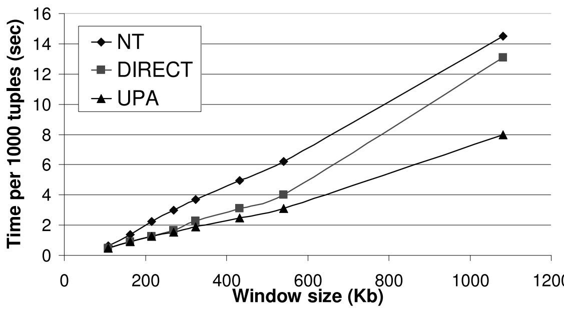


Figure 3.8: Processing times of Query 1 using $protocol = ftp$ as the selection predicate

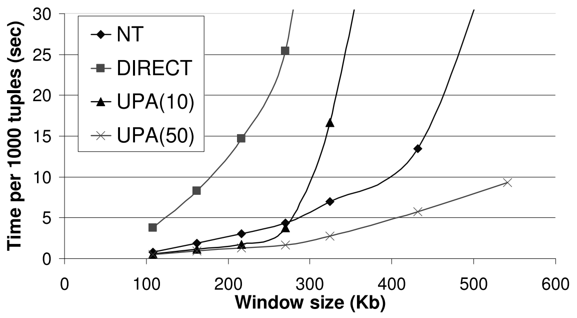


Figure 3.9: Processing times of Query 1 using *protocol=telnet* as the selection predicate

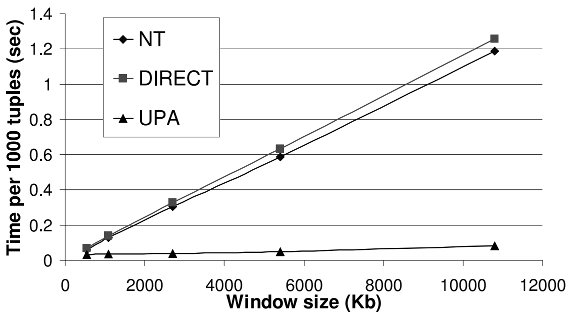


Figure 3.10: Processing times of Query 2 with duplicate elimination on the source IP address

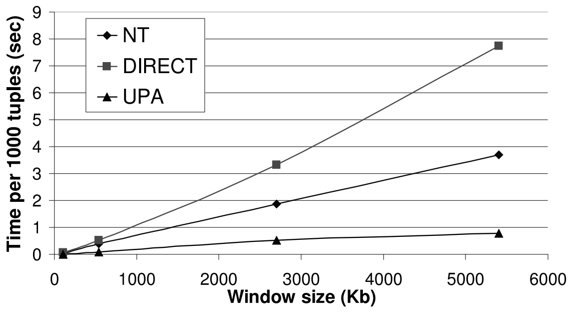


Figure 3.11: Processing times of Query 2 with duplicate elimination on source and destination IP address pairs

IP addresses. The former produces a small results set (roughly 2000 distinct IP addresses); the result set of the latter is approximately ten times as large. Combining the δ^* operator with the calendar queue for storing the result yields significant performance improvements. In Figure 3.10, *UPA* is one order of magnitude faster than the other two. In Figure 3.11, *UPA* is roughly twice as fast as *NT*, which was tested with the default weak non-monotonic version of duplicate elimination. Furthermore, *DIRECT* performs poorly when the result size is large. The reason why *UPA* has a greater performance advantage when the result size is small is because the size of the auxiliary output state is also smaller, and therefore it is faster to maintain and probe (recall Section 3.5.2).

The average space requirements of Query 2 are graphed in Figure 3.12 (duplicate elimination on the source IP address) and Figure 3.13 (duplicate elimination on source and destination IP addresses). The former is very selective, therefore *UPA* is up to two orders of magnitude more

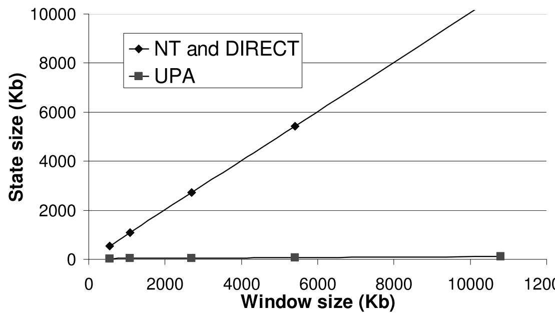


Figure 3.12: Space consumption of Query 2 with duplicate elimination on the source IP address

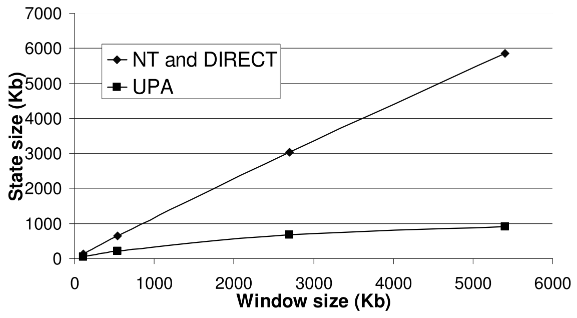


Figure 3.13: Space consumption of Query 2 with duplicate elimination on source and destination IP address pairs

space-efficient that *NT* and *DIRECT* (recall that the space requirements of δ^* are proportional to the output size, not the input size). The latter is less selective, but *UPA* is still significantly more space-efficient.

3.6.4 Query 3

Figure 3.14 shows the running time of negation on the source IP address for *NT* and *UPA*, with the latter employing a calendar queue to store the results and using negative tuples only for premature expirations. *UPA* slightly outperforms *NT* for window sizes of up to roughly 500 Kilobytes. This is because the result size is small and the penalty for scanning the entire result buffer when expiring a negative tuple is lower than the overhead of generating negative tuples. However, as the window size (and the result size) grows, *UPA* begins to perform worse because the cost of expiring negative tuples becomes high. In this experiment, the fraction of premature expirations was counted explicitly and came to approximately 40 percent. This is a fairly high proportion, which explains why *UPA* was competitive only for small window sizes.

3.6.5 Query 4

The final test evaluates the two plans for Query 4 illustrated in Figure 3.5. As discussed in the context of Query 3, negation on the source IP address produces a relatively large number of premature expirations and works best with the negative tuple approach. In this case, *UPA* pulls up the negation operator (recall Section 3.5.3) and uses the negative tuple approach only for those operators which follow negation in the query plan. That is, the plan on the left of Figure 3.5 is executed by maintaining the intermediate state directly and maintaining the final result via negative tuples generated by the negation operator (the join operator does not have to generate or process negative tuples). In the plan on the right, negation is pushed down, therefore *UPA* is

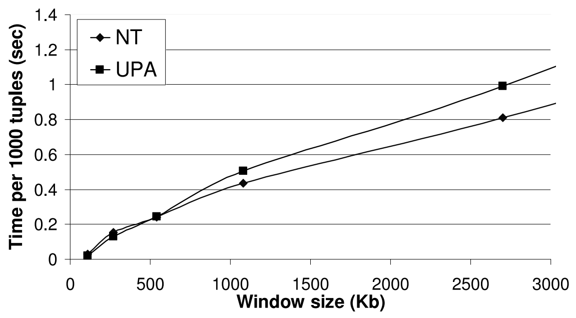


Figure 3.14: Processing times of Query 3

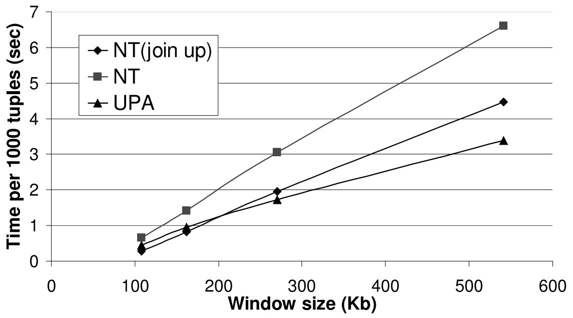


Figure 3.15: Processing times of Query 4

equivalent to the negative tuple approach throughout the plan. Figure 3.15 shows the processing times of Query 4 for the three possible approaches: negative tuples with the negation operator pulled up, negative tuples with the join pulled up (denoted *NT(join up)*) and negation pushed down, and *UPA* with negation pulled up.

First, note that *NT(join up)* outperforms *NT* because the negation operator is more selective than the join and therefore the former plan costs less if both are using the negative tuple approach. However, *UPA* performs best for sufficiently large window sizes because of the decrease in the number of negative tuples that have to be generated and processed. Note that *NT(join up)* is optimal for small window sizes because the number of negative tuples generated is small and their processing overhead is not as large as the penalty of using a sub-optimal ordering. However, if the join in Query 4 produced a large number of results, then pushing the join below negation would be highly sub-optimal, despite the savings in negative tuple processing. On the other hand, if an ordering with the join pushed down is optimal to begin with, then *UPA* can make the plan even more efficient by eliminating the overhead of processing negative tuples below the negation operator.

3.6.6 Lessons Learned

The above experiments have illustrated the advantages of update-pattern-aware query processing as compared to the direct approach, which performs state maintenance inefficiently, and the negative approach, which effectively doubles the query processing time because “positive” and negative tuples must be processed by each operator. Query 4 has additionally shown the power of update-pattern-aware query optimization in finding efficient query plans that would not have been considered by a relational query optimizer employing standard optimization heuristics.

Chapter 4

Indexing Time-Evolving Data with Variable Lifetimes

4.1 Introduction

This chapter extends update pattern awareness to accommodate DSMS applications that spool data to disk for off-line analysis [113]. Applications under consideration in this chapter monitor data generated by one or more sources, perform light-weight processing on-the-fly, and periodically append new data to a disk-based archive. The archive is responsible for removing expired data and facilitating complex off-line queries that are too expensive to be done in real time. Examples include network traffic analysis, where the archive is mined by an Internet Service Provider (ISP) in order to discover recent usage patterns and plan changes in the network infrastructure [43]; transaction logging, where recent point-of-sale purchase records or telephone call logs are examined for customer behaviour analysis and fraud detection [65, 114]; and networks of sensors that measure physical phenomena such as temperature and humidity, where recent observations are used to discover trends and make predictions [88].

Recall from Section 15 that previous work on indexing sliding windows in secondary storage (the wave index) partitions the data chronologically into separate sub-indices. As a result, only one partition needs to be accessed (loaded into memory) during a periodic update. The wave index is based upon the assumption that the order in which the data are inserted is equivalent to the expiration order, which means that the lifetime of each data item is the same. As explained in Section 3.2, this assumption (i.e., weakest non-monotonic update patterns) holds if the application maintains time-based sliding windows or materialized results of simple queries such as selection. However, one may also choose to store (indexed) materialized results derived from one or more base windows, such as those of a sliding window join. If many queries over the archive compute the same join, then materializing the join result removes the need for each interested query to compute it from scratch[1]. Instead, the index may be used by each interested query to efficiently

[1]Deciding which sub-expressions to materialize is an orthogonal problem that is not pursued here; see, e.g.,

	Equal lifetimes	Variable lifetimes
Memory	FIFO queue	Calendar queue
Disk	Wave index	

Table 4.1: Classification of previous work on maintenance of time-evolving data

extract relevant data for further processing. As noted in Section 3.2, a join of time-based sliding window is weak non-monotonic and therefore its results may have different lifetimes.

In addition to introducing variable lifetimes by way of materialized results of weak non-monotonic query plans, sources may explicitly assign different lifetimes to the data that they generate. For example, one sensor may produce temperature measurements every ten minutes (giving each value a lifetime of ten minutes before being replaced with a new value), whereas another sensor may report humidity values every fifteen minutes. Similarly, various sources may be polled explicitly with different frequencies. For instance, the humidity sensor may require more energy to compute and/or transmit a new value than the temperature sensor, and should therefore be polled less often in order to save battery power [182].

As illustrated in Table 4.1, existing work on storing time-evolving data may be classified according to two criteria: main memory versus secondary storage, and equal versus variable lifetimes of the data items. Main-memory solutions were presented in Chapter 3, while the wave index is appropriate for disk-based storage of data having equal lifetimes[2]. This chapter exploits update pattern awareness to solve the most challenging of the four scenarios: disk-based indexing of data items having variable lifetimes. In the remainder of this chapter, Section 4.2 explains the limitations of previous work, Section 4.3 presents a solution, and Section 4.4 experimentally shows the advantages of the proposed solution in terms of index update and access times.

4.2 Assumptions and Motivation

The problem addressed in this chapter concerns indexing a time-evolving set of data items with associated lifetimes, such that index lookups and periodic updates may be done efficiently. The expiration time of each item is assumed to be known at generation time, but the lifetimes of various items may have different lengths, up to some pre-determined upper bound (i.e., weak non-monotonic update patterns). Applications that generate data with unpredictable or approximate lifetimes, or lifetimes whose length may change in response to the availability of storage space [85] (i.e., strict non-monotonic update patterns), are not considered. New data are continually generated by one or more sources and buffered in main memory between index updates. During an update, new items which have arrived since the last update are inserted and items whose lifetimes

[27, 52] for possible solutions in the context of data streams and sliding windows.

[2]There has also been previous work on storing large sliding windows on disk and avoiding deletions altogether by materializing multiple append-only prefixes of the window [95]. Again, the underlying assumption in [95] is that the lifetimes of all the data items are equal to the window length.

have expired are deleted. This involves bringing one or more pages into memory, updating them, and writing them back to disk. Two access types must be supported: probes (retrieval of items having a particular search key value or range), and scans of the entire index. Probes may be performed by queries that access a shared materialized result and extract a relevant subset of the data for further processing. Scans are performed by complex queries that must examine the entire data set in order to update their answers.

A chronologically partitioned index similar to the wave index illustrated in Figure 2.6 is inappropriate for disk-based storage of data with variable lifetimes. First, suppose that the index is partitioned by insertion time, as on the left of Figure 2.6. At time 18, only I_1 is accessed in order to insert new items, as before. However, all four sub-indices need to be scanned in order to determine which records have expired (it is no longer the case that only the items inserted between times one and 2 expire at time 18). This may require a large number of disk I/Os and cause unacceptably slow updates. Similarly, partitioning the index according to deletion times, as on the right of Figure 2.6, means that all the expired items at time 18 can be found in I_1, but there may be insertions into every sub-index (it is not the case that all records inserted between times 17 and 18 will expire between times 33 and 34). Again, all the sub-indices may need to be read into memory during index updates.

Recall that the calendar queue, which is conceptually similar to the wave index, was used in Chapter 3 to store the results of weak non-monotonic queries. However, this was under the assumption that all the data fit in main memory. Using a calendar queue split by expiration time was sufficient in the main-memory scenario because the goal was to make expirations more efficient (by not having to scan the entire result); insertions were allowed to be scattered across the entire result because of the luxury of random access in main memory. In this chapter, the fact that the data are stored on disk means that both insertions and expirations must be localized to a small number of sub-indices in order to prevent the entire index from being brought into main memory during each update.

4.3 Proposed Solution

Recall from Section 15 that a wave index balances two requirements: clustering by search key for efficient probing and by insertion (or expiration) time so that updates are confined to a single sub-index. Disk-based indexing of time-evolving data with variable lifetimes involves three conflicting requirements: clustering by search key for efficient probing, by insertion time for efficient insertions, and by expiration time for efficient deletions. This section propose a solution, referred to as a *doubly partitioned index*, that reconciles these three constraints. The idea is to simultaneously partition the index on insertion and expiration times.

4.3.1 Double Partitioning

A simple example of a doubly partitioned index (an improved variant will be presented shortly) is shown in Figure 4.1, given that the lifetimes of all the data records are at most 16 minutes

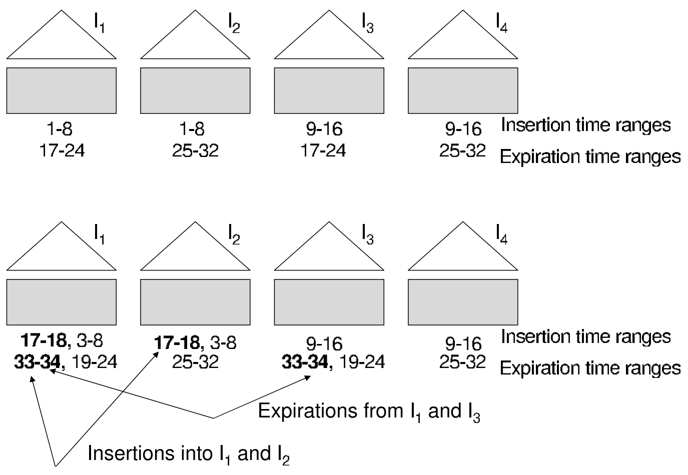


Figure 4.1: Example of a doubly partitioned index, showing an update at time 18 (bottom)

and that updates are performed every 2 minutes. As was the case with the wave index, each sub-index contains a directory on the search key and stores data records on disk, clustered by search key. However, the ranges of insertion and expiration times are now chronologically divided into two partitions each, creating a total of four sub-indices. As illustrated, at time 16, sub-index I_1 stores data items inserted between times one and 8 that will expire between times 17 and 24 (the other three sub-indices may be described similarly). The update illustrated on the bottom of Figure 4.1 takes place at time 18, inserts new items into I_1 and I_2, and deletes expired items from I_1 and I_3. Observe that I_4 does not have to be accessed during this update, or during the next three updates at times 20, 22, and 24. Then, the next four updates at times 26, 28, 30, and 32 will insert into I_3 and I_4, and delete from I_2 and I_4 (I_1 will not be accessed). In general, increasing the number of partitions leads to more sub-indices not being accessed during updates, thereby decreasing the index maintenance costs.

The flaw with chronological partitioning of the insertion and expiration times is that the sub-indices may have widely different sizes. Recall Figure 4.1 and note that at time 16, I_2 stores items that arrived between times one and 8 and will expire between times 25 and 32. That is, I_2 is empty at this time because there are no items whose lifetimes are larger than 16. As a result, the other sub-indices are large and their update costs may dominate the overall maintenance cost. This problem may be addressed by adjusting the intervals spanned by each sub-index. The improved technique, referred to as *round-robin partitioning*, is illustrated in Figure 4.2 for the same parameters as in Figure 4.1 (items have lifetimes of up to 16 minutes and index updates are done every two minutes). The two rows of intervals underneath each sub-index correspond to the insertion time and expiration time ranges, respectively. Rather than dividing the insertion and expiration time ranges chronologically, round-robin partitioning distributes updates in a round-robin fashion such that no sub-index experiences two consecutive insertions or expirations. For instance, the update illustrated on the bottom of Figure 4.2 takes place at time 18, inserts new

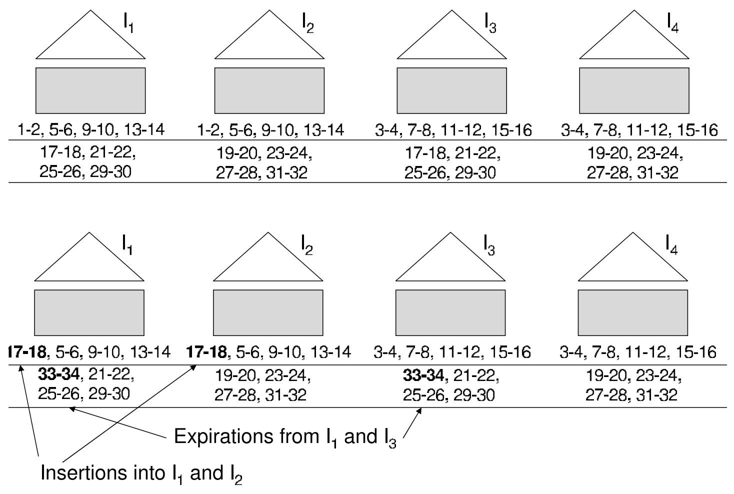


Figure 4.2: Example of a round-robin doubly partitioned index, showing an update at time 18 (bottom)

tuples into I_1 and I_2 and expires tuples from I_1 and I_3. The next update at time 20 inserts new tuples into I_3 and I_4, and deletes old tuples from I_2 and I_4. The fact that consecutive updates are spread out over different sub-indices ensures that the sub-indices have similar sizes, as formalized in Theorem 4.1 below. As will be shown experimentally in Section 4.4, this property translates to more efficient index updates.

Theorem 4.1 *Given a constant rate of insertion into the result and uniform distribution of data lifetimes, the average variance of sub-index sizes using round-robin partitioning is lower than the average variance of sub-index sizes using chronological partitioning.*

Proof. See Appendix A.
□

Doubly partitioned indices are compatible with two bulk-update strategies, denoted *Rebuild* and *NoRebuild*. With *Rebuild*, updated sub-indices are completely rebuilt and re-clustered, such that all the records with the same search key are stored contiguously in one dynamically-sized bucket (spanning one or more contiguous disk pages). With *NoRebuild*, each sub-index allocates multiple fixed-size buckets for each search key (usually not contiguously). Therefore, updates may cause additional buckets to be created or existing buckets to be deleted, if empty.

Let n be the number of sub-indices, G be the number of partitions of generation (insertion) times, and E be the number of partitions of expiration times ($n = G \times E$)[3]. Furthermore, let S be the upper bound on the lifetimes of data items and Δ be the time interval between two consecutive index updates. Algorithm 1 implements the round-robin doubly partitioned index and

[3]For example, in Figures 4.1 and 4.2, $G = 2$, $E = 2$, and $n = G \times E = 4$.

Algorithm 1 Round-robin doubly partitioned index

Input: number of partitions of insertion times G, number of partitions of expiration times E, maximum lifetime of a data item S, and interval between two consecutive index updates Δ

Initial stage at time τ

```
1  for $i = 0$ to $\frac{S}{\Delta G} - 1$
2      for $j = 0$ to $G - 1$
3          $I_{jE+1}$ through $I_{(j+1)E}$ are assigned an insertion time range of
           $\tau - S + (iG + j)\Delta + 1$ to $\tau - S + (iG + j + 1)\Delta$
4      end for
5  end for
6  for $i = 0$ to $\frac{S}{\Delta E} - 1$
7      for $j = 0$ to $E - 1$
8          $I_{j+1}, I_{E+j+1}, \ldots, I_{(G-1)E+j+1}$ are assigned an expiration time range of
           $\tau + S + (iG + j)\Delta + 1$ to $\tau + S + (iG + j + 1)\Delta$
9      end for
10 end for
11 Insert initial result tuples to appropriate sub-indices
```

Periodic update stage at time $\tau + j\Delta$, $j = 1, 2, \ldots$

```
1  for each sub-index with expiration time range of $\tau + (j-1)\Delta + 1$ to $\tau + j\Delta$
2      Replace above range with $\tau + (j-1)\Delta + 1 + S$ to $\tau + j\Delta + S$
3      Delete tuples with expiration times of $\tau + (j-1)\Delta + 1$ to $\tau + j\Delta$
4  end for
5  for each sub-index with insertion time range of $\tau + (j-1)\Delta + 1 - S$ to $\tau + j\Delta - S$
6      Replace above range with $\tau + (j-1)\Delta + 1$ to $\tau + j\Delta$
7      Insert new result tuples to appropriate sub-indices
8  end for
```

contains two stages: the initial stage and the periodic update stage. Suppose that the algorithm starts with a set of data that are assumed to be valid at some time τ. The initial stage partitions the insertion and expiration times, inserts the data records into the appropriate sub-indices, and builds the corresponding sub-index directories. The update stage periodically accesses selected sub-indices in order to adjust the insertion and expiration times that they span, insert and/or delete tuples in the appropriate sub-indices according to the insertion and expiration times, and update the corresponding sub-index directories. A detailed implementation of insertions and deletions is not shown in the algorithm as this depends on the clustering technique (*Rebuild* versus *NoRebuild*). Similarly, specific details concerning directory updates are omitted since the algorithm is compatible with a wide range of directory data structures.

Note that lines 1 and 4 of the periodic update stage look up a sub-index according to the range of insertion or expiration times that it spans. One way to speed up these lookups is to maintain a meta-index, over the time intervals spanned by individual sub-indices (see, e.g., [260] in the context of data stream processing or [209] in the context of traditional spatio-temporal indexing). However, this improvement is not considered any further in this chapter because the number of sub-indices in a doubly partitioned index is not expected to be large and therefore determining which sub-indices are to be accessed during a given update is not expensive.

4.3.2 Cost Analysis

The number of sub-indices accessed during each update is $G + E - 1$. To choose optimal values for G and E with respect to the number of sub-index accesses, it suffices to minimize $G + E$ given that $G \times E = n$ and $G, E \geq 2$, which yields $G{=}E{=}\sqrt{n}$. Thus, the value of n should be chosen to be a perfect square.

Increasing n increases the space requirements (each sub-index requires its own directory on the search key) and leads to slower query times because index scans and probes need to access all n sub-indices. Additionally, more individual sub-indices are accessed during updates as n increases. However, the sub-indices are faster to update because they are smaller, and the fraction of the data that need to be updated decreases. For instance, setting $G = E = 2$ (as in Figures 4.1 and 4.2) means that three of the four sub-indices are scanned during updates, but increasing G and E to four means that only seven of sixteen sub-indices are accessed. As will be shown in Section 4.4, increasing n initially decreases update times, but eventually a breakpoint is reached where the individual sub-indices are small and making any further splits is not helpful (note that the breakpoint value of n is expected to be higher for larger data sets).

The other part of the maintenance and query costs is contributed by the operations done after a sub-index is accessed. Fixing G and E, *Rebuild* should be faster to query (only one bucket is accessed to find all records with a given search key), but slower to update (especially as the data size grows, because rebuilding large indices may be expensive). Access into *NoRebuild* is slower because records with the same search key may be scattered across many buckets, but *NoRebuild* should be faster to update because individual updates are less costly than rebuilding an entire sub-index. Furthermore the total size of *NoRebuild* may be larger than *Rebuild* because some pre-allocated buckets may not be full.

4.3.3 Handling Fluctuating Stream Conditions

Round-robin partitioning creates sub-indices with similar sizes if the amount of new data arriving between updates does not change. However, in the worst case, the data rate may alternate between slow and bursty periods, causing the round-robin allocation policy to create some sub-indices that are very large and some that are very small. The algorithmic solution in this case is to randomize the update allocation policy. In practice, though, random fluctuations in the data rate are expected. Furthermore, the change in the data rate may be persistent for several index updates, or short-lived between two consecutive updates. In both cases, round-robin partitioning is expected to adapt to the new conditions. Given a persistent change, round-robin update allocation ensures that updates are spread out across the sub-indices. Thus, if the number of new data items increases (or decreases), then each sub-index will in turn get larger (or smaller), until all the sub-indices have similar sizes again. Using chronological partitioning, the same sub-index would receive a number of consecutive updates and become either much larger or much smaller than the others. If a change is short-lived, then it is also better to begin with equal sub-index sizes. Otherwise, a burst of new data could be inserted into a large sub-index, which would become even larger.

4.4 Experiments

This section contains an overview of the implementation of doubly partitioned indices (Section 4.4.1) and experimental results. Sections 4.4.2 through 4.4.4 present results of experiments with a small data set of approximately 500 Megabytes (this corresponds to data generated over a time of 500000 time units, with an average of one record generated per time unit). Section 4.4.5 investigates index performance over larger data sets with sizes of up to 5 Gigabytes (i.e., data produced over a time of 5 million time units, with an average of one record generated per time unit). The experimental findings are summarized in Section 4.4.6.

4.4.1 Implementation Details

The doubly partitioned indices (*Rebuild* and *NoRebuild*) were implemented in Java, and tested on a Linux PC with a Pentium IV 2.4Ghz processor and 2 Gigabytes of RAM. For comparison, two chronologically partitioned wave indices from [222] were also implemented (recall Figure 2.6): *REINDEX*, which is similar to *Rebuild* in that it reclusters sub-indices after updates, and *DEL*, which is similar to *NoRebuild* as it maintains multiple fixed-size buckets per key. Both *REINDEX* and *DEL* may be partitioned by insertion time (abbreviated *R-ins* or *D-ins*, respectively) or by expiration time (abbreviated *R-exp* or *D-exp*). The indexing techniques will be referred to by their abbreviations, followed by the value of n (number of sub-indices) or values of G and E (number of partitions of insertion and expiration times, respectively), e.g., *R-ins*4 or *Rebuild*2x2.

Each test consists of an initial building stage and an update stage. The building stage populates the index using records with randomly generated lifetimes and search key values (both generated from a uniform distribution). The total data size in the initial stage varies from 500 Megabyte to five Gigabytes. Next, periodic updates are generated using the same lifetime and search key distribution, and inserted into the index, at the same time removing expired tuples. After each update, an index probe is performed (retrieving tuples having a randomly chosen search key value), followed by an index scan. The average processing time of each operation is reported. 36 updates are performed until the amount of new data generated equals the initial data size. This corresponds to an index update frequency of roughly 14000 to 140000 time units, with a relative data rate of one record per time unit.

Each indexing technique consists of an array of sub-indices, with each sub-index containing a main-memory directory (implemented as a linked list sorted by search key) and a random access file storing the results. The file is a collection of buckets storing tuples with the same

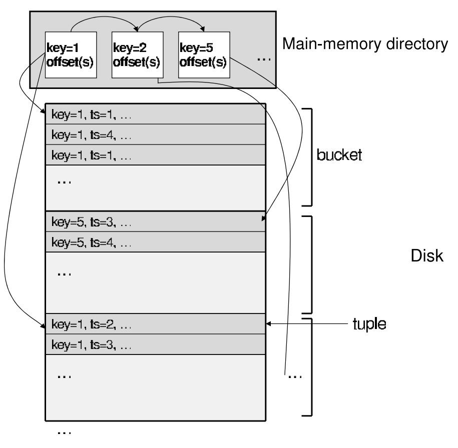


Figure 4.3: Structure of an individual sub-index

search key, sorted by expiration time. Individual data records are 1000 bytes long and contain an integer search key, an integer expiration timestamp, as well as a string that abstractly represents the contents of the data. The structure of an individual sub-index in *NoRebuild* and *DEL* is illustrated in Figure 4.3, showing the directory with offset pointers to locations of buckets in the file (note that there may be more than one bucket per search key in case of overflow). The count of records in each bucket is also stored as not all buckets are full. *Rebuild* and *REINDEX* are structured similarly, except that one variable-size bucket is maintained for each search key.

A number of simplifications have been made to focus the experiments on the relative performance of doubly partitioned indices. First, bucket sizes are not adjusted upon overflow; this issue was studied in [89] in the context of skewed distributions and is orthogonal to this work. Instead a simple strategy was implemented that allocates another bucket of the same size for the given key. Garbage-collection of empty buckets is also ignored because it adds a constant amount of time to the maintenance costs of each indexing technique. Second, the number of search key values is fixed at 100 in order to bound the length of the directory. Otherwise, query times may be dominated by the time it takes to scan a long list; handling a larger set of key values can be done with a more efficient directory, such as a B+-tree, and is orthogonal to this work. Third, the number of tuples per bucket in *NoRebuild* and *DEL* is based upon the initial distribution of key values in the building stage, such that each sub-index contains an average of 2.5 buckets per search key. This value was found to be a good compromise between few large buckets per key (which wastes space because many newly allocated buckets never fill up) and too many buckets (which results in slower query times).

4.4.2 Optimal Values for G and E

The first experiment validates the result from Section 4.3.2 regarding the optimal assignment of values for G and E given a value for n. Figure 4.4 shows the normalized update, probe, and

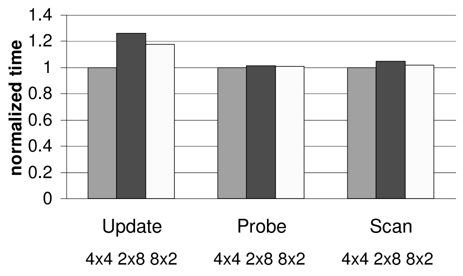


Figure 4.4: Relative performance of *Rebuild*4x4, *Rebuild*2x8, and *Rebuild*8x2

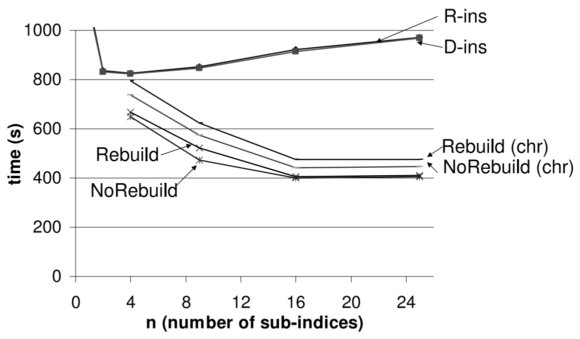


Figure 4.5: Update times of index partitioning techniques given a small window size

scan times for *Rebuild*4x4, *Rebuild*2x8, and *Rebuild*8x2; other index types give similar relative results. *Rebuild*4x4 performs best in terms of scan and probe times, though the difference is negligible because all three techniques probe the same number of sub-indices to obtain query results and all the sub-indices have roughly equal sizes. The average update time of *Rebuild*4x4 is approximately 20 percent lower than the other techniques because the number of sub-indices updated by *Rebuild*4x4 is 7, versus 9 for the other two strategies. Notably, *Rebuild*8x2 can be updated faster than *Rebuild*2x8 because tuples inside buckets are ordered by expiration time, and therefore deletions are simple (tuples are removed from the front of the bucket) but insertions are more complex (whole bucket must be scanned). Since the number of insertions is determined by the number of partitions in the lower level, the technique with a smaller value of E wins.

4.4.3 Performance of Doubly Partitioned Indices

Doubly partitioned indices are now compared with the existing algorithms. As per the previous experiment, only the following partitions are considered: 2x2, 3x3, 4x4, and 5x5. Results for *R-exp* and *D-exp* are omitted because these techniques always incur longer update times than *R-ins* and *D-ins*. As before, this is because insertions are more expensive than deletions if buckets are sorted by expiration time, therefore splitting an index by expiration time forces insertions into every sub-index. Figures 4.5, 4.6, and 4.7 show the average update, probe, and scan times, respectively, as functions of n (number of sub-indices). Figure 4.5 additionally shows the update times of doubly partitioned indices with chronological partitioning (denoted by *chr*) in order to single out the benefits of round-robin partitioning. Even chronological partitioning outperforms the existing strategies by a factor of two as n grows, with round-robin partitioning additionally improving the update times by ten to 20 percent. As explained in Section 4.3.2, *NoRebuild* is faster to update than *Rebuild*, but is slower to probe and scan.

