CONTINUOUS QUERIES OVER DATA STREAMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Arvind Arasu
February 2006
I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

____________________________
Jennifer Widom  Principal Adviser

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

____________________________
Hector Garcia-Molina

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

____________________________
David Maier
(Portland State University)

Approved for the University Committee on Graduate Studies.
Continuous queries (CQs) represent a new paradigm for interacting with dynamically-changing data. Unlike traditional one-time queries, a CQ is registered with a data management system and provides continuous results as data and updates stream into the system. Applications include tracking real-time trends in stock market data, monitoring the health of a computer network, and online processing of sensor data.

This thesis addresses several fundamental challenges in building a system for processing declaratively-specified continuous queries. We first present a new language—an intuitive and natural extension of a traditional database query language—for specifying CQs. The language has been implemented in a comprehensive, publicly-available research prototype called STREAM (for STanford stREam datA Manager). Since CQs are long-running, potentially requiring large amounts of memory, we next present a precise characterization of the amount of memory required for any query in the language. For an important class of queries that require unbounded memory, we describe algorithms that trade off answer accuracy for a lower memory requirement. Finally, we describe techniques for sharing resources such as computation and state across multiple CQs, enabling scalability to a very large number of concurrent CQs.
Acknowledgments

I am extremely grateful to my adviser, Jennifer Widom, for her invaluable guidance at every stage of my PhD. She has spent an incredible amount of time and energy helping me with all aspects of PhD ranging from problem formulation and solving to thesis writing. Her practice of having regular, weekly meetings was particularly helpful in making sure that I made continuous progress and had no long periods of slack. More broadly, my research has benefited immensely from her vast experience and knowledge.

I am also grateful to Hector, who advised me during the first few years of my PhD. His patient guidance was invaluable when I was working on a somewhat open-ended and difficult research problem. I am also thankful to my undergraduate adviser, Pandu Rangan, for his motivation and guidance during my early research days.

I thank Dave, Hector, and Jennifer for kindly serving as readers of this thesis and providing valuable feedback.

I thank all the database group members who overlapped with me, for making the time I spent in the group very memorable and a wonderful learning experience. Special thanks to Gurmeet, for a most enjoyable collaboration, and Shivnath, for being a friend, colleague, and competitor (in a positive, mutually beneficial sense) for over a decade now. I thank Marianne for her administrative help and Andy for patiently handling my computer issues.

On the personal front, I thank Ultra for being a most accommodating apartment mate and a wonderful friend, always ready with his advice and help when I needed them most. Arun and Shankar were great friends, and I have very pleasant memories of weekends spent in their company. Also, many thanks to Anish for the wonderful chess games.

Finally, I would not have reached this point in my academic career without the untiring efforts, support, and love of my mother and father. I dedicate this thesis to them.
Contents

Abstract v

Acknowledgments vi

1 Introduction 1
   1.1 Research Challenges .............................................. 3
   1.2 Contributions ...................................................... 5
   1.3 Related Work ....................................................... 7
   1.4 Thesis Outline ..................................................... 8

2 Continuous Query Language 9
   2.1 Introduction ....................................................... 9
   2.2 Related Work ....................................................... 11
   2.3 Running Example: Linear Road .................................... 12
   2.4 Streams and Relations ............................................ 13
   2.5 Abstract Semantics ................................................ 15
      2.5.1 Formal Specification .......................................... 18
   2.6 CQL ................................................................. 25
      2.6.1 Stream-to-Relation Operators ................................. 26
      2.6.2 Relation-to-Relation Operators ............................... 30
      2.6.3 Relation-to-Stream Operators ................................. 31
      2.6.4 Syntactic Shortcuts and Defaults ............................. 33
      2.6.5 Linear Road in CQL .......................................... 34
      2.6.6 Operations over Timestamps ................................. 37
      2.6.7 Common Constructs .......................................... 38
2.6.8 Equivalences in CQL ........................................ 41
2.6.9 Streams versus Relations .................................. 43
2.6.10 Comparison with Other Languages .......................... 44
2.7 An Algebra of Continuous Operators ............................ 47
  2.7.1 Relation-to-Relation Operators .............................. 48
  2.7.2 Stream-to-Relation and Relation-to-Stream Operators ...... 48
  2.7.3 Shorthand Notation for Relational Operations over Streams 49
2.8 Time Management ............................................. 50
2.9 Conclusion .................................................... 51

3 Memory Requirements of Queries ................................. 53
  3.1 Introduction .................................................. 53
    3.1.1 Examples ................................................. 55
    3.1.2 Contributions ............................................. 57
    3.1.3 Overview and Organization ................................. 58
  3.2 Problem Statement ............................................ 59
  3.3 Notation and Definitions ...................................... 60
  3.4 Bounded-Memory Computability of SPJ Queries ................. 63
    3.4.1 LTO Queries with Duplicate-Preserving Projection ....... 66
    3.4.2 LTO Queries with Duplicate-Eliminating Projection ...... 70
    3.4.3 Only-If Proofs of Theorems 3.2 and 3.3 .................. 73
    3.4.4 Summary ................................................ 79
  3.5 Checking Algorithm ........................................... 79
  3.6 Query Evaluation Strategy .................................... 82
    3.6.1 Evaluation Strategy for Duplicate-Preserving Queries .. 83
    3.6.2 Evaluation Strategy for Duplicate-Eliminating Queries .. 86
  3.7 Extensions .................................................. 91
    3.7.1 Queries with Self-Joins .................................. 92
    3.7.2 Queries with Grouping and Aggregation .................. 93
    3.7.3 Queries with Sliding Windows ............................ 99
  3.8 Conclusion ................................................ 101
# 4 Approximate Statistics over Streams

## 4.1 Introduction

## 4.2 Definitions

### 4.2.1 Quantiles

### 4.2.2 Frequent Elements

## 4.3 Problem Statement

### 4.3.1 Sliding Windows

### 4.3.2 Problem Statement

## 4.4 Sketches

### 4.4.1 Static Sketches

### 4.4.2 Sliding-Window Sketches

## 4.5 Deterministic Bounded-Window Sketches

### 4.5.1 Details of $C_{W, \epsilon}$

### 4.5.2 Details of $Q_{W, \epsilon}$

## 4.6 Randomized, Bounded-Window Sketches

## 4.7 Unbounded-Window Sketches

### 4.7.1 General Technique

### 4.7.2 Specific Constructions

## 4.8 Related Work

## 4.9 Conclusion

# 5 Resource Sharing

## 5.1 Introduction

## 5.2 Preliminaries

## 5.3 Space-Update-Lookup Tradeoff and Our Approach

## 5.4 ASW Queries

### 5.4.1 Distributive and Algebraic Aggregates

### 5.4.2 Subtractable Aggregates

### 5.4.3 Quantiles

### 5.4.4 Extensions for Time-Based Windows

## 5.5 SSF Queries

### 5.5.1 CI-COUNT

### 5.5.2 CI-SUM
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Description of the terms used in the abstract syntax of Figure 2.4.</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Relation-to-relation operators in ACO</td>
<td>47</td>
</tr>
<tr>
<td>2.3</td>
<td>Stream-to-relation and relation-to-stream operators in ACO</td>
<td>48</td>
</tr>
<tr>
<td>2.4</td>
<td>Shortcuts for relational operators over streams.</td>
<td>49</td>
</tr>
<tr>
<td>3.1</td>
<td>Bounded-memory computability of example queries over data streams S(A, B, C) and T(D, E).</td>
<td>56</td>
</tr>
<tr>
<td>3.2</td>
<td>Notation related to a query Q and stream S</td>
<td>61</td>
</tr>
<tr>
<td>4.1</td>
<td>List of symbols used to describe $C_{W, \epsilon}$ and $Q_{W,\epsilon}$</td>
<td>117</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of results. $N_{max} =$ Max left end of a window; $W =$ width of lookup window; $</td>
<td>K</td>
</tr>
<tr>
<td>6.1</td>
<td>Operators used in STREAM query plans</td>
<td>167</td>
</tr>
<tr>
<td>6.2</td>
<td>Synopses used in STREAM</td>
<td>169</td>
</tr>
<tr>
<td>6.3</td>
<td>Queries used in STREAM performance measurements</td>
<td>175</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance of STREAM for representative queries</td>
<td>177</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 The Linear Road highway system ........................................... 12
2.2 Operator classes used in abstract semantics ............................. 16
2.3 Output produced by operators of Example 2.3 as time progresses. ..... 18
2.4 An abstract syntax for continuous queries. .............................. 19
2.5 Meaning functions in the denotational semantics ....................... 21
2.6 Abstract syntax for example operators ................................. 23
2.7 Syntax for specifying the time-interval in a time-based sliding window . . 27
2.8 Derived relations and streams for Linear Road queries ................. 35
3.1 \textit{MaxRef} and \textit{MinRef} for Example 3.1 ............................. 66
3.2 Algorithm to check bounded memory computability of SPJ Queries. ..... 80
3.3 Example illustrating checking algorithm of Figure 3.2. Sub-figure (a) shows an input query \( Q \), (b) shows an LTO query \( Q_L \) derived from \( Q \), (c) shows a query \( Q' \) constructed from \( Q \) is Step 4 of our algorithm, and (d) shows an LTO query \( Q'_L \) derived from \( Q' \). ................................................. 82
4.1 Bag of elements used in Example 4.3 ....................................... 106
4.2 Example fixed-size and variable-size windows .......................... 109
4.3 Levels and blocks used in \( C_{W,\epsilon} \) and \( Q_{W,\epsilon} \) ...................... 116
4.4 Construction of \( C_{2^k+1,\epsilon}(m,2^k) \) from \( C_{2^k,\epsilon}(m,2^k) \) ............ 126
5.1 Base-intervals used by B-INT when current length of \( S \) is \( m \) .......... 141
5.2 Algorithm B-INT ............................................................. 142
5.3 Landmark intervals that begin or end at the landmark \( \alpha W_{\text{min}} \) for some \( \alpha \geq 0 \) 144
5.4 Algorithm L-INT for \( W_{\text{max}}/W_{\text{min}} \leq 2 \) ................................. 145
5.5 Algorithm R-INT ............................................................. 147
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Algorithm CI-COUNT</td>
<td>151</td>
</tr>
<tr>
<td>5.7</td>
<td>Timestamps (within boxes) and positions (above boxes) of elements belonging to three substreams.</td>
<td>152</td>
</tr>
<tr>
<td>5.8</td>
<td>(a)-(c) Elements of three substreams. Elements are shown as vertical lines with height indicating the value of the element. Horizontal lines indicate answer intervals of suffix-max elements. (d) Mapping lookup to the line-segment-intersection problem.</td>
<td>155</td>
</tr>
<tr>
<td>5.9</td>
<td>Algorithm CI-MAX</td>
<td>156</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparing R-INT against the naive alternatives</td>
<td>159</td>
</tr>
<tr>
<td>5.11</td>
<td>Stream update rates for the ASW algorithms</td>
<td>160</td>
</tr>
<tr>
<td>5.12</td>
<td>Maximum lookup rates for the ASW algorithms</td>
<td>161</td>
</tr>
<tr>
<td>5.13</td>
<td>Performance of CI-COUNT for various values of $p$. The naive approach processes 1000 inputs/second.</td>
<td>162</td>
</tr>
<tr>
<td>6.1</td>
<td>Plan for the query in Section 6.2.4</td>
<td>166</td>
</tr>
<tr>
<td>6.2</td>
<td>STREAM Visualizer</td>
<td>173</td>
</tr>
<tr>
<td>6.3</td>
<td>The M-Pattern Query ($Q_5$) in CQL [42].</td>
<td>176</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This thesis studies data processing issues for an emerging, important class of applications characterized by continuous processing of dynamic data in a form called data streams. Traditional database management systems do not have the right primitives or overall design to handle the data processing requirements of these applications. The basic motivation of the work presented here stems from these limitations of traditional data management systems.

Continuous Processing

Continuous processing refers to processing that goes on indefinitely, without a well-defined termination. For example, the operating system on a computer is a continuous process, since it executes in an “infinite loop” managing the computer’s resources. On the other hand, an instance of a sorting algorithm is not a continuous process, since it terminates once it produces the sorted output.

There are numerous examples of data processing applications that involve continuous processing: Network monitoring applications continuously analyze network traffic to detect performance problems or malicious activity [37, 64, 86, 88]; financial applications track stock ticker data in real time to identify interesting trends [69, 109]; telecom fraud detection involves constantly monitoring call-record data to detect fraudulent activity. More generally, such applications arise naturally whenever some dynamic system (e.g., computer network, stock exchange) needs to be monitored continuously.
Data Streams

In all the applications above, a natural way to model the data is as a data stream (or simply a stream). Informally, a stream has the following defining characteristics:

- It is a potentially unbounded, continuously arriving sequence of data units called stream elements. Stream elements are usually relational tuples, but could also be less structured, such as XML documents, or more complex objects. By continuously arriving we mean that the stream is not intended to terminate after a specific amount of time, and new elements of the stream keep appearing as time progresses.

- The stream elements are pushed by the source of the stream, and an application processing the stream has no control over the arrival of stream elements. Specifically, the arrival rate of stream elements and their ordering could be unpredictable and change over time.

- Stream elements are accessed sequentially in the order in which they arrive. To revisit an element that arrived in the past, it must be stored explicitly.

In the network-monitoring example, the network traffic data can be modeled as a stream, with one stream element corresponding to each network packet. Each stream element has packet-related information such as the packet source, destination, length, and content. We can similarly model the stock ticker data and the call record data from the other examples above as data streams.

Thesis: Motivation and Summary

Most of the applications that perform continuous processing over data streams are implemented today using custom code, scripts, or other ad hoc tools. This approach has well-known drawbacks: The applications tend to be brittle; changing the application logic is hard; and even minor changes could potentially require extensive testing and debugging.

The creation and management of these applications can be simplified greatly if they are implemented over a data processing system that supports a database-style, declarative query interface. Application logic can then be expressed as concise queries over the system, and changes to the application can be made by simply changing the queries.

Traditional database management systems (DBMSs), however, are ill-suited for this purpose. They lack the primitives to express continuous processing logic. Queries in
DBMSs are meant for one-time processing: they operate on a static snapshot of data and terminate once they produce their output. Also, DBMSs are not designed to handle rapid data inserts, and this feature is crucial to handling rapid data streams, with or without continuous processing.

As part of this thesis research, we have designed and developed a general purpose, declarative, data management system called STREAM (STanford stREam datA Manager), that specifically addresses the needs of these applications. STREAM differs from traditional DBMSs in two ways: It supports a new class of queries called continuous queries, and it is designed to process streams very efficiently. Continuous queries (also known as standing queries) are long-running queries that operate on dynamic input and produce continuous output as new input arrives. They are naturally suited to express continuous processing operations.

The general class of systems like STREAM that support continuous queries over streams are called Data Stream Management Systems (DSMSs). There are a few other research DSMS prototypes; most of them were developed around the same time as STREAM—we discuss these systems in detail in Section 1.3.

1.1 Research Challenges

This section lists the broad research challenges that arise when building a DSMS. The specific contributions of this thesis are summarized in Section 1.2, and other work that has addressed these challenges is described in Section 1.3.

Language and Semantics

Designing a query language and associated semantics for continuous queries involves unique challenges not seen for traditional one-time queries: First, since, by definition, continuous queries produce output spread over time, the semantics for continuous queries should specify not only what the output is that corresponds to a given input, but also when that output is produced. This requirement also means that the semantics need to have a built-in notion of time and its progress; from temporal databases we know that incorporating the notion of time could be intricate. Second, the query language should include a new data type, stream, and support operations over it.
Unknown Environment

Traditional DBMSs operate in a relatively known environment. For example, they have complete control over how relations are physically stored, they have detailed meta-information about relations in the form of statistics, and they determine how and when relations are accessed during query processing.

In contrast, a data stream management system has little information about its inputs. A stream has fundamentally unpredictable dynamics. For example, the rate or content of a stream could change over time. This lack of information has at least two fundamental implications: First, the traditional approach to query processing—planning followed by execution—does not work, since planning requires advance knowledge of the factors affecting query execution. Second, the DSMS needs to be designed to handle unpredictable overloading—for example, a sudden increase in input stream rate—gracefully.

Resource Management

Resource management is an integral part of any complex computing system. For example, DBMSs have to apportion CPU, memory, and disk bandwidth among competing, concurrent queries. Again, the long-running nature of continuous queries introduces new challenges: Queries hold on to resources longer, and their resource requirements could change over time. These challenges make resource allocation harder, and necessitates a more elastic resource management subsystem to handle the changing requirements. Also, due to their greater temporal overlap, continuous queries present more opportunities for resource sharing than traditional queries. For example, two or more queries could share the same state (memory) or could benefit from some common computation. Identifying these resource sharing opportunities and exploiting them are interesting research problems.

Stream Data Model

Even without continuous queries, the stream data model introduces new algorithmic challenges. General algorithms are designed assuming random access to their input. By contrast, algorithms processing data streams access their input sequentially. They can use some scratch space (memory) to store state regarding previously seen stream elements. But this space is finite, while the stream itself is unbounded.

Two interesting classes of problems arise in this context: The first class, which is mostly
of theoretical interest, seeks to characterize what kinds of data processing is feasible in the stream data model. Many natural problems (e.g., computing a median\(^1\)) cannot be solved in this model. The second class explores techniques for trading output accuracy for consuming less scratch space. This class of problems has important practical applications since approximate output turns out to be good enough for many applications.

1.2 Contributions

This section highlights the main contributions of this thesis, and places them in the context of the research challenges presented in the previous section.

Query Language and Semantics

The thesis presents CQL (for Continuous Query Language) [9, 10], one of the first query languages for continuous queries. CQL is an expressive SQL-based declarative language, and it handles both streams and relations uniformly. As we will argue in Chapter 2, supporting both streams and relations is important for many applications. CQL is based on an underlying abstract semantics for continuous queries that relies only on “black-box” classes of operators over streams and relations. CQL is one particular instantiation of these classes, but others are possible. This thesis also includes a formal denotational specification of the abstract semantics [13]. Most of CQL has been implemented in the STREAM prototype.

Memory Requirements of Queries

Since streams are unbounded, and memory is not, memory is an important resource in a data stream processing system. Our next contribution [7, 8] provides a precise, asymptotic characterization of the memory required by continuous queries expressed in CQL or a similar language. This work presents two interesting results: First, it proves that all queries fall into just two classes based on their asymptotic memory requirements—queries that require a bounded amount of memory, and those that require memory that grows linearly with the input. Second, it shows that the class of queries that can be computed in bounded memory is nontrivial, even containing some multi-join queries.

\(^1\)Specifically, we cannot compute the median of \(N\) values in one pass over the values using \(o(N)\) scratch space [76].
CHAPTER 1. INTRODUCTION

Approximate Statistics over Stream Sliding Windows

Since computing exact answers to queries can require prohibitively large amounts of memory, there is a need for techniques that compute approximate answers using less memory. This thesis contains new contributions in this area [12]: We present algorithms for computing approximate quantiles and frequency counts over stream sliding windows that significantly outperform previous algorithms. Quantiles and frequency counts are two commonly used statistics [41, 76], while a sliding window is a near-universal operation over streams [17]. For most applications, the recent elements of a data stream are more important than older ones, and a sliding window captures this preference.

Resource Sharing

The thesis next considers the resource sharing problem mentioned in Section 1.1. Resource sharing is both possible and important when there are a large number of similar, concurrent queries. This scenario occurs commonly in subscription-based applications, such as Traderbot [109], that allow users to independently monitor data of interest. This work again focuses on aggregates over stream sliding windows. We present new techniques for a wide range of possible scenarios: different classes of aggregation functions (algebraic, distributive, holistic), and different types of sliding windows and streams [14]. All of our techniques have precise theoretical guarantees and show very good empirical performance.

System Architecture

Finally, the thesis describes the architecture of the STREAM prototype. As mentioned earlier, STREAM is a general purpose, declarative data stream management system developed at Stanford [101]. STREAM allows users and applications to register continuous queries specified in CQL, and execute them over input streams and relations. STREAM has a sophisticated graphical user interface (GUI) to interact with the execution engine. Apart from basic functionality of registering queries and viewing results, the GUI lets users monitor the performance of the system, while it is executing queries. STREAM has been released as public software [102].

STREAM is a joint effort of several members of the STREAM project.
1.3 Related Work

In this section, we describe work that is broadly related to the entire thesis. More detailed discussion of work related to specific technical contributions appears in the relevant chapters.

STREAM Project

This thesis work was done as part of the STREAM project [101], so most other work in the project is closely related. Initial motivation for the project is discussed in references [17, 25], and a broad overview of the goals and directions is presented in references [6, 80]. Among more specific work, Babu et al. [22, 23] propose a new adaptive approach for continuous query processing and optimization, radically different from the traditional approaches; Babcock et al. [16] present optimal operator scheduling algorithms for a STREAM-like architecture; Babcock et al. [19] study load shedding strategies for handling overloading in a DSMS, and Srivastava et al. [98] explore time-related issues.

Other Data Stream Management Systems

Several other data stream management systems have appeared in recent times. While the overall goals of these systems are similar to that of STREAM, they differ significantly in the design details. Aurora [28] was developed at jointly Brandeis, Brown, and MIT. Unlike STREAM, Aurora does not support declarative queries. Instead, an application administrator manually creates a network of operators to implement application logic. For more details about Aurora operators, please refer to Chapter 2. Another closely related system is TelegraphCQ [29], developed at UC Berkeley. TelegraphCQ and STREAM differ mainly in their query processing strategies: Query processing in STREAM is based on a traditional tree-of-operators approach, while in TelegraphCQ, it is based on the eddy operator [15]. NiagaraCQ [32] is a system for continuous monitoring of persistent data sets spread over a wide-area network, e.g., web sites over the Internet. The focus of NiagaraCQ is on continuous queries, but not on data streams. However, a “branch” of the Niagara project called NiagaraST deals with data streams [70, 84], and is closely related to STREAM. Gigascope [37] and Tribeca [105] are two systems designed specifically to process network packet streams. Nile [63] and CAPE [94] are two recent systems with similar goals and approaches as STREAM.
Algorithmic Aspects

A large body of work in data stream processing focuses on algorithmic aspects of the stream model. The most seminal work in this category is the paper by Alon, Matias, and Szegedy [4], which won the 2005 Godel prize for contributions to Theoretical Computer Science [55]. That paper presents algorithms for computing various frequency moments over a data stream. Other important work in this area include techniques for maintaining a random sample over a stream [112], techniques for constructing various kinds of histograms [53, 58], and algorithms for computing frequent elements in a stream [41, 67, 78]. Datar et al. [39] introduce the problem of computing statistics over stream sliding windows, and present several results; this work is discussed in greater detail in Chapter 4.

1.4 Thesis Outline

The organization of the rest of the thesis closely follows the listing of the thesis contributions in Section 1.2. Chapter 2 presents CQL and its semantics. Chapter 3 characterizes memory requirements of queries. Chapter 4 presents our results on computing approximate statistics over stream sliding windows. Chapter 5 discusses resource sharing issues. Chapter 6 describes the architecture of the STREAM prototype. Conclusions and directions for future work are presented in Chapter 7.
Chapter 2

Continuous Query Language

In this chapter, we define data streams formally, extend the definition of traditional relations for continuous queries, and present CQL (for Continuous Query Language), the declarative query language used in STREAM. We also present an algebra of continuous operators that we use to present Chapters 3-5. Work presented in this chapters appeared in references [9, 10, 13].

2.1 Introduction

Semantics of continuous queries can be thought of as consisting two parts: a data processing part that determines how output data is produced from input data, and a timing-related part that determines which part of the output is produced at what time. For example, for the semantics of “joining” two streams, the data processing part might determine that each output element is produced by combining two input elements that satisfy a join condition, while the timing part might determine that an output element is produced when both the joining input elements have arrived.

We use this conceptual division of continuous query semantics to structure our language design. We first develop an abstract semantics that deals only with continuous, timing-related aspect of queries. The abstract semantics does not contain any specific data processing primitives (operators), but works with generic, “black-box” operators. The only information about the black-box operators used is their input and output data types, which include streams and relations.1

1We use the term “operator” with two slightly different meanings. In this chapter, an “operator” means
We can derive a concrete continuous query language by using specific operators in place of the black-box operators of our abstract semantics. One such derivation is CQL: Most of the operators of CQL are based on SQL queries; in addition, there are a few windowing operators, and some special operators that act on a relation to produce a stream. We also present a second language that uses operators from relational algebra instead of SQL queries. The latter language is used for presenting the technical results in Chapters 3-5.

Design Goals

Designing a language presents many choices. We used the following high-level design goals while making our design choices:

1. We wanted to exploit well-understood relational semantics (and by extension relational rewrites and execution strategies) to the extent possible.

2. We wanted queries performing simple tasks to be easy and compact to write. Conversely, we wanted simple-appearing queries to do what one expects.

3. We wanted the language to have sufficient constructs to capture a wide variety of continuous data stream applications, without allowing the “feature creep” that can result in an esoteric, difficult-to-understand, or difficult-to-implement language. That is, we wanted to keep the language as simple as possible without sacrificing too much expressiveness.

We will illustrate how these goals influenced our language design at various relevant points in this chapter.

Chapter Organization

We begin by presenting related work in Section 2.2: we introduce several related languages and reference them at various points in the chapter for comparison. Section 2.3 introduces a detailed running example used throughout the chapter. Section 2.4 presents formal definitions of streams and relations. Section 2.5 presents our abstract semantics for continuous queries. Section 2.6 contains a detailed description of CQL including syntax, semantics, illustrative examples, common constructs, and comparison with other languages.

Any entity that operates over data, as in the sentence “A SQL query is an operator over traditional relations.” In later chapters, we use “operator” to mean a small unit (physical or logical) of data processing, as in the sentence “A SQL query is converted into a tree of operators by the query processor.”
CHAPTER 2. CONTINUOUS QUERY LANGUAGE

present our algebra-based continuous query language in Section 2.7. Section 2.8 discusses time management issues that arise in a DSMS implementing CQL.

2.2 Related Work

Continuous queries have been used either explicitly or implicitly for quite some time. Materialized views [61] are a form of continuous query, since a view is continuously updated to reflect changes to its base relations. Reference [66] extends materialized views to include chronicles, which essentially are continuous data streams. Operators are defined over chronicles and relations to produce other chronicles, and also to transform chronicles to materialized views. The operators are constrained to ensure that the resulting materialized views can be maintained incrementally without referencing entire chronicle histories.

Continuous queries were introduced explicitly for the first time in Tapestry [107] with a SQL-based language called TQL. (Barbara [26] considers a similar language.) Conceptually, a TQL query is executed once every time instant as a one-time SQL query over the snapshot of the database at that instant, and the results of all the one-time queries are merged using set union. Several systems [32, 73, 83] use continuous queries for information dissemination. The semantics of continuous queries in these systems is also based on periodic execution of one-time queries as in Tapestry.

The query language used in the TelegraphCQ system [29] is very similar to CQL, and was developed around the same time as CQL. One important difference is that the TelegraphCQ query language supports only streams, while CQL supports both streams and relations. We discuss this difference in more detail in Section 2.6.9. As we indicated in Chapter 1, Aurora [28] does not support a declarative query language; instead, users create a network of operators. Also, some Aurora operators are procedural, i.e., they contain user-defined code. In contrast, CQL in its present form is purely declarative. The GSQL [37] query language used in Gigascope and the query language used in Tribeca [105] focus on network monitoring and are closely related to CQL. We describe important differences between CQL and these languages in Section 2.6.10. ATLas [68, 113] proposes simple extensions to SQL-99 user-defined aggregates (UDAs) that make the resulting language Turing-complete, suitable for various data-mining and data streams applications. Intuitively, the extensions let users express initialize, iterate, and terminate parts of a SQL-99 UDA specification using SQL update constructs rather than procedural code.
Data stream processing involves temporal and sequential aspects, making CQL related to temporal query languages [85] and sequential query languages [97]. While these languages are more expressive than CQL and contain many special-purpose operators, they are designed for one-time, not continuous, processing. We intentionally refrained from including esoteric operators derived from these query languages to keep CQL simple, as mentioned in goal #3 earlier. Finally, event-processing languages are geared largely toward matching single events or specific event patterns against queries, usually in a publish-subscribe setting [115]. The fine-grained event-matching constructs of these languages were not needed for the stream applications we studied.

### 2.3 Running Example: Linear Road

Our running example is based on a hypothetical road traffic management application introduced in the Linear Road benchmark for data stream management systems [11]. We use a simplified version of the Linear Road application to illustrate various aspects of our language and semantics; full details can be found in the original specification [11].

![Figure 2.1: The Linear Road highway system](image)

The Linear Road application implements variable tolling—adaptive, real-time computation of vehicle tolls based on traffic conditions—to regulate vehicular traffic on a highway. To enable variable tolling, each vehicle is equipped with a sensor that periodically relays its position and speed to a central server. The server aggregates the information received from all vehicles on the highway system, computes tolls in real-time, and transmits tolls back to vehicles using the sensor network.

Figure 2.1 shows the highway system used by our simplified Linear Road. There is a single highway 100 miles long, which is divided into 100 one-mile segments. The highway
has entrance and exit ramps at segment boundaries. Traffic flows in a single direction from left to right. When a vehicle is on the highway, it reports its current speed (miles per hour) and position (number of feet from the left end) to the server once every 30 seconds. (In the complete Linear Road application [11] there are 10 highways with multiple lanes, and traffic flows in both directions.)

Vehicles pay a toll whenever they drive in a congested segment, while no toll is charged for uncongested segments. A segment is congested if the average speed of all vehicles in the segment over the last 5 minutes is less than 40 miles per hour. The toll for a congested segment is given by the formula \(2 \times (\text{numvehicles} - 50)^2\), where \(\text{numvehicles}\) is the number of vehicles currently in the segment. Note that the toll for a congested segment changes dynamically as vehicles enter and leave the segment. When the server detects that a vehicle has entered a congested segment, the server outputs the current toll for the segment, which is reported back to the vehicle.

Using our terminology, the simplified Linear Road application has a single input stream—the stream of positions and speeds of vehicles—a single continuous query computing the tolls, and a single output stream containing the tolls for vehicles.

### 2.4 Streams and Relations

In this section we formally define streams and relations. As in the standard relational model, each stream and relation has a fixed schema consisting of a set of named attributes. For stream element arrivals and relation updates we assume a discrete, ordered time domain \(T\). A \textit{time instant} (or simply \textit{instant}) is any value from \(T\). Further, each instant \(\tau \in T\) has a unique, well-defined \textit{successor}, which is the smallest instant in \(T\) larger than \(\tau\). For concreteness, we represent \(T\) as the nonnegative integers \(\{0, 1, \ldots\}\); in particular note that 0 stands for the earliest time instant. Time domain \(T\) models an application’s notion of time, not particularly system or wall-clock time. Thus, although \(T\) may often be of type \textit{Datetime}, the data type used to represent date and time in standard SQL [40], our semantics only requires any discrete, ordered domain.

**Definition 2.1 (Stream)** A stream \(S\) is a possibly infinite bag of elements \((s, \tau)\), where \(s\) is a tuple belonging to the schema of \(S\) and \(\tau \in T\) is the \textit{timestamp} of the element. We require that there be a finite (but unbounded) number of elements with a given timestamp. \(\square\)
Intuitively, the element \( (s, \tau) \) indicates that tuple \( s \) arrived on stream \( S \) at time \( \tau \). Note that the timestamp is not part of the schema of a stream, and there could be zero, one, or multiple elements with the same timestamp in a stream. There are two classes of streams: base streams, which are the source data streams that arrive at the DSMS, and derived streams, which are intermediate streams produced by operators in a query. We use the term tuple of a stream to denote the data (non-timestamp) portion of a stream element.

**Example 2.1** In the Linear Road application there is just one base stream containing vehicle speed-position measurements with schema:

\[
\text{PosSpeedStr} \ (\text{vehicleId}, \text{speed}, \text{xPos})
\]

Attribute \text{vehicleId} identifies the vehicle, \text{speed} denotes its current speed in MPH, and \text{xPos} denotes its current position within the highway in feet as described in Section 2.3. The time domain is of type \text{Datetime}, and for this application the timestamp of a stream element denotes the physical time when the position and speed measurements were taken.

**Definition 2.2 (Relation)** A relation \( R \) is a mapping from time domain \( T \) to a finite but unbounded bag (multiset) of tuples belonging to the schema of \( R \).

A relation \( R \) defines an unordered bag of tuples at any time instant \( \tau \in T \), denoted \( R(\tau) \). Note the difference between this definition for relation and the standard one: In the standard relational model, a relation is simply a set (or bag) of tuples, with no notion of time as far as the semantics of relational query languages are concerned. We use the term instantaneous relation to denote the bag of tuples in a relation at a given point in time. Thus, if \( R \) denotes a relation according to Definition 2.2, \( R(\tau) \) denotes an instantaneous relation. Often, when there is no ambiguity, we omit the term instantaneous. We use the term base relation for input relations and derived relation for relations produced by query operators.

**Example 2.2** The Linear Road application contains no base relations, but several derived relations are useful for toll computation. For example, the toll for a congested segment depends on the current number of vehicles in the segment. We can represent the current number of vehicles in a segment using the derived relation:

\[
\text{SegVolRel} \ (\text{segNo}, \text{numVehicles})
\]
Attribute segNo denotes the segment (0–99) and numVehicles the number of vehicles in the segment. SegVolRel(τ) contains the count of vehicles in each highway segment as of time τ. Section 2.6.5 shows how SegVolRel can be computed from the base stream PosSpeedStr, and how it can be used to compute tolls.

As this example suggests, the concept of a relation is useful even in applications whose inputs and outputs are all streams. It seems more natural to model “the current number of vehicles in a segment” as a time-varying relation, rather than as a stream of the latest values. From an expressiveness point of view, it is not necessary to have both streams and relations: We could have picked just one of streams and relations and designed our language around it without loss of expressiveness; this issue is discussed further in Section 2.6.9.

2.5 Abstract Semantics

Our abstract semantics is based on three classes of operators over streams and relations:

- **stream-to-relation** operators that produce a relation from a stream
- **relation-to-relation** operators that produce a relation from one or more other relations
- **relation-to-stream** operators that produce a stream from a relation

By “relation” we mean a time-varying relation of Definition 2.2, not a traditional one. Stream-to-stream operators are absent—they have to be composed from operators of the three classes above. As we will discuss in detail in Section 2.6, the rationale for this decision is based primarily on our Goal #1 from Section 2.1: exploiting well-understood relational semantics (and by extension relational rewrites and execution strategies) to the extent possible.

First some terminology: \( S \text{ up to } \tau \) denotes the bag of elements in stream \( S \) with timestamps \( \leq \tau \), i.e., \( \{(s, \tau') \in S : \tau' \leq \tau\} \). \( S \text{ at } \tau \) denotes the bag of elements of \( S \) with timestamp \( \tau \), i.e., \( \{(s, \tau') \in S : \tau' = \tau\} \). Similarly, \( R \text{ up to } \tau \) denotes the collection of instantaneous relations \( R(0), \ldots, R(\tau) \), and \( R \text{ at } \tau \) denotes the instantaneous relation \( R(\tau) \).

1. A **stream-to-relation** operator takes a stream \( S \) as input and produces a relation \( R \) as output with the same schema as \( S \). At any instant \( \tau \), \( R(\tau) \) should be computable from \( S \text{ up to } \tau \).
CHAPTER 2. CONTINUOUS QUERY LANGUAGE

2. A relation-to-relation operator takes one or more relations \( R_1, \ldots, R_n \) as input and produces a relation \( R \) as output. At any instant \( \tau \), \( R(\tau) \) should be computable from \( R_1(\tau), \ldots, R_n(\tau) \).

3. A relation-to-stream operator takes a relation \( R \) as input and produces a stream \( S \) as output with the same schema as \( R \). At any instant \( \tau \), \( S \) at \( \tau \) should be computable from \( R \) up to \( \tau \).

Figure 2.2 schematically illustrates the three operator classes in our abstract semantics. The rationale for constraining stream-to-relation and relation-to-stream operators to be unary is again based on Goal #1—to exploit relational semantics to the extent possible; we elaborate on this design decision in Section 2.6 (under “Overall Approach”). Now we define our abstract semantics.

**Definition 2.3 (Abstract Semantics)** Consider a query \( Q \) that is any type-consistent composition of operators from the above three classes. Suppose the set of all inputs to the innermost (leaf) operators of \( Q \) are streams \( S_1, \ldots, S_n \) \((n \geq 0)\) and relations \( R_1, \ldots, R_m \) \((m \geq 0)\). We define the result of continuous query \( Q \) at a time \( \tau \), which denotes the result of \( Q \) once all inputs up to \( \tau \) are “available” (a notion discussed below). There are two cases:

- Case 1: The outermost (topmost) operator in \( Q \) is relation-to-stream, producing a stream \( S \) (say). The result of \( Q \) at time \( \tau \) is \( S \) up to \( \tau \), produced by recursively applying the operators comprising \( Q \) to streams \( S_1, \ldots, S_n \) up to \( \tau \) and relations \( R_1, \ldots, R_m \) up to \( \tau \).

- Case 2: The outermost (topmost) operator in \( Q \) is stream-to-relation or relation-to-relation, producing a relation \( R \) (say). The result of \( Q \) at time \( \tau \) is \( R(\tau) \), produced by
recursively applying the operators comprising \( Q \) to streams \( S_1, \ldots, S_n \) up to \( \tau \) and relations \( R_1, \ldots, R_m \) up to \( \tau \).

Based on this definition, informally we can think of continuous queries operationally as follows. Let time “advance” within domain \( T \), as further discussed below. First consider a query producing a stream. At time \( \tau \in T \), all inputs up to \( \tau \) are processed and the continuous query emits any new stream result elements with timestamp \( \tau \). Because of our assumptions on operators, stream elements with timestamp \( \tau \) do not depend on “future” inputs, i.e., inputs with timestamp \( > \tau \). A query producing a relation is similar: At time \( \tau \), all inputs up to \( \tau \) are processed and the continuous query updates the output relation to state \( R(\tau) \).

Now let us understand what it means for time to advance within domain \( T \). The relationship between application time, wall-clock time, and system time is a complex issue, discussed in depth by Srivastava et al. [98]. However, for precise query semantics we need to make no additional assumptions beyond those already made here. Time “advances” to \( \tau \) from \( \tau - 1 \) when all inputs up to \( \tau - 1 \) have been processed. It appears we are tacitly assuming that streams arrive in timestamp order, relations are updated in timestamp order, and there is no timestamp “skew” across streams or relations. In practice, to implement our semantics correctly, systems cope with out-of-order and skewed inputs. This issue is revisited in Section 2.8 and thoroughly covered by Srivastava et al. [98].

**Example 2.3** Consider the query \( \text{Istream(Filter (LastRow(S)))} \) constructed from three operators and operating on stream \( S \). Let stream \( S \) have a single attribute and consist of the elements \( \{(a_0, 0), (a_1, 1), (a_2, 2), \ldots\} \). \( \text{LastRow} \) is a stream-to-relation operator; at any point in time the relation output by \( \text{LastRow} \) contains the last tuple that arrived on \( S \). \( \text{Filter} \) is a relation-to-relation operator that produces its output relation by applying a filter condition on its input. Suppose that tuples \( (a_0), (a_2), (a_4), \ldots \) satisfy the filter condition while tuples \( (a_1), (a_3), (a_5), \ldots \) do not. Finally, \( \text{Istream} \) is a relation-to-stream operator (defined formally in Section 2.6.3) that “streams” every new tuple inserted into its input relation. Figure 2.3 shows the outputs produced by each of the three operators as time progresses.
<table>
<thead>
<tr>
<th>Time</th>
<th>S</th>
<th>LastRow</th>
<th>Filter</th>
<th>Istream</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\langle a_0, 0 \rangle)</td>
<td>(a_0)</td>
<td>(\langle a_0, 0 \rangle)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(\langle a_0, 0 \rangle) , (\langle a_1, 1 \rangle)</td>
<td>(a_1)</td>
<td>(\phi)</td>
<td>(\langle a_0, 0 \rangle)</td>
</tr>
<tr>
<td>2</td>
<td>(\langle a_0, 0 \rangle) , (\langle a_1, 1 \rangle) , (\langle a_2, 2 \rangle)</td>
<td>(a_2)</td>
<td>(\phi)</td>
<td>(\langle a_0, 0 \rangle) , (\langle a_2, 2 \rangle)</td>
</tr>
<tr>
<td>3</td>
<td>(\langle a_0, 0 \rangle) , (\langle a_1, 1 \rangle) , (\langle a_2, 2 \rangle) , (\langle a_3, 3 \rangle)</td>
<td>(a_3)</td>
<td>(\phi)</td>
<td>(\langle a_0, 0 \rangle) , (\langle a_2, 2 \rangle)</td>
</tr>
<tr>
<td>4</td>
<td>(\langle a_0, 0 \rangle) , (\langle a_1, 1 \rangle) , (\langle a_2, 2 \rangle) , (\langle a_3, 3 \rangle) , (\langle a_4, 4 \rangle)</td>
<td>(a_4)</td>
<td>(\phi)</td>
<td>(\langle a_0, 0 \rangle) , (\langle a_2, 2 \rangle) , (\langle a_4, 4 \rangle)</td>
</tr>
</tbody>
</table>

Figure 2.3: Output produced by operators of Example 2.3 as time progresses.

**Example 2.4** In the Linear Road application, the sequence of operators producing derived relation \(SegVolRel\) of Example 2.2 conceptually produces, at every time instant \(\tau\), the instantaneous relation \(SegVolRel(\tau)\) containing the current number of vehicles in each segment. In a DSMS implementing our semantics, \(SegVolRel(\tau)\) cannot be produced until it is known that all elements on input stream \(PosSpeedStr(\text{vehicleId}, \text{speed}, \text{xPos})\) with timestamp \(\leq \tau\) have been received. Furthermore, once they have, there may be additional lag before the relation is actually updated due to query processing time. Our semantics does not dictate “liveness” of continuous query output—that issue is relegated to latency management in the query processor [16]. □

### 2.5.1 Formal Specification

In this section, we present a formal specification of our abstract semantics using denotational semantics [95, 100]. We refer the interested reader to reference [95] for a detailed discussion of the merits of a formal semantics specification. In database research, an example of a formal specification, for an active database rule language, can be found in reference [114]. (The remainder of the thesis does not depend on the material of this section,
so a reader who is not interested in the intricate details of a formal semantics may skip this section.)

A denotational semantics for a query language is specified by defining a meaning function \( M \). Function \( M \) takes any query \( Q \) belonging to the language and returns the “input-output” function, denoted \( M[Q] \), computed by \( Q \). For example if \( Q \) is the SQL query “\( R \) NATURAL JOIN \( S \)”, then \( M[Q] \) is a function \( f \) that takes two relations as inputs and produces the join of the two relations as output. The functions such as \( f \) returned by the meaning function \( M \) are represented using lambda calculus [90].

For a continuous query \( Q \), \( M[Q] \) is a function that takes as input instances of the streams and relations referenced in \( Q \), along with a time instant \( \tau \), and produces as output the new stream tuples corresponding to time \( \tau \) or the relation instance at time \( \tau \).

Preliminaries

Figure 2.4 presents an abstract syntax for continuous queries using BNF-style rules, which we use to represent queries in the denotational semantics specification. Table 2.1 provides meanings of the terms used in the syntax specification. Recall that our abstract semantics is defined for continuous queries built from three classes of operators—stream-to-relation, relation-to-relation, and relation-to-stream—and this is reflected in the abstract syntax. Note that this syntax is specified solely for the purpose of presenting our abstract semantics; the actual syntax of a language (such as CQL) derived from our abstract semantics could be significantly different.

The following domains are used in the semantics specification:

- *Time domain* (\( T \)): \( T = \{0, 1, \ldots\} \) as described in Section 2.4.

- *Tuple domain* (\( TP \)): The domain of tuples. A tuple is a finite sequence of atomic values. We do not distinguish between tuples with different schemas.
CHAPTER 2. CONTINUOUS QUERY LANGUAGE

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q)</td>
<td>Continuous Query (CQ)</td>
<td>(Query)</td>
</tr>
<tr>
<td>(Q_R)</td>
<td>CQ producing a relation</td>
<td>(RelQuery)</td>
</tr>
<tr>
<td>(Q_S)</td>
<td>CQ producing a stream</td>
<td>(StrQuery)</td>
</tr>
<tr>
<td>(S2R-Op)</td>
<td>Stream-to-Relation Operator</td>
<td>(S2ROp)</td>
</tr>
<tr>
<td>(R2R-Op)</td>
<td>Relation-to-Relation Operator</td>
<td>(R2ROp)</td>
</tr>
<tr>
<td>(R2S-Op)</td>
<td>Relation-to-Stream Operator</td>
<td>(R2SOp)</td>
</tr>
<tr>
<td>(RName)</td>
<td>Relation Name</td>
<td>(Identifier)</td>
</tr>
<tr>
<td>(SName)</td>
<td>Stream Name</td>
<td>(Identifier)</td>
</tr>
<tr>
<td>(Id)</td>
<td>Identifier</td>
<td>(Identifier)</td>
</tr>
</tbody>
</table>

Table 2.1: Description of the terms used in the abstract syntax of Figure 2.4.

- **Tuple multiset domain** (\(\Sigma\)): The domain of finite, but unbounded, bags of tuples.

- **Relation domain** (\(R\)): \(R = T \rightarrow \Sigma\), i.e., the domain of functions that map time instants to bags of tuples (Definition 2.2).

- **Stream domain** (\(S\)): The domain of (possibly infinite) multisets over \(TP \times T\) (Definition 2.1).

- **Relational operator domain** (\(R_{op}\)): \(R_{op} = \Sigma \times \cdots \times \Sigma \rightarrow \Sigma\), i.e., the domain of functions that produce a bag of tuples from one or more bags of tuples. For example, the standard relational algebra operators (e.g., \(\sigma\), \(\pi\), \(\bowtie\)) and SQL queries belong to this domain. (Note that “relational” here refers to traditional relations, not time-varying ones.)

- **Syntactic domains**: The domains associated with the syntactic terms listed in Table 2.1. For example, \(Query\) denotes the domain of valid continuous queries according to the syntax in Figure 2.4, and \(R2ROp\) denotes the domain of relation-to-relation operators.

- **Relation Lookup domain** (\(RelLookup\)): The domain of functions that map an identifier (relation name) to its corresponding relation, i.e., \(RelLookup = Identifier \rightarrow R\).