The update overhead of *Rebuild* relative to *NoRebuild* is roughly five percent for $n < 9$ and decreases to under two percent for large n. The relative savings in index probe times of *Rebuild* are less than one percent. This is because a relatively small data size is assumed in this

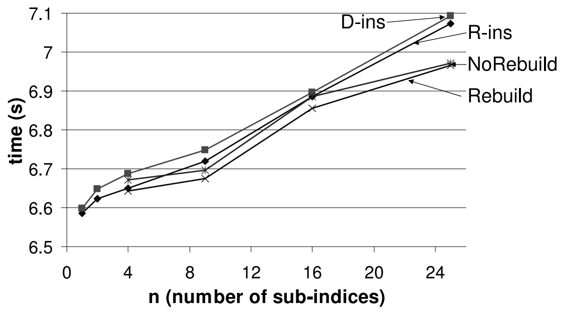


Figure 4.6: Probe times of index partitioning techniques given a small window size

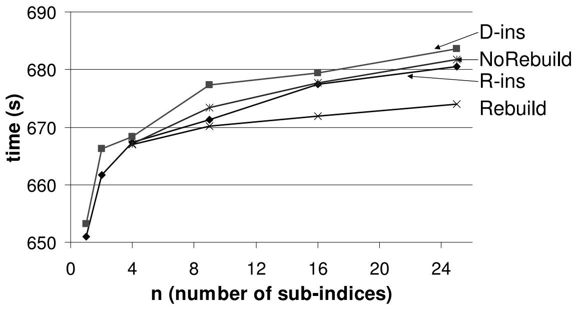


Figure 4.7: Scan times of index partitioning techniques given a small window size

experiment (roughly 500 Megabytes), meaning that the individual sub-indices are small and can be rebuilt quickly. Additionally, all the buckets with a particular search key may be found with a small number of disk accesses, even if the buckets are scattered across the file. Hence, probing *NoRebuild* is only slightly more expensive than probing *Rebuild*, where records with the same search keys are found in the same bucket. In general, *Rebuild* and *NoRebuild* perform probes slightly faster than *R-ins* and *D-ins* because the sub-indices in our techniques have similar sizes, and therefore avoid "bad cases" where probing one or more large sub-indices inflates the access cost.

As n increases, the access times grow because more sub-indices must be probed separately, whereas update times decrease initially, but begin growing for $n \geq 25$ (or $n \geq 9$ for *R-ins* and *D-ins*). As mentioned in Section 4.3.2, this is due to two factors influencing the update costs: as n increases, the amount of data to be updated decreases, but the number of individual sub-index accesses increases. The latter is the reason why the update costs of *R-ins* and *D-ins* start increasing for smaller values of n than those for our doubly partitioned techniques: the existing techniques access all n sub-indices during updates, whereas the doubly partitioned techniques only access $G + E - 1 = 2\sqrt{n} - 1$ sub-indices.

Given a fixed value of n, *Rebuild* and *R-ins* both have the lowest space requirements, followed by *NoRebuild* and *D-ins*. *NoRebuild* and *D-ins* incur the overhead of pre-allocating buckets which may never fill up (the exact space penalty depends on the bucket allocation strategy, which is orthogonal to this work). As n increases, all techniques require more space in order to store sub-index directories.

To measure the overhead associated with doubly partitioned indices, chronological and round-robin partitioning were also tested without indices. Update times were down by approximately 20 percent because index directories did not have to be updated and files were not reclustered. However, probes and scans both required a sequential scan of the data, and took approximately the same amount of time as index scans in Figure 4.7, namely on the order of 600 seconds. Thus, doubly partitioned indices incur modest update overhead, but allow probe times that are two orders of magnitude faster than sequential scan.

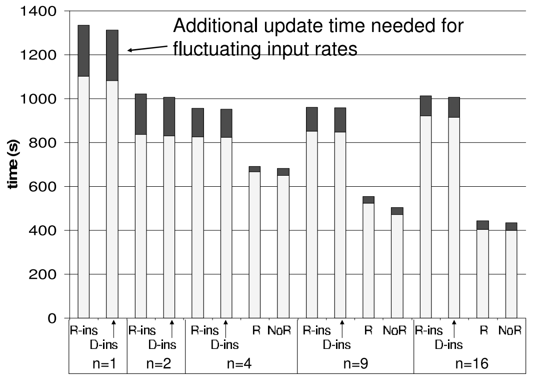


Figure 4.8: Effect of data rate fluctuations on index update performance.

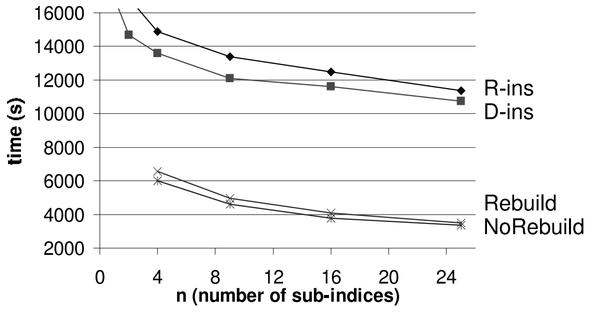


Figure 4.9: Update times of index partitioning techniques given a large data set.

4.4.4 Fluctuating Stream Conditions

In this experiment, the data rate varies randomly by a factor of up to four. The total amount of data items generated is set to be approximately the same as in the previous experiment in order to enable a head-to-head comparison. The average access times were slower by several percent; the index update times are illustrated in Figure 4.8 for selected techniques (R and NoR denote *Rebuild* and *NoRebuild*, respectively). The darkened portion of each bar corresponds to the increase in update time caused by the fluctuating data rate.

Doubly partitioned indices are more adaptable to fluctuating data rates than *R-ins* and *D-ins*. *NoRebuild* and *Rebuild* are more significantly affected by fluctuations for larger values of n, whereas *R-ins* and *D-ins* exhibit the worst performance for small values of n. This can be explained as follows. *Rebuild* and *NoRebuild* use round-robin partitioning, meaning that updates are scattered across sub-indices, therefore a large value of n means that it takes longer for the new data rate to take effect in all the sub-indices. On the other hand, *R-ins* and *D-ins* use chronological partitioning, therefore a large value of n means that the sub-indices have shorter time spans and therefore bursty updates spread out faster across the sub-indices. Finally, *Rebuild* and *R-ins* are more resilient to fluctuations than *NoRebuild* and *D-ins* because the latter two use a simple (non-adaptive) bucket allocation technique.

4.4.5 Scaling up to Large Index Sizes

This test investigates the behaviour of the proposed techniques when indexing large amounts of data (five Gigabytes). The average update, probe, and scan times as functions of n are shown in Figures 4.9, 4.10, and 4.11, respectively. Doubly partitioned indices are now up to three times as fast to update as the existing techniques. Additionally, the gap between the update and query times of *Rebuild* versus *NoRebuild* is now wider. *Rebuild* is two to three percent faster to probe, but between five (for $n = 25$) and nine (for $n = 4$) percent slower to update; the

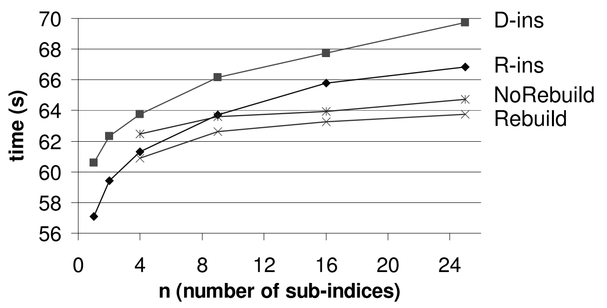


Figure 4.10: Probe times of index partitioning techniques given a large data set.

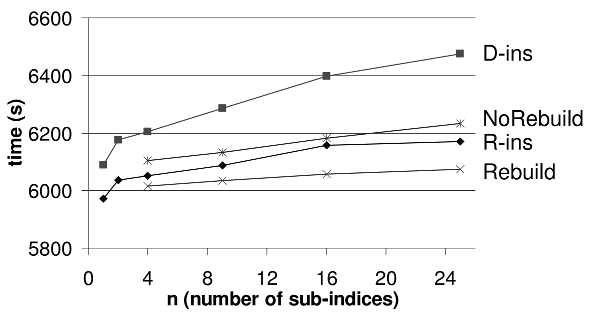


Figure 4.11: Scan times of index partitioning techniques given a large data set.

corresponding percentages from Figure 4.6 are less than one percent and roughly three percent, respectively. This is the expected outcome of indexing a large data set: *Rebuild* becomes slower to update because it must re-cluster larger sub-indices, whereas *NoRebuild* becomes slower to probe because there are more result tuples with the same search key, spread over multiple buckets and possibly multiple disk pages.

Another difference between Figures 4.9 and 4.5 is the behaviour of update times as n grows. In Figure 4.5, there is a turning point (at $n = 16$ for *NoRebuild* and *Rebuild*) after which update times do not decrease. This is not the case in Figure 4.9, where update times continue to drop for all tested values of n. This is because the window size, and hence individual sub-index sizes, are larger, therefore the drop in performance caused by making the sub-indices too small is not an issue for $n \leq 25$.

4.4.6 Lessons Learned

Based upon the above experiments, the following recommendations can be made regarding the best index partitioning strategy. The guidelines depend upon the data size and the expected number of queries to be executed over the archive between updates.

- For a small window size and small number of queries, *NoRebuild*4x4 is a good choice as it incurs low update times.
- For a small window size and large number of queries, *Rebuild*2x2 works best because its probe and scan times are low. The probing times of *R-ins*1, and *R-ins*2 are slightly lower than those of *Rebuild*2x2, but updating *R-ins* is slower.
- For a large window size and small number of queries, *NoRebuild* is a good choice, but with a smaller value of n than recommended for small window sizes to ensure that probing times are not excessively high (e.g., *NoRebuild*3x3).

- For a large window size and large number of queries, *Rebuild* becomes expensive to update, therefore *Rebuild*2x2 or *Rebuild*3x3 are recommended only if fast probing times are crucial. Otherwise, *NoRebuild*2x2 is a better (more balanced) choice.

Note that the query workload may fluctuate over time, therefore it may be advantageous to switch to a different indexing technique at some point. This problem is similar to plan migration in the context of sliding window queries that store state [275]. Two possible solutions are either stopping the old plan, migrating the state, and starting the new plan, or running both plans in parallel and discarding the old plan when all the windows roll over. Both strategies are compatible with doubly partitioned indices in that the system can migrate from one index type to another either by discarding the old index and building a new index, or maintaining both indices in parallel until the old index gradually empties out.

Chapter 5

Multi-Join Processing over Sliding Windows

5.1 Introduction

Recall from Chapter 2 that persistent queries may be executed continuously (eagerly) or periodically (lazily). Furthermore, recall from Section 2.2.2 that some operators, including the join, may maintain their state eagerly or lazily, with the trade-off that lazy expiration requires more memory for temporarily storing expired tuples. This chapter studies algorithms for evaluating an equi-join of n sliding windows in main memory using eager or lazy evaluation and expiration [111]. Joins are important operators in a DSMS as they facilitate cross-referencing of similar events on multiple input streams which have occurred within a window length of each other [99].

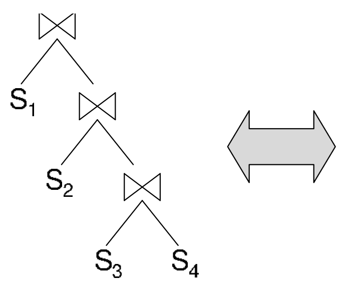
Section 5.2 begins by defining several multi-way join algorithms compatible with the above four options. Additionally, two lazy-evaluation variants are presented for ensuring restore and no-restore semantics (recall Section 3.3.4), and dealing with weakest, weak, and strict non-monotonic inputs is discussed. All of the proposed algorithms are fully pipelined (intermediate results are not materialized due to main memory constraints). Next, Section 5.3 outlines join ordering heuristics that attempt to minimize the number of tuples in the pipeline. The heuristics are based upon a per-unit-time cost model also used in [147] in the context of binary sliding window joins. Finally, Section 5.4 discusses experimental results, including the effects of re-evaluation and expiration strategies on the overall processing cost.

A multi-way hash join for unbounded streams has been proposed in [250], but extensions to sliding windows were not presented. Moreover, while the join ordering problem has been identified in the context of optimizing for the highest output rate of queries over unbounded streams [249, 250], ordering sliding window joins has not been discussed. Generally, main-memory join ordering techniques in DBMSs push expensive predicates to the top of the plan [256].

Table 5.1 lists the symbols used in this chapter and their meanings. Furthermore, Figure 5.1 explains the convention for describing join ordering. In the example, the join order $S_1 \bowtie (S_2 \bowtie$

Table 5.1: Explanations of symbols used in this chapter

λ_i	Arrival rate of stream i in tuples per unit time
S_j	Sliding window corresponding to stream j
T_j	Length of the j^{th} time-based window
C_j	Number of tuples in S_j
v_j	Number of distinct values in S_j
b_j	Number of hash buckets in the hash index of S_j, if such an index exists
Δ	Periodic re-execution interval
$a \circ b$	Concatenation of tuples a and b
ts	Timestamp attribute



```
For all tuples in $S_1$
  For all tuples in $S_2$
    If $S_1$.a = $S_2$.a
      For all tuples in $S_3$
        If $S_2$.a = $S_3$.a
          For all tuples in $S_4$
            If $S_3$.a = $S_4$.a
              Return join tuple
```

Figure 5.1: Join order $S_1 \bowtie (S_2 \bowtie (S_3 \bowtie S_4))$ expressed as a join tree and a series of for-loops

$(S_3 \bowtie S_4))$ is expressed as a join tree on the left and as a series of nested for-loops on the right; the join predicate is an equality condition on a common attribute named a. S_1 is said to be "ordered first", S_2 "ordered second", and so on. For brevity, parentheses will be omitted and the notation S_1, S_2, S_3, S_4 will be used to represent the join order shown in Figure 5.1.

5.2 Sliding Window Join Algorithms

5.2.1 Eager Evaluation

The discussion of join algorithms begins with the simplest case of eager re-evaluation and eager expiration. Assume that the join conditions are equality predicates on a common attribute across all streams, call it a, which means that all permutations of the join order are possible.

A binary sliding window works as follows [147], assuming that its inputs have weakest or weak non-monotonic update patterns and therefore negative tuples need not be used. Let S_1 and S_2 be the two inputs. Each newly arrived S_1-tuple is inserted into its state. Next, expired tuples are removed from the other state (S_2), which is then scanned to produce new results involving the new S_1-tuple. Recall that a tuple expires if its *exp* timestamp is older than the current time, or, in this case, the timestamp of the newly arrived S_1-tuple. Moreover, observe that expiration from S_2 is done before the scan so that tuples which have expired relative to the timestamp of the new

Table 5.2: Probing orders given that the global join order is $S_1 \bowtie (S_2 \bowtie S_3)$

Origin of new tuple	Join order
S_1	$S_1 \bowtie (S_2 \bowtie S_3)$
S_2	$S_2 \bowtie (S_1 \bowtie S_3)$
S_3	$S_3 \bowtie (S_1 \bowtie S_2)$

tuple do not participate in join processing. A similar procedure is followed for each newly arrived S_2-tuple. Note that each result tuple is assigned an expiration timestamp that is the minimum of the *exp* timestamps of the individual tuples. It is then up to the next operator in the pipeline (or an application attached to the final output of the query) to expire old result tuples.

On the other hand, if at least one of the inputs of the join is strict non-monotonic (e.g., count-based window), then negative tuples must be used. In this case, negative tuples are processed in the same way as regular tuples and produced on the output stream. Additionally, expiration is performed when a negative tuple arrives and the corresponding regular tuple is deleted.

Extending the binary join to deal with more than two inputs is straightforward: for each newly arrived tuple k, expired tuples are removed in all the other state buffers, and the buffers are then scanned in the order prescribed by the query plan. For example, suppose that three windows, call them S_1, S_2, and S_3, are joined using the plan $S_1 \bowtie (S_2 \bowtie S_3)$. Upon arrival of a new S_1-tuple, S_2 and S_3 are purged of expired tuples and then probed in that order. If a new S_2-tuple arrives, then S_1 is probed first, followed by S_3. Similarly, upon arrival of a new S_3-tuple, S_1 is probed first, and then S_2. In effect, the window at the top of the join order always consists of only one tuple (i.e., the newly arrived tuple) and the join order changes in response to the origin of the incoming tuple, as shown in Table 5.2. The global join order is defined to be the order followed when processing new tuples from the first window. In the above example, the global order is S_1, S_2, S_3.

Algorithm 2 implements the sliding window join using eager evaluation and expiration. Without loss of generality, a global join order of $S_1, S_2 \ldots S_n$ is assumed. As in the binary join, negative tuples are used to remove the corresponding regular tuples in the join state (not shown in the algorithm) and are processed by subroutine *ComputeJoin* in the same way as regular tuples. If negative tuples are used to announce all expirations (not only the premature ones), then line 3 may be removed as expired tuples are being removed when the corresponding negative tuples arrive. If hash tables are maintained, then only the appropriate hash buckets are scanned inside subroutine *ComputeJoin* (lines 1, 4, 6, and 9), not the entire inputs. If a new tuple arrives while a previously arrived tuple is being processed, then the new tuple is assumed to wait in a queue. Furthermore, if two tuples arrive simultaneously, then ties may be broken arbitrarily in line 1.

In order to allow lazy expiration, Algorithm 2 must identify and ignore expired tuples during processing. This can be accomplished by padding the join predicates with timestamp comparisons, as summarized in Algorithm 3. The timestamp comparisons guarantee that newly arrived tuples do not join with expired tuples (relative to the timestamp of the new tuple) that have not yet been

Algorithm 2 Eager Multi-Way Join

```
Input: $n$ sliding windows, $S_1$ through $S_n$, over $n$ input streams
if a new tuple $k$ arrives on stream $i$ then
  insert new tuple in window $S_i$
  remove expired tuples from all other windows
  ComputeJoin($k$, $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
end if

ComputeJoin(new tuple $k$, join order $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
for each $u \in S_1$
  if $k.a = u.a$ then
    \\ loop through $S_2$ up to $S_{i-2}$
    for each $v \in S_{i-1}$
      if $k.a = v.a$ then
        for each $w \in S_{i+1}$
          if $k.a = w.a$ then
            \\ loop through $S_{i+2}$ up to $S_{n-1}$
            for each $x \in S_n$
              if $k.a = x.a$ then
                return $k \circ u \circ \ldots \circ v \circ \ldots \circ x$
              end if
            end for
            \\ end for loops $S_{i+2}$ up to $S_{n-1}$
          end if
        end for
      end if
    end for
    \\ end for loops $S_2$ up to $S_{i-2}$
  end if
end for
```

removed. Expiration is not specified in the algorithm as it may be performed at arbitrary times, but not during the execution of subroutine *ComputeJoinLazyExp*. Note that Algorithm 3 applies only when all of its inputs have weakest or weak non-monotonic update patterns. Otherwise, it is not possible to predict the expiration times of tuples in the join state and expirations must be performed eagerly via negative tuples.

5.2.2 Lazy Evaluation

Recall from Section 3.3.4 that lazy re-evaluation of weak and strict non-monotonic query plans does not produce the same result set as continuous evaluation. In the context of the sliding window join, newly arrived tuples joining with tuples that are about to expire may never be reported if they have short lifetimes falling between two re-execution intervals. If no-restore

Algorithm 3 Eager Multi-Way Join with Lazy Expiration

```
Input: $n$ sliding windows, $S_1$ through $S_n$, over $n$ input streams
1  if a new tuple $k$ arrives on stream $i$ then
2    insert new tuple in window $S_i$
3    ComputeJoinLazyExp($k$, $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
4  end if

ComputeJoinLazyExp(new tuple $k$, join order $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
1   for each $u \in S_1$ and $k.ts \leq u.exp$
2     if $k.a = u.a$ then
3       \\ loop through $S_2$ up to $S_{i-2}$
4       for each $v \in S_{i-1}$ and $k.ts \leq v.exp$
5         if $k.a = v.a$ then
6           for each $w \in S_{i+1}$ and $k.ts \leq w.exp$
7             if $k.a = w.a$ then
8               \\ loop through $S_{i+2}$ up to $S_{n-1}$
9               for each $x \in S_n$ and $k.ts \leq x.exp$
10                if $k.a = x.a$ then
11                  return $k \circ u \circ \ldots \circ v \circ \ldots \circ x$
12                end if
13              end for
14              \\ end for loops $S_{i+2}$ up to $S_{n-1}$
15            end if
16          end for
17        end if
18      end for
19      \\ end for loops $S_2$ up to $S_{i-2}$
20    end if
21  end for
```

semantics are acceptable, then one possibility is to re-use Algorithm 2, augmented to clean out expired tuples before each re-execution. This augmentation is correct because short-lived results that have expired just before the re-execution need not be recovered and therefore expired tuples do not need to be retained for processing. However, improvements can be made to reduce the time and space requirements of lazy join evaluation. First, processing time can be reduced by sorting or hashing newly arrived tuples on their a-attributes on-the-fly and calling subroutine *ComputeJoin* only once per group of new tuples from the same input having the same a-value. This reduces the number of times that the other inputs are probed and is similar to the traditional block-oriented nested-loops join [96]. Second, space usage may be decreased by removing tuples that will expire before the next re-evaluation immediately after the current re-evaluation (there is no need to store these tuples between re-evaluations). This gives rise to the lazy multi-way join outlined as Algorithm 4; without loss of generality, assume that *NOW* is the time at which re-execution

Algorithm 4 Lazy Multi-Way Join with No-Restore Semantics

```
Input: $n$ sliding windows, $S_1$ through $S_n$, over $n$ input streams
Every time the query is to be re-executed
for $i = 1 \ldots n$
   insert newly arrived tuples on stream $i$ into $S_i$
   for each group $G$ of newly arrived tuples in $S_i$ having the same $a$-value $val$
      LazyComputeJoin($G$, $val$, $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
   end for
end for
remove tuples from each window $S_i$ having $exp$ timestamps smaller than $NOW + \Delta$

LazyComputeJoin(group of tuples $G$, $a$-value $val$, join order $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
for each $u \in S_1$
   if $val = u.a$ then
      \\ loop through $S_2$ up to $S_{i-2}$
      for each $v \in S_{i-1}$
         if $val = v.a$ then
            for each $w \in S_{i+1}$
               if $val = w.a$ then
                  \\ loop through $S_{i+2}$ up to $S_{n-1}$
                  for each $x \in S_n$
                     if $val = x.a$ then
                        for each $g \in G$
                           return $g \circ u \circ \ldots \circ v \circ \ldots \circ x$
                        end for
                     end if
                  end for
                  \\ end for loops $S_{i+2}$ up to $S_{n-1}$
               end if
            end for
         end if
      end for
      \\ end for loops $S_2$ up to $S_{i-2}$
   end if
end for
```

begins. Note that timestamp comparisons are not required during join processing because expired tuples are guaranteed to have been removed at the end of the previous re-execution.

As was the case with Algorithm 3, Algorithm 4 relies on the knowledge of expiration times of tuples in its state (line 8), therefore it is not compatible with strict non-monotonic inputs. However, the lazy multi-way join may be modified to handle strict non-monotonic inputs by performing expiration eagerly in response to newly arrived negative tuples, and evaluating the join lazily as in subroutine *LazyComputeJoin*.

Conforming to restore semantics requires two changes to Algorithm 4. First, line 8 changes to "remove tuples from S_i having *exp* timestamps smaller than NOW" because tuples that will expire before the next re-execution (at time $NOW + \Delta$) could join with newly arrived tuples and produce short-lived results. Second, subroutine *LazyComputeJoin* must now carry out timestamp comparisons because not all expired tuples will be used to restore short-lived results. Algorithm 5 implements restore semantics. Note that currently processed tuples having the same a-values may have different timestamps in the range between $NOW - \Delta$ and NOW. Therefore, the algorithm produces a superset of possible results in the outer for-loops, which is why the smallest timestamp of the newly arrived tuples is used along with the join predicate. Then, in lines 11 through 13, subroutine *LazyRestoreComputeJoin* verifies that none of the base tuples have expired relative to the specific timestamp of each newly arrived tuple.

One way to adapt Algorithm 5 to strict non-monotonic input is as follows. When a negative tuple arrives, the corresponding regular tuple is looked up, but not deleted right away. Instead, the expired tuple is given an *exp* timestamp corresponding to the generation timestamp of the negative tuple. This allows subroutine *LazyRestoreComputeJoin* to recover short-lived join results. When the join processing stage completes, line 9 of Algorithm 5 may be used to garbage-collect the expired tuples since their *exp* timestamps have been set by the corresponding negative tuples.

5.2.3 Analysis of Algorithms

The trade-offs involved in the above join algorithms are analyzed by considering their relative costs in terms of window maintenance and join computation. In terms of join processing, a hash join is faster than a nested-loops join, but incurs higher window maintenance costs. For example, handling new tuples is more expensive in a hash join because the hash function must be applied to each new tuple. Moreover, expiration is more costly because each bucket must be probed separately (depending upon the update patterns of the input, each hash bucket may be a FIFO queue, a calendar queue, or a simple list, as discussed in Section 21). Additionally, if many tuples have the same value of the join attribute, then lazy evaluation is more efficient than eager evaluation. However, the downside of lazy evaluation is that answers are reported to the user with a delay (recall the discussion in Section 3.3.4). Now, in terms of window maintenance, eager expiration is more costly. For instance, if expiration is performed frequently, than some windows or hash buckets may not contain any stale tuples, yet the algorithm must still pay for the cost of accessing them. However, under eager evaluation, eager expiration reduces the join processing cost by eliminating the need to pad join predicates with timestamp comparisons. In the context

Algorithm 5 Lazy Multi-Way Join with Restore Semantics

```
1  Every time the query is to be re-executed
2  for $i = 1 \ldots n$
3    insert newly arrived tuples on stream $i$ into $S_i$
4    for each group $G$ of newly arrived tuples in $S_i$ having the same $a$-value $val$
5      let $mints$ be the smallest timestamp of all tuples in $G$
6      LazyRestoreComputeJoin($G$, $val$, $mints$, $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
7    end for
8  end for
9  remove tuples from each window $S_i$ having timestamps smaller than $NOW - T_i$
```

LazyRestoreComputeJoin(group of tuples G, a-value val, timestamp $mints$, join order $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)

```
1   for each $u \in S_1$ and $mints- \leq u.exp \leq NOW$
2     if $val = u.a$ then
3       \\ loop through $S_2$ up to $S_{i-2}$
4       for each $v \in S_{i-1}$ and $mints \leq v.exp \leq NOW$
5         if $val = v.a$ then
6           for each $w \in S_{i+1}$ and $mints \leq w.exp \leq NOW$
7             if $val = w.a$ then
8               \\ loop through $S_{i+2}$ up to $S_{n-1}$
9               for each $x \in S_n$ and $mints \leq x.exp \leq NOW$
10                if $val = x.a$ then
11                  for each $g \in G$
12                    if $u.exp \geq g.ts$ and ... and $v.exp \geq g.ts$ and $w.exp \geq g.ts$
                      and ... and $x.exp \geq g.ts$
13                      return $g \circ u \circ \ldots \circ v \circ \ldots \circ x$
14                    end if
15                  end for
16                end if
17              end for
18              \\ end for loops $S_{i+2}$ up to $S_{n-1}$
19            end if
20          end for
21        end if
22      end for
23      \\ end for loops $S_2$ up to $S_{i-2}$
24    end if
25  end for
```

of lazy evaluation, enforcing restore semantics is more expensive because timestamp comparisons are made during join processing. These issues will be explored in more depth in Section 5.4.

Given that evaluating the join using restore semantics is more expensive, one might choose to accept no-restore semantics if the proportion of lost results is small. First, suppose that the distribution of the lifetime lengths of join results is uniform. That is, a new tuple has an equal probability of joining with new or old tuples from the other windows. In this case, the proportion of lost results may be approximated by the ratio of the re-evaluation interval Δ and the window length (weighted by the stream arrival rates if each stream is bounded by a window with a different length). For example, if a join of 60-second windows is evaluated lazily every ten seconds, then 17 percent of the results are likely to be missing. On the other hand, consider a join of two windows, where one of the input streams "lags" behind the other one in a sense that newly arrived tuples in one window join with old tuples in the other window [228]. If the lag is approximately equal to the window size, then most of the join results are short-lived and lazy re-evaluation may return a small fraction of the result set.

Finally, recall that if the input to the join operator is strict non-monotonic, then the choice of join algorithms becomes somewhat more limited. In particular, Algorithm 3 (eager evaluation and lazy expiration) does not apply, Algorithm 4 must perform eager expiration, but cannot predict which tuples will expire before the next re-evaluation and remove them early, and Algorithm 5 must attach *exp* timestamps to each tuple invalidated by a corresponding negative tuple. This underscores the need for update pattern simplification (recall Section 3.5.3), where strict non-monotonic operators are pulled up so that at least some operators have weakest or weak non-monotonic inputs. Of course, if the query references count-based windows, then the input to the join is always strict non-monotonic.

5.3 Join Ordering

Prior to experimentally evaluating the join algorithms, this section deals with choosing an efficient join order from the $n!$ pipelined possibilities (without having to examine all the possibilities). The examples in this section are presented in the context of a nested-loops join with eager evaluation and expiration over time-based windows, but the general solution is applicable to other scenarios as well. Join orders are evaluated using a per-unit-time model of the relative join processing cost, specifically the number of attribute comparisons per unit time [147]. Tuple insertion and expiration costs are omitted as they do not vary across different join orderings within the same algorithm. When estimating join sizes, standard assumptions are made regarding containment of value sets and uniform distribution of attribute values. Containment of value sets states that if the common join attribute a takes on values $a_1, a_2, \ldots, a_v$, then each window must choose its values from a prefix of this list. Given that the stream parameters (e.g., arrival rates and the number of distinct values inside the current window) are expected to fluctuate during the lifetime of a persistent queries, the join order may need to be adjusted periodically; an algorithm for deciding when to change the join order is an orthogonal issue and is not pursued further.

5.3.1 Cost Formulae

Suppose that four windows are to be joined using the global join order S_1, S_2, S_3, S_4. First, the expected intermediate result size of a join of two windows, S_i and S_j, is $\phi_{ij} = \frac{\lambda_i T_i \lambda_j T_j}{\max(v_i, v_j)}$ and the number of distinct values in this intermediate result is $\min(v_i, v_j)$ (these follow from the assumption of containment of value sets). This formula may be extended to multi-joins in an obvious way; for example, the size of the join of S_i, S_j, and S_k is as follows.

$$\phi_{ijk} = \frac{\left(\frac{\lambda_i T_i \lambda_j T_j}{\max(v_i, v_j)}\right)\lambda_k T_k}{\max(\min(v_i, v_j), v_k)} = \frac{\phi_{ij}\lambda_j T_j}{\max(\min(v_i, v_j), v_k)}$$

That is, the size of the join of S_i and S_j is multiplied by the size of the window on S_k, and the result is divided by the maximum of the number of distinct values in the join of S_i and S_j (which is $\min(v_i, v_j)$) and the window on S_k.

The join processing cost can now be computed as follows. When processing new S_1-tuples, the join is evaluated λ_1 times per unit time. Each time, S_2 is scanned for a cost of $\lambda_2 T_2$ tuple accesses. For each S_2-tuple that joins with the new S_1-tuple (the expected number of such tuples is $\frac{\lambda_2 T_2}{\max(v_1, v_2)}$ because one S_1-tuple is joined with $\lambda_2 T_2$ S_2-tuples), S_3 is scanned for a cost of $\lambda_3 T_3$ tuple accesses. Finally, for each S_3-tuple that joins with each composite $S_1 \circ S_2$-tuple (there are $\frac{\left(\frac{\lambda_2 T_2}{\max(v_1, v_2)}\right)\lambda_3 T_3}{\max(\min(v_1, v_2), v_3)}$ such tuples because one S_1-tuple is first joined with $\lambda_2 T_2$ S_2-tuples, followed by joining the result with $\lambda_3 T_3$ S_3-tuples), S_4 is scanned for a cost of $\lambda_4 T_4$ tuple accesses. The total cost per unit time of processing S_1-tuples, call it X_1, is as follows.

$$X_1 = \lambda_1 \left(\lambda_2 T_2 + \frac{\lambda_2 T_2}{\max(v_1, v_2)}\lambda_3 T_3 + \frac{\frac{\lambda_2 T_2}{\max(v_1, v_2)}\lambda_3 T_3}{\max(\min(v_1, v_2), v_3)}\lambda_4 T_4\right) = \lambda_1 \lambda_2 T_2 + \frac{1}{T_1}\left(\phi_{12}\lambda_3 T_3 + \phi_{123}\lambda_4 T_4\right)$$

When processing new S_2-tuples, the cost is calculated as before to get X_2:

$$X_2 = \lambda_2 \left(\lambda_1 T_1 + \frac{\lambda_1 T_1}{\max(v_2, v_1)}\lambda_3 T_3 + \frac{\frac{\lambda_1 T_1}{\max(v_2, v_1)}\lambda_3 T_3}{\max(\min(v_2, v_1), v_3)}\lambda_4 T_4\right) = \lambda_2 \lambda_1 T_1 + \frac{1}{T_2}\left(\phi_{12}\lambda_3 T_3 + \phi_{123}\lambda_4 T_4\right)$$

The cost per unit time of processing new S_3 tuples is:

$$X_3 = \lambda_3 \left(\lambda_1 T_1 + \frac{\lambda_1 T_1}{\max(v_3, v_1)}\lambda_2 T_2 + \frac{\frac{\lambda_1 T_1}{\max(v_3, v_1)}\lambda_2 T_2}{\max(\min(v_3, v_1), v_2)}\lambda_4 T_4\right) = \lambda_3 \lambda_1 T_1 + \frac{1}{T_3}\left(\phi_{13}\lambda_2 T_2 + \phi_{123}\lambda_4 T_4\right)$$

Finally, the cost per unit time of processing new S_4-tuples is:

$$X_4 = \lambda_4 \left(\lambda_1 T_1 + \frac{\lambda_1 T_1}{\max(v_4, v_1)}\lambda_2 T_2 + \frac{\frac{\lambda_1 T_1}{\max(v_4, v_1)}\lambda_2 T_2}{\max(\min(v_4, v_1), v_2)}\lambda_3 T_3\right) = \lambda_4 \lambda_1 T_1 + \frac{1}{T_4}\left(\phi_{14}\lambda_2 T_2 + \phi_{124}\lambda_3 T_3\right)$$

The total cost per unit time is the sum of X_1, X_2, X_3, and X_4.