- **Stream Lookup domain** (\(StrLookup\)): The domain of functions that map an identifier (stream name) to its corresponding stream, i.e., \(StrLookup = Identifier \rightarrow S\).
### Denotational Semantics

Recall that denotational semantics for a query language specifies a meaning function \( M \) that maps queries in the language to the input-output function that they compute. Following convention [95], we specify the meaning function recursively, using subsidiary meaning functions for subcomponents of a query. Figure 2.5 lists the meaning functions that we use in our specification. \( M \) is the “main” meaning function, which assigns a meaning to an entire query. The other functions assign meaning to (sub)components of a query; for example, \( M_{S2R} \) maps a relation-to-relation operator to a function over conventional relations.

As we mentioned earlier, lambda calculus is used to represent functions. The lambda calculus expression \( \lambda x_1 \ldots \lambda x_n . E \) defines a function that takes \( n \) arguments \( v_1, \ldots, v_n \), and returns the result of evaluating expression \( E \) with all free occurrences of \( x_i \) in \( E \) replaced by \( v_i, 1 \leq i \leq n \). The arguments \( v_i \) and the returned result could themselves be functions, i.e., lambda calculus expressions.

Our abstract semantics treats the three classes of operators as black boxes. Therefore we assume that the meaning functions \( M_{S2R}, M_{R2R}, \) and \( M_{R2S} \) are given. Section 2.5.1 specifies these meaning functions for some example operators.

The specifications of the remaining three meaning functions are given below. Each meaning function is specified in separate parts, one part for each BNF rule for its corresponding syntactic term (Figure 2.4). For example, \( M \) is specified in two parts: one corresponding to the derivation of \( Q \) from \( Q_R \) and the other to the derivation from \( Q_S \). The complete meaning function can be thought of as a combination of its parts using appropriate “if-then-else” statements. In these functions, we use the \( \in \) symbol as if we are considering sets rather than multisets. In the presence of duplicates, each duplicate must be considered

---

### Figure 2.5: Meaning functions in the denotational semantics

<table>
<thead>
<tr>
<th>Query component</th>
<th>Meaning Fn.</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q )</td>
<td>( M )</td>
<td>( Query \rightarrow (RelLookup \times StrLookup \times T \rightarrow (\Sigma \cup \mathcal{S})) )</td>
</tr>
<tr>
<td>( Q_R )</td>
<td>( M_R )</td>
<td>( RelQuery \rightarrow (RelLookup \times StrLookup \times T \rightarrow \Sigma) )</td>
</tr>
<tr>
<td>( Q_S )</td>
<td>( M_S )</td>
<td>( StrQuery \rightarrow (RelLookup \times StrLookup \times T \rightarrow \mathcal{S}) )</td>
</tr>
<tr>
<td>( S2R-0p )</td>
<td>( M_{S2R} )</td>
<td>( S2ROp \rightarrow (\mathcal{S} \times T \rightarrow \Sigma) )</td>
</tr>
<tr>
<td>( R2R-0p )</td>
<td>( M_{R2R} )</td>
<td>( R2ROp \rightarrow \mathcal{R}_{op} )</td>
</tr>
<tr>
<td>( R2S-0p )</td>
<td>( M_{R2S} )</td>
<td>( R2SOp \rightarrow (\mathcal{R} \times T \rightarrow \mathcal{S}) )</td>
</tr>
</tbody>
</table>
separately when evaluating $\in$—in all cases the interpretation is obvious from context.

- $\mathcal{M}$: The input-output function $\mathcal{M}[Q]$ produced by $\mathcal{M}$ for a query $Q$ takes three parameters: the first two parameters are functions that are used to map relation or stream names in the query to the corresponding relation and stream instances, and the third parameter is a time instant. $\mathcal{M}[Q](r, s, \tau)$ specifies the output produced by $Q$ at time instant $\tau$. $\mathcal{M}[Q](r, s, \tau)$ invokes $\mathcal{M}_R[Q_R](r, s, \tau)$ if $Q = Q_R$ produces a relation as output, and invokes $\mathcal{M}_S[Q_S](r, s, \tau)$ if $Q = Q_S$ produces a stream as output. An additional filtering operation is required for the latter case since $\mathcal{M}_S[Q_S](r, s, \tau)$ returns all elements of its output stream with timestamp $\leq \tau$. (See the definition of $\mathcal{M}_S$ below.)

$$\mathcal{M}[Q] = \lambda r.\lambda s.\lambda \tau.\mathcal{M}_R[Q_R](r, s, \tau)$$

$$\mathcal{M}[Q_S] = \lambda r.\lambda s.\lambda \tau.\{\langle e, \tau \rangle : (e, \tau) \in \mathcal{M}_S[Q_S](r, s, \tau)\}$$

- $\mathcal{M}_R$: If $Q_R$ is a (sub)query producing a relation, $\mathcal{M}_R[Q_R](r, s, \tau)$ specifies the bag of tuples in the output relation at time $\tau$. Parameters $r$ and $s$, as before, are stream and relation lookup functions. $\mathcal{M}_R$ calls functions $\mathcal{M}_R$ (recursively), $\mathcal{M}_S$, $\mathcal{M}_{R2R}$, $\mathcal{M}_{S2R}$, and $\mathcal{M}_R$, depending on the structure of $Q_R$. For example, if $Q_R = \text{RName}$, $\mathcal{M}_R[Q_R](r, s, \tau)$ uses function $r$ to look up the time-varying relation corresponding to $\text{RName}$, and applies the relation to identify the bag of tuples at time $\tau$.

$$\mathcal{M}_R[\text{RName}] = \lambda r.\lambda s.\lambda \tau.\text{RName}(\tau)$$

$$\mathcal{M}_R[\text{R2R-Op}(Q^1_R, \ldots, Q^n_R)] = \lambda r.\lambda s.\lambda \tau.\mathcal{M}_{R2R}[\text{R2R-Op}](\mathcal{M}_R[Q^1_R](r, s, \tau), \ldots, \mathcal{M}_R[Q^n_R](r, s, \tau))$$

$$\mathcal{M}_R[\text{S2R-Op}(Q_S)] = \lambda r.\lambda s.\lambda \tau.\mathcal{M}_{S2R}[\text{S2R-Op}](\mathcal{M}_S[Q_S](r, s, \tau), \tau)$$

- $\mathcal{M}_S$: If $Q_S$ is a (sub)query producing a stream, $\mathcal{M}_S[Q_S](r, s, \tau)$ specifies the bag of stream elements in the output stream with timestamp $\leq \tau$. In the specification of $\mathcal{M}_S[Q_S]$ for the case $Q_S = \text{R2S-Op}(Q_R)$, the lambda calculus expression \(\lambda \tau'. \mathcal{M}_R[Q_R](r, s, \tau')\)' defines a function that takes a single parameter $\tau'$ and returns the bag of tuples at time $\tau'$ in the relation produced by subquery $Q_R$, which is just a
formal representation for the relation produced by $Q_R$.  

\[
\begin{align*}
&M_S[SName] = \lambda r.\lambda s.\lambda \tau. \{ \langle e, \tau' \rangle : \langle e, \tau' \rangle \in s(SName) \land \tau' \leq \tau \} \\
&M_S[R2S-Op(Q_R)] = \lambda r.\lambda s.\lambda \tau. M_{R2S}[R2S-Op]((\lambda \tau'. M_R[Q_R](r, s, \tau'))(\tau)
\end{align*}
\]

There are a variety of choices in the details of how we specify the denotational semantics, all adhering to the meaning of the language. For example, we could have presented the semantics so that $M_S[Q_S](r, s, \tau)$ specifies the bag of output stream elements with timestamp equal to $\tau$, instead of those with timestamp $\leq \tau$. However, doing so would have made the semantics specification for stream-to-relation operators more complicated. Overall, we considered several alternatives and picked the ones that we felt were most intuitive and easy to understand.

**Semantics for Example Operators**

Our abstract semantics treats the semantics of operators (the meaning functions $M_{S2R}$, $M_{R2R}$, and $M_{R2S}$) as black boxes. We now present formal semantics for a few concrete operators for illustration. Figure 2.6 lists the example operators we consider and their abstract syntax using the BNF-style rules. (Most of these operators have CQL analogues and are covered extensively in later sections.)

- $M_{S2R}$: We consider three kinds of sliding windows as examples of stream-to-relation operators. All three operators take a stream $S$ and a timestamp $\tau$ as input and return a bag of tuples as output: The $\text{Now}$ window operator returns the tuples of $S$ with timestamp $\tau$; the $\text{Range}$ window operator, specified using a parameter $T$, returns the tuples of $S$ with timestamps in the range $[\tau - T, \tau]$; the $\text{Row}$ window operator, specified using an integer parameter $N$, returns the $N$ most recent tuples of $S$ with
timestamps ≤ τ.²

\[ M_{S2R}[\text{Now}] = \lambda S. \lambda \tau. \{ e : \langle e, \tau \rangle \in S \} \]
\[ M_{S2R}[\text{Range}(T)] = \lambda S. \lambda \tau. \{ e : \langle e, \tau' \rangle \in S \land \max(\tau - T, 0) \leq \tau' \leq \tau \} \]
\[ M_{S2R}[\text{Row}(N)] = \lambda S. \lambda \tau. \{ e : \langle e, \tau' \rangle \in S \land (\tau' \leq \tau) \land
\quad (N \geq \vert \{ \langle e, \tau'' \rangle \in S : \tau' \leq \tau'' \leq \tau \} \vert) \} \]

- **M_{R2R}**: We present formal semantics for restricted versions of two standard relational operators: semijoin and filter (selection). `SemiJoin(i, j)` performs a semijoin on the \( i \)th attribute of its first input with the \( j \)th attribute of its second input, where both inputs are bags of tuples. `Filter(i, v)` returns all tuples from its input bag having value \( v \) in the \( i \)th attribute. In the definitions, \( e.i \) abuses standard notation to denote the value in the \( i \)th attribute of a tuple \( e \).

\[ M_{R2R}[\text{SemiJoin}(i, j)] = \lambda E_1. \lambda E_2. \{ e_1 : e_1 \in E_1 \land (\exists e_2 \in E_2 : e_1.i = e_2.j) \} \]
\[ M_{R2R}[\text{Filter}(i, v)] = \lambda E. \{ e : e \in E \land e.i = v \} \]

- **M_{R2S}**: We present formal semantics for three relation-to-stream operators. The `IStream` operator takes a (time-varying) relation \( R \) and a time instant \( \tau \), and streams the new tuples inserted into \( R \) at time \( \tau \), i.e., tuples that appear in \( R(\tau) \) but not in \( R(\tau - 1) \). Analogously, the `DStream` operator streams the tuples that were deleted from \( R \) at time \( \tau \), i.e., tuples that appear in \( R(\tau - 1) \) but not in \( R(\tau) \). Finally, the `RStream` operator streams all the tuples in \( R(\tau) \). In the definitions, assume \( R(-1) = \phi \).

\[ M_{R2S}[\text{IStream}] = \lambda R. \lambda \tau. \{ (e, \tau') : \tau' \leq \tau \land e \in R(\tau) \land e \notin R(\tau - 1) \} \]
\[ M_{R2S}[\text{DStream}] = \lambda R. \lambda \tau. \{ (e, \tau') : \tau' \leq \tau \land e \in R(\tau - 1) \land e \notin R(\tau) \} \]
\[ M_{R2S}[\text{RStream}] = \lambda R. \lambda \tau. \{ (e, \tau') : \tau' \leq \tau \land e \in R(\tau) \} \]

²If the stream has duplicate timestamps, our formal specification of the `Row` window operator may return fewer than \( N \) elements. An alternate definition presented in Section 2.6.1 ensures that there are always \( N \) elements, but introduces nondeterminism in the presence of duplicate timestamps.
CQL uses the same three relation-to-stream operators, and they are discussed in detail in Section 2.6.3.

**Example 2.5** Consider the following query over PosSpeedStr:

\[
Q = \text{RStream(Filter(1, 21)(Now(PosSpeedStr))))}
\]

The query applies a \text{Now} window operator over PosSpeedStr, applies a filter over the resulting relation that selects all tuples with \text{vehicleId} 21 (recall that the first column of PosSpeedStr is \text{vehicleId}), and applies an \text{Rstream} operator over the relation output by the filter. The reader can verify that the meaning of \(Q\) is the following expression, after some simplifications:

\[
\mathcal{M}[Q] = \lambda r.\lambda s.\lambda \tau.\{ (e, \tau') : (e, \tau') \in s(\text{PosSpeedStr}) \land \tau' = \tau \land e.1 = 21 \}
\]

### 2.6 CQL

This section contains a detailed description of CQL: Sections 2.6.1-2.6.3 describe the three classes of operators in CQL. Section 2.6.4 presents syntactic shortcuts and defaults to simplify expression of common constructs. Section 2.6.5 specifies our running example in CQL. The remaining sections deal with various expressiveness related issues (Sections 2.6.6, 2.6.9, and 2.6.10), equivalences (Section 2.6.8), and common constructs (Section 2.6.7).

**Overall Approach**

Broadly, our approach to designing operators in CQL is as follows: Support a large class of relation-to-relation operators, which perform the bulk of data manipulation in a typical CQL query, along with a small set of stream-to-relation and relation-to-stream operators that convert streams to relations and back. The primary advantage of this approach is the ability to reuse the formal foundations and huge body of implementation techniques for relation-to-relation languages such as relational algebra and SQL, instead of starting from scratch with a heavily stream-based language.

Technically, we cannot directly import existing conventional relation-to-relation operators into our concrete language, since they operate on traditional relations while we operate on time-varying relations, but the mapping is obvious: Let \(O_r\) denote a traditional
$n$-ary relational operator. The corresponding relation-to-relation operator $O_c$ in CQL produces the time-varying relation $R$ such that at each time $\tau$, $R(\tau) = O_c(R_1(\tau), \ldots, R_n(\tau))$.

An apparent drawback of our approach is that even a simple filter on a stream requires three operators: one to turn the stream into a relation, one to perform a relational filter, and one to turn the relation back into a stream. However, CQL’s defaults and syntactic shortcuts make filters and other simple queries easy to express (Section 2.6.4).

Although we do not specify them explicitly as part of our language, incorporating user-defined procedures, aggregates, and windows, as may be required for more complex, application-specific stream processing, is straightforward in CQL, at least from the semantics perspective.

Next, in Sections 2.6.1–2.6.3, we cover the three classes of operators in CQL.

### 2.6.1 Stream-to-Relation Operators

The stream-to-relation operators in CQL are based on the concept of a *sliding window*. A sliding window (or simply a *window*) is a well-studied, fundamental operation over a stream. In most data stream applications, the recent tuples of the stream contain more useful, actionable information than the tuples that arrived long in the past. A sliding window is commonly used to express this preference for recent tuples. Intuitively, at any given point in time, a sliding window identifies a collection of recent tuples of the stream. There are different kinds of sliding windows, which differ from each other on how the collection is identified. CQL uses three kinds: *time-based*, *tuple-based*, and *partitioned*. Other kinds such as *fixed* [105] windows and *value-based* [97] windows can be incorporated into CQL easily—new syntax must be added, but the semantics of incorporating a new window type relies solely on the semantics of the window operator itself, thanks to the development of our abstract semantics.

#### Time-based sliding windows

A time-based sliding window on a stream $S$ takes a time-interval $T$ as a parameter and is specified by following the reference to $S$ with [Range $T$]. When time domain $T$ is *DateTime* [40], we use the syntax described in Figure 2.7 to specify a time-interval; new syntax must be added for other domains. Intuitively, a time-based window defines its output relation over time by sliding an interval of size $T$ time units capturing the latest portion of
an ordered stream. More formally, the output relation \( R \) of \(" S \ [\text{Range } T]\)" is defined as:

\[
R(\tau) = \{ s \mid (s, \tau') \in S \land (\tau' \leq \tau) \land (\tau' \geq \max\{\tau - T + 1, 0\})\}
\]

Two important special cases are \( T = 1 \) and \( T = \infty \). When \( T = 1 \), \( R(\tau) \) consists of tuples obtained from elements of \( S \) with timestamp \( \tau \). In CQL we introduce the syntax \"S \ [\text{Now}]\" for this special case. When \( T = \infty \), \( R(\tau) \) consists of tuples obtained from all elements of \( S \) up to \( \tau \) and uses the SQL-99 syntax \"S \ [\text{Range Unbounded}]\." We use the terms \textit{Now window} and \textit{Unbounded window} to refer to these two special windows.

Example 2.6 “PosSpeedStr \ [\text{Range 30 Seconds}]” denotes a time-based sliding window of 30 seconds over input stream PosSpeedStr. At any time instant, the output relation of the sliding window contains the bag of position-speed measurements from the previous 30 seconds. Similarly, at any instant “PosSpeedStr \ [\text{Now}]” contains the (possibly empty) bag of position-speed measurements from that instant, and “PosSpeedStr \ [\text{Range Unbounded}]” contains the bag of all position-speed measurements so far. \( \square \)

Tuple-based windows

A tuple-based sliding window on a stream \( S \) takes a positive integer \( N \) as a parameter and is specified by following the reference to \( S \) in the query with \[\text{Rows } N\]. At any given point in time the window contains the last \( N \) tuples of \( S \). More formally, let \( s_1, s_2, \ldots \), denote the tuples of \( S \) in increasing order of their timestamps, breaking ties arbitrarily. The output relation \( R \) of \"S \ [\text{Rows } N]\" is defined as:

\[
R(\tau) = \{ s_i \mid \max\{1, n(\tau) - N + 1\} \leq i \leq n(\tau)\}
\]

where \( n(\tau) \) denotes the size of \( S \) at time \( \tau \), i.e., the number of elements of \( S \) with timestamps \( \leq \tau \). The special case of \( N = \infty \) is specified by \[\text{Rows Unbounded}\], and is equivalent to \[\text{Range Unbounded}\]. Since we break ties arbitrarily when defining the tuple sequence \( s_1, s_2, \ldots \), tuple-based windows are nondeterministic—and therefore may not be
appropriate—when timestamps are not unique.

**Example 2.7** A tuple-based sliding window does not make much sense over PosSpeedStr (except the case of $N = \infty$), since stream element timestamps are not unique. For example, at any instant sliding window $\text{PosSpeedStr}[\text{Rows 1}]$ denotes the “latest” position-speed measurement, which is ambiguous whenever multiple measurements carry the same timestamp—a common occurrence in the Linear Road application.

**Partitioned windows**

Partitioned sliding windows are closely related to the idea of a *substream*. Sometimes, it is useful to logically partition a stream into a collection of smaller streams based on one or more attributes, and apply an operation to each substream independently. This overall operation is analogous to logically dividing a traditional relation into *groups* and applying aggregation over each group using the group-by aggregation operator. For instance, in our running example, we might partition PosSpeedStr into a collection of substreams based on vehicleId, and compute average speed for each vehicle. In general, the partitioning attribute(s) are called the *substream key*.

A partitioned sliding window applies a tuple-based window over each substream of a stream. It takes two parameters—the length of the tuple-based window $N$ and the substream key $\{A_1, \ldots, A_k\}$—and is specified by following the reference to $S$ in the query with $[\text{Partition By } A_1, \ldots, A_k \text{ Rows } N]$. At any instant, the output relation contains the union of the windows over all the substreams. More formally, a tuple $s$ with values $a_1, \ldots, a_k$ for attributes $A_1, \ldots, A_k$ occurs in the output instantaneous relation $R(\tau)$ exactly when there exists an element $\langle s, \tau' \rangle \in S$, $\tau' \leq \tau$ such that $\tau'$ is among the $N$ largest timestamps of elements whose tuples have values $a_1, \ldots, a_k$ for attributes $A_1, \ldots, A_k$. Note that the analogous time-based partitioned windows would not provide additional expressiveness over nonpartitioned time-based windows.

**Example 2.8** The partitioned window “PosSpeedStr [Partition By vehicleId Rows 1]” partitions stream PosSpeedStr into substreams based on vehicleId and picks the latest element in each substream. (Note that there is no ambiguity in picking the latest element in each substream, since position-speed reports for a particular vehicle are made only once in 30 seconds, and the granularity of Datetime is one second.) At any time instant, the relation
CHAPTER 2. CONTINUOUS QUERY LANGUAGE

defined by the window contains the latest speed-position measurement for each vehicle that has ever transmitted a measurement.

Windows with a “slide” parameter

Windows can optionally contain a slide parameter, indicating the granularity at which the window slides. The slide parameter is a time-interval for time-based windows and a positive integer for row-based and partitioned windows.

A time-based window over stream \( S \) with window size \( T \) and slide parameter \( L \) is denoted as \( S \ [\text{Range} \ T \ \text{Slide} \ L] \). Its output relation is:

\[
R(\tau) = \begin{cases} 
\phi & \text{if } \tau < L - 1 \\
\{ s \mid \langle s, \tau' \rangle \in S \land (\tau' \geq \tau_{end}) \land (\tau' \leq \tau_{start}) \} & \text{otherwise}
\end{cases}
\]

where

\[
\tau_{start} = \lfloor (\tau + 1)/L \rfloor \cdot L - 1
\]
\[
\tau_{end} = \max\{\tau_{start} - T + 1, 0\}
\]

(The expression \( \lfloor (\tau + 1)/L \rfloor \cdot L - 1 \) computes the largest number of the form \( k \cdot L - 1 \) that is not greater than \( \tau \). For example, the expression evaluates to \( L - 1 \) for all values of \( \tau \) between \( L - 1 \) and \( 2L - 2 \), evaluates to \( 2L - 1 \) for all values of \( \tau \) between \( 2L - 1 \) and \( 3L - 2 \), and so on.) Intuitively, “\( S \ [\text{Range} \ T \ \text{Slide} \ L] \)” defines its output relation by sliding an interval of width \( T \) time units over \( S \), but the interval slides once only every \( L \) time units, by an amount \( L \). Note that we can treat “\( S \ [\text{Range} \ T] \)” as an abbreviation for “\( S \ [\text{Range} \ T \ \text{Slide} \ 1] \)”, where 1 is the granularity of the time domain. The following example illustrates the use of slide parameter.

Example 2.9 “\( \text{PosSpeedStr} \ [\text{Range} \ 1 \ \text{Minute} \ \text{Slide} \ 1 \ \text{Minute}] \)” denotes a one-minute window over \( \text{PosSpeedStr} \) that slides at a one-minute granularity. At any point in time the window contains the speed-position reports from the last clock minute. For example, at time instant 59s, the window contains the first minute (0-59) of speed-position reports, and continues to contain the same bag of tuples until the time instant 119s, when it shifts to the next minute (60-119), and so on. This type of window, in which \( L = T \), has been referred to as a tumbling window in previous work [28, 29].
A tuple-based window over $S$ with window size $N$ and slide parameter $L$ is denoted as “$S \ [\text{Rows} \ N \ \text{Slide} \ L]$”. As in the definition of tuple-based windows, let $s_1, s_2, \ldots$, denote tuples of $S$ in increasing order of their timestamps, breaking ties arbitrarily. The output relation $R$ of the window is:

$$R(\tau) = \begin{cases} 
\phi & \text{if } n(\tau) < L - 1 \\
\{s_i \mid p_{\text{end}} \leq i \leq p_{\text{start}}\} & \text{otherwise}
\end{cases}$$

where $n(\tau)$ is the size of $S$ at time $\tau$, and

\[
p_{\text{start}} = \lfloor n(\tau)/L \rfloor \cdot L \quad p_{\text{end}} = \max\{p_{\text{start}} - N + 1, 1\}
\]

The definition of a partition-based window with a slide parameter is a natural extension of the original definition: the window applies a tuple-based window with a slide parameter over each substream and takes the union of all the substream windows. We omit the formal definition.

### 2.6.2 Relation-to-Relation Operators

The relation-to-relation operators in CQL are derived from traditional relational queries expressed in SQL, with the straightforward semantic mapping to time-varying relations specified at the beginning of this section. Anywhere a traditional relation is referenced in a SQL query, a (base or derived) relation can be referenced in CQL.

**Example 2.10** Consider the following CQL query for the Linear Road application:

```
Select Distinct vehicleId 
From PosSpeedStr [Range 30 Seconds]
```

This query is composed from a stream-to-relation sliding-window operator, followed by a relation-to-relation operator that performs projection and duplicate-elimination. The output relation of this query contains, at any time instant, the set of “active vehicles”—those vehicles having transmitted a position-speed measurement within the last 30 seconds. □
2.6.3 Relation-to-Stream Operators

CQL has three relation-to-stream operators: $Istream$, $Dstream$, and $Rstream$. In the following formal definitions, operators $\cup$, $\times$, and $-$ are assumed to be the bag versions.

1. $Istream$ (for “insert stream”) applied to relation $R$ contains a stream element $\langle s, \tau \rangle$ whenever tuple $s$ is in $R(\tau) - R(\tau - 1)$. Assuming $R(-1) = \phi$ for notational simplicity, we have:

$$Istream(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\})$$

2. Analogously, $Dstream$ (for “delete stream”) applied to relation $R$ contains a stream element $\langle s, \tau \rangle$ whenever tuple $s$ is in $R(\tau - 1) - R(\tau)$. Formally:

$$Dstream(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\})$$

3. $Rstream$ (for “relation stream”) applied to relation $R$ contains a stream element $\langle s, \tau \rangle$ whenever tuple $s$ is in $R$ at time $\tau$. Formally:

$$Rstream(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\})$$

(An analogous denotational specification for these operators was presented in Section 2.5.1.) A careful reader may observe that $Istream$ and $Dstream$ are expressible using $Rstream$ along with time-based sliding windows and some relational operators. However, we retain all three operators in CQL in keeping with goal #2 from Section 2.1: easy queries should be easy to write.

Example 2.11 Consider the following CQL query for stream filtering:

```
Select Istream(*)
From PosSpeedStr [Range Unbounded]
Where speed > 65
```

(Note the syntax of the relation-to-stream operator in the Select clause.) This query is composed from three operators: an Unbounded window producing a relation that at time $\tau$ contains all speed-position measurements up to $\tau$, a relational filter operator that restricts
the relation to those measurements with speed greater than 65 MPH, and an Istream operator that streams new values in the (filtered) relation as the result of the query. The effect is a simple filter over PosSpeedStr that outputs all input elements with speed greater than 65 MPH. The same filter query can be written using the Rstream operator and a Now window:

\[
\text{Select Rstream(*)} \\
\text{From PosSpeedStr [Now]} \\
\text{Where speed > 65}
\]

As we will see shortly, our defaults also permit this query to be written in its most intuitive form:

\[
\text{Select *} \\
\text{From PosSpeedStr} \\
\text{Where speed > 65}
\]

**Example 2.12** The following query illustrates the use of Dstream:

\[
\text{Select Dstream(VehicleId)} \\
\text{From PosSpeedStr [Range 30 Seconds]}
\]

This query is composed from three operators: The time-based window operator produces the relation containing the speed-position reports in the previous 30 seconds. The relation-to-relation operator (Select-From clause) projects the vehicleId attribute from this relation. Finally, the operator Dstream produces a vehicleId in the output whenever a vehicle is deleted from the above relation. In other words, the element \((v, \tau)\) appears in the output stream whenever vehicle \(v\) reported its position and speed at time \(\tau - 30\), but did not do so at time \(\tau\). This query thus detects when vehicles exit from the highway. (Recall that a vehicle on a highway reports its speed and position every 30 seconds.)

The Istream operator is used most commonly with Unbounded windows to express filter conditions as shown above, or to stream the results of sliding-window join queries. The Rstream operator is used most commonly with Now windows to express filter conditions as shown above, or to stream the results of joins between streams and relations. (See Section 2.6.7 for more details of such common constructs.) The Dstream operator is used less frequently than Istream or Rstream; see Stream Query Repository [104] for some examples of its use.
2.6.4 Syntactic Shortcuts and Defaults

In keeping with Goal #2 in Section 2.1, we permit some syntactic “shortcuts” in CQL that result in the application of certain defaults. Of course there may be cases where the default behavior is not what the author intended, so we assume that when queries are registered the system informs the author of the defaults applied and offers the opportunity to edit the expanded query. There are two classes of shortcuts: omitting window specifications and omitting relation-to-stream operators.

Default Windows

When a stream is referenced in a CQL query where a relation is expected (most commonly in the From clause), an Unbounded window is applied to the stream by default. While the default Unbounded window usually produces appropriate behavior, there are cases where a Now window is more appropriate, e.g., when a stream is joined with a relation; see Query 6 in Section 2.6.5.

Default Relation-to-Stream Operators

There are two cases in which it seems natural for authors to omit an intended Istream operator from a CQL query:

1. On the outermost query, even when streamed results rather than stored results (i.e., relations) are desired [80].

2. On an inner subquery, even though a window is specified on the subquery result.

For the first case we add an Istream operator by default whenever the query produces a relation that is monotonic. A relation $R$ is monotonic if $R(\tau_1) \subseteq R(\tau_2)$ whenever $\tau_1 \leq \tau_2$. Since we cannot test monotonicity in the general case, we use a conservative static monotonicity test. For example, a base relation is monotonic if it is known to be append-only, “S [Range Unbounded]” is monotonic for any stream $S$, and the join of two monotonic relations also is monotonic. If the result of a CQL query is a monotonic relation, then it makes intuitive sense to convert the relation into a stream using Istream. If it is not monotonic, the author might intend Istream, Dstream, or Rstream, so we do not add a relation-to-stream operator by default.
Similarly, for the second case we add an Istrea operator by default whenever the subquery is monotonic. If it is not monotonic, then the intended meaning of a window specification on the subquery result is somewhat ambiguous, so a semantic (type) error is generated, and the author must add an explicit relation-to-stream operator.

**Example 2.13** Now we see why the filter query of Example 2.11 can written in its most intuitive form:

```sql
Select *
From PosSpeedStr
Where speed > 65
```

Since `PosSpeedStr` is referenced without a window specification, an Unbounded window is applied by default. Further, since the output relation of the window and filter operators is monotonic, we add a default Istream operator to the result. The expanded query is identical to that of Example 2.11.

□

### 2.6.5 Linear Road in CQL

In this section, we specify our running example, the Linear Road application, in CQL. Recall that the Linear Road application has one base input stream, `PosSpeedStr`, containing speed-position measurements of vehicles using the highway. The output is a single stream `TollStr(vehicleId, toll)` specifying tolls for vehicles. Whenever a vehicle with `vehicleId v` enters a congested segment at time $\tau$, `TollStr` contains the element $\langle (v, l), \tau \rangle$ where $l$ denotes the toll for the congested segment at time $\tau$.

We incorporate two assumptions suggested in the original Linear Road specification [11] for computing tolls:

1. A vehicle is considered to have entered a segment when the first speed-position measurement for the vehicle is transmitted from that segment. The vehicle is considered to remain in the segment until it exits (see Assumption 2 below) or enters another segment (i.e., a speed-position measurement is transmitted from a different segment).
2. A vehicle is considered to have exited the highway when no speed-position report for that vehicle is transmitted for 30 seconds.
These assumptions are necessary given that each vehicle transmits its speed-position measurement only once every 30 seconds.

Since the continuous query producing TollStr is fairly complex, we express it using several named derived relations and streams. Figure 2.8 shows the derived relations and streams that we use, and their interdependencies. For example, TollStr is produced from derived stream VehicleSegEntryStr and derived relations CongestedSegRel and SegVolRel. Our one base input stream PosSpeedStr naturally appears as the source. We present specifications for the derived streams and relations in topological order according to Figure 2.8. For each derived stream and relation, we first describe its meaning, followed by the CQL (sub)query that produces it.

**Query 1** SegSpeedStr (vehicleId, speed, segNo): This stream is obtained from PosSpeedStr by replacing the xPos attribute of each element with the corresponding segment number. Since segments are exactly 1 mile long, the segment number is computed by (integer-)dividing xPos by 5280, the number of feet in a mile.

```
Select vehicleId, speed, xPos/5280 as segNo
From PosSpeedStr
```

Note the use of a default Unbounded window and a default Istream operator in this query.

□

---

Figure 2.8: Derived relations and streams for Linear Road queries
Query 2 ActiveVehicleSegRel (vehicleId, segNo): At any instant $\tau$, this relation contains the current segments of “active” vehicles, i.e., vehicles currently using the highway system.

Select vehicleId, segNo
From SegSpeedStr [Range 30 Seconds]

Informally, the query uses a time-based window to identify currently active vehicles based on Assumption 2 above.\(^3\)

Query 3 VehicleSegEntryStr (vehicleId, segNo): A vehicle $v$ entering a segment $s$ at time $\tau$ produces an element $\langle (v, s), \tau \rangle$ on this stream.

Select Istream(*)
From ActiveVehicleSegRel

VehicleSegEntryStr is produced by applying the Istream operator to ActiveVehicleSegRel. A vehicle $v$ entering a segment $s$ at time $\tau$ causes a new tuple to appear in ActiveVehicleSegRel at $\tau$, which causes the Istream operator to produce an element $\langle (v, s), \tau \rangle$ in VehicleSegEntryStr.\(^4\)

Query 4 CongestedSegRel (segNo): At any instant $\tau$, this relation contains the current set of congested segments. Recall from Section 2.3 that a segment is considered congested if the average speed of vehicles in the segment in the previous 5 minutes is less than 40 MPH\(^4\).

Select segNo
From SegSpeedStr [Range 5 Minutes]
Group By segNo
Having Avg(speed) < 40

Query 5 SegVolRel (segNo, numVehicles): This relation was introduced in Example 2.2. At any instant $\tau$, this relation contains the current count of vehicles in each segment.

---

\(^3\)In this query, we assume that a vehicle does not exit the highway and re-enter within 30 seconds. We could handle this case by using an additional Partition By window.

\(^4\)The average speed computation in the original Linear Road specification [11] is more complex than the one used in this query.
Select segNo, count(vehicleId) as numVehicles
From ActiveVehicleSegRel
Group By segNo

Query 6 TollStr(vehicleId,toll): This query gives the final output toll stream.

Select Rstream(E.vehicleId,
                   2 * (V.numVehicles-50) * (V.numVehicles-50) as toll)
From VehicleSegEntryStr [Now] as E, CongestedSegRel as C,
SegVolRel as V
Where E.segNo = C.segNo and C.segNo = V.segNo

At any instant τ, the Now window on the stream VehicleSegEntryStr identifies the set of
garvicles that have entered new segments at τ. This set of vehicles is joined with
CongestedSegRel and SegVolRel to determine which vehicles have entered congested
segments, and to compute tolls for such vehicles. Recall from Section 2.3 that the toll for a
congested segment is given by the formula 2 × (numvehicles − 50)2, where numvehicles
is the number of vehicles currently in the segment.

This query provides an example where the default Unbounded window would not yield
the intended behavior if a window specification were omitted. In general, if a stream is
joined with a relation in order to add attributes to and/or filter the stream, then a Now
window on the stream coupled with an Rstream operator usually provides the desired
behavior.

Recall that the Linear Road specification in this thesis is a simplified version of the
original [11]. A CQL specification of the complete Linear Road benchmark as well as a
number of other stream applications, such as network monitoring and online auctions [82],
is available at Stream Query Repository [104].

2.6.6 Operations over Timestamps

Timestamps of stream elements are not part of the schema, so we cannot reference timest-
amps within queries. We decided to make timestamps implicit for the following reasons:
1. Timestamps have certain properties (e.g., monotonicity) that we rely on in our abstract semantics (and in our CQL implementation). Therefore, we cannot permit queries to perform arbitrary transformations on timestamps.

2. Making timestamps implicit limits the operations performed on them, simplifying query plan generation and optimization.

If an application wishes to pose queries referring to timestamps explicitly, it can do so by simply mirroring the timestamp attribute in its stream schemas. The following example illustrates this point.

**Example 2.14** We can add an explicit timestamp attribute to PosSpeedStr, resulting in the schema PosSpeedStr(vehicleId, speed, xPos, tstamp). When a vehicle $v$ reports its position $x$ and speed $s$ at timestamp $\tau$, the element $\langle (v, s, x, \tau), \tau \rangle$ arrives on PosSpeedStr. The following query computes the delay between the last two speed-position reports received from any vehicles:

```sql
Select Max(tstamp) - Min(tstamp)
From PosSpeedStr [Rows 2]
```

This query cannot be expressed through implicit timestamps only.

### 2.6.7 Common Constructs

In this section, we illustrate and discuss a few constructs we found to appear frequently in CQL queries, primarily based on our experience with the Stream Query Repository [104].

**Stream Filters**

A filter over a stream can be expressed in two ways: using an $I_{stream}$-Unbounded window combination or an $R_{stream}$-Now window combination. Both of these were illustrated in Example 2.11. Note the $I_{stream}$-Unbounded window combination is the default for streams whose window specification is omitted (recall Section 2.6.4).

**Stream-Relation Joins**

When a stream is joined with a relation, it is usually most meaningful to apply a $Now$ window over the stream, and an $R_{stream}$ operator over the join result. Consider a stream Item
of purchased items and a relation `PriceTable` of current item prices. The query:

\[
\text{Select Rstream(Item.id, PriceTable.price)} \\
\text{From Item [Now], PriceTable} \\
\text{Where Item.id = PriceTable.itemId}
\]

produces the streamed items with their current price appended. Using other types of windows or other relation-to-stream operators usually does not produce intuitive results. For example, the query:

\[
\text{Select Istream(Item.id, PriceTable.price)} \\
\text{From Item [Range Unbounded], PriceTable} \\
\text{Where Item.id = PriceTable.itemId}
\]

produces, along with new items, the (new) price for all previously-purchased items whenever the price for an item changes.

**Sliding-Window Joins**

Sliding-window joins (SWJ) of two streams is an operation that has received a great deal of attention [22, 38, 99]. If neither stream in the join has duplicates, the usual semantics for SWJ can be expressed in CQL using an `Istream` operator. For example:

\[
\text{Select Istream(*) From S1 [Rows 5], S2 [Rows 10] Where S1.A = S2.A}
\]

is a sliding-window natural-join of `S1` and `S2` with a 5-tuple window on `S1` and a 10-tuple window on `S2`. A new tuple of `S1` produces one or more output join tuples if it joins with one or more of the last 10 tuples of `S2`; a new tuple of `S2` produces output join tuples in a similar manner.

If either stream can have duplicates, the query above may not have the expected SWJ semantics. Suppose a new tuple of `S2` is identical to the tuple 10 positions earlier. Then the new `S2` tuple does not produce any result tuples, even if it joins with one of the last 5 tuples of `S1`, because the relation produced by joining the two windows is unchanged. The more general SWJ that handles duplicates correctly can be expressed in CQL, but it is somewhat more involved.
Aggregation over Sliding Windows

Another common operation that will be the focus of Chapters 4 and 5 is *Aggregation over a Sliding Window (ASW)*. This operation applies a sliding window over its input stream and computes an aggregation function over the resulting relation. For example,

```sql
Select Avg(speed)
From PosSpeedStr [Range 5 Minutes]
```

computes the average speed of vehicles over the previous 5 minutes. An important variant is to apply the ASW operator over substreams, producing one windowed-aggregation value for each substream of the stream. (Recall substreams were first introduced under “Partitioned Windows” in Section 2.6.1.) In CQL, we express this variant using the `Group By` clause along with a sliding window. We use a partitioned window if we wish to apply a tuple-based window over each substream, otherwise we use a regular time-based window. For example:

```sql
Select vehicleId, Avg(speed)
From PosSpeedStr [Range 5 Minutes]
Group By vehicleId
```

computes the average speed of each vehicle over the previous 5 minutes, while

```sql
Select vehicleId, Avg(speed)
From PosSpeedStr [Partition By vehicleId Rows 5]
Group By vehicleId
```

computes the average speed of each vehicle using the last 5 speed reports made by it.

**Streamed Aggregations**

The aggregation operation in CQL produces relations by default, not streams, since it is a relation-to-relation operators. We describe two common types of aggregation queries that produce streams:

1. *Stream the value of the aggregation whenever it changes*: This behavior can be expressed using the `Istream` operator over the aggregation. For example:
Select Istream(Count(*))
From PosSpeedStr [Range 1 Minute]

counts the speed-position reports over the previous minute, and streams the count whenever it changes.

2. **Stream the value of the aggregation periodically:** This behavior can be expressed using windows with a slide parameter (recall Section 2.6.1) and Istream. For example:

Select Istream(Count(*))
From PosSpeedStr [Range 1 Minute Slide 1 Minute]

streams the number of speed-position reports over the last minute once every minute. A small subtlety with this query is that it will not stream the aggregation value if it remains unchanged from one minute to the next. Ensuring the value is streamed even when unchanged requires a more complex query. (Note that using an Rstream instead of an Istream in the query above will stream the aggregation value every time instant, not once every minute as required.)

### 2.6.8 Equivalences in CQL

In this section we briefly consider syntactic equivalences in the CQL language. As in any declarative language, equivalences can enable important query-rewrite optimizations.

All equivalences that hold in SQL with standard relational semantics carry over to the relational portion of CQL, including join reordering, predicate pushdown, and subquery flattening [50]. Furthermore, since any CQL query or subquery producing a relation can be thought of as a materialized view that is updated over time, all equivalences from materialized view maintenance [61] can be applied to CQL. For example, a materialized view joining two relations generally is maintained incrementally rather than by recomputation, and the same approach can be used to join two relations (or windowed streams) in CQL. Here we present two new stream-based equivalences: *window reduction* and *filter-window commutativity*.

**Window Reduction**

The following equivalence can be used to rewrite any CQL query or subquery with an Unbounded window and an Istream operator into an equivalent (sub)query with a Now
window and an Rstream operator. Here, \( L \) is any select-list, \( S \) is any stream (possibly a subquery producing a stream), and \( C \) is any condition.

\[
\text{Select Istream}(L) \text{ From } S \ [\text{Range Unbounded}] \text{ Where } C \\
\equiv \\
\text{Select Rstream}(L) \text{ From } S \ [\text{Now}] \text{ Where } C
\]

Furthermore, if stream \( S \) has a key (no duplicates), then we need not replace the Istream operator with Rstream, although once a Now window is applied there is little difference in efficiency between Istream and Rstream. (More generally, Istream and Rstream are equivalent over any relation \( R \) for which \( R(\tau) \cap R(\tau - 1) = \emptyset \) for all \( \tau \).)

Transforming Unbounded to Now obviously suggests a much more efficient implementation: logically, Unbounded windows require buffering the entire history of a stream, while Now windows allow a stream tuple to be discarded as soon as it is processed.

**Filter-Window Commutativity**

Another equivalence that can be useful for query-rewrite optimization is the commutativity of selection conditions and time-based windows. Here, \( S \) is any stream (including a subquery producing a stream), \( C \) is any condition, and \( T \) is any time interval.

\[
(\text{Select } * \text{ From } S \text{ Where } C) \ [\text{Range } T] \\
\equiv \\
\text{Select } * \text{ From } S \ [\text{Range } T] \text{ Where } C
\]

If the system uses a query evaluation strategy based on materializing the windows specified in a query, then filtering before applying the window instead of after is preferable, since it reduces steady-state memory overhead [80]. Note that the converse transformation might also be applied: We might prefer to move the filtering condition out of the window in order to allow the window to be shared by multiple queries with different selection conditions [80]. Finally note that filters and tuple-based windows generally do not commute.
2.6.9 Streams versus Relations

Our abstract semantics and therefore CQL distinguish two fundamental data types, relations and streams. However, having two separate data types is not required from an expressiveness standpoint. Informally, a stream can be encoded as a relation, and vice-versa. Essentially, the relation and stream types are isomorphic. Given a language $L$ (such as CQL) derived from our abstract semantics, we can derive a stream-only language $L_s$ with the same expressiveness as follows:

1. Corresponding to each $n$-ary relation-to-relation operator $O$ in $L$, there is an $n$-ary stream-to-stream operator $O_s$ in $L_s$. The semantics of $O_s(S_1, \ldots, S_n)$ when expressed in $L$ is $\text{Rstream}(O(S_1[\text{Now}], \ldots, S_n[\text{Now}])).$

2. Corresponding to each stream-to-relation operator $W$ in $L$, there is a unary stream-to-stream operator $W_s$ in $L_s$. The semantics of $S[W_s]$ when expressed in $L$ is $\text{Rstream}(S[W]).$

3. There are no operators in $L_s$ corresponding to relation-to-stream operators of $L$.

The only requirement for this derivation is that $L$ contain a basic relation-to-stream operator ($\text{Rstream}$) and a basic window operator ($\text{Now}$ window). $L$ and $L_s$ have essentially the same expressive power. Clearly any query in $L_s$ can be rewritten in $L$. Given a query $Q$ in $L$, we obtain a query $Q_s$ in $L_s$ by performing the following three steps. First, transform $Q$ to an equivalent query $Q'$ that has $\text{Rstream}$ as its only relation-to-stream operator (this step is always possible as indicated in Section 2.6.3). Second, replace every input relation $R_i$ in $Q'$ with $\text{Rstream}(R_i)$. Finally, replace every relation-to-relation and stream-to-relation operator in $Q$ with its $L_s$ equivalent according to the definitions above. As it turns out, the language $L_s$, derived from CQL is quite similar to the stream-to-stream approach taken in $\text{TelegraphCQ}$ [29].

We chose our dual approach over the stream-only approach for at least three reasons:

1. Reiterating Goal #1 from Section 2.1, we wanted to exploit the wide body of understanding and work on the existing relational model to the extent possible.

2. Our experience with a large number of queries [104] suggests that the dual approach results in more intuitive queries than the stream-only approach. As illustrated in our
Linear Road examples (Section 2.6.5), even applications with purely stream-based input and output specifications may include fundamentally relational components.

3. Having both relations and streams cleanly generalizes materialized views, as discussed in Section 2.6.10.

Recall from Section 2.2 that the Chronicle Data Model takes a similar approach, since it has both streams (chronicles) and relations.

2.6.10 Comparison with Other Languages

Now that we have presented CQL we can provide a more detailed comparison against some of the related languages for continuous queries over streams and relations that were discussed briefly in Section 2.2. Specifically, we show that basic CQL (without user-defined functions, aggregates, or window operators) is strictly more expressive than Tapestry [107], Tribeca [105], GSQL [37], and materialized views over relations with or without chronicles [66]. We also discuss Aurora [28], although it is difficult to compare CQL against Aurora because of Aurora’s graphical, procedural nature. We do not discuss the TelegraphCQ [29] query language here since we discussed it in the previous section.

Views and Chronicles

Any conventional materialized view defined using an SQL query $Q$ can be expressed in CQL using the same query $Q$ with CQL semantics.