Table 5.3: Stream parameters in the initial heuristic example

Stream 1	$\lambda_1 = 10,\ T_1 = 100,\ v_1 = 500$
Stream 2	$\lambda_2 = 1,\ T_2 = 100,\ v_2 = 50$
Stream 3	$\lambda_3 = 1,\ T_3 = 200,\ v_3 = 40$
Stream 4	$\lambda_4 = 3,\ T_4 = 100,\ v_4 = 5$

Table 5.4: Sizes of intermediate join results

ϕ_{12}	$\frac{\lambda_1 T_1 \lambda_2 T_2}{\max(v_1,v_2)} = \frac{10\cdot100\cdot1\cdot100}{500} = 200$
ϕ_{13}	$\frac{\lambda_1 T_1 \lambda_3 T_3}{\max(v_1,v_3)} = \frac{10\cdot100\cdot1\cdot200}{500} = 400$
ϕ_{14}	$\frac{\lambda_1 T_1 \lambda_4 T_4}{\max(v_1,v_4)} = \frac{10\cdot100\cdot3\cdot100}{500} = 600$
ϕ_{23}	$\frac{\lambda_2 T_2 \lambda_3 T_3}{\max(v_2,v_3)} = \frac{1\cdot100\cdot1\cdot200}{50} = 400$
ϕ_{24}	$\frac{\lambda_2 T_2 \lambda_4 T_4}{\max(v_2,v_4)} = \frac{1\cdot100\cdot3\cdot100}{50} = 600$
ϕ_{34}	$\frac{\lambda_3 T_3 \lambda_4 T_4}{\max(v_3,v_4)} = \frac{1\cdot200\cdot3\cdot100}{40} = 1500$
ϕ_{123}	$\frac{\phi_{12} \lambda_3 T_3}{\max(\min(v_1,v_2),v_3)} = \frac{200\cdot1\cdot200}{50} = 800$
ϕ_{124}	$\frac{\phi_{12} \lambda_4 T_4}{\max(\min(v_1,v_2),v_4)} = \frac{200\cdot3\cdot100}{50} = 1200$

5.3.2 Join Ordering Heuristic

To motivate the cost differences among various join orders, first suppose that each window has the same number of distinct values. It it sensible to (globally) order the joins in ascending order of the window sizes (in tuples) $\lambda_i T_i$, or average hash bucket sizes $\lambda_i \frac{T_i}{b_i}$ if the sliding windows are stored as hash tables. By placing a small window in the outer for-loop, this strategy minimizes the number of tuples passed down to the inner for-loops for processing. More generally, a sensible heuristic is to assemble the joins in descending order of binary join selectivities, leaving as little work as possible for the inner loops (a join predicate p_1 is more selective than another, p_2, if p_1 produces a smaller result set than p_2). Consider four streams with parameters as shown in Table 5.3 and suppose that hash tables are not available. The intermediate result sizes are shown in Table 5.4. In descending order of the binary join selectivities (i.e., the join which produces the fewest intermediate results is ordered first), the heuristic chooses the ordering S_1, S_2, S_3, S_4. Equations for X_1 through X_4 developed earlier may be used to calculate the cost as follows.

$$X_1 = \lambda_1\lambda_2 T_2 + \frac{1}{T_1}\left(\phi_{12}\lambda_3 T_3 + \phi_{123}\lambda_4 T_4\right) = 10\cdot1\cdot100 + \frac{1}{100}(200\cdot1\cdot200 + 800\cdot3\cdot100) = 3800$$

$$X_2 = \lambda_2\lambda_1 T_1 + \frac{1}{T_2}\left(\phi_{12}\lambda_3 T_3 + \phi_{123}\lambda_4 T_4\right) = 1\cdot10\cdot100 + \frac{1}{100}(200\cdot1\cdot200 + 800\cdot3\cdot100) = 3800$$

$$X_3 = \lambda_3\lambda_1 T_1 + \frac{1}{T_3}\left(\phi_{13}\lambda_2 T_2 + \phi_{123}\lambda_4 T_4\right) = 1\cdot10\cdot100 + \frac{1}{200}(400\cdot1\cdot100 + 800\cdot3\cdot100) = 2400$$

Table 5.5: Stream parameters in the augmented heuristic example

Stream 1	$\lambda_1 = 100,\ T_1 = 100,\ v_1 = 40$
Stream 2	$\lambda_2 = 1,\ T_2 = 100,\ v_2 = 100$
Stream 3	$\lambda_3 = 1,\ T_3 = 100,\ v_3 = 8$
Stream 4	$\lambda_4 = 1,\ T_4 = 100,\ v_4 = 5$

Table 5.6: Stream parameters in the example with two fast streams and two slow streams

Stream 1	$\lambda_1 = 11,\ T_1 = 100,\ v_1 = 200$
Stream 2	$\lambda_2 = 10,\ T_2 = 100,\ v_2 = 100$
Stream 3	$\lambda_3 = 1,\ T_3 = 100,\ v_3 = 65$
Stream 4	$\lambda_4 = 1,\ T_4 = 100,\ v_4 = 20$

$$X_4 = \lambda_4\lambda_1T_1 + \frac{1}{T_4}\left(\phi_{14}\lambda_2T_2 + \phi_{124}\lambda_3T_3\right) = 3 \cdot 10 \cdot 100 + \frac{1}{100}(600 \cdot 1 \cdot 100 + 1200 \cdot 1 \cdot 200) = 6000$$

The total cost per unit time is $3800+3800+2400+6000 = 16000$. For comparison, the worst plan's cost is nearly 90000. In this example, it turns out that the cheapest plans are those with S_1 ordered first in the global order, which suggests a possible augmentation of the heuristic. First, the heuristic computes the cost of the join order according to selectivities, but also calculates the cost of join orders where fast streams are ordered at or near the top of the plan (say in the top one-third of the plan). Finally, the cheapest join order is selected.

To test the augmented heuristic, consider four streams with parameters given in Table 5.5. In the absence of hash tables, the best global order is S_2, S_1, S_3, S_4 and costs 16200. The initial heuristic chooses S_2, S_3, S_4, S_1, whose cost is approximately 42600. Note that Stream 1 is faster than the others, so the improved heuristic moves it up to get S_2, S_1, S_3, S_4, which is the optimal global ordering. Interestingly, moving the fast stream all the way up to get S_1, S_2, S_3, S_4 is worse as it costs 21000. For comparison, the worst plan costs over 135000.

If more than one stream is significantly faster than the others, as in the parameters shown in Table 5.6, then the augmented heuristic still works well. Ordered by ascending join selectivity, the initial plan is S_3, S_4, S_1, S_2, whose cost is 49542. The two fast streams are ordered last, therefore the heuristic moves them up. Moving up S_1 gives S_3, S_1, S_4, S_2 for a cost of 47977, which is the optimal ordering in this scenario, and moving up both S_1 and S_2 gives S_3, S_1, S_2, S_4 for a cost of 51954. Each combination considered by the heuristic costs less than the average cost per unit time over all orderings, which, in this case, is 63362. Again, moving the fast streams all the way to the top to get S_1, S_2, S_3, S_4 or S_2, S_1, S_3, S_4 is not recommended as these two plans cost 68200 and 79000 respectively. In summary, a reasonable heuristic for eager re-evaluation of the multi-way join is to initially order the joins in descending order of their selectivities. If one or more streams are faster than the others, then it is also beneficial to consider orderings where the fast streams are moved up the join order (but not all the way to the top). The number of orderings considered is on the order of n^f where f is the number of "fast" streams.

5.4 Experimental Results

5.4.1 Experimental Setting

Experiments were performed to validate the join ordering heuristic and compare the performance of the proposed join algorithms. The algorithms were built into the update-pattern-aware query processor from Chapter 3 and tested in the same experimental environment. However, rather than using a real packet header stream, a synthetic packet stream is used in order to control the distribution of join attribute values. The stream is produced as follows. A continuous for-loop is connected to the query processor and generates one tuple per iteration from a random stream i with probability equal to $\frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}$. The tuple is given a timestamp equal to the current loop index and a join attribute value chosen uniformly at random from the set $\{1, 2, \ldots, v_i\}$ (other fields are not used in the experiments and assigned random values). This procedure simulates relative stream rates and guarantees containment of value sets. Note that uniform distribution of join attribute values leads to a uniform distribution of lifetime lengths of the results, therefore the proportion of lost tuples is fairly small (recall the discussion in Section 5.2.3). The stream is then pushed into a query plan consisting of one multi-way join operator over time-based windows. The join state is therefore implemented as a FIFO queue (or a hash table whose buckets are FIFO queues). All hash functions are simple modular divisions by the number of hash buckets. Each experiment is repeated ten times and the average processing time per 1000 input tuples is reported.

5.4.2 Validation of Cost Model and Join Ordering Heuristic

Recall the join ordering example with stream parameters given in Table 5.5. Ordering the joins by selectivity gives the order S_2, S_3, S_4, S_1. However, stream one is faster than the others, therefore the improved heuristic additionally considers S_2, S_1, S_3, S_4, which is the optimal global ordering, and S_1, S_2, S_3, S_4. Figure 5.2 illustrates the processing time of four join orderings given the same stream parameters, normalized with respect to the optimal ordering. In addition to the predicted cost of each ordering, the following eight algorithms are tested: eager execution, eager execution with lazy expiration (every ten time units), lazy execution (every ten time units), lazy execution with restore semantics (abbreviated "lazyR"), and the same four algorithms, but using hash tables with ten buckets on each stream. The relative costs of the four join orders tested are similar to the predicted costs for each algorithm, although the differences among the four orderings are not quite as high as predicted. This is because the cost model only considers join processing and ignores other costs such as insertion of new tuples and expiration of old tuples. That is why performing eager evaluation but lazy expiration (third set of bars from the left) matches the predicted costs the most closely. On the other hand, lazy evaluation and the four hash-based joins show reduced differences among the four join orderings because join processing costs make up a smaller fraction of the overall query processing cost.

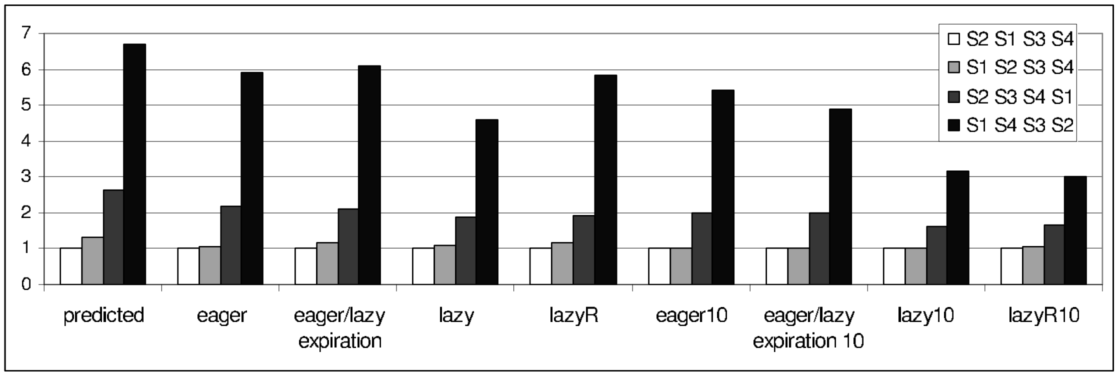


Figure 5.2: Validation of join ordering heuristic

5.4.3 Relative Performance of Join Algorithms

The next set of experiments highlights the cost differences among the proposed join algorithms. The stream parameters are the same as in the previous experiment, but the join ordering is fixed at S_2, S_1, S_3, S_4 (i.e., the optimal ordering). Nested-loop-based algorithms, namely eager evaluation, lazy evaluation, and lazy evaluation with restore semantics, are summarized in Figure 5.3. Their costs in units of milliseconds are graphed as function of Δ, which is understood to be the expiration interval in the context of eager evaluation and re-evaluation interval in the context of lazy evaluation. Note that only eager evaluation includes a data point at $\Delta = 0$, denoting eager expiration whenever a new tuple arrives for processing. As expected, lazy join evaluation is more efficient than eager evaluation, though there is a noticeable penalty for enforcing restore semantics. As Δ increases, the cost of eager evaluation increases. This means that the savings in expiration costs are outweighed by the added complexity of Algorithm 2 as compared to Algorithm 3, namely padding join predicates with timestamp comparisons in order to ensure that expired tuples do not produce any new join results. On the other hand, the two lazy algorithms become more efficient as Δ increases, both in absolute terms and relative to eager evaluation. This is because new tuples having the same values of the join attribute are processed together, and more duplicates are expected in the buffer when Δ is large. In particular, this enhancement speeds up the lazy join with restore semantics as Δ goes up despite the fact that more results are being lost and must be recovered.

The cost of hash-based algorithms is shown in Figure 5.4, with the three lines on top corresponding to hash tables with two buckets on each stream and the three lines on the bottom denoting hash tables with five buckets on each stream. As expected, the query is now faster to execute. First, the performance results with two buckets per hash table are similar to those from Figure 5.3 with one noticeable difference: for small values of Δ, the lazy join with restore semantics performs worse as Δ increases. This is because the overhead of restore semantics is higher than the savings gained by lazy evaluation. When Δ is small, there are few tuples in the buffer with the same values of the join attribute and therefore there are few opportunities for

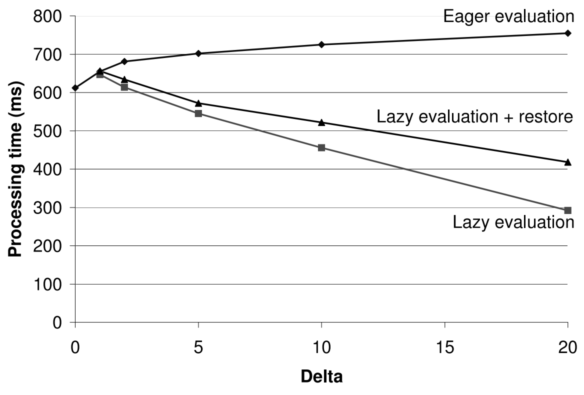


Figure 5.3: Cost of nested-loops algorithms

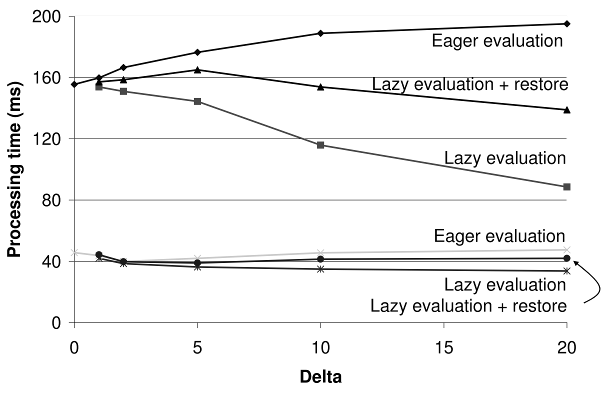


Figure 5.4: Cost of hash-based algorithms

batching new tuples and probing the other windows once per batch.

Moving on to the results using five buckets per hash table, the query processing costs are now much lower and the performance differences among the join algorithms less noticeable (a straightforward technique for further improvement of the join processing cost is to allocate more hash buckets to windows expected to store more tuples). Interestingly, eager evaluation improves as Δ grows from zero to five, but then begins to perform worse. This is because frequent expiration is now expensive as all five hash buckets must be probed separately in order to determine if any of their tuples have expired. However, when Δ is large, the expiration costs are less severe, but the algorithm must execute timestamp comparisons during the probing state. Therefore, the performance of eager evaluation eventually declines, as was observed in the previous experiments. The other two algorithms (lazy evaluation with restore or no-restore semantics) become slightly more efficient as Δ grows, but the difference is now far smaller because using five buckets per hash table already cuts down the join processing costs significantly and the additional improvement of lazy evaluation is relatively small.

5.4.4 Lessons Learned

The above experiments have shown that the best way to reduce the cost of a query containing sliding window joins is to store the join inputs as hash buckets. In some cases (namely, lazy evaluation), additional improvement can be made by increasing the re-evaluation interval, although this is beneficial only if the batch of newly arrived tuples is expected to contain duplicate values of the join attribute. Furthermore, it was shown that the overhead of enforcing restore semantics has a noticeable impact on join performance, but lazy evaluation with restore semantics is still more efficient than eager evaluation, especially if the re-evaluation interval Δ is large.

Chapter 6

Sliding Window Aggregation: Finding Frequent Items

6.1 Introduction

This is the second of two chapters on sliding window query operators, focusing on periodically refreshed `TOP k` queries over sliding windows on Internet traffic streams [108]. Internet traffic [161, 200] and Web page popularity patterns [4, 7] have been observed to be highly skewed (i.e., obeying a Zipf-like distribution [278]). This implies that most of the outgoing (or incoming) bandwidth is consumed by (or directed to) a small set of heavy users (or popular destinations). Hence, queries that return a list of frequently occurring items[1] are important in the context of traffic engineering, routing system analysis, customer billing, and detection of anomalies such as denial-of-service attacks. For instance, an Internet Service Provider (ISP) may be interested in monitoring streams of IP packets originating from its clients and identifying the users who consume the most bandwidth during a given time interval (see, e.g., [72, 86, 191] for additional motivating examples). These types of queries, in which the objective is to return a list of the most frequent items (called *top-k queries* or *hot list queries*) or items that occur above a given frequency (called *threshold queries*), are generally known as *frequent item queries.* However, to make such analysis meaningful, bandwidth usage statistics should be kept for only a limited amount of time—for example, a sliding window of recently arrived data—before being replaced with new measurements. Failure to remove stale data leads to statistics aggregated over the entire lifetime of the stream, which are unsuitable for identifying recent usage trends.

If the entire window fits in main memory, then answering threshold queries over sliding windows is simple; it suffices to maintain frequency counts of each distinct item in the window and update the counters as new items arrive and old items expire. If periodically refreshed answers are acceptable (as is assumed in this chapter), then a basic interval synopsis may be used (recall the discussion in Section 2.3.4), storing frequency counters in each interval. To speed up query

[1]The terms item, item type, value, and category are used interchangeably in this chapter.

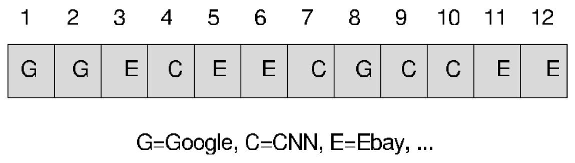


Figure 6.1: A basic window synopsis storing the most popular URL in each sub-window

evaluation, a set of global counters may additionally be stored and updated whenever a new interval is added to the synopsis (and the oldest interval is dropped). However, Internet traffic on a high-speed link arrives so fast that useful sliding windows may be too large to fit in memory and the system may not have sufficient resources to keep up with the stream. As discussed in Section 2.3.4, one way to reduce the memory requirements of a basic window synopsis is to store sketches in each sub-window rather than complete sets of frequency counters. This chapter evaluates an alternate solution: rather than approximating the complete frequency distribution using sketches, it is more space-efficient to store the exact frequency counts of only the most frequent packet types observed in each sub-window. In effect, by storing only a constant amount of information per sub-window, the `TOP k` aggregate, which is holistic, is treated as if it were distributive (recall Section 2.3.4). Consequently, the technical problem with this approach is that there is no obvious rule for merging constant-size partial information stored in sub-windows into a final answer.

To illustrate the technical challenges of the proposed approach, consider the simple example illustrated in Figure 6.1, showing a basic window synopsis of length 12 minutes that stores the most popular destination URL in each individual minute. E-bay was the most popular URL in five of the 12 minutes, CNN in four, and Google in three. Based upon this information, it is not clear what the most popular URL was over the entire 12-minute window. In general, if each sub-window stores counts of its top k categories, then one cannot say that any item appearing in any of the top-k lists is one of the k most frequent types in the sliding window—a bursty packet type that dominates one sub-window may not appear in any other sub-windows at all. It is also not necessarily true that a frequent item must have appeared in at least one top-k sub-window list—if k is small, say $k = 3$, then it is possible to ignore a frequent item type that consistently ranks fourth in each sub-window and therefore never appears on any of the top-k lists. Fortunately, it will be shown empirically that these problems are far less serious if the sliding window conforms to a power-law-like distribution, in which case several very frequent categories are expected (e.g., popular source IP addresses or protocol types), which are likely to be repeatedly included in nearly every top-k list.

There has been recent work on answering top-k queries over sliding windows, with the objective of refreshing the answer whenever a new item arrives or an old item expires from the window [191]. The algorithms presented therein store all the tuples inside the window and therefore are not compatible with the goals of this chapter (i.e., periodic query evaluation using limited storage). Furthermore, computing top-k queries from multiple sub-window top-k lists is similar to rank aggregation in conventional (possibly distributed) DBMSs [135]. The main idea is to combine multiple ranked lists, which are assumed to be available in their entirety, into a single list. The difference in the DSMS context is that a top-k query does not have access to complete sets of frequency counters in each sub-window and therefore must calculate the overall top-k list over the entire sliding window based upon limited information. Finally, [24] presents a framework for distributed top-k monitoring, but their goal is to minimize the amount of data transferred from distributed sources to a central processing system. Furthermore, they do not consider the sliding window model.

In the remainder of this chapter, an algorithm for finding frequent items in on-line data streams is presented in Section 6.2 and experimentally evaluated in Section 6.3. The proposed algorithm identifies frequently occurring items in the sliding windows and, to some extent, also estimates their true frequencies. It is deterministic, uses limited memory (each sub-window stores a constant amount of data), requires constant processing time per packet (amortized), makes only one pass over the data, and is shown to work well when tested on TCP traffic logs.

6.2 Proposed Solution

6.2.1 Algorithm Description

This section proposes the following algorithm that stores a top-k list in each sub-window. Assume that a single sub-window fits in main memory, within which item frequencies may be counted exactly. Let δ_i be the frequency of the kth most frequent item in the ith sub-window. Then $\delta = \sum_i \delta_i$ is the upper limit on the frequency of an item type that does not appear on any of the top-k lists. Now, the reported frequencies for each item present in at least one top-k list are summed up and if there exists a category whose reported frequency exceeds δ, then that this category is guaranteed to have a true frequency of at least δ. The pseudocode is given as Algorithm 6, assuming that N is the (count-based) window size, b is the number of elements per sub-window, and N/b is the total number of sub-windows. An updated answer is generated whenever the window slides forward by b packets.

Let $k = 3$ and let $a, b, c, \ldots$ be distinct item types. Figure 6.2 provides an example of a single execution of Algorithm 6 over a sliding window consisting of 13 sub-windows. Note that as shown above, Algorithm 6 assumes that all sub-windows have the same number of items, as is the case in count-based windows. However, this assumption is not necessary to ensure the algorithm's correctness—line 2 may be replaced with another condition for emptying the queue, say every t time units. Therefore, Algorithm 6 may be used with time-based windows without any modifications. However, the algorithm is not compatible with inputs resulting from weak or strict non-monotonic query plans because these do not satisfy the FIFO property. The remainder of this section maintains the assumption of equal item counts in sub-windows in order to simplify the analysis.

Algorithm 6 Frequent

Input: number of tuples in the window N, number of elements per sub-window b
Local variables: integer δ, local counters, global counters, queue Q, summary S

```
1  loop
2      for each tuple $e$ in the next $b$ tuples
3          if a local counter exists for the type of tuple $e$ then
4              increment the local counter
5          else
6              create a new local counter for this type and set it equal to 1
7          end if
8      end for
9      add a summary $S$ containing identities and counts of the $k$ most frequent items
       to the back of queue $Q$
10     delete all local counters
11     for each type named in $S$
12         if a global counter exists for this type then
13             add to it the count recorded in $S$
14         else
15             create a new global counter for this type and set it equal to the count recorded in $S$
16         end if
17     end for
18     add the count of the $k$th largest type in $S$ to $\delta$
19     if $sizeOf(Q) > N/b$ then
20         remove the summary $S'$ from the front of $Q$ and subtract the count of the $k$th
           largest type in $S'$ from $\delta$
21         for all types named in $S'$
22             subtract from their global counters the counts recorded in $S'$
23             if a counter is decremented to zero then
24                 delete it
25             end if
26         end for
27         output the identity and value of each global counter $> \delta$
28     end if
29 end loop
```

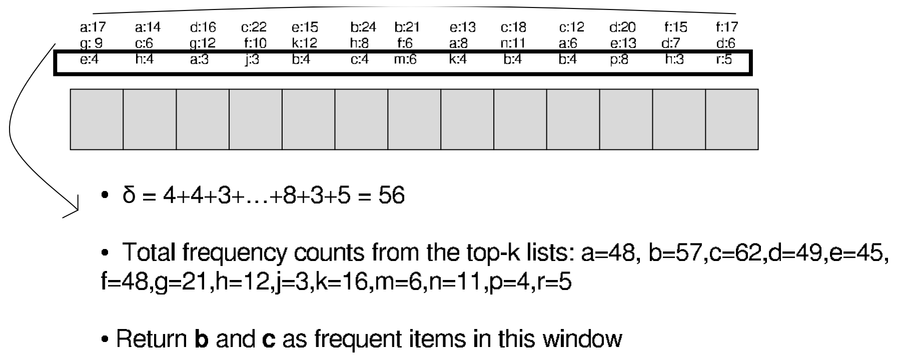


Figure 6.2: Example of a single execution of Algorithm 6

6.2.2 Analysis

Algorithm 6 accepts three parameters: N, b, and k[2]. The choice of b is governed by the latency requirements of the application: choosing a small value of b increases the frequency with which new results are generated. However, the amount of available memory dictates the maximum number of sub-windows and the value of k.

The space requirement of Algorithm 6 consists of two parts: the working space needed to create a summary for the current sub-window and the storage space needed for the top-k lists. Let d be the number of distinct item types in a sub-window (the value of d may be different for each sub-window, but this point is ignored in order to simplify the analysis) and D be the number of distinct values in the sliding window. In the worst case, the working space requires d local counters of size $\log b$. For storage, there are N/b sub-window summaries, each requiring k counters of size at most $\log b$. There are also at most kN/b global counters of size at most $\log N$. This gives a total worst-case space bound of $O(d \log b + \frac{kN}{b}(\log b + \log N))$. The time complexity of Algorithm 6 is $O(\min(k, b) + b)$ for each pass through the outer loop. Since each pass consumes b arriving elements, this gives $O(1)$ amortized time per element.

Algorithm 6 may return false negatives. Consider an item that appears on only a few top-k lists, but summing up its frequency from these top-k lists does not exceed δ. This item may be sufficiently frequent in other sub-windows (but not frequent enough to register on the top-k lists of these other windows) that its true frequency count exceeds δ. The obvious solution for reducing the number of false negatives is to increase k[3], but this also increases space usage. Alternatively, decreasing b increases the number of sub-windows, which may also help eliminate false negatives.

Another possible downside of Algorithm 6 is that if k is small, then δ may be very large and the algorithm will not report any frequent flows. On the other hand, if k is large and each synopsis contains items of a different type (i.e., there are very few repeated top-k "winners"),

[2]Note that N and b must be specified in units of time in time-based windows.

[3]A conceptually similar idea is used in [267] in the context of incremental maintenance of a top-k view in a conventional DBMS: maintain a top-k' view instead, where $k' > k$.

then the algorithm may require a great deal of storage space, perhaps as much as the size of the sliding window. Notably, when b is only slightly larger than k, then there may be fewer than k distinct items in any sub-window. In this case, Algorithm 6 will track the exact frequencies of most (if not all) of the distinct packet types.

6.3 Experimental Results

6.3.1 Experimental Setup

Algorithm 6 was tested in the same experimental environment and using the same TCP traffic trace as in Chapter 3. The trace contains 1647 distinct source IP addresses, which are treated as distinct item types. The value of N is set to 100000 and three values of b are considered: $b = 20$ (5000 sub-windows in total), $b = 100$ (1000 sub-windows in total), and $b = 500$ (200 sub-windows in total). The value of k varies from one to ten. In each experiment, one hundred starting points within the trace are chosen at random and packets following these starting points are used to form an input stream. A brute-force algorithm is also executed over the same input in order to calculate the true item type frequencies. The quantities being measured are the average threshold δ, the average number of over-threshold flows reported, accuracy, and space usage over one hundred trials, as shown in Figure 6.3.

6.3.2 Accuracy

Recall that Algorithm 6 identifies a category as being over the threshold δ if this category's frequency count recorded in the top-k synopses exceeds δ. As k increases, the frequency of the kth most frequent item decreases and the overall threshold δ decreases, as seen in Figure 6.3 (a). Furthermore, increasing the number of sub-windows by decreasing b increases δ as smaller sub-windows capture burstiness on a finer scale. Consequently, as k increases, the number of packet types that exceed the threshold increases, as seen in Figure 6.3 (b) and (c). The former plots the number of over-threshold IP addresses, while the latter shows the number of IP addresses that were identified by the algorithm as being over the threshold. For example, when $k = 5$, the threshold frequency is roughly five percent (Figure 6.3 (a)) and there are between three and four source IP addresses whose frequencies exceed this threshold (Figure 6.3 (b)).

It can be seen in Figure 6.3 (b) and (c) that Algorithm 6 does not identify all the packet types that exceed the threshold (there may be false negatives, but recall that there are never any false positives). Figure 6.3 (d) shows the percentage of over-threshold IP addresses that were identified by Algorithm 6. The general trend is that for $k \geq 3$, at least 80 percent of the over-threshold IP addresses are identified. Increasing the number of sub-windows (i.e., decreasing the sub-window size b) also improves the chances of identifying all of the above-threshold packet types. For instance, if $k > 7$ and $b = 20$, then false negatives occur very rarely because all of the frequently occurring item types appear on many of the top-k lists within individual sub-windows.

6.3.3 Space Usage

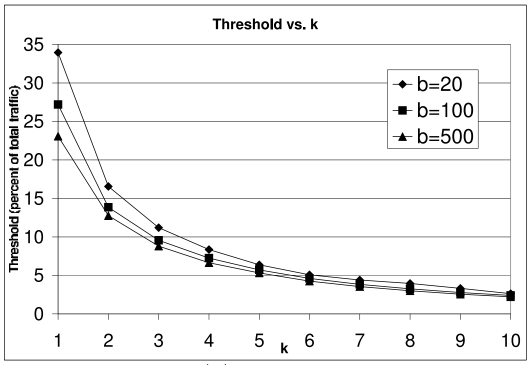
Figure 6.3 (e) shows the space usage of Algorithm 6 in terms of the number of attribute-value, frequency-count pairs that need to be stored. Recall that the sliding window size in the experiments is 100000, which may be considered as a rough estimate for the space usage of a naive technique that stores the entire window. The space usage of Algorithm 6 is significantly smaller, especially when b is large and/or k is small. Because a top-k list must be stored for each sub-window, the number of sub-windows has the greatest effect on the space requirements.

6.3.4 Precision

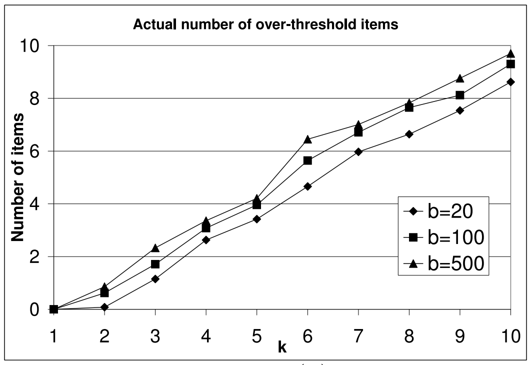
Recall from Figure 6.3 (d) that Algorithm 6 may report false negatives. However, unreported frequent types typically have frequencies that only slightly exceed the threshold, meaning that the most frequent types are always reported. Furthermore, the reported frequency estimates were in many cases very close to the actual frequencies, meaning that the reported frequent IP addresses were arranged in the correct order (though item types with similar frequencies were often ordered incorrectly). To quantify this statement, Figure 6.3 (f) plots the average relative error (i.e., the difference between the measured frequency and the actual frequency divided by the actual frequency) in the frequency estimation of the over-threshold IP addresses for ten values of k and three values of b. The relative error decreases as k increases and as b decreases. For example, when $b = 20$ and $k \geq 7$, then the average relative error is below two percent. Therefore, the reported IP addresses are nearly always ordered correctly, unless there are two IP addresses with frequencies within two percent of each other, and only those IP addresses which exceed the threshold by less than two percent may remain unreported.

6.3.5 Lessons Learned

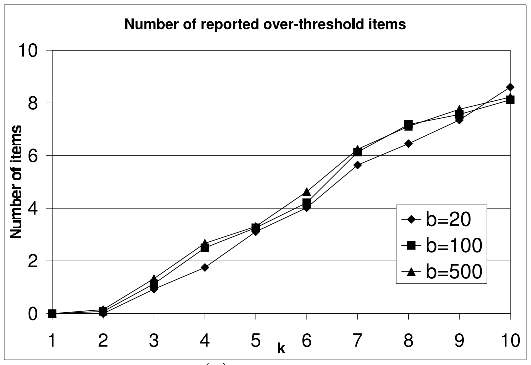
Algorithm 6 works well as an identifier of frequent items and, to some extent, their approximate frequencies, when used on Internet traffic streams. As expected, increasing the size of the top-k synopses increases the number of frequent flows reported, decreases the number of false negatives, and improves the accuracy of the frequency estimates. Increasing the number of sub-windows reduces the refresh delay, decreases the proportion of false negatives and increases the accuracy of the frequency estimates. However, space usage grows when either k increases or b decreases.



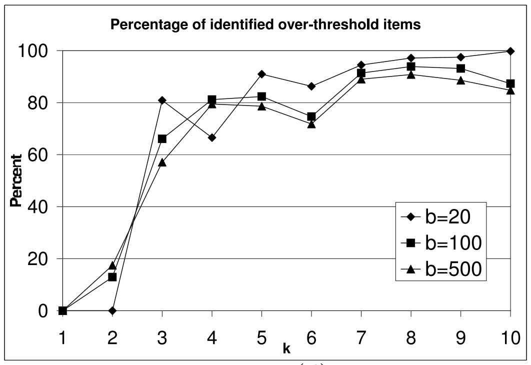
(a)



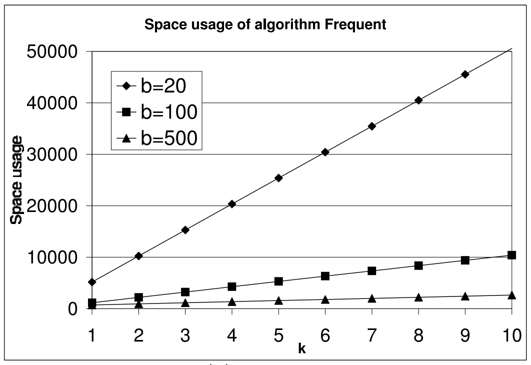
(b)



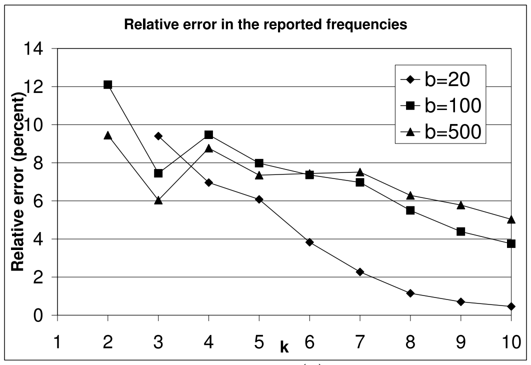
(c)



(d)



(e)



(f)

Figure 6.3: Analysis of Algorithm 6. Part (a) shows the average value of the threshold δ, part (b) shows the number of packet types whose frequencies exceed the threshold, part (c) graphs the number of packet types reported as exceeding the threshold, part (d) shows the percentage of over-threshold packets identified, part (e) plots the space usage, and part (f) shows the relative error in the frequency estimates of over-threshold items (all as a function of k).