The Chronicle Data Model (CDM) [66] defines chronicles, relations, and persistent views, which are equivalent to streams, base relations, and derived relations in our terminology. For consistency, we use our terminology instead of theirs. CDM supports two classes of operators based on relational algebra, both of which can be expressed in CQL. The first class takes streams and (optionally) base relations as input and and produces streams as output. Each operator in this class can be expressed equivalently in CQL by applying a Now window on each input stream, translating the relational algebra operator to SQL, and applying an Rstream operator to produce a streamed result. For example, the join query $S_1 \bowtie S_1.A = S_2.B$ $S_2$ in CDM is equivalent to the CQL query:

Select Rstream(*)
From S1 [Now], S2 [Now]
Where \( S1.A = S2.B \)

The second class of operators takes a stream as input and produces a derived relation as output. These operators can be expressed in CQL by applying an Unbounded window on the input stream and translating the relational algebra operator to SQL.

The operators in CDM are strictly less expressive than CQL. CDM does not support sliding windows over streams, although it has implicit Now and Unbounded windows as described above. Furthermore, CDM distinguishes between base relations, which can be joined with streams, and derived relations (persistent views), which cannot. These restrictions ensure that derived relations in CDM can be maintained incrementally in time logarithmic in the size of the derived relation.

**Tapestry**

Tapestry queries [107] are expressed using SQL syntax. At time \( \tau \), the result of a Tapestry query \( Q \) contains the set of tuples logically obtained by executing \( Q \) as a relational SQL query at every instant \( \tau' \leq \tau \) and taking the set-union of the results. This semantics for \( Q \) is equivalent to the CQL query:

\[
\text{Select Istream(Distinct *)} \\
\text{From (Istream(Q)) [Range Unbounded]}
\]

Tapestry does not support sliding windows over streams or any relation-to-stream operators.

**Tribeca**

Tribeca is based on a set of stream-to-stream operators and we can show that all of the Tribeca operators specified in reference [105] can be expressed in CQL. We do not provide the details since they are straightforward but lengthy. Two of the more interesting operators are \( \text{demux} \) (demultiplex) and \( \text{mux} \) (multiplex). In a Tribeca query, the \( \text{demux} \) operator is used to split a single stream into substreams, the substreams are processed separately using other (stream-to-stream) operators, then the resulting substreams are merged into a single result stream using the \( \text{mux} \) operator. This type of query is expressed in CQL using a combination a sliding window (partitioned or time-based) and the Group By clause. Section 2.6.7 contains examples of such queries.
Like chronicles and Tapestry, Tribeca is strictly less expressive than CQL. Tribeca queries take a single stream as input and produce a single stream as output, with no notion of relation. CQL queries can have multiple input streams and can freely mix streams and relations.

**Gigascope**

GSQL is a SQL-like query language developed for the Gigascope DSM for network monitoring [37]. GSQL, as originally proposed by Cranor *et al.* [37], is a one-time query language over finite data streams, not a continuous one. Here we present a comparison between CQL and a "continuous" version of GSQL.

GSQL's primary operators are selection, join, aggregation, and merge (union). Constraints on join and aggregation ensure that they are nonblocking: a join operator must contain a predicate involving an "ordered" attribute from each of the joining streams, and an aggregation operator must have at least one grouping attribute that is ordered. (Ordered attributes are generalizations of CQL timestamps.)

When the ordered attributes correspond exactly to timestamps, all the four primary operators of GSQL can be expressed in CQL: Selection is straightforward. The GSQL merge operator can be expressed using `Union` in CQL. The GSQL join operator can be expressed using a sliding-window join and an `Istream` in CQL. Finally, although it is nontrivial to express GSQL aggregation in CQL (requiring grouping and aggregation, projection, and join), it always is expressible.

**Aurora**

Aurora queries are built from a set of seven operator types [1]. Operators are composed by users into a global query execution plan via a "boxes-and-arrows" graphical interface. It is somewhat difficult to compare the procedural query interface of Aurora against a declarative language like CQL, but we can draw some distinctions.

The aggregation operator of Aurora can be defined using an user-defined function, yielding nearly unlimited expressive power. The aggregation operator also has optional parameters set by the user. For example, these parameters can direct the operator to take certain action if no stream elements have arrived for $T$ wall-clock seconds, making the semantics dependent on stream arrival and processing rates.


<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_P(R)$</td>
<td>Selection with predicate $P$</td>
</tr>
<tr>
<td>$\pi_L(R)$</td>
<td>Duplicate eliminating projection of attributes $L$</td>
</tr>
<tr>
<td>$\hat{\pi}_L(R)$</td>
<td>Duplicate preserving projection of attributes $L$</td>
</tr>
<tr>
<td>$R_1 \times \cdots \times R_n$</td>
<td>Cross product</td>
</tr>
<tr>
<td>$R_1 \bowtie_A R_2$</td>
<td>Natural join of $R_1$ and $R_2$ on attribute $A$</td>
</tr>
<tr>
<td>$R_1 \cup R_2$</td>
<td>Union</td>
</tr>
<tr>
<td>$R_1 \cap R_2$</td>
<td>Intersection</td>
</tr>
<tr>
<td>$R_1 - R_2$</td>
<td>Difference</td>
</tr>
<tr>
<td>$G^G_F(R)$</td>
<td>Group by attributes $G$ and aggregate over functions in $F$</td>
</tr>
<tr>
<td>$\mathcal{G}_F(R)$</td>
<td>Aggregate over functions in $F$ without grouping</td>
</tr>
</tbody>
</table>

Table 2.2: Relation-to-relation operators in ACO

All operators in Aurora are stream-to-stream, and Aurora does not explicitly support relations. In order to express CQL queries involving derived relations and relation-to-relation operators, Aurora has to procedurally manipulate state corresponding to a derived relation.

### 2.7 An Algebra of Continuous Operators

This section presents another instantiation of our abstract semantics that we use for presentation in the rest of the thesis. This instantiation is based on an algebra of operators, not textual queries like CQL; query expressions in this algebra are therefore more concise. We call this algebra ACO for *Algebra of Continuous Operators*. The main difference between ACO and CQL is that ACO uses relational algebra for its relation-to-relation operators instead of SQL. The stream-to-relation and relation-to-stream operators are unchanged, but we introduce new, more compact notation for them. As part of ACO, we also present shorthands for common stream-to-stream operations.
### 2.7.1 Relation-to-Relation Operators

Table 2.2 lists the relation-to-relation operators in ACO. These operators are derived from traditional relational algebra, and their semantics is a straightforward extension of their traditional semantics as described in the beginning of Section 2.6. As a simple example, the expression $\text{AG}_{\text{SUM}(B)}(\sigma_{C>10}(R))$ is equivalent to the CQL query:

```
Select A, Sum (B)
From R
Where C > 10
Group By A
```

### 2.7.2 Stream-to-Relation and Relation-to-Stream Operators

Table 2.3 lists the stream-to-relation (window) and relation-to-stream operators. Here $S$ denotes an arbitrary stream, $R$, an arbitrary relation, type S2R, a stream-to-relation operator and type R2S, a relation-to-stream operator. The stream-to-relation and relation-to-relation operators of ACO are identical to those of CQL, except for their more compact notation.

A window operator over stream $S$ is denoted by following $S$ with the window size within a square parenthesis. For time-based windows, we subscript the parenthesis with the letter $T$, while tuple-based windows carry no subscripts. For example, $S[5]$ denotes a tuple-based window of the last 5 tuples, while $S[5]_T$ denotes a time-based window containing tuples that arrived in the last 5 time units. We use the notation $S[1]_T$ and $S[\infty]$ to

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S[N]$</td>
<td>S2R</td>
<td>Tuple-based sliding window of width $N$ tuples</td>
</tr>
<tr>
<td>$S[W]_T$</td>
<td>S2R</td>
<td>Time-based sliding window of width $W$ time units</td>
</tr>
<tr>
<td>$S[1]_T$</td>
<td>S2R</td>
<td>Now window</td>
</tr>
<tr>
<td>$S[\infty]$</td>
<td>S2R</td>
<td>Unbounded window</td>
</tr>
<tr>
<td>$TS(R)$</td>
<td>R2S</td>
<td>Istream</td>
</tr>
<tr>
<td>$DS(R)$</td>
<td>R2S</td>
<td>Dstream</td>
</tr>
<tr>
<td>$RS(R)$</td>
<td>R2S</td>
<td>Rstream</td>
</tr>
</tbody>
</table>

Table 2.3: Stream-to-relation and relation-to-stream operators in ACO
Table 2.4: Shortcuts for relational operators over streams.

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_P(S) )</td>
<td>( IS(\sigma_P(S[\infty])) )</td>
</tr>
<tr>
<td>( \pi_L(S) )</td>
<td>( IS(\pi_L(S[\infty])) )</td>
</tr>
<tr>
<td>( \hat{\pi}_L(S) )</td>
<td>( IS(\hat{\pi}_L(S[\infty])) )</td>
</tr>
<tr>
<td>( S_1 \times \cdots \times S_n )</td>
<td>( IS(S_1[\infty] \times \cdots \times S_n[\infty]) )</td>
</tr>
<tr>
<td>( S_1 \bowtie_A S_2 )</td>
<td>( IS(S_1[\infty] \bowtie_A S_2[\infty]) )</td>
</tr>
<tr>
<td>( S_1 \cup S_2 )</td>
<td>( IS(S_1[\infty] \cup S_2[\infty]) )</td>
</tr>
<tr>
<td>( S_1 \cap S_2 )</td>
<td>( IS(S_1[\infty] \cap S_2[\infty]) )</td>
</tr>
<tr>
<td>( S_1 - S_2 )</td>
<td>( S_1[\infty] - S_2[\infty] )</td>
</tr>
<tr>
<td>( G_{\mathcal{F}}(S) )</td>
<td>( G_{\mathcal{F}}(S[\infty]) )</td>
</tr>
</tbody>
</table>

denote \texttt{Now} window and \texttt{Unbounded} windows, respectively; note that this notation is consistent with their definitions. We did not include notation for partitioned-windows since they are not used in the remainder of the thesis.

Finally, we use \( IS(R) \), \( DS(R) \), and \( RS(R) \) to denote an \texttt{Istream}, \texttt{Dstream}, and \texttt{Rstream} operator over relation \( R \), respectively.

### 2.7.3 Shorthand Notation for Relational Operations over Streams

We now introduce some shortcuts to facilitate expressing relational operations such as filters, joins, and aggregations over streams. These shortcuts can be thought of as macros that expand to some appropriate combination of stream-to-relation, relation-to-relation, and relation-to-stream operators. Table 2.4 lists the shortcuts and their expansions. Each shortcut is derived from one relation-to-relation operator in Table 2.2. When expanded, the shortcut applies an unbounded window to each input stream converting them to relations and applies the relation-to-relation operator over the resulting relations. Whenever the output of this operator is monotonic (which is the case for all operators except group-by), an \( IS \) operator is applied to the output relation producing a stream.

**Example 2.15** Consider two streams \( S_1(A, B) \) and \( S_2(A, C) \), and consider the operation of merging these two streams after projecting each of them on attribute \( A \). Expressing this operation in regular ACO (without shortcuts) requires two \texttt{Unbounded} windows and one
Istream operator:
\[ IS(\hat{\pi}_A(S_1[\infty]) \cup \hat{\pi}_A(S_2[\infty])) \]

Using the shorthand notation, we can express this operation more concisely as:
\[ \hat{\pi}_A(S_1) \cup \hat{\pi}_A(S_2) \]

In CQL, this operation can be expressed as:

```
Select A From S1
Union
Select A From S2
```

Note that the CQL query itself uses defaults introduced in Section 2.6.4.

2.8 Time Management

Recall from Sections 2.4 and 2.5 that our abstract semantics assumes a discrete, ordered time domain \( T \). Specifically, our continuous semantics is based on time logically advancing within domain \( T \). Conceptually, at time \( \tau \in T \) all inputs up to \( \tau \) are processed and the output corresponding to \( \tau \) (stream elements with timestamp \( \tau \) or the instantaneous relation at time \( \tau \)) is produced. In this section we briefly discuss how a DSMS might implement this semantics under realistic conditions. The topic is covered in much more depth by Srivastava and Widom [98].

For exposition in the remainder of this section, let us assume that relations are updated via timestamped relational update requests that arrive on a stream. Thus, without loss of generality, we can focus on streams only. For a DSMS to produce output corresponding to a time \( \tau \in T \), it must have processed all input stream elements at least through \( \tau \). In other words, it must know at some “real” (wall-clock) time \( t \) that no new input stream elements with timestamp \( \leq \tau \) will arrive after \( t \). Making this determination is straightforward when all of the input streams are producing elements continuously, and their elements arrive in timestamp order. However, in many stream applications (including the Linear Road), input streams may be generated by remote sources, the network conveying the stream elements to the DSMS may not guarantee in-order transmission, there may be timing skew across sources, and streams may pause and restart.
In the Stream prototype our approach is to assume additional “meta-input” to the system called heartbeats. A heartbeat consists simply of a timestamp $\tau \in T$, and has the semantics that after arrival of the heartbeat the system will receive no future stream elements with timestamp $\leq \tau$. There are various ways by which heartbeats may be generated. Here are three examples:

1. In the easiest and a fairly common case, timestamps are assigned using the DSMS clock when stream tuples arrive at the system. Therefore stream elements are ordered, and the clock itself provides the heartbeats.

2. The source of an input stream might generate source heartbeats, which indicate that no future elements in that stream will have timestamp less than or equal to that specified by the heartbeat. Source heartbeats are useful only if the heartbeats and the stream elements within a single input stream reach the DSMS in timestamp order.

3. Properties of stream sources and the system or networking environment may be used to generate heartbeats. For example, if we know that all sources of input streams use a global clock for timestamping and there is an upper bound $D$ in delay of stream elements reaching the DSMS, at every global time $t$ we can generate a heartbeat with timestamp $t - D$.

Heartbeats are also important internally within the Stream implementation of CQL, in order to communicate time-related information among different operators in a query plan. These and other details related to heartbeat generation can be found in reference [98].

2.9 Conclusion

In this chapter, we first presented an abstract semantics for continuous queries. The abstract semantics was based on two data types, streams and relations, and three black-box classes of operators over these types: stream-to-relation, relation-to-relation, and relation-to-stream. Concrete continuous query languages can be derived from our abstract semantics by instantiating the black boxes using specific operators. We also presented a formal denotational specification of our abstract semantics.

We then presented two instantiations of our abstract semantics. The first instantiation, called CQL, is based on the widely used SQL query language. CQL is one of the first
comprehensive languages for continuous queries, and it is the language supported by the STREAM prototype. Empirically, CQL is very expressive, and we have used it to express a wide variety of continuous query applications [104]. CQL also includes several syntactic shortcuts and defaults to simplify expressing common query constructs. Our second instantiation is based on relational algebra, and we use it mainly for presentation purposes in the rest of the thesis.

Finally, we briefly discussed some time-related issues that arise when implementing our languages, and described an approach based on heartbeats that we use in the STREAM prototype to handle these issues.
Chapter 3

Memory Requirements of Queries

Beginning with this chapter, we study various issues in executing continuous queries. This chapter focuses on understanding the memory requirement of queries as they execute, and it provides the setting for Chapter 4, which deals with trading required memory for answer accuracy. Work presented in this chapter appeared in references [7, 8].

3.1 Introduction

Most continuous queries are stateful, i.e., they require some storage (memory) over time for correct execution. This requirement stems from the fact that streams can be read only once from their source. If a query needs historical information about a stream, that information must be explicitly maintained by the query in its local store. The query cannot expect to get this information from the stream source.

As an example, consider the natural join query $S_1 \Join_A S_2$. Recall from Section 2.7.3 that this query uses implicit unbounded windows over $S_1$ and $S_2$. At any given point in time, this query needs to “remember” all the tuples of $S_1$ that have arrived so far, since these tuples could join with future tuples of $S_2$. By the same argument, the query needs to remember all tuples of $S_2$ as well. Not all queries require storing entire stream histories. For example, the simple filter $\sigma_{A=10}(S)$ requires no state\(^1\), since it can simply process one tuple at a time. As we will see shortly, even some join queries might require very little state\(^2\).

\(^1\)Other than for the tuple being processed.
\(^2\)We use the terms state, storage, and memory interchangeably in this chapter. In particular, note that
This chapter focuses on a precise, theoretical characterization of memory requirements of continuous queries. A good understanding of memory requirements not only provides us with memory-optimal algorithms for query evaluation, but it also enables us to explore alternatives such as approximate query answering for queries with provably high memory requirements.

We present two interesting results on memory requirements of queries. First, we prove that all queries we consider (queries involving selection, projection, joins, windows, and aggregation) fall into just two classes based on their asymptotic memory requirements: queries that can be evaluated with bounded memory (i.e., a finite amount of memory), and those that require memory that grows linearly with the input data. Therefore, a large part of this chapter focuses on distinguishing the class of queries that can be evaluated with bounded memory from those that cannot. Note that we are concerned with exact evaluation of queries in this chapter; there are many queries that can be approximately evaluated with logarithmic memory, as we will see in Chapter 4. On first thought, it might seem that only simple filter queries or queries with bounded windows over all participating streams can be evaluated with bounded memory. The second interesting result shown here is that there is a relatively large class of queries, including some queries that join an arbitrary number of streams with unbounded windows, that can be computed with bounded memory.

Informally, three kinds of operations—join, duplication elimination, and group-by aggregation—cause a query to require unbounded memory.

1. When we join two or more streams, we usually need to store the current window of tuples for each input stream. Therefore, if the window size for any input stream can grow in an unbounded fashion, the query usually requires unbounded memory. (Apart from unbounded windows, time-based windows and partitioned windows with an unbounded number of groups can grow in an unbounded fashion—we formalize this notion in Section 3.7.3.) However, there exist queries having windows of unbounded size that can be processed in bounded memory by exploiting some properties of the query predicate (as we will illustrate shortly in Section 3.1.1).

2. In order to perform duplicate elimination, we need to store the current output to remove future duplicates, so a query with duplicate elimination whose output size memory refers not only to main memory, but to any form of storage.
CHAPTER 3. MEMORY REQUIREMENTS OF QUERIES

is unbounded requires unbounded memory.

3. Finally, for group-by aggregation, we need to maintain the current aggregation value for every group in the output, so a group-by aggregation query with an unbounded number of output groups requires unbounded memory. Even if the number of output groups is bounded, a group-by aggregation query might require unbounded memory, if it involves a holistic aggregate, an aggregate that cannot be computed incrementally.

As we will show in Section 3.1.1, most of the difficulty in precisely characterizing whether a query is computable in bounded memory or not arises due to the subtle interactions among joins, duplicate elimination, and selections in a query. Therefore, we will first present our results for the well-known class of Select-Project-Join (SPJ) queries that contain just the above three operations (joins, duplicate elimination, and selections). In our algebra ACO, an SPJ query is of the form $\pi_L(\sigma_P(S_1 \times \cdots \times S_n))$ (duplicate preserving) or of the form $\dot{\pi}_L(\sigma_P(S_1 \times \cdots \times S_n))$ (duplicate eliminating), where each stream has a default Unbounded window. We will then extend our results to queries involving grouping and aggregation, and those involving explicit windows.

3.1.1 Examples

Our set of example queries is shown in Table 3.1. All of the example queries are SPJ queries. Two data streams, $S(A, B, C)$ and $T(D, E)$, are used in the example queries. The domain of attributes $A$–$E$ is the set of integers. We use $\Pi$ as a place-holder for either one of $\pi$ and $\dot{\pi}$. We will formalize the queries addressed in this chapter in Section 3.3.

Consider Query $Q_1$, $\Pi_A(\sigma_{A > 10}(S))$, a selection and projection over one data stream. When the projection is duplicate-preserving ($\Pi = \pi$), $Q_1$ is a simple filter on $S$ and can be evaluated by tuple-at-a-time processing of the stream. Thus, it can always be evaluated without using any extra memory for storage of stream tuples or intermediate state. If the projection in $Q_1$ is duplicate-eliminating ($\Pi = \dot{\pi}$), we need to keep track of each distinct value of $A$ greater than 10 in $S$ so far, in order to eliminate duplicates in the answer. In this case, there is no finite bound on the amount of memory required for evaluating this query over all possible instances of Stream $S$. Query $Q_2$, $\Pi_A(\sigma_{A = D}(S \times T))$, is an equi-join over streams $S$ and $T$. In order to correctly evaluate Query $Q_2$ it is necessary to store every
CHAPTER 3. MEMORY REQUIREMENTS OF QUERIES

56

Bounded-memory computable?

<table>
<thead>
<tr>
<th>Query</th>
<th>Bounded-memory computable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi_A(\sigma(A &gt; 10) \ (S))$</td>
<td>$\Pi = \hat{\pi}$ Yes $\Pi = \pi$ No</td>
</tr>
<tr>
<td>$\Pi_A(\sigma(A = D) \ (S \times T))$</td>
<td>No No</td>
</tr>
<tr>
<td>$\Pi_A(\sigma(A = D) \land (A &gt; 10) \land (D &lt; 20) \ (S \times T))$</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>$\Pi_A(\sigma(B &lt; D) \land (A = 10) \ (S \times T))$</td>
<td>No Yes</td>
</tr>
<tr>
<td>$\Pi_A(\sigma(B &lt; D) \land (C &lt; E) \land (A = 10) \ (S \times T))$</td>
<td>No No</td>
</tr>
<tr>
<td>$\Pi_A(\sigma(B &lt; D) \land (B &gt; 10) \land (B &lt; 20) \ (S \times T))$</td>
<td>No Yes</td>
</tr>
<tr>
<td>$\Pi_A(\sigma(B &lt; D) \land (A = 10) \ (S \times T))$</td>
<td>Yes Yes</td>
</tr>
</tbody>
</table>

Table 3.1: Bounded-memory computability of example queries over data streams $S(A, B, C)$ and $T(D, E)$.

distinct value of $A$ seen so far in Stream $S$ and every distinct value of $D$ seen so far in Stream $T$, which requires unbounded space.

Query $Q_3$, $\Pi_A(\sigma(A = D) \land (A > 10) \land (D < 20) \ (S \times T))$, is similar to Query $Q_2$ but has two additional selection predicates on attributes $A$ and $D$. Observe that a tuple of stream $S$ can join with a tuple of stream $T$ only if their corresponding $A$ and $D$ values lie within the interval $[11, 19]$; this observation can be used to evaluate Query $Q_3$ in bounded memory.

We briefly describe an evaluation strategy for computing $Q_3$ in bounded memory. The evaluation strategies that we describe for bounded-memory evaluation of queries involve keeping constant-sized synopses for each stream $S$ and $T$. A synopsis for a stream is a summary of the tuples of the stream seen so far that contains sufficient state information to compute future answers correctly. For the duplicate-preserving case of $Q_3$, the synopsis for $S$ contains, for each value $v$ in the interval $[11, 19]$, the count of tuples seen so far with $A = v$. Similarly, the synopsis for $T$ contains for each value $v \in [11, 19]$ the count of tuples with $D = v$. It is easy to see that the synopses above are sufficient to compute $Q_3$ correctly.

For example, consider a new tuple arriving on stream $S$ with $A = v$. If $v$ does not lie in the interval $[11, 19]$, then the new tuple cannot join with any tuple of $T$ and it can be ignored. If $v$ lies in $[11, 19]$, the exact number of past $T$ tuples that the new tuple joins with is stored in

\footnote{If $\Pi = \pi$, we only need to store every distinct value of $D$ that has not participated in a join so far.}

\footnote{This strategy uses bounded memory only if we assume that counts can be stored in bounded space. We revisit this assumption in Section 3.2}
the synopsis for $T$, and that many copies of the $(v)$ tuple are generated in the output. Note that it was crucial that the output project list of $Q_3$ only has attribute $A$ and not, for instance, $A$ and $C$, which would have made the query not computable in bounded memory. A similar evaluation strategy can be designed to compute $Q_3$ in the duplicate-eliminating case.

Consider the series of queries $Q_4, Q_5, Q_6$ for duplicate-eliminating projection. Note that $Q_5$ is derived from $Q_4$ by adding an additional predicate $(C < E)$ and $Q_6$ is derived from $Q_5$ by adding two additional predicates $(B < E)$ and $(C < D)$. While $Q_4$ and $Q_6$ are computable in bounded memory, $Q_5$ is not. Query $Q_4$ can be evaluated by maintaining as synopsis of $S$ the minimum value of attribute $B$ among all tuples of $S$ (so far) which have $A = 10$, and maintaining as synopsis of $T$, the maximum value of attribute $D$ among all tuples of $T$ so far. In order to see why the above synopses are sufficient, consider the arrival of a new tuple $t$ on Stream $T$. Assume $t$ joins with some past tuple $s$ of $S$ with $A = 10$. From the join condition it follows that $s[B] < t[D]$. Clearly, $t$ also joins with that tuple of $S$ that has the minimum value of attribute $B$ among all tuples with $A = 10$. Therefore, we can determine if $t$ joins with some tuple of $S$ by just checking if $t[D]$ is strictly larger than the value stored in the synopsis for $S$. Similarly, $Q_6$ can be evaluated by maintaining as synopsis of $S$ the value $\min\{\max\{s[B], s[C]\}\}$ over all tuples $s$ of $S$ so far, and maintaining the value $\max\{\min\{t[D], t[E]\}\}$ over all tuples $t$ of $T$ so far. We leave it to the reader to verify that the above synopses are sufficient to correctly evaluate $Q_6$.

None of the queries $Q_4, Q_5, Q_6$ can be computed in bounded memory for duplicate-preserving projection. Note that for duplicate-preserving projection, it is not sufficient to just know whether or not a new stream tuple joins with any past tuples—we also need to determine the exact number of past tuples with which it joins. One can easily verify that the synopses described earlier for queries $Q_4$ and $Q_6$ for duplicate-eliminating projection cannot be used to determine the exact number of past tuples with which a new stream tuple joins.

3.1.2 Contributions

As the examples of the previous section suggest, the problem of determining the bounded-memory computability of continuous queries is nontrivial. To summarize, we make the following contributions in this chapter:
1. We consider continuous queries involving selection, projection, join, group-by and aggregation, and sliding window, and we specify an algorithm that determines whether or not any given query can be evaluated using a bounded amount of memory for all possible instances of input data streams. Broadly, the class of queries that we consider corresponds to the class of single-block CQL queries, although, for presentation purposes, we will mostly use our algebra, ACO, that we introduced in Section 2.7 of the previous chapter to represent queries.

2. When a query can be evaluated using bounded memory, we produce an execution strategy based on constant-sized synopses of the data streams, characterizing the memory requirements of the query for all possible instances of the streams.

3. When a query cannot be evaluated using bounded memory, for any query execution strategy we identify specific instances of input streams for which the strategy requires memory at least linear in the sum of lengths of the input streams.

### 3.1.3 Overview and Organization

We formally state the problem studied in this chapter in Section 3.2 and present some relevant notation and definitions in Section 3.3.

Sections 3.4, 3.5 and 3.6 deal with bounded-memory computability of Select-Project-Join (SPJ) queries without self-join. All the example queries used in Section 3.1.1 belong to this class. Since it seems hard to determine bounded-memory computability of arbitrary SPJ queries directly (as the examples suggest), we take an indirect approach. We first rewrite a given SPJ query \( Q \) as a union of queries each of which belongs to a special class that we call *Locally Totally Ordered* queries, or LTO queries for short. Next, we check if each LTO query in the union is bounded-memory computable. An LTO query has a special structure that makes it easier to determine if it is bounded-memory computable. The original query \( Q \) is bounded-memory computable if and only if all the LTO queries in the union are bounded-memory computable. This LTO-rewriting approach to determining bounded-memory computability of SPJ queries is presented in Section 3.4.

The basic LTO-rewriting approach to determining bounded-memory computability requires time exponential in the size of the query. Section 3.5 refines this approach and presents an efficient polynomial-time algorithm to determine if an SPJ query is computable.
in bounded-memory. Section 3.6 presents an execution strategy for bounded-memory computable queries.

Finally, Section 3.7 extends our results to a larger class of queries, namely, SPJ queries that may include self-joins and an optional aggregation, and queries with sliding windows. We do not include a separate section on related work, since we know of no work related specifically to this chapter; please refer to Section 1.3 for a discussion of work broadly related to the entire thesis.

3.2 Problem Statement

As we mentioned in Section 3.1, the worst-case memory requirement of all the queries that we consider falls into two asymptotic classes: linear in the size of the input, and bounded by a constant. Therefore, most of this chapter deals with the following problem: Given a query $Q$ determine if it is computable using a bounded amount of memory or not. As part of this characterization we will prove that a query that is not bounded-memory computable requires linear space. The formal definition of a bounded-memory computable query follows.

**Definition 3.1 (Bounded-Memory Computability)** A query is computable in bounded memory if there exists a constant $M$ and an algorithm that evaluates the query using fewer than $M$ units of memory for all possible instances of the input streams of the query.

We assume that one unit of memory can store one attribute value or a count. This assumption is slightly imprecise since the number of bits required to represent a count grows logarithmically with the number being counted. In practice, this imprecision is not critical, since no count is likely to require more than one or two words of memory on a modern computer architecture.

We will mostly concentrate on determining bounded-memory computability of Select-Project-Join (SPJ) queries. This restriction is mainly for exposition, and we will extend our results for SPJ queries to more general query classes in Section 3.7. An SPJ query is of the form $\Pi_L(\sigma_P(S_1 \times S_2 \times \ldots \times S_n))$, where the $\Pi$ symbol stands for either a duplicate-eliminating projection ($\pi$) or a duplicate-preserving projection ($\dot\pi$), $L$ is the list of projected attributes, $P$ the selection predicate, and $S_1, \ldots, S_n$ the input streams. Note that the general form of SPJ queries above uses shortcuts introduced in Section 2.7.3 for applying relational
operations directly to streams. The shortcuts hide an implicit unbounded window over the input streams, and an IS operator over the projection. The expanded form of an SPJ query without shortcuts is $T S(\Pi_L(\sigma_P(S_1[\infty] \times S_2[\infty] \times \ldots \times S_n[\infty])))$. The CQL formulation of this class of queries when $\Pi = \pi$ is:

```plaintext
Select distinct L
From S1, ..., Sn
Where P
```

When $\Pi = \dot{\pi}$, the CQL formulation is identical except that the distinct keyword is absent.

We restrict ourselves to SPJ queries where the selection predicate $P$ is a conjunction of atomic predicates. An atomic predicate is of the form $(S_i.A Op S_j.B)$ (where $i$ and $j$ can be the same or different) or of the form $(S_i.A Op k)$, where $Op$ is one of the comparison operators in $\{<, =, >\}$ and $k$ is some constant. Our results can be extended in a straightforward way to include the operators $\{\leq, \geq\}$. An atomic predicate of the form $(S_i.A Op S_i.B)$ or $(S_i.A Op k)$ involving attributes of just one stream is called a filter; otherwise, it is called a join. To simplify the presentation, we start off assuming that there are no self-joins in the query, i.e., $S_i \neq S_j$ for $i \neq j$. In Section 3.7 we extend our results to include self-joins. Further, we assume that the domain of all the attributes and constants is the set of integers: $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

### 3.3 Notation and Definitions

This section introduces notation and terminology and reviews some basic concepts from discrete mathematics that are used in our results.

As described in Section 3.2, we initially consider SPJ queries of the form $\Pi_L(\sigma_P(S_1 \times S_2 \times \ldots \times S_n)$, where $\Pi = \pi$ for duplicate-eliminating projection and $\Pi = \dot{\pi}$ for duplicate-preserving projection. When the streams and the list of projected attributes are not important to the discussion, we may write a query $Q$ as $Q(P)$, where $P$ is the selection predicate. This notation is also used to represent two queries that are identical except for their selection predicate. For example, query $Q(P_2)$ is obtained from query $Q(P_1)$ by just replacing the selection predicate $P_1$ with the predicate $P_2$. For convenience, we represent the selection predicate as the set of its conjuncts.

A set of atomic predicates $P$ is satisfiable if there exists some assignment of integer values to the attributes in $P$ that makes every predicate in the set $P$ evaluate to true. For
example, the set of atomic predicates \{((A < B), (B < C), (C < A))\} is not satisfiable. Note that although the general problem of boolean expression satisfiability is intractable, there exist efficient algorithms to check satisfiability for a conjunction (and by our convention, a set) of atomic predicates [110]. Observe that any query \(Q(P)\) with an unsatisfiable selection predicate \(P\) has an empty output stream and therefore is trivially computable in bounded memory. In the rest of the chapter we assume that the selection predicates of the queries considered are satisfiable unless mentioned otherwise.

We use the term query entity (or simply an entity) to refer to either a constant integer value or an attribute of a stream. Table 3.2 lists various notation related to constants, attributes, and query entities that we use in this chapter. For example, for Query \(Q_3\) in Table 3.1, \(S(Q_3) = \{S, T\}, C(Q_3) = \{10, 20\}, E(Q_3) = \{10, 20, A, B, C, D, E\}\).

The transitive closure of a set of atomic predicates \(P\), denoted \(P^+\), is the set of atomic predicates logically implied by the predicates in \(P\). The linear ordering of integers is implicitly used in determining transitive closure. For example, the transitive closure of the set of atomic predicates \{((A < 5), (8 < B))\} is the set \{((A < 5), (A < 8), (5 < B), (8 < B), (A < B), (A = A), (B = B), (5 = 5), (8 = 8))\}. We assume that the transitive closure \(P^+\) only contains atomic predicates involving entities occurring in \(P\). Continuing the previous example, the predicate \((A < 6)\) does not occur in the transitive closure although it is logically implied by the set of predicates \{((A < 5), (8 < B))\}. This assumption ensures that the transitive closure of a finite set of atomic predicates is also finite. Note that two queries \(Q(P)\) and \(Q(P')\) are equivalent whenever \(P^+ = (P')^+\).

**Definition 3.2** An inequality predicate \((e_1 < e_2)\) \(\in P\) is said to be redundant in \(P\) if one of the following three conditions hold: (1) there exists an entity \(e\) such that \((e_1 < e)\) \(\in P^+\) and \((e < e_2)\) \(\in P^+\); (2) there exists an integer constant \(k\) such that \((e_1 = k)\) \(\in P^+\) and \((k < e_2)\) \(\in P^+\); (3) there exists an integer constant \(k\) such that \((e_1 < k)\) \(\in P^+\) and \((e_2 = k)\) \(\in P^+\). 

\[\square\]

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S(Q))</td>
<td>set of streams that appear in (Q)</td>
</tr>
<tr>
<td>(C(Q))</td>
<td>set of constants that appear in (Q)</td>
</tr>
<tr>
<td>(A(S))</td>
<td>set of attributes in Stream (S)</td>
</tr>
<tr>
<td>(A(Q))</td>
<td>set of attributes in all streams in (Q), i.e., (\bigcup_{S \in S(Q)} A(S))</td>
</tr>
<tr>
<td>(E(Q))</td>
<td>set of entities in (Q), i.e., (A(Q) \cup C(Q))</td>
</tr>
</tbody>
</table>

Table 3.2: Notation related to a query \(Q\) and stream \(S\)
For example, the predicate \((A < C)\) is redundant for both the sets \(P = \{(A < B), (B < C), (A < C)\}\) and \(P = \{(A < C), (A = 5), (C > 5)\}\). Removing all the redundant predicates from \(P\) leaves its transitive closure unchanged, and therefore it is sufficient to consider only nonredundant predicates of a query in determining its bounded-memory computability. The converse is not true however: any atomic predicate whose removal leaves the transitive closure unchanged is not necessarily redundant according to our definition. For example, none of the predicates in the set \(\{(A = B), (A < C), (B < C)\}\) are redundant, while removing the predicate \((A < C)\) still leaves the transitive closure unchanged.

Our definition of redundancy depends on constant values in a subtle way: for example, the predicate \((A < C)\) is redundant for the set \(P = \{(A = 5), (5 < C), (A < C)\}\), while it is not so for the set \(P = \{(A = B), (B < C), (A < C)\}\). This subtlety arises because evaluating filter conditions requires no additional memory (see Theorem 3.1); exact details should become clearer from our characterization of bounded-memory computable queries in Section 3.4.

**Definition 3.3** For a given set of atomic predicates \(P\), the set of atomic predicates induced by a set of entities \(E\), denoted \(\text{IND}(P, E)\), is the set of predicates in \(P^+\) that involve only entities in \(E\).

**Definition 3.4** A set of entities \(E\) is totally ordered by a set of predicates \(P\) if for any two entities \(e_1\) and \(e_2\) in \(E\), exactly one of the three atomic predicates \((e_1 < e_2)\) or \((e_1 = e_2)\) or \((e_1 > e_2)\) is in \(P^+\).

**Definition 3.5** For a given set of predicates \(P\), the equality predicates in the set partition the entities in \(P\) into equivalence classes: two entities \(e_1\) and \(e_2\) belong to the same equivalence class if \((e_1 = e_2)\) is in \(P^+\).

**Definition 3.6** Given a set of predicates \(P\), an attribute \(A\) is lower-bounded if there exists an atomic predicate \((A > k)\) is in \(P^+\), or an atomic predicate \((A = k)\) is in \(P^+\) for some constant \(k\). Similarly, an attribute \(A\) is upper-bounded by \(P\) if there exists an atomic predicate \((A < k)\) is in \(P^+\), or an atomic predicate \((A = k)\). An attribute is bounded if it is both upper-bounded and lower-bounded, and unbounded otherwise.

Note that an attribute that is upper-bounded (resp. lower-bounded) but not lower-bounded (resp. upper-bounded) is unbounded by our definition.
Definitions 3.2–3.6, when used in the context of a query $Q(P)$, implicitly refer to the selection condition $P$ of the query. For example, the term “bounded attributes of a query” refers to the attributes of the query that are bounded given the selection predicate of the query.

### 3.4 Bounded-Memory Computability of SPJ Queries

This section provides a characterization for bounded-memory computable SPJ queries.

In order to determine the bounded-memory computability of SPJ queries, we use a special class of queries that we call *Locally Totally Ordered* queries, or LTO queries for short. Informally, an LTO query imposes, for each stream in the query, a total ordering of the attributes of the stream and the constants in the query. Any SPJ query can be converted into an LTO query by repeatedly adding filter conditions to its selection predicate; adding different sets of filter conditions results in different LTO queries. We show that an SPJ query is bounded-memory computable if and only if all the LTO queries derived from it by adding filter conditions are also bounded-memory computable. The special structure of LTO queries makes it easier to determine if they are bounded-memory computable.

**Definition 3.7 (Locally Totally Ordered Query)** An SPJ query $Q(P)$ is *Locally Totally Ordered* if, for every $S \in S(Q)$, the set of entities $(A(S) \cup C(Q))$ is totally ordered (Definition 3.4) by $P$. □

LTO queries are “maximal filter queries” in a certain sense: Adding additional filter conditions involving only $E(Q)$ to the selection predicate of an LTO query $Q$ either results in an equivalent query, or makes the selection predicate of the resulting query unsatisfiable. Consider an SPJ query $Q(P)$ over two streams, $S(A, B, C)$ and $T(D, E)$, where $P = \{(A = 10), (B < A), (A < C), (D < E)\}$. The set of entities $(A(S) \cup C(Q)) = \{A, B, C, 10\}$ is totally ordered since $B < A = 10 < C$; however, the set of entities $(A(T) \cup C(Q)) = \{D, E, 10\}$ is not totally ordered, since $D$ and $E$ are not comparable to $10$. Therefore, $Q(P)$ is not an LTO query. However, the query $Q(P \cup \{(E < A)\})$ formed by adding an additional predicate to $P$ is an LTO query, since now the set of entities $\{D, E, 10\}$ is totally ordered: $D < E < A = 10$.

We can form an LTO query from an SPJ query $Q(P)$ by adding filter conditions to the selection predicate $P$ to ensure that for each stream $S$ appearing in $Q$, the set $A(S) \cup C(Q)$ is
totally ordered. The LTO query formed in this way is said to be derived from Q. The formal
definition of derived LTO queries takes into account the equivalence of queries with the
same transitive closure of their selection predicates.

**Definition 3.8** An LTO Query Q(P_L) is said to be derived from an SPJ Query Q(P) if
(P_L)⁺ = (P ∪ F)⁺, where F is some arbitrary set of filter predicates.

For instance, from the example query Q_3 = Π_A(σ_{A = D} ∧ (A > 10) ∧ (D < 20) (S×T)) in Ta-
ble 3.1, we can derive an LTO query by adding the filter predicates \{((B > 20), (C < 10), (E = D))\} to the selection predicate.

The bounded-memory computability of an SPJ query is directly related to the bounded-
memory computability of the set of LTO queries derived from it, as formalized by the
following theorem.

**Theorem 3.1** An SPJ query Q(P) is bounded-memory computable if and only if all LTO queries
derived from it are bounded-memory computable.

**Proof:** Assume Q is bounded-memory computable. Then any query Q(P ∪ F), where
F is a set of filter conditions, is also bounded-memory computable. A straightforward
bounded-memory evaluation strategy for Q(P ∪ F) is as follows: Use a bounded-memory
evaluation strategy for Q, and check the additional filter conditions F on the output of Q,
which can be done without any additional memory. Since any LTO query derived from Q
is also formed by adding just filter conditions to P, it is bounded-memory computable as
well. This completes the “only-if” part of the proof.

Since all LTO queries derived from Q involve just the entities of Q, there is only a finite
number of such LTO queries, if we do not differentiate queries with the same transitive
closure of their selection predicates. Let Q_1, . . . , Q_m be an enumeration of all LTO queries
derived from Q. We claim that Q is equivalent to the union of the LTO queries derived from
it, i.e., Q = ∪_{1≤i≤m} Q_i. The union operator ∪ is duplicate-preserving if the projection
operator of Q is duplicate-preserving, and duplicate-eliminating if the projection of Q is
duplicate-eliminating. The following example illustrates this equivalence. Its formal proof
is generalized easily from the example and we do not present it.

Consider the SPJ query Q(P) = Π_A(σ_{B = C} ∧ (A = 10) (S×T)), involving two streams
S(A, B) and T(C). Three LTO queries Q_1, Q_2, and Q_3 are derived from Q(P); these are
formed by adding predicates \{(B < 10), (B = 10), and (B > 10)\}, respectively, to the selec-
tion predicate P. Consider a tuple s of stream S and a tuple t of stream T that successfully
join in Query $Q$, i.e., satisfy all the predicates in $P$. Then, depending on the whether the value of $s[B]$ ($= t[C]$) is less than, equal to, or greater than 10, tuples $s$ and $t$ join in exactly one of $Q_1$–$Q_3$, and fail to do so in the other two. Conversely, any two tuples $s$ of stream $S$ and $t$ of stream $T$ join in at most one of the queries $Q_1$–$Q_3$, and if they join in some query they also join in $Q$ (since the predicate of $Q$ is a subset of the predicate of any of $Q_1$–$Q_3$). Therefore, $Q$ is equivalent to the duplicate-preserving (when $\Pi = \hat{\pi}$) or duplicate-eliminating (when $\Pi = \pi$) union of $Q_1$–$Q_3$. Note that when $\Pi = \pi$, although each derived LTO query is duplicate-eliminating, the same output tuple may be produced by different LTO queries, and hence a duplicate-eliminating union is necessary to remove the duplicates from the outputs of these queries.

The equivalence $Q \equiv \bigcup_{1 \leq i \leq m} Q_i$ can be used to derive a bounded-memory evaluation strategy for $Q$, if all the LTO queries $Q_i \ (1 \leq i \leq m)$ are bounded-memory computable. The evaluation strategy is simpler when $Q$ has a duplicate-preserving projection: evaluate in bounded-memory each LTO query $Q_1, \ldots, Q_m$, and transfer any output of these queries into the output of $Q$. When $Q$ has a duplicate-eliminating projection, the evaluation strategy, as before, evaluates each LTO query independently. However, in addition, it remembers the set of all output tuples produced so far by any LTO query, and uses this set to remove duplicates. Remembering this set of output tuples does not require unbounded memory, since, as we will prove in Theorem 3.3, the number of output tuples of a bounded-memory computable LTO query with duplicate-eliminating projection is bounded. □

Theorem 3.1 reduces the problem of determining if an SPJ query is bounded-memory computable to that of determining if an LTO query is bounded-memory computable. The bounded-memory computability of an LTO query depends on two special sets of attributes identified by the selection predicate of the query.

**Definition 3.9** Consider a query $Q(P)$ and a stream $S_i \in S(Q)$. $\text{MaxRef}(S_i)$ is the set of all unbounded attributes (Definition 3.6) $A$ of $S_i$ that participate in a nonredundant (Definition 3.2) inequality join $(S_j.B < S_i.A), i \neq j$, in $P^+$. $\text{MinRef}(S_i)$ is similarly defined as the set of all unbounded attributes $A$ of $S_i$ that participate in a nonredundant inequality join of the form $(S_i.A < S_j.B), i \neq j$, in $P^+$. □

Note that $\text{MaxRef}$ and $\text{MinRef}$ of a stream depend only on the selection predicate of the query. In particular, they do not depend on the query being an LTO query.
Example 3.1 Figure 3.1 illustrates $\text{MaxRef}$ and $\text{MinRef}$ for an example set of predicates involving attributes from three streams $S_1$, $S_2$, and $S_3$. An inequality join predicate between two entities is represented in Figure 3.1 by a directed edge between the entities; for example, the edge from 10 to $E$ represents the atomic predicate $(10 < E)$. Attributes belonging to the same stream are indicated using an enclosing rectangle; for example, the attributes of $S_1$ are $A$, $B$, $C$, and $D$.

For stream $S_1$, $\text{MaxRef}(S_1) = \{C, A\}$, due to the predicates $(G < C)$ and $(H < A)$, and $\text{MinRef}(S_1) = \phi$. Attribute $B$ does not appear in $\text{MaxRef}(S_1)$ since the predicate $(G < B)$ is redundant. For stream $S_2$, $\text{MaxRef}(S_2) = \phi$ and $\text{MinRef}(S_2) = \{G\}$. Attribute $F$ does not appear in $\text{MaxRef}(S_2)$ since $F$ is bounded. For stream $S_3$, $\text{MaxRef}(S_3) = \phi$, and $\text{MinRef}(S_3) = \{H, J\}$. □

We consider the cases of LTO queries with duplicate-preserving projection and duplicate-eliminating projection separately.

3.4.1 LTO Queries with Duplicate-Preserving Projection

Duplicate-preserving LTO queries that involve only one stream can always be computed in bounded memory, since without joins every predicate is a filter and can be computed one tuple at a time. The following theorem characterizes bounded-memory computability for queries involving more than one stream.
Theorem 3.2 Let \( Q = \hat{\pi}_L(\sigma_P(S_1 \times \cdots \times S_n)) \) be an LTO query where \( n > 1 \). \( Q \) is bounded-memory computable (Definition 3.1) if and only if:

C1: every attribute in the project list \( L \) is bounded,

C2: for every equality join predicate \((S_i.A = S_j.B)\), where \( i \neq j \), \( S_i.A \) and \( S_j.B \) are both bounded, and

C3: \(|\text{MaxRef}(S_i)| = |\text{MinRef}(S_i)| = 0\) for \( i = 1, \ldots, n\).

The remainder of Section 3.4.1 informally discusses the ideas behind Theorem 3.2. We begin with a discussion of the theorem statement and then provide intuition for the “if” and “only-if” parts of the theorem proof. A formal proof for the “if” part can be derived in a straightforward manner from our informal discussion. A formal proof for the “only-if” part is relegated to Section 3.4.3, since, for presentation convenience, we combine it with the only-if proof of Theorem 3.3 (presented in Section 3.4.2).

Comments on Theorem Statement

Informally, conditions C1–C3 state that, if we ignore the attributes that are involved only in filter conditions (since we can handle filter conditions with no memory), all the attributes that influence the output of \( Q \) are bounded. Conditions C1 and C2 directly state that any attribute in the project list or any attribute involved in an equality join condition is bounded. For LTO queries, Condition C3 is equivalent to the statement that any nonredundant inequality join predicate involves only bounded attributes.

Lemma 3.1 Consider an LTO query \( Q(P) \) with a satisfiable \( P \). \(|\text{MaxRef}(S_i)| = |\text{MinRef}(S_i)| = 0\) for all \( S_i \in S(Q) \) if and only if for every nonredundant, non-equality join predicate \((S_j.A < S_k.B)\), attributes \( S_j.A \) and \( S_k.B \) are bounded.

Proof: The “if” part of the proof follows directly from the definition of \( \text{MaxRef} \) and \( \text{MinRef} \) (Definition 3.9).

For the “only-if” part of the proof, consider any nonredundant atomic predicate \((S_j.A < S_k.B)\) in \( P \). Clearly, at least one of \( S_j.A \) and \( S_k.B \) is bounded since otherwise \( \text{MinRef}(S_j) \) and \( \text{MaxRef}(S_k) \) would be nonempty. We show that it is impossible for exactly one of \( S_j.A \) and \( S_k.B \) to be bounded, and the other to be not bounded. Without loss of generality, assume that \( S_j.A \) is bounded and \( S_k.B \) is not; it follows that \((S_j.A > k_1) \in P^+\)
and \((S_j.A < k_2) \in P^+\) for some constant integers \(k_1\) and \(k_2\). Since \(S_k.B\) is not bounded and \(Q\) is an LTO query, either \((S_k.B < k_1) \in P^+\) or \((S_k.B > k_2) \in P^+\); the former implies that \(P\) is unsatisfiable, and the latter that \((S_i.A < S_k.B)\) is redundant; either case leads to a contradiction. □

“If” part

For the “if” part of Theorem 3.2, we describe a bounded-memory evaluation strategy for \(Q\), when conditions C1–C3 hold. The evaluation strategy uses the observation above that only bounded attributes influence the output of \(Q\), once the filter conditions are accounted for. The evaluation strategy maintains synopses for each of the \(n\) streams. The synopsis for \(S_i\) is conceptually the bag of tuples formed by projecting on the bounded attributes of \(S_i\) all the tuples of \(S_i\) seen so far that satisfy the filter conditions of \(S_i\). To keep memory bounded, the synopsis stores only distinct tuples and maintains the number of times each distinct tuple appears in the bag. The number of distinct tuples in the synopsis is bounded, since each projected attribute is bounded. When a new stream tuple \(s_i\) arrives on stream \(S_i\), the synopsis for \(S_i\) is updated; also, the synopses for other streams are used to determine any new output tuples resulting from the join of \(s_i\) with the earlier tuples of these streams. To update the synopsis for \(S_i\), tuple \(s_i\) is checked against all the filter conditions of \(S_i\) that appear in \(P^+\); if \(s_i\) satisfies all the filter conditions, its projection on the bounded attributes of \(S_i\) is computed and stored in the synopsis. It is straightforward to see that the synopses for \(n\) streams, maintained similarly, contain sufficient information to compute new output tuples resulting from the join of \(s_i\) with all earlier tuples of other streams, since all the nonredundant join conditions and projections involve only bounded attributes. The following example illustrates our evaluation strategy.

**Example 3.2** Consider \(Q(P) = \pi_A(\sigma(A < 20) \land (A = C) \land (C > 10) \land (B > 20) (S \times T))\), an LTO query over streams \(S(A, B)\) and \(T(C)\). The synopsis for stream \(S\) is formed by projecting onto attribute \(A\) the bag of tuples of \(S\) seen so far that satisfy all the filter conditions of \(S\), i.e., the tuples having a value strictly between 10 and 20 on their \(A\) attribute and a value greater than 20 on their \(B\) attribute. Note that we enforce the filter condition \((A > 10)\) although it occurs only in \(P^+\) and not in \(P\). The synopsis for \(S\) is therefore a bag of tuples over \(A\) with values between 10 and 20. This synopsis is compactly represented by storing, for
each value \( v \in [11, 19] \), the number of tuples in the synopsis with \( A = v \). Similarly, the synopsis for \( T \) contains for each \( v \in [11, 19] \), the count of tuples of \( T \) seen so far with \( C = v \). Consider the arrival of a new \( T \) tuple, \( t = \langle v \rangle \). If \( v \) lies outside the range \([11, 19]\), it does not join with any \( S \) tuple, and can be ignored; if \( v \) lies within the range it joins with all earlier \( S \) tuples having \( A = v \) and \( B > 20 \). By definition, these tuples are stored as part of the synopsis for \( S \). Therefore, the new output tuples resulting from the arrival of \( t \) can be determined using the synopsis for \( S \). Similarly, the new output tuples resulting from the arrival of a new \( S \) tuple can be determined from the synopsis for \( T \). □

“Only-If” part

Now consider the “only-if” part of Theorem 3.2. A formal proof is given in Section 3.4.3, where we combine it with the only-if proof of Theorem 3.3. Here we give intuition and examples. If some condition \( C_1–C_3 \) is not satisfied, some unbounded attribute influences the output of \( Q \), either as a projection attribute or as an attribute in a nonredundant join predicate. Thus, any evaluation algorithm for \( Q \) might be forced, in the worst case, to remember an unbounded number of values of this attribute, thus requiring unbounded memory. Example 3.3 illustrates how the existence of an unbounded attribute in the project list can force any evaluation algorithm to use unbounded space, and Example 3.4 illustrates the same for an unbounded attribute in a nonredundant join predicate.

**Example 3.3** Consider the LTO query \( Q = \pi_A (\sigma(A > 10) \land (B = C) \land (B = 10) \ (S \times T)) \) over streams \( S(A, B) \) and \( T(C) \). The projection attribute \( A \) is clearly not bounded. Consider an instant in the evaluation of \( Q \), where input tuples, \( \langle v_1, 10 \rangle, \ldots, \langle v_N, 10 \rangle \), such that each \( v_i > 10 \ (1 \leq i \leq N) \), have arrived on stream \( S \), and no tuple has arrived on stream \( T \). Any evaluation algorithm for \( Q \) has to remember all \( N \) values of \( A \), \( \{v_1, \ldots, v_N\} \) at this instant, since a future arrival of a \( \langle 10 \rangle \) tuple on stream \( T \) would require the algorithm to produce all the \( N \) values in the output. Since \( N \) can be made arbitrarily large, any evaluation algorithm requires unbounded memory in the worst-case. □

**Example 3.4** Consider the LTO query \( Q = \pi_A (\sigma(A = 10) \land (B < C) \land (B > 10) \land (C > 10) \ (S \times T)) \) over streams \( S(A, B) \) and \( T(C) \). The projection attribute \( A \) of this query is bounded; however, there exists a nonredundant inequality join predicate \( (B < C) \) involving unbounded attributes. Therefore, \( \text{MinRef}(S) = \{B\} \) and \( \text{MaxRef}(T) = \{C\} \) are both nonempty, violating Condition C3. We assert that \( Q \) cannot be computed in bounded memory. For the
sake of contradiction, suppose there exists an algorithm $A$ that can evaluate $Q$ with fewer than a constant $M$ units of memory. We will construct a class of input scenarios and show that Algorithm $A$ will fail to produce the correct output of $Q$ for at least one of the input scenarios.

Define two sets of $N$ tuples, $S = \{\langle 10, 11 \rangle, \langle 10, 12 \rangle, \ldots, \langle 10, 10+N \rangle \}$ and $T = \{\langle 12 \rangle, \langle 13 \rangle, \ldots, \langle 11+N \rangle \}$. Consider a class of input scenarios where some subset of tuples chosen from $S$ arrives on stream $S$ (the order in which the tuples arrive does not matter), followed by one tuple chosen from $T$ that arrives on stream $T$. For each input scenario, consider the instant when all the tuples of stream $S$ have arrived, but the single tuple of stream $T$ has not. Algorithm $A$ will be in some state at this instant; since Algorithm $A$ has fewer than $M$ units of memory, the number of distinct states that it can be in is finite. However, since there are $2^N$ subsets of $S$, for some sufficiently large $N$, there exist two distinct subsets of $S$ such Algorithm $A$ ends up in the same state after seeing the tuples of either subset. Let $S'$ and $S''$ be two such subsets, and let $\langle 10, k \rangle$ be the tuple with the smallest value of $k$ that is present in one of $S'$ or $S''$ but not in the other. Assume, without loss of generality, that $\langle 10, k \rangle \in S'$. Now consider two input scenarios from the class above: $S'$ followed by tuple $\langle k+1 \rangle$ on stream $T$, and $S''$ followed by the same tuple $\langle k+1 \rangle$. The output of $Q$ after the arrival of the $\langle k+1 \rangle$ tuple on $T$ differs for the two scenarios; the count of $\langle 10 \rangle$ tuples in the output for the former case, constructed from $S'$, is always one greater than the output for the latter case, constructed from $S''$. Since Algorithm $A$ is unable to distinguish between $S'$ and $S''$, it will give the same answer in both cases and at least one answer will be incorrect. □

### 3.4.2 LTO Queries with Duplicate-Eliminating Projection

This section presents a characterization of LTO queries with duplicate-eliminating projection.

**Theorem 3.3** Let $Q = \pi_L(\sigma_P(S_1 \times \cdots \times S_n))$ be an LTO query. $Q$ is bounded-memory computable (Definition 3.1) if and only if:

- **C1**: every attribute in the project list $L$ is bounded,
- **C2**: for every equality join predicate $(S_i.A = S_j.B)$, $i \neq j$, $S_i.A$ and $S_j.B$ are both bounded, and
- **C3**: $|\text{MaxRef}(S_i)|_{eq} + |\text{MinRef}(S_i)|_{eq} \leq 1$ for $i = 1, \ldots, n$. 


CHAPTER 3. MEMORY REQUIREMENTS OF QUERIES

In condition C3, \(|E|_{eq}\) denotes the number of equivalence classes into which the entity set \(E\) is partitioned by the set of predicates \(P\).

Theorem 3.3 differs from Theorem 3.2 in two respects. First, unlike Condition C3 of Theorem 3.2, Condition C3 of Theorem 3.3 allows each stream to have one non-bounded attribute (or a set of attributes belonging to the same equivalence class) participating in a nonredundant inequality join predicate. Second, Theorem 3.3 is applicable to all duplicate-eliminating \(L\) \(TO\) queries, including queries with just one stream, while Theorem 3.2 holds only for duplicate-preserving \(L\) \(TO\) queries involving more than one stream. (Duplicate-preserving \(L\) \(TO\) queries involving just one stream are always bounded-memory computable.)

As we did for the duplicate-preserving case, we only discuss the ideas behind Theorem 3.3 in this section; the formal proof for the “if” part of the theorem can be derived from the discussion, while the formal proof for the “only-if” case is presented in Section 3.4.3 along with the “only-if” proof of Theorem 3.2.

“If” part

For the “if” part of Theorem 3.3, we describe a bounded-memory evaluation strategy for \(Q\), when conditions C1–C3 are satisfied. As in the duplicate-preserving case, the evaluation algorithm maintains a synopsis for each stream summarizing the set of tuples seen so far in the stream; in addition, it also remembers the output tuples produced so far, to ensure duplicate elimination. Note that Condition C1 guarantees that the set of possible output tuples is bounded in size.

The synopsis for Stream \(S_i\) depends on the values of \(\text{MaxRef}(S_i)\) and \(\text{MinRef}(S_i)\). The simplest case occurs when both \(\text{MaxRef}(S_i)\) and \(\text{MinRef}(S_i)\) are empty. To maintain the synopsis of \(S_i\) for this case, each new input tuple \(s_i\) of \(S_i\) is checked against all the filter conditions of \(S_i\) in \(P^+\); if \(s_i\) satisfies all the filter conditions, its projection on the bounded attributes of \(S_i\) is inserted into the synopsis. However, duplicate insertions into the synopsis are ignored. In other words, the synopsis for \(S_i\) is the set (not bag) of tuples resulting from first projecting on the bounded attributes of \(S_i\) all tuples seen so far on \(S_i\) that satisfy its filter conditions, and then removing duplicates among the resulting projected tuples. The size of this set is clearly bounded, since the projected tuples involve only bounded attributes. It is easy to verify, using the fact that all nonredundant join predicates of \(S_i\)
involve only bounded attributes, that any output tuple produced by the original bag of
tuples of \( S_i \), by joining with tuples of other streams, is also produced by the projected set
of tuples in the synopsis for \( S_i \) by joining with same set of tuples of other streams.

Next, consider the case where \( \text{MaxRef}(S_i) \) is not empty. From Condition C3, it follows
that \( \text{MinRef}(S_i) \) is empty. Now, some unbounded attribute \( A \) of \( S_i \) is involved in a nonre-
dundant, inequality predicate \((S_i.A > S_j.B), (i \neq j)\). For this case, the synopsis of \( S_i \)
contains attribute \( A \) in its schema in addition to the usual bounded attributes of \( S_i \), and is
maintained as follows. For each input tuple \( s_i \) on stream \( S_i \) that satisfies all the filter con-
ditions of \( S_i \) in \( P^+ \), the tuple \( s_p \) formed by projecting \( s_i \) onto \( A \) and the bounded attributes
of \( S_i \) is computed. If there is no tuple \( s_p' \) in the current synopsis that agrees with \( s_p \) on all
the bounded attributes, \( s_p \) is inserted into the synopsis; however, if there exists some such
tuple \( s_p' \), the tuple among \( s_p \) and \( s_p' \) having the larger value on attribute \( A \) is retained in
the synopsis, and the other is discarded. Since the two tuples differ only on their value on
attribute \( A \), and this value is used only to check the predicate \((S_i.A > S_j.B)\), any output
tuple produced using the discarded tuple can be produced using the tuple retained in the
synopsis. Therefore, among all tuples of \( S_i \) that agree on all the bounded attributes and
satisfy the filter conditions of \( S_i \), the tuple with the maximum value of \( A \) is chosen, and its
projection is stored in the synopsis for \( S_i \). Again, it is relatively straightforward to verify
that at any instant, the set of tuples in the synopsis behave exactly like the entire bag of
tuples of \( S_i \), as far as the output of \( Q \) is concerned.

Finally, the case where \( \text{MinRef}(S_i) \) is not empty and \( \text{MaxRef}(S_i) \) is empty is handled
symmetrically. The following example illustrates the evaluation strategy described above.

**Example 3.5** Consider the query
\[
Q = \pi_A (\sigma_{(A = 10) \land (B < C) \land (B > 10) \land (C > 10)}(S \times T))
\]
over streams \( S(A, B) \) and \( T(C) \), obtained from the query in Example 3.4 by replacing the
duplicate-preserving projection by a duplicate-eliminating one. The duplicate-eliminating
version of the query is bounded-memory computable, unlike the duplicate-preserving ver-
sion. The schema of the synopsis for \( S \) contains both attribute \( A \), since it is bounded, and
attribute \( B \), since it occurs in \( \text{MinRef}(S) \). As there is only a single possible value of \( A \) that
satisfies the filter conditions of \( S \), the synopsis for \( S \) contains at most one tuple: the tuple,
if it exists, with the minimum value of \( B \) among all tuples with \( A = 10 \) and \( B > 10 \). The
synopsis for \( T \) also contains at most one tuple: the tuple with the maximum value of \( C \) so
far with \( C > 10 \). Any new tuple of \( T \) is joined with the synopsis of \( S \), and analogously, a
new tuple of \( S \) is joined with the synopsis of \( T \). For this query, the output of any successful
join is always the tuple \( \langle 10 \rangle \). Therefore, the first successful join results in the output \( \langle 10 \rangle \), and all subsequent joins are ignored.

Conditions C1–C3 are less restrictive for duplicate-eliminating queries than for duplicate-preserving queries, because for the duplicate-eliminating case, when a new stream tuple arrives, it is sufficient to know whether or not the new tuple produces a given output tuple by joining with the earlier tuples of the other streams, while for the duplicate-preserving case, it is necessary to know the number of different ways the new tuple joins with tuples of other streams to produce the same output tuple. For Query \( Q \) of this example, to check if a new \( S \) tuple joins with some earlier \( T \) tuple producing output tuple \( \langle 10 \rangle \), the \( T \) tuple \( t_{\text{max}} \) with the maximum value of attribute \( C \) greater than 10 that we store in our synopsis suffices, since if the new \( S \) tuple joins with any other \( T \) tuple, it joins with \( t_{\text{max}} \) as well. However, to determine the exact number of \( T \) tuples that the new \( S \) tuple joins with, we need to remember the entire distribution of \( C \) values seen so far on stream \( T \), as we illustrated in Example 3.4.

□

“Only-If” part

For the “only-if” part of Theorem 3.3, the following example illustrates how violation of Condition C3 can force any evaluation strategy for \( Q \) to use unbounded memory.

Example 3.6 Consider query \( Q = \pi_A (\sigma_{(A = 10)} \land (B > D) \land (C > E) \land (B > 10) \land (C < 10) \land (D > 10) \land (E < 10)) \ (S \times T) \) over streams \( S(A, B, C) \) and \( T(D, E) \). This query has two nonredundant inequality predicates between \( S \) and \( T \), which violates Condition C3. Consider an instant in the evaluation of \( Q \) where \( N \) tuples, \( \langle 10, 11, -11 \rangle, \ldots, \langle 10, 10 + N, -(10 + N) \rangle \), have arrived on stream \( S \), and no tuple has arrived on stream \( T \). For each \( S \) tuple \( \langle 10, c, -c \rangle \), there exists a potential \( T \) tuple \( \langle c - 1, -c - 1 \rangle \) that joins only with \( \langle 10, c, -c \rangle \) and not with any of the other \( N - 1 \) tuples. Therefore, any evaluation strategy for \( Q \) has to remember all the \( N \) tuples of \( S \), and since \( N \) can be made arbitrarily large, requires unbounded memory.

□

3.4.3 Only-If Proofs of Theorems 3.2 and 3.3

This section presents the formal “only-if” proofs of Theorems 3.2 and 3.3. The remainder of the thesis does not depend on the material of this section, so a reader who is not interested in the intricate details of the proof may skip ahead to Section 3.4.4.
Proof Structure

Since Theorems 3.2 and 3.3 are very similar—the only differences being Condition C3 and the special case of \( n = 1 \) (single stream queries)—we combine their only-if proofs for the most part. Our proof technique is a generalization of the technique used in Examples 3.4 and 3.6: We show that when a query violates one of the Conditions C1-C3 of Theorems 3.2 or 3.3, we can generate an arbitrarily large set of input tuples such that any evaluation strategy for the query provably needs to remember every tuple in this set. Since this set can be made arbitrarily large, the query cannot be computed in bounded memory. We handle the violation of Conditions C1-C3 separately: Lemma 3.2 handles the violation of Condition C1 for both Theorems 3.2 and 3.3, Lemma 3.3 handles the violation of Condition C2 for both Theorems 3.2 and 3.3, Lemma 3.4 handles the violation of Condition C3 for Theorem 3.2, and Lemma 3.5 handles the violation of Condition C3 for Theorem 3.3.

Notation

The rest of this section, including statements of lemmas and their proofs, is described in the context of an LTO query \( Q = \Pi_L \sigma_P (S_1 \times \ldots \times S_n) \). For simplicity, we assume that there exists at least one constant in the query, i.e., \( |C(Q)| > 0 \). Extending the proof to handle the case \( |C(Q)| = 0 \) is straightforward. In our proofs, we use hypothetical, “universal” tuples that assign a value to every attribute in \( Q \). In other words, the schema of a universal tuple is the concatenation of the schemas of all the streams in \( Q \). We call a universal tuple valid if it satisfies the selection predicate \( P \). If \( t \) is a universal tuple, \( t[S_i] \) denotes the projection of \( t \) onto attributes of \( S_i \), and \( t[A] \) denotes the value of an attribute \( A \) in the tuple \( t \).

Proof Details

**Lemma 3.2** Let attribute \( A \in L \) be unbounded. Then \( Q \) is not computable in bounded memory if:

1. \( n > 1 \), or

2. \( Q \) has a duplicate-eliminating projection.

(Note: The above formulation combines Condition C1 of both Theorem 3.2 and Theorem 3.3.)

**Proof:** Without loss of generality, assume \( A \in A(S_1) \), i.e., attribute \( A \) belongs to \( S_1 \). We claim that we can construct valid universal tuples \( t_1, \ldots, t_N \) for any \( N \), such that each \( t_i \) has
a different value for attribute \( A \), i.e., \( t_i[A] \neq t_j[A] \) if \( i \neq j \). This construction is possible because \( A \) is unbounded. From these \( N \) tuples, we construct \( 2^N \) input instances, \( I_1, \ldots, I_{2^N} \) as follows: Each instance consists of a set of tuples belonging to \( S_1 \) and no tuples belonging to other streams (if any). For the \( S_1 \) tuples, each instance contains a different subset of the set \( \{t_1[S_1], \ldots, t_N[S_1]\} \). We will prove that any algorithm for evaluating \( Q \) must be in a different state after seeing each of the above input instances. Our proof considers two cases:

**Case 1:** (\( n > 1 \)). Consider two input instances \( I_i \) and \( I_j (i \neq j) \). Without loss of generality, there exists a tuple \( t_k[S_1] \) in \( I_i \) but not in \( I_j \). Consider the set of tuples \( \Delta I = \{t_1[S_2], \ldots, t_k[S_n]\} \) of streams \( S_2, \ldots, S_n \). Tuple \( t_k[L] \) must appear in the output if \( \Delta I \) appears in the input after \( I_i \), but not if \( \Delta I \) appears after \( I_j \). So an algorithm that has the same state after seeing \( I_i \) and \( I_j \) produces a wrong output for at least one of the two input instances: \( I_i \) followed by \( \Delta I \), or \( I_j \) followed by \( \Delta I \).

**Case 2:** (\( Q \) has duplicate-eliminating projection). Assume \( n = 1 \), otherwise we can use the argument from Case 1. Again consider two instances \( I_i \) and \( I_j \) and a tuple \( t_k[S_1] = t_k \) that appears in \( I_i \) but not in \( I_j \). If tuple \( t_k \) appears again in the input after \( I_i \), no new tuple should be produced in the output because \( t_k \) is a duplicate. However, if \( t_k \) appears after \( I_j \), a new tuple \( t_k[L] \) must appear in the output. Thus an algorithm for \( Q \) that has the same memory state after seeing either \( I_i \) and \( I_j \) will produce a wrong answer for at least one of the two input scenarios above.

Therefore, any algorithm for evaluating \( Q \) needs at least the \( \log 2^N = \Omega(N) \) bits of memory to encode different states for each of the \( 2^N \) instances. Since \( N \) can be made arbitrarily large, \( Q \) is not computable in bounded memory.

**Lemma 3.3** If \( Q \) contains an equality-join predicate \( S_u.A = S_v.B \) such that both attributes \( A \) and \( B \) are unbounded, it is not computable in bounded memory.

**Proof:** Let \( E_i \) denote the common equivalence class to which \( A \) and \( B \) belong. Since \( A \) and \( B \) (as well as the rest of \( E_i \)) are unbounded, we can construct valid universal tuples \( t_1, \ldots, t_N \) for any \( N \), such that each tuple assigns a different value to the equivalence class \( E_i \). As in Lemma 3.2, from these \( N \) tuples we construct \( 2^N \) input instances \( I_1, \ldots, I_{2^N} \), where each instance contains a different subset of \( \{t_1[S_u], \ldots, t_N[S_u]\} \) for tuples of stream
$S_u$, and no tuples for the other streams. Again, we will prove that any algorithm for evaluating $Q$ must be in a different state after seeing each of these instances.

Consider any two input instances $I_i$ and $I_j$, and let $t_k[S_u]$ occur in $I_i$ and not in $I_j$. Consider the set of tuples $\Delta I = \{t_k[S_w] : w \neq u\}$, one tuple for each non-$S_u$ stream. If $\Delta I$ appears in the input after $I_i$, the tuple $t_k[L]$ must appear in the output, but not if $\Delta I$ appears after $I_j$. There is no output in the latter case because the non-$S_u$ tuples in $\Delta I$ do not join with the $S_u$ tuples in $I_j$: For any $t_m[S_u]$ in $I_j$, by construction, $t_m[B] \neq t_k[A]$, so the join condition $S_u.A = S_v.B$ is not satisfied.

Therefore an algorithm that has the same state after seeing $I_i$ and $I_j$ will produce a wrong output for at least one of the two input scenarios. It follows that any algorithm will require at least $\Omega(N)$ bits of memory, and so $Q$ is not bounded-memory computable. □

**Lemma 3.4** Let $Q$ be duplicate-preserving ($\Pi = \bar{\Pi}$). If $|\text{MaxRef}(S_u)| \neq 0$ or $|\text{MinRef}(S_u)| \neq 0$ for some stream $S_u$, $Q$ is not computable in bounded memory.

**Proof**: We prove the case $|\text{MaxRef}(S_u)| \neq 0$. The proof for $|\text{MinRef}(S_u)| \neq 0$ is symmetric. By definition of MaxRef (Definition 3.9), there exists a nonredundant inequality join $S_u.B < S_u.A (u \neq v)$, such that $A$ is unbounded. Using the fact that $Q$ is an LTO query, we can show that $B$ is also unbounded. We claim that we can construct $N$ valid universal tuples $t_1, \ldots, t_N$ for any $N$ with the following property:

$$t_1[B] < t_1[A] < \ldots < t_N[B] < t_N[A]$$

This construction is possible because $A$ and $B$ are unbounded. Exactly as in the proofs above, we construct $2^N$ inputs instances, $I_1, \ldots, I_{2^N}$, where each instance consists of a different subset of $\{t_1[S_u], \ldots, t_N[S_u]\}$ for stream $S_u$ and no tuples for the other streams. We will prove that any algorithm for $Q$ has to be in a different state after seeing each of the inputs above.

Consider any two input instances $I_i$ and $I_j (i \neq j)$. Let $t_k[S_u]$ be the tuple with the largest value of $k$ among all the tuples appearing in one of $I_i$ and $I_j$ but not in the other. Without loss of generality, assume $t_k[S_u]$ appears in $I_i$ and not in $I_j$. Note that all the tuples $t_l[S_u]$ ($l > k$) occur either in both $I_i$ and $I_j$ or in neither. Now consider the set of tuples $\Delta I = \{t_k[S_w] : w \neq u\}$, one for each non-$S_u$ stream. Consider the new output tuples produced in two input scenarios—$\Delta I$ occurring after $I_i$ and $\Delta I$ occurring after $I_j$.  


CHAPTER 3. MEMORY REQUIREMENTS OF QUERIES

The new output in the former scenario is equal to the output in the latter along with an additional tuple \( t_k[L] \). By construction, tuples in \( \Delta I \) join with \( t_k[S_u] \) to produce \( t_k[L] \) in the output. Also, tuples in \( \Delta I \) do not join with any tuple \( t_l[S_u] \) \((l < k)\) since this would cause the predicate \( B < A \) to be violated. Tuples in \( \Delta I \) could possibly join with some tuple \( t_l[S_u] \) \((l > k)\) to produce an output tuple, but by our choice of \( k \), \( t_l[S_u] \) either occurs in both \( I_i \) and \( I_j \) or in neither, so the output tuple is produced in both input scenarios. The main point of the argument is that the new output for the first scenario \((I_i + \Delta I)\) differs from that for the second scenario \((I_j + \Delta I)\). Hence an algorithm that has the same memory state after seeing \( I_i \) and \( I_j \) will produce a wrong answer for at least one of the two input scenarios. Therefore any correct evaluation algorithm for \( Q \) needs at least \( \log_2 N = \Omega(N) \) bits of memory, so \( Q \) is not computable in bounded memory. \( \square \)

**Lemma 3.5** Let \( Q \) be duplicate eliminating \((\Pi = \pi)\). If \( |\text{MaxRef}(S_u)|_{eq} + |\text{MinRef}(S_u)|_{eq} > 1 \) for some stream \( S_u \), then \( Q \) is not bounded-memory computable.

**Proof:** We assume that there are no equality joins involving unbounded attributes, otherwise we can use Lemma 3.3 to show that \( Q \) is not bounded-memory computable. The proof is split into three cases. Each case uses the same proof template that we used in Lemmas 3.2-3.4: We construct valid, universal tuples \( t_1, \ldots, t_N \) for an arbitrary \( N \). These tuples will satisfy some properties specific to the case being proved. From these tuples we generate \( 2^N \) input instances, \( I_1, \ldots, I_{2^N} \), where each instance contains a different subset of \( \{t_1[S_u], \ldots, t_N[S_u]\} \) for the stream \( S_u \) and no tuples for the other streams. Finally, we prove that any algorithm for evaluating \( Q \) needs to be in a different memory state after seeing each of the instances above.

**Case 1:** \((|\text{MinRef}(S_u)|_{eq} > 1)\). By definition, there exist nonredundant atomic predicates \( p_1 = (S_u.A < S_v.C) \) and \( p_2 = (S_u.B < S_w.D) \), \( u \neq v, u \neq w \) in \( P^+ \), such that \( A \) and \( B \) are unbounded and belong to different equivalence classes. Since the attributes of any single stream are totally ordered in an LTO query, we can assume without loss of generality that \( A < B \in P^+ \). Using the fact that \( Q \) is an LTO query, we can show that attributes \( B \) and \( C \) are also unbounded.

We construct valid, universal tuples \( t_1, \ldots, t_N \) for an arbitrary \( N \) with the following property:

\[
t_1[A] < t_1[C] < \ldots < t_N[A] < t_N[C] < t_N[B] < t_N[D] < \ldots t_1[B] < t_1[D]
\]
We claim that under this ordering, a tuple \( t \) satisfies all the predicates of \( P^+ \) involving only attributes \( A, B, C, \) and \( D \). Clearly, \( t \) satisfies the predicates \((A < C), (B < D), \) and \((A < B)\), which we already know to be in \( P^+ \). The only predicates involving \( A, B, C, \) and \( D \) that are consistent with the three above, but do not logically follow from them are: \((B = C), (B < C), (D < C), \) and \((D = C)\). The predicate \((B = C)\) cannot be in \( P^+ \), since we assumed there are no equality joins involving unbounded attributes. If any of the remaining three predicates, \((B < C), (D < C), \) and \((D = C)\) is present in \( P^+ \), the predicate \( p_1 = (A < C) \) would be redundant, which is a contradiction. The previous argument completes the proof of the claim. A complete proof that we can construct the tuples \( t_1, \ldots, t_N \) for any \( P^+ \) builds on this claim, and uses the fact that attributes \( A, B, C, \) and \( D \) are unbounded. It is fairly straightforward and we do not include it here.

The rest of the proof follows our usual template: From these \( N \) tuples we generate \( 2^N \) instances, \( I_1, \ldots, I_{2^N} \), as described in the beginning of the proof. Consider any two instances \( I_i \) and \( I_j \), and let \( t_k[S_u] \) occur in \( I_i \) and not in \( I_j \). Define the set of tuples \( \Delta I = \{t_k[S_x] : x \neq u\} \) containing one tuple for each non-\( S_u \) stream. If \( \Delta I \) arrives in the input after \( I_i \) the tuple \( t_k[L] \) has to appear in the output, but not if \( \Delta I \) appears after \( I_j \). Therefore any algorithm for evaluating \( Q \) needs to be in a different state after each of the \( 2^N \) instances, so it requires at least \( \Omega(N) \) memory.

**Case 2:** (\(|\text{MaxRef}(S)|_{eq} > 1\)). Symmetric to the proof of Case 1.

**Case 3:** (\(|\text{MaxRef}(S_u)|_{eq} = |\text{MaxRef}(S_u)|_{eq} = 1\)). In this case there exist nonredundant atomic predicates \( p_1 = (S_u.A > S_v.C) \) and \( p_2 = (S_u.B < S_w.D), u \neq v, u \neq w \) in \( P^+ \), where attributes \( A, B, C, \) and \( D \) are all unbounded. Note that \( C \) and \( D \) can be the same attribute or belong to the same equivalence class; similarly, \( A \) and \( B \) can be the same attribute or belong to the same equivalence class. We consider two subcases. For each subcase we only specify how tuples \( t_1, \ldots, t_N \) are constructed. The remaining details are analogous to the proof for Case 1a.

**Case 3a:** (Either all of \( A, B, C, D \) are lower-bounded or all are upper-bounded). We construct valid tuples \( t_1, \ldots, t_N \) with the following property: for any \( t_i, t_j \) \((i < j)\), the values assigned by \( t_i \) to attributes \( A, B, C, D \) are all smaller than the values assigned by \( t_j \) to \( A, B, C, D \). That is, \( t_i[X] < t_j[Y] \), for \( X, Y \in \{A, B, C, D\} \).
Case 3b: (Some among $A, B, C, D$ are lower-bounded and some upper-bounded). Either $A$ and $C$ (respectively $B$ and $D$) are both lower-bounded or are both upper-bounded, otherwise the predicate $p_1$ (respectively $p_2$) is redundant. Let us assume that $A, C$ are upper-bounded and $B, D$ lower-bounded. For this case the tuples $t_1, \ldots, t_N$ satisfy:

$$t_1[C] < t_3[A] < t_2[C] < t_2[A] < \ldots < t_N[C] <_m t_N[A] < k_{\text{min}}$$

$$\leq k_{\text{max}} < t_1[B] < t_1[D] < t_2[B] < t_2[D] < \ldots < t_N[B] < t_N[D]$$

Here $k_{\text{max}}$ and $k_{\text{min}}$ denote the smallest and the largest constant values in $Q$, respectively. The case when $A, C$ are lower-bounded and $B, D$ upper-bounded is symmetric. □

3.4.4 Summary

To summarize this section, we first reduced the problem of characterizing bounded-memory computability of SPJ queries to the problem of characterizing bounded-memory computability of a special class of queries called LTO queries. Then, in Sections 3.4.1 and 3.4.2, we presented a characterization of bounded-memory computable duplicate-preserving and duplicate-eliminating LTO queries, respectively. The results of this section can not only be used to determine if an SPJ query $Q$ is computable in bounded memory—by checking using either Theorem 3.2 or Theorem 3.3 if each LTO query derived (Definition 3.8) from $Q$ is computable in bounded memory—but they also suggest an evaluation strategy if $Q$ is computable in bounded memory: Rewrite $Q$ as a union of all LTO queries derived from it, evaluate each LTO query independently, and accumulate the output of all LTO queries (Theorem 3.1).

This naive technique for checking bounded-memory computability of SPJ queries and evaluating them is, however, very inefficient. The number of LTO queries derived from an SPJ query is usually exponential in the number of attributes of the SPJ query. In the next two sections, we use the basic ideas from this section to derive more efficient techniques for checking and evaluation that avoid explicitly rewriting the query into LTO queries.

3.5 Checking Algorithm

This section presents a simple polynomial algorithm that determines if an SPJ query $Q = \Pi_L(\sigma_P(S_1 \times \cdots \times S_n))$ is bounded-memory computable, without explicitly checking each
LTO query derived from \( Q \).

The algorithm is shown in Figure 3.2. It handles both duplicate-preserving and duplicate-eliminating queries. The terms \( k_{\text{max}} \) and \( k_{\text{min}} \) in the algorithm denote the maximum and minimum constant values, respectively, appearing in \( Q \).

**Input:** SPJ query \( Q = \Pi_L(\sigma_P(S_1 \times \cdots \times S_n)), \Pi \in \{\hat{\pi}, \pi\} \)

**Output:** *Yes*, if \( Q \) is computable in bounded memory. *No*, otherwise.

1. If \( P \) is not satisfiable, or if \( n = 1 \) and \( \Pi = \hat{\pi} \), return *Yes*.

2. If some attribute \( A \in L \) is unbounded, return *No*.

3. If there exists a predicate \( S_i.A = S_j.B \in P \ (i \neq j) \), and at least one attribute \( A \) or \( B \) is unbounded, return *No*.

4. For each \( X \subseteq A(Q) \) with \( |X| \leq 4 \), form a query \( Q' \) with an empty projection list, \( \text{IND}(X \cup \{k_{\text{max}}, k_{\text{min}}\}, P) \) as the selection predicate, and joining the (at most 4) streams that have at least one of their attributes in \( X \). If any such \( Q' \) is not computable in bounded memory (using Theorems 3.3 and 3.2), return *No*.

5. Return *Yes*.

Figure 3.2: Algorithm to check bounded memory computability of SPJ Queries.

The algorithm proceeds in five steps. Step 1 checks if \( Q \) is trivially computable in bounded memory. Steps 2, 3, and 4, respectively, check if one of the Conditions C1, C2, or C3 of Theorem 3.2 (if \( \Pi = \hat{\pi} \)) or Theorem 3.3 (if \( \Pi = \pi \)) is violated by some LTO query derived from \( Q \). If one of the three conditions is violated, then \( Q \) is not bounded-memory computable, and a *No* is returned in the corresponding step; otherwise, a *Yes* is returned in Step 5.

We briefly describe how steps 2–4 check the Conditions C1–C3 without explicitly enumerating each derived LTO query. Steps 2 and 3 are based on the observation that an unbounded attribute of \( Q(P) \) remains unbounded in some LTO query derived from \( Q \), and, conversely, any unbounded attribute of an LTO query derived from \( Q \) is unbounded in \( Q \) as well. For Step 2, this observation implies that an unbounded attribute belonging to \( L \) remains unbounded in some LTO query derived from \( Q \) violating Condition C1. Similarly, for Step 3, two unbounded attributes \( A \) and \( B \) involved in an equality join (\( A = B \)) remain unbounded in some LTO query derived from \( Q \), thereby violating Condition C2.
Step 4 of the algorithm uses the fact that just two nonredundant predicates suffice as “witnesses” to violation of Condition C3. Two predicates involve at most four attributes, so any violation of Condition C3 can be detected by checking all queries with four attributes or less constructed from $Q$ in a particular way. For concreteness, let $Q$ have a duplicate-eliminating projection (the case of duplicate-preserving projection can be argued similarly). Assume that some LTO query, $Q_L$, derived from $Q$ violates Condition C3 (of Theorem 3.3). Then, there exist two nonredundant inequality join predicates $p_1$ and $p_2$, involving a common stream $S_i$, that cause $|\text{MaxRef}(S_i)|_{eq} + |\text{MinRef}(S_i)|_{eq} > 1$. Let $X$ denote the set of (at most 4) attributes participating in $p_1$ and $p_2$. Then we can show that query $Q'$ constructed from $X$ in Step 4 is not computable in bounded memory, and therefore our algorithm correctly returns No. Conversely, if some query $Q'$ constructed in Step 4 is not bounded-memory computable due to a violation of Condition C3 by some LTO query $Q'_L$ derived from $Q'$, then Condition C3 is also violated by some LTO query $Q_L$ derived from $Q$. The following example illustrates Step 4 of our algorithm for an example input query.

**Example 3.7** Figure 3.3 illustrates Step 4 of our algorithm for an example input query. A query is represented in Figure 3.3 using its selection predicate alone (since the projected attributes are not relevant to Step 4). An inequality join predicate between two entities is represented by a directed edge between the entities, and attributes belonging to the same stream are indicated using an enclosing rectangle.

Figure 3.3(a) shows an input query $Q$ with duplicate-eliminating projection over three streams $S_1(A, B, C, D)$, $S_2(E, F, G)$ and $S_3(H, I, J)$, and Figure 3.3(b) shows an LTO query $Q_L$ derived from $Q$. Query $Q$ is not bounded-memory computable, since $Q_L$ violates Condition C3 of Theorem 3.3: $\text{MaxRef}(S_1) = \{A, C\}$ (due to $(H < A)$ and $(G < C)$). Figure 3.3(c) shows the query $Q'$ constructed from $X = \{H, A, C, G\}$, the set of attributes involved in violation of Condition C3 in $Q_L$, in Step 4 of our algorithm. Query $Q'$ is not bounded-memory computable since an LTO query, $Q'_L$ (Figure 3.3(d)), derived from $Q'$ violates Condition C3. Observe that $Q'_L$ is related to $Q_L$ the same way $Q'$ is related to $Q$.□

Clearly, Steps 1–3 can be executed in polynomial time in the size of the input query. Each query $Q'$ in Step 4 is of $O(1)$ size, so we can check its LTO queries in constant time. Since there are $\Theta(|\mathcal{A}(Q)|^4)$ subsets of $\mathcal{A}(Q)$ having cardinality at most 4, Step 4 takes polynomial time. Thus our algorithm is polynomial in the size of the input query.
Figure 3.3: Example illustrating checking algorithm of Figure 3.2. Sub-figure (a) shows an input query $Q$, (b) shows an LTO query $Q_L$ derived from $Q$, (c) shows a query $Q'$ constructed from $Q$ is Step 4 of our algorithm, and (d) shows an LTO query $Q_L'$ derived from $Q'$.

### 3.6 Query Evaluation Strategy

As described in Section 3.4.4, a naive evaluation strategy for SPJ queries is to rewrite a query as a union of LTO queries, evaluate each LTO query independently, and accumulate their output. In this section, we present an evaluation strategy that is more efficient computationally and uses less memory.

Intuitively, our efficient evaluation strategy logically merges all the individual synopses maintained by each LTO query of the naive evaluation strategy into a single synopsis. Since there is substantial redundancy among the different synopses of the naive strategy, the merged synopsis is not much larger than any of the individual synopses, and therefore the efficient evaluation strategy uses significantly less memory than the naive
strategy, by a factor roughly equal to the number of LTO queries.

We first introduce some notation used to describe our evaluation strategy. We use uppercase letters to denote streams and the corresponding lowercase letters to denote instances of streams. For example, \( s_i \) denotes an instance of Stream \( S_i \). The instance (prefix) of a stream at any point in time is the bag of tuples seen so far on that stream. For a query \( Q \) over streams \( S_1, \ldots, S_n \), \( Q(s_1, \ldots, s_n) \) denotes the output of \( Q \) when the instances \( s_i \) \((1 \leq i \leq n)\) of streams have been seen.

We present our evaluation strategy for duplicate-preserving queries and duplicate-eliminating queries separately.

### 3.6.1 Evaluation Strategy for Duplicate-Preserving Queries

For this entire subsection (3.6.1), consider a bounded-memory computable, duplicate-preserving query \( Q = \pi_L(\sigma_P(S_1 \times \cdots \times S_n)) \). We ignore the trivial case of \( n = 1 \). As for the case of LTO queries, our evaluation strategy for \( Q \) maintains a synopsis summarizing the current instance \( s_i \) of each stream. The synopsis, which we denote \( \text{Syn}(s_i) \), is a bag of tuples having the same schema as \( S_i \), and it has the property that it is equivalent to \( s_i \) for the purposes of evaluating \( Q \). Formally, \( Q(s_1, \ldots, s_n) = Q(\text{Syn}(s_1), \ldots, \text{Syn}(s_n)) \). When a new tuple arrives on a stream, our evaluation strategy joins the new tuple with the synopses for all other streams, and outputs any resulting tuples. The property of the synopses described above guarantees the correctness of this evaluation strategy.

#### Buckets

In order to maintain the synopsis for \( S_i \), the tuples of \( S_i \) are logically partitioned into a bounded number of buckets. Buckets are defined as follows: Let \( k_{\text{max}} \) and \( k_{\text{min}} \) denote the maximum and minimum constant value appearing in \( Q \), respectively.\(^5\) Let \( \mathcal{K} \) denote the set of integers between \( k_{\text{max}} \) and \( k_{\text{min}} \) inclusive, i.e., \( \mathcal{K} = \{ k_{\text{min}}, k_{\text{min}} + 1, \ldots, k_{\text{max}} \} \). Partition the domain of integers into \( |\mathcal{K}| + 2 \) non-overlapping ranges: \((-\infty, k_{\text{min}} - 1], [k_{\text{min}}, k_{\text{min}}], \ldots, [k_{\text{max}}, k_{\text{max}}], [k_{\text{max}} + 1, \infty)\); the first and the last ranges are unbounded, and the intermediate ones span exactly one integer. A bucket \( b \) of \( S_i \) assigns to each attribute \( A \) of \( S_i \) one of the \( |\mathcal{K}| + 2 \) ranges. Thus, there are \( (|\mathcal{K}| + 2)^{|A(S_i)|} \) distinct buckets of \( S_i \) corresponding to all the possible assignments of ranges to attributes. We use the notation

\(^5\)We are implicitly assuming the existence of constants in \( Q \). Removing this assumption is straightforward.
$b[A]$ to denote the range identified by bucket $b$ for attribute $A$, writing $b[A] = -\infty$ if the range is $(-\infty, k_{\text{min}} - 1]$, $b[A] = k$ if the range is $[k, k]$, $k \in \mathbb{K}$, and $b[A] = \infty$ if the range is $[k_{\text{max}} + 1, \infty)$. A tuple $t$ belongs to a bucket if its value for each attribute falls within the range assigned by the bucket. Clearly, each tuple belongs to a single bucket.

**Example 3.8** Consider Query $Q_7 = \pi_A(\sigma_{(B < D) \land (D > 10) \land (B < 20) \land (A = 10)}(S \times T))$ from Table 3.1. For this query $k_{\text{max}} = 20$ and $k_{\text{min}} = 10$. Therefore there are 13 different buckets for stream $S(A, B, C)$. An example of a bucket of $S$ is the bucket $b$ with ranges $b[A] = -\infty$, $b[B] = 14$, and $b[C] = \infty$. All tuples of $S$ with a value less than 10 for attribute $A$, the value 14 for attribute $B$, and a value greater than 20 for attribute $C$ belong to this bucket.