Chapter 7

Concurrency Control in Periodic Queries over Sliding Windows

7.1 Introduction

Thus far, the notion of update pattern awareness was introduced, and its significance in the context of sliding window maintenance and query processing was explained. This chapter examines concurrency issues arising from simultaneous execution of periodic queries and periodic window-slides, and presents an update-pattern-aware transaction scheduler [107].

Recall from Figure 2.4 that a periodically-sliding window can be implemented as a circular array of sub-windows, with periodic window updates replacing the oldest sub-window with a batch of newly arrived data. As the windows slide forward, a DSMS executes a dynamic workload of persistent and one-time queries. Assume that query execution involves scanning a window exactly once, one sub-window at a time (this assumption will be justified in Section 7.2). Combined with periodic window movements[1], DSMS data access can be modeled in terms of two atomic operations: sub-window scan (read) and replacement of the oldest sub-window with new data (write). Thus, a window update is a single write operation[2], whereas a query is a sequence of sub-window read operations such that each sub-window is read exactly once.

A window may slide while being accessed by a query, leading to a *read-write conflict*. Consider a sequence of operations illustrated in Figure 7.1(a), where the processing times of window updates (U) and queries (Q_1, Q_2, and Q_3) are shown on a time axis. This represents an ideal scenario, where it is possible to execute all three queries between every pair of window updates, thereby avoiding read-write conflicts. However, the system environment, such as the query workload, stream arrival rates, and availability of system resources, can change greatly during the lifetime

[1]In the remainder of this chapter, the terms window update, window movement, and window-slide will be used interchangeably.

[2]More precisely, a window update is a single logical write operation that logically performs a series of delete operations in order to clear out the oldest sub-window, and a series of insert operations that write the newly arrived tuples from the buffer into the empty sub-window.

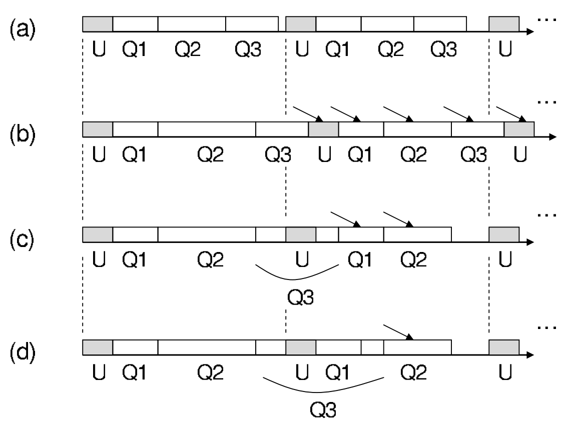


Figure 7.1: Examples of query and window update sequences in a DSMS

of a persistent query. Thus, a more realistic sequence is shown in Figure 7.1(b), where Q_2 takes longer to execute than expected. Q_3 is still running when the second update is ready to be applied, causing a delay in performing the update, and, in turn, causing another read-write conflict when Q_3 is re-executed and the third update is about to take place.

It may appear that read-write conflicts can be prevented by increasing the time interval between window updates, i.e., the sub-window size. However, all sub-windows must have the same size (either in terms of time or the number of tuples, depending on the type of sliding window) so that the overall window size is fixed at all times. Therefore, either the system must be taken off-line to re-partition the entire window, or two sets of sub-windows must be maintained during the transition period until the window "rolls over" and all the sub-windows have the new size. The first case is inappropriate for an on-line DSMS, whereas the second solution does not immediately eliminate read-write conflicts until the transition period is over.

Existing data stream solutions avoid read-write conflicts by serially executing queries and window movements. In other words, a query locks the window that it is scanning in order to prevent concurrent window movements. This approach eliminates the need for (and performance overhead of) sophisticated concurrency control solutions. However, interleaved execution of updates while a window is being scanned by a query allows more up-to-date answers to be generated, provided that the following issue is resolved (and the drop in throughput due to the overhead of concurrency control is minimal). Consider suspending the processing of Q_3 in order to perform a window update, as in Figure 7.1(c). Recall that each query is assumed to perform a sequence of atomic sub-window reads, therefore it may be interrupted after it has read one or more sub-windows. It must be ensured that when resumed, Q_3 can correctly read the updated window state. If so, then the answer of Q_3 is slightly delayed (by the time taken to perform the update), but it is more up-to-date because it reflects the second update as well as the first. Otherwise, the end result is worse than in Figure 7.1(b), because the answer of Q_3 is delayed, but it is still not up-to-date. Another example is illustrated in Figure 7.1(d), where Q_3 is suspended not only to perform a window update, but also to run Q_1 immediately afterwards. This is desirable if Q_1 is an important query that requires an immediate and up-to-date answer.

This chapter studies concurrency control issues in a DSMS with periodic window movements, periodic executions of persistent queries, and on-demand snapshot querying. As motivated above, the goal is to provide query scheduling flexibility and guarantee up-to-date results with minimal loss in throughput due to the overhead of concurrency. The particular contributions of this chapter are as follows.

- By modeling window movements and queries as transactions consisting of atomic sub-window reads and writes, this chapter extends concurrency theory to cover queries over periodically-advancing windows. It is shown that conflict serializability is not sufficient in the presence of interleaved queries and window movements because some serialization orders produce incorrect answers.
- A new isolation level is proposed, which is stronger than both snapshot isolation and conflict serializability in that it restricts the permissible serialization orders.
- An update-pattern-aware transaction scheduler is designed that efficiently enforces the desired isolation level. The insight behind the scheduler is to predict which sub-window will be overwritten next and ensure that the new copy of it is read by concurrent queries. The scheduler is proven to be optimal in the sense that it aborts the smallest possible number of transactions while allowing immediate (optimistic) scheduling of window updates.
- An experimental evaluation of the transaction scheduler shows improved query freshness and response times with a minimal drop in throughput.

The remainder of this chapter is organized as follows. Section 7.2 explains the system model and assumptions, and motivates why concurrency control is an issue in a DSMS. Section 7.3 defines new isolation levels for DSMS transactions, and Section 7.4 presents an update-pattern-aware transaction scheduler for enforcing them. Section 7.5 presents experimental results, and Section 7.6 compares the contributions of this chapter to previous work.

7.2 Motivation and Assumptions

7.2.1 Data and Query Model

Each stream is assumed to be bounded by a time-based window of length nt, stored as a circular array of n sub-windows, each spanning a time-length of t. Every t time units, the oldest sub-window is replaced with a buffer containing incoming tuples that have arrived in the last t time units. The value of t is assumed to be significantly larger than the time taken to perform a window update (otherwise, the system would spend all of its time advancing the windows rather than executing queries). Each persistent query Q specifies its desired re-execution frequency, which must be a multiple of t, i.e., Q will be scheduled for re-execution every m window updates, where $1 \leq m < n$. Q also specifies a window size of jt for some j such that $1 \leq j \leq n$ (that is, queries over windows shorter than nt are allowed, provided that the window lengths are multiples

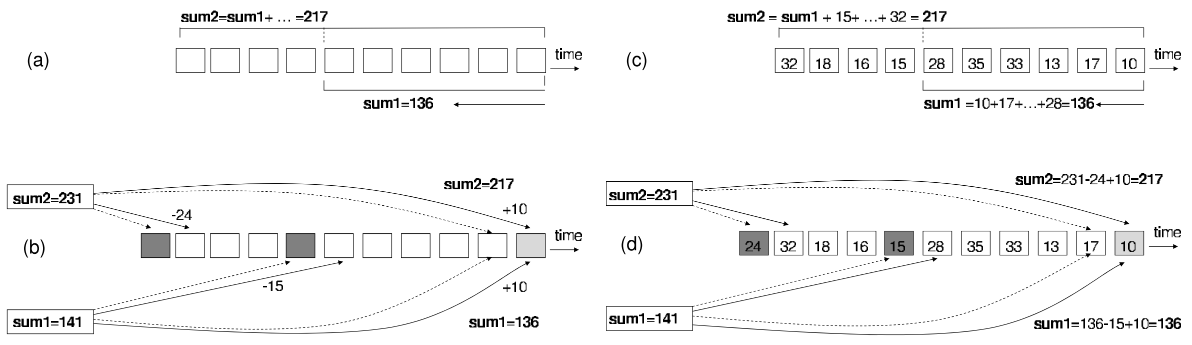


Figure 7.2: Four techniques for computing sliding window sums

of the sub-window length t). It is assumed that the plan for Q is weakest or weak non-monotonic; strict non-monotonic update patterns are not considered in this chapter. The DSMS attempts to execute all the queries with the desired frequencies, but it cannot guarantee that this will be the case at all times due to unpredictable system conditions (scheduling re-executions of persistent queries will be covered in detail in Chapter 8).

Periodic query execution strategies may be classified into four general types: *window scan* (WS), *incremental scan* (IS), *synopsis scan* (SS), and *incremental synopsis scan* (ISS). To illustrate them, suppose that two sums over the same attribute (and stream) are being computed, *sum1* over a window of size $6t$ (six sub-windows) and *sum2* over a window of size $10t$ (ten sub-windows).

WS is a default access path that scans the entire window (or windows), one sub-window at a time, and computes a query from scratch. As shown in Figure 7.2(a), if sub-windows are read from youngest to oldest, then the sum over the shorter window may be re-used when computing the sum over the longer window.

One way to speed up the execution of some types of periodic queries is to store permanent state that allows answers to be refreshed incrementally. IS, shown in Figure 7.2(b), stores the previously calculated answer of each query and a pair of pointers denoting the window over which the answer was computed (indicated by the dotted arrows). Upon re-evaluation, the query scans (all the tuples in) only those sub-windows which have been added or expired since the last re-execution; a sub-window may be deleted only if all the interested queries have advanced their pointers forward. To compute new sums, the sum of the tuples in the new sub-window (lightly shaded) are added and the sum of expired tuples (darkly shaded) is subtracted. This strategy applies to *subtractable* queries (recall Section 2.3.4), where the contribution of expired tuples may be subtracted from the stored answer. Furthermore, as discussed in Section 2.2.2, incremental computation of holistic aggregates requires that each query store a list of distinct values occurring in its window and their multiplicities (it is not sufficient to store the previous answer). Observe that expired tuples are found in different sub-windows, depending upon the window size of the query. For instance, in Figure 7.2(b), expired tuples with respect to a window of size $10t$ (needed by *sum2*) are found in the left-most (oldest) sub-window, whereas expired tuples with respect to a shorter window of size $6t$ (needed by *sum1*) are found four sub-windows to the right. Therefore, if the workload includes many sum queries over windows of different sizes, then the entire window may need to be scanned in order to re-compute all the answers, as in WS.

Rather than storing separate state per query, SS maintains sliding window synopses, as already discussed in Section 2.3.4. An example is shown in Figure 7.2(c), illustrating shared processing of two sums over different window sizes given a basic interval synopsis (running synopses are not considered in this chapter as they apply only to subtractable aggregates). Note that shared query evaluation involves merging pre-aggregated sub-windows, from youngest to oldest.

Finally, ISS is a combination of IS and SS. As illustrated in Figure 7.2(d), it stores synopses as well as individual query state. To refresh the answer of the two sum queries, it suffices to look up the (pre-aggregated) sums of new and expired tuples. That is, only the summaries of new or expired sub-windows are accessed. Similarly to IS, ISS is suitable only for subtractable queries.

7.2.2 Motivation for Study of Concurrency Control

In the remainder of this chapter, WS and SS are used for query evaluation. First, WS is the default access plan for one-time queries, which are not known ahead of time and therefore may not find any applicable synopses. Moreover, WS may be the only option for initial evaluation of a new persistent query. In this case, the window may be scanned to produce the initial answer and optionally, to build a synopsis that the query can use for future re-executions.

Next, observe that SS and IS have comparable space usage, provided that the workload includes similar queries over various window sizes. To see this, note that IS requires each query to store permanent state, even if many queries are identical except for their window sizes. For example, in Figure 7.2(b), *sum2* cannot be used to help compute *sum1*. On the other hand, SS stores one piece of data per sub-window (e.g., a pre-aggregated value or a list of frequency counters), whose size is at most as large as the state of an individual query. In addition, IS must retain some expired sub-windows until each query has subtracted the contribution of expired tuples from its stored answer. In contrast, SS simply overwrites the oldest sub-window interval with a new value (or new counters). Moreover, SS yields faster processing times (IS must read all the sub-windows containing new or expired tuples) and may be used by non-subtractable queries.

Finally, note that the space usage of ISS (window summaries and query state) may be prohibitive. First, the number of synopses stored by the DSMS may be large. Moreover, one-time queries may need to suspend any persistent queries currently running, as in Figure 7.1(d). In this case, the system must set aside state space for the suspended queries so that they can resume later, and reserve state space for any potential one-time queries.

Recall Figure 7.1(c) and (d). A window must be allowed to slide while being accessed by a query and the query must read the new window state correctly. In terms of concurrency control, a query cannot read old tuples that expire before the query terminates. This problem does not

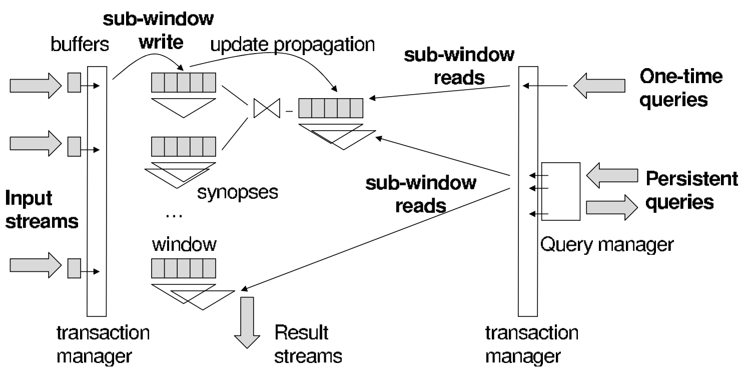


Figure 7.3: Assumed system architecture

exist in IS and ISS because a window update does not actually replace expired data. Instead, expired sub-windows must be retained until each query has subtracted the contribution of expired tuples from its stored answer. However, WS and SS both scan the window (or its synopsis), one sub-window at a time. Queries may see an old copy of the window if they have read an old sub-window that is about to be overwritten. Expired sub-windows could be retained, but this requires more space and processing time (old sub-windows may need to be re-scanned by queries in order to remove the contribution of expired tuples from the answer currently being computed). Even then, it is desirable to prevent double-scanning.

7.2.3 System Model

The assumed system architecture is illustrated in Figure 7.3. It is an expanded view of the shaded part of the architecture from Figure 1.2 with the local storage component omitted. Let $w[i]$ denote the replacement of the ith sub-window with newly arrived data, for $0 \leq i \leq n-1$. Each data stream generates periodic write-only transactions T_j in subscript order, defined as $T_j = \{w_j[j \bmod n], c_j\}$, where c_j signifies the commit of T_j. They are processed by the transaction manager, which propagates updates to all the synopses that reference the window or a join of this window with another. For each stream, the transaction manager initially executes T_0 through T_{n-1} to fill up the windows. Thereafter, each T_j has the effect of moving the window forward by one sub-window. In order to ensure that queries have access to the latest data, the transaction scheduler executes (and immediately commits) each T_j as soon as the buffer is full.

One-time queries are executed by accessing a suitable synopsis, if available, or scanning the underlying window(s). Persistent queries are re-executed periodically throughout their lifetimes by using existing synopses or building new synopses, as will be discussed in more detail in Chapter 8 in the context of multi-query optimization. The interface between the query manager and the transaction scheduler consists of read-only transactions corresponding to re-execution of one or several similar queries. Let $r[i]$ be a scan (read) of the ith sub-window, or its summary, for $0 \leq i \leq n-1$ (without loss of generality, in the remainder of this chapter, either of these will be referred to as a sub-window). A one-time query or a particular re-execution of one or more persistent queries is a read-only transaction T_{Qk}, defined as $T_{Qk} = \{r_{Qk}[0], r_{Qk}[1], \ldots, r_{Qk}[n-1], c_{Qk}\}$ if T_{Qk} commits successfully (otherwise, T_{Qk} will include an abort operation a_{Qk} and will not contain the commit operation c_{Qk}). That is, each T_{Qk} performs a scan of a window, sub-result, or summary, by reading each sub-window exactly once. Queries over windows shorter than nt may be defined similarly as transactions reading the appropriate number of sub-windows, starting with the youngest sub-window in the circular array. For example, if the zero-th sub-window is currently the youngest, than a (committed) query over a window of length $(n-1)t$ corresponds to $T_{Qk} = \{r_{Qk}[0], r_{Qk}[1], \ldots, r_{Qk}[n-2], c_{Qk}\}$. That is, the oldest sub-window, which currently is the $(n-1)$st sub-window, is not read. However, if the fifth sub-window is currently the youngest, then the same query corresponds to $T_{Qk} = \{r_{Qk}[5], r_{Qk}[6], \ldots, r_{Qk}[n], r_{Qk}[0], r_{Qk}[1], r_{Qk}[2], r_{Qk}[3], c_{Qk}\}$. That is, the fourth sub-window is now the oldest and is not read by the query.

7.3 Conflict Serializability of Sliding Window Queries

7.3.1 Serializability and Serialization Orders

The isolation level requirements of queries over periodically-sliding windows are analyzed next. For now, assume that queries access a single window and correspond to plans that produce weakest non-monotonic update patterns; queries accessing synopses over joins of multiple windows (weak non-monotonic update patterns) will be discussed in Section 7.4.3. First, consider the possible types of conflicts arising from concurrent execution of transactions. A conflict occurs when two interleaved transactions operate on the same sub-window and at least one of the operations is a write. Clearly, a read-write conflict occurs whenever T_j interrupts T_{Qk}, as in Figure 7.1(c) and (d). This is because each T_{Qk} reads every sub-window, including the sub-window overwritten by T_j. Since window movements were assumed to be executed and committed immediately, write-write conflicts do not occur.

The traditional method for dealing with conflicts requires an execution history H to be serializable. However, conflict serializability is insufficient in the context of sliding windows, as demonstrated in the following example. Assume a sliding window partitioned into five sub-windows, numbered zero through four, with sub-window zero being the oldest at the current time. Consider the following four histories: H_a, H_b, H_c, and H_d (the initial transactions T_0 through T_4 that fill up the window are omitted for brevity).

$H_a = r_{Q1}[0]\ w_5[0]\ c_5\ w_6[1]\ c_6\ r_{Q1}[1]\ r_{Q1}[2]\ r_{Q1}[3]\ r_{Q1}[4]\ c_{Q1}$
$H_b = r_{Q1}[0]\ w_5[0]\ c_5\ r_{Q1}[1]\ w_6[1]\ c_6\ r_{Q1}[2]\ r_{Q1}[3]\ r_{Q1}[4]\ c_{Q1}$
$H_c = w_5[0]\ c_5\ r_{Q1}[0]\ r_{Q1}[1]\ r_{Q1}[2]\ r_{Q1}[3]\ w_6[1]\ c_6\ r_{Q1}[4]\ c_{Q1}$
$H_d = w_5[0]\ c_5\ r_{Q1}[0]\ w_6[1]\ c_6\ r_{Q1}[1]\ r_{Q1}[2]\ r_{Q1}[3]\ r_{Q1}[4]\ c_{Q1}$

Each history represents interleaved execution of a read-only transaction T_{Q1} and two window movements, T_5 and T_6. The difference among the histories is that the window movements take place at different times. The associated serialization graphs are drawn in Figure 7.4. The di-

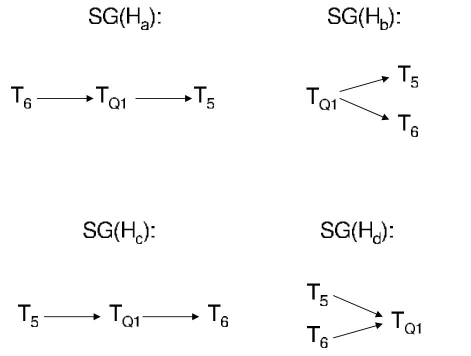


Figure 7.4: Serialization graphs for H_a, H_b, H_c, and H_d

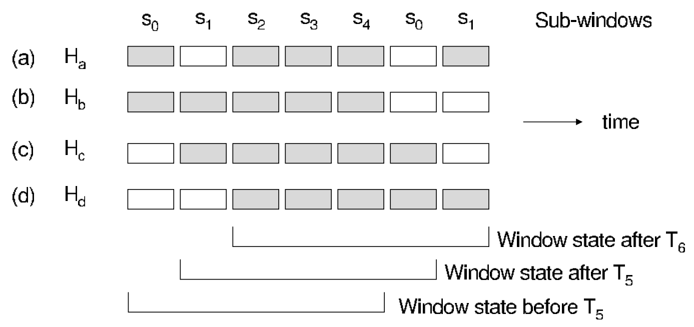


Figure 7.5: Differences in the results returned by T_{Q1} in H_a, H_b, H_c, and H_d

rection of the edges corresponds to the order in which conflicting operations are serialized. In particular, there are two pairs of conflicting operations in each schedule: $r_{Q1}[0]$ and $w_5[0]$, and $r_{Q1}[1]$ and $w_6[1]$. Note that all four graphs are acyclic, therefore all four histories are serializable, but their serialization orders are different.

Consider the state of the sliding window read by T_{Q1} in each of the four histories. This is illustrated in Figure 7.5, where the first sub-windows on the left (s_0 through s_4) correspond to the initial state of the window after T_0 through T_4 were executed. Next, T_5 advances the window forward by one sub-window, which may be thought of as overwriting the old copy of sub-window s_0 (on the far left) with a new copy, appended after s_4. Thus, the state of the window after T_5 commits is represented by the contiguous sequence of sub-windows $\{s_1, s_2, s_3, s_4, s_0\}$. Then, T_6 advances the window again by appending a new copy of s_1 on the far right and implicitly deleting the old copy of s_1 on the left. Hence, the state of the window after T_6 commits is equivalent to the contiguous sequence of sub-windows $\{s_2, s_3, s_4, s_0, s_1\}$. Shaded sub-windows represent those which were read by T_{Q1} in each of the four histories, as explained next.

First, consider $SG(H_a)$ and note that H_a serializes T_6 before T_{Q1}, meaning that the window movement caused by T_6 (creation of a new version of sub-window s_1) is reflected in the query. However, H_a serializes an earlier window update T_5 after T_{Q1}, therefore the prior window movement caused by T_5 (creation of a new version of s_0) is hidden from the query. Hence, H_a causes T_{Q1} to read an old copy of s_0 and a new copy of s_1, as illustrated in Figure 7.5(a), which does not correspond to a window state at any point in time. This is because the shaded rectangles do not form a contiguous sequence of five sub-windows. Next, recall that H_b serializes both window movements after T_{Q1}, therefore the query reads old versions of s_0 and s_1, as illustrated in Figure 7.5(b). This corresponds to the state of the window after T_4 commits. By similar reasoning, H_c allows T_{Q1} to read the state of the window after T_5 commits (Figure 7.5(c)), and only H_d ensures that T_{Q1} reads the most up-to-date state of the window that reflects both T_5 and T_6 (Figure 7.5(d)). Again, this is because only $SG(H_d)$ serializes both window movements before

T_{Qk}, meaning that T_{Qk} sees both updates.

7.3.2 Isolation Levels for Sliding Window Queries

Having shown that the serialization order affects the semantics of read-only transactions, this section proposes two stronger isolation levels that restrict the allowed serialization orders.

Definition 7.1 *A history H is said to be* window-serializable (WS) *if all of its committed T_{Qk} transactions read a true state of the sliding window(s) as of some point in time on or before their respective commit times (i.e., a contiguous sequence of sub-windows is read by each query, as in Figure 7.5 (b), (c), and (d)).*

Definition 7.2 *A window-serializable history H is said to be* latest-window-serializable (LWS) *if all of its committed T_{Qk} transactions read the state of the window(s) as of their respective commit times.*

Note that only *LWS* guarantees that queries read the most up-to-date state of the window, which is the central requirement of DSMS concurrency control, as explained in Section 7.1. Furthermore, note that *WS* corresponds to DBMS snapshot isolation [255], whereas the related notion of strong snapshot isolation (read-only transactions see snapshots of the data as of their start times [71]) is stronger than *WS* but weaker than *LWS*. Motivated by Figure 7.4, the following theorems hold with respect to the transaction workload assumed in this chapter.

Theorem 7.1 *A history H is window-serializable iff $SG(H)$ has the following property: for any T_{Qk}, if any T_i is serialized before T_{Qk}, then for all T_j serialized after T_{Qk}, $i < j$.*

Proof. Suppose that H is *WS*. If all transactions T_{Qk} contained in H incur at most one concurrent window movement, then clearly, $SG(H)$ satisfies the desired property. Otherwise, note that for T_{Qk} to read a sliding window state from some point in the past or present, it must be the case that either T_{Qk} is isolated from all the concurrent window updates, or it only reads the least recent update, or it only reads the two oldest updates, and so on. In all cases, $SG(H)$ contains less recent updates serialized before the query and more recent updates serialized after the query, as wanted. Now suppose that $SG(H)$ satisfies the property that all T_j serialized after any T_{Qk} have higher subscripts than those serialized before T_{Qk}. Let m be the maximum subscript of any transaction T_i serialized before T_{Qk}. It follows that T_{Qk} reads a sliding window state that resulted from applying all the updates up to T_m and therefore H is *WS*.
□

Theorem 7.2 *A history H is latest-window-serializable iff $SG(H)$ has the following property: for any T_{Qk}, all concurrent T_i transactions must be serialized before T_{Qk}.*

Proof. Suppose that H is *LWS* and let T_{Qk} be any query that incurs at least one concurrent window movement. It follows that T_{Qk} reads a state of the window that results from applying all the concurrent updates. Hence, concurrent window updates must be serialized before queries, as wanted. Now suppose that $SG(H)$ does not contain any links pointing from any T_{Qk} to any T_i. This means that there are no queries that have been interrupted by window updates which the queries then did not see. Hence, H is *LWS*.
□

7.4 Transaction Scheduler Design

7.4.1 Producing *LWS* Histories

This section presents the design of a DSMS transaction scheduler that produces *LWS* histories. Recall from Section 7.2 that write-only transactions T_j must be executed with highest priority so that queries have access to an up-to-date version of the window. Given this requirement, the proposed scheduler executes window movements immediately and aborts any read-only transactions that participate in read-write conflicts.

The scheduler is summarized as Algorithm 7. For each query T_{Qk}, let $n_k t$ be the length of the window that it references (i.e., T_{Qk} reads the n_k youngest sub-windows). Note that $1 \leq n_k \leq n$ for all k. Lines 2 and 3 serially execute window movements immediately (technically, line 3 must wait for an acknowledgement that the write operation has been performed). Lines 11 through 13 initialize a bit array B_{Qk} for each newly arrived T_{Qk}, where bit i is set if T_{Qk} has already read sub-window i. Lines 15 through 23 execute read-only transactions, one sub-window scan at a time, and set the corresponding bit in B_{Qk} to true[3]. Again, before committing T_{Ql} in line 20, the algorithm must wait for an acknowledgement of performing the read operation from line 17. Note that Algorithm 7 allows multiple read-only transactions to be executed at the same time in any order (line 16) because they do not conflict with one another. Lines 4 through 8 resolve *LWS* conflicts, as proven below.

Theorem 7.3 *Algorithm 7 produces* LWS *histories.*

Proof. As per Definition 7.2, all committed read-only transactions T_{Qk} must have the property that any window movements T_j that were executed at the same time as T_{Qk} are serialized before T_{Qk}. First, note that the only time that a new *LWS* violation may possibly appear is after a window update T_j commits (which is done immediately after performing the write operation) while one or more T_{Qk} transactions are still running. Furthermore, a *LWS* conflict appears only

[3]Recall that a query over a window shorter than nt, say $n_k t$, reads the n_k youngest sub-windows. Therefore, a concurrent window movement changes the set of sub-windows that must be read by queries over windows shorter than nt (recall the discussion in Section 7.2.3). For now, assume that Algorithm 7 explicitly changes the contents of affected query transactions immediately after each window movement so that line 17 has access to the correct set of sub-window reads that must be performed by each query at any given time. A simpler solution will be presented shortly.

Algorithm 7 DSMS Transaction Scheduler

Input: list of currently running T_{Qk} transactions L
Local variables: bit array B

```
loop
    if new transaction $T_j$ arrives for scheduling then
        execute $w_j[j \bmod n]$, $c_j$
        for each $T_{Qk}$ in $L$
            if $B_{Qk}[(j + n - n_k) \bmod n]$ = true then
                execute $a_{Qk}$ (abort $T_{Qk}$)
            end if
        end for
    elseif new transaction $T_{Qk}$ arrives for scheduling then
        add $T_{Qk}$ to $L$
        for $i = 0$ to $n - 1$
            set $B_{Qk}[i]$ = false
        end for
    end if
    if $L$ is not empty then
        choose any $T_{Ql}$ from $L$
        execute next operation of $T_{Ql}$, call it $r_{Ql}[m]$
        set $B_{Ql}[m]$ = true
        if no more read operations left in $T_{Ql}$ then
            execute $c_{Ql}$
            remove $T_{Ql}$ and $B_{Ql}$ from $L$
        end if
    end if
end loop
```

if any T_j has updated a sub-window (an older copy of) which has already been read by any of the currently running T_{Qk} transactions, in which case T_j would be serialized before T_{Qk}. This occurs if $B_{Qk}[(j - n + n_k) \bmod n]$ is set for any currently running T_{Qk}. This is easy to see if $n = n_k$; if the $(j \bmod n)$th bit is set to true, then the query has already read the $(j \bmod n)$th sub-window and must be aborted. If $n_k \neq n$, then recall Figure 7.2 (b) and (d) and note the following observation: if a query references a window of length $n_k t$, then an update of the jth sub-window implies that the $(j + n - n_k \bmod n)$th sub-window now contains expired tuples with respect to the window of length $n_k t$. Hence, even though the query has not explicitly read the updated sub-window, it must be aborted because it has seen data that have expired from the shorter window of length $n_k t$. In either case, Algorithm 7 aborts T_{Qk} (line 6), ensuring that all T_{Qk} transactions committed in line 20 satisfy Definition 7.2.
□

Algorithm 7 supports read-only transactions with different priorities, such as one-time queries or "important" persistent queries (as in Q_1 from Figure 7.1(d)). To do this, assume that the query manager embeds a priority p within each T_{Qk} and change line 16 in Algorithm 7 to read: "let T_{Ql} be the transaction in L with the highest value of p". Consequently, if a low-priority T_{Qk} is currently being executed, then a higher-priority T_{Qm} transaction has the effect of suspending T_{Qk}. This extension does not impact the correctness of Algorithm 7 as it does not introduce any new *LWS* conflicts.

7.4.2 Optimal Ordering of Read Operations

Given that Algorithm 7 may abort read-only transactions in order to guarantee *LWS*, it is desirable to minimize the required number of aborts. The idea is to shuffle the read operations within T_{Qk} transactions[4] given the following insight. Since aborts occur when a sub-window is updated but an older version of it has already been read by a concurrent T_{Qk} transaction, T_{Qk} should be executed by first reading the sub-window which is scheduled to be updated the farthest out into the future. More precisely, define the *time-to-update (TTU)* of a sub-window as the number of window-movement transactions T_j that must be applied until this sub-window is updated. When the scheduler chooses a read-only transaction T_{Qk} to process, it always executes the remaining read operation of T_{Qk} whose sub-window has the highest TTU value at the given time. Note that at any time, the sub-window with the highest TTU value is the one that has been updated by the most recent window movement transaction.

The revised scheduler is shown below as Algorithm 8 (again, adding support for multiple priority levels can be done by changing line 20 to process the highest-priority transaction). There are two main changes. First, lines 4 through 6 update the TTU values of each sub-window after every window movement. The newly updated sub-window receives a value of n (it will take n write-only transaction until this sub-window is updated again), whereas the TTU values of the remaining sub-windows are decremented. Furthermore, line 21 selects m to be the index of the sub-window which has the highest TTU value and has not been read by T_{Ql}.

Recall from Section 7.4.1 that Algorithm 7 was assumed to change the contents of read-only transactions corresponding to queries over windows shorter than nt whenever a concurrent window update took place. This technique applies to Algorithm 8, but a simpler solution is now possible. Given that line 22 reads sub-windows in TTU order, a query over a window of length $n_k t$ can simply be defined as reading the n_k sub-windows with highest TTU values (i.e., the $n_k t$ youngest sub-windows). Even if a concurrent window movement takes place, the TTU values of all the sub-windows are updated in lines 4 through 6, therefore an explicit modification of the contents of query transactions is not necessary. Instead, lines 21 and 22 may be replaced by instructions that maintain a counter of the number of sub-windows that must be read by each query T_{Qk} (namely n_k), and schedule a read of the youngest window if the counter of the given query is non-zero (if the counter drops to zero, then the condition in line 24 is true and the query transaction may be committed). Again, if a sub-window is read and is later modified, then line 9

[4]It is assumed that the order in which the data are read does not affect the final answer of the query, as is the case in traditional relational query optimization, where several different access paths into the data are often possible.

correctly identifies transactions that need to be aborted, even if they correspond to queries that scan windows shorter than nt (recall Theorem 7.3).

The idea in Algorithm 8 is similar to the Longest Forward Distance (LFD) cache replacement algorithm [33], which always evicts the page whose next access is latest. LFD is optimal in the off-line case in terms of the number of page faults, given that the system knows the entire page request sequence and that all page faults have the same cost.

Algorithm 8 DSMS Transaction Scheduler with TTU

```
Input: list of currently running $T_{Qk}$ transactions $L$
Local variables: bit array $B$, array of sub-window $TTU$ values $TTU[n]$
loop
    if new transaction $T_j$ arrives for scheduling then
        execute $w_j[j \bmod n]$, $c_j$
        for $i = 0$ to $n - 1$
            set $TTU[i] = TTU[i] - 1$
        end for
        set $TTU[j \bmod n] = n$
        for each $T_{Qk}$ in $L$
            if $B_{Qk}[(j + n - n_k) \bmod n] =$ true then
                execute $a_{Qk}$ (abort $T_{Qk}$)
            end if
        end for
    elseif new transaction $T_{Qk}$ arrives for scheduling then
        add $T_{Qk}$ to $L$
        for $i = 0$ to $n - 1$
            set $B_{Qk}[i] =$ false
        end for
    end if
    if $L$ is not empty then
        choose any $T_{Ql}$ from $L$
        let $m = \text{argmax}_{B_{Ql}[i]=false} TTU[i]$
        execute $r_{Ql}[m]$
        set $B_{Ql}[m] =$ true
        if no more read operations left in $T_{Ql}$ then
            execute $c_{Ql}$
            remove $T_{Ql}$ and $B_{Ql}$ from $L$
        end if
    end if
end loop
```

Theorem 7.4 *Algorithm 8 is optimal for ensuring* LWS *in the sense that it performs the fewest possible aborts for any history H.*

Proof. Let A be the scheduler in Algorithm 8 and let S be any other transaction scheduler that serializes transactions in the same way as A, but only differs in the ordering of read operations inside one or more read-only transactions. That is, S corresponds to Algorithm 7 with some arbitrary implementation of the meaning of "next operation" in line 17. The objective is to prove that S performs no fewer aborts than A for any history H. Let H_i be the prefix of H containing the first i read operations (interleaved with zero or more write operations, and zero or more commit or abort operations). The proof proceeds by inductively transforming the sequence of read operations produced by S into that produced by A, one read operation at a time. To accomplish this, let $S_0 = S$ and define a transaction scheduler S_{i+1} that, given S_i, has the following two properties.

1. Both S_i and S_{i+1} order all the read operations in H_i in the same way as A.
2. S_{i+1} orders all the read operations in H_{i+1} in the same way as A and performs no more aborts than S_i in H_{i+1}.

Let $r_k[y]$ be the $(i+1)$st read operation executed by S_i and $r_k[z]$ be the $(i+1)$st read operation executed by S_{i+1}. Due to the assumption that A and S only differ in the ordering of read operations inside read-only transactions, the $(i+1)$st read operations done by S_i and S_{i+1} belong to the same transaction, call it T_{Qk}. Thus, sub-window $z \pmod n$ has the highest TTU value at this time. Now, if $z = y$ then $S_{i+1} = S_i$, therefore property 2 holds and the proof is completed. Otherwise, S_{i+1} and S_i differ in the $(i+1)$st read operation. First, suppose that T_{Qk} is not interrupted by any write-only transactions before the next read operation. Then, T_{Qk} is not aborted by S_i or by S_{i+1} in H_{i+1} and the proof is completed (property 2 holds). Next, suppose that T_{Qk} is interrupted by at least one write-only transaction before the next read operation. The remainder of the proof is broken into three cases, which collectively prove property 2.

In the first case, suppose that the set of interrupting transactions contains T_y, but not T_z. Given that sub-window $z \pmod n$ has the highest TTU value at this time, and that write-only transactions are generated and serially executed in increasing order of their subscripts, the most recent write-only transaction can have a subscript no higher than $z - 1$. Then, S_i aborts T_{Qk} in H_{i+1}. This is because T_{Qk} has already read an old version of sub-window $y \pmod n$ and therefore T_y would have been serialized after T_{Qk}. However, S_{i+1} does not abort T_{Qk} in H_{i+1}. To see this, observe that T_{Qk} could not have possibly read any of the sub-windows that have just been updated. This is due to the fact that those sub-windows must have lower TTU values than sub-window $z \pmod n$ and must necessarily be scheduled after sub-window $z \pmod n$ by S_{i+1}.

In the second case, suppose that the set of interrupting transactions does not contain T_y or T_z. By the same reasoning, the most recent write-only transaction can have a subscript no higher than $y - 1$. S_{i+1} does not abort T_{Qk} in H_{i+1} because T_{Qk} could not have possibly read any of the sub-windows updated by or before T_{y-1} (they all have lower TTU values than sub-window z $\pmod n$). In terms of satisfying property 2, it does not matter what S_i does in this case.

Finally, in the third case, suppose that the set of interrupting transactions contains both T_y and T_z. Then, both S_i and S_{i+1} abort T_{Qk} in H_{i+1} because both schedulers allow T_{Qk} to read a sub-window that has now been updated.

□

Algorithm 8 takes advantage of the weakest non-monotonic update patterns of sliding windows in order to anticipate possible LWS violations and prevent them to the greatest possible extent. For an example of the need for reordering the read operations within read-only transactions, consider the following interleaved schedule of a query T_{Q1} and two window movements, T_5 and T_6, given the same parameters as in histories H_a through H_d defined in Section 7.3.

$H_e = r_{Q1}[4]\ w_5[0]\ c_5\ r_{Q1}[0]\ r_{Q1}[3]\ r_{Q1}[2]\ w_6[1]\ c_6\ r_{Q1}[1]\ c_{Q1}$

At the beginning, sub-window zero is the oldest and has a TTU of one (it will be updated after the next write transaction, namely T_5). Sub-window one is next with a TTU of two, and so on until sub-window four, which has a TTU of five. Thus, the first read operation to be scheduled reads the sub-window with the highest TTU value, namely sub-window four. Next, sub-window zero is updated and therefore its TTU changes to five; the TTU values of all the other sub-windows are decremented. Thus, sub-window zero is the next one to be read by T_{Q1} because its TTU value of five is the highest. Sub-windows three and two are next (with TTU values of three and two, respectively). At this point, only sub-window one remains to be read, which is done after it is updated by T_6. In contrast, had the above schedule sequentially executed the read operations within T_{Q1} (as shown in schedule H_f below), latest-window serializability would have been violated immediately after T_5 has committed.

$H_f = r_{Q1}[0]\ w_5[0]\ c_5\ r_{Q1}[1]\ r_{Q1}[2]\ r_{Q1}[3]\ w_6[1]\ c_6\ r_{Q1}[4]\ c_{Q1}$

Thus far, it was assumed that newly arrived data overwrite the oldest sub-window, meaning that a query which has read an old copy of a freshly updated sub-window must be aborted in order to guarantee *LWS*. Recall the discussion in Section 7.2.2, mentioning the possibility of temporarily keeping expired sub-windows and allowing queries to re-read them in order to "undo" the contribution of expired tuples from the query state. Algorithm 8 still applies in this case. The difference is that when a *LWS* conflict is discovered, the transaction re-scans the old copy of the newly updated sub-window, followed by reading the new copy. Therefore, no aborts are necessary. In this context, Algorithm 8 is optimal in the sense of minimizing the number of required sub-window re-scans.

7.4.3 Note on Queries Accessing a Materialized Sub-Result

Up to now, it was assumed that the window or synopsis scanned by a query takes its input from a weakest non-monotonic query plan. If a query reads a weak non-monotonic result, then it is no longer the case that the oldest sub-window may be dropped and the newest sub-window inserted in its place. The solution in Chapter 3 employed a calendar queue to store state corresponding to the intermediate result of a weak non-monotonic query plan. Suppose that the calendar queue is partitioned on expiration time and that each window to be joined contains five sub-windows. As illustrated in Figure 7.6, when the windows slide, the oldest sub-window expires, as before. However, each sub-window may incur insertions because the new tuples may have various expiration

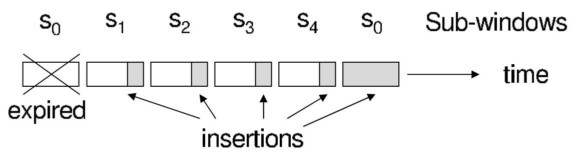


Figure 7.6: Update of a materialized sub-result having weak non-monotonic update patterns

times[5]. To deal with this issue, the following modification is made to the scheduling algorithms. When a materialized sub-result having weak non-monotonic update patterns is updated while a query is scanning it (or a summary over it), the tuples inserted into sub-windows which have already been read are copied and passed to the query prior to being inserted into their sub-windows (and synopses). This way, the query is guaranteed to see all the updates and *LWS* is preserved, but the query and update logic becomes more complex. Note that the query need not read new tuples that were inserted into sub-windows which it has not read yet (they will be read when the query finally scans these sub-windows).

7.5 Experiments

7.5.1 Implementation Details and Experimental Procedure

A simple DSMS query manager was implemented in Java along with the following four transaction schedulers: Algorithm 8 (abbreviated *TTU*), Algorithm 7 (which does not re-order the read operations within transactions, abbreviated *LWS*), a scheduler similar to Algorithm 8 that only enforces window-serializability (abbreviated *WS*), and a scheduler that executes transactions serially (as in current DSMSs, abbreviated *Serial*). Experiments were performed on a Pentium-IV PC with a 3 GHz CPU and 1 Gb of RAM, running Linux. The input stream is a sequence of simulated IP header packets with randomly generated attribute values. For example, the source and destination IP addresses have one of one thousand random values, whereas the data size is a random integer between one and 100. The steady-state stream arrival rate is one packet per millisecond, but the specific arrival rate over a particular sub-window is allowed to deviate from the steady-state rate by a factor of up to ten.

The query workload simulates network traffic analysis [62, 66]. There are two levels of transaction priorities: those corresponding to re-executions of periodic queries with low-priority, and those corresponding to one-time queries with high-priority. A total of between 40 and 100 periodic queries are executed, arranged into groups of five for shared processing (e.g., queries computing the same aggregate over different window lengths require only one scan of a window or a basic interval synopsis). Periodic queries are chosen randomly from the following set: `Top k` queries over the source or destination IP addresses, and percentile queries (25th, 50th, 75th, 90th, 95th,

[5]A synopsis over the result of a weak non-monotonic query plan may be maintained in a similar fashion, as will be discussed in Section 8.3.2.

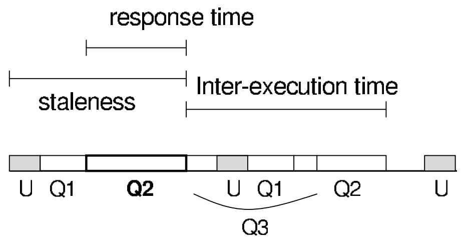


Figure 7.7: Staleness, response time, and inter-execution time of Q_2

and 99th percentiles) over the total bandwidth consumed by (or directed to) distinct IP addresses. The window sizes referenced by queries are generated randomly between one and n, where n is the total number of sub-windows. For simplicity of implementation, periodic queries are executed by scanning the window and building a hash table on the required attribute. One-time queries are chosen from a set of simple aggregates over a random subset of the source and destination IP addresses. Each query references the same time-based window, which is stored in main memory.

After initializing the sliding window using a randomly generated input stream, the four transaction schedulers are tested over an identical query workload. The tests proceed for a time equal to the window length and are repeated five times using different input streams. Results are averaged over the five trials. The parameters being varied in (and across) the experiments are the query workload, the window size (controlled via the number of sub-windows), and the length of each sub-window (which controls the frequency of window movements). The following performance metrics are used to evaluate the four transaction schedulers, as illustrated in Figure 7.7 corresponding to the execution time line from Figure 7.1(d).

- *Query staleness* is the difference between the time that a query reports an answer and the time of the last window update reflected in the answer.
- *Response time* is the difference between the query execution start time and end time. This metric is important for one-time queries, which are usually time-sensitive as they may be posed in order to investigate a suspicious change in the result of a persistent query.
- *Inter-execution time* of a long-running query is the length of the interval between its re-executions[6]. A DSMS is expected to tolerate slightly longer inter-execution times if the returned answers are more up-to-date. The motivation for this is that even an older answer is returned earlier, the system would have to re-execute the query soon in order to produce an answer that reflects the new state of the window.

7.5.2 Percentage of Aborted Transactions

The first experiment illustrates Theorem 7.4 by comparing the percentage of aborted read-only transactions using Algorithms 7 and 8. Two sub-window sizes are tested: $t = 1$ sec. and $t = 5$

[6]As will be explained in more detail in Chapter 8, the inter-execution interval is typically supplied by the query. However, for simplicity, the experiments in this chapter assume that each query has the same desired inter-execution interval and all queries are continually re-executed in a round-robin fashion.

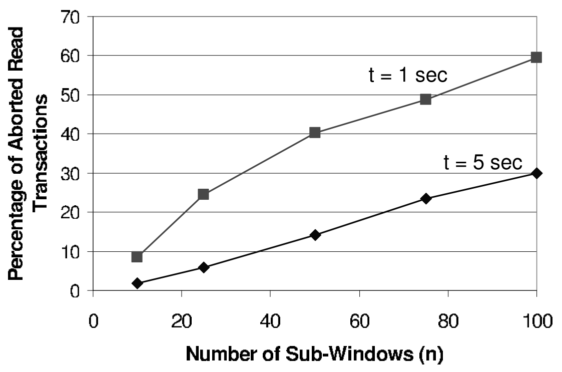


Figure 7.8: Percentage of read-only transactions aborted by Algorithm 7

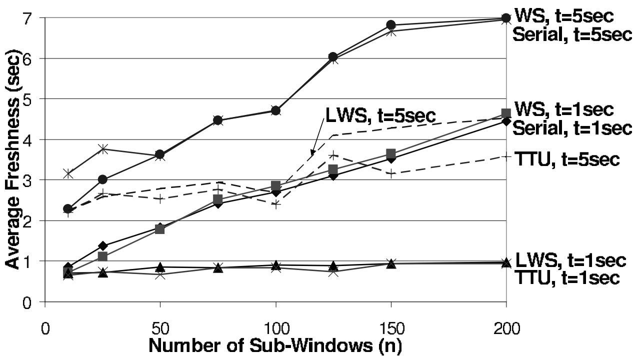


Figure 7.9: Comparison of query staleness for Serial, WS, LWS, and TTU

sec., with the number of sub-windows varied from ten to 100. The number of periodic queries is set to 40 for $t = 1$ sec. and 100 for $t = 5$ sec. The percentage of transactions aborted by Algorithm 7 is shown in Figure 7.8. In contrast, Algorithm 8 did not abort any transactions in this experiment. This is because during normal execution, a periodic query does not incur more than one concurrent window update, unless suspended for a long time in order to run a heavy workload of one-time queries. Since Algorithm 8 ensures that read-only transactions postpone reading the sub-window that is about to be updated until the end, aborts can be easily avoided if the number of concurrent window updates is small.

Note that the proportion of read-only transactions aborted by Algorithm 7 is higher for $t = 1$ sec. because a smaller sub-window size implies that window movements are executed more often, thereby increasing the chances of read-write conflicts. Aborts are also more frequent as the number of sub-windows increases (i.e., as the length of the sliding window grows) because this causes longer query evaluation times, therefore the number of read-only transactions between window updates decreases. In turn, this increases the proportion of read-only transactions that execute at the same time as window movements and raises the chances of read-write conflicts. In the worst case of frequent window movement and long window size ($t = 1$ sec. and 100 sub-windows), Algorithm 7 aborts over half of the read-only transactions because nearly every transaction incurs a concurrent window movement, which often ends up causing a *LWS* conflict. Similar results were obtained when one-time queries were added to the workload and issued at random times. In particular, Algorithm 8 still did not abort any transactions.

7.5.3 Experiments with a Workload of Periodic Queries

This section presents the results of executing *Serial*, *WS*, *LWS*, and *TTU* on a workload consisting of periodic queries and interleaved window movements. The sub-window sizes, number of sub-windows, and number of periodic queries are the same as in the previous experiment; for now, assume that no one-time queries are posed. The variables being measured are average staleness,

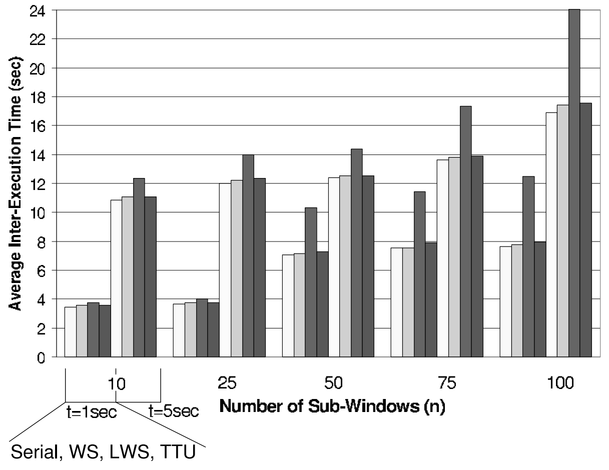


Figure 7.10: Comparison of query inter-execution times for Serial, WS, LWS, and TTU

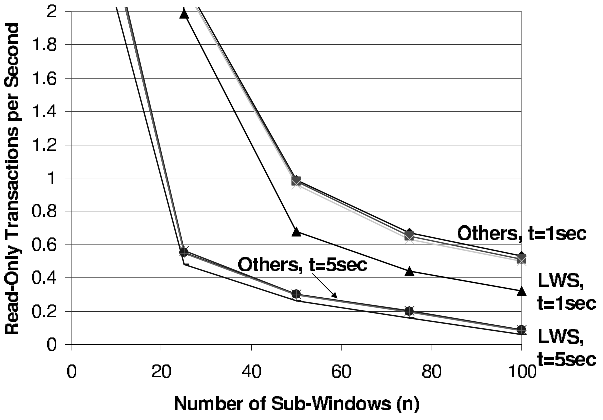


Figure 7.11: Comparison of throughput of read-only transactions for LWS and "Others"

inter-execution time, and throughput.

The average query staleness, in units of seconds, is shown in Figure 7.9 (the lower the value, the better). *TTU* and *LWS* clearly outperform *WS* and *Serial* because the first two guarantee latest-window-serializable schedules, where queries have access to an up-to-date state of the window. As expected, staleness increases for all four schedulers as the sub-window size grows to $t = 5$ sec. and window movements become less frequent. Moreover, increasing the number of sub-windows (or equivalently, increasing the window length) generally has an adverse effect on staleness because the query execution times increase. Note that *Serial* performs slightly better than *WS* because *WS* adds to the query execution time by performing concurrent window movements, yet the answer does not reflect any of the updates. Overall, *TTU* provides the lowest query staleness in all tested scenarios.

The average query inter-execution times, in units of seconds, are graphed in Figure 7.10. Each cluster of eight bars corresponds, in order, to *Serial*, *WS*, *LWS*, and *TTU* for $t = 1$ sec., followed by *Serial*, *WS*, *LWS*, and *TTU* for $t = 5$ sec. Serial has the best (lowest) inter-execution times because it does not incur the overhead of serialization graph testing, therefore its total query execution time is slightly lower. Notably, *LWS* (corresponding to the third and seventh bars in each cluster) performs the worst. For instance, aborting every second re-execution of a periodic query means that its inter-execution time doubles. In general, increasing the sub-window size to $t = 5$ sec. (and hence, increasing the total window size) leads to longer inter-execution times for all four schedulers as queries take longer to process. Similarly, increasing the number of sub-windows increases the query evaluation times and therefore negatively affects the inter-execution times. Overall, *Serial* yields the best query inter-execution times, with *WS* and *TTU* following

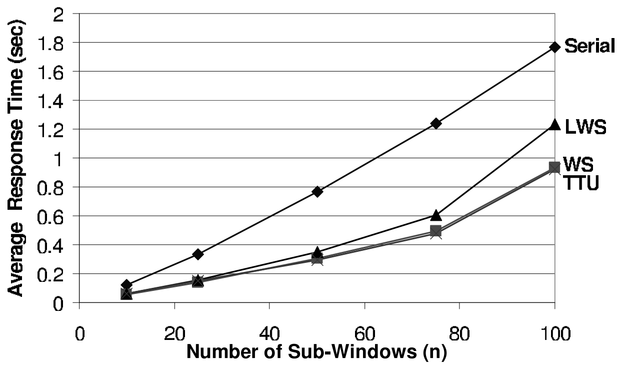


Figure 7.12: Comparison of one-time query response times for Serial, WS, LWS, and TTU

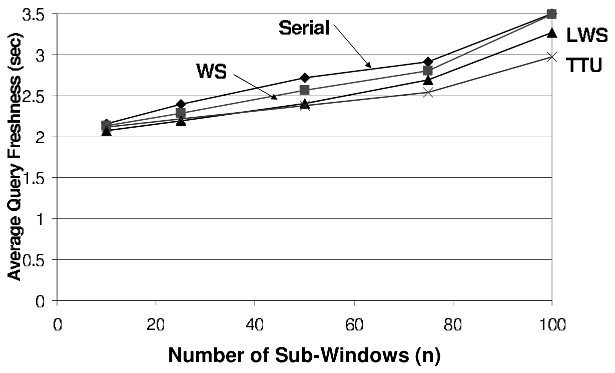


Figure 7.13: Comparison of one-time query staleness for Serial, WS, LWS, and TTU

very closely behind, whereas *LWS* performs badly due to aborted transactions.

Figure 7.11 illustrates the throughput (in units of the number of read-only transactions per second) of *LWS* versus the other three schedulers (namely *Serial*, *WS*, and *TTU*, labeled Others), which all yield very similar results. In particular, the throughput penalty of *TTU* versus *Serial* is very small—typically below two percent and at most four percent. This is because the serialization graph testing done by *TTU* consists of simple bit operations after each window movement and causes negligible overhead. Note the poor performance of *LWS*; its throughput is lower in all cases due to aborted transactions. As expected, the throughput of all schedulers decreases when transactions take longer to execute, which occurs when the sliding window is large (i.e., the number of sub-windows increases or the sub-window size increases).

7.5.4 Experiments with a Mixed Workload of Periodic and One-Time Queries

This section reports the results of experiments with a mixed workload of periodic and one-time queries (and concurrent window movements). The sub-window size is fixed at five seconds, the number of periodic queries at 100, and the number of one-time queries per sub-window length at five. One-time queries are scheduled at random times with an average time between requests set to one second.

The average one-time query response times, in units of seconds, are shown in Figure 7.12. *TTU* and *WS* perform best and yield nearly identical response times. The response times of *LWS* are noticeably longer because it is forced to abort and restart some one-time queries. *Serial* exhibits the worst results in this experiment because it is unable to suspend a periodic query and execute a one-time query immediately; in general, *Serial* is inappropriate for any situation involving prioritized scheduling. As the number of sub-windows increases, the response time achieved by each of the four schedulers worsens because it is now more costly to execute each query.

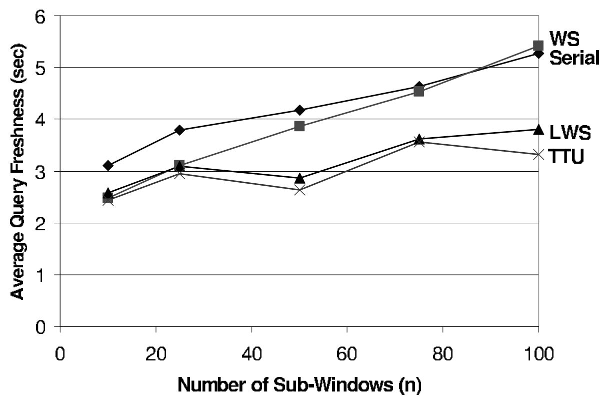


Figure 7.14: Comparison of periodic query staleness for Serial, WS, LWS, and TTU

Figure 7.13 plots the average one-time query staleness, in units of seconds. As expected, *TTU* outperforms the other schedulers because it guarantees latest-window-serializability and does not abort any transactions. The performance of *LWS* is somewhat worse because some of the transactions corresponding to one-time queries are aborted and restarted at a later time. *WS* and *Serial* do not guarantee latest-window serializability and therefore exhibit the worst performance. Overall, *TTU* yields the best results in terms of one-time query staleness and is tied for best in terms of the response time.

Finally, the average staleness of periodic queries is examined separately in order to verify that the performance edge of *TTU* in the context of one-time query staleness does not come at a cost of poor periodic query staleness. Results are shown in Figure 7.14. It can be seen that *TTU* maintains its superiority in producing the most up-to-date results of periodic queries.

7.5.5 Lessons Learned

The above experiments have shown the advantages of executing latest-window-serializable histories, particularly when enforced using the proposed update-pattern-aware transaction scheduler. Staleness and response time were shown to improve at the expense of minimal loss in throughput.

7.6 Comparison with Related Work

The concurrency control mechanisms presented in this chapter are compatible with any DSMS that employs periodic updates of sliding windows and query results, e.g., [2, 12, 48, 52, 109, 222, 276]. The techniques are also applicable to a system such as PSoup [49], where mobile users connect to a DSMS intermittently and retrieve the latest results of sliding window queries. In this context, asynchronous requests may be modeled as one-time queries issued at various times. Given that mobile users may have low connectivity with the system (e.g., via a wireless channel), it is particularly important to guarantee low response times and up-to-date query answers. The transaction scheduler proposed in this chapter fulfills both of these requirements.

The transaction model used in this chapter resembles multi-level concurrency control and multi-granularity locking as it considers a sub-window, rather than an entire window, to be an atomic data object. The novelty is that the order in which sub-window read operations are performed is maintained in such a way as to minimize the number of aborted transactions. Furthermore, note that the order of read operations changes whenever the window slides and the sub-window TTU values change. In contrast, fixed resource ordering has been employed to prevent deadlocks [96]. However, the system model studied in this chapter does not encounter deadlocks due to the relative simplicity of the transactional workload. Therefore, Algorithm 8 is able to change the TTU ordering whenever the window slides (thereby minimizing the number of aborted read-only transactions) without causing deadlocks.

The TTU scheduler employed conflict-based concurrency control. Other scheduling techniques include two-phase locking and timestamping [35]. However, two-phase locking may not be appropriate in this context because it is not clear how to force a particular serialization order using locks. Moreover, the possible problem with using timestamping for DSMS concurrency control is the difficulty of ensuring latest-window serializability. Suppose that each transaction receives a timestamp when it is passed to the transaction scheduler and that serialization order is determined by timestamps. In this case, any concurrent window update transaction is assigned a higher timestamp than a read-only transaction and is therefore serialized before the read-only transaction. Hence, Algorithm 8 would be forced to abort every read-only transaction that is interrupted by a window movement.

Latest-window serializability is similar to commit-order preserving serializability (*CPS*) [255], in which conflicting operations must be ordered in the same way as commits. The venn diagram in Figure 7.15 illustrates the relationship among the isolation levels in the general case. Some histories, such as, H_d from Section 7.3.1, are both *CPS* and *WS* (or *LWS*). In contrast, the following history, call it H_g, is *CPS*, but neither *WS* nor *LWS*.

$H_g = r_{Q1}[0]\ w_5[0]\ w_6[1]\ c_6\ r_{Q1}[1]\ r_{Q1}[2]\ r_{Q1}[3]\ r_{Q1}[4]\ c_{Q1}\ c_5$

To see this, observe that H_g is similar to H_a from Section 7.3.1, except that the commitment of T_5 has been delayed until after T_{Q1} commits in order to correspond to the serialization order. Clearly, it is still not *WS*, yet it is *CPS* because the commit order matches the serialization order. On the other hand, the following history, call it H_h, is *WS* and *LWS*, but not *CPS*. This is because the query transaction T_{Q1} can in fact see the concurrent update made by T_9 and therefore reads the latest state of the sliding window. However, the update transaction happens to commit after the query commits, yet it is serialized before the query.

$H_h = r_{Q1}[0]\ r_{Q1}[1]\ r_{Q1}[2]\ r_{Q1}[3]\ w_9[4]\ r_{Q1}[4]\ c_{Q1}\ c_9$

Finally, it is worth noting that *LWS*=*CPS*, as shown below.

Theorem 7.5 *Given the specific transaction workload assumed in this chapter and given the assumption that window movements commit immediately,* LWS=CPS.

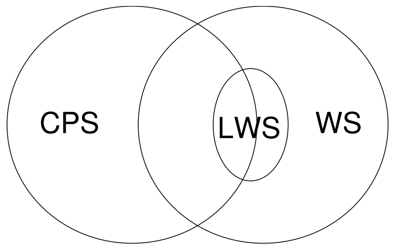


Figure 7.15: Relationship between commit-order preserving serializability (CPS), window serializability (WS), and latest window serializability (LWS)

Proof. Suppose that a history H is *LWS*. This means that, for any read-only transaction T_{Qk} whose operations are interleaved with at least one write transaction, it is true that all of the concurrent write transactions are serialized before T_{Qk}. Since write transactions are assumed to commit immediately, the serialization order is equivalent to the commit order. Hence, H is also *CPS*. Now suppose that H is *CPS*. This means that for any T_{Qk}, any concurrent write transactions are serialized before T_{Qk}. This is because the write transactions commit immediately and therefore they commit before the query transaction. Hence, the write transactions are serialized before T_{Qk}. This means that H is *LWS* because all the query transactions are guaranteed to see any concurrent window movements.

Chapter 8

Multi-Query Optimization of Sliding Window Aggregates

8.1 Introduction

The preceding chapters dealt with the time-evolving nature of data streams, and proposed and exploited the notion of update pattern awareness in sliding window query processing. The focus of this chapter is on multi-query optimization in the context of periodically re-executed monitoring queries. One example is a query that monitors the median packet length over a stream of IP packet headers (call it S), computed every two minutes over a ten-minute window. Using syntax similar to CQL, this query may be specified as follows.

```
Q1: SELECT MEDIAN(length)
    FROM S [WINDOW 10 min SLIDE 2 min]
```

The current workload may include queries similar to Q_1, but having additional `WHERE` predicates, different window sizes, or different periods (the expressions "`SLIDE` interval" and "period" will be used interchangeably). For instance, the workload may also include the following query.

```
Q2: SELECT MEDIAN(length)
    FROM S [WINDOW 14 min SLIDE 3 min]
```

These two queries may be posed by different users (e.g., network engineers), or by the same user, who wishes to summarize the network traffic at different time scales and may simultaneously ask similar aggregate queries over different window lengths. Other examples of applications that issue many aggregate queries in parallel over different window lengths include detecting bursts of unusual activity identified by abnormally high or low values of an aggregate such as `SUM` or `COUNT` [272, 277]. A burst of suspicious activity may be a denial-of-service attack on a network or a stock with an unusually high trading volume. Since the length of the burst is typically unknown (e.g., a short-lived burst could produce several unusual values in a span of a few seconds, whereas

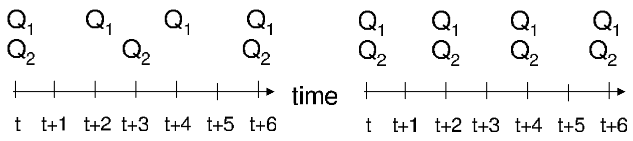


Figure 8.1: Two possible ways to schedule queries Q_1 and Q_2

a longer-term burst would have to generate a hundred suspicious packets over the last minute), a series of aggregates over a range of window lengths needs to be monitored.

Q_1 and Q_2 compute the same aggregate, but their window lengths and periods are different. Therefore, it is not obvious whether they can share *memory* or *computation*. Furthermore, even if one can determine which queries can share resources, the novel problem in this context is that similar queries having different `SLIDE` intervals may be scheduled for re-execution at different times. For example, Figure 8.1(a) illustrates a time axis with a possible execution sequence of Q_1 and Q_2 defined above. Assuming that Q_2 can re-use the answer of Q_1 computed over a shorter window, computation sharing can be exploited only once every six minutes (the least-common multiple of the `SLIDE` intervals). Another possibility is to always schedule Q_2 when Q_1 is due for a refresh, as in Figure 8.1(b). In this case, more computation is shared, but the savings in processing time may be defeated by the work expended on updating Q_2 more often than necessary.

Motivated by the above issues, this chapter presents a multi-query optimization framework for periodically-refreshed aggregates over sliding windows. The framework consists of two components: identifying which queries can share state and computation, and synchronizing the execution times of similar queries. The first component starts with existing techniques for sharing state among aggregates over different window sizes [16]. These techniques are extended in two directions: 1) allowing state sharing among queries containing aggregation, selection, and joins, and 2) discovering opportunities for shared computation, where the (intermediate) result of a previously executed query is used by another query. In the second part, a novel definition of the `SLIDE` clause is proposed, which gives rise to a novel specification of the semantics of periodic queries. Next, monitoring queries are modeled as periodic tasks and a query scheduler is developed on the basis of the earliest-deadline-first (EDF) algorithm [229]. The scheduler improves the two basic approaches discussed above in the following two ways.

1. A cost-based heuristic is proposed for deciding whether a query should be re-executed more often than necessary if its execution times can be synchronized with those of a similar query (as in Figure 8.1(b)).
2. Additional sharing opportunities are shown to arise during periods of overload, which are likely to be experienced by a DSMS due to changes in the (one-time and persistent) query workload and fluctuations in the stream arrival rates. Suppose that Q_1 is due for a refresh at time t and Q_2 at time $t + 1$, but the system was unable to run Q_1 at time t. There are two choices at time $t + 1$: clear the backlog by executing Q_1 and other late queries before moving on to Q_2, or recognize that Q_1 and Q_2 are similar and execute them together before

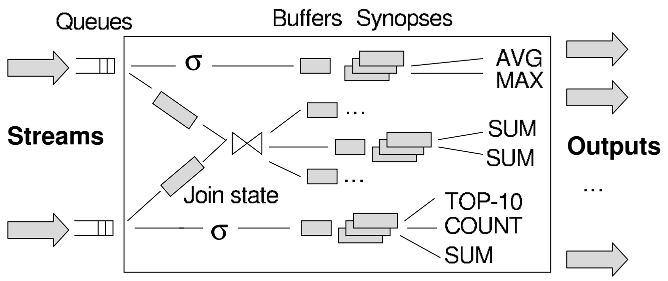


Figure 8.2: Assumed query processing architecture

moving on to other late queries (in which case Q_2 is much cheaper to evaluate because it can re-use the answer computed by Q_1). It will be shown that looking for similar queries in the "late query set" and the "currently-scheduled query set" increases overall throughput.

The remainder of this chapter is organized as follows. Section 8.2 introduces the system model and assumptions. The first component of the multi-query optimization framework is presented in Section 8.3 and the second in Section 8.4. The solution is evaluated experimentally in Section 8.5 and compared to related work in Section 8.6.