**Equivalence Property of Buckets**

Any two tuples $t_1$ and $t_2$ belonging to the same bucket are equivalent for the purposes of evaluating $Q$. In other words, whenever $t_1$ joins with a set of tuples from other streams to produce an output tuple, $t_2$ also joins with the same set of tuples to produce the same output tuple. This equivalence property holds only for bounded-memory computable, duplicate-preserving queries, not for arbitrary queries. Example 3.9 illustrates this equivalence property and Lemma 3.7 proves it property formally. (Lemma 3.6 is a “helper” lemma used within the proof of Lemma 3.7.)

**Example 3.9** Consider Query $Q_7$ shown in Example 3.8 again. Consider bucket $b_1$ with ranges $b_1[A] = 10$, $b_1[B] = -\infty$ and $b_1[C] = \infty$. All tuples of bucket $b_1$ have a value less than 10 on attribute $B$. Consequently, they all join with any tuple of stream $T$ of the form $(d, e)$, $d > 10$ producing the same output $(10)$, and fail to join with any other $T$ tuple.

**Lemma 3.6** For any inequality join predicate $(S_i.A < S_j.B) \in P^+$, $i \neq j$, if both $A$ and $B$ are unbounded then $A$ is upper-bounded and $B$ is lower-bounded.

**Proof:** Assume otherwise. Then our checking algorithm of Table 3.2 returns “No” in Step 4 for $X = \{A, B\}$, implying that $Q$ is not computable in bounded memory, which is a contradiction.

**Lemma 3.7** Let $T = \{t_1, \ldots, t_n\}$ be a set of tuples belonging to streams $S_1, \ldots, S_n$, respectively, that join (i.e., satisfy $P$) to produce the output tuple $t$. Let $t'_i$ be any tuple of $S_i$ that satisfies
all the filter conditions of $S_i$ and belongs to the same bucket as $t_i$. Then the set of tuples $T' = T \cup \{t'_i\} - \{t_i\}$ join to produce the same output tuple $t$.

**Proof:** Since $Q$ is computable in bounded memory, all the attributes in the project list $L$ are bounded. Clearly, $t_i$ and $t'_i$ agree on all the bounded attributes. Therefore, if the tuples in $T'$ satisfy $P$, they produce the same output tuple as the tuples in $T$.

The tuples $T'$ satisfy all the atomic predicates of $P$ that do not involve $S_i$, since $T$ and $T'$ differ only in their $S_i$ tuple. We prove that $T'$ also satisfies all atomic predicates of $P$ involving some attribute of $S_i$. We consider filter predicates, equality join predicates, and inequality join predicates separately. By definition, $t'_i$ satisfies all filter predicates of $S_i$. Since $Q$ is computable in bounded memory, all the equality join predicates of $P$ involve only bounded attributes. Since $t_i$ and $t'_i$ agree on all the bounded attributes, the tuples $T'$ satisfy all the equality join predicates of $P$.

Consider an inequality join predicate $(S_i.A < S_j.B)$. We prove that $t'_i[A] < t_j[B]$ is valid by considering three cases.

1. $(k_{\text{min}} \leq t_i[A] \leq k_{\text{max}})$ In this case, $t'_i[A] = t_i[A]$. Therefore, $t'_i[A] < t_j[B]$.

2. $(t_i[A] < k_{\text{min}})$ In this case, by definition of a bucket, $t'_i[A] < k_{\text{min}}$. Also, $t_i[A] < k_{\text{min}}$ implies that the attribute $A$ is unbounded in $P$. If attribute $B$ is bounded, $(k_{\text{min}} \leq t_j[B] \leq k_{\text{max}})$, and therefore $t'_i[A] < t_j[B]$. If attribute $B$ is unbounded, from Lemma 3.6 it follows that $A$ is upper-bounded and $B$ is lower-bounded. Since $B$ is lower-bounded, $k_{\text{min}} \leq t_j[B]$, and therefore $t'_i[A] < t_j[B]$.

3. $(t_i[A] > k_{\text{max}})$ This case cannot occur. Since $t_j[B] > t_i[A]$, $t_j[B] > k_{\text{max}}$. Therefore, both $A$ and $B$ are unbounded. From Lemma 3.6 it follows that $A$ is upper-bounded, which implies that $t_i[A] < k_{\text{max}}$, a contradiction.

We can similarly show that the tuples $T'$ satisfy any inequality join predicate $(S_i.A > S_j.B)$.

**Details of Synopses and Evaluation Strategy**

The equivalence property described above is used to maintain the synopsis for $S_i$. For a bucket $b$ of stream $S_i$, let $\text{count}(b)$ denote the number of tuples of $S_i$ seen so far that belong to $b$, and let $t_{\text{first}}(b)$ denote the tuple belonging to $b$ that first appears on $S_i$; we discount
tuples that fail the filter conditions of $S_i$ while defining $\text{count}(b)$ and $\text{t}_{\text{first}}(b)$. The synopsis for $S_i$ consists of $\text{count}(b)$ copies of $\text{t}_{\text{first}}(b)$ for each bucket $b$ of $S_i$. From the equivalence property above, $\text{count}(b)$ copies of the tuple $\text{t}_{\text{first}}(b)$ is equivalent to the bag of $\text{count}(b)$ tuples that belong to $b$; hence, our synopsis for $S_i$ is equivalent to the bag of tuples of $S_i$ seen so far. Physically, the synopsis for $S_i$ is represented by storing, for each bucket $b$, $\text{t}_{\text{first}}(b)$ and $\text{count}(b)$, which requires bounded memory since the number of buckets is bounded. The following theorem follows trivially from Lemma 3.7.

**Theorem 3.4** For all instances $s_i$ of input streams $S_i$ ($1 \leq i \leq n$), $Q(s_1, s_2, \ldots, s_n) = Q(\text{Syn}(s_1), \ldots, \text{Syn}(s_n))$.

### 3.6.2 Evaluation Strategy for Duplicate-Eliminating Queries

For this entire subsection (3.6.2) consider a bounded-memory computable, duplicate-eliminating query $Q = \pi_L (\sigma_P (S_1 \times \cdots \times S_n))$. As for the case of duplicate-preserving queries, our evaluation strategy for $Q$ summarizes the current stream instances $s_i$ using a bounded memory synopsis $\text{Syn}(s_i)$ that is equivalent to $s_i$ for evaluating $Q$, i.e., $Q(s_1, \ldots, s_n) = Q(\text{Syn}(s_1), \ldots, \text{Syn}(s_n))$.

To maintain the synopsis for $S_i$, the tuples of $S_i$ are partitioned into buckets exactly as in the case of duplicate-preserving queries. However, unlike the case of duplicate-preserving queries, the property that any two tuples belonging to the same bucket are equivalent with respect to $Q$ is not valid for duplicate-eliminating queries. However, a weaker property still holds: For any finite set (bag) of tuples belonging to the same bucket, it is always possible to pick a subset of one or two representative tuples that is equivalent to the entire set for the purpose of evaluating $Q$. Using this property, our synopsis contains, for each bucket $b$ of $S_i$, at most two tuples representing the bag of tuples of $S_i$ seen so far that belong to $b$ and satisfy all the filter conditions of $S_i$.

Informally, we pick the tuple with the maximum value in each $\text{MaxRef}$ attribute (Definition 3.9) and the tuple with the minimum value in each $\text{MinRef}$ attribute as representative tuples. For each bucket $b$, associate a set of filter predicates $P_b$ such that only tuples belonging to the bucket satisfy $P_b$. The set $P_b$ contains one atomic predicate corresponding to each attribute $A$ of $S_i$: If $b[A] = -\infty$, the atomic predicate corresponding to $A$ is $(A < k_{\text{min}})$; if $b[A] = k \in K$, then the atomic predicate is $(A = k)$; if $b[A] = \infty$, then the atomic predicate is $(A > k_{\text{max}})$. The definition of $\text{MaxRef}$ and $\text{MinRef}$ for a bucket $b$ differs
slightly from Definition 3.9, and takes into account the additional filter predicates \( P_b \) that the tuples belonging to \( b \) satisfy.

**Definition 3.10** Consider a query \( Q(P) \) and a stream \( S_i \in S(Q) \). For a bucket \( b \) of \( S_i \), \( \text{MaxRef}(S_i, b) \) is defined as the set of all unbounded attributes \( A \) of \( S_i \) in \( (P \cup P_b) \) that participate in a nonredundant inequality join \((S_j.B < S_i.A), i \neq j, (P \cup P_b)^+\). \( \text{MinRef}(S_i, b) \) is similarly defined as the set of all unbounded attributes \( A \) of \( S_i \) that participate in a nonredundant inequality join of the form \((S_i.A < S_j.B), i \neq j, (P \cup P_b)^+\).

The representative tuples for a bucket \( b \) of \( S_i \) are picked as follows:

1. If \( \text{MaxRef}(S_i, b) \) is nonempty, pick the tuple \( t \) that has the maximum value of \( \min\{t[A] : A \in \text{MaxRef}(S_i, b)\} \). We denote this representative tuple by \( t_{\text{max}}(b) \) in the rest of this section.

2. If \( \text{MinRef}(S_i, b) \) is nonempty, pick the tuple \( t \) that has the minimum value of \( \max\{t[A] : A \in \text{MinRef}(S_i, b)\} \). We denote this tuple by \( t_{\text{min}}(b) \).

3. If both \( \text{MaxRef}(S_i, b) \) and \( \text{MinRef}(S_i, b) \) are empty, pick the first tuple that appears on \( S_i \). We denote this tuple by \( t_{\text{first}}(b) \).

**Example 3.10** Consider the query \( Q(P) = \pi_A (\sigma_{(B > F)} \land (C > F) \land (D > F) \land (E = A)) (S \times T) \) over streams \( S(A, B, C, D) \) and \( T(E, F) \). The reader can verify that \( Q \) is bounded-memory computable. Consider a bucket with ranges \( b[A] = 10, b[B] = 10, b[C] = -\infty \) and \( b[D] = -\infty \). The predicate \( P_b \) for this bucket is \{\((A = 10), (B = 10), (C < 10), (D < 10)\)\}. Attributes \( C \) and \( D \) are unbounded for predicate \( (P \cup P_b) \); attribute \( B \) is bounded for \( (P \cup P_b) \) although it is unbounded for predicate \( P \). Both \( C \) and \( D \) occur in \( \text{MaxRef}(S_i, b) \) due to nonredundant predicates \( (C > F) \) and \( (D > F) \). \( \text{MinRef}(S_i, b) \) is empty.

One representative tuple is sufficient for bucket \( b \): It is the tuple \( s \) with the maximum value of \( \min\{s[C], s[D]\} \) among all tuples of \( S_i \) seen so far that belong to \( b \).

**Formal Proof of Correctness**

To simplify our proofs we partition \( \text{MaxRef}(S_i, b) \) into two sets, \( L\text{MaxRef}(S_i, b) \) and \( U\text{MaxRef}(S_i, b) \), defined as follows. (”L” stands for lower-bounded and “U” for upper-bounded.)
1. \( L_{\text{MaxRef}}(S_i, b) = \{ A \mid A \in \text{MaxRef}(S_i, b) \text{ and } b[A] = \infty \} \)

2. \( U_{\text{MaxRef}}(S_i, b) = \{ A \mid A \in \text{MaxRef}(S_i, b) \text{ and } b[A] = -\infty \} \).

Similarly, we partition \( \text{MinRef}(S_i, b) \) into two sets \( L_{\text{MinRef}}(S_i, b) \) and \( U_{\text{MinRef}}(S_i, b) \) based on the ranges assigned to attributes by bucket \( b \).

**Lemma 3.8** For any bucket \( b \) of \( S_i \):

1. Let \( A_1, A_2 \in \text{MaxRef}(S_i, b) \). Let \( (B < A_1) \in (P \cup P_b)^+ \). Then \( (B < A_2) \in (P \cup P_b)^+ \).

2. Let \( A_1, A_2 \in \text{MinRef}(S_i, b) \). Let \( (A_1 < B) \) be a nonredundant predicate in \( (P \cup P_b)^+ \). Then \( (A_2 < B) \in (P \cup P_b)^+ \).

**Proof:** We sketch the proof of Part 1; the proof of Part 2 is symmetric. Since \( Q(P) \) is computable in bounded memory, and all the predicates of \( P_b \) are filters, \( Q(P \cup P_b) \) is also computable in bounded memory. Suppose for the sake of contradiction that Part 1 is incorrect, i.e., \( (B < A_2) \notin (P \cup P_b)^+ \). Since \( A_2 \in \text{MaxRef}(S_i, b) \), by definition there exists a nonredundant predicate \( (C < A_2) \in (P \cup P_b)^+ \). It can be verified that our checking algorithm of Figure 3.2 returns “No” in Step 4 when \( X = \{ A_1, A_2, B, C \} \) for the query \( Q(P \cup P_b) \). This implies that \( Q(P \cup P_b) \) is not computable in bounded memory, which is a contradiction. \( \square \)

**Lemma 3.9** For any bucket \( b \) of \( S_i \):

1. At least one of \( L_{\text{MaxRef}}(S_i, b) \) and \( U_{\text{MaxRef}}(S_i, b) \) is empty.

2. At least one of \( L_{\text{MinRef}}(S_i, b) \) and \( U_{\text{MinRef}}(S_i, b) \) is empty.

**Proof:** We provide the proof of Part 1; the proof of Part 2 is symmetric. Suppose for the sake of contradiction that Part 1 does not hold. Then there exist attributes \( A_1 \in \text{MaxRef}(S_i, b) \) and \( A_2 \in \text{MaxRef}(S_i, b) \). By the definition of \( \text{MaxRef} \), there exists a nonredundant predicate \( (B < A_1) \in (P \cup P_b)^+ \). It follows from Lemma 3.8 that \( (B < A_2) \in (P \cup P_b)^+ \). However, since \( (A_2 < k_{\text{min}}) \in P_b \) and \( (k_{\text{max}} < A_1) \in P_b \), it follows that \( (B < A_1) \) is redundant in \( (P \cup P_b)^+ \), which is a contradiction. \( \square \)

**Lemma 3.10** For any bucket \( b \) of \( S_i \), if both \( \text{MaxRef}(S_i, b) \) and \( \text{MinRef}(S_i, b) \) are nonempty then:
CHAPTER 3. MEMORY REQUIREMENTS OF QUERIES

1. \( \text{LM}_{\text{MaxRef}}(S_i, b) \) and \( \text{UM}_{\text{MinRef}}(S_i, b) \) are nonempty; \( \text{UM}_{\text{MaxRef}}(S_i, b) \) and \( \text{LM}_{\text{MinRef}}(S_i, b) \) are empty.

2. Let \( Y \) denote the set of attributes \( D \) that are involved in nonredundant predicates (\( D < A \)), \( A \in \text{LM}_{\text{MaxRef}}(S_i, b) \), or involved in nonredundant predicates (\( A < D \)), \( A \in \text{UM}_{\text{MinRef}}(S_i, b) \). Here “nonredundant” means nonredundant in \( (P \cup P_b) \). The set of attributes \( Y \) belong to the same equivalence class (Definition 3.5) in \( (P \cup P_b) \).

Proof: Let \( A_1 \in \text{MaxRef}(S_i, b) \) and \( A_2 \in \text{MinRef}(S_i, b) \). By definition there exist attributes \( B \) and \( C \) such that the atomic predicates (\( B < A_1 \)) and (\( A_2 < C \)) are nonredundant in \( (P \cup P_b) \). Unless \( b[A_1] = \infty \), and \( b[A_2] = -\infty \), and \( B, C \) belong to the same equivalence class, our checking algorithm (Figure 3.2) returns “No” in Step 4, when \( X = \{A_1, A_2, B, C\} \), for the query \( Q(P \cup P_b) \), implying that \( Q \) is not computable in bounded memory, a contradiction. Hence it follows that all the attributes in \( \text{MaxRef}(S_i, b) \) occur in \( \text{LM}_{\text{MaxRef}}(S_i, b) \), implying that \( \text{LM}_{\text{MaxRef}}(S_i, b) \) is nonempty and \( \text{UM}_{\text{MaxRef}}(S_i, b) \) is empty. By a symmetric argument it follows that \( \text{UM}_{\text{MinRef}}(S_i, b) \) is nonempty and \( \text{LM}_{\text{MinRef}}(S_i, b) \) is empty, proving Part 1.

For the proof of Part 2, let \( Y_\text{max} \) denote the set of attributes \( D \) involved in nonredundant inequality joins of the form (\( D < A \)), \( A \in \text{LM}_{\text{MaxRef}}(S_i, b) \). Similarly, let \( Y_\text{min} \) denote the set of attributes \( D \) involved in nonredundant inequality joins (\( A < D \)), \( A \in \text{UM}_{\text{MinRef}}(S_i, b) \). (Here “nonredundant” means nonredundant in \( (P \cup P_b) \).) By definition \( Y = Y_\text{min} \cup Y_\text{max} \). The attributes \( B \) and \( C \) (from the proof of Part 1) are arbitrary attributes of \( Y_\text{max} \) and \( Y_\text{min} \), respectively. Since \( B \) and \( C \) belong to the same equivalence class, for an arbitrary choice of \( B \) and \( C \), every attribute in \( Y_\text{max} \) belongs to the same equivalence class as every attribute in \( Y_\text{min} \). Since both \( Y_\text{min} \) and \( Y_\text{max} \) are nonempty it follows that all attributes of \( Y \) belong to the same equivalence class.

Theorem 3.5 For all instances \( s_i \) of input streams \( S_i \) \((1 \leq i \leq n)\), \( Q(s_1, \ldots, s_n) = Q(\text{Syn}(s_1), \ldots, \text{Syn}(s_n)) \).

Proof: Since \( \text{Syn}(s_i) \subseteq s_i \) and \( Q \) is monotone, it follows that \( Q(\text{Syn}(s_1), \ldots, \text{Syn}(s_n)) \subseteq Q(s_1, \ldots, s_n) \). We prove that \( Q(s_1, \ldots, s_n) \subseteq Q(\text{Syn}(s_1), \ldots, \text{Syn}(s_n)) \) holds as well. Let \( t \in Q(s_1, \ldots, s_n) \) be formed by joining tuples \( t_1, \ldots, t_n, t_i \in s_i \). We claim that there exist tuples \( t_i' \in \text{Syn}(s_i) \) \((1 \leq i \leq n)\) that join to produce the same output tuple \( t \). Let \( b_i \) be the bucket of stream \( S_i \) to which the tuple \( t_i \) belongs. The tuple \( t_i' \) is one of the representative tuples of \( b_i \) described earlier, and is identified as follows.
CHAPTER 3. MEMORY REQUIREMENTS OF QUERIES

1. If \( \text{MaxRef}(S_i, b_i) \) is nonempty and \( \text{MinRef}(S_i, b_i) \) empty, pick \( t_{\text{max}}(b_i) \) as \( t_i' \).
2. If \( \text{MinRef}(S_i, b_i) \) is nonempty and \( \text{MaxRef}(S_i, b_i) \) empty, pick \( t_{\text{min}}(b_i) \) as \( t_i' \).
3. If both \( \text{MaxRef}(S_i, b_i) \) and \( \text{MinRef}(S_i, b_i) \) are empty, pick \( t_{\text{first}}(b_i) \) as \( t_i' \), and
4. If neither \( \text{MaxRef}(S_i, b_i) \) and \( \text{MinRef}(S_i, b_i) \) is empty, then both \( t_{\text{max}}(b_i) \) and \( t_{\text{min}}(b_i) \) exist in \( \text{Syn}(s_i) \). We pick one of these two tuples as \( t_i' \) depending on the tuples \( t_1, \ldots, t_n \) as described below. Since neither \( \text{MaxRef}(S_i, b_i) \) and \( \text{MinRef}(S_i, b_i) \) is empty, we can infer using Lemma 3.10 that \( \text{LMaxRef}(S_i, b_i) \) and \( \text{UMinRef}(S_i, b_i) \) are both nonempty, and \( \text{UMaxRef}(S_i, b_i) \) and \( \text{LMinRef}(S_i, b_i) \) are both empty. Further, let \( Y \) denote the set of attributes of other streams involved in a nonredundant (in \( (P \cup P_b)^* \)), inequality join with an attribute of \( \text{MaxRef}(S_i, b_i) \) and \( \text{MinRef}(S_i, b_i) \). Lemma 3.10 states that all the attributes of \( Y \) belong to the same equivalence class. Therefore, the tuples \( t_1, \ldots, t_n \) assign the same value, say \( v \), to all the attributes in \( Y \).

If \( v \) is smaller than \( k_{\text{min}} \), pick \( t_{\text{min}}(b_i) \) as \( t_i' \); if \( v \) is greater than \( k_{\text{max}} \), pick \( t_{\text{max}}(b_i) \) as \( t_i' \); otherwise, arbitrarily pick one of \( t_{\text{min}}(b_i) \) or \( t_{\text{max}}(b_i) \) as \( t_i' \).

Since \( Q \) is computable in bounded memory, all attributes in \( \text{project list} \ L \) are bounded. Clearly, \( t_i \) and \( t_i' \) agree on all the bounded attributes. Therefore, if the tuples \( t_i' (1 \leq i \leq n) \) satisfy all the conditions in \( P \), they produce the same output tuple \( t \). Since \( Q \) is computable in bounded memory, all the equality join predicates of \( P \) involve only bounded attributes. Therefore, the set of tuples \( t_i' \) satisfy all the equality join predicates in \( P \). By definition, all the tuples of \( \text{Syn}(s_i) \) satisfy the filter conditions in \( P \) involving only the attributes of \( S_i \). Therefore, the tuples \( t_i' \) satisfy all the filter conditions in \( P \).

Inequality join predicates are the remaining type of predicate to be considered. Since it simplifies the proof, we prove the stronger statement that the tuples \( t_i' (1 \leq i \leq n) \) satisfy all the inequality join predicates in \( (P \cup P_{b_1} \cup \ldots \cup P_{b_n}) \). Consider a nonredundant inequality join predicate \( (A < B) \) in \( (P \cup P_{b_1} \cup \ldots \cup P_{b_n}) \). Let attributes \( A \) and \( B \) belong to streams \( S_i \) and \( S_m \), respectively. Clearly the tuples \( t_1, \ldots, t_n \) satisfy all the predicates in \( (P \cup P_{b_1} \cup \ldots \cup P_{b_n}) \), and therefore \( t_i[A] < t_m[B] \). Since \( (A < B) \) is nonredundant, it follows that either both \( A \) and \( B \) are upper-bounded but not lower-bounded, or both are lower-bounded but not upper-bounded, with respect to the predicates \( (P \cup P_{b_1} \cup \ldots \cup P_{b_n}) \). Without loss of generality, assume that \( A \) and \( B \) are lower-bounded but not upper-bounded. Since \( (A < B) \) is nonredundant in \( (P \cup P_{b_1} \cup \ldots \cup P_{b_n}) \), and each \( P_{b_i} (1 \leq i \leq n) \)
is a filter, it follows that \((A < B)\) is nonredundant in \((P \cup P_{b_1})\) and in \((P \cup P_{b_n})\) as well. Therefore, \(A \in LMinRef(S, b_l)\) and \(B \in LMaxRef(S, b_m)\). By repeated application of Lemma 3.8, it can be shown that the predicate \((C < D) \in (P \cup P_{b_1} \cup \ldots \cup P_{b_n})^+\) for every \(C \in LMinRef(S, b_l)\) and \(D \in LMaxRef(S, b_m)\). Therefore:

\[
\max\{t_l[C] : C \in LMinRef(S, b_l)\} < \min\{t_m[D] : D \in LMaxRef(S, b_m)\} \quad (3.1)
\]

Next, we claim that \(t_{min}(b_l)\) is picked as \(t'_l\) and \(t_{max}(b_m)\) is picked as \(t'_m\). Since \(LMinRef(S, b_l)\) is nonempty, it follows from Lemma 3.10 that \(MaxRef(S, b_l)\) is empty, and therefore we pick \(t_{min}(b_l)\) as \(t'_l\). Similarly, if \(MinRef(S, b_m)\) is empty we pick \(t_{max}(b_m)\) are \(t'_m\). However, it is possible that \(MinRef(S, b_m)\) is nonempty since Lemma 3.10 allows \(LMaxRef(S, b_m)\) (which is obviously nonempty) and \(UMinRef(S, b_m)\) to be nonempty. For this case, we pick \(t'_m\) depending on the common value assigned by \(t_1, \ldots, t_n\) to the attributes \(Y\) involved in a nonredundant (in \((P \cup P_m)^+)\), inequality join with attributes of \(MaxRef(S, b_m)\) \(\cup MinRef(S, b_m)\). This value is greater than \(k_{max}\) since \(A \in Y\) and \(A\) lower-bounded but not upper-bounded, and therefore \(t_{max}(b_m)\) is picked for this case as well.

From the definition of \(t_{min}(b_l)\) and \(t_{max}(b_m)\), it follows that:

\[
\begin{align*}
t'_l[A] & \leq \max\{t'_l[C] : C \in LMinRef(S, b_l)\} \\
& \leq \max\{t_l[C] : C \in LMinRef(S, b_l)\} \\
& < \min\{t_m[D] : D \in LMaxRef(S, b_m)\} \\
& \leq \min\{t'_m[D] : D \in LMaxRef(S, b_m)\} \\
& \leq t'_m[B]
\end{align*}
\]

Therefore, \(t'_l[A] < t'_m[B]\) holds, implying that the tuples \(t'_1, \ldots, t'_n\) satisfy all the inequality join predicates in \((P \cup P_{b_1} \cup \ldots \cup P_{b_n})\).

\[\square\]

### 3.7 Extensions

In this section we extend our results to a larger class of queries. Section 3.7.1 considers queries with self-joins, Section 3.7.2, queries with grouping and aggregation, and Section 3.7.3, queries with sliding windows.
3.7.1 Queries with Self-Joins

In a self-join query, at least one stream appears more than once in the join list. We use the notation $S^1, S^2, \ldots$ to denote different occurrences of the same stream $S$ in a query. For instance, query $\pi_{S^1.A}(\sigma_{(S^1.A = S^2.A)}(S^1 \times S^2))$ is a join of stream $S$ with itself on attribute $A$.

The characterization of bounded-memory computable self-join queries differs slightly from that of non-self-join queries due to an implicit constraint on self-join streams: at any point of time, the instances of any two self-join streams, $S^j$ and $S^k$, are the same. This additional constraint affects our characterization for duplicate-eliminating queries only. For duplicate-preserving queries, our characterization of bounded-memory computability using Theorems 3.1, 3.2, and 3.3 continues to hold. However, the “only-if” proofs of Theorems 3.2 and 3.3 have to be modified slightly, since currently they assume that instances of streams could be arbitrary. Since our characterization remains unchanged, the checking algorithm of Section 3.5 and the efficient evaluation strategy of Section 3.6 can be used for duplicate-preserving queries with self-joins as well.

For duplicate-eliminating queries, our reduction of bounded-memory computability of SPJ queries to that of LTO queries still holds (Theorem 3.1); however, our characterization of bounded-memory computable LTO queries (Theorem 3.3) does not, as the following example illustrates.

**Example 3.11** Consider the self-join LTO query $Q = \pi_{S^1.A}(\sigma_P(S^1 \times S^2))$, where $P = \{(S^1.A = 10), (S^1.A = S^2.A), (S^1.B = S^2.B), (S^1.B > 10)\}$. The equality join of unbounded attributes $S^1.B$ and $S^2.B$ violates Condition C2 of Theorem 3.3, but $Q$ is equivalent to the query $Q' = \pi_A(\sigma_{(A = 10) \land (B > 10)}(S))$, which is clearly bounded-memory computable. □

Conditions C1–C3 of Theorem 3.3 are still sufficient to ensure that a self-join LTO query is computable in bounded memory, but they are not necessary, i.e., there exist queries (e.g., query $Q$ of Example 3.11) that violate one or more of Conditions C1–C3, but are computable in bounded memory.

Informally, one of the two occurrences of stream $S$ in query $Q$ of Example 3.11 was “redundant,” which allowed $Q$ to be rewritten as $Q'$ using just one occurrence of $S$. Theorem 3.3 fails only for queries with such redundant streams. In other words, a duplicate-eliminating LTO query without any redundant streams is bounded-memory computable if and only if Conditions C1–C3 hold. Our strategy for handling duplicate-eliminating LTO
queries is, therefore, to first rewrite these queries without redundant streams and then use Theorem 3.3 to check bounded-memory computability of the rewritten queries. We formalize redundant streams, and show how a query can be rewritten without redundant streams.

**Definition 3.11** A stream $S_j^i$ is said to be redundant in an SPJ query $Q(P)$ if there exists a stream $S_k^i (j \neq k)$ such that:

1. If $(S_j^i.A \ Op \ k) \in P^+$, then $(S_k^i.A \ Op \ k) \in P^+$, and if $(S_j^i.A \ Op \ S_j^i.B) \in P^+$, then $(S_k^i.A \ Op \ S_k^i.B) \in P^+$ (i.e., any filter condition of $S_j^i$ is also a filter condition of $S_k^i$).
2. If $(S_j^i.A \ Op \ S_i.B) \in P^+$, then $(S_k^i.A \ Op \ S_i.B) \in P^+$.
3. If $S_j^i.A \in L$, where $L$ is the list of projected attributes, then $(S_j^i.A = S_k^i.A) \in P^+$.

In such a case, we say that $S_k^i$ covers $S_j^i$ in $Q$. □ □

In Example 3.11, $S_1^1$ covers $S_2^2$, and vice-versa. If $S_k^i$ covers $S_j^i$ in an SPJ query $Q$, the query $Q'$ obtained from $Q$ by eliminating $S_j^i$, and replacing all occurrences of $S_j^i.A$ by $S_k^i.A$ is equivalent to $Q$. By repeatedly using this rewriting step, we can eliminate all redundant streams from a query.

To summarize, a duplicate-eliminating query with self-joins is bounded-memory computable if and only if all the LTO queries derived from it are bounded-memory computable. In order to determine if a duplicate-eliminating LTO query is bounded-memory computable, we remove all redundant streams from the query and check if the resulting rewritten query is bounded-memory computable using Theorem 3.3. Currently, we do not know how to extend the more efficient checking (Section 3.5) and evaluation algorithms (Section 3.6) to handle duplicate-eliminating SPJ queries with self-joins; extending these algorithms is future work.

### 3.7.2 Queries with Grouping and Aggregation

We now extend our main results to queries involving aggregation and (optionally) grouping. For queries involving only the standard aggregate functions (SUM, COUNT, MIN, MAX, AVG, COUNT-DISTINCT, and MEDIAN), we provide necessary and sufficient conditions for bounded-memory computability, and argue that any query that is not bounded-memory computable requires a worst-case space that is linear in the size of its input (Theorems 3.8
For queries that involve user-defined aggregate functions, we only provide a set of sufficient conditions for bounded-memory computability (Theorems 3.6 and 3.7). Stating necessary conditions for bounded-memory computability that apply to all aggregate functions is difficult because one can easily design nonstandard aggregate functions that admit special optimization tricks.

A query involving grouping and aggregation (hereafter simply an “aggregate query”) is of the general form $G \land F(\sigma_P(S_1 \times \cdots \times S_n))$, where $G$ is a (possibly empty) set of grouping attributes and $F$ is a set of aggregate expressions. (Recall the grouping and aggregation operator $G \land F$ that we introduced in Section 2.7 in the previous chapter.) An aggregate expression is of the form $f(S_i.A)$, where $f$ is an aggregate function such as $\text{SUM}$ or $\text{COUNT}$. The aggregate query $G \land F(\sigma_P(S_1 \times \cdots \times S_n))$ is equivalent to the CQL query:

```
SELECT G, F FROM S_1, \ldots, S_n WHERE P GROUP BY G
```
aggregate function cannot. Among the standard aggregate functions, \( \text{SUM}, \text{COUNT}, \text{MIN}, \text{MAX} \) are distributive, \( \text{AVG} \) is algebraic since it can be computed from a synopsis containing \( \text{SUM} \) and \( \text{COUNT} \), and \( \text{COUNT-DISTINCT} \) and \( \text{MEDIAN} \) are holistic.

Further, we classify aggregate functions as \textit{duplicate-sensitive} or \textit{duplicate-insensitive}. Let \( X \) be a bag of values, and \( X' \) the set of distinct elements in \( X \). An aggregate function \( f \) is duplicate-insensitive if \( f(X) = f(X') \) for all bags \( X \), and duplicate-sensitive otherwise. Functions \( \text{MIN}, \text{MAX}, \text{COUNT-DISTINCT} \) are duplicate-insensitive, while \( \text{SUM}, \text{COUNT}, \text{AVG}, \text{MEDIAN} \) are duplicate-sensitive.

**Theorem 3.6** A single-stream aggregate query \( Q = G G F(\sigma_P(S)) \) can be computed in bounded memory if: (1) every grouping attribute in \( G \) is bounded; and (2) there is no aggregate expression \( f(A) \in F \) such that \( f \) is holistic and \( A \) is unbounded.

**Proof:** Partition the input tuples that satisfy \( P \) into the groups defined by the grouping attributes. There are a bounded number of groups because the grouping attributes are all bounded. Within each group, the values of distributive and algebraic aggregates can be maintained using bounded memory, by definition. For each attribute \( A \) that is aggregated using a holistic aggregate function, maintain counts of the number of times each value of \( A \) occurs within each group; by condition (2) attribute \( A \) must be bounded, so these counts can be maintained using bounded memory. The counts completely capture the distribution of \( A \) within the group, allowing the holistic aggregates to be computed from them. \( \square \)

Now consider a multi-stream aggregate query \( Q = G G F(\sigma_P(S_1 \times \cdots \times S_n)) \). Let \( A(F) = \{S_i.A : f(S_i.A) \in F\} \) denote the set of attributes used in aggregate expressions in \( F \). We define the \textit{characteristic query} \( Q' \) for \( Q \) as \( \Pi_{G \cup A(F)}(\sigma_P(S_1 \times \cdots \times S_n)) \), where \( \Pi = \pi \) if all aggregate functions in \( F \) are duplicate-insensitive and \( \Pi = \bar{\pi} \) otherwise.

**Theorem 3.7** Consider an aggregate query \( Q = G G F(\sigma_P(S_1 \times \cdots \times S_n)) \). If the characteristic query \( Q' \) for \( Q \) can be computed in bounded memory, then so can \( Q \).

**Proof:** We provide a bounded-memory evaluation strategy for evaluating \( Q \). Evaluate as a subroutine the characteristic query \( Q' \) using the appropriate algorithm from Section 3.6. As output tuples are generated by the subroutine, partition them into groups defined by the grouping attributes, and within each group, maintain counts of the number of occurrences of each value for each of the attributes being aggregated. If \( Q' \) is duplicate-preserving, then the counts are a complete description of the distribution of the aggregated attributes,
so they are sufficient to compute the aggregate functions for each group. If $Q'$ is duplicate-eliminating, then the number of times each attribute value occurred is lost, but that does not matter since $Q'$ is only duplicate-eliminating when all aggregates in $Q$ are duplicate-insensitive. The fact that $Q'$ is computable in bounded memory implies that all grouping attributes and all aggregated attributes in $Q$ are bounded, so the total memory required for “post-processing” the output of $Q'$ also is bounded.

Next we present the necessary and sufficient conditions for determining whether an aggregate query, involving only the standard aggregates, is bounded-memory computable. Again, we consider single- and multi-stream queries separately.

**Theorem 3.8** Let $Q = GGF(\sigma_P(S))$ be an aggregate query over a single stream, where the aggregate functions in $F$ are drawn from SUM, COUNT, MIN, MAX, AVG, COUNT-DISTINCT, and MEDIAN. $Q$ is bounded-memory computable if and only if: (1) every grouping attribute in $G$ is bounded; and (2) there is no aggregate expression $f(A) \in F$ such that $f$ is holistic (i.e., COUNT-DISTINCT or MEDIAN) and $A$ is unbounded.

**Proof:** This theorem states that Conditions (1) and (2) from Theorem 3.6 are necessary as well as sufficient for the standard aggregates. The “if” direction is a special case of Theorem 3.6. The “only if” direction is straightforward. As we observed earlier, every grouping attribute has to be bounded to keep the size of the output bounded. For Condition (2), it is well known that computing the number of distinct values or the median of a bag requires memory proportional to the number of distinct values in the bag, which implies that all attributes aggregated by COUNT-DISTINCT and MEDIAN must be bounded.

The reduced characteristic query of an aggregate query $Q = GGF(\sigma_P(S_1 \times \cdots \times S_n))$ is defined in the same way as the characteristic query for $Q$ defined earlier, except now the attributes in the project list are only the grouping attributes $G$ rather than $G \cup A(F)$. Formally, the reduced characteristic query $Q'_R$ for $Q$ is $\Pi_G(\sigma_P(S_1 \times \cdots \times S_n))$, where $\Pi = \pi$ if all aggregate functions in $F$ are duplicate-insensitive and $\Pi = \hat{\pi}$ otherwise.

**Theorem 3.9** Let $Q = GGF(\sigma_P(S_1 \times \cdots \times S_n))$ be an aggregate query over multiple streams, involving only the standard aggregates SUM, COUNT, AVG, MAX, MIN, COUNT-DISTINCT, and MEDIAN. Let $Q'_R$ be the reduced characteristic query for $Q$. Then $Q$ is bounded-memory computable if and only if:

- **C1:** $Q'_R$ is computable in bounded memory.
C2: For every aggregate expression \( \text{COUNT-DISTINCT}(S_i.A) \) or \( \text{MEDIAN}(S_i.A) \) in \( F \), \( S_i.A \) is bounded.

C3: For every aggregate expression \( \text{MAX}(S_i.A) \), if \( S_i.A \) is unbounded, either \( \text{MaxRef}(S_i) \) is empty or \( |\text{MaxRef}(S_i)|_{eq} = 1 \) and \( S_i.A \in \text{MaxRef}(S_i) \); similarly, for every aggregate expression \( \text{MIN}(S_i.A) \), if \( S_i.A \) is unbounded, either \( \text{MinRef}(S_i) \) is empty or \( |\text{MinRef}(S_i)|_{eq} = 1 \) and \( S_i.A \in \text{MinRef}(S_i) \).

We discuss the ideas behind Theorem 3.9. A formal proof can be derived from the discussion here.

Without loss of generality, we can assume that \( Q \) consists of just one aggregate expression. An aggregate query having \( n > 1 \) aggregate expressions can be equivalently rewritten as a natural join (on the grouping attributes) of the output of \( n \) aggregate queries, each having one aggregate expression of the original query, but otherwise identical. For example, the query \( A \text{G} \text{SUM}(B) \text{MAX}(C)(\sigma_{A = 10}(S(A, B, C))) \) is equivalent to \( A \text{G} \text{SUM}(B)(\sigma_{A = 10}(S)) \bowtie A \text{G} \text{MAX}(C)(\sigma_{A = 10}(S)) \). Since the set of possible groups in a bounded-memory-computable aggregate query is bounded, a query with more than one aggregate expression is bounded-memory computable if and only if all the queries with one aggregate expression “derived” from it are bounded-memory computable. Therefore, for the rest of this discussion we assume that \( F \) contains only one expression of the form \( f(S_i.A) \).

We break the discussion of Theorem 3.9 into three cases depending on the type of the aggregate function used in the single aggregate expression—holistic (\( \text{COUNT-DISTINCT} \) and \( \text{MEDIAN} \)), non-holistic and duplicate-sensitive (\( \text{SUM} \), \( \text{COUNT} \), \( \text{AVG} \)), non-holistic and duplicate-insensitive (\( \text{MAX} \), \( \text{MIN} \)). For each of these cases, we discuss both the “if” and “only-if” arguments of the proof.

For the holistic aggregates, \( \text{COUNT-DISTINCT} \) and \( \text{MEDIAN} \), the relevant Conditions C1 and C2 of Theorem 3.9 reduce to the sufficiency conditions of Theorem 3.7. Therefore, the “if” part of the proof for holistic aggregates is a special case of the proof of Theorem 3.7. For the “only-if” part of the proof, first assume that \( Q'_R \) is not bounded-memory computable, violating Condition C1. The non-bounded-memory computability of \( Q'_R \) results either from some unbounded attribute in its project list \( G \) (which causes the violation of Condition C1 of Theorem 3.2 for some LTO query), or from some join predicates in \( P \) (which cause the violation of Conditions C2 or C3 of Theorem 3.2). In the former case, \( Q \) is clearly not bounded-memory computable, since one of its grouping attributes is not...
bounded; in the latter case, informally, since even checking all the join predicates of \( P \) requires unbounded memory, evaluating an aggregation on top of the join is not feasible in bounded memory either. Finally, by the definition of holistic aggregates, the aggregated attribute \( S_i.A \) has to be bounded for bounded-memory computability of \( Q \). Therefore, Condition C2 is necessary as well.

Next consider the case of duplicate-sensitive aggregates \( \text{SUM}, \text{COUNT}, \text{AVG} \). For these aggregates, only Condition C1 is relevant, i.e., \( Q \) is bounded-memory computable if and only if \( Q'_R \) is bounded-memory computable. First consider the “if” part of the proof. Note that Condition C1 relaxes the sufficiency conditions of Theorem 3.7: For these aggregates, it is possible to evaluate the query even if the aggregated attribute is unbounded. A query involving these aggregates need not be evaluated, as suggested by the proof of Theorem 3.7, by first computing the join of the streams, projecting the grouping and the aggregated attributes from the result of the join, and then computing the aggregate on the projections. Instead, we can partially “push” the aggregation below the join, which makes it possible to compute these aggregates even for unbounded aggregated attributes. The following example illustrates this evaluation strategy.

Example 3.12 Consider the aggregate query \( Q = _{A,C}G_{\text{SUM}(B)}(\sigma_P(S \times T)) \) over two streams \( S(A,B) \) and \( T(C) \), where \( P = \{(A < 20), (C < 20), (A > 10), (C > 10)\} \). Although the aggregation is over an unbounded attribute \( B \), we assert that \( Q \) is computable in bounded-memory. As always, our evaluation strategy maintains a synopsis for \( S \) and \( T \); in addition, it also maintains the current answer to the query as required by the semantics of aggregate queries. The synopsis for \( S \) contains, for each value \( v \) in the interval \([11,19]\), the sum of attribute \( B \) of all \( S \) tuples seen so far with \( A = v \). The synopsis for \( T \) contains, for each value \( v \) in \([11,19]\), the count of \( T \) tuples seen so far with \( C = v \). The arrival of a new \( T \) tuple \( \langle c \rangle \) \((10 < c < 20)\) changes the output as follows: For each value \( v \in [11,19] \), add the sum corresponding to \( v \) in the current synopsis for \( S \) to each group \( A = v, C = c \) in the output. Similarly, the arrival of a new \( S \) tuple \( \langle a, b \rangle \) \((10 < a < 20)\) changes the output as follows: For each \( v \in [11,19] \), let \( n_v \) denote the current count of tuples in the synopsis for \( T \) with \( C = v \); add a value \( b \cdot n_v \) to each group \( A = a, C = v \) in the output relation. The reader can verify that this evaluation strategy correctly computes the output of \( Q \).

If the aggregated attribute \( S_i.A \) is bounded, we can use the evaluation strategy of Theorem 3.7. If it is not, the evaluation strategy of Section 3.6.1 can be modified as follows.
As before, maintain a synopsis for each stream. The synopses for all streams other than \( S_i \) (recall that \( S_i.A \) is the aggregated attribute) remain unchanged. In order to maintain the synopsis of \( S_i \) the tuples of \( S_i \) are partitioned into buckets exactly as described in Section 3.6.1. For each bucket, in addition to remembering one representative tuple and the count of tuples, remember the aggregate over attribute \( A \). For example, if the aggregate function is SUM, remember the sum of attribute \( A \) over all tuples belonging to each bucket.

The reader can verify that these synopses are sufficient to answer \( Q \) using a technique similar to the one shown in Example 3.12. For the “only-if” proof, if \( Q'_R \) is not bounded-memory computable, we can argue that \( Q \) is not bounded-memory computable as well, exactly as we did for holistic aggregates.

Finally, consider the case of MIN and MAX. We only discuss MAX since the discussion for MIN is analogous. Again consider the “if” part. If the aggregated attribute \( S_i.A \) is bounded, then \( Q \) satisfies the conditions of Theorem 3.7, and, therefore, we can use the evaluation strategy presented in the proof of Theorem 3.7. If \( S_i.A \) is not bounded, then the bounded-memory computability of \( Q \) depends on \( MaxRef(S_i) \). If \( MaxRef(S_i) \) is empty, we can partially push the MAX aggregate below the join exactly as we did for the duplicate-sensitive queries above: For each bucket in the synopsis for \( S_i \), maintain the maximum value of attribute \( A \) over all the tuples that belong to the bucket. If \( MaxRef(S_i) \) is non-empty, Condition C3 states that \( S_i.A \) must be the only attribute in \( MaxRef(S_i) \) (ignoring attributes belonging to the same equivalence class) for \( Q \) to be bounded memory computable. Informally, the maximum value of the attributes in \( MaxRef \) is needed (in some buckets) for the evaluation of the predicate \( P \). If \( A \) is the only attribute in \( MaxRef(S_i) \) (again ignoring attributes belonging the same equivalence class as \( A \)), then there is no conflict, and the maximum value of \( A \) can be stored for each bucket exactly as before. This maximum value is now used both for checking \( P \) and for computing \( \text{MAX}(A) \). For the “only-if” part, if \( Q'_R \) is not bounded-memory computable, we can argue that \( Q \) is not bounded-memory computable exactly as we did for the case of holistic aggregates. If \( A \) is not the only attribute in \( MaxRef(S_i) \), the proof is similar to the only-if proof of Theorem 3.3 for the case \(|MaxRef(S_i)|_{eq} > 1\).

### 3.7.3 Queries with Sliding Windows

We now describe how our results can be extended to handle SPJ queries with explicit sliding window operators over streams. We present only the LTO characterization of
SPJ queries with windows. Using the LTO characterization to derive efficient checking and evaluation algorithms is straightforward, and can be derived from our checking and evaluation algorithms for SPJ queries with default Unbounded windows presented in Sections 3.5 and 3.6, respectively. We do not handle windowed queries with grouping and aggregations, which is left for future work.

We classify each sliding window operator as either memory-bounded or memory-unbounded. A window is memory-bounded if we can bound the maximum number of tuples that can be in the window at any given point in time; it is memory-unbounded otherwise. Time-based windows are memory-unbounded, since any number of stream tuples could fall into the same window. Tuple-based windows are clearly memory-bounded. Partitioned windows are memory-bounded if all the partitioning attributes are bounded (Definition 3.6), and memory-unbounded otherwise.

Intuitively, an LTO query with windows is bounded-memory computable iff every stream has either a memory-bounded window or satisfies all the conditions of Theorem 3.2 (if it is duplicate-preserving) or Theorem 3.3 (if it is duplicate-eliminating). The above intuition is formalized in the following two theorems, which we state without proof. We use the notation $S_i[X_i]$ to denote an arbitrary window $[X_i]$ over $S_i$.

Theorem 3.10 Let $Q = \hat{\pi}_L(\sigma_P(S_1[X_1] \times \cdots \times S_n[X_n]))$ be an LTO query. $Q$ is bounded-memory computable (Definition 3.1) if and only if all three conditions below are satisfied:

C1: If $S_i.A \in L$ is unbounded, then the window $[X_i]$ is memory-bounded.

C2: For every equality join predicate $(S_i.A = S_j.B)$, where $i \neq j$, $S_i.A$ and $S_j.B$ are both bounded, or $[X_i]$ and $[X_j]$ are both memory-bounded.

C3: For each input stream $S_i$, $|\text{MaxRef}(S_i)| = |\text{MinRef}(S_i)| = 0$ or the window $[X_i]$ is memory-bounded.

Theorem 3.11 Let $Q = \pi_L(\sigma_{\bar{P}}(S_1[X_1] \times \cdots \times S_n[X_n]))$ be an LTO query. $Q$ is bounded-memory computable (Definition 3.1) if and only if all three conditions below are satisfied:

C1: If $S_i.A \in L$ is unbounded, then the window $[X_i]$ is memory-bounded.

C2: For every equality join predicate $(S_i.A = S_j.B)$, where $i \neq j$, $S_i.A$ and $S_j.B$ are both bounded, or $[X_i]$ and $[X_j]$ are both memory-bounded.
C3: For each stream $S_i$, $|\text{MaxRef}(S_i)|_{eq} + |\text{MinRef}(S_i)|_{eq} \leq 1$ or the window $[X_i]$ is memory-bounded.

3.8 Conclusion

In this chapter, for a large class of continuous queries based on select-project-joins with optional grouping and aggregation, and optional sliding windows, we presented a precise characterization of queries that can be evaluated in bounded amount of memory. We also proved that queries from our class that cannot be evaluated in bounded amount of memory require memory that grows linearly with the input size.

We initially focused on Select-Project-Join (SPJ) queries without self-joins. We illustrated using several examples why it might be hard to directly characterize bounded-memory computability of SPJ queries. Instead, we presented an indirect characterization that relies on a new subclass of SPJ queries called LTO queries. We showed that several LTO queries can be “derived” from a given SPJ query, and that an SPJ query is bounded-memory computable if and only if every LTO query that can be derived from it is bounded-memory computable. We then characterized when an LTO query is bounded-memory computable. LTO queries have a special structure that makes characterizing their bounded-memory computability easier.

Our characterization using LTO queries, although mathematically simple, does not lend itself to efficient implementation. A naive algorithm based on the LTO characterization requires time exponential in the query size to determine if a query is bounded-memory computable or not. We presented a more sophisticated algorithm that checks if a query is bounded-memory computable or not more efficiently, in polynomial time. For queries that are bounded-memory computable, we also presented a memory-efficient evaluation strategy.

Finally, we extended our characterization to queries with self-joins, queries with grouping and aggregation, and queries with sliding windows.
Chapter 4

Approximate Statistics over Streams

In Chapter 3, we studied memory requirements of continuous queries when queries are required to produce exact, accurate answers. In this chapter, we show for an important, practical class of queries that memory requirements can be significantly less if we are willing to tolerate slight inaccuracies in the query answers. Work presented in this chapter appeared in reference [12].

4.1 Introduction

In Chapter 3, we characterized memory requirements of continuous queries, and we showed that many queries require prohibitively large amounts of memory. However, our characterization assumed that answers of queries need to be fully accurate. For some queries, if we slightly relax the requirement that their answers be exact, we can get large memory savings. In other words, we can compute approximate answers to these queries using significantly less memory. Such approximate query processing can be of great practical utility since for many applications exact answers are not critical. For example, a network monitoring application that queries average volume of network traffic might not be very sensitive to small (say 1%) errors in the answer.

In this chapter, we present new algorithms for computing approximate answers for queries that compute an aggregation over a sliding window (ASW), which we introduced in Section 2.6.7. In our algebra (recall Section 2.7), these queries are of the form $G_F(S[W]_T)$ or of the form $G_F(S[N])$, where $F$ denotes the aggregation function, and $S[W]_T$ and $S[N]$
denote time-based and tuple-based sliding windows, respectively. The problem of computing approximate answers for this class of queries was first studied by Datar et al. [39]. They provide algorithms for COUNT and SUM aggregation functions, and they also prove that approximation does not reduce the space requirement for MIN and MAX aggregation functions.

Our algorithms handle two types of aggregation functions (hereafter “statistics”) not considered by Datar et al.: quantiles and frequent elements. Both these statistics are well-known and widely used in practice. Informally, a quantile identifies an element with a specified rank in a bag of elements, and the frequent elements statistic identifies all elements in a bag with frequency above a specified threshold. Both these statistics are useful to identify skew in a collection of values. We present more details and applications in Section 4.2 after formally defining the statistics.

For almost all known statistics, including quantiles and frequent elements, we can show that any algorithm that computes the statistic exactly requires space that is linear in the current window size.¹ Informally, the algorithms need to “remember” every element in the current window, since they have to “undo” the effect of the element when it slides out of the window. In contrast, all of our approximation algorithms require memory that grows logarithmically² in the current window size, which represents an exponential reduction in memory requirement over exact algorithms.

In terms of accuracy, all of our algorithms have an associated error parameter, usually denoted $\epsilon$. Our algorithms guarantee that, for any input instance and at every time instant, the error of the approximate value of the statistic when compared to the true value is within $\epsilon$. (The exact meaning of “error” depends on the statistic—we define it formally in Section 4.2. Informally, the smaller the error, the higher the accuracy of the approximate answer.) The error parameter $\epsilon$ is tunable, i.e., it can be made as small as desired, but shrinking $\epsilon$ results in an increased memory requirement that is (roughly) inversely proportional to $\epsilon$.

¹The only exception that we know of is the degenerate case of COUNT over a tuple-based window. The answer is the size of the window, which is a fixed value.
²A small-degree (< 3) polynomial in the logarithm of the current window size, to be precise.
Chapter Overview and Organization

We present formal definitions of quantiles and frequent elements, and define approximate versions of these statistics in Section 4.2. We introduce (in Section 4.2.2) a related statistic called approximate frequency counts, which is closely related to frequent elements and is easier to discuss and reason about. We prove that any algorithm for approximate frequency counts can be modified to derive an algorithm for approximate frequent elements. The remainder of the chapter covers the approximate frequency counts statistic.

In Section 4.3, we formally state the set of problems that we study in this chapter. We do not present our algorithms from scratch, but use some previously known algorithms for computing statistics over entire streams (not sliding windows) as building blocks. In order to describe precisely how one algorithm is built on top of another, we need to formalize the state and interfaces of these algorithms. We introduce such formalized representations of algorithms called sketches in Section 4.4. Our suite of algorithms (sketches) is presented in Sections 4.5–4.7: Section 4.5 presents deterministic algorithms for tuple-based windows, Section 4.6 presents randomized algorithms for tuple-based windows, and Section 4.7 presents deterministic and randomized algorithms for time-based windows. Section 4.8 covers related work, and we conclude in Section 4.9.

4.2 Definitions

In this section, we present formal definitions of quantiles and frequent elements. We also define $\epsilon$-approximate versions of these statistics, i.e., we specify when an approximate value of the statistic has error less than $\epsilon$.

4.2.1 Quantiles

Quantiles is not a single statistic, but a class of statistics defined using a parameter between 0 and 1, usually denoted $\phi$. Consider a bag $B$ of $N$ elements drawn from an ordered domain. The quantile with parameter $\phi \in (0, 1]$, denoted $\phi$-quantile, of $B$ is the element at position $\lceil \phi N \rceil$ in a sorted arrangement of the elements of $B$; we use the convention that the position of the smallest element in the sorted arrangement is 1, and that of the largest, $N$. The $\phi$-quantile with $\phi = 0.5$ is the median. We note that the $\phi$-quantile of $B$ is uniquely defined even if $B$ has duplicates.
**Example 4.1** Consider the bag of 9 elements \{11, 12, 12, 13, 13, 13, 14, 14, 15\}. For illustration, we have listed the elements of the bag in sorted order. The 0.75-quantile of this bag is the element at position \(\lceil 0.75 \cdot 9 \rceil = 7\), which happens to be 14. The 0.5-quantile (median) is the element at position 5, which is 13, and the 0.25-quantile is the element at position 3, which is 12.

In our algebra, we use the notation \(\text{QUANTILE}(\phi, A)\) to denote the \(\phi\)-quantile over attribute \(A\). For example, the query \(G_{\text{QUANTILE}(0.5, A)}(S[100])\) computes the median \(A\) value in the last 100 tuples of stream \(S\). As an example use of quantiles, in network monitoring, computing the median, the 0.9-quantile, and the 0.05-quantile of the round-trip times of network packets (say) over the last 5 minutes can reveal useful information about skew in network performance [34]. For instance, a large gap between the 0.95-quantile and the median indicates a high skew, and therefore non-uniform network performance.

**Definition 4.1** (\(\epsilon\)-approximate \(\phi\)-quantile) An element \(e \in B\) is said to be an \(\epsilon\)-approximate \(\phi\)-quantile of \(B\) (or an \((\epsilon, \phi)\)-quantile for short) if there exists \(\phi' \in [\phi - \epsilon, \phi + \epsilon]\) such that \(e\) is the \(\phi'\)-quantile of \(B\). Clearly, more than one element could qualify to be an \((\epsilon, \phi)\)-quantile of \(B\).

**Example 4.2** Consider the same bag of elements from Example 4.1: \{11, 12, 12, 13, 13, 13, 14, 14, 15\}. An element is a \((0.25, 0.5)\)-quantile if it is a \(\phi'\)-quantile for \(\phi' \in [0.25, 0.75]\). All the elements in between positions \([0.25 \cdot 9] = 3\) and \([0.75 \cdot 9] = 7\) have this property, so any of the elements \{12, 13, 14\} is a \((0.25, 0.5)\)-quantile of the bag.

We note that our definition of approximation bounds the error in the quantile parameter. In particular, the definition places no bound on the difference between the value of the \(\phi\)-quantile and the value of an \((\epsilon, \phi)\)-quantile.

### 4.2.2 Frequent Elements

Like quantiles, frequent elements are defined using a parameter \(s \in [0, 1]\). Consider a bag \(B\) of \(N\) elements. The frequency of an element \(e\) is the number of times \(e\) occurs in \(B\). The frequent-elements statistic of \(B\) with parameter \(s\), denoted \(s\)-frequent elements (or \(s\)-FE for short), is the set of elements with frequency \(\geq s \cdot N\). Since there can be at most \(1/s\) elements with frequency \(\geq s \cdot N\), the size of \(s\)-FE of \(B\) is at most \(1/s\). We note that \(s\)-FE of \(B\) can also be the empty set.
Consider a bag of 100 elements shown in Figure 4.1. The elements of the bag are the letters $a$–$z$, and Figure 4.1 specifies the frequency of each letter in the bag. The 0.1-$FE$ of this bag contains all letters with frequency at least $0.1 \cdot 100 = 10$, which is the set \{a, b, c\}. Similarly, the 0.2-$FE$ is \{a\}, and 0.5-$FE$ is the empty set.

In our algebra, we use the notation $\text{FREQ}(s, A)$ to denote the $s$-frequent elements ($s$-$FE$) over attribute $A$. For example, the query $\mathcal{G}_{\text{FREQ}(0.1, A)}(S[100])$ computes the set of $A$ values that occur more than 10 times in the last 100 tuples of $S$. Like quantiles, the frequent-elements statistic is useful to identify skew in a collection of values. However, the frequent-elements statistic is applicable to values from an unordered domain, while quantiles require values from an ordered domain. In network monitoring, the frequent elements statistic can be used to identify network addresses that are generating the most network traffic. A detailed application of frequent-elements for network monitoring is described in Estan et al. [47].

**Definition 4.2 (ε-approximate s-frequent elements)** An $\epsilon$-approximate $s$-frequent elements statistic of $B$ (or an $(\epsilon, s)$-$FE$ for short) is a set $S$ with the following properties: (1) Every element whose frequency in $B$ is $\geq s \cdot N$ occurs in $S$; and (2) The frequency of every element in $S$ is $\geq (s - \epsilon) \cdot N$.\[\square\]
Notice that an \((\epsilon, s)\)-FE of \(B\) allows some false positives, but no false negatives, when compared to \(s\)-FE of \(B\). In other words, any \((\epsilon, s)\)-FE of \(B\) is a superset of \(s\)-FE of \(B\). Further, a false positive element is “almost” a frequent element, and has a frequency at least \((s - \epsilon) \cdot N\). Note that there could be several sets that qualify to be \((\epsilon, s)\)-FE of \(B\); on the other hand, \(s\)-FE of \(B\) is unique.

**Example 4.4** Consider the bag shown in Figure 4.1. Any \((\epsilon, s)\)-FE of the bag with \(\epsilon = 0.02\) and \(s = 0.1\) satisfies the following two properties: (1) It contains every element with frequency \(\geq 0.1 \cdot 100 = 10\), i.e., the elements \(\{a, b, c\}\); and (2) does not contain any element with frequency less than \((0.1 - 0.02) \cdot 100 = 8\), i.e., the elements \(\{f, g, \ldots, z\}\). An example of such a set is \(\{a, b, c, e\}\). □

In this chapter we present memory-efficient algorithms for computing \((\epsilon, s)\)-FE over stream sliding windows. However, instead of presenting algorithms that directly compute \((\epsilon, s)\)-FE, for clarity we present algorithms that compute a related statistic called \(\epsilon\)-approximate frequency counts (or \(\epsilon\)-FC for short). The \(\epsilon\)-FCs have the property that, for any bag \(B\) and any \(s > \epsilon\), an \((\epsilon, s)\)-FE of \(B\) can be computed from any \(\epsilon\)-FC of \(B\). This property is formalized in Proposition 4.1 below.

**Definition 4.3** (\(\epsilon\)-approximate frequency count) An \(\epsilon\)-approximate frequency count (\(\epsilon\)-FC) for a bag \(B\) of \(N\) elements is a set \(S\) of \(\langle e, \tilde{f}_e \rangle\) pairs \((e \in B)\) such that: (1) for each pair \(\langle e, \tilde{f}_e \rangle\) in \(S\), \(\tilde{f}_e\) is an approximate frequency of \(e\) satisfying the property \((f_e - \epsilon N) \leq \tilde{f}_e \leq f_e\), where \(f_e\) is the true frequency of \(e\) in \(B\); (2) For any element \(e \in B\) with a frequency \(f_e \geq \epsilon N\), there exists an entry \(\langle e, \tilde{f}_e \rangle\) in \(S\); other elements may or may not be represented in \(S\). □

**Example 4.5** Consider the bag shown in Figure 4.1. Any \(0.02\)-FC of the bag contains approximate frequencies for all elements whose true frequency \(\geq 0.02 \cdot 100 = 2\), i.e., the elements \(\{a, b, c, d, e\}\). Further, the approximate frequencies are less than the true frequencies, but by at most 2. An example of \(0.02\)-FC for this bag is the set \(\{(a, 33), (b, 17), (c, 9), (d, 8), (e, 6), (p, 1), (w, 1)\}\). □

**Proposition 4.1** For any bag \(B\) and any \(s > \epsilon\), we can construct some \((\epsilon, s)\)-FE of \(B\) from an \(\epsilon\)-FC of \(B\).

**Proof:** Let \(S_{\epsilon}\) be any \(\epsilon\)-FC of \(B\). Consider the set \(S_{\epsilon} = \{e \mid \langle e, \tilde{f}_e \rangle \in S_{\epsilon} \land \tilde{f}_e \geq (s - \epsilon) \cdot N\}\). We can verify that \(S_{\epsilon}\) is an \((\epsilon, s)\)-FE of \(B\). □
Example 4.6  From the $0.02$-FC shown in Example 4.5, we can construct an $(\epsilon, s)$-FE with $\epsilon = 0.02$ and $s = 0.1$ by selecting all elements with approximate frequencies at least $(0.1 - 0.02) \cdot 100 = 8$, which results in the set $\{a, b, c, d\}$. □

4.3 Problem Statement

In this section, we present simpler definitions for sliding windows that omit details not relevant to the problems of this chapter, and therefore more convenient for presenting our algorithms. We then formally state the problems that we are addressing in this chapter. For the rest of this section, we fix a stream $S$ that is an infinite sequence of values (elements) $a_1, a_2, \ldots$. For quantiles the elements are drawn from some ordered domain, and for frequency counts, the domain is arbitrary.

4.3.1 Sliding Windows

Our sliding-window definitions of Section 2.6.1 involve many details such as timestamps for stream elements, time units (e.g., Minutes, Seconds), and syntactic conveniences (e.g., Now) orthogonal to the problems studied in this chapter. We therefore present simpler definitions for sliding windows that discard the unnecessary details and retain only those relevant to our problems.

A sliding window over stream $S$ is a dynamic bag of elements. Elements are inserted into the bag in the same sequence as they occur in $S$ (i.e., $a_1, a_2, \ldots$), and they are deleted from the bag in the same sequence as well. Therefore, at any given point in time, the window contains some contiguous elements $[a_{m-N+1}, \ldots, a_m]$ of $S$. We use the term step to denote the unit change interval of the bag, i.e., the contents of the bag may change at every step. In particular, step does not correspond to any physical notion of time.

We define two types of sliding windows with different dynamics. A fixed-size window slides one element to the right at every step. If the current window is $[a_{m-N+1}, \ldots, a_m]$, in the next step, the window becomes $[a_{m-N+2}, \ldots, a_{m+1}]$. Note that the window size is always $N$. The window contains the elements $[a_1, \ldots, a_N]$ at the beginning.

In a variable-size window, at every step the left and the right ends slide independently. If the current window is $[a_{m-n+1}, \ldots, a_m]$, in the next step, the window can be either $[a_{m-n+1}, \ldots, a_{m+1}]$ or $[a_{m-n+2}, \ldots, a_m]$. In the former case, the right end slides by 1 element and the window size increases by 1, and in the latter, the left end slides and the
Fixed-Size Window | Variable-Size Window
--- | ---
0 | \{a_1\}
1 | \{a_1, a_2\}
2 | \{a_1, a_2, a_3\}
3 | \{a_2, a_3\}
4 | \{a_2, a_3, a_4\}
5 | \{a_2, a_3, a_4, a_5\}
6 | \{a_3, a_4, a_5\}
7 | \{a_4, a_5\}
8 | \{a_4, a_5\}

Figure 4.2: Example fixed-size and variable-size windows

window size decreases by 1. At the beginning, the window contains the single element \([a_1]\). Figure 4.2 shows the changing bags of an example fixed-size window with window size 3, and an example variable-size window over \(S\).

Fixed-size and variable-size windows abstract the essential features of tuple-based and time-based sliding windows, respectively. Tuple-based windows are equivalent to fixed-size windows, if we identify a step in the definition of fixed-size windows with the arrival of a new stream element in the definition of a tuple-based windows. The relationship between time-based windows and variable-size windows is less straightforward. We define a variable-size window \(V_w\) corresponding to a time-based window \(T_w\) as follows: Let there be \(i\) insertions into and \(d\) deletions from \(T_w\) at a time instant \(\tau\). We expand \(\tau\) to \(i + d\) steps for defining \(V_w\): In the first \(i\) steps, we insert new elements into \(V_w\), and in the next \(d\) steps, we delete the current oldest element from \(V_w\). Clearly, every state of \(T_w\) corresponds to some state of \(V_w\). Therefore, an algorithm that computes statistics over \(V_w\) can be used to compute statistics over \(T_w\).

### 4.3.2 Problem Statement

The formal problem statements for computing approximate quantiles and approximate frequency counts over sliding windows is given below. Recall from Section 4.2.2 that we use approximate frequency counts as an intermediate statistic to compute approximate
frequent elements.

**Problem Statement (\(\epsilon\)-approximate quantiles over sliding windows)** Maintain sufficient state so that, at any point in time and any for \(\phi \in (0, 1]\), \((\epsilon, \phi)\)-quantiles for the current bag of elements in the sliding window can be computed.

**Problem Statement (\(\epsilon\)-approximate frequency counts over sliding windows)** Maintain sufficient state so that, at any point in time, \(\epsilon\)-approximate frequency counts for the current bag of elements in the sliding window can be computed.

We seek memory-efficient algorithms for the problems above. For each problem, we study two variants corresponding to the two window types, fixed-size and variable-size. We require that our algorithms work for any value of error parameter \(\epsilon \in (0, 1]\). However, \(\epsilon\) remains fixed within a given run of an algorithm, and we assume that \(\epsilon\) is provided as an input parameter to the algorithms at the time of their instantiation. Similarly, for algorithms over fixed-size windows, we assume that the window size \(N\) is provided as an input parameter at the beginning.

We note that our problem formulation is very general. We do not require our algorithms to actually compute any statistic; the algorithms just need to maintain enough state to enable computation of the relevant statistic. By this definition, the simple algorithm that stores all the elements in the current window is a correct one, but our goal is to design algorithms that use exponentially less memory than that required by this simple algorithm.

We also note that our problem formulation for quantiles requires an algorithm to maintain state for computing any quantile parameter \(\phi \in (0, 1]\), not just a single one. In fact, we do not know how to design algorithms for specific quantile parameters that are more memory-efficient than the more general algorithms that we present here. However, some preliminary evidence suggests that the memory required for maintaining a single quantile parameter is as high as the memory required for maintaining all of them [65].

**Algorithm Input**

The input to an algorithm maintaining statistics over a fixed-size window of stream \(S\) is the sequence of insertions of elements into the window, i.e., the sequence \(a_1, a_2, \ldots\). After \(N\) insertions, the algorithm should automatically expire the oldest element for every new insertion. The input to an algorithm maintaining statistics over a variable-size window is
a sequence of new element insertions and deletions. For deletions, the identity of deleted elements is not provided in the input: The algorithm has sufficient information to deduce the identity of a deleted element, since the deleted element is always the oldest element in the current window.

**Example 4.7** An example input sequence to a variable-size window algorithm is:

\[ a_1, a_2, a_3, d, a_4, a_5, d, d, \ldots \]

where \( d \) denotes a deletion. This sequence corresponds to the variable-size window shown in Figure 4.2.

4.4 Sketches

All of our algorithms are designed using various other algorithms as building blocks. Specifically, our algorithms for variable-size windows use our algorithms for fixed-size windows as subroutines, which in turn use previously known summarization techniques for approximate statistics computation as subroutines. In order to precisely describe how one algorithm invokes other algorithms as subroutines, we add additional “structure” to the algorithms: We represent the algorithms we deal with as stateful objects\(^3\), called sketches, with a well-defined interface to access and update the state; one algorithm invokes another using the latter’s interface.

Specifically, sketches summarize a bag of elements, and support at least one operation called **query**, which computes approximate statistics over the bag. For sketches representing our algorithms for variable- and fixed-size sliding windows, the bag of elements is dynamic, and defined by a sliding window. These **sliding-window sketches** support two additional operators: **insert**, to insert a new element into the bag, and **delete**, to delete the oldest element from the bag. We introduce sliding-window sketches formally in Section 4.4.2. As suggested earlier, our sliding-window sketches use simpler sketches called **static sketches** which we describe next.

---

\(^3\) A stateful object model is more suitable than a stateless functional model, since all the algorithms that we deal with are continuous, not one-time.
4.4.1 Static Sketches

A static sketch summarizes a fixed, unchanging bag of elements. The only operation supported by a static sketch is QUERY, which computes some approximate statistic over the bag. In this section, we introduce two static sketches, one for quantiles and one for frequency counts. As mentioned earlier, we will use these sketches as building blocks within our sliding-window sketches.

Static Sketches for Quantiles

Consider a bag $B$ of $N$ elements. A static sketch of $B$ for a particular quantile parameter (say $\phi = 0.5$, the median) is trivial: the sketch precomputes the (exact) value of the quantile and stores it; QUERY simply returns the stored value. We are interested in sketches that allow any $(\epsilon, \phi)$-quantile to be computed. The following lemma shows how we can design such sketches.

**Lemma 4.1** For any bag $B$ of $N$ elements, there exists a static sketch that uses $O(1/\epsilon)$ space and allows $(\epsilon, \phi)$-quantiles of $B$ to be computed, for any $\phi \in (0, 1]$.

**Proof:** We precompute and store $\phi$-quantiles for $\phi = \epsilon, 3\epsilon, 5\epsilon, \ldots, 1$ as part of the sketch. The space used by the sketch is clearly $O(1/\epsilon)$. For any $\phi \in (0, 1]$, let $\phi'$ be the number of the form $(2k + 1)\epsilon$ that is closest to $\phi$. We observe that the $\phi'$-quantile of $B$ is an $(\epsilon, \phi)$-quantile of $B$, and, by our construction, the $\phi'$-quantile of $B$ was stored in the sketch. The lemma follows. \qed

A naive algorithm for constructing a quantile sketch for $B$ would be to sort the elements of $B$, and pick the elements at positions $\lceil \epsilon N \rceil, \lceil 3\epsilon N \rceil, \ldots$ (these are the $\phi$-quantiles for $\phi = \epsilon, 3\epsilon, \ldots$), as suggested by the proof of Lemma 4.1. However, this algorithm requires $O(N)$ “scratch” space during the construction of the sketch. Greenwald and Khanna [58] show that we can do much better, as summarized in Lemma 4.2.

**Lemma 4.2** [58] For any bag $B$ of $N$ elements, we can construct a static quantile sketch that uses $O(1/\epsilon)$ space and allows $(\epsilon, \phi)$-quantiles to be computed for any $\phi \in (0, 1]$. Further, the construction itself makes one pass over $B$ and uses $O(\frac{1}{\epsilon} \log(\epsilon N))$ scratch space.

We refer to sketches constructed by the Greenwald and Khanna algorithm as $GK$-sketches.
Static Sketches for Frequency Counts

Consider a bag $B$ of $N$ elements for which we need to construct a static frequency-counts sketch. Unlike quantiles, $\epsilon$-$FC$ is a single statistic, so we can simply precompute an $\epsilon$-$FC$ for $B$ and store it in the sketch. Recall from Section 4.2.2 that an $\epsilon$-$FC$ is required to contain an entry for every element of $B$ with frequency $\geq \epsilon N$; it may or may not contain an entry for other elements. Since there are at most $1/\epsilon$ elements with frequency $\geq \epsilon N$, there always exists an $\epsilon$-$FC$ for $B$ of size $O(1/\epsilon)$. Misra and Gries [78] show that an $\epsilon$-$FC$ for $B$ of size $O(1/\epsilon)$ can be constructed in one pass over $B$:

**Lemma 4.3**  For any bag $B$ of $N$ elements, we can construct an $\epsilon$-$FC$ (and hence a static frequency-count sketch) of size $O(1/\epsilon)$ in one pass over $B$ using $O(1/\epsilon)$ scratch space.

We call the sketches constructed by the Misra-Gries algorithm as $MG$-sketches. An $MG$-sketch for a bag $B$ is just a compact $\epsilon$-$FC$ of $B$.

4.4.2 Sliding-Window Sketches

A sliding-window sketch allows an approximate statistic to be computed over a dynamic bag of elements defined by a sliding window. A sliding-window sketch supports three operations:

1. **QUERY**: returns an approximate value of the statistic over the current bag of elements. (In quantile sketches, QUERY takes a parameter $\phi \in (0, 1]$, and returns an approximate $\phi$-quantile.)

2. **INSERT($a_m$)**: inserts a new element into the current bag of elements.

3. **DELETE**: deletes the oldest element (i.e., the element that was inserted the earliest) from the current bag of elements.

We recall that any dynamic bag of elements where a deletion always deletes the oldest element can be thought of as a sliding window over a stream, where the order of insertions defines the sequence of elements in the stream. We emphasize that the sketch need not explicitly materialize the bag and perform insertions and deletions; the insertions and deletions only serve to logically define the current bag of elements for the purposes of querying. We also note that the **DELETE** operation is not provided the element to be deleted as input, and the sketch needs to deduce the oldest element by itself.
We now introduce two kinds of sliding-window sketches: unbounded-window sketches and bounded-window sketches. Unbounded-window sketches are used to compute statistics over variable-size windows, and bounded-window sketches are used to compute statistics over fixed-size windows. These two sketches differ in the error guarantees that their QUERY operation provides, and in some additional constraints they have on the size of bags that they can handle.

**Definition 4.4 (Unbounded-Window Sketch)** An unbounded-window sketch is defined using a error parameter. We denote an unbounded-window sketch $S$ with error parameter $\epsilon$ as $S_\epsilon$. At any given point in time, $S_\epsilon$ allows an $\epsilon$-approximate statistic to be computed over the current bag of elements. An unbounded-window sketch has no constraints on the size of the bag of elements: the bag can grow and shrink arbitrarily, unlike a bounded-window sketch.

**Definition 4.5 (Bounded-Window Sketch)** A bounded-window sketch is defined using two parameters: an error parameter $\epsilon$ and a max-window parameter $W$. We denote a bounded-window sketch $S$ with error parameter $\epsilon$ and max-window parameter $W$ as $S_{W,\epsilon}$. At any given point in time, $S_{W,\epsilon}$ allows an $(\epsilon W/N)$-approximate statistic to be computed over the current bag of elements, where $N$ is the size of the bag. Further, the size of the bag is constrained to be $\leq W$ at all times.

Both bounded- and unbounded-window sketches allow approximate statistics to be computed over variable-size windows. For bounded-window sketches, the size of the window is constrained to be less than a fixed max-window parameter $W$, while there are no such constraints for unbounded-window sketches.

We can use bounded- and unbounded-window sketches to compute approximate statistics over fixed-size and variable-size sliding windows, respectively. For computing $\epsilon$-approximate statistics over a fixed-size window of size $N$, we use a bounded-window sketch $S$ for that statistic with error parameter $\epsilon$ and max-window parameter $N$, i.e., $S_{N,\epsilon}$. When the window slides by one element, we delete the oldest element from $S_{N,\epsilon}$ using the DELETE operation, and insert the new element using the INSERT operation. By definition, we can use the QUERY operation to compute $\epsilon N/N = \epsilon$-approximate statistics over the current window at any point in time. For computing an $\epsilon$-approximate statistic over variable-size windows, we simply use a unbounded-window sketch $S_\epsilon$ for that statistic,
which, by definition, allows ε-approximate statistics to be computed over the current window at any point in time.

Our definition of bounded-window sketches is more general than required for computing statistics over fixed-size windows. A bounded-window sketch allows the size of the bag to vary between 0 and \( W \), where \( W \) is the max-window parameter of the sketch. However, for computing statistics over fixed-size windows, we keep the size fixed at \( W = N \), the window size. We use the additional generality in the definition for constructing unbounded-window sketches from bounded-window sketches, as described in Section 4.7.

We also define randomized variants of these sketches: A randomized (bounded-window or unbounded-window) sketch is similar to the corresponding non-randomized sketch, except that the statistic computed by the QUERY operation is ε-approximate with a probability \((1 - \delta)\), where \( \delta \) denotes the failure probability of the sketch. We use \( S_{\epsilon, \delta} \) to denote an unbounded-window sketch \( S \) with error parameter \( \epsilon \) and failure probability \( \delta \), and \( S_{W, \epsilon, \delta} \) to denote a bounded-window sketch \( S \) with max-window \( W \), error parameter \( \epsilon \), and failure probability \( \delta \).

### 4.5 Deterministic Bounded-Window Sketches

In this section, we present two deterministic bounded-window sketches called \( Q \) and \( C \) for quantiles and frequency counts, respectively. For the rest of this section, we fix \( \epsilon \in (0, 1) \) to be an arbitrary error parameter, and \( W \) to be an arbitrary max-window parameter. We will describe how \( Q_{W, \epsilon} \) and \( C_{W, \epsilon} \) are constructed. Since \( Q_{W, \epsilon} \) and \( C_{W, \epsilon} \) are very similar, we will combine our presentation of \( Q_{W, \epsilon} \) and \( C_{W, \epsilon} \) for the most part.

We assume that \( 1/\epsilon \) and \( W \) are both powers of 2 to avoid floors and ceilings in expressions. In order to construct \( C_{W, \epsilon} \) for arbitrary \( W \) and \( \epsilon \), we identify the unique \( W' \) and the maximum \( \epsilon' \) such that:

1. \( W' \) and \( 1/\epsilon' \) are powers of 2,
2. \( W \leq W' < 2W \), and
3. \( W' \epsilon' \leq W \epsilon \).

We then use the sketch \( C_{W', \epsilon'} \) in place of \( C_{W, \epsilon} \). By construction, \( C_{W', \epsilon'} \) is more accurate than \( C_{W, \epsilon} \), and we can show that both have the same asymptotic space complexity. We also
assume that $W \gg (1/\epsilon)$. Otherwise, a simple construction of $C_{W,\epsilon}$ and $Q_{W,\epsilon}$ is to store all the elements in the current window, and use these to compute exact counts and quantiles. This construction has space complexity $O(1/\epsilon)$, which is smaller than the space complexity of $Q_{W,\epsilon}$ and $C_{W,\epsilon}$ that we present for a general $W$.

We assume that $C_{W,\epsilon}$ and $Q_{W,\epsilon}$ are used to maintain statistics over a sliding window of stream $S$ with elements $a_1, a_2, \ldots$. We describe the state maintained by $C_{W,\epsilon}$ and $Q_{W,\epsilon}$ when the current window is $[a_{m-N+1}, \ldots, a_m], N \leq W$.

### Blocks and Levels

To describe the state maintained by $C_{W,\epsilon}$ and $Q_{W,\epsilon}$, we first introduce the notion of blocks and levels: We conceptually make $L + 1$ copies of $S$, where $L = \log_2(4/\epsilon)$. We say that these copies are at different levels, which are numbered sequentially as 0, 1, $L$. For each level, the copy of the stream for that level is divided into non-overlapping blocks. Within each level-$\ell$, all blocks have the same size, i.e., the same number of elements. We denote the size of level-$\ell$ blocks by $N_\ell$. We set $N_0 = \frac{\epsilon W}{4}$ and for $\ell \geq 1$, $N_\ell = 2N_{\ell-1}$. It follows that $N_\ell = 2^\ell N_0 = \frac{\epsilon W}{4}2^\ell$. We observe that $N_{L} = W$. Within a level, blocks are numbered 0, 1, $\ell$; smaller numbered blocks contain older elements. For example, in level 0, block 0 contains the first $\frac{\epsilon W}{4}$ elements, block 1 the next $\frac{\epsilon W}{4}$, and so on. In general, block $b$ in level $\ell$ contains the bag of elements $\{a_i\}, i \in [b\ell^\ell\frac{\epsilon W}{4} + 1, (b+1)2^\ell\frac{\epsilon W}{4}]$. Figure 4.3 schematically illustrates blocks and levels.

At any given point in time, a block is assigned to one of four states, depending on $m$, the index of the last element $a_m$ in the current window. Consider block $b$ of level $\ell$, which
contains the bag of elements \( \{a_i\}, i \in [b2^\ell \ell W + 1, (b + 1)2^\ell \ell W] = [l, r] \). The state of this block is:

1. active, if \( m - N < l \leq r \leq m \);
2. expired, if \( l \leq m - N \);
3. partially active, if \( m - N < l \leq m \) and \( r > m \);
4. inactive, if \( l > m \).

A block is active if all its elements are in the current window; expired if at least one of its elements has been deleted from the window; partially active if some of its elements are in the window and the rest are yet to be inserted; and inactive if none of the elements in the block have been inserted into the window yet. Each block is initially inactive, becomes partially active, then (optionally) becomes active, and finally becomes expired. See Figure 4.3 for an illustration of blocks and their states.

**State maintained by \( C_{W,\epsilon} \) and \( Q_{W,\epsilon} \)**

\( C_{W,\epsilon} \) and \( Q_{W,\epsilon} \) store static sketches, one for elements of each active block. \( Q_{W,\epsilon} \) uses \( GK \)-sketches, and \( C_{W,\epsilon} \) \( MG \)-sketches. The sketch corresponding to a level-\( \ell \) block has an error parameter \( \epsilon_\ell = \frac{\epsilon}{2^\ell (2L + 2)} 2^{L-\ell} \). The error parameter halves as we go from one level to the next higher, so sketches for higher-level blocks are more accurate than sketches for lower-level blocks.

The *query* operation uses the static sketches corresponding to active blocks collectively to compute approximate statistics (quantiles or frequency counts). The details of how this computation is done differs for frequency counts and quantiles, and we present them separately in Sections 4.5.1 and 4.5.2, respectively.
We now describe how the static sketches are constructed. From Section 4.4, we know that there exist algorithms for constructing static sketches over any bag $B$ in one scan of $B$. When the first element of a block is inserted into the window, i.e., when the block transitions from inactive state to partially active state, we start an instance of the one-pass algorithm. When a new element of the block is inserted into the window, we input the element to this algorithm. When all the elements of the block have been inserted into the window, and the block becomes active, we terminate the algorithm and get its output static sketch. The one-pass algorithm instances require some state while running, and we will factor this state into our analysis of $C_{W, \varepsilon}$ and $Q_{W, \varepsilon}$.

4.5.1 Details of $C_{W, \varepsilon}$

We now describe how $C_{W, \varepsilon}$ computes $\varepsilon$-approximate frequency counts over the current window, and we analyze its space requirement. We present both these details within the proof of Theorem 4.1. We begin by presenting a lemma that shows how $MG$-sketches for different bags can be combined to compute an $\varepsilon$-FC for the union of the bags.

**Lemma 4.4** Consider bags $B_1, \ldots, B_s$ with sizes $N_1, \ldots, N_s$, respectively. Given $MG$-sketches for these bags with error parameters $\varepsilon_1, \ldots, \varepsilon_s$, respectively, we can compute an $\varepsilon$-FC for $B_1 \cup \cdots \cup B_s$, with error $\varepsilon = \frac{\varepsilon_1 N_1 + \varepsilon_2 N_2 + \cdots + \varepsilon_s N_s}{N_1 + N_2 + \cdots + N_s}$.

**Proof:** We query each $MG$-sketch and compute $A_1, \ldots, A_s$, where $A_i$ is an $\varepsilon_i$-FC of $B_i$. We construct an approximate frequency counts $A$ for $B_1 \cup \cdots \cup B_s$ as follows: An element $e$ occurs as part of $A$ if and only if $e$ occurs as part of some $A_i$ ($1 \leq i \leq s$), i.e., $A_i$ contains a pair of the form $\langle e, \tilde{f}_{ei} \rangle$. If so, $A$ contains the pair $\langle e, \tilde{f}_e \rangle$. The approximate frequency $\tilde{f}_e$ is the sum of the approximate frequencies $\tilde{f}_{ei}$ of $e$ stored in each $A_i$. (If $A_i$ does not store the approximate frequency of $e$, we take $\tilde{f}_{ei}$ to be 0.) By definition of $A_i$, $f_{ei} - \varepsilon_i N_i \leq \tilde{f}_{ei} \leq f_{ei}$, where $f_{ei}$ is the true frequency of $e$ in bag $B_i$. It follows that $f_e \geq \frac{\varepsilon_1 N_1 + \cdots + \varepsilon_s N_s}{N_1 + N_2 + \cdots + N_s}$, where $f_e = f_{e1} + \cdots + f_{es}$ is the true frequency of $e$ in the union of the bags. If the (true) frequency $f_e$ of an element $e$ in the union of the bags is at least $\frac{\varepsilon_1 N_1 + \cdots + \varepsilon_s N_s}{N_1 + N_2 + \cdots + N_s}$, then the frequency $f_{ei}$ of $e$ in at least one of the bags $B_i$ exceeds $\varepsilon_i N_i$, implying that $e$ occurs in $A_i$. By construction, $e$ occurs in $A$ as well. Therefore $A$ is an approximate frequency count for the union of the bags with error parameter $\frac{\varepsilon_1 N_1 + \cdots + \varepsilon_s N_s}{N_1 + N_2 + \cdots + N_s}$. □
Theorem 4.1 \( C_{W,e} \) allows an \( \epsilon W \)-FC to be computed over the current window \( \{a_{m-N+1}, \ldots, a_m\} \). Further, \( C_{W,e} \) uses \( O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon}) \) space.

Proof: We prove the approximation part and the space requirement part separately.

Approximation

Let \( B_w = \{a_{m-N+1}, \ldots, a_m\} \) denote the bag of elements in the current window. Let \( \alpha \) be the smallest integer such that \( \alpha \frac{W}{N} \geq m - N \). Let \( \beta \) be the largest integer such that \( \beta \frac{W}{N} \leq m \). Let \( B_{ig} \) (for bag of ignored elements) denote the bag of all elements \( a_i \) such that \( m - N < i \leq \alpha \frac{W}{N} \) or \( \beta \frac{W}{N} < i \leq m \). Intuitively, \( B_{ig} \) denotes the bag of elements that belong to the current window but do not belong to any active block. Let \( B_{con} \) (for considered elements) denote the bag of all elements \( e_i \) such that \( \alpha \frac{W}{N} + 1 \leq i \leq \beta \frac{W}{N} \). \( B_{con} \) denotes the bag of elements belonging to the current window that also belong to some active block. Note that \( B_w = B_{ig} \cup B_{con} \).

We claim without proof that \( B_{con} \) can be expressed as a union of \( k \) non-overlapping active blocks such that \( k \leq (2L + 2) \). (See Figure 4.3 for an illustration.) Let \( B_1, \ldots, B_k \) denote the bag of elements belonging to these blocks. Let \( A_1, \ldots, A_k \) denote the sketches stored by \( C_{W,e} \) for these blocks. Let \( A_{ig} \) denote the trivial (empty set) sketch with error parameter \( \epsilon_{ig} = 1 \) for the bag of elements \( B_{ig} \). We use Lemma 4.4 to compute an frequency approximate counts \( A \) for the bag of elements \( B_w = B_1 \cup \ldots \cup B_k \cup B_{ig} \) using the sketches \( A_1, \ldots, A_k, A_{ig} \).

We now prove that \( A \) is an \( \epsilon W \)-approximate count for \( B_w \). Let \( \epsilon_i \) denote the approximation parameter for sketch \( A_i \) (1 \( \leq i \leq k \)). Let \( N_i \) denote the number of elements in \( B_i \), and let \( N_{ig} \) denote the number of elements in \( B_{ig} \). We first note that for any level \( \ell \), \( \epsilon_i N_{i\ell} = \frac{\epsilon W}{2(2L+2)} \), which is independent of \( \ell \). Since each \( A_i \) is some block-level sketch, \( \epsilon_i N_i = \frac{\epsilon W}{2(2L+2)} \) for (1 \( \leq i \leq k \)). Using Lemma 4.4, the approximation parameter for \( A \) (say \( \epsilon_A \)) is given by:

\[
\epsilon_A = \frac{(\epsilon_{ig} N_{ig} + \sum_{i=1}^{k} \epsilon_i N_i)}{N} \tag{4.1}
\]

\[
= \frac{(N_{ig} + k \cdot \frac{\epsilon W}{2(2L+2)})}{N} \tag{4.2}
\]

\[
\leq \frac{(N_{ig} + \frac{\epsilon W}{2})}{N} \tag{4.3}
\]
CHAPTER 4. APPROXIMATE STATISTICS OVER STREAMS

\begin{align*}
\leq \left( \frac{\epsilon W}{2} + \frac{\epsilon W}{2} \right) / N \quad &\text{(4.4)} \\
\leq \frac{\epsilon W}{N} \quad &\text{(4.5)}
\end{align*}

Equation 4.3 follows from the fact that \( k \leq (2L + 2) \), and Equation 4.4 from the fact that \( N_{ig} \leq 2 \frac{\epsilon W}{4} \) (which can be shown from the definition of \( B_{ig} \)).

**Space Requirement**

We divide the space required by \( C_{W, \epsilon} \) into two parts: the space required to store the \( MG \)-sketches for active blocks, and the space required by one-pass algorithm instances currently running over elements of partially active blocks.

There are at most \( \frac{N}{4W/4} \) active blocks at level 0; this quantity is \( \leq \frac{1}{2} \), since \( N \leq W \). Since \( L = \log_2(4/\epsilon) \), the number of active blocks at level 0 is bounded from above by \( 2^L \). We can show that the maximum number of active blocks at level 1 is \( 2^{L/2} \) and, more generally, the maximum number of active blocks at level \( \ell \) is \( 2^{L-\ell} \). An \( MG \)-sketch for a level-\( \ell \) block has error parameter \( \epsilon_\ell \), and so requires \( O\left( \frac{1}{\epsilon_\ell} \right) \) space (Lemma 4.3). The total space required for \( MG \)-sketches of all active blocks is therefore (ignoring constants):

\[
\sum_{\ell} \frac{2^{L-\ell}}{\epsilon_\ell} = \sum_{\ell} \frac{2^{L-\ell}}{\epsilon} \cdot \frac{2(2L + 2)}{\epsilon^2} \quad &\text{(4.6)} \\
= \sum_{\ell} \frac{2(2L + 2)}{\epsilon} \quad &\text{(4.7)} \\
= \frac{4}{\epsilon} \sum_{\ell} (L + 1) \quad &\text{(4.8)} \\
= O\left( \frac{1}{\epsilon} \log^2 \frac{1}{\epsilon} \right) \quad &\text{(4.9)}
\]

We get Equation 4.6 by substituting \( \epsilon_\ell = \frac{\epsilon}{2(2L+2)} 2^{(L-\ell)} \), and Equation 4.9 by substituting \( L = \log_2 \frac{4}{\epsilon} \).