8.2 Preliminaries

The assumed query processing architecture is illustrated in Figure 8.2; it corresponds to Figure 7.3 described in Section 7.2.3 with the transaction manager abstracted out. A global query plan is constructed from selections, shared joins [52, 126, 155, 253], and final aggregates over time-based windows. Joins maintain state corresponding to (hash tables over) windows on their inputs, while aggregates are connected to *synopses*. The allowed aggregates are `SUM`, `COUNT`, `AVG`, `MAX`, `MIN`, `COUNT DISTINCT`, `QUANTILE`, or `TOP k` (the last three may be exact or approximate). Three types of synopses are used: running, interval, and basic interval (as discussed in Section 2.3.4 and illustrated in Figure 2.5). Each window or join of several windows may be associated with many synopses, which may be used by different aggregates or the same aggregates over different attributes. Furthermore, one synopsis may be shared by many queries, as will be discussed in Section 8.3. The part of the shared plan which lies upstream (to the left) of the buffers is executed continuously in the background; note that this part of the shared plan is weakest or weak non-monotonic. New results are continually deposited in the buffers and pre-aggregated, counted, or sketched on-the-fly; recall that a synopsis stores pre-aggregated values, frequency counters, or sketches of various intervals within the current sliding window. As described in Section 2.3.4, synopses are updated periodically by retrieving new data from their buffers, inserting the new interval, and removing intervals corresponding to expired tuples. When one or more queries are scheduled for re-execution, the appropriate synopses are probed to obtain the answer; query scheduling will be discussed in Section 8.4.

When the lifetime of a query expires, the query may be removed from the global plan, but shared components of the plan still in use must be retained. Conversely, a new query must be merged into the global plan (if not dropped outright, as could be the case if the system is overloaded[1]). If appropriate synopses exist (or the underlying windows are stored, as in the *WS* approach from Section 7.2.1), then the new query may begin execution immediately. An interesting case occurs when a matching synopsis is found, but its window length is too short for the new query. If so, then the time span of the matching synopsis may be extended by not deleting its oldest intervals for several updates. Note that this causes a delay before the query begins generating output. If all else fails, then a new synopsis may be built for the new query from scratch, but it will take one window length before the synopsis fills up with data.

8.3 Identifying and Processing Similar Queries

The first part of the solution identifies which queries can share state and computation. For now, the `SLIDE` intervals of queries are ignored. That is, the focus is on determining which queries benefit from shared evaluation if their re-execution times happen to coincide. State-sharing (abbreviated SS) and computation-sharing (abbreviated CS) rules are presented, beginning with simple aggregates and working up to complex multi-operator queries. The rule set presented here is by no means exhaustive and new rules can easily be added to the proposed framework.

8.3.1 Sliding Window Aggregates

The first type of queries considered are those consisting of a single aggregate operator over a single window. The following straightforward rule was motivated in Section 2.3.4:

SS1 A synopsis with parameters s and b may be used by queries computing the same aggregate, or different aggregates that require the same data per interval, over different window lengths of size ns, $1 \leq n \leq b$.

The extent of computation sharing depends upon the type of synopsis. Recall Figure 2.5 from Section 2.3.4 and observe that shared probing of a running or interval synopsis by queries over different window lengths is not faster than separate probing. In the former, different running intervals must be subtracted for different window sizes. In the latter, generally, a different set of non-overlapping intervals must be probed. Thus:

CS1 If multiple queries share a running or interval synopsis as state, then they may be executed together if they reference the same window length.

For example, `COUNT DISTINCT` and `TOP k` queries may be computed using an interval synopsis with frequency counts by subtracting the counts of the appropriate two intervals and performing appropriate post-processing. Note that the first part of the computation is shared by the two queries. On the other hand:

[1]Issues related to admission control are orthogonal to this work and are not discussed further.

CS2 All queries accessing the same basic interval synopsis may be computed together.

CS2 holds because answers over different window sizes may be returned along the way as the synopsis is being probed, from youngest interval to oldest. Regardless of the number of queries, at most b intervals are accessed. Since interval synopses access $\log b$ intervals per query, CS1 and CS2 suggest that a basic interval synopsis should be used if at least $\frac{b}{\log b}$ queries can be executed together. In this case, the basic interval synopsis exceeds the query efficiency of the interval synopsis, while using only half the space. However, to determine how many queries are scheduled to be executed together, their `SLIDE` intervals need to be examined. Therefore, further treatment of this issue is deferred to Section 8.4. Finally, the following rules hold for `AVG`:

SS2 If there exists a running synopsis with parameters s_1 and b_1 storing sums, and a running synopsis with parameters s_2 and b_2 storing counts, then they may be used by `AVG` over window size ns if their update times are aligned, $s_1 = s_2$, and $n \leq b_1, b_2$.

CS3 If SS2 holds and there exist queries computing `SUM`, `COUNT`, and `AVG` over the same window length, then they may be executed together.

SS2 requires the synopses to be aligned (i.e., they must be updated at the same times) so that the sum and count are computed over exactly the same window. The significance of CS3 is that `AVG` can simply divide the answers already computed by `SUM` and `COUNT`.

8.3.2 Aggregates over Joins

Synopsis Design

Weakest non-monotonic update patterns are exploited by the synopses in Figure 2.5; during updates, the oldest interval, which by then summarizes expired tuples, is dropped. However, recall from Chapter 3 that a sliding window join is a weak non-monotonic operator. As a result, the existing synopses are not suitable. For instance, in Figure 2.5, the aggregate value would be computed over the join results generated in the last $7s$ time units, which is different from join results over windows of length $7s$ each.

The existing synopses may be extended to handle join results by defining their intervals in terms of tuple expiration times[2]. A *basic-interval-join (BIJ) synopsis* over a join of windows with length $8s$ is illustrated in Figure 8.3 during an update at time $t + s$. The data stored in each interval are not shown, but, as before, may consist of pre-aggregated values, frequency counters, or sketches. Each interval is labeled with the expiration time ranges of the join results summarized within. First, the behaviour of the buffer preceding a BIJ synopsis is modified to pre-compute each of the following: $f((t+s, t+2s]), f((t+2s, t+3s]) \ldots, f((t+8s, t+9s])$. Now, $f((t, t+s])$ is dropped as it summarizes tuples that were due to expire between times t and $t+s$. Next, $f((t+8s, t+9s])$, which refers to new join results that will expire between times $t + 8s$ and $t + 9s$, is appended.

[2]Recall that a similar concept (calendar queue) was used in Chapter 3 in the context of storing intermediate results of weak non-monotonic queries.

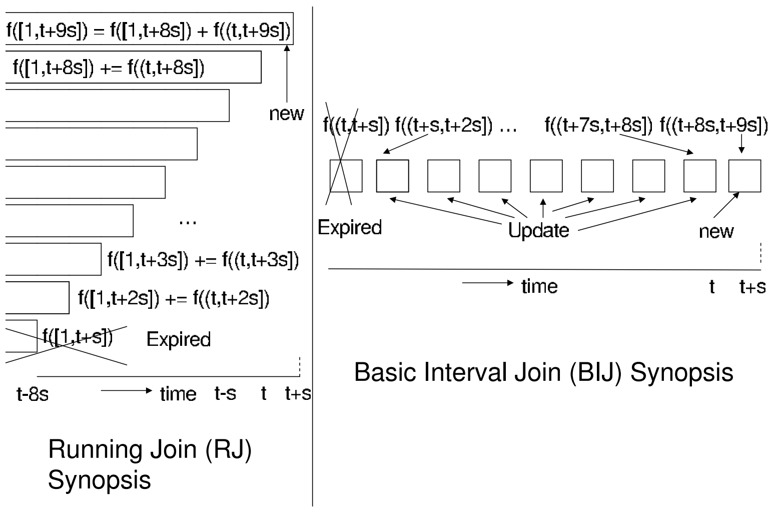

f([1,t+9s]) = f([1,t+8s]) + f((t,t+9s])
f([1,t+8s]) += f((t,t+8s])
new
...
f([1,t+3s]) += f((t,t+3s])
f([1,t+2s]) += f((t,t+2s])
f([1,t+s])
Expired
t-8s
time
t-s
t
t+s
Running Join (RJ) Synopsis
f((t,t+s]) f((t+s,t+2s]) ... f((t+7s,t+8s]) f((t+8s,t+9s])
Expired
Update
new
time
t
t+s
Basic Interval Join (BIJ) Synopsis

Figure 8.3: Examples of RJ and BIJ synopses

Finally, the remaining intervals are updated by merging the values of the corresponding pre-computed intervals. An *interval-join (IJ) synopsis* may also be constructed from an interval synopsis, in which case all $2b$ of its intervals must be updated every s time units. Finally, a *running-join (RJ) synopsis* for $b = 8$ is shown in Figure 8.3 during an update at time $t + s$. It requires its buffer to pre-compute aggregates for $f(t+s, t+2s]), f((t+s, t+3s]), \ldots, f((t+s, t+9s])$. The update removes $f([1, t+s])$, adds $f([1, t+9s])$, and updates the other intervals as illustrated. The space usage of the new synopses is the same as for the respective existing synopsis, but the update complexity grows because all the intervals must be modified.

Sharing State and Computation

Denote a join of two windows of sizes w_1 and w_2, respectively, as a (w_1, w_2)-join. First, suppose that a group of queries contains the same join and that $w_1 = w_2$. In this case, all the rules defined thus far apply. For instance, probing the first seven intervals of the BIJ synopsis in Figure 8.3, from right to left, yields a result over a $(7s, 7s)$-join. To see this, consider the $(8s, 8s)$-join results that are not part of the $(7s, 7s)$-join and observe that all such results belong to the oldest interval in the join synopsis. This is true because these join results must contain at least one tuple that is in the oldest part of its corresponding window (i.e., the part that has arrived between $7s$ and $8s$ time units ago). This means that the expiration times of these join results must be between now (time t) and time $t + s$. Similar reasoning also applies to the RJ synopsis.

Next, suppose that $w_1 \neq w_2$ and consider a `COUNT` query over a $(7s, 8s)$-join. Intuitively, the count should be higher than a $(7s, 7s)$-join, but lower than a $(8s, 8s)$-join. Therefore, it is necessary to subtract appropriate tuples from the oldest interval in the join synopsis. This example suggests a solution that modifies the join operator to flag results involving tuples from

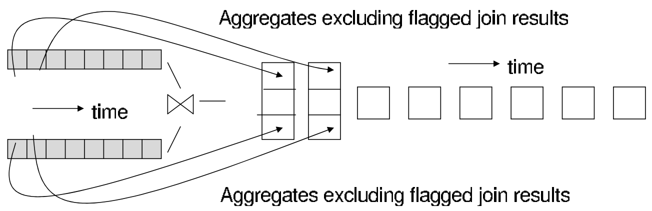

Aggregates excluding flagged join results
time
time
Aggregates excluding flagged join results

Figure 8.4: Example of a generalized BIJ synopsis

the oldest part of the first window, i.e., those which are not part of the $(7s, 8s)$-join. Figure 8.4 illustrates a generalized version of the BIJ synopsis, associated with an additional parameter g $(0 \leq g \leq b-1)$, that stores three pieces of data in its g oldest intervals ($g = 2$ and $b = 8$ in the figure). The IJ and RJ synopses may be generalized similarly. The values in the middle of each interval correspond to those of a BIJ synopsis. However, two additional sets of values are stored in the g oldest intervals, corresponding to aggregates that exclude flagged join results from the first and second window, respectively. In terms of buffer design, it must now separately pre-aggregate each of the three values needed for the g oldest intervals. The sharing rules for generalized join synopses are as follows.

SS3 A generalized BIJ, IJ, or RJ synopsis with parameters b, s, and g may be shared by aggregates over joins of windows with length ns each, where $1 \leq n \leq b$, as well as $(n_1 s, n_2 s)$-joins, where $b - g \leq n_1, n_2 \leq b$.

CS4 Queries accessing the same generalized IJ or RJ synopsis may be computed together if they reference the same window lengths in the join.

CS5 All queries accessing the same generalized BIJ synopsis may be computed together.

8.3.3 Aggregates with Selection

A simple solution for queries with various selection predicates is to execute them separately using separate synopses. However, synopses may be shared among queries with overlapping predicates. Assume that query predicates are in disjunctive normal form and that duplicate predicates are removed. Furthermore, assume that each term in the disjunction is an atom of the form *attribute op constant*, where *op* is a simple operation (equality, inequality, less-than, or greater-than). Define a query Q with predicate $p_1 \vee p_2 \vee \ldots \vee p_i$ to be *covered* by a set of synopses C if each synopsis computes the same aggregate (or contains the same data, e.g., counters or sketches) over the same input(s) as Q, and the terms p_k correspond exactly to the concatenation of predicates associated with the synopses in C. Additionally, C can be a *subcover* (*supercover*) of Q if a subset (superset) of terms p_k appear in the concatenation of its predicates. These definitions are illustrated using the following queries over a stream of IP packets headers S (the `WINDOW` and `SLIDE` clauses are omitted for clarity).

```
Q1: SELECT SUM(length) FROM S WHERE protocol=TCP
Q2: SELECT SUM(length) FROM S WHERE protocol=UDP
Q3: SELECT SUM(length) FROM S WHERE protocol=TCP OR protocol=UDP
Q4: SELECT MAX(length) FROM S WHERE protocol=TCP
Q5: SELECT MAX(length) FROM S WHERE user=U
Q6: SELECT MAX(length) FROM S WHERE protocol=TCP OR user=U
```

Suppose that Q_1 and Q_2 already have their own synopses. These two synopses cover Q_3 because they compute the same aggregate over the same stream and include both terms in Q_3's predicate. Moreover, Q_1's synopsis is a subcover for Q_3. Similarly, if Q_4 and Q_5 have their own synopses, then they cover Q_6. Additionally, if Q_3 had its own synopsis, then it would be a supercover for Q_1 and Q_2.

A `WHERE` predicate is disjoint if all its terms p_k are disjoint, e.g., if each term is an equality predicate on the same attribute. A set of synopses C is a *disjoint cover* for Q if it covers Q, each synopsis in C has a disjoint predicate, and the concatenation of predicates appearing in C is disjoint. For example, the synopses belonging to Q_1 and Q_2 are a disjoint cover for Q_3, but those belonging to Q_4 and Q_5 are not a disjoint cover for Q_6 (because the terms `protocol=TCP` and `user=U` are not necessarily disjoint).

The final definition needed for predicate matching deals with duplicates. A distributive aggregate f (recall Section 2.3.4) is duplicate-insensitive if, for two overlapping multi-sets X and Y, $f(X \cup Y) = f(X) \cup f(Y)$. f is duplicate-sensitive if $f(X \cup Y) = f(X) \cup f(Y) - f(X \cap Y)$. Both subtractable aggregates considered here (`SUM` and `COUNT`) are duplicate-sensitive, as are complex aggregates that require their synopses to store counters or sketches. `MIN` and `MAX` are duplicate-insensitive.

The above examples suggest that new queries may re-use synopses that cover them. For example, Q_3 can obtain an answer from Q_1's synopsis, an answer from Q_2's synopsis, and add them. This leads to a decrease in space usage and synopsis maintenance costs by one-third (only two synopses are needed). Motivated by this observation, the following three-step framework may be used for predicate overlap identification.

First, a set of *relevant synopses*, i.e., those which may be used to answer a new query Q, must be found. The update times of each relevant synopsis must be aligned (recall rule SS2). For duplicate-sensitive aggregates, the relevant synopses must form a disjoint cover of Q (otherwise, incorrect answers could be produced due to overlap among the predicates). For duplicate-insensitive aggregates, it suffices that the set is a cover of Q. Moreover, for subtractable aggregates, the relevant set may be a disjoint supercover of Q, provided that there exists a disjoint cover for all terms that appear in the supercover of Q, but not in Q (e.g., the synopses of Q_3 and Q_2 may be used to answer Q_1 by subtraction). The second step provides a rule for answering Q by combining the results obtained from each relevant synopsis. This operation may be maximum or minimum (e.g., Q_6 takes the maximum of Q_4's answer and Q_5's answer) addition (of sums, counts, counters or sketches), or subtraction (instead of addition, if a supercover is found). Finally, the third step returns a set of *candidate queries* for shared computation with Q. For example, Q_4 and Q_5 are candidates for Q_6. This procedure yields the following rules.

SS4 A non-empty set of relevant synopses may be used by a new query, provided that the window lengths of the matching synopses are at least as long as those of the new query[3].

CS6 If the relevant synopses are of type running, interval, generalized RJ, or generalized IJ, then the candidate queries, followed by the new query, may be executed together if they have the same window lengths.

CS7 If the relevant synopses are of type basic interval or generalized BIJ, then the candidate queries, followed by the new query, may be executed together.

Note that the soundness of the above framework follows from the distributive property of sliding window synopses employed in this chapter—merging partial answers from multiple synopses is equivalent to merging aggregated information (or frequency counts) from the intervals within an individual synopsis. Details regarding finding a set of relevant synopses for an incoming query as well as maintaining the sets of relevant synopses as queries come and go are orthogonal to this work and are not discussed further.

Finally, note that with the exception of SS4, CS6, and CS7, the rules presented in this section are universally applicable. That is, SS1 through SS3 should always be applied, rather than building separate synopses for similar queries over shorter window sizes. Similarly, CS1 through CS4 always lead to situations where at least some computation is amortized across multiple queries. However, SS4, CS6, and CS7 represent a space-versus-time tradeoff because saving space by sharing synopses may lead to higher query processing times (even though fewer synopses are maintained, multiple synopses must be probed to answer some queries).

8.4 Multi-Query Scheduling

Having defined a set of rules for state and computation sharing, this section presents the second component of the proposed solution, namely scheduling similar queries together in order to exploit sharing opportunities. Section 8.4.1 begins with a novel definition of the semantics of the `SLIDE` clause. Based upon this definition, an earliest-deadline-first scheduler is designed for periodically-executed queries (Section 8.4.2). The scheduler is then extended to schedule multiple queries together (Section 8.4.3). Finally, two improvements are given for synchronizing the schedules of similar queries (Sections 8.4.4 and 8.4.5).

8.4.1 Semantics of Periodic Query Re-Execution

Consider query Q accessing one or more synopses. Recall that a synopsis is updated when its buffer completes pre-aggregating the current interval, say $[t - s, t]$ (s is the interval between updates). This means that at time t, all the buffers that have been computing aggregates over this interval are ready to send pre-aggregated values to the synopses. After all the updates have

[3]Recall from Section 8.2 that it is also possible to extend the time span of existing synopses if the new query can tolerate a delay between its arrival and the time that it begins to produce answers.

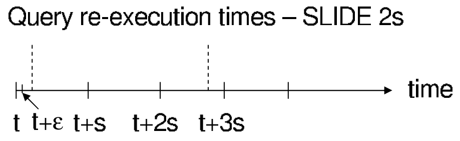


Figure 8.5: Semantics of periodic query re-execution

taken place at some time $t + \epsilon$, the synopses reflect the state of their window(s) as of time t. Assume that $\epsilon << s$, i.e., there should be ample time between updates to execute queries.

Let Q have a `SLIDE` interval of $2s$ and let Q's synopses have update times $t + is, i \in \mathbb{N}$. A time line is illustrated in Figure 8.5, showing a re-execution of Q some time between $t + \epsilon$ and $t+s$. The answer of Q reflects the state of its synopses as of time t. A `SLIDE` interval of ns means that the number of times Q's synopses are updated between consecutive re-executions *should* be no more than n. Hence, the next re-evaluation of Q should reflect the state of the synopses as of time $t + 2s$, therefore it should be done before time $t + 3s$, as illustrated. Note that the period of a query must be a multiple of s (the inter-update interval of its synopses).

Two practical remarks regarding the above definition are worth making. First, by defining only an upper bound on the number of synopsis updates between query re-executions, it is assumed that queries may be refreshed more often than specified. This assumption has two consequences. First, if the system is lightly loaded, then it may in fact be possible to re-execute queries more often; had the definition assumed a rigid re-execution interval, the system would experience periods of idle time during underload. Second, allowing a query to be refreshed sooner than required enables the synchronization of its re-execution times with those of a similar query, as illustrated in Figure 8.1(b). Again, had the definition required a fixed re-execution interval, resource sharing would have been significantly limited, as illustrated in Figure 8.1(a).

The second remark involves using the words *should* rather than *must* in the definition of periodic re-execution. This accommodates periods of overload, which are likely to occur due to fluctuations in the query workload and stream arrival rates. More precisely, it is assumed that for each periodic query, the DSMS must follow the above definition whenever possible, but is permitted to break it if necessary. Consequently, the definition allows at least the following two solutions for handling overload. First, the DSMS may drop a fraction of queries when overload is detected and block users from re-registering the dropped queries until the overload has subsided. Second, the DSMS may continue to re-execute all of its queries during overload, but temporarily increase all of their periods. The second solution is assumed in the remainder of this chapter as it ensures fairness across the query workload.

Finally, it may be argued that some users are interested in tracking events that are expected to occur regularly, say every three minutes, and would therefore insist that their queries be refreshed exactly every three minutes. However, these situations are different from the queries discussed in this chapter in the following two ways. First, tracking a specific event is not a periodic query, but rather a continuous query that keeps listening for new input and immediately reacts upon observing the specified event (e.g., by raising an alarm). Second, event tracking corresponds to selection queries, possibly with complex predicates, rather than aggregation over sliding windows.

8.4.2 Earliest-Deadline-First Scheduling

Figure 8.5 suggests the following execution sequence: at time t, the system updates all the synopses that are waiting for aggregates over $(t - s, t]$, finds all the queries due for a refresh, and attempts to execute them before the synopses are updated again. However, only shared sets of synopses need to be synchronized. Thus, while the sequence that started at time t is running, another sequence, corresponding to a different group of synopses and queries, may begin. The remainder of this section deals with "local" scheduling of one task sequence, containing queries that access the same group of synopses. The scheduling solutions are compatible with any underlying "global" scheduler, e.g., allocating a weighted time slice to each local scheduler.

To reflect the `SLIDE` semantics defined above, each persistent query is modeled as a task T_i with period $n_1 s$. Denote the rth re-execution of T_i as T_i^r and assign a *time-until-deadline* to each task, denoted $d(T_i)$, as follows. After T_i^r is done, set $d(T_i) = n_i$. After each update of the synopses, set $d(T_i) = d(T_i) - 1$. T_i^r is said to *execute on-time* if $d(T_i) \geq 0$ when it is done. For example, in Figure 8.5, $d(T_i) = 2$ after the query is executed, $d(T_i) = 1$ in the time interval $(t + s + \epsilon, t + 2s]$, $d(T_i) = 0$ at time $t + 2s + \epsilon$ until the next re-execution of the query (i.e., the query is executed on-time), and $d(T_i) = 2$ again after the re-execution.

Due to fluctuating stream arrival rates and changes in the query workload, it may not be possible to estimate how long it will take to execute each T_i^r, nor is it possible to schedule all tasks off-line. Consequently, the on-line earliest-deadline-first (EDF) algorithm [229] was chosen as a starting point for a (local) query scheduler, and is presented as Algorithm 9. For now, each query is scheduled separately. The algorithm maintains a task queue $Q(T)$ containing the current query workload. New queries are translated into new tasks and added to $Q(T)$; queries whose lifetimes have expired are removed. It is assumed that an update notification is sent when the interval currently being pre-aggregated in the buffers fills up and the synopses are due for an update. For clarity, Algorithm 9 and the remaining scheduling algorithms that will follow assume that queries and window updates are executed in isolation. Nevertheless, the concurrency control techniques presented in Chapter 7 are compatible with the scheduling algorithms described here.

The following observations regarding Algorithm 9 are worth noting:

- Even if T_i is re-executed late, $d(T_i)$ is set to n_i when it is done. The reasoning behind this is that lateness may imply system overload, therefore attempting to make up for the late re-execution by scheduling the next re-execution early could make the overload worse.
- Ties in line 7 may be broken arbitrarily. Alternatively, each query may be assigned a user-defined priority, in which case the highest-priority task (of all the tasks whose deadline is zero) is executed first.
- Algorithm 9 adaptively adjusts the query periods in response to system load. In underload, some tasks may be executed before their $d(T_i)$-values reach zero. During overload, the algorithm attempts to clear the backlog by executing queries with the largest negative value of $d(T_i)$. Since tasks may be executed when their $d(T_i)$-values are negative, their

Algorithm 9 Local scheduler

```
Input: task periods $n_i$, task queue $Q(T)$
Local variables: array $d(T_i)$, initially $d(T_i) = n_i$
 1  loop
 2    if update notification arrives then
 3        update all synopses
 4        decrement $d(T_i)$ for all tasks $T_i$
 5    end if
 6    if $Q(T)$ is not empty then
 7        execute task $T_i$ with the lowest value of $d(T_i)$
 8        reset $d(T_i) = n_i$
 9    end if
10  end loop
```

periods are lengthened implicitly. This behaviour may be thought of as a form of automatic *load shedding* (recall Section 2.4.3).

8.4.3 Scheduling Multiple Queries Together

Recall the computation sharing rules, CS1 through CS7, presented in Section 8.3. They identify which queries may be executed together if they are scheduled at the same time. Suppose that an application of these rules returns a set of query groups $G_1, G_2, \ldots, G_q$. Each group contains queries that may have different `SLIDE` intervals, and, if the queries use basic interval or generalized join synopses, they may also have different `WINDOW` lengths (by CS2, CS5, and CS7). In the remainder of this section, suppose that group G_1, contains seven queries, each computing `MAX` over the same window, and using a single basic interval synopsis with $s = 1$ minute (by CS2, queries within this group may have different window sizes). The `WINDOW` and `SLIDE` parameters of the queries are as follows.

```
Q1: ... [WINDOW 10 min SLIDE 2 min]
Q2: ... [WINDOW  5 min SLIDE 2 min]
Q3: ... [WINDOW  6 min SLIDE 2 min]
Q4: ... [WINDOW 15 min SLIDE 3 min]
Q5: ... [WINDOW 12 min SLIDE 3 min]
Q6: ... [WINDOW 20 min SLIDE 5 min]
Q7: ... [WINDOW 30 min SLIDE 5 min]
```

To incorporate multi-query scheduling into Algorithm 9, each group G_i is defined to be a single task T_i, with period n_i equal to the shortest period among its queries. This technique is referred to as *aggressive scheduling.* As illustrated in Figure 8.6(a), aggressive scheduling executes all seven queries in G_1 every two minutes. Another simple technique jointly executes similar queries only if they are due for a refresh at the same time. This can be achieved by splitting each group G_i into sub-groups, $G_{i,1}, G_{i,2}, \ldots, G_{i,q_i}$, containing queries with the same `SLIDE` interval.

Algorithm 10 Conservative scheduler

```
Input: sub-group periods $n_i$, task queue $Q(T)$
Local variables: array $d(T_{i,j})$, initially $d(T_{i,j}) = n_{i,j}$
1  loop
2    if update notification arrives
3        update all synopses
4        decrement $d(T_{i,j})$ for all tasks $T_{i,j}$
5    end if
6    if $Q(T)$ is not empty
7        let $v$ be the lowest $d(T_{i,j})$-value of any task $T_{i,j}$
8        let $V(T) = \{T_{i,j} \mid d(T_{i,j}) = v\}$
9        choose any task $T_{i,j}$ from $V(T)$
10       jointly execute $T_{i,j}$ and any other task $T_{i,m}$ in $V(T)$
11       reset the $d(T)$-values of all tasks just executed
12   end if
13 end loop
```

Furthermore, each task $T_{i,j}$ corresponds to all the queries in $G_{i,j}$, all of which are always executed together. This technique is given the name *conservative scheduling*. In the above example, G_1 is partitioned into $G_{1,1}$ containing Q_1, Q_2 and Q_3 (with $n_{1,1} = 2$ minutes), $G_{1,2}$ containing Q_4 and Q_5 (with $n_{1,2} = 3$ minutes), and $G_{1,3}$ containing Q_6 and Q_7 (with $n_{1,3} = 5$ minutes). The resulting schedule is shown in Figure 8.6(b).

Algorithm 10 summarizes conservative scheduling. Line 10 ensures that queries across sub-groups $G_{i,j}$ of the same group G_i are executed together when the `SLIDE` intervals of different sub-groups coincide. For instance, every six minutes, $G_{1,1}$ and $G_{1,2}$ may be executed together.

The meaning of "joint execution" in line 10 depends upon the computation sharing rule used to create the group. In case of CS1 or CS4, the synopsis is scanned, the result is saved (e.g., a sketch corresponding to the shared window length), and each query (separately) post-processes it as appropriate. In case of CS2 or CS5, the synopsis is scanned from youngest interval to oldest and queries over shorter windows are computed first. In case of CS3, then `SUM` and `COUNT` are executed first by probing their respective synopses, and their answers are then divided to obtain the average. In case of CS6, the predecessor queries are executed first and their answers are saved for use by the new query. Finally, in case of CS7, the predecessor queries are executed first, but answers over shorter windows may need to be saved for the new query. For example, if Q_4 and Q_5 both have a window length of $10s$, but Q_6 has a window length of $8s$, then, as Q_4 and Q_5 are processed, the maximum over a window of length $8s$ is saved.

The following rule applies to Algorithm 10 and its improvements that will be discussed later in this section.

CS8 If query Q is already registered with the system and another query, Q', arrives that is identical to Q (including the same window size) except that its period is longer, then Q' can share the output stream of Q.

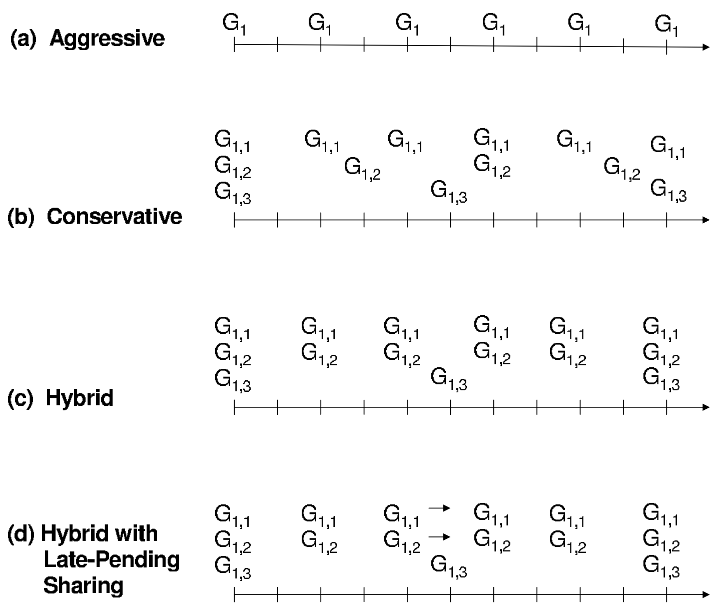


Figure 8.6: Execution schedules of queries in group G_1 using various scheduling techniques

In other words, if the system is already computing the same query (Q) more often, then it may as well shorten the period of Q' and re-use the results generated by Q. Note that when the lifetime of Q expires, the period of Q' may have to be reset to its original value and Q' may need to be relocated to a different sub-group $G_{i,j}$.

8.4.4 Hybrid Scheduling

The first improved scheduling technique is called *hybrid scheduling*. Using a relative cost model, it determines whether some sub-groups should be re-executed more often in order to synchronize their schedules with other sub-groups. The relative cost of a schedule may be computed by adding the synopsis access costs (i.e., how many intervals are accessed and what is needed to combine the pre-aggregated values from two intervals) as well as any post-processing costs (e.g., sorting frequency counts to find the top-k largest ones or combining answers from multiple synopses). Synopsis maintenance costs may be ignored as they do not vary across schedules. Hybrid scheduling is introduced using queries Q_1 through Q_7 from Section 8.4.3; recall that they are partitioned by conservative scheduling into three sub-groups: $G_{1,1}$, $G_{1,2}$, and $G_{1,3}$.

The first step is to calculate the relative cost of a single re-execution of each sub-group. Given that a basic interval synopsis is used, all the queries in $G_{1,1}$ can be computed by scanning the ten youngest intervals of the synopsis (answers over shorter windows, as needed by Q_2 and Q_3, will be computed along the way). The cost is 9 (comparisons to determine the maximum of ten maximum values stored in each interval). Similarly, the cost of $G_{1,2}$ is 14 and the cost of $G_{1,3}$ is

29. Next, the execution cost of $G_{1,1}$ and $G_{1,2}$ is needed, as incurred by conservative scheduling (recall Figure 8.6 (b)). Every six minutes (least-common-multiple of $n_{1,1}$ and $n_{1,2}$), both sub-groups are executed together for a cost of 14 (again, while scanning the synopsis to compute the maximum over the 15-minute window needed by Q_4, the answers of other queries are computed along the way). In the interim, $G_{1,1}$ is executed separately twice, for a cost of $9 * 2 = 18$, and $G_{1,2}$ once for a cost of 14. The total cost is 46 per six minutes, or 7.67 per minute.

Now suppose that $G_{1,2}$ is executed whenever $G_{1,1}$ is due for a refresh. In this case, both sub-groups are executed every two minutes. The cost per minute is $\frac{14}{2} = 7$. Therefore, the best way to execute sub-groups $G_{1,1}$ and $G_{1,2}$ is to schedule them both with a period of two minutes. In total, there are five possibilities: none of the sub-groups change their periods (which corresponds to conservative scheduling and costs 11.63 per minute), $G_{1,2}$ shortens its period to two minutes (11.4 per minute), both $G_{1,2}$ and $G_{1,3}$ shorten their periods to two minutes (14.5 per minute), $G_{1,3}$ changes its period to three minutes (12.67 per minute), and $G_{1,3}$ shortens its period to two minutes (16.83 per minute). Hybrid scheduling chooses the most efficient of these five possibilities, namely reducing the period of $G_{1,2}$ to that of $G_{1,1}$ and always executing queries in these two sub-groups together, as illustrated in Figure 8.6 (c).

Algorithm 10 may be used by hybrid scheduling without any modifications; the only difference is that the periods of some sub-groups are shorter. Hybrid scheduling is expected to outperform aggressive and conservative scheduling as it performs the right amount of sharing—whenever appropriate, it shortens the periods of some queries in order to synchronize them with other similar queries. However, hybrid scheduling is more expensive to maintain. As the query workload changes, the cost of separate and merged execution of various sub-groups must be recomputed. This is not expected to be a major source of overhead because the cost model is simple, and an efficient schedule may be computed using dynamic programming and memorizing the execution cost of overlapping subsets of sub-groups in order to prevent duplicate calculations.

At this point, it is worthwhile to revisit the observation made in Section 8.3.1 regarding the choice of basic interval synopsis versus interval synopsis. Recall that the former uses half as much space, but has slower query processing times. However, if a basic interval synopsis is always accessed by at least $\frac{b}{\log b}$ queries at the same time, then it outperforms the interval synopsis both in space usage and query processing time. Since hybrid scheduling is likely to align the schedules of similar queries, the breakpoint value of $\frac{b}{\log b}$ queries is more likely to be achieved.