To compute the space required by the one-pass, static-sketch construction algorithms, we note that there can be at most one partially active block at any given level, at any given point in time. Using Lemma 4.3, an instance of the algorithm at level \( \ell \) requires \( O\left( \frac{1}{\epsilon_\ell} \right) \) space. The total space required by all instances of the algorithm is therefore bounded by
(ignoring constants):

\[
\sum_{0}^{L} \frac{1}{\epsilon_{\ell}} = \sum_{0}^{L} \frac{2(2L+2)}{\epsilon 2^{L-\ell}} \tag{4.10}
\]

\[
= \frac{4(L+1)}{\epsilon} \sum_{0}^{L} \frac{1}{2^{L-\ell}} \tag{4.11}
\]

\[
= O\left(\frac{L}{\epsilon}\right) \tag{4.12}
\]

\[
= O\left(\frac{1}{\epsilon \log \frac{1}{\epsilon}}\right) \tag{4.13}
\]

Equation 4.12 follows from Equation 4.11, since the summation in Equation 4.11 is bounded by a constant.

The total space requirement of \(C_{W,\epsilon}\) is therefore \(O\left(\frac{1}{\epsilon \log \frac{1}{\epsilon}}\right) + O\left(\frac{1}{\epsilon \log \frac{1}{\epsilon}}\right) = O\left(\frac{1}{\epsilon \log \frac{1}{\epsilon}}\right)\)

\(\square\)

### 4.5.2 Details of \(Q_{W,\epsilon}\)

We now present the details of \(Q_{W,\epsilon}\): We show that \(Q_{W,\epsilon}\) can compute \((\epsilon, \phi)\)-quantiles for any \(\phi \in (0, 1]\), and analyze its space requirement. Mirroring our presentation of \(C_{W,\epsilon}\), we begin with a lemma that states how \(GK\)-sketches for different bags can be combined to compute an \((\epsilon, \phi)\)-quantile for the union of the bags, for any \(\phi \in (0, 1]\). We do not include a proof of the lemma, which, unfortunately, requires a detailed introduction to \(GK\)-sketches. We refer the interested reader to references [59, 71], which contain proofs for slight variants of the lemma.

**Lemma 4.5** Consider bags \(B_1, \ldots, B_s\) with sizes \(N_1, \ldots, N_s\), respectively. Given \(GK\)-sketches for these bags with error parameters \(\epsilon_1, \ldots, \epsilon_s\), respectively, we can compute an \((\epsilon, \phi)\)-quantile for \(B_1 \cup \cdots \cup B_s\) for any \(\phi \in (0, 1]\), with error \(\epsilon = \frac{\epsilon_1 N_1 + \cdots + \epsilon_s N_s}{N_1 + \cdots + N_s}\).

**Theorem 4.2** For any \(\phi \in (0, 1]\), \(Q_{W,\epsilon}\) allows \((\frac{1}{N}, \phi)\)-quantiles to be computed over the current window \(\{a_{m-N+1}, \ldots, a_m\}\). Further, \(Q_{W,\epsilon}\) uses \(O\left(\frac{1}{\epsilon \log \frac{1}{\epsilon \log W}}\right)\) space.

**Proof:** The proof of the approximation part is identical to that of Theorem 4.1, using Lemma 4.5 instead of Lemma 4.4.
As we did for $C_{W,\epsilon}$, we divide the space required by $Q_{W,\epsilon}$ into two parts: the space required to store the $GK$-sketches for the elements of the active blocks, and the space required by the instances of the one-pass algorithm used to construct $GK$-sketches.

The memory requirement of $GK$-sketches is similar to that of $MG$-sketches. A $GK$-sketch for a level-$\ell$ block, with error parameter $\epsilon_\ell$, requires $O(\frac{1}{\epsilon_\ell})$ memory. The total memory required to store the $GK$-sketches for all the active blocks is therefore identical to the total memory required to store the $MG$-sketches in $C_{W,\epsilon}$, which we know to be $O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon})$ from the proof of Theorem 4.1.

An instance of the one-pass algorithm to compute $GK$-sketches for level $\ell$, with error parameter $\epsilon_\ell$, requires $O(\frac{1}{\epsilon_\ell} \log(\epsilon_\ell N_\ell))$ space (using Lemma 4.1). Using the fact that $\epsilon_\ell N_\ell = \frac{\epsilon W}{2(2L+2)}$, and the fact that there is at most one instance of the algorithm for each level, at any given point in time, we can bound the total space required by the one-pass algorithms by:

$$\sum_{\ell=0}^{L} \frac{1}{\epsilon_\ell} \log \frac{\epsilon W}{2(2L+2)} = \log \frac{\epsilon W}{2(2L+2)} \sum_{\ell=0}^{L} \frac{1}{\epsilon_\ell} \tag{4.14}$$

$$= \log \frac{\epsilon W}{2(2L+2)} O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon}\right) \tag{4.15}$$

$$= O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log \frac{\epsilon W}{\log(1/\epsilon)}\right) \tag{4.16}$$

We get Equation 4.15 using the fact $\sum_{\ell=0}^{L} \frac{1}{\epsilon_\ell} = O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$, which was part of the proof of Theorem 4.1.

Therefore, the overall space requirement is $O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon}) + O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log \frac{\epsilon W}{\log(1/\epsilon)}\right)$, which can be simplified to the more conservative expression $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log W)$.

\section*{4.6 Randomized, Bounded-Window Sketches}

In this section, we present a randomized bounded-window sketch, $RQ$, for approximate frequency counts. We describe $R_{W,\epsilon,\delta}$ for an arbitrary error parameter $\epsilon$, max-window parameter $W$, and failure probability $\delta$. $R_{W,\epsilon,\delta}$ allows approximate frequency counts to be computed over the current window, and the approximation error is less than $\epsilon$ with a probability at least $1 - \delta$. As we did in Section 4.5, we assume that $R_{W,\epsilon,\delta}$ is used
to compute approximate frequency counts over a sliding window of stream $S$, with elements $a_1, a_2, \ldots$. We describe the state maintained by $RC_{W, \epsilon, \delta}$ when the current window is $[a_{m-N+1}, \ldots, a_m], N \leq W$.

$RC$ is based on a sampling technique called *sticky sampling* first proposed by Manku and Motwani [75]. We sample positions of $S$ independently with probability $p = \frac{1}{\epsilon W \log(\frac{1}{\delta})}$. For each position, we define a “indicator” random variable $Smp(i)$: $Smp(i) = 1$ if position $i$ was sampled, and $Smp(i) = 0$, otherwise. The state maintained by $RC_{W, \epsilon, \delta}$ for the current window is as follows: For every position $\langle a_i, pf_i \rangle$ is currently stored by $RC_{W, \epsilon, \delta}$. We call $pf_i$ the partial frequency of $a_i$, and we define it as follows:

$$ pf_i = |\{ j : (a_j = a_i) \land (i \leq j \leq m) \land \exists k((a_k = a_i) \land (i < k \leq j) \land (Smp(k) = 1))\} | $$

Intuitively, $pf_i$ denotes the number of occurrences of $a_i$ until $a_i$ is “sampled” again in the current window.

**Theorem 4.3** $RC_{W, \epsilon, \delta}$ allows an $\frac{\epsilon W}{N}$-FC to be computed over the current window $\{a_{m-N+1}, \ldots, a_m\}$ with a probability at least $(1 - \delta)$. Further, $RC_{W, \epsilon, \delta}$ uses $O(\frac{1}{\epsilon W \log(\epsilon \delta)} - 1)$ expected space.

**Proof:** We construct an approximate count $A$ of the current window in a natural way: For each distinct element $e$ that occurs in a sampled position in the current window, $A$ contains the pair $\langle e, \tilde{f}_e \rangle$, where $\tilde{f}_e$ is the sum of all partial frequencies of $e$, i.e., the sum of all $pf_i$ such that $\langle e, pf_i \rangle$ is currently stored by $RC_{W, \epsilon, \delta}$. We will prove that $A$ is an $\frac{\epsilon W}{N}$-FC of the current window with probability at least $1 - \delta$.

We will first focus on a particular element $e$, whose frequency $f_e$ in the current window is at least $\epsilon W$, and prove that, with a probability at least $(1 - \epsilon \delta)$, $A$ contains an entry $\langle e, \tilde{f}_e \rangle$ such that $f_e - \epsilon W \leq \tilde{f}_e \leq f_e$. Let $\{i_1, i_2, \ldots, i_{f_e}\}$ denote the set of positions in which $e$ occurs within the current window, in increasing order. Let $i_s$ denote the first of these positions that happens to be a sampled position. We can show that $\tilde{f}_e$ is the number of occurrences of element $e$ in the current window after position $i_s$. Therefore, $\tilde{f}_e = f_e - s$. Therefore, $A$ does not contain an entry for $e$ or $\tilde{f}_e < f_e - \epsilon W$ if and only if $s > \epsilon W$. The
probability of this event is:

\[
\Pr[s \geq \epsilon W] \leq (1 - p)^s \quad (4.17)
\]

\[
\leq (1 - p)^{\epsilon W} \quad (4.18)
\]

\[
\leq e^{-\epsilon W p} \quad (4.19)
\]

\[
e^{-\epsilon W \frac{1}{W} \log(\epsilon\delta)^{-1}} \quad (4.20)
\]

\[
e^{-\log(\epsilon\delta)^{-1}} \quad (4.21)
\]

\[
e\delta \quad (4.22)
\]

Therefore, with a probability at least \((1 - \epsilon\delta)\), \(A\) contains an entry \((e, \hat{f}_e)\) such that \(f_e - \epsilon W \leq \hat{f}_e \leq f_e\), as we claimed earlier.

Since \(N \leq W\), there are at most \(\frac{N}{\epsilon W} \leq \frac{1}{\epsilon} \) elements with frequency \(\geq \epsilon W\) in the current window. Using union bound [79], we can show that, with probability at least \((1 - \delta)\), for every such element \(e\), \(A\) contains an entry \((e, \hat{f}_e)\) such that \(f_e - \epsilon W \leq \hat{f}_e \leq f_e\), where \(f_e\) is the true frequency of \(e\) in the current window, proving that \(A\) is an \(\epsilon W/N\)-FC of the current window with probability at least \((1 - \delta)\).

We observe that the space required by \(RC_{W,\epsilon,\delta}\) is proportional to the number of sampled positions in the current window, which is \(pN = (\frac{1}{\epsilon W} \log \frac{1}{\epsilon\delta})N = O(\frac{1}{\epsilon} \log \frac{1}{\epsilon\delta})\) in expectation.

\(\square\)

### 4.7 Unbounded-Window Sketches

We present a general technique for constructing unbounded-window sketches from bounded-window sketches that satisfy a certain property. All of our bounded-window sketches presented in Sections 4.5 and 4.6 satisfy this property, and therefore we can obtain an unbounded-window sketch corresponding to each of our bounded-window sketches.

As we did in Sections 4.5 and 4.6, we use stream \(S = a_1, a_2, \ldots\) as a reference stream, and we describe our sketches for the case where they are maintain statistics for sliding windows over \(S\). We introduce some new notation: If \(F\) denotes a sliding-window sketch, \(F(m, n)\) denotes an instance of \(F\) whose current window is \([a_{m-n+1}, \ldots, a_m]\).
4.7.1 General Technique

Let $F$ denote a bounded-window sketch that satisfies the following property:

**PROPERTY P:** For any positive integers $k$ and $m$, and real $\epsilon \in (0, 1]$, $F_{2^{k+1}, \epsilon}(m, 2^k)$ can be constructed using $F_{2^k, \epsilon}(m, 2^k)$.

For any error parameter $\epsilon$, an unbounded-window sketch $V_\epsilon$ can be constructed from $F$ as follows: $V_\epsilon(m, n)$ is the collection of the following $\lfloor \log_2 \epsilon n \rfloor$ bounded-window sketches, where $k$ is the integer satisfying $2^k - 1 \leq n < 2^k$:

\[
\{F_{2^k, \epsilon}(m, n), F_{2^{k-1}, \epsilon}(m, 2^{k-1}), \ldots, F_{2^2, \epsilon}(m, \frac{2}{\epsilon})\}
\]

The three basic operations QUERY, INSERT, and DELETE of $V_\epsilon(m, n)$ are performed as follows:

- **QUERY:** To compute $\epsilon$-approximate statistics for the current window, we use the QUERY operation of the sketch $F_{2^k, \epsilon}(m, n)$. By definition, the QUERY operation of $F_{2^k, \epsilon}(m, n)$ produces $\left(\frac{\epsilon}{2}\right)^{2^k} n$-approximate statistics. Since $n > 2^k/2$, the approximation $\left(\frac{\epsilon}{2}\right)^{2^k} n < \epsilon$, as required.

- **INSERT:** We compute $V_\epsilon(m + 1, n + 1)$ from $V_\epsilon(m, n)$ when the element $a_{m+1}$ is inserted as follows: We insert the element $a_{m+1}$ into the sketch $F_{2^k, \epsilon}(m, n)$ using its INSERT operation to produce $F_{2^k, \epsilon}(m + 1, n + 1)$. For all the remaining sketches $\{F_{2^{k-1}, \epsilon}(m, 2^{k-1}), \ldots, F_{2^2, \epsilon}(m, \frac{2}{\epsilon})\}$, we first perform a DELETE operation and then insert the element $e_{m+1}$ using their INSERT operations. This sequence of operations results in the following collection of sketches:

\[
\{F_{2^k, \epsilon}(m + 1, n + 1), F_{2^{k-1}, \epsilon}(m + 1, 2^{k-1}), \ldots, F_{2^2, \epsilon}(m + 1, \frac{2}{\epsilon})\}
\]

Further, if $n + 1 = 2^k$, we construct $F_{2^{k+1}, \epsilon}(m + 1, 2^k)$ from $F_{2^k, \epsilon}(m + 1, 2^k)$ and add it to the collection of sketches. This step is possible since $F$ satisfies PROPERTY P.

- **DELETE:** We compute $V_\epsilon(m, n - 1)$ as follows: We perform a DELETE operation for the sketch $F_{2^k, \epsilon}(m, n)$. All other sketches remain unchanged. This update results in the collection of sketches:

\[
\{F_{2^k, \epsilon}(m, n - 1), F_{2^{k-1}, \epsilon}(m, 2^{k-1}), \ldots, F_{2^2, \epsilon}(m, \frac{2}{\epsilon})\}
\]
Further, if \( n - 1 < 2^{k-1} \), we drop the sketch \( F_{2^k, 2^k} (m, n - 1) \) from the collection of sketches.

### 4.7.2 Specific Constructions

In this section we show that all the bounded-window sketches that we presented in Sections 4.5 and 4.6 satisfy PROPERTY P.

**Lemma 4.6** The bounded-window sketches \( C \) and \( Q \) satisfy PROPERTY P.

**Proof:** We present the proof only for the bounded-window sketch \( C \). The proof for \( Q \) is very similar.

In order to prove that \( C \) satisfies PROPERTY P, we need to show that \( C_{2^{k+1}, \epsilon} (m, 2^k) \) can be constructed from \( C_{2^k, \epsilon} (m, 2^k) \). Intuitively, this construction is feasible since \( C_{2^k, \epsilon} (m, 2^k) \) is a more accurate sketch than \( C_{2^{k+1}, \epsilon} (m, 2^k) \): Both sketches compute statistics over the same window, but \( C_{2^k, \epsilon} (m, 2^k) \) allows \( \epsilon \)-approximate statistics to be computed over the current window, while \( C_{2^{k+1}, \epsilon} (m, 2^k) \) only allows \( 2\epsilon \)-approximate statistics to be computed.

For presentation clarity, we assume that no block of \( C_{2^k, \epsilon} (m, 2^k) \) is partially active. This assumption is equivalent to assuming that \( m \) is an exact multiple of \( 2^k \). The proof for the more general case uses essentially the same ideas as the proof presented here.

Let \( L = \log_2 (\frac{1}{\epsilon}) \) denote the number of different levels: This value is the same for both \( C_{2^{k+1}, \epsilon} (m, 2^k) \) and \( C_{2^k, \epsilon} (m, 2^k) \), since \( L \) is a function of \( \epsilon \) alone. For \( \ell < L \), we can show that...
each level-ℓ active block of $C_{2k+1,\epsilon}(m, 2^k)$ is the same as some level-(ℓ + 1) active block of $C_{2k,\epsilon}(m, 2^k)$. For this common block, $C_{2k,\epsilon}(m, 2^k)$ stores an MG-sketch with error parameter $\epsilon_{\ell+1}$, while $C_{2k+1,\epsilon}(m, 2^k)$ stores an MG-sketch with error parameter $\epsilon_{\ell}$. By definition, $\epsilon_{\ell+1} = \frac{\epsilon_{\ell}}{2}$. In general, we can show that an MG-sketch with error parameter $\epsilon'$ can be constructed from an MG-sketch with error parameter $\frac{\epsilon}{2}$. (We prove this fact formally in Lemma 4.7.) Therefore, for $\ell < L$, we can construct an MG-sketch for a level-ℓ active block of $C_{2k+1,\epsilon}(m, 2^k)$ using the MG-sketch maintained for the same block by $C_{2k,\epsilon}(m, 2^k)$. (See Figure 4.4 for an illustration.)

We now consider level-L blocks of $C_{2k+1,\epsilon}(m, 2^k)$. Since the current window size for $C_{2k+1,\epsilon}(m, 2^k)$ is $2^k$, and the size of a level-L block in $C_{2k+1,\epsilon}(m, 2^k)$ is $2^k+1$ (by definition), there can be no active level-L block in $C_{2k+1,\epsilon}(m, 2^k)$. We consider two possible cases depending on whether $C_{2k+1,\epsilon}(m, 2^k)$ contains a partially active level-L block or not. If $C_{2k+1,\epsilon}(m, 2^k)$ does not contain a partially active level-L block, we do not need to do anything further, and the construction of $C_{2k+1,\epsilon}(m, 2^k)$ is complete.

In order to be able to handle the second case, where $C_{2k+1,\epsilon}(m, 2^k)$ contains a partially active level-L block, we make one minor assumption about how we run the one-pass algorithm for constructing MG-sketches: When a block becomes active, we do not immediately terminate the instance of the algorithm as described in Section 4.5.1, but we do so only when the next block in the same level becomes partially active. None of the results so far are affected by this assumption. With this assumption, an instance of the one-pass algorithm, capable of producing an $\epsilon_L$-sketch is running over the elements of the single active level-L block of $C_{2k,\epsilon}(m, 2^k)$. We simply continue this algorithm on behalf of the level-L block of $C_{2k+1,\epsilon}(m, 2^k)$ that is partially active, and after the next $2^k$ elements arrive, this algorithm can be used to produce an $\epsilon_L$-sketch. With this step, the construction of $C_{2k+1,\epsilon}(m, 2^k)$ is complete.

\[\square\]

Lemma 4.7 For any bag of elements $B$ of size $N$, given an MG-sketch with error parameter $\frac{\epsilon}{2}$, we can construct an MG-sketch with error parameter $\epsilon$ that uses $O(\frac{1}{\epsilon})$ space.

Proof: We use the $\frac{\epsilon}{2}$-MG-sketch to first compute an $\frac{\epsilon}{2}$-FC of $B$. By definition, this contains a set of pairs $\{(e, f_e)\}$. We discard all pairs from this set with $f_e < \frac{N}{\frac{\epsilon}{2}}$, and retain the rest. We can show that the resulting set is an $\epsilon$-FC of $B$, which is also an MG-sketch of $B$, by definition. There can be at most $\left(\frac{2}{\epsilon}\right)$ elements with $f_e \geq \frac{N}{\frac{\epsilon}{2}}$, and therefore the size of the constructed sketch is $O(\frac{1}{\epsilon})$. \[\square\]
Lemma 4.8 The randomized bounded-window sketch \( RC \) satisfies PROPERTY P.

Proof: (Sketch) We need to show that \( RC_{2^{k+1},\epsilon,\delta}(m,2^k) \) can be constructed using \( RC_{2^k,\epsilon,\delta}(m,2^k) \). We can easily show that if \( RC_{2^k,\epsilon,\delta} \) samples positions with probability \( p \), \( RC_{2^{k+1},\epsilon,\delta} \) samples positions with probability \( \frac{p}{2} \). The construction of \( RC_{2^{k+1},\epsilon,\delta}(m,2^k) \) from \( RC_{2^k,\epsilon,\delta}(m,2^k) \) essentially uses the fact that a \( p/2 \) sample can be obtained from a \( p \) sample. \( \square \)

The theorems below follow directly from the general technique for constructing unbounded-window sketches and the space requirements of our bounded-window sketches.

Theorem 4.4 There exists an unbounded-window sketch \( V_\epsilon \) for computing \( \epsilon \)-approximate frequency counts, such that \( V_\epsilon(m,n) \) requires \( O\left(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon} \log \epsilon n\right) \) space.

Theorem 4.5 There exists an unbounded-window sketch \( V_\epsilon \) for computing \( \epsilon \)-approximate quantiles, such that \( V_\epsilon(m,n) \) requires \( O\left(\frac{1}{\epsilon} \log\frac{1}{\epsilon} \log n \log \epsilon n\right) \) space.

Theorem 4.6 There exists a randomized, unbounded-window sketch \( V_{\epsilon,\delta} \) for computing \( \epsilon \)-approximate frequency counts, such that \( V_{\epsilon,\delta}(m,n) \) requires \( O\left(\frac{1}{\epsilon} \log(\epsilon \delta)^{-1} \log \epsilon n\right) \) space.

4.8 Related Work

Frequency Counting Algorithms

For data streams, the earliest deterministic algorithm for \( \epsilon \)-approximate frequency counts is by Misra and Gries [78]. Their algorithm requires \( \frac{1}{\epsilon} \) space and \( O(1) \) amortized processing time per element. The same algorithm has been re-discovered recently by Demaine et al. [41] and Karp et al. [67], who reduced the processing time to \( O(1) \) in the worst case. Manku and Motwani [75] presented LOSSY COUNTING, a deterministic algorithm that requires \( O\left(\frac{1}{\epsilon^2} \log \epsilon N\right) \) space, when \( N \) elements of the stream have arrived. Though the space requirements are worse than the Misra-Gries algorithm, LOSSY COUNTING is superior when the input is skewed. Further, the algorithm can be adapted to compute association rules [2] over data streams.

In a random sample of size \( O\left(\frac{1}{\epsilon^2} \log (\epsilon \delta)^{-1}\right) \), the relative frequency of any element in the sample differs from its true frequency in the base dataset by at most \( \epsilon \). This observation can be exploited to obtain approximate frequency counts in a single pass such that the space
requirements are independent of $N$. For example, Toivonen [108] identifies a candidate set of frequent itemsets in the context of association rule mining [2]. For data streams, Manku and Motwani [75] presented STICKY SAMPLING, a randomized algorithm that requires only $O\left(\frac{1}{\epsilon} \log(\epsilon \delta)^{-1}\right)$ space, beating the sampling bound. Cormode and Muthukrishnan [35] recently presented randomized algorithms for identifying frequency counts in the presence of both additions and deletions. However, their algorithms assume that the identity of the deleted element is provided for a deletion, and are therefore not applicable to sliding windows where the identity of the deleted, oldest element is not provided as part of a deletion.

**Deterministic Quantile-Finding Algorithms**

The history of quantile-finding algorithms dates back to the early days of computer science. Early work focused on main memory datasets. The celebrated paper of Blum, Floyd, Pratt, Rivest, and Tarjan [27] shows that selecting the $k$th largest element among $N$ elements requires at least $1.5N$ and at most $5.43N$ comparisons. For an account of progress since then, see the survey by Paterson [87].

For large datasets in external memory, Pohl [91] established that any deterministic algorithm that computes the exact median in one pass needs to store at least $N/2$ data elements. Munro and Paterson [81] generalized the idea and showed that for $p \geq 2$, memory to store $\Theta(N^{1/p})$ elements is necessary and sufficient for finding the exact median (or any $\phi$-quantile) in $p$ passes.

For large-sized data streams, where only one pass is allowed, the lower bound of $N/2$ for the exact median [91] motivated the definition of approximate quantiles in the hope of reducing space to $o(N)$. Manku et al. [76] defined the notion of $\epsilon$-approximate quantiles and devised a deterministic one-pass algorithm that requires only $O\left(\frac{1}{\epsilon} \log^2 \epsilon N\right)$ space. However, their algorithm requires advance knowledge of an upper bound for $N$. Greenwald and Khanna [58] improved the space requirements to $O\left(\frac{1}{\epsilon} \log \epsilon N\right)$. Their algorithm does not require advance knowledge of $N$. Moreover, experiments indicate that their algorithm requires only $O\left(\frac{1}{\epsilon}\right)$ space if the input is a random permutation of elements.

For sliding windows, Lin et al. [71] recently devised the first algorithms for $\epsilon$-approximate quantiles. Their space bounds are $O\left(\frac{1}{\epsilon^2} + \frac{1}{\epsilon} \log(\epsilon^2 N)\right)$ for fixed-size windows and $O\left(\frac{1}{\epsilon^2} \log^2(\epsilon N)\right)$ for variable-size windows. Our algorithms have smaller space requirements: $O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log N\right)$ for fixed-size windows and $O\left(\frac{1}{\epsilon^2} \log \frac{1}{\epsilon} \log \epsilon N \log N\right)$ for variable-size
Randomized Quantile-Finding Algorithms

For identifying exact quantiles in main memory, a simple linear time randomized algorithm was presented by Floyd and Rivest [49]. For approximate quantiles, randomization reduces the space requirements significantly. The key insight is the well-known fact that the $\phi$-quantile of a random sample of size $O(1/\epsilon \log (\epsilon \delta)^{-1})$ is an $(\epsilon, \phi)$-quantile of $N$ elements with probability at least $1 - \delta$. This observation has been exploited, for example, by DeWitt et al. [43] to identify splitters of large datasets in the context of distributed sorting. For data streams, a randomized quantile-finding algorithm was proposed by Manku et al. [77] that requires only $O(1/\epsilon \log(1/\epsilon \log(\epsilon \delta)^{-1}))$ space, beating the sampling bound. Further improvement in space is possible, as described in Section 4.6. Recently Cormode and Muthukrishnan [36] proposed a sketching technique called CM-sketches that can be used to maintain $\epsilon$-approximate quantiles over streams and updateable relations. Their approach requires an advance knowledge of the domain $U$ from which the elements of the stream are drawn. The space requirement for their approach is $O(1/\epsilon \log^2 |U| \log(|U| / (\epsilon \delta)))$, where $|U|$ denotes the size of the domain $U$.

Related Problems

A problem related to approximate counting is the top-$k$ problem, also known as the Hot Item problem, where the goal is to identify the $k$ items that are the most frequent. Algorithms for the problem over data streams have been developed by Charikar et al. [31], Cormode and Muthukrishnan [35] and Gibbons and Matias [51].

Sliding window algorithms have been developed for a variety of problems: bit-counting (Datar et al. [39]), sampling (Babcock et al. [18]), variance and $k$-medians (Datar et al. [20], distinct values and bit-counts (Gibbons and Tirthapura [52]) and quantiles (Lin et al. [71]).

Frequent Elements and Quantiles over Graphical Processors

Recently, Govindaraju et al. [56] have explored the possibility of implementing algorithms for computing data stream statistics using Graphical Processing Units (GPUs) instead of traditional CPUs. They argue that GPUs, with their inherent parallelism and high memory bandwidth, are ideally suited for data stream processing. In support of this argument,
they have implemented the algorithms in this chapter over GPUs, and show significant performance improvements over a traditional CPU-based implementation.

4.9 Conclusion

In this chapter, we studied the problems of computing approximate frequent elements and quantiles over stream sliding windows. We first introduced two abstract models for sliding windows—fixed-size sliding windows and variable-size sliding windows—which capture the essential features of tuple-based windows and time-based windows. These abstract windows are mathematically very simple, and are useful for algorithm design.

We showed that the problem of computing approximate frequent elements can be reduced to the problem of computing approximate frequency counts. We focused on the latter statistic for most of the chapter.

We then introduced the notion of a sketch, which is a convenient representation for our algorithms and for the algorithms used within our algorithms. A sketch is a stateful object with a well-defined interface to access and update the state. We formalized two types of sketches: bounded-window sketches which represent algorithms for computing statistics over fixed-size windows, and unbounded-window sketches which represent algorithms for computing statistics over variable-size windows. We presented a deterministic bounded-window sketch for quantiles, and both deterministic and randomized bounded-window sketches for approximate frequency counts.

Instead of designing unbounded-window sketches from scratch, we presented a general technique for constructing an unbounded-window sketches using bounded-window sketches. We used our general technique to derive unbounded-window sketches corresponding to each of our bounded-window sketches.

All of our sketches are very compact: They all require \( O(\frac{1}{\epsilon^2} \text{polylog}(\epsilon, n)) \) space for computing \( \epsilon \)-approximate quantiles or frequency counts, where \( n \) denotes the current size of the window, and therefore represent memory-efficient solutions for computing these statistics over sliding windows.
Chapter 5

Resource Sharing

In the previous two chapters, our primary focus was on the memory required by individual continuous queries. In this chapter, we study another important aspect of continuous-query processing: sharing resources such as computation and memory across different continuous queries. The work presented in this chapter appeared in reference [14].

5.1 Introduction

Continuous queries require various resources such as CPU cycles, main memory, and disk storage and bandwidth for their execution. When there are multiple concurrent continuous queries, a DSMS needs to allocate the available resources among the different queries. One simple and natural strategy for resource allocation is to do the allocation at the granularity of individual queries: The available resources are partitioned among the competing queries, and there is no sharing of resources across different queries.

This simple resource allocation strategy can be very wasteful: Two or more queries may be very similar in the state that they maintain or the processing that they perform. In such cases, a better approach is to exploit the queries’ similarity to share resources such as memory and computation across queries. We provide two examples that illustrate such resource sharing.

Example 5.1 Consider two queries: $Q_1 = S_1 \bowtie S_2 \bowtie S_3$ and $Q_2 = S_1 \bowtie S_2 \bowtie S_4$. Since the join $S_1 \bowtie S_2$ is common to both queries, we can create just one instance of this join operator. The output of this operator can be joined with $S_3$ to get the final output for query $Q_1$, and
with \( S_4 \) to get the final output for query \( Q_2 \). This strategy shares both computation (to perform the join between \( S_1 \) and \( S_2 \)) and state (the state required by \( S_1 \bowtie S_2 \)) across the two queries.

**Example 5.2** Consider a large number of queries (say, millions) of the form \( \sigma_{A=k}(S) \), where \( k \) is a parameter that varies from one query to another, i.e., one query could be of the form \( \sigma_{A=1}(S) \), the second of the form \( \sigma_{A=5}(S) \), and so on. A naive evaluation strategy processes the queries independently: when a new tuple \( t \) arrives on stream \( S \), for each query, check if \( t \) satisfies the selection predicate of the query and if it does, produce \( t \) in the output of the query. A more efficient strategy is to process the queries collectively: we build, as a preprocessing step, a “predicate” index over the values of parameter \( k \) corresponding to all the queries. When a new tuple \( t \) arrives on stream \( S \), the index is used to quickly find the set of queries whose \( k \) parameter equals \( t.A \), and tuple \( t \) is produced in the output of these queries.

As we mentioned briefly in Section 1.1 (under “Resource Management”), continuous queries present more opportunities for resource sharing than traditional one-time queries. Resource sharing is usually feasible only among queries running concurrently (one-time or continuous), especially if the input data is highly dynamic as with data streams. Continuous queries, by definition, are long-running and therefore more likely to be concurrent with one-another than one-time queries.

Examples 5.1 and 5.2 illustrate two different types of resource sharing: Example 5.1 illustrates resource sharing techniques that identify and exploit common subexpressions between two or more queries. This type of resource sharing is related to multi-query optimization [93, 96], studied in the context of traditional one-time queries. We are more interested in the type of resource sharing illustrated in Example 5.2, where the participating queries do not necessarily share subexpressions. Here we collectively process a large number of structurally similar queries using a single, monolithic query plan. Example 5.2 presented a specific example that showed how we can process a large number of filter queries using a single predicate index.

In this chapter, we study the latter style of resource sharing for two classes of queries:

1. The first class of queries performs an aggregation over a sliding window (ASW). We introduced this class in Chapter 2, and used it extensively in Chapter 4. In our algebra, these queries are of the form \( \mathcal{G}_F(S[W]_T) \) or of the form \( \mathcal{G}_F(S[N]) \).
2. The second class of queries is called substream filters (SSF). A substream filter partitions a stream into substreams, applies an ASW subquery over each substream, and finally applies a filter over the resulting relation. In our algebra, these queries are of the form $\sigma_P(GG_F(S[W]_T))$.

**Example 5.3** Consider a stream of stock trades with the schema `StockTrades(symbol, price)`. An example of a substream filter (SSF) is “the set of stocks whose average price over the last 30 minutes is greater than 15.” In CQL (recall Chapter 2), this query can be expressed as:

```sql
Select symbol
From StockTrades [Range 30 Minutes]
Group By symbol
Having Avg(price) > 15
```

**Motivation**

Resource sharing is important for applications that involve a large number of concurrent continuous queries. For such applications, handling each query separately, without resource sharing, may be very inefficient or even infeasible for large numbers of queries or high data rates. Examples of such applications include publish-subscribe systems, which allow a large number of users to independently monitor published information of interest using subscriptions. Another example is intrusion detection, where a large number of rules are used to continuously monitor system and network activity [72, 111]. In these applications, subscriptions and rules are continuous queries.

A concrete example of an application that uses aggregations over sliding windows is Traderbot [109], an online publish-subscribe system that allows users to monitor realtime stock data using continuous queries. The following two queries were taken directly from the Traderbot site.

1. **NASDAQ Short-term Downward Momentum**: Find all NASDAQ stocks between $20 and $200 that have moved down more than 2% in the last 20 minutes and there has been significant buying pressure (70% or more of the volume has traded toward the ask price) in the last 2 minutes.
2. **High Volatility with Recent Volume Surge**: Find all stocks between $20 and $200 where the spread between the high tick and the low tick over the past 30 minutes is greater than 3% of the last price and in the last 5 minutes the average volume has surged by more than 300%.

Both queries apply sliding-window aggregation over stock substreams. In the examples above, ASW queries do not occur as independent queries but rather as subqueries within larger, more complex queries. Query 1 uses a \texttt{SUM} aggregate over a 2-minute sliding window. Query 2 uses \texttt{MAX} (high-tick), \texttt{MIN} (low-tick), and \texttt{AVG} aggregates over two different windows ("last 5 minutes" and "last 30 minutes"). Various relational operators like filters and joins are performed over the relations output by the ASW subqueries.

**Chapter Organization**

Section 5.2 presents formal definitions and notation. Section 5.3 highlights some fundamental tradeoffs faced by any solution to our problem. Sections 5.4 and 5.5 present our resource sharing algorithms for ASW and SSF queries, respectively. Section 5.7 contains an experimental evaluation of our algorithms. Section 5.8 covers related work, and Section 5.9 concludes.

**5.2 Preliminaries**

**Streams and Sliding Windows**

Recall from Section 2.4 of Chapter 2 that a stream $S$ is a continuous sequence of elements $\langle s_1, \tau_1 \rangle, \langle s_2, \tau_2 \rangle, \ldots$, where $s_i$ denotes a tuple and $\tau_i$ its timestamp. At any given point in time, we refer to the number of tuples of $S$ that have arrived so far as the current length of $S$.

For simplicity, we assume that timestamps are system-generated. This assumption means that elements of a stream are available for processing in increasing timestamp order, and that at any given time $T$, all elements of the stream with timestamps less than $T$ have arrived. When timestamps are not system-generated, we need to use special techniques for time-related issues, as discussed in Section 2.8 of Chapter 2.

\footnote{The term “high-tick” is parenthetical here. In particular, we are not applying \texttt{MAX} over attribute “high-tick”.}
In this chapter, we generalize sliding windows to include historical windows, i.e., windows that do not necessarily end with the most recent element of the stream. Consider a stream \( S \) with the sequence of elements \( \langle s_1, \tau_1 \rangle, \langle s_2, \tau_2 \rangle, \ldots \). A historical tuple-based window over stream \( S \) is of the form \( S[N_L, N_R] \); formally, when the current length of \( S \) is \( r \), \( S[N_L, N_R] \) contains the set of tuples \( \{s_i \mid \max\{r - N_L + 1, 1\} \leq i \leq (r - N_R)\} \). We observe that \( S[N_L, 0] \) is equivalent to \( S[N_L] \), and, more generally, \( S[N_L, N_R] = S[N_L] - S[N_R] \) in our algebra. Similarly, a historical time-based window is of the form \( S[T_L, T_R] \); when the current time is \( \tau \), it contains the set of tuples \( \{s_i \mid (\tau - T_L + 1) \leq \tau_i \leq (\tau - T_R)\} \). Again, we observe that \( S[T_R, 0] = S[T_R] \) and \( S[T_R, T_L] = S[T_R] - S[T_L] \) in our algebra. We denote a generic window over \( S \) that could be either time-based or tuple-based using \( S[X_L, X_R] \). We call our “regular” windows of the form \( S[X_L, 0] \) suffix windows.

**ASW and SSF Queries**

An ASW query over stream \( S \) with aggregation function \( f \) over attribute \( A \) is of the form \( G_f(A)(S[X_L, X_R]|X) \). At any given point in time, its current answer is obtained by applying the aggregation function \( f \) over the values of attribute \( A \) of all tuples in the current window. We do not explicitly consider ASW queries with grouping attributes, i.e., queries of the form \( G_f(A)(S[X_L, X_R]|X) \). In order to handle grouping, we can simply use one instance of our non-grouping ASW algorithms for each group in the input stream. Recently, Zhang et al. [116] have proposed more sophisticated techniques for ASW style queries with grouping.

It is slightly messy to represent SSF queries in our algebra, so we introduce more convenient notation for their representation. Recall from Section 2.6.1 that substreams of a stream are defined using a set of key attributes. For simplicity, we assume throughout this chapter that there is a single key attribute for all the streams that we consider; generalizing is straightforward. Let \( S \) denote a stream, \( K \) its key attribute, and \( \mathcal{K} \) the domain of \( K \). We denote the substream of \( S \) corresponding to \( k \in \mathcal{K} \) as \( S_k \). All tuples of \( S \) with a value \( k \) for attribute \( K \) belong to \( S_k \), and their relative ordering within \( S_k \) is the same as their relative ordering in \( S \). As SSF query over \( S \) is denoted by:

\[ \{k \in \mathcal{K} \mid G_f(A)(S_k[X_L, X_R]|X) \in (v_1, v_2)\} \]

At any given point in time, its answer is the set of substream keys \( k \) for which the ASW
subquery $G_{f(A)}(S_k[X_L, X_R])$ produces an answer that lies in the range $(v_1, v_2)$.

**Example 5.4** The SSF query \( \{ k \in K \mid G_{\text{SUM}(A)}(S_k[300, 0]_{T}) \in (1000, \infty) \} \) continuously computes all substreams for which the sum over attribute $A$ values in the last 300 time units is greater than 1000. In CQL this query can be expressed as:

```sql
Select K
From S [Range 300]
Group By K
Having Sum (A) > 1000
```

### Output Model

For the approach in this chapter, we assume that an ASW or SSF query does not actively stream its current answer, but instead produces answers only when requested. We call such a request an *answer lookup* (or simply *lookup*). We chose the lookup model over the streaming model for several reasons:

1. The lookup model is more general, i.e., we can simulate the streaming model using the lookup model, but not the other way.\(^2\)

2. As we indicated in Section 5.1 (under “Motivation”) ASW and SSF might occur as subqueries within a larger query. In such cases, we can sometimes produce more efficient plans under the lookup model than under the streaming model. For example, consider the query:

$$\mathcal{RS}(S1[Now] \bowtie G_{f(A)}(S2[T_L, T_R]))$$

For this query, the current answer for $G_{f(A)}(S2[T_L, T_R])$ is needed only when a tuple arrives on $S1$: If $S1$ is a low frequency stream, a plan under the lookup model can be more efficient than a plan under the streaming model, since in the latter case, we will be doing a lot of wasteful work streaming answers that are never used.

3. Reference [30] cites several applications that prefer to periodically refresh continuous query answers, rather than keep them fully up-to-date. For these applications, the lookup model is a better fit.

\(^2\)The simulation might possibly require information about arrival of stream elements, e.g., to evaluate the query $G_{f(A)}(S[1])$. 
Aggregation Functions

We use the classification suggested by Gray et al. [57] that divides aggregation functions into three categories: distributive, algebraic, and holistic. Let $X$, $X_1$, and $X_2$ be arbitrary bags of elements drawn from a numeric domain. An aggregation function $f$ is distributive if $f(X_1 \cup X_2)$ can be computed from $f(X_1)$ and $f(X_2)$ for all $X_1$, $X_2$. An aggregation function $f$ is algebraic if it is not distributive, but there exists a “synopsis function” $g$ such that for all $X$, $X_1$, $X_2$: (1) $f(X)$ can be computed from $g(X)$; (2) $g(X)$ can be stored in constant memory; and (3) $g(X_1 \cup X_2)$ can be computed from $g(X_1)$ and $g(X_2)$. An aggregation function is holistic if it is not algebraic. Among the standard aggregates, \textsc{sum}, \textsc{count}, \textsc{max}, and \textsc{min} are distributive, \textsc{avg} is algebraic, since it can be computed from a synopsis containing \textsc{sum} and \textsc{count}, and \textsc{quantile} is holistic.

5.3 Space-Update-Lookup Tradeoff and Our Approach

Any algorithm for processing a large number of continuous queries with the lookup model involves three cost parameters: the memory required to maintain state (space), the time to compute an answer (lookup time), and the time to update state when a new stream tuple arrives (update time). There is a tradeoff among these three costs and generally no single, optimal solution.

For example, we can make lookups efficient by maintaining up-to-date answers for all queries. However, this approach has a high update cost, since arrival of a new stream tuple potentially requires updating answers of all the queries, and a large space requirement, since current answers for all the queries need to be stored. Alternatively, we can maintain a single historical snapshot of the input that is large enough to compute the answer for any given query, but defer actual answer computation to lookup time. This approach has small update and space costs, but potentially high lookup cost.

These two approaches are appropriate only for the extreme scenarios—very high update rates compared to lookups, or vice-versa. Many applications lie between the extremes. Our algorithms are designed to capture a wide range of these “in between” scenarios, by performing partial answer computation at update time and using the partial results to compute the final answer at lookup time. Furthermore, our partial answer schemes are designed specifically so that partial answers can be shared by a large number of queries.
5.4 ASW Queries

In this section, we present our algorithms for collectively processing a set of ASW queries. Sharing resources is not possible between queries over different streams, or between queries with different aggregated attributes, since the aggregated data is completely different: There is no benefit to processing them collectively. Sharing is sometimes possible between queries with different aggregation functions (e.g., \textit{AVG} and \textit{SUM}) over the same input stream and aggregated attribute. However, we do not address this special case.

Therefore, our algorithms are designed for collectively processing a set of ASW queries with the same input stream, aggregation function, and aggregated attribute. The only difference between different queries is the sliding window specification. One exception is the \textit{QUANTILE} aggregation function, where we allow the quantile parameter to be different; see Section 5.4.3.

\textbf{Notation:} For the rest of this section, let \(q_1, \ldots, q_n\) denote the set of queries we wish to execute. Let \(S\) denote the common input stream, \(f\) the common aggregation function, and \(A\) the common aggregated attribute. For these queries, only the sequence of attribute \(A\) values are relevant, which we denote \(a_1, a_2, \ldots\). We call each \(a_i\) a stream \textit{element}.

For simplicity, we initially (Sections 5.4.1–5.4.3) assume that the windows of all the queries \(q_i\) are tuple-based. In Section 5.4.4 we extend our algorithms to handle time-based windows. For Sections 5.4.1–5.4.3, we assume \(q_i = G_{f(A)}(S[N_{L_i}, N_{R_i}])\).

\textbf{Intervals:} The notion of an \textit{interval} over positions of \(S\) is useful to describe our algorithms. The interval \(I = (l, r)\) \((l \leq r)\) denotes the positions \(l, l+1, \ldots, r\) of \(S\), and the elements \(a_l, a_{l+1}, \ldots, a_r\) belong to interval \(I\). For an interval \(I\), \(f(I)\) denotes the aggregation over the elements of \(S\) belonging to \(I\).

For each algorithm, we specify: (1) the state that it maintains; (2) an operation \texttt{UPDATE}(\(a_m\)) that describes how the state is updated when element \(a_m\) arrives; and (3) an operator \texttt{LOOKUP}(\(I\)) that describes, for certain intervals \(I\), how \(f(I)\) can be computed using the current state. \texttt{LOOKUP}(\(I\)), as the name suggests, is used to perform answer-lookups for queries \(q_i\): when the current size of \(S\) is \(m\), the current answer for \(q_i\) can be obtained using \texttt{LOOKUP}(\(I\)) where \(I = (m - N_{L_i} + 1, m - N_{R_i})\). Therefore, for correctness, when the current length of \(S\) is \(m\), we require that \texttt{LOOKUP}(\(I\)) correctly compute \(f(I)\) for all intervals \(I = (m - N_{L_i} + 1, m - N_{R_i})\) \((1 \leq i \leq n)\); it may or may not compute \(f(I)\) correctly for
other intervals.

### 5.4.1 Distributive and Algebraic Aggregates

In this section, we present two algorithms, B-INT and L-INT, for the case where \( f \) is distributive or algebraic. For presentation clarity, we assume that \( f \) is distributive; the generalization to the algebraic case is straightforward.

Both algorithms are based on a simple but fairly general approach: For certain intervals \( I \), precompute \( f(I) \) and store it as part of the state. The basic intuition behind this step is that, since \( f \) is distributive, the precomputed aggregate values can be used to compute lookups more efficiently. For example, \( f(101, 200) \) and \( f(201, 300) \) can be used to compute \( f(101, 300) \), and therefore, \( \text{LOOKUP}(101, 300) \). More generally, \( f(I) \) can potentially help compute \( \text{LOOKUP}(I') \) for any \( I' \) that contains \( I \).

For what intervals \( I \) should we precompute \( f(I) \)? Selecting more intervals for precomputation is likely to improve lookup efficiency, but at the cost of space and update time—a manifestation of the space-lookup-update tradeoff discussed in Section 5.3. Also, any precomputed aggregate \( f(I) \) loses its utility eventually, once all the windows of \( q_i \) slide past \( I \). (In fact, the answer to this question of which intervals to precompute is not very obvious even for processing a single query, i.e., \( n = 1 \).)

Next we present our two algorithms, which are essentially two different schemes for dynamically selecting the intervals \( I \) to precompute, along with the details of how \( f(I) \) aggregates for selected intervals are (pre)computed and used for lookups.