8.4.5 Additional Sharing during Overload

Additional computation sharing is possible during overload, when some tasks have negative $d(T_{i,j})$-values. Define the *late set* to contain all tasks $T_{i,j}$ with $d(T_{i,j}) < 0$ and the *pending set* to contain all tasks $T_{i,j}$ with $d(T_{i,j}) = 0$. Suppose that a particular execution of queries in sub-group $G_{1,1}$ is late with $d(T_{1,1}) = -2$. Suppose further that a particular execution of queries in sub-group $G_{1,2}$ is also late, but with $d(T_{1,2}) = -1$. Although these two sub-groups may be scheduled together (they belong to the same group), Algorithm 10 (conservative scheduling) schedules $G_{1,1}$ and any other tasks with $d(T_{i,j})$-values of -2, before moving on to $G_{1,2}$. This is a missed

Algorithm 11 Conservative/Hybrid scheduler with late-pending sharing

```
Input: sub-group periods $n_i$, task queue $Q(T)$
Local variables: array $d(T_{i,j})$, initially $d(T_{i,j}) = n_{i,j}$
1  loop
2    if update notification arrives
3       update all synopses
4       decrement $d(T_{i,j})$ for all tasks $T_{i,j}$
5    end if
6    if $Q(T)$ is not empty
7       let $v$ be the lowest $d(T_{i,j})$-value of any task $T_{i,j}$
8       if $v < 0$
9          let $V(T) = \{T_{i,j} \mid d(T_{i,j}) \leq 0\}$
10      else
11         let $V(T) = \{T_{i,j} \mid d(T_{i,j}) = v\}$
12      end if
13      choose any task $T_{i,j}$ from $V(T)$
14      jointly execute $T_{i,j}$ and any other task $T_{i,m}$ in $V(T)$
15      reset the $d(T)$-values of all tasks just executed
16   end if
17 end loop
```

sharing opportunity, which, if exploited, could help clear the overload faster. Thus, a possible extension of conservative or hybrid scheduling, call it *late sharing*, schedules together matching sub-groups in the entire late set, not only those which have the lowest $d(T_{i,j})$-value.

There are possibilities for even more sharing. Suppose that, in addition to $G_{1,1}$ and $G_{1,2}$ being late, $G_{1,3}$ has a value of $d(T_{1,3}) = 0$ at the current time. It is possible to schedule all three sub-groups together. Although this shifts some of the system resources away from clearing the backlog of late tasks, it is beneficial in the long run because $G_{1,3}$ will not become late when the window slides again. Thus, the overall system throughput is likely to improve. This technique is referred to as *late-pending sharing* and is summarized in Algorithm 11. It differs from Algorithm 10 in that it defines $V(T)$ to be the union of the late set and pending set during overload (lines 8 and 9). Note that late sharing could be implemented by replacing line 9 with "let $V(T) = \{T_{i,j} \mid d(T_{i,j}) < 0\}$".

A possible schedule produced by hybrid scheduling with late-pending sharing in the context of the example from Section 8.4.3 is illustrated in Figure 8.6(d). As indicated by the arrows, suppose that $G_{1,1}$ and $G_{1,2}$ are late (with $d(T_{1,1}) = d(T_{1,2}) = -1$) and, at the same time, $G_{1,3}$ is pending (i.e., $d(T_{1,3}) = 0$). All queries in all three sub-groups are executed together.

8.5 Experimental Evaluation

8.5.1 Setting

The optimization rules and query scheduling techniques presented in this chapter were implemented in the DSMS query manager from Chapter 7 and tested in the same environment. As in the previous chapter, the input consists of simulated IP packet headers with randomly generated attribute values. Initially, the number of distinct source and destination IP addresses is set to 1000 each, whereas the number of distinct protocols and ports is 100 each. The tested workload consists of eight groups of periodic queries, each group computing top-k lists and quantiles over the bandwidth usage for one of the following: source IP address, destination IP address, source-destination pairs, protocol, port, and protocol-port pairs. Three workload sizes are considered: five, 10, 20, or 30 queries per group, giving a total number of periodic queries of 40, 80, 160, and 240, respectively. Each query has a randomly generated `WINDOW` length between 20 and 100 and a randomly generated `SLIDE` interval between one and half of its window length. Additionally, one-time queries requesting bandwidth usage statistics for a random source IP address are executed (one average) every second.

As per rule CS2, each group of periodic queries shares a basic interval synopsis storing appropriate counters. For simplicity, all the synopses are updated at the same time and all queries have the same priority. One-time queries are served by a separate running synopsis. The queue $Q(T)$ (recall Algorithms 9 through 11) is implemented as a heap sorted by task deadlines. However, rather than storing time-to-deadline values and decrementing them whenever the synopses are updated, it stores the actual deadline times, which need to be revised and re-inserted into the heap only after a task has been executed.

The variables measured in the experiments are throughput, in queries re-executed per second, as well as the average latency per query, defined by the additional number of times its synopsis is updated between re-executions. For example, if a query requests a `SLIDE` of two seconds (i.e., two synopsis updates between re-executions) and the synopsis always slides three times between each re-execution, then its average latency is one. However, if a query is re-executed too early, then its latency remains at zero. Every experiment is repeated with two average data rates: 1500 and 3000 tuples per second. When using the higher data rate, the number of distinct values of all the packet fields is doubled in order to force queries to do more work when generating answers (and cause overload). Results of varying the window size are omitted as the effects are the same as when the data rates change (in both cases, queries must access more data during re-execution).

Table 8.5.1 lists the scheduling techniques and their abbreviations. A baseline no-sharing technique was also implemented; it is equivalent to Algorithm 9 with tasks corresponding to sub-groups $G_{i,j}$. That is, no-sharing is similar to conservative scheduling, but does not execute two matching sub-groups together, even if they are due for a refresh at the same time. Late sharing and late-pending sharing were also added to conservative scheduling, but these always performed worse than when used with hybrid scheduling, and therefore will not be discussed further.

Table 8.1: Abbreviations of scheduling techniques

No-S	No sharing
AS	Aggressive scheduling
CS	Conservative scheduling
HS	Hybrid scheduling
H-LS	Hybrid scheduling with late sharing
H-LP	Hybrid scheduling with late-pending sharing

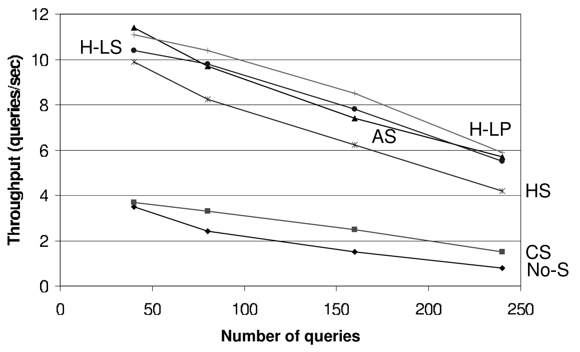


Figure 8.7: Throughput measurements using a data rate of 1500 tuples per second

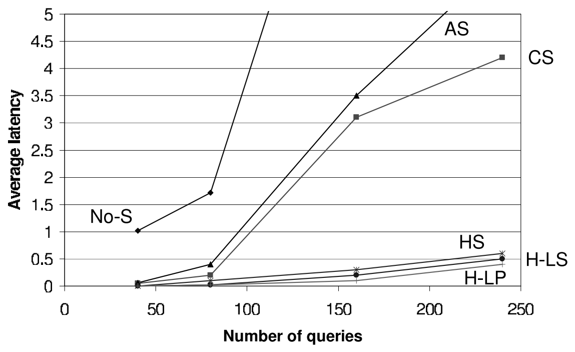


Figure 8.8: Average latency measurements using a data rate of 1500 tuples per second

8.5.2 Results with Low Data Rate

Results of experiments with a data rate of 1500 tuples per second are presented first. Throughput is shown in Figure 8.7 and average latency in Figure 8.8, for 40, 80, 160, and 240 queries. As expected, *No-S* and *CS* have the lowest throughput, with the gap between them growing as the number of queries increases and it is more likely that similar queries with different periods will be due for a refresh at the same time. Similarly, *No-S* and *CS* have high latency—*No-S* is particularly bad—even for a small number of queries, meaning that they cause system overload easily (overload occurs when the average latency is above zero). Note that *HS* alone easily doubles the throughput of *CS* and reduces its latency by an order of magnitude. In particular, *HS* was found to routinely shorten the periods of more than half the sub-groups in order to enable shared computation (recall Section 8.4.4). That is, if g sub-groups are used by conservative scheduling, it was often the case that hybrid scheduling requires only $\frac{g}{2}$ separately scheduled sub-groups (tasks). Additional improvements in throughput and latency can be gained via *H-LS* and *H-LP*, especially as the number of queries grows and the system falls into overload. Finally, note that *AS* achieves good throughput, but poor latency. This is because many queries are needlessly refreshed before their deadlines.

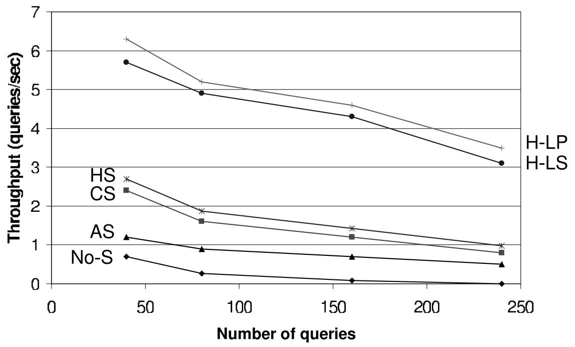


Figure 8.9: Throughput measurements using a data rate of 3000 tuples per second

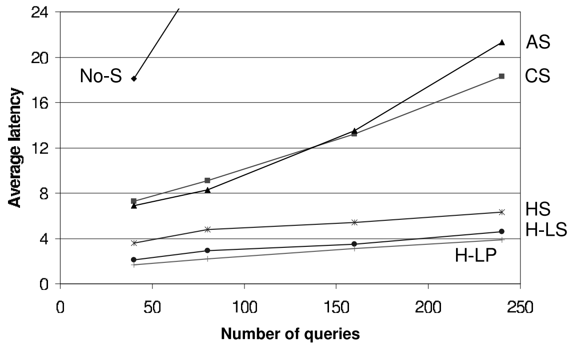


Figure 8.10: Average latency measurements using a data rate of 3000 tuples per second

8.5.3 Results with High Data Rate

Figures 8.9 and 8.10, respectively, graph the throughput and latency achieved with a data rate of 3000 tuples per second. In general, latencies are now higher and throughput is lower for all techniques due to heavier overload. *No-S* and *CS* continue to yield poor throughput and high latency (extremely high latency in case of no-sharing). However, the relative improvement of *HS* is now more modest, even though it continues to shorten the periods of more than half the sub-groups so that the queries within are always re-executed together. On the other hand, *H-LP* more than doubles the throughput of *HS*. Moreover, *H-LP* is the clear winner in throughput and latency. This is because overload is now more severe, therefore exploiting additional sharing opportunities across queries with different deadlines is crucial. Curiously, the throughput of *AS* drops significantly in this experiment, most likely because it is now more costly to re-execute queries, especially those over long window sizes that would normally be refreshed sporadically (but are done frequently by *AS*).

8.5.4 Lessons Learned

Based upon the experimental results, *HS* alone significantly improves the overall throughput if the system is underloaded or lightly overloaded. During overload, *H-LP* is the superior technique as it is the clear winner in the second set of experiments.

8.6 Comparison with Related Work

In terms of multi-query optimization in DSMSs, much of the previous work concentrates on shared execution of filters and joins [49, 84, 126, 155, 183, 253]. These works assume that incoming tuples are processed immediately, typically by updating any materialized joins affected by the new tuple and matching the new tuple against a query predicate index. This body of work is largely orthogonal to the framework introduced in this chapter and may be used to speed up the processing of the continuous part of the shared query plan illustrated in Figure 8.2.

Of the four existing DSMS solutions that consider periodic query execution, one deals with sharing state among simple aggregates, and proposes the running and interval synopses [16]. This chapter extends that work by enlarging the set of supported queries and proposing computation sharing rules in addition to state sharing. Another solution supports shared computation of similar (distributive) aggregates with different group-by columns [271] and may be incorporated into the rule set presented in Section 8.3. Furthermore, periodic re-evaluation of selections over unbounded streams and joins of streams with tables is discussed in [52]. The system model is similar to that illustrated in Figure 8.2 in that shared query plans are employed. Each query has a buffer and, when due for re-execution, activates the shared plan which also computes new results of similar queries into their buffers. Sliding windows are not used, therefore these new results computed during the re-execution of a different query will not expire when the original query is issued later on. Neither aggregation nor schedule synchronization are supported. Finally, the fourth solution deals with multi-query optimization for aggregates over sliding windows with different lengths and different selection predicates [156]. This work contains two steps that are relevant (and also orthogonal) to the ideas outlined in this chapter. First, given a particular workload, it is shown how to optimally reduce the number of intervals that need to be stored in a basic interval synopsis. This modification is fully compatible with the approach presented here. Second, different selection predicates are accommodated by storing query lineage within each tuple. In particular, each tuple contains a bitmap that specifies which queries it satisfies and aggregate values are pre-computed for each distinct bitmap vector. Effectively, this creates a separate basic interval synopsis per bit vector (rather than storing a separate synopsis per query). Hence, this technique is one possible implementation of the relevant synopsis creation and matching process outlined in Section 8.3.3. Note that schedule synchronization was not considered in [156].

The join synopses presented in Section 8.3.2 are based upon the calendar queue from Section 21, with appropriate extensions for multi-query processing. Computing approximate aggregates over data stream joins has been discussed before [83, 94], but these two works did not consider periodic updates of join synopses and sharing them across queries referencing joins with different window lengths.

Semantics of sliding windows are defined in [167], wherein a periodically-sliding window is represented as a sequence of overlapping extents. A new aggregate value is returned when an extent closes, i.e., when no more tuples will be mapped to it. This definition corresponds to the way in which synopses are updated: when the new interval computed by the buffer fills up, it is inserted into the synopsis. However, the semantics of the `SLIDE` clause presented in Section 8.4.1 separate the underlying synopsis updates from query execution times since it is not possible to execute all queries instantaneously after every window-slide.

Research on scheduling in DSMSs considers scheduling at the level of individual tuples and operators, i.e., choosing which tuple(s) to process at any given time [20, 46, 142, 195]. The goals are to bound the sizes of inter-operator queues or control output latency. Scheduling at the level of whole queries was not considered.

Also related to the scheduling approach described in this chapter is a deadline-based load shedding framework described in [261], where a window-slide and all pending query re-executions are dropped if the system can predict that there is insufficient time before the next window update (i.e., the next deadline) to perform the scheduled tasks. The approach taken in this chapter is different for the following two reasons. First, it does not require a mechanism for predicting the cost of advancing the windows and re-executing queries. Second, query re-executions are never dropped, but rather the periods of all queries are increased during overload. This avoids situations in which a query with a long period has one of its re-executions dropped and must wait a long time for the next refresh.

The earliest-deadline-first algorithm (EDF) was adapted to scheduling sliding window queries. The dynamic nature of the query workload, lack of reliable estimates of task completion times, and desire to prioritize late tasks rather than dropping them eliminate the use of other real-time and job-shop scheduling algorithms, among them rate-monotonic, shortest-time-to-completion, least-slack, and highest-value-first [3, 140]. Given that EDF is known to perform poorly during overload [131], one may wonder why it was chosen as a basis for a DSMS query scheduler. The answer is that EDF performs badly in terms of the goals of real-time systems, namely completing as many tasks as possible before their deadlines. EDF is a poor choice in this case because it gives priority to transactions which likely will not finish on time since their deadlines are very close. However, EDF is a plausible technique in the context of a DSMS, where the goal is to re-execute queries with the desired periods. Consequently, it is better to prioritize queries with earliest re-execution deadlines rather than unnecessarily executing another query that is not yet due for a refresh.

Moreover, it is known that EDF is optimal in terms of minimizing the maximum task lateness [150]. However, one of the assumptions behind the proof is that tasks are executed separately. An interesting area for future work involves finding an optimal scheduling algorithm for the scenario presented in this chapter, namely shared execution of periodic tasks with the possibility of overload. (both in terms of minimizing the maximum task lateness and maximizing system throughput).

Finally, the rules from Section 8.3 are related to the extensive body of work on traditional multi-query optimization [213] and answering queries using views [122], particularly in the context of queries with aggregation and group-by [130, 225]. Much of this work is likely to be applicable in the DSMS scenario and may be added to the framework presented in Section 8.3. Additionally, in this chapter, novel rules have been proposed for detecting similar queries based upon the properties of their aggregate functions (i.e., duplicate sensitivity and subtractability). Another difference between traditional multi-query optimization and the DSMS multi-query optimization framework proposed in this chapter is that the latter includes an additional step of synchronizing the schedules of similar queries.

Chapter 9

Conclusions and Future Work

9.1 Summary of Contributions

The focus of this research was on sliding window query processing over unbounded data streams, with an emphasis on persistent query semantics, query processing and optimization, and concurrency control. The central notion of update pattern awareness was introduced, which defines the way in which the inputs and outputs of persistent queries change over time as new data arrive on the stream and the windows slide forward (Chapter 3). The insight behind update pattern awareness is that a DSMS can often predict the order in which data expire from the underlying windows and intermediate operator state, unlike a traditional DBMS that deals with updates far less frequently, but allows users to modify any piece of data at any time (subject to any integrity constraints defined over the database). As a consequence, a data item present in a sliding window or intermediate operator state may often be associated with a lifetime of predictable length, whereas the lifetime of a row in a relational table is typically unknown in advance. Update pattern awareness was used to define the semantics of persistent queries, and build a framework for sliding window query processing and optimization. Order-of-magnitude performance gains were observed under various conditions, thereby identifying update-pattern-aware query processing as one of two key issues in efficient support for DSMS applications occupying the middle of the requirements spectrum illustrated in Figure 1.5.

In terms of sliding window maintenance, solutions were presented for efficient main-memory storage of time-evolving data having equal or variable lifetimes (Chapter 3). The need to store time-evolving data on disk was also motivated in the context of applications that perform off-line stream mining and analysis. It was shown that existing work on indexing sliding windows on disk applies only to data having fixed lifetimes. Next, an index was proposed for data items having variable lifetimes (Chapter 4). The insight behind the solution was to simultaneously partition the data by insertion and expiration times in order to ensure fast bulk-updates. In particular, the proposed index employs update pattern information in order to avoid bringing the entire data set into memory during an update. Significant improvements in index update times were observed, especially for large data sets.

Implementation and optimization of the sliding window join was also discussed (Chapter 5). The impact of eager versus lazy evaluation and eager versus lazy expiration was shown, as was the significance of the update patterns of the join inputs in terms of its implementation. Furthermore, a join ordering heuristic was proposed after showing that the classical ordering heuristic based upon join selectivities does not produce efficient orders in the context of sliding windows. The improved heuristic takes into account the stream arrival rates as well as the window sizes and examines only a small fraction of the possible join space.

Next, an update-pattern-aware algorithm was presented for detecting frequently occurring item types in sliding windows (Chapter 6). The objective was to avoid storing and maintaining a histogram of the entire window. However, rather than summarizing the underlying distribution of item types in sub-linear space, the insight behind the solution was to maintain exact frequencies of frequently occurring items in non-overlapping partitions of the sliding window. The proposed algorithm was shown to perform well on bursty TCP/IP streams containing a small set of popular item types.

A DSMS concurrency control mechanism was then motivated and developed (Chapter 7). It was shown that a window may slide forward while it, or an associated synopsis data structure, is scanned by a query. It was then explained that it is worthwhile to temporarily interrupt the processing of a query in order to update the state of the window, but only if the query can read the changes when resumed. The proposed solution was based upon a model that views DSMS data access as a mix of concurrent read-only and write-only transactions. Conflict serializability was shown to be insufficient in order to guarantee that suspended queries will read an up-to-date state of the sliding window when restarted. This observation prompted the need for stronger isolation levels for DSMSs running periodic queries over sliding windows. An update-pattern-aware transaction scheduler was also developed for enforcing the new isolation levels. The idea was to reorder the read operations of queries such that the tuples expected to expire next are not read until the end of the transaction. The scheduler is provably optimal in reducing the number of aborted transactions, and was experimentally shown to improve query freshness and response times while maintaining high transaction throughput.

Finally, multi-query optimization was identified as the second of two key issues in DSMS query processing over sliding windows (Chapter 8). The novel problem was that similar queries which could potentially be executed together may have different periods (`SLIDE` intervals). As a result, these queries are likely to be scheduled at different times, thereby missing the opportunity to share resources. Two components of a multi-query optimization framework were presented. First, an extensible set of rules was given for identifying which queries may share state and/or computation. The rule set covered queries over individual windows and joins, and queries containing disjunctive selection conditions. Second, a query scheduling algorithm was given that attempts to synchronize the re-execution times of similar queries whenever possible, including during periods of system overload. Experimental results showed the advantages of the proposed techniques in terms of system throughput.

9.2 Directions for Future Research

This research may be extended along the following lines.

- **Update-pattern-aware query algebra:** Chapter 3 introduced update pattern awareness in terms of sliding window maintenance, operator implementation, and query optimization. Adding update pattern awareness into a data stream algebra is a possibility for future research. An example of a problem in this domain is finding a set of algebraic operators that can express all possible weakest (or weak) non-monotonic queries.
- **Optimization of strict non-monotonic queries:** As discussed in Chapter 3, strict non-monotonic queries, such as those over count-based windows or those containing negation, are processed using the negative tuple approach. One way to speed up these types of queries could be to reduce the number of negative tuples flowing through the plan. This may be done by temporarily buffering some result tuples. For example, rather than producing results of negation right away, the query could wait and see if a matching tuple will arrive on the other input in the near future, which would cause a negative tuple to be produced had the original result been appended to the output stream immediately.
- **Adaptive query processing:** A persistent query plan may need to be adjusted over time. A possible question is whether update pattern awareness can also be used to improve the adaptivity of sliding window query plans to changing stream conditions. For instance, plans obtained by pulling up operators having complex update patterns may be more resilient to changes in the stream arrival rates because the plan would be shielded from a sudden increase in the number of negative tuples that must be processed. Another issue involves increasing the robustness of the join ordering heuristic from Chapter 5. One possibility is to overestimate the arrival rates of those streams which are expected to be more bursty so that the chosen join order does not become significantly sub-optimal if the estimated stream parameters deviate from the actual stream conditions.
- **Sliding window join processing using materialized sub-results:** Another possible extension of Chapter 5 involves join ordering in the presence of materialized sub-results (stored in update-pattern-aware data structures in order to minimize the view maintenance costs) or, more generally, selection of sub-results to materialize if spare memory is available.
- **Multi-query optimization:** Chapter 8 dealt with multi-query optimization in the context of periodic queries with aggregation (recall Figure 1.6). One possibility for future research is to investigate shared processing of continuous queries, as in Chapter 3. For instance, if two similar strict non-monotonic queries are executed using the same plan and one of the queries references a smaller window than the other, then two sets of negative tuples may need to be propagated—one to announce expirations from the larger window and one for the smaller window. Furthermore, the rule set from Chapter 8 could be extended to cover a wider range of queries. One particularly interesting type of queries are those containing user-defined aggregates. It may be possible to identify similar parts of two different user-defined aggregates and partially share state and/or computation.

- **Concurrency control for complex queries:** Chapter 7 may be extended to investigate concurrency control issues in complex query plans containing a number of pipelined window operators. Another problem appears when the same sub-query occurs more than once within a query, in which case the query may need to read the same data more than once during its execution (unless the sub-query can be flattened).
- **DSMS recovery:** The treatment of DSMS concurrency control in Chapter 7 may also be extended to include the semantics of data loss and crash recovery, e.g., loss of data for a particular time interval, which might make it impossible for queries to read a full window.
- **Update-pattern-aware DSMS security and access control:** This is an important problem because data streams, such as those transmitted by wireless sensors or generated by Internet users, are more easily accessible than information stored in a private database. As a simple example of a novel issue in this context, consider an application that stores one or more sliding windows with confidential (or otherwise privileged) information. When old data expire, should they be completely removed from the system, archived and remain confidential, or available for access by a larger group of users?

Bibliography

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J-H. Hwang, W. Lindner, A. Rasin, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 277–289, 2005.

[2] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug 2003.

[3] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1–12, 1988.

[4] L. Adamic and B. Huberman. The nature of markets in the world wide web. *Quarterly Journal of Electronic Commerce*, 1:5–12, 2000.

[5] C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for projected clustering of high dimensional data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 852–863, 2004.

[6] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 456–467, 2004.

[7] R. Albert, H. Jeong, and A.-L. Barabasi. Diameter of the world wide web. *Nature*, 401:130–131, 1999.

[8] M. Ali, W. Aref, R. Bose, A. Elmagarmid, A. Helal, I. Kamel, and M. Mokbel. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1295–1298, 2005.

[9] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 20–29, 1996.

[10] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *Proc. Int. Conf. on Parallel and Distr. Inf. Sys. (PDIS)*, pages 208–219, 1996.

[11] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Trans. Database Sys.*, 29(1):162–194, March 2004.

[12] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, to appear.

[13] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbets. Linear road: a stream data management benchmark. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 480–491, 2004.

[14] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 286–296, 2004.

[15] A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *ACM SIGMOD Record*, 33(3):6–11, 2004.

[16] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 336–347, 2004.

[17] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 261–272, 2000.

[18] A. Ayad and J. Naughton. Static optimization of conjunctive queries with sliding windows over unbounded streaming information sources. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 419–430, 2004.

[19] A. Ayad, J. Naughton, S. Wright, and U. Srivastava. Approximate streaming window joins under CPU limitations. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 142, 2006.

[20] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, 2004.

[21] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data streams. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 1–16, 2002.

[22] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 350–361, 2004.

[23] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 234–243, 2003.

[24] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 28–39, 2003.

[25] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 238–249, 2005.

[26] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 407–418, 2004.

[27] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 118–129, 2005.

[28] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Sys.*, 29(3):545–580, Sep 2004.

[29] S. Babu and J. Widom. StreaMon: an adaptive engine for stream query processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 931–932, 2004.

[30] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[31] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, 2005.

[32] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Load management and high availability in the Medusa distributed stream processing engine. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 929–930, 2004.

[33] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5:78–101, 1966.

[34] S. Ben-David, J. Gehrke, and D. Kifer. Detecting change in data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 180–191, 2004.

[35] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[36] H. Berthold, S. Schmidt, W. Lehner, and C.-J. Hamann. Integrated resource management for data stream systems. In *Proc. ACM Symp. on Applied Comp. (SAC)*, pages 555–562, 2005.

[37] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 757–768, 2005.

[38] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 61–71, 1986.

[39] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proc. Int. Conf. on Mobile Data Management (MDM)*, pages 3–14, 2001.

[40] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, Oct 1988.

[41] A. Bulut and A. K. Singh. SWAT: Hierarchical stream summarization. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 303–314, 2003.

[42] A. Bulut and A. K. Singh. A unified framework for monitoring data streams in real time. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 44–55, 2005.

[43] Y. Dora Cai, D. Clutter, G. Pape, J. Han, M. Welge, and L. Auvil. MAIDS: Mining alarming incidents from data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 919–920, 2004.

[44] M. Cammert, C. Heiz J. Krämer, T. Riemenschneider, M. Schwartzkopf, B. Seeger, and A. Zeiss. Stream processing in production-to-business software. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 168, 2006.

[45] M. Cammert, J. Krämer, B. Seeger, and S.Vaupel. An approach to adaptive memory management in data stream systems. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 137, 2006.

[46] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 838–849, 2003.

[47] Y. Chai, H. Wang, and P. Yu. Loadstar: Load shedding in data stream mining. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1302–1305, 2005.

[48] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 269–280, 2003.

[49] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, Aug 2003.

[50] S. Chandrasekaran and M. J. Franklin. Remembrance of streams past: overload-sensitive management of archived streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 348–359, 2004.

[51] J. Chen, D. DeWitt, and J. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 345–357, 2002.

[52] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 379–390, 2000.

[53] R. Cheng, B. Kao, S. Prabhakar, A. Kwan, and Y. Tu. Adaptive stream filters for entity-based queries with non-value tolerance. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 37–48, 2005.

[54] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 257–268, 2003.

[55] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Sys.*, 20(2):149–186, 1995.

[56] J. Chomicki and D. Toman. Implementing temporal integrity constraints using an active DBMS. *IEEE Trans. Knowledge and Data Eng.*, 7(4):566–582, 1995.

[57] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network: Model and algorithms. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 707–718, 2004.

[58] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 223–233, 2003.

[59] S. Cohen. User-defined aggregate functions: bridging theory and practice. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 49–60, 2006.

[60] G. Cormode and M. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 13–24, 2005.

[61] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: distributed tracking of approximate quantiles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 25–36, 2005.

[62] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, I. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 35–46, 2004.

[63] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 29–38, 2004.

[64] G. Cormode, S. Muthukrishnan, and W. Zhuang. What's different: distributed, continuous monitoring of duplicate-resilient aggregates on data streams. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 57, 2006.

[65] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Disc. and Data Mining*, pages 9–17, 2000.

[66] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: High performance network monitoring with an SQL interface. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 647–651, 2003.

[67] A. Das, S. Ganguly, M. Garofalakis, and R. Rastogi. Distributed set-expression cardinality estimation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 312–323, 2004.

[68] A. Das, J. Gehrke, and M. Riedewald. Semantic approximation of data stream joins. *IEEE Trans. Knowledge and Data Eng.*, 17(1):44–59, 2005.

[69] T. Dasu, S. Krishnan, S. Venkatasubramanian, and K. Yi. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *Proc. Symp. on the Interface of Statistics, Computing Science, and Applications*, 2006.

[70] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. ACM-SIAM Symp. on Discrete Alg. (SODA)*, pages 635–644, 2002.

[71] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[72] E. Demaine, A. Lopez-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proc. European Symp. on Algorithms (ESA)*, pages 348–360, 2002.

[73] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 627–644, 2006.

[74] B. Deng, Y. Jia, and S. Yang. Supporting efficient distributed top-k monitoring. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 496–507, 2006.

[75] M. Denny and M. Franklin. Predicate result range caching for continuous queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 646–657, 2005.

[76] M. Denny and M. Franklin. Operators for expensive functions in continuous queries. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 147, 2006.

[77] A. Deshpande. An initial study of overheads of Eddies. *ACM SIGMOD Record*, 33(1):44–49, 2004.

[78] A. Deshpande, C. Guestrin, and S. Madden. Using probabilistic models for data management in acquisitional environments. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 317–328, 2005.

[79] A. Deshpande and J. Hellerstein. Lifting the burden of history from adaptive query processing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 948–959, 2004.

[80] L. Ding, N. Mehta, E. Rundensteiner, and G. Heineman. Joining punctuated streams. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 587–604, 2004.

[81] L. Ding and E. Rundersteiner. Evaluating window joins over punctuated streams. In *Proc. Int. Conf. on Information and Knowledge Management (CIKM)*, pages 98–107, 2004.

[82] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 299–310, 2002.

[83] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 61–72, 2002.

[84] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 551–568, 2004.

[85] F. Douglis, J. Palmer, E. Richards, D. Tao, W. Hetzlaff, J. Tracey, and J. Lin. Position: short object lifetimes require a delete-optimized storage system. In *Proc. ACM SIGOPS European Workshop*, 2004.

[86] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proc. Int. Conf. on App., Tech., Arch., and Prot. for Comp. Communications (SIGCOMM)*, pages 323–336, 2002.

[87] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 115–126, 2001.

[88] C. Faloutsos. Sensor data mining: Similarity search and pattern analysis. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2002.

[89] C. Faloutsos and H. Jagadish. On B-tree indices for skewed distributions. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 363–374, 1992.

[90] P. Flajolet and G. N. Martin. Probabilistic counting. In *Proc. Symp. on Foundations of Comp. Sci. (FOCS)*, pages 76–82, 1983.

[91] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng. Optimizing refresh of a set of materialized views. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1043–1054, 2005.

[92] M. Franklin, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The hifi approach. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 290–304, 2005.

[93] M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, 34(2):18–26, 2005.

[94] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing data-stream join aggregates using skimmed sketches. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 569–586, 2004.

[95] V. Ganti, J. Gehrke, and R. Ramakrishnan. DEMON: Mining and monitoring evolving data. *IEEE Trans. Knowledge and Data Eng.*, 13(1):50–63, 2001.

[96] H. Garcia-Molina, J. Ullman, and J. Widom. *Database system implementation*. Prentice Hall, 2000.

[97] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 635, 2002.

[98] B. Gedik and L. Liu. Quality-aware distributed data delivery for continuous query services. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 419–430, 2006.

[99] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *Proc. Int. Conf. on Information and Knowledge Management (CIKM)*, pages 171–178, 2005.

[100] T. Ghanem, W. Aref, and A. Elmagarmid. Exploiting predicate-window semantics over data streams. *ACM SIGMOD Record*, 35(1):3–8, 2006.

[101] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 541–550, 2001.

[102] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 331–342, 1998.

[103] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proc. ACM Symp. on Parallel Alg. an Arch. (SPAA)*, pages 281–291, 2001.

[104] P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proc. ACM Symp. on Parallel Alg. an Arch. (SPAA)*, pages 63–72, 2002.

[105] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 79–88, 2001.

[106] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 454–465, 2002.

[107] L. Golab, K. G. Bijay, and M. T. Özsu. On concurrency control in sliding window queries over data streams. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 608–626, 2006.

[108] L. Golab, D. DeHaan, E. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. ACM SIGCOMM Internet Measurement Conf. (IMC)*, pages 173–178, 2003.

[109] L. Golab, S. Garg, and M. T. Özsu. On indexing sliding windows over on-line data streams. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 712–729, 2004.

[110] L. Golab and M. T. Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[111] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 500–511, 2003.

[112] L. Golab and M. T. Özsu. Update-pattern aware modeling and processing of continuous queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 658–669, 2005.

[113] L. Golab, P. Prahladka, and M. T. Özsu. Indexing time-evolving data with variable lifetimes. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 265–274, 2006.

[114] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analyzing massive RFID data sets. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 83, 2006.

[115] A. Gounaris, N. Paton, A. Fernandes, and R. Sakellariou. Adaptive query processing: A survey. In *Proc. British Nat. Conf. on Databases (BNCOD)*, pages 11–25, 2002.

[116] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 152–159, 1996.

[117] J. Greenwald and F. Khanna. Space efficient on-line computation of quantile summaries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 58–66, 2001.

[118] J. Greenwald and F. Khanna. Power-conserving computation of order-statistics over sensor networks. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 275–285, 2004.

[119] S. Guha and A. McGregor. Approximate quantiles and the order of the stream. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 273–279, 2006.

[120] S. Guha and K. Shim. Offline and data stream algorithms for efficient computation of synopsis structures. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, page 1364, 2005.

[121] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 287–298, 1999.

[122] A. Halevy. Answering queries using views: a survey. *The VLDB Journal*, 10(4):270–294, 2001.