**B-INT Algorithm**

Our first algorithm is called B-INT (for Base-Intervals), since it precomputes aggregate values \( f(I_b) \) for intervals \( I_b \) that belong to a special class called base-intervals. Intuitively, base-intervals form a “basis” for intervals: any interval can be expressed as a disjoint union of a small number of base-intervals. Using this property, any \( f(I) \) can be computed using a small number of precomputed \( f(I_b) \) values. At any point of time, B-INT stores \( f(I_b) \) values for only recent or “active” base-intervals—only these are potentially useful for future lookups of the queries \( q_1, \ldots, q_n \).

Figure 5.1 abstractly illustrates the state maintained by algorithm B-INT, when the current length of \( S \) is \( m \). The active base-intervals, for which B-INT precomputes aggregate
values, are shown as solid rectangles. The base-intervals which are not active are shown using dotted rectangles. The figure also shows how the aggregate value for a lookup interval is computed using precomputed aggregates for active base-intervals.

**Definition 5.1 (Base-Interval)** An interval $I_b$ is a base-interval if it is of the form $(i \cdot 2^\ell + 1, (i + 1) \cdot 2^\ell)$ for some integer $i \geq 0$, in which case it is called a level-$\ell$ base-interval.

For example, $(385, 512) = (3 \cdot 2^7 + 1, 4 \cdot 2^7)$ is a level-7 base-interval. A level-$\ell$ base-interval has a width $2^\ell$ and is a disjoint union of exactly two level-$(\ell - 1)$ base-intervals. Base-intervals are closely related to the intervals that we used in Chapter 4 for computing approximate quantiles and frequency counts over sliding windows. The only difference between the two is the sizes of intervals at various levels. A level-$\ell$ base interval has size $2^\ell$, while a level-$\ell$ interval in Chapter 4 had size $2^\ell \cdot \frac{W}{4}$, for a max-window parameter $W$ and error parameter $\epsilon$. A similar class of intervals, called dyadic intervals, is used by Gilbert et al. [54] over an ordered domain for computing approximate quantiles over relations.

The following theorem, whose proof is straightforward, formally states that any interval can be expressed as a union of a small number of base-intervals.

**Theorem 5.1** Any interval $I = (l, r)$ of width $W = (r - l + 1)$ can be expressed as a disjoint-union of $k = O(\log W)$ base-intervals of the form $I_{bi} = (l_i, r_i)$ ($1 \leq i \leq k$), where $l_1 = l$, $r_k = r$, and $r_i = l_{i+1} - 1$, $(1 \leq i < k)$. Given interval $I$, the intervals $I_{b1}, \ldots, I_{bk}$ can be determined in $O(k) = O(\log W)$ time.

**Example 5.5** The interval $(1, 43)$ can be expressed as a union of base-intervals $(1, 2)$, $(3, 4)$, $(5, 8)$, $(9, 16)$, $(17, 32)$, $(33, 40)$, $(41, 42)$, $(43, 43)$. We can use a simple greedy algorithm to
chapter 5. Resource Sharing

State:
1. For each currently active base-interval $I_b$, store $f(I_b)$.

Update $(a_m)$:
1. Compute $f(a_m, a_m)$ and store it.
2. Let $z$ denote the number of trailing 0’s in binary representation of $m$.
3. For $i = 1$ to $z$
   /∗ assert: $(m - 2^i + 1, m)$ is a base-interval /∗
4. If $(l, r) = (m - 2^i + 1, m)$ is active
5. Compute $f(l, r)$ using $f(l, l + r - 1)$ and $f(l + r + 1, r)$ and store it.
6. For each $I_b$ that ceases to be active, discard $f(I_b)$.

Lookup$(I)$:
1. Express $I$ as a union of base-intervals $I_{b1}, \ldots, I_{bk}$ as in Theorem 5.1.
2. Compute $f(I)$ using $f(I_{b1}), \ldots, f(I_{bk})$ and return the same.

Figure 5.2: Algorithm B-INT

Identify these base-intervals: Pick the largest base-interval contained within $(1, 43)$, which happens to be $(17, 32)$. Next, pick the largest base-interval contained within $(1, 43)$ that does not overlap with $(17, 32)$, which is $(9, 16)$. Next, pick the largest base-interval that overlaps with neither $(17, 32)$ nor $(9, 16)$, and so on.

Active Intervals: Let $N_{\text{max}} = \max_i(N_{Li})$ denote the “earliest” left end of a window in $q_1, \ldots, q_n$. When the current size of $S$ is $m$, we call an interval $I = (l, r)$ active if $(l > m - N_{\text{max}})$ and $(r \leq m)$. Intuitively, an interval is active at some point of time, if it is completely within the last $N_{\text{max}}$ positions of the stream.

Figure 5.2 contains the formal description of B-INT. When the current size of $S$ is $m$, Lookup$(I)$ computes $f(I)$ for all intervals $I = (m - N_{Li} + 1, m - N_{Ri})$ ($1 \leq i \leq n$) that correspond to lookups of queries $q_1, \ldots, q_n$. By definition, any such interval $(m - N_{Li} + 1, m - N_{Ri})$ is active, and therefore each interval $I_{b1}, \ldots, I_{bk}$ in Step 1 of Lookup$(I)$ is active as well, implying that $f(I_{bi})$ is stored as part of the state.

Conceptually, Update$(a_m)$ computes $f(I_b)$ for all base-intervals $I_b$ that become newly active and adds it to the current state (Steps 1–5), and discards $f(I_b)$ for all intervals that cease to be active (Step 6). Update$(a_m)$ always introduces at least one new base-interval: $(a_m, a_m)$. In general, if $2^z$ denotes the largest power of 2 which divides $m$, then...
UPDATE\((a_m)\) introduces \(z + 1\) new base-intervals. One obvious technique for computing \(f(I_b)\) is to do so from scratch, using the elements of \(S\) that belong to \(I_b\). A more efficient technique is to compute it recursively, using aggregate values corresponding to base-intervals of the next lower-level, as shown in Step 5 of UPDATE\((a_m)\).

**Theorem 5.2** Algorithm B-INT requires \(O(N_{\text{max}})\) space, has an amortized update time complexity of \(O(1)\), and has a worst-case lookup time complexity of \(O(\log W)\), where \(W\) denotes the width of the lookup interval.

**Proof:** Since the size of a level-\(\ell\) base-interval is \(2^\ell\), at any point of time there are at most \(\lfloor N_{\text{max}} \rfloor / 2^\ell\) active, level-\(\ell\) base-intervals. The total number of active base-intervals is therefore less than \(N_{\text{max}} + \lfloor N_{\text{max}} / 2 \rfloor + \lfloor N_{\text{max}} / 4 \rfloor + \ldots = O(N_{\text{max}})\). For updates, note that computing each \(f(I_b)\) requires constant time (Step 1 or Step 5). Any sequence of \(u\) updates introduces \(O(u)\) base-intervals, implying that the amortized cost of updates is \(O(1)\). The lookup time complexity follows directly from Theorem 5.1.

\[\Box\]

**L-INT Algorithm**

Our second algorithm for distributive and algebraic aggregates, called L-INT (for *Landmark Intervals*), uses an interval scheme based on certain landmarks or specific positions of the stream. L-INT is more efficient than B-INT for lookups, but its update and space costs are higher. Further, L-INT is more input-specific: while B-INT depends only on \(N_{\text{max}}\), L-INT, in addition to \(N_{\text{max}}\), depends on the distribution of window widths.

We first present L-INT for a special case, where all the window widths are close to equal. Specifically, we assume that they are within factor 2 of each other, i.e., \(W_{\text{max}} / W_{\text{min}} \leq 2\), where \(W_{\text{max}} = \max_i (N_{Li} - N_{Ri})\) denotes the maximum width, and \(W_{\text{min}} = \min_i (N_{Li} - N_{Ri})\), the minimum width of a window. For this special case, L-INT is optimal: it uses \(O(N_{\text{max}})\) space and has \(O(1)\) update and lookup time. Then, we show how we can extend L-INT to handle the general case.

**Definition 5.2 (Landmark Interval)** Landmark intervals are defined for two width parameters \(W_{\text{min}} \leq W_{\text{max}}\). A landmark interval is of the form \((\alpha W_{\text{min}}, \alpha W_{\text{min}} + d)\) or of the form \((\alpha W_{\text{min}} - d, \alpha W_{\text{min}} - 1)\), for some \(\alpha \geq 0\) and \(d \leq W_{\text{max}}\).

We call stream positions of the form \(\alpha W_{\text{min}} \ (\alpha \geq 0)\) landmarks. A landmark interval is one that begins at or ends just before a landmark, and has a width less than \(W_{\text{max}}\). For example,
Theorem 5.3  Any interval $I = (l, r)$, such that $W_{\min} \leq (r - l + 1) \leq W_{\max}$, can be expressed as a disjoint union of at most two landmark intervals defined for parameters $W_{\min}$ and $W_{\max}$.

Proof: Let $\alpha_m = \min_{\alpha}\{\alpha W_{\min} \geq l\}$. If $\alpha_m W_{\min} = l$, $(l, r) = (\alpha_m W_{\min}, r)$ is itself a landmark interval. If $\alpha_m W_{\min} > l$, then $(l, r)$ is the union of the two landmark intervals $(l, \alpha_m W_{\min} - 1)$ and $(\alpha_m W_{\min}, r)$. □

Example 5.6  Let $W_{\max} = 2000$ and $W_{\min} = 1000$. The interval $(3257, 5164)$ can be expressed as a union of landmark intervals $(3257, 3999)$ and $(4000, 5164)$. □

Figure 5.4 contains the formal description of L-INT for the special case of $W_{\max}/W_{\min} \leq 2$. Using a reasoning similar to that for algorithm B-INT, we can argue that algorithm L-INT is correct, i.e., it can be used to compute lookups corresponding to the queries $q_1, \ldots, q_n$. The update operation is also similar to that of B-INT: it computes $f(I_i)$ for all landmark intervals $I_i$ that newly become active, and discards $f(I_i)$ for intervals $I_i$ that cease to be active.
STATE:
1. For each currently active landmark interval \( I_l \), store \( f(I_l) \).
2. For each currently active element \( a_i \), store \( f(a_i) \).

UPDATE \((a_m)\):
1. Compute \( f(a_m) \) and store it.
2. If \( m = \alpha W_{\text{min}} \) for some \( \alpha \geq 0 \)
3. For \( d = 2 \) to \( W_{\text{max}} \)
   4. Compute \( f(m-d, m-1) \) using \( f(m+1-d, m-1) \) and \( f(a_{m-d}) \) and store it.
5. For each \( \beta \) such that \( m - W_{\text{max}} \leq \beta W_{\text{min}} \leq m - 1 \)
   6. Compute \( f(\beta W_{\text{min}}, m) \) from \( f(\beta W_{\text{min}}, m-1) \) and \( f(a_m) \) and store it.
7. For each \( I_l \) that ceases to be active, discard \( f(I_l) \).

LOOKUP\((I)\):
1. Express \( I \) as a union of landmark intervals \( I_{l1} \) and \( I_{l2} \) as in Theorem 5.3
2. Compute \( f(I) \) using \( f(I_{l1}) \) and \( f(I_{l2}) \) and return the same.

Figure 5.4: Algorithm L-INT for \( W_{\text{max}}/W_{\text{min}} \leq 2 \)

**Theorem 5.4** Algorithm L-INT presented in Figure 5.4 requires \( O(N_{\text{max}}) \) space, has an amortized update time of \( O(1) \), and has a worst case lookup time of \( O(1) \).

**Proof:** Each landmark has \( 2W_{\text{max}} \) landmark intervals associated with it (Figure 5.3). Since landmark intervals are spaced \( W_{\text{min}} \) apart, at any given time, there are at most \( (N_{\text{max}}/W_{\text{min}}) \) landmarks that are active (i.e., among the last \( N_{\text{max}} \) positions). Therefore, the number of active landmark intervals is always less than \( 2W_{\text{max}}(N_{\text{max}}/W_{\text{min}}) \leq 4N_{\text{max}} \), which is \( O(N_{\text{max}}) \). Computing each \( f(I_l) \) requires constant time (at one of steps 1, 4, or 6), and any sequence of \( u \) updates introduces \( O(u) \) new landmark intervals, which implies that the amortized cost of updates is \( O(1) \) (worst case is \( O(W_{\text{max}}) \)). The lookup time complexity follows directly from Theorem 5.3. \( \square \)

Extending L-INT algorithm for the general case is straightforward: partition the set of queries \( q_1, \ldots, q_n \) into \( \gamma \) partitions \( P_1, P_2, \ldots, P_\gamma \), such that for the queries belonging to each partition, the property \( W_{\text{max}}/W_{\text{min}} \leq 2 \) is satisfied. Use \( \gamma \) instances of the special-case version of the L-INT algorithm (Figure 5.4) to process these partitions independently. This extended algorithm requires \( O(\gamma N_{\text{max}}) \) space, has an update cost of \( O(\gamma) \), and a lookup cost of \( O(1) \).

For any set of queries \( q_1, \ldots, q_n \), we can always define a partitioning scheme with \( \gamma = \)
$O(\log N_{\text{max}})$. However, for many real-world applications, it seems natural to expect the window widths to be clustered around a few values. For such applications, $\gamma$ could be significantly smaller than $\log N_{\text{max}}$.

Further, if all the queries $q_1, \ldots, q_n$ have suffix windows, or even “approximately” suffix windows, we can reduce the space required from $O(\gamma N_{\text{max}})$ to $O(N_{\text{max}})$. A tuple-based window $[N_L, N_R]$ is approximately suffix if $(N_L - N_R)$ is comparable in value to $N_L$.

### 5.4.2 Subtractable Aggregates

An algebraic aggregation function $f$ is **subtractable** if its synopsis function $g$ has the following property: for any bags $X_1 \subseteq X_2$, $g(X_2 - X_1)$ is computable from $g(X_1)$ and $g(X_2)$. Among the standard algebraic aggregation functions SUM, COUNT, and AVG are subtractable, while MAX and MIN are not. For instance, SUM is subtractable since $\text{SUM}(X_2 - X_1) = \text{SUM}(X_2) - \text{SUM}(X_1)$, if $X_1 \subseteq X_2$.

For subtractable aggregates, we present a simple algorithm called R-INT (for **running intervals**) that has $O(1)$ update and lookup cost. For presentation clarity, we assume that $f$ is subtractable and distributive, i.e., $f(X_2 - X_1)$ is computable from $f(X_2)$ and $f(X_1)$, whenever $X_1 \subseteq X_2$. Generalizing to the case where $f$ is algebraic and subtractable is straightforward.

A **running interval** is an interval of the form $(1, r)$, whose left end is at the beginning of $S$. Any interval $(l, r)$ can be expressed as a difference of two running intervals: $(1, r) - (1, l - 1)$. R-INT (Figure 5.5) is based on this observation. R-INT stores aggregate values corresponding to currently active running intervals, and uses these to compute lookup answers. When the current length of $S$ is $m$, a running interval $(1, r)$ is **active**, if $m - N_{\text{max}} \leq r \leq m$. It can be easily shown that R-INT uses $O(N_{\text{max}})$ space and has $O(1)$ update and lookup time.

### 5.4.3 Quantiles

Recall from Chapter 4 that the quantile aggregation function is specified using a parameter $\phi \in (0, 1]$ and is denoted $\text{QUANTILE}(\phi)$. The output of $\text{QUANTILE}(\phi)$ for a bag of $N$ elements is the element at position $\lceil \phi \cdot N \rceil$ in a sorted sequence of these elements.

We briefly sketch one algorithm (called B-INT-QNT) for processing ASW queries with quantiles (possibly with different parameters $\phi$) that uses the base-intervals defined in
Chapter 5. Resource Sharing

State:
1. For each currently active running interval $I$, store $f(I)$.

Update $(a_m)$:
1. Compute $f(1, m)$ using $f(1, m - 1)$ and $f(a_m)$.
2. For each $I$ that is no longer active, discard $f(I)$.

Lookup $(I = (l, r))$:
1. Compute and return $f(l, r)$ using $f(1, l - 1)$ and $f(1, r)$.

Figure 5.5: Algorithm R-INT

Section 5.4.1: Corresponding to each active base-interval $I_b$ store a sorted array of the elements of $S$ that belong to $I_b$. To perform Lookup($I$) for a quantile parameter $\phi$, express $I$ as a union of base-intervals $I_{b1}, \ldots, I_{bk}$ using Theorem 5.1, and compute QUANTILE($\phi$) using the sorted arrays corresponding to $I_{b1}, \ldots, I_{bk}$. In general, given a set of $p$ sorted arrays of length $\leq q$, we can compute any quantile over all the elements in the sorted arrays in $O(p \log p \log q)$ time, using a greedy algorithm.

5.4.4 Extensions for Time-Based Windows

As in the case of tuple-based windows, a lookup for an ASW query with a time-based window corresponds to some interval of stream positions. For example, if 121 tuples of $S$ arrived in the first 100 time units, and 140 in the next 100, a lookup at time 200 units involving a time-window $[100, 0]$T corresponds to the stream position interval $(122, 261)$.

However, for tuple-based windows the position interval corresponding to a lookup depends only on the length of $S$ at the time of the lookup, while for time-based windows the interval depends on the exact timestamps of tuples.

Let $t_{\max}$ denote the maximum left-end of a time-based window over all queries. All of our algorithms for tuple-based windows can be extended to handle queries with time-based windows using the following two modifications: (1) The algorithm stores information about element timestamps in order to compute position intervals corresponding to time-based window lookups. Interval computation can be done in $O(1)$ time using $O(t_{\max})$ space, or in $O(\log N)$ time using $O(N)$ space, where $N$ is the current number of active elements. At any point in time $\tau$, $N$ is defined as the number of elements with timestamps between $\tau$ and $\tau - t_{\max}$. (2) The algorithm defines active intervals using $t_{\max}$ instead of a
fixed $N_{\text{max}}$.

5.5 SSF Queries

We now consider the problem of processing a collection of SSF queries $q_1, \ldots, q_n$. As in Section 5.4, we assume that all queries have the same input stream $S$, the same aggregation function $f$, and the same aggregated attribute $A$. So they differ only in their window specification and range predicate. Therefore each query $q_i$ has the form \( \{ k \in K \mid \mathcal{G}_{f(A)}(S_k[X_{L_i}, X_{R_i}] \in (v_{l_i}, v_{h_i})) \} \).

A simple strategy for processing these queries is as follows: For each substream $S_k$, process the set of ASW subqueries $\mathcal{G}_{f(A)}(S_k[X_{L_i}, X_{R_i}]X) (1 \leq i \leq n)$ using the algorithms of Section 5.4. To perform a lookup for query $q_i$, perform a lookup on the ASW subquery $\mathcal{G}_{f(A)}(S_k[X_{L_i}, X_{R_i}]X)$ for each substream $S_k$, and return those substreams for which the lookup output lies within $(v_{l_i}, v_{h_i})$. Clearly, the SSF lookup cost for this approach depends linearly on $|K|$, the number of substreams.

In this section, we present algorithms for certain combinations of aggregation functions ($\text{COUNT}$, $\text{SUM}$ over positive values, $\text{MAX}$, $\text{MIN}$) and window types (suffix, time-based) that have lookup cost sublinear in $|K|$. We call our algorithms CI-COUNT, CI-SUM, CI-MAX, and CI-MIN. CI stands for collective index since, conceptually, the algorithms can be thought of as a collection of search indexes, one for each ASW subquery.

Notation: Throughout this section, let $a_{k1}, a_{k2}, \ldots$ denote the sequence of attribute $A$ values for substream $S_k$. As before, we call them elements of $S_k$. Let $\tau_{k1}, \tau_{k2}, \ldots$ denote the timestamps of these elements. As in Section 5.4, for each algorithm we present: (1) the state that it maintains; (2) an operation UPDATE($a_{km}, \tau_{km}$) that describes how the state is modified when element $a_{km}$ with timestamp $\tau_{km}$ arrives on substream $S_k$; and (3) an operation LOOKUP($\tau, T, (v_1, v_2)$) that describes how the current answer for the SSF query \( \{ k \in K \mid \mathcal{G}_{f(A)}(S_k[T, O]) \in (v_1, v_2) \} \) can be computed using the current state.

5.5.1 CI-COUNT

CI-COUNT is an approximate algorithm for processing a collection of SSW queries of the form $q_i = \{ k \in K \mid \mathcal{G}_{\text{COUNT}(A)}(S_k[T, O]) \in R \}$, where $R$ is a one-sided range condition of the form $(v, \infty)$ or $(0, v)$. The approximation produced by CI-COUNT for LOOKUP($\tau, T, R$) is
as follows. If \( K_{\text{co}} \) denotes the correct output, the output \( K_{\text{ao}} \) produced by CI-COUNT has the following guarantees:

1. The approximate output \( K_{\text{ao}} \) is a superset of the exact output \( K_{\text{co}} \).

2. The current ASW answer for each substream in the approximate output satisfies a relaxed version of the range condition \( R \). Specifically, if \( R \) is of the form \((v, \infty)\), for every key \( k \in K_{\text{ao}} \), \( G_{\text{COUNT}}(A) (S_k[T, 0]) \in (v(1 - \epsilon), \infty) \), for some approximation parameter \( \epsilon \in (0, 1) \). Similarly, if \( R \) is of the form \((0, v)\), for every key \( k \in K_{\text{ao}} \), \( G_{\text{COUNT}}(A) (S_k[T, 0]) \in (0, v(1 + \epsilon)) \).

The approximation parameter \( \epsilon \) in the above guarantees can be made as small as desired, but decreasing \( \epsilon \) increases the required space and update cost: both grow linearly in \( \frac{1}{\epsilon} \).

Although CI-COUNT supports only approximate lookups, it can be used along with our ASW algorithms for performing exact lookups. Specifically, we can compute the correct answer \( K_{\text{co}} \) from the approximate output \( K_{\text{ao}} \) by checking for each \( k \in K_{\text{ao}} \) whether \( G_{\text{COUNT}}(A) (S_k[T, 0]) \in R \) using an ASW-lookup.

We first present a non-parameterized version of CI-COUNT that yields a fixed \( \epsilon = 0.75 \), and then describe how this algorithm is generalized to produce the parameterized version. Also, for clarity, we assume that the range conditions of all the queries are of the form \((v, \infty)\). Handling range conditions of the form \((0, v)\) is a straightforward modification. In the rest of this subsection, we abbreviate \( \text{LOOKUP}(\tau, T, (v, \infty)) \) as \( \text{LOOKUP}(\tau, T, v) \).

**Non-Parameterized CI-COUNT**

To get an intuition for CI-COUNT, consider \( \text{LOOKUP}(\tau, T, v) \). This operation identifies all substreams \( S_k \) that have received more than \( v \) elements in the last \( T \) time units. An alternate but equivalent view of this operation is that it identifies all substreams \( S_k \) for which the \( v^{th} \) element from the end has a timestamp greater than \( \tau - T \). Based on this observation, one idea for improving the efficiency of lookup is as follows: Maintain an index over the timestamp values of the \( v^{th} \) element from the end of all the substreams. Use this index to determine, in \( O(\log |K|) \) time, all the substreams for which the timestamp of the \( v^{th} \) element from the end is greater than \( \tau - T \).

Since \( v \) is a parameter in \( \text{LOOKUP}(\tau, T, v) \), we would need to maintain such an index for every possible \( v \) in order to use the idea above. However, doing so would dramatically
increase the update cost: Every new element \( a_{k_i} \) of substream \( S_k \) changes the timestamp of the \( v^{th} \) element from the end for every \( v \), and so requires updating all the indexes.

However, if we are permitted to approximate \( v \), i.e., use a different value \( v' \) that is close to \( v \), we can reduce the number of different indexes that need to be maintained, since many values \( v \) can use the same approximation \( v' \).

This observation forms the basis for algorithm CI-COUNT: CI-COUNT divides the positions from the end of a substream \( S_k \) into different levels (not to be confused with levels of base-intervals in Section 5.4.1). It maintains one search index for each level. The index for level-\( \ell \) contains, for each substream \( S_k \), the timestamp of some element that currently belongs to level-\( \ell \). These indexed timestamps are used for approximate answer lookups.

**Definition 5.3 (Level)** Let \( m \) be the current length of substream \( S_k \). Then the current level of a position \( p \leq m \) of \( S_k \) is defined to be \( \lfloor \log_2 (m - p + 1) \rfloor \).

The last position, \( m \), of substream \( S_k \) belongs to level-0, the previous two \( (m - 1 \) and \( m - 2 \)) to level-1, and so on. In general, \( 2^\ell \) positions belong to level-\( \ell \).

Figure 5.6 contains the formal description of algorithm CI-COUNT. The variable \( TSTAMP[k, \ell] \) contains the timestamp of the element of \( S_k \) that currently belongs to level-\( \ell \) and whose position is a multiple of \( 2^\ell \). Note that, at any point of time, such an element (if it exists) is unique, since at most \( 2^\ell \) contiguous positions belong to level-\( \ell \). For each level \( \ell \), all the \( TSTAMP[k, \ell] \) values are indexed using a search tree \( SEARCHTREE[\ell] \). In order to perform \( LOOKUP(\tau, T, v) \), CI-COUNT uses \( SEARCHTREE[\ell] \) for \( \ell = \lfloor \log_2 v \rfloor - 1 \) to determine all substreams \( S_k \) such that \( TSTAMP[k, \ell] \geq (\tau - T) \).

**Example 5.7** Figure 5.7 shows the timestamps (within boxes) and positions (above boxes) of elements belonging to three substreams. The elements themselves are not shown. The timestamps that are stored in \( TSTAMP[k, \ell] \) are circled. For example, \( TSTAMP[k_3, 2] = 85 \). The search trees over \( TSTAMP[k, \ell] \) values are not shown.

Consider \( LOOKUP(106, 20, 11) \) which seeks at time 106 all substreams that have received more than 11 tuples in the last 20 time units, i.e., in the time interval \((87, 106)\). Clearly, the correct output is \( \{k_1\} \). The same output can also be obtained by checking if the timestamp of the 11th tuple from the end (which is 90, 85, and 68, for \( S_{k_1}, S_{k_2} \), and \( S_{k_3} \), respectively) is \( \geq 87 \).

Since \( \lfloor \log_2 11 \rfloor - 1 = 2 \), CI-COUNT returns those substream keys \( k \) for which \( TSTAMP[k, 2] \geq 87 \), which is \( \{k_1, k_2\} \) for this example. In other words, for each substream,
STATE:
1. For each substream $S_k$
2. 
3. Let $i$ be unique position of $S_k$ such that:
4. (a) $i$ currently belongs to level-$\ell$, and
5. (b) $i$ is a multiple of $2^\ell$
6. $\text{TSTAMP}[k, \ell] = \tau_i$
7. For each level-$\ell$
8. Store $\text{TSTAMP}[k, \ell]$, for all valid $k$, using a search tree $\text{SEARCHTREE} [\ell]$.

UPDATE($a_{km}, \tau_{km}$):
1. $p = 1$.
2. Let $z =$ number of trailing 0’s in binary representation of $m + p$.
3. For $\ell = z$ downto 1 do:
4. $\text{TSTAMP}[k, \ell] = \text{TSTAMP}[k, \ell - 1]$.
5. $\text{TSTAMP}[k, 0] = \tau_{km}$.

LOOKUP($\tau, T, v$):
1. Let $\ell = \lceil \log_2 v \rceil - 1$.
2. Determine $A = \{ k \in K \mid \text{TSTAMP}[k, \ell] \geq \tau - T \}$ using $\text{SEARCHTREE} [\ell]$.
3. Return $A$.

Figure 5.6: Algorithm CI-COUNT

CI-COUNT uses the timestamp of a position at a distance 4–7 from the end of the substream, instead of the timestamp of the position that is at a distance 11, and checks if it is greater than 87.

We now briefly comment on the update operation. Consider any one particular $\text{TSTAMP}[k, \ell]$ for $\ell > 1$. By definition, $\text{TSTAMP}[k, \ell]$ stores the timestamp of the element that currently belongs to level-$\ell$ and whose position is $i2^\ell$ for some $i$. Clearly, $\text{TSTAMP}[k, \ell]$ changes only when this element moves to level-$(\ell + 1)$, and the element corresponding to position $(i + 1)2^\ell$ enters level-$\ell$. Since $(i + 1)2^\ell$ is also a multiple of $2^{\ell-1}$, the timestamp of this element would previously have been stored in $\text{TSTAMP}[k, \ell - 1]$. Therefore, $\text{TSTAMP}[k, \ell]$ can be updated by just copying the previous value of $\text{TSTAMP}[k, \ell - 1]$ (Step 4 in UPDATE($a_{km}, \tau_{km}$)).

Lemma 5.1 Algorithm CI-COUNT presented in Figure 5.6 has approximation parameter $\epsilon = 0.75$: If $k \in K$ is returned in the output of LOOKUP($\tau, T, v$), then $G_{\text{COUNT}}(A)(S_k[T, 0]) \in (v/4, \infty)$
at time $\tau$. If at time $\tau$, $G_{\text{COUNT}(A)}(S_k[T,0]) \in (v, \infty)$, then $k$ is returned in the output of $	ext{LOOKUP}(\tau, T, v)$.

**Proof:** Let $\ell = [\log_2 v] - 1$. Let $m$ denote the length of $S_k$ at current time $\tau$. Let $i$ denote the position currently belonging to level-$\ell$ that is a multiple of $2^\ell$. By definition, $\text{TSTAMP}[k, \ell] = \tau_i k_i$.

Assume $k$ is returned in the output of $	ext{LOOKUP}(\tau, T, v)$. Then, $\tau_{ki} = \text{TSTAMP}[k, \ell] \geq (\tau - T)$. Since all the elements after position $i$ have a timestamp at least $\tau_{ki}$, it follows that $G_{\text{COUNT}(A)}(S_k[T,0]) \geq (m - i + 1)$. Since $i$ belongs to level-$\ell$, the minimum value of $(m - i + 1)$ is $2^\ell$. Therefore, $G_{\text{COUNT}(A)}(S_k[T,0]) \geq 2^\ell \geq 2^{[\log_2 v] - 1} \geq v/4$.

Now assume $G_{\text{COUNT}(A)}(S_k[T,0]) \in (v, \infty)$. Let $j$ denote the position $(m - v + 1)$ of $S_k$. Then it follows that $\tau_{kj} \geq (\tau - T)$. The level of position $j$ is $[\log_2 v]$, which is $\ell + 1$. Since position $i$ belongs to level-$\ell$, $i \geq j$. Therefore, $\tau_{kj} \leq \tau_{ki} = \text{TSTAMP}[k, \ell]$, which implies that $k$ is returned as part of the output. $\square$

**Theorem 5.5** Let $t_{\text{max}}$ denote maximum time interval of a sliding window that CI-COUNT supports, and let $N_{\text{max}}$ denote the current number of elements belonging to all substreams with timestamps in the last $t_{\text{max}}$ time-units. The CI-COUNT algorithm presented in Figure 5.6 requires $O(|K| \log N_{\text{max}})$ space, has an amortized update time complexity of $O(|K|)$, and a lookup time complexity of $O(|K_o|)$, where $|K_o|$, the number of substreams in the output of lookup. Further, the lookup has an approximation parameter $\epsilon = 0.75$. 

Figure 5.7: Timestamps (within boxes) and positions (above boxes) of elements belonging to three substreams.
Proof: The maximum value of $G_{\text{COUNT}}(A)(S_{k}[T, 0])$ for any substream is less than $N_{\text{max}}$. Therefore, it is sufficient to maintain $\text{TStamp}[k, \ell]$ values for $\log N_{\text{max}}$ levels: $\text{TStamp}[k, \ell]$ values for $\ell > \log N_{\text{max}}$ would be required only for lookups with $v > N_{\text{max}}$, and the output for such lookups is always the empty set. With this modification the space complexity of CI-COUNT is $O(|\mathcal{K}| \log N_{\text{max}})$, since CI-COUNT maintains $O(|\mathcal{K}|)$ $\text{TStamp}[k, \ell]$ values for each level $\ell$.

For the update cost, we can show that each update changes $\text{TStamp}[k, \ell]$ values for $O(1)$ levels in an amortized sense. Further each change of a $\text{TStamp}[k, \ell]$ requires updating the search tree for level-$\ell$, which requires $O(\log |\mathcal{K}|)$ time. Therefore, the overall update time complexity is $O(\log |\mathcal{K}|)$ amortized. Each lookup involves a range search using a search tree. All the range searches are one-sided (i.e., of the form $\geq (\tau - T)$). One-sided range searches can be performed in $O(|\mathcal{K}|)$ time, by maintaining pointers to the maximum element of search trees. The approximation $\epsilon = 0.75$ follows directly from Lemma 5.1. □

Parameterized CI-COUNT

The technique that we use to parameterize CI-COUNT is well-known and has been suggested before [39]. We only present the main ideas and the statement of the results. Consider a simple generalization of the non-parameterized CI-COUNT. The generalized version has $p$ levels of size 1, $p$ levels of size 2, and so on. As before, for each level whose size is $2^\ell$, CI-COUNT stores the timestamp of the element that belongs to the level, and whose position is a multiple of $2^\ell$. We can extend the lookup and update operations presented earlier to the general case in a straightforward manner. The update complexity is now $O(p \log |\mathcal{K}|)$, while the lookup complexity remains unchanged at $O(|K_{\text{ao}}|)$.

As we increase $p$, the relative error, $\epsilon$, of the generalized version reduces. For example, we can show that relative error for the case $p = 2$ is $\epsilon = 0.5$, instead of $\epsilon = 0.75$ for $p = 1$. In general, we can prove that as $p$ increases the relative error falls roughly as $2/p$. Therefore, by setting $p$ to be roughly $2/\epsilon$, we can achieve any desired relative error $\epsilon$.

5.5.2 CI-SUM

CI-SUM is derived from CI-COUNT in a straightforward manner: Replace the SUM aggregation functions in the input queries with COUNT aggregation functions, and process a modified stream $S'$ using algorithm CI-COUNT. Corresponding to every element $a_{ki}$ of $S_k$,
there are \( a_{ki} \) copies of the same element in \( S'_k \) with the same timestamp \( \tau_{ki} \). Any lookup involving the \text{SUM} aggregation function can be translated into an equivalent lookup involving \text{COUNT} aggregation function on the modified stream \( S'_k \), and therefore can be processed using CI-COUNT. The CI-SUM algorithm constructed in this manner has exactly the same approximation guarantees as that of the CI-COUNT instance used in the construction. The only problem with this approach is that naively performing an update for each of the \( a_{ki} \) copies of an element \( a_{ki} \) of the original stream \( S_k \) would result in an update operation whose time complexity grows linearly in \( a_{ki} \). However, we can show that the updates corresponding to all the \( a_{ki} \) duplicate copies can be collectively performed in \( O(\log a_{ki}) \) time.

### 5.5.3 CI-MAX and CI-MIN

We only present algorithm CI-MAX—algorithm CI-MIN is symmetric. CI-MAX is an algorithm for processing a collection of SSW queries of the form \( q_i = \{ k \in \mathcal{K} \mid \mathcal{G}_{\text{MAX}}(A)(S_k[T, 0]) \in (v_{li}, v_{hi}) \} \). Unlike CI-SUM and CI-COUNT, CI-MAX is exact and supports arbitrary range conditions.

We require some properties of the \text{MAX} aggregate over suffix windows to describe CI-MAX. Consider a substream \( S_k \). Assume for clarity that elements of \( S_k \) are distinct. At any point in time, an element \( a_{ki} \) of \( S_k \) can be the maximum element for a suffix window only if all the succeeding elements that have arrived so far are smaller than \( a_{ki} \). We call such elements \textit{suffix-max} elements. Note that a suffix-max element ceases to be one when a larger element arrives.

\textbf{Example 5.8} Figures 5.8(a)–(c) show recent elements belonging to three substreams \( S_{k1} \), \( S_{k2} \), and \( S_{k3} \) at time 57. Substream elements are shown using vertical lines with the height indicating the value of the element. The suffix-max elements are shown using solid lines, and the non-suffix-max elements using dotted lines. For example, element 30 of \( S_{k1} \) is a suffix-max element, while element 9 is not, since 20, a later element, is larger. \( \square \)

Every suffix-max element \( a_{ki} \) has an associated time interval such that whenever the left end of a suffix window falls within this interval, \( a_{ki} \) is the maximum element in the window. Specifically, let \( a_{kj} \) denote the suffix-max element that immediately precedes \( a_{ki} \) (i.e., no element between \( a_{kj} \) and \( a_{ki} \) is suffix-max). Then, \( \mathcal{G}_{\text{MAX}}(A)(S_k[T, 0]) = a_{ki} \) for all \( T \) such that \( \tau_{kj} < (\tau - T) \leq \tau_{ki} \), where \( \tau \) denotes the current time. We call the time interval
Consider \( \text{LOOKUP}(\tau, T, (v_l, v_h)) \), which seeks all substreams \( S_k \) with \( \text{MAX}_A(S_k[T, 0]) \in (v_l, v_h) \). From the definition of answer intervals, it follows that a substream \( S_k \) belongs to the output of this lookup operation if and only if \( (\tau - T) \in I_a(a_{ki}) \) and \( a_{ki} \in (v_l, v_h) \), for some suffix-max element \( a_{ki} \) of \( S_k \). This operation can be mapped to a well-known problem of line-segment intersection in computational geometry [92]: Associate with each suffix-max element \( a_{ki} \) a line-segment \( L_{ki} \) parallel to the x-axis. If \( I_a(a_{ki}) = (\tau_1, \tau_2) \), the endpoints of \( L_{ki} \) are \( (\tau_1, a_{ki}) \) and \( (\tau_2, a_{ki}) \). Similarly, associate a line-segment \( L_l \) with the lookup operation with endpoints \( (\tau - T, v_l) \) and \( (\tau - T, v_2) \). Then it can be verified easily that substream \( S_k \) belongs to the output of the lookup operation if and only if \( L_l \) intersects some line-segment of the form \( L_{ki} \).

**Example 5.9** Consider \( \text{LOOKUP}(57, 5, (30, 65)) \) over substreams \( S_{k1}, S_{k2}, \) and \( S_{k3} \) of Figure 5.8. The \( \text{MAX} \) aggregate for the window \( [5, 0] \) is 51, 23, and 40 for \( S_{k1}, S_{k2}, \) and \( S_{k3} \), respectively. Therefore, the output of the lookup is \( \{k_1, k_3\} \). Figure 5.8(d) shows the mapping of this lookup operation to the line-segment-intersection problem.

The CI-MAX algorithm is based on this mapping from the lookup operation to the line-segment intersection problem: At any point of time, CI-MAX stores all the current suffix-max elements and their answer intervals as line-segments in a *dynamic interval tree*,
STATE:
1. Let $t_{\text{max}}$ denote size of maximum window, and let $\tau$ denote current time.
2. $\text{ANSINTS} = \{\langle a_{ki}, I_a(a_{ki}), k \rangle : a_{ki} \text{ is suffix-max and } \tau_{ki} \geq (\tau - t_{\text{max}})\}$
3. Index $\text{ANSINTS}$ using an interval tree.

UPDATE($a_{km}, \tau_{km}$):
1. Delete all $\langle a_{kj}, I_a(a_{kj}), k \rangle$ such that $a_{kj} \leq a_{km}$ from $\text{ANSINTS}$.
2. Add $\langle a_{km}, I_a(a_{km}), k \rangle$ to $\text{ANSINTS}$.

LOOKUP($\tau, T,(v_l,v_h)$):
1. Compute the set $\{k \mid \langle m, I, k \rangle \in \text{ANSINTS} \land (\tau - T) \in I \land m \in (v_l,v_h)\}$ using the interval tree and return the same.

Figure 5.9: Algorithm CI-MAX

which is a well-known data structure for the line-segment intersection problem [92]. The lookup operation uses the dynamic interval tree to efficiently retrieve the desired answer.

**Theorem 5.6** Let $t_{\text{max}}$ denote maximum time interval of a sliding window that CI-MAX supports, and let $N_{\text{max}}$ denote the current number of elements belonging to all substreams with timestamps in the last $t_{\text{max}}$ time-units. The CI-MAX algorithm requires $O(N_{\text{max}} \log N_{\text{max}})$ space, has an amortized update time complexity of $O(\log^2 N_{\text{max}})$, and a lookup time complexity of $O(\log^2 N_{\text{max}} + |K_o|)$, where $|K_o|$ denotes the number of substreams in the output.

**Proof:** The maximum number of suffix-max elements with timestamps in the last $t_{\text{max}}$ time-units is $N_{\text{max}}$. Therefore, the maximum number of line-segments indexed by the dynamic interval tree is $N_{\text{max}}$. The space requirement and lookup complexity follow directly from that of the dynamic interval tree. For updates, we can show that the number of suffix-max elements added or deleted per update operation is $O(1)$ amortized. Therefore, the update time complexity of CI-MAX is the same as that of dynamic interval trees, which is $O(\log^2 N_{\text{max}})$.

5.6 Summary of Algorithms

Table 5.1 summarizes all of the algorithms presented in this chapter. For each algorithm the table lists the class of queries (ASW/SSF) and the class or type of aggregation function that the algorithm supports, and characterizes its asymptotic performance in terms of lookup
time, update time, and space requirement. Our first two ASW algorithms, B-INT and L-INT, work for any algebraic (A) or distributive (D) aggregation functions. They have different performance characteristics, and represent two different points on the Space-Update-Lookup tradeoff that we presented in Section 5.3. Our third ASW algorithm, R-INT, is designed for a subclass of algebraic and distributive aggregation functions called subtractive (S), and its performance is better than both B-INT and L-INT. All of the remaining algorithms are designed for specific aggregation functions, not general classes.

The output of an SSF lookup is a set, not a single value, and therefore the lookup time of all the SSF algorithms depends necessarily on the size of this set, $|K_o|$. Two of our SSF algorithms, CI-COUNT and CI-SUM, are approximate which is reflected in the error parameter $\epsilon$ in their update time and space requirement. Interestingly, the space requirement of these two algorithms is logarithmic in the maximum left end of a window, $N_{\text{max}}$, just like our algorithms of Chapter 4, while the space requirement of all the other algorithms is linear or slightly superlinear in $N_{\text{max}}$.

5.7 Experiments

We present three sets of representative experimental results illustrating the performance of our algorithms:

1. Comparison against naïve alternatives: For SUM, we compare the performance of the R-INT algorithm against the naïve extreme approaches suggested in Section 5.3. We
show that the naive approaches quickly become impractical, even for modest lookup and stream update rates.

2. **Performance of ASW algorithms**: We present raw performance numbers for three basic ASW algorithms, showing they are capable of handling very high lookup and stream update rates (millions of events per second).

3. **Performance of CI-COUNT**: Using real stock trade data, we compare the performance of algorithm CI-COUNT against the alternative approach of processing each ASW subquery independently. We show that CI-COUNT provides orders of magnitude improvement in overall performance.

   Note that the first two experiments are data-independent, since our algorithms and the naive approaches do not depend on the actual stream values. Thus, we use synthetically generated data for those experiments, while we use real financial data for the data-dependent third experiment.

   All experiments were performed on a 4-processor 700 Mhz Pentium III machine running Linux with 4 GB of main memory RAM. We implemented all the algorithms in this chapter as separate, independent code and not as part of the STREAM prototype.

### 5.7.1 Comparison with naive alternatives

For \textit{SUM} aggregation function, we compared the performance of the R-INT algorithm against the two naive approaches discussed in Section 5.3: (1) Materialize the results of all queries at all times; (2) Maintain the maximum required window of the input stream and perform all answer computations only at lookup time.

We used 1000 queries of the form $G_{\text{SUM}}(A)_S[N, 0]$ with tuple-based windows varying in size from $N = 0$ to $N = 1000$. We measured the total input rate each of the techniques was able to handle for different ratios of lookups to updates. Figure 5.10 shows the results. As expected, the performance of the full materialization approach is good when update rates are low, while that of the on-demand approach is good (although never better than R-INT) when lookup rates are low. However, both approaches deteriorate quickly as we move away from their favorable ends. The performance of R-INT remains stable throughout.
5.7.2 Performance of ASW algorithms

We present raw performance numbers for three basic algorithms: R-INT (for \texttt{SUM}), B-INT (for \texttt{MAX}), and B-INT-QNT (for \texttt{QUANTILE}). For each one we measured its performance handling updates and handling lookups separately. From these numbers we can easily derive the expected performance for a combined workload. Update handling was measured for different values of maximum window left-end $N_{\text{max}}$, and lookup handling was measured for different values of the query windows ($W$). For lookups, we set $N_{\text{max}} = 100,000$. Individual query windows were distributed uniformly over the entire permitted range.

Figures 5.11 and 5.12 present the results, showing that our algorithms handle up to millions of events per second, depending on window sizes. Although we did not focus on detailed implementation issues, all our algorithms can be implemented using simple arrays instead of more complicated structures. Further, many common operations such as computing the base-intervals can be implemented very efficiently using low-level bit operations.

Note that the $y$-axis in these figures does not go to zero. For example, in Figure 5.12 B-INT-QNT handles a lookup rate of about 22,000 per second even at the maximum window size (not obvious from the graph). In Figure 5.12 we deleted the plot for R-INT in order to better depict those for B-INT and B-INT-QNT. The performance of R-INT is uniformly
5.7.3 Performance of CI-COUNT

Unlike the algorithms in our first two sets of experiments, the performance of CI-COUNT is highly dependent on actual data and queries, specifically the selectivity of range conditions and the “spread” of aggregation answers across different substreams. Further, picking the right value of $\epsilon$ represents a tradeoff between update performance and lookup performance. A detailed study of these issues is beyond the scope of this thesis.

Here, we report on one particular experiment for CI-COUNT. We used a one-day stream of real stock trades from the TAQ database [106], containing approximately 5000 substreams based on ticker symbol. Our queries were synthetically generated by specifying a suffix window ranging from 15 minutes to 3 hours, monitoring stocks over the window with total trades above a given threshold. We selected the threshold to make its selectivity roughly .03–.05. Lookups and updates were equally interleaved.

We compared the performance of two approaches: (1) The naïve approach that uses algorithm R-INT to process each substream independently; (2) The approach that uses CI-COUNT in conjunction with R-INT to produce exact answers. For the second approach we
varied the parameter $p$, which directly affects the relative error $\epsilon$.

The naive approach processes only about 1000 inputs (lookups and updates) per second. In Figure 5.13 we see that CI-COUNT with an appropriately chosen value of $p$ (or $\epsilon$) processes about 25000 inputs per second. Note that the selectivity of the range condition imposes an upper bound on the relative performance of CI-COUNT