[123] M. Hammad, W. Aref, and A. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 75–84, 2003.

[124] M. Hammad, W. Aref, and A. Elmagarmid. Optimizing in-order execution of continuous queries over streamed sensor data. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 143–146, 2005.

[125] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A. Elmagarmid. Efficient execution of sliding window queries over data streams. Technical Report CSD TR 03-035, Purdue University, 2003.

[126] M. Hammad, M. J. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 297–308, 2003.

[127] M. Hammad, M. Mokbel, M. Ali, W. Aref, A. Catlin, A. Elmagarmid, M. Eltabakh, M. Elfeky, T. Ghanem, R. Gwadera, I. Ilyas, M. Marzouk, and X. Xiong. Nile: a query processing engine for data streams. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 851, 2004.

[128] D. Han, C. Xiao, R. Zhou, G. Wang, H. Huo, and X. Hui. Load shedding for window joins over streams. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 472–483, 2006.

[129] E. Hanson, C. Carnes, L. Huang, M. Konyala, and L. Noronha. Scalable trigger processing. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 266–275, 1999.

[130] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 205–216, 1996.

[131] J. Haritsa, M. Carey, and M. Livny. Earliest-deadline scheduling for real-time database systems. In *In Proc. IEEE Real-Time Systems Symp.*, 1991.

[132] J. M. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 171–182, 1997.

[133] J. M. Hellerstein, W. Hong, and S. Madden. The sensor spectrum: Technology, trends, and requirements. *ACM SIGMOD Record*, 32(4):22–27, Dec. 2003.

[134] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 779–790, 2005.

[135] I. Ilyas. *Rank-aware query processing and optimization.* PhD thesis, Purdue University, 2004.

[136] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 113–124, 1995.

[137] A. Jain, E. Y. Chang, and Y-F. Wang. Adaptive stream resource management using Kalman Filters. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 11–22, 2004.

[138] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the Stream Processing Core. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 431–442, 2006.

[139] S. Jeffery, G. Alonso, M. Franklin, W. Hong, and J. Widom. A pipelined framework for on-line cleaning of sensor data streams. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 140, 2006.

[140] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *In Proc. IEEE Real-Time Sytems Symp.*, pages 112–122, 1985.

[141] Q. Jiang and S. Chakravarthy. Queueing analysis of relational operators for continuous data streams. In *Proc. Int. Conf. on Information and Knowledge Management (CIKM)*, pages 271–278, 2003.

[142] Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *Proc. British Nat. Conf. on Databases (BNCOD)*, pages 16–30, 2004.

[143] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, 2005.

[144] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in Gigascope. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1079–1088, 2005.

[145] T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Streams, security and scalability. In *Proc. 19th Annual IFIP Conf. on Data and Applications Security, LNCS 3654*, pages 1–15, 2005.

[146] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 106–117, 1998.

[147] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 341–352, 2003.

[148] R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of threshold counts. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 289–300, 2006.

[149] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Disc. and Data Mining*, pages 91–101, 2002.

[150] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[151] F. Korn, S. Muthukrishnan, and Y. Wu. Modeling skew in data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 181–192, 2006.

[152] N. Koudas and D. Srivastava. Data stream query processing: A tutorial. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, page 1149, 2003.

[153] J. Krämer and B. Seeger. PIPES - a public infrastructure for processing and exploring streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 925–926, 2004.

[154] J. Krämer and B. Seeger. A temporal foundation for continuous queries over data streams. In *Proc. 11th Int. Conf. on Management of Data (COMAD)*, pages 70–82, 2005.

[155] S. Krishnamurthy, M. Franklin, J. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 972–986, 2004.

[156] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 623–634, 2006.

[157] Y-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 492–503, 2004.

[158] I. Lazaridis and S. Mehrotra. Capturing sensor-generated time series with quality guarantees. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 429–440, 2003.

[159] K. C. K. Lee, H. V. Leong, and A. Si. QUAY: A data stream processing system using chunking. In *Proc. Int. Database Eng. and App. Symp. (IDEAS)*, pages 17–26, 2004.

[160] L. K. Lee and H. F. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 290–297, 2006.

[161] W. Leland, M. Taqqu, M. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Trans. on Networking*, 2(1):1–15, 1994.

[162] A. Lerner and D. Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 345–356, 2003.

[163] A. Lerner and D. Shasha. The virtues and challenges of ad hoc + streams querying in finance. *IEEE Quarterly Bulletin on Data Engineering*, 26(1):49–56, 2003.

[164] F. Li, C. Chang, G. Kollios, and A. Bestavros. Characterizing and exploiting reference locality in data stream applications. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 81, 2006.

[165] H-G. Li, S. Chen, J. Tatemura, D. Agrawal, K. S. Candan, and W-P. Hsiung. Safety guarantee of continuous join queries over punctuated data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[166] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Record*, 34(1):39–44, 2005.

[167] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 311–322, 2005.

[168] H.-S. Lim, J.-G. Lee, M.-J. Lee, K.-Y. Whang, and I.-Y. Song. Continuous query processing in data streams using duality of data and queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 313–324, 2006.

[169] J. Lin, E. Keogh, S. Lonardi, J. P. Lankford, and D. M. Nystrom. Visually mining and monitoring massive time series. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Disc. and Data Mining*, pages 460–469, 2004.

[170] X. Lin, H. Lu, J. Xu, and J. Xu Yu. Continuously maintaining quantile summaries of the most recent N elements over a data stream. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 362–373, 2004.

[171] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 502–513, 2005.

[172] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1338–1341, 2005.

[173] B. Liu, Y. Zhu, and E. Rundensteiner. Run-time operator state spilling for memory intensive long running queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 347–358, 2006.

[174] H. Liu, Y. Lu, J. Han, and J. He. Error-adaptive and time-aware maintenance of frequency counts over data streams. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 484–495, 2006.

[175] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Trans. Knowledge and Data Eng.*, 11(4):610–628, 1999.

[176] B. Livezey and R. R. Muntz. Aspen: A stream processing environment. In *Proc. Parallel Architectures and Languages Europe, Volume II: Parallel Language*, pages 374–388, 1989.

[177] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A native extension of SQL for mining data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 873–875, 2005.

[178] G. Luo, J. Naughton, and C. Ellmann. A non-blocking parallel spatial join algorithm. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 697–705, 2002.

[179] L. Ma, S. Viglas, M. Li, and Q. Li. Stream operators for querying data streams. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 404–415, 2005.

[180] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 555–566, 2002.

[181] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proc. Symp. on Operating Systems Design and Implementation (OSDI)*, 2002.

[182] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 491–502, 2003.

[183] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 49–60, 2002.

[184] A. Manjhi, S. Nath, and P. Gibbons. Tributaries and deltas: efficient and robust aggregation in sensor network streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 287–298, 2005.

[185] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 767–778, 2005.

[186] G. Manku, S. Rajagopalan, and B. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 251–262, 1999.

[187] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 346–357, 2002.

[188] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 398–412, 2005.

[189] M. F. Mokbel, M. Lu, and W. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 251–262, 2004.

[190] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 245–256, 2003.

[191] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 635–646, 2006.

[192] S. Muthukrishnan. Data streams: Algorithms and applications. Manuscript.

[193] S. Nath, P. Gibbons, S. Seshan, and Z. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. ACM Conf. on Embedded Networked Sensor Sys. (SENSYS)*, pages 250–262, 2004.

[194] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 563–574, 2003.

[195] Z. Ou, G. Yu, Y. Yu, S. Wu, X. Yang, and Q. Deng. Tick scheduling: A deadline based optimal task scheduling approach for real-time data stream systems. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 725–730, 2005.

[196] T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 338–349, 2004.

[197] D.S. Parker. *Stream Data Analysis in Prolog.* MIT Press, 1990.

[198] N. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[199] A. Pavan and S. Tirthapura. Range-efficient computation of F_0 over massive data streams. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 32–43, 2005.

[200] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, Jun 1995.

[201] L. Qiao, D. Agrawal, and A. El Abbadi. Supporting sliding window queries for continuous data streams. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 85–94, 2003.

[202] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, and K. S. Beyer. Srql: Sorted relational query language. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 84–97, 1998.

[203] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 353–364, 2003.

[204] F. Reiss and J. Hellerstein. Data triage: an adaptive architecture for load shedding in telegraphCQ. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 155–156, 2005.

[205] S. Rizvi, S. Jeffery, S. Krishnamurthy, M. Franklin, N. Burkhart, A. Edakkunni, and L. Liang. Events on the edge. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 885–887, 2005.

[206] E. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: continuous query engine with heterogeneous-grained adaptivity. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1353–1356, 2004.

[207] E. Ryvkina, A. Maskey, I. Adams, B. Sandler, C. Fuchs, M. Cherniack, and S. Zdonik. Revision processing in a stream processing engine: A high-level design. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 141, 2006.

[208] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Sys.*, 29(2):282–318, June 2004.

[209] B. Salzberg and V. Tsotras. A comparison of access methods for time evolving data. *ACM Computing Surveys*, 31(2):158–221, June 1999.

[210] A. Schmidt, C. Jensen, and S. Saltenis. Expiration times for data management. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 36, 2006.

[211] S. Schmidt, H. Berthold, and T. Legler. QStream: Deterministic querying of data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1365–1368, 2004.

[212] S. Schmidt, T. Legler, S. Schar, and W. Lehner. Robust real-time query processing with QStream. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1299–1301, 2005.

[213] T. Sellis. Multiple-query optimization. *ACM Trans. Database Sys.*, 13(1):23–52, 1988.

[214] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 430–441, 1994.

[215] P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: A model for sequence databases. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 232–239, 1995.

[216] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 99–110, 1996.

[217] S. Seshardi, V. Kumar, and B. Cooper. Optimizing multiple queries in distributed data stream systems. In *Proc. IEEE Int. Workshop on Networking Meets Databases (NetDB)*, 2006.

[218] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 827–838, 2004.

[219] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 25–36, 2003.

[220] M. Sharaf, A. Labrinidis, P. Chrysanthis, and K. Pruhs. Freshness-aware scheduling of continuous queries in the dynamic web. In *Proc. Int. Workshop on the Web and Databases (WebDB)*, pages 73–78, 2005.

[221] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 301–312, 2006.

[222] N. Shivakumar and H. García-Molina. Wave-indices: indexing evolving databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 381–392, 1997.

[223] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proc. ACM Conf. on Embedded Networked Sensor Sys. (SENSYS)*, pages 239–249, 2004.

[224] A. Silberstein, K. Munagala, and J. Yang. Energy-efficient monitoring of extreme values in sensor networks. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2006.

[225] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering queries with aggregation using views. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 318–329, 1996.

[226] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 250–258, 2005.

[227] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Princ. of Database Sys. (PODS)*, pages 263–274, 2004.

[228] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 324–335, 2004.

[229] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms.* Kluwer Academic Publishers, 1998.

[230] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[231] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[232] B. Subramanian, T. W. Leung, S. L. Vandenberg, and S. B. Zdonik. The aqua approach to querying lists and trees in object-oriented databases. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 80–89, 1995.

[233] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proc. USENIX Annual Technical Conf.*, 1998.

[234] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE Trans. Knowledge and Data Eng.*, 18(3):377–391, 2006.

[235] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: Producing fast join results on streams through rate-based optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 371–382, 2005.

[236] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 309–320, 2003.

[237] N. Tatbul and S. Zdonik. Dealing with overload in distributed stream processing systems. In *Proc. IEEE Int. Workshop on Networking Meets Databases (NetDB)*, 2006.

[238] N. Tatbul and S. Zdonik. A subset-based load shedding approach for aggregation queries over data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[239] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 321–330, 1992.

[240] F. Tian and D. DeWitt. Tuple routing strategies for distributed Eddies. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 333–344, 2003.

[241] W. H. Tok and S. Bressan. Efficient and adaptive processing of multiple continuous queries. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 215–232, 2002.

[242] G. Trajcevski, P. Scheuermann, O. Ghica, A. Hinze, and A. Voisard. Evolving triggers for dynamic environments. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 1039–1048, 2006.

[243] N. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. Multi-query optimization for sensor networks. In *Proc. Int. Conf. on Distr. Comp. in Sensor Sys. (DCOSS)*, pages 307–321, 2005.

[244] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[245] P. Tucker, D. Maier, T. Sheard, and L. Faragas. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowledge and Data Eng.*, 15(3):555–568, 2003.

[246] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Quarterly Bulletin on Data Engineering*, 23(2):27–33, June 2000.

[247] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 501–510, 2001.

[248] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 130–141, 1998.

[249] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 37–48, 2002.

[250] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-join queries over streaming information sources. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 285–296, 2003.

[251] E. Vossough and J. R. Getta. Processing of continuous queries over unlimited data streams. In *Proc. Int. Conf. on Database and Expert Sys. App. (DEXA)*, pages 799–809, 2002.

[252] H. Wang, C. Zaniolo, and R. Luo. Atlas: A small but complete SQL extension for data mining and data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1113–1116, 2003.

[253] S. Wang, E. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[254] W. Wang, J. Li, D. Zhang, and L. Guo. Processing sliding window join aggregate in continuous queries over data streams. In *Proc. East-European Conf. on Advances in Databases and Inf. Sys. (ADBIS)*, pages 348–363, 2004.

[255] G. Weikum and G. Vossen. *Transactional Information Systems. Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kauffman, 2002.

[256] K.-Y. Whang and R. Krishnamurthy. Query optimization in a memory-resident domain relational calculus database system. *ACM Trans. Database Sys.*, 15(1):67–95, 1990.

[257] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

[258] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. Int. Conf. on Parallel and Distr. Inf. Sys. (PDIS)*, pages 68–77, 1991.

[259] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 407–418, 2006.

[260] K.-L. Wu, S.-K. Chen, and P. Yu. Interval query indexing for efficient stream processing. In *Proc. Int. Conf. on Information and Knowledge Management (CIKM)*, pages 88–97, 2004.

[261] S. Wu, G. Yu, Y. Yu, Z. Ou, X. Yang, and Y. Gu. A deadline-sensitive approach for real-time processing of sliding windows. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 566–577, 2005.

[262] J. Xie, J. Yang, and Y. Chen. On joining and caching stochastic streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 359–370, 2005.

[263] Y. Xing, J-H. Hwang, U. Cetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[264] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 791–802, 2005.

[265] J. Xu, X. Lin, and X. Zhou. Space efficient quantile summary for constrained sliding windows on a data stream. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 34–44, 2004.

[266] Y. Yao and J. Gehrke. Query processing for sensor networks. In *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, pages 233–244, 2003.

[267] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *Proc. Int. Conf. on Data Eng. (ICDE)*, pages 189–200, 2003.

[268] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, V. Subrahmanian, and R. Zicari. *Advanced database systems.* Morgan Kaufmann, 1997.

[269] D. Zhang, J. Li, K. Kimeli, and W. Wang. Sliding window based multi-join algorithms over distributed data streams. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 139, 2006.

[270] D. Zhang, J. Li, Z. Zhang, W. Wang, and L. Guo. Dynamic adjustment of sliding window over data streams. In *Proc. Int. Conf. on Advances in Web-Age Inf. Management (WAIM)*, pages 24–33, 2004.

[271] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 299–310, 2005.

[272] X. Zhang and D. Shasha. Better burst detection. In *Proc. Int. Conf. on Data Eng. (ICDE)*, page 146, 2006.

[273] Y. Zhou, Y. Yan, F. Yu, and A. Zhou. PMJoin: Optimizing distributed multi-way stream joins by stream partitioning. In *Proc. Int. Conf. on Database Syst. for Advanced App. (DASFAA)*, 2006.

[274] S. Zhu and C. Ravishankar. A scalable approach to approximating aggregate queries over intermittent streams. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 85–94, 2004.

[275] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 431–442, 2004.

[276] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 358–369, 2002.

[277] Y. Zhu and D. Shasha. Efficient elastic burst detection in data streams. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Disc. and Data Mining*, pages 336–345, 2003.

[278] G. K. Zipf. *Human Behaviour and the Principle of Least-Effort.* Addison-Wesley, 1949.

Appendix A

Proof of Theorem 4.1

This appendix presents a proof of Theorem 4.1 from Section 4.3, which asserts that round-robin partitioning produces sub-indices whose sizes are more uniform than those created by chronological partitioning. Recall the notation used in Chapter 4: n is the total number of sub-indices[1], S is the upper bound on the lifetimes of data items[2], and Δ is the interval between two consecutive index updates[3]. Define $p = \frac{S}{\Delta}$ to be the number of times the index must be updated before all the data items initially present in the index have expired. Next, enumerate the insertion timestamp (ts) and expiration timestamp (exp) ranges as one through p; for simplicity, set $exp = exp - S$ so that the two ranges are the same. Without loss of generality, suppose that range 1 is the oldest and range p is the youngest. Furthermore, assume that $p > \sqrt{n}$, i.e., each sub-index is required to span at least two refresh intervals (otherwise, each sub-index would store tuples with one particular ts range and one particular exp range, and there would be no difference between chronological partitioning and round-robin partitioning). Consequently, the smallest value that p can take is $2\sqrt{n}$, in which case each sub-index spans a time of two refresh intervals of the insertion and expiration times. Given this notation and assumptions, a 2x2 chronologically partitioned index from Figure 4.1 (repeated here as Figure A.1) may be characterized as follows.

- I_1 stores tuples with ts ranges of one through $\frac{p}{2}$ and exp ranges of one through $\frac{p}{2}$.
- I_2 stores tuples with ts ranges of one through $\frac{p}{2}$ and exp ranges of $\frac{p}{2}$ through p.
- I_3 stores tuples with ts ranges of $\frac{p}{2}$ through p and exp ranges of one through $\frac{p}{2}$.
- I_4 stores tuples with ts ranges of $\frac{p}{2}$ through p and exp ranges of $\frac{p}{2}$ through p.

Similarly, a 2x2 round-robin partitioned index from Figure 4.2 (repeated here as Figure A.2) may be summarized as follows.

- I_1 stores tuples with ts ranges of $1, 3, \ldots, p-1$ and exp ranges of $1, 3, \ldots, p-1$.

[1]Assume that there are $\sqrt{n}$ upper-level and $\sqrt{n}$ lower-level partitions, as proven to be optimal in Section 4.3.2.
[2]Assume that the distribution of lifetime lengths is uniform.
[3]Assume that the rate of insertion into the index is constant.

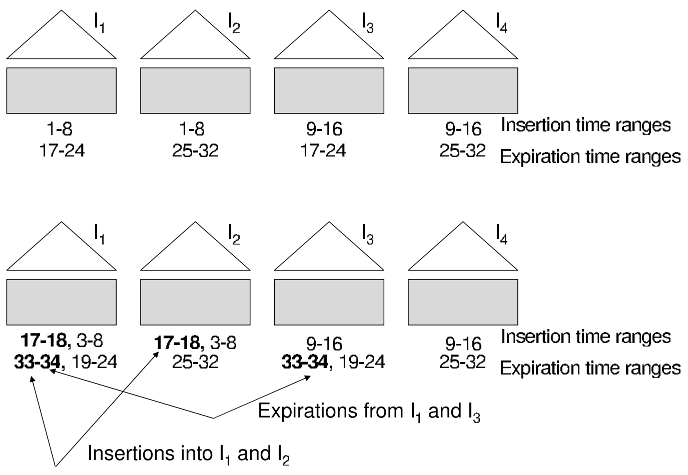


Figure A.1: Example of a doubly partitioned index, showing an update at time 18 (bottom)

- I_2 stores tuples with ts ranges of $1, 3, \ldots, p-1$ and exp ranges of $2, 4, \ldots, p$.
- I_3 stores tuples with ts ranges of $2, 4, \ldots, p$ and exp ranges of $1, 3, \ldots, p-1$.
- I_4 stores tuples with ts ranges of $2, 4, \ldots, p$ and exp ranges of $2, 4, \ldots, p$.

Define the size of a sub-index as the expected number of tuples that it stores. Assume that one unit of size corresponds to the number tuples with some ts range i and some exp range j, such that $1 \leq i, j \leq p$ and $i \geq j$. Due to the assumption of uniform result generation rate and uniform distribution of tuple lifetimes, all such range pairs contain the same number of tuples. For instance, the number of tuples with $ts = 1$ and $exp = 2$ is assumed to be the same as the number of tuples for which $ts = 5$ and $exp = 7$. Note that the expected number of tuples in the entire index is $1+2+\ldots+p$ (because there is one possible exp range for $ts = 1$, two possible exp ranges for $ts = 2$, and so on), which is $\frac{p(p+1)}{2}$. The following two lemmas will be used to prove Theorem 4.1.

Lemma A.1 *Let var_{Ch} be the average variance of the sub-index sizes in chronological partitioning. Then $var_{Ch} = \frac{p^2}{12\sqrt{n}} - \frac{p^2}{4n} + \frac{p^4}{12n} - \frac{p^4}{12n\sqrt{n}} + \frac{p^2}{6}$.*

Proof. Generalizing the discussion of Figure A.1, a $\sqrt{n}$-by-$\sqrt{n}$ chronologically partitioned index may be summarized as follows.

- I_1 stores tuples with ts and exp ranges of one through $\frac{p}{\sqrt{n}}$. Tuples with $ts = 1$ can only have $exp = 1$, therefore there is one unit of them. Tuples with $ts = 2$ can have $exp = 1$ or $exp = 2$, therefore there are two units of results with $ts = 2$. Continuing to $\frac{p}{\sqrt{n}}$, there are $\frac{p}{\sqrt{n}}$ units of tuples with $ts = \frac{p}{\sqrt{n}}$. The total size of I_1 is therefore $1+2+\ldots+\frac{p}{\sqrt{n}} = \frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$.

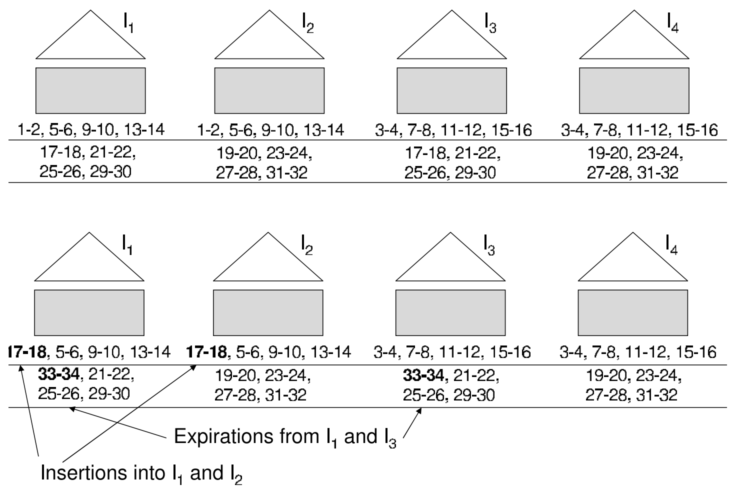


Figure A.2: Example of a round-robin doubly partitioned index, showing an update at time 18 (bottom)

- I_2 stores tuples with ts ranges of one through $\frac{p}{\sqrt{n}}$ and exp ranges of $\frac{p}{\sqrt{n}}+1$ through $\frac{2p}{\sqrt{n}}$. Since the exp time ranges are all greater than the ts time ranges, a tuple would have to have a lifetime larger than the maximum lifetime S in order to reside in this sub-index, and therefore I_2 is empty. Similarly, each of I_3 through $I_{\sqrt{n}}$ are empty.
- $I_{\sqrt{n}+1}$, which is the first sub-index in the second insertion-time partition, stores tuples with ts ranges of $\frac{p}{\sqrt{n}}$ through $\frac{2p}{\sqrt{n}}$ and exp ranges of one through $\frac{p}{\sqrt{n}}$. Tuples with $ts = \frac{p}{\sqrt{n}}+1$ can have $exp = 1$ up to $exp = \frac{p}{\sqrt{n}}$, therefore there are $\frac{p}{\sqrt{n}}$ units of them. The situation is similar for tuples with all the other ts values up to $\frac{2p}{\sqrt{n}}$, therefore the size of $I_{\sqrt{n}+1}$ is $\frac{p}{\sqrt{n}}\frac{p}{\sqrt{n}} = \frac{p^2}{n}$.
- $I_{\sqrt{n}+2}$ stores tuples with ts and exp ranges of $\frac{p}{\sqrt{n}}$ through $\frac{2p}{\sqrt{n}}$. Given that its ts and exp ranges are the same, the size calculation is analogous to I_1 and therefore the size of $I_{\sqrt{n}+2}$ is $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$.
- $I_{\sqrt{n}+3}$ through $I_{2\sqrt{n}}$ are empty because their exp time ranges are greater than their ts time ranges, similar to I_2.
- In terms of the third insertion-time partition ($I_{2\sqrt{n}+1}$ through $I_{3\sqrt{n}}$), similar analysis can show that $I_{2\sqrt{n}+1}$ and $I_{2\sqrt{n}+2}$ have size $\frac{p^2}{n}$, $I_{2\sqrt{n}+3}$ has size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$, whereas $I_{2\sqrt{n}+4}$ through $I_{3\sqrt{n}}$ are empty.
- Continuing to the last insertion-time partition ($I_{n-\sqrt{n}+1}$ through I_n), all but I_n have size $\frac{p^2}{n}$, whereas I_n has size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$.

Summarizing the above analysis, the initial sub-index sizes of a $\sqrt{n}$-by-$\sqrt{n}$ chronologically partitioned index are as follows.

- There are $\sqrt{n}$ sub-indices whose sizes are $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$.
- Of the remaining $n-\sqrt{n}$ sub-indices, half begin with size zero and half begin with size $\frac{p^2}{n}$.

Now consider the next $\frac{p}{\sqrt{n}}$ index updates. For each such update, I_1 through $I_{\sqrt{n}}$ incur insertions, I_1, $I_{\sqrt{n}+1}$, $\ldots$, $I_{n-\sqrt{n}+1}$ incur deletions, and the remaining sub-indices do not change in size. Using the same procedure as above, it can be shown that at the end of the $\frac{p}{\sqrt{n}}$ updates, the size of I_1 is the same (its *ts* range is always equivalent to its *exp* range), the size of the indices which were only inserted into grows from zero to $\frac{p^2}{n}$ (their *ts* ranges are now younger than their *exp* ranges), whereas the size of the indices which were only deleted from shrinks from $\frac{p^2}{n}$ down to zero (their *exp* ranges are now larger than their *ts* ranges).

After that, during the next $\frac{p}{\sqrt{n}}$ updates, the process repeats, with the exception that $I_{\sqrt{n}+1}$ through $I_{2\sqrt{n}}$ incur insertions, I_2, $I_{\sqrt{n}+2}$, $\ldots$, $I_{n-\sqrt{n}+2}$ incur deletions, and the remaining sub-indices are unchanged. Again, all the sub-indices which now only incur insertions begin with size $\frac{p^2}{n}$ and drop down to zero. Furthermore, all the sub-indices which now only incur insertions begin empty and grow to size $\frac{p^2}{n}$. Sub-index $I_{\sqrt{n}+2}$ begins with size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$, is inserted into and deleted from, and does not change in size.

Continuing this analysis, a repeating sequence of $\frac{p}{\sqrt{n}}$ steps emerges, during which the sub-indices evolve in size in the same way (though the set of indices which are inserted from and deleted to changes). In particular, there are $\sqrt{n}$ sub-indices whose sizes are $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$. Moreover, of the remaining $n-\sqrt{n}$ sub-indices, half begin with size zero and half begin with size $\frac{p^2}{n}$. After each update, the former grow by $\frac{p}{\sqrt{n}}$ and the latter decrease by $\frac{p}{\sqrt{n}}$.

Now, to obtain var_{Ch}, it suffices to add the variance of the $\sqrt{n}$ sub-indices whose size does not change, and average the possible sizes of the remaining sub-indices. Recall that the entire index is expected to have size $\frac{p(p+1)}{2}$. Thus, the mean sub-index size is $\mu = \frac{p(p+1)}{2n}$. This gives:

$$\begin{aligned} var_{Ch} &= \sqrt{n}\left[\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}-\mu\right]^2 + \frac{\sqrt{n}}{p}\left[\left(\frac{n-\sqrt{n}}{2}\right)\sum_{i=0}^{\frac{p}{\sqrt{n}}-1}\left((\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}-i)-\mu)^2+(\frac{ip}{\sqrt{n}}-\mu)^2\right)\right] \\ &= \frac{p^2}{12\sqrt{n}}-\frac{p^2}{4n}+\frac{p^4}{12n}-\frac{p^4}{12n\sqrt{n}}+\frac{p^2}{6} \end{aligned}$$

□

Lemma A.2 *Let var_{RR} be the average variance of the sub-index sizes in round-robin partitioning. Then $var_{RR} = \frac{p^2}{2}(\frac{1}{2}+\frac{1}{2n}+\frac{1}{n\sqrt{n}})$.*

Proof. Generalizing the discussion of Figure A.2, a $\sqrt{n}$-by-$\sqrt{n}$ round-robin index may be summarized as follows.

- I_1 stores tuples with ts and exp ranges of $1, \sqrt{n}+1, \ldots, n-\sqrt{n}+1$. Tuples with $ts = 1$ can only have $exp = 1$, therefore there is one unit of them. Tuples with $ts = \sqrt{n}+1$ can have $exp = 1$ or $exp = \sqrt{n}+1$, therefore there are two units of them. Continuing to $n-\sqrt{n}+1$, the possible exp values are $\sqrt{n}+1, \ldots, n-\sqrt{n}+1$, therefore there are $\frac{p}{\sqrt{n}}$ units of tuples with $ts = n-\sqrt{n}+1$. The total size of I_1 is therefore $1+2+\ldots+\frac{p}{\sqrt{n}} = \frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$.
- I_2 stores tuples with ts ranges of $1, \sqrt{n}+1, \ldots, n-\sqrt{n}+1$ and exp ranges of $2, \sqrt{n}+2, \ldots, n-\sqrt{n}+2$. There are no tuples with $ts = 1$ (they can only have $exp = 1$, meaning that they would in fact be stored in I_1). Tuples with $ts = \sqrt{n}+1$ can only have $exp = 2$, therefore there is one unit of them. Tuples with $ts = 2\sqrt{n}+1$ can have $exp = 2$ or $exp = \sqrt{n}+2$, therefore there are two units of them. Continuing to $ts = n-\sqrt{n}+1$, the possible ts values are $2, \sqrt{n}+2, \ldots, n-2\sqrt{n}+2$, therefore there are $\frac{p}{\sqrt{n}}-1$ units of tuples with $ts = n-\sqrt{n}+1$. The total size of I_2 is $1+2+\ldots+\frac{p}{\sqrt{n}}-1 = \frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$.
- The analysis of I_3 through $I_{\sqrt{n}}$ is similar to I_2; they all have size $\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$.
- In terms of the second insertion-time partition ($I_{\sqrt{n}+1}$ through $I_{2\sqrt{n}}$), $I_{\sqrt{n}+1}$ and $I_{\sqrt{n}+2}$ have size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$, whereas the other sub-indices have size $\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$.
- In terms of the third insertion-time partition ($I_{2\sqrt{n}+1}$ through $I_{3\sqrt{n}}$), $I_{2\sqrt{n}+1}$, $I_{2\sqrt{n}+2}$, and $I_{2\sqrt{n}+3}$ have size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$, whereas the other sub-indices have size $\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$.
- Continuing to the last insertion-time partition ($I_{n-\sqrt{n}+1}$ through I_n), all sub-indices have size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$.

In summary, a round-robin index begins with $\frac{\sqrt{n}(\sqrt{n}+1)}{2}$ sub-indices having size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$ and the remaining $n-\frac{\sqrt{n}(\sqrt{n}+1)}{2}$ sub-indices having size $\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$. From then on, every update accesses a different set of sub-indices, but it can be shown that the $\sqrt{n}-1$ sub-indices that are only inserted into grow in size from $\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$ to $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$, whereas the $\sqrt{n}-1$ sub-indices that only incur deletions shrink from size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$ to size $\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$ (and the single index that incurs insertions and deletions does not change in size). Therefore, a round-robin index goes through a repeating sequence of two states, in which either the initial size distribution is true or $\frac{\sqrt{n}(\sqrt{n}+1)}{2}-(\sqrt{n}-1) = \frac{\sqrt{n}(\sqrt{n}-1)}{2}+1$ sub-indices have size $\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}$ and the remaining sub-indices have size $\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}$. Consequently, the average variance of sub-index sizes is as follows (recall that $\mu = \frac{p(p+1)}{2n}$ is the mean sub-index size).

$$\begin{aligned}
var_{RR} &= \frac{1}{2}\left[\frac{\sqrt{n}(\sqrt{n}+1)}{2}\left[\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}-\mu\right]^2+(n-\frac{\sqrt{n}(\sqrt{n}+1)}{2})\left[\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}-\mu\right]^2\right] \\
&+ \frac{1}{2}\left[(\frac{\sqrt{n}(\sqrt{n}-1)}{2}+1)\left[\frac{\frac{p}{\sqrt{n}}(\frac{p}{\sqrt{n}}+1)}{2}-\mu\right]^2+(n-(\frac{\sqrt{n}(\sqrt{n}-1)}{2}+1))\left[\frac{(\frac{p}{\sqrt{n}}-1)\frac{p}{\sqrt{n}}}{2}-\mu\right]^2\right] \\
&= \frac{p^2}{2}(\frac{1}{2}+\frac{1}{2n}+\frac{1}{n\sqrt{n}})
\end{aligned}$$

□

Theorem A.1 (Theorem 4.1 from Section 4.3) *The average variance of sub-index sizes using round-robin partitioning is lower than the average variance of sub-index sizes using chronological partitioning.*

Proof. Let $d(n,p) = var_{Chr} - var_{RR}$. By Lemma 1 and Lemma 2:

$$\begin{aligned}
d(n,p) &= \frac{p^2}{12\sqrt{n}}-\frac{p^2}{4n}+\frac{p^4}{12n}-\frac{p^4}{12n\sqrt{n}}+\frac{p^2}{6}-\frac{p^2}{2}(\frac{1}{2}+\frac{1}{2n}+\frac{1}{n\sqrt{n}}) \\
&= \frac{p^2}{12\sqrt{n}}-\frac{p^2}{2n}+\frac{p^4}{12n}-\frac{p^4}{12n\sqrt{n}}-\frac{p^2}{12}+\frac{p^2}{2n\sqrt{n}}
\end{aligned}$$

By taking partial derivatives of d with respect to n and p and setting them to zero, it can be shown that d is positive for all values of n and p considered here (i.e., $n \geq 4$ since a two level-index has at least two sub-indices and $p \geq 2\sqrt{n}$, as explained earlier). Therefore, the difference in the average sub-index variance of chronological partitioning versus round-robin partitioning is always positive.

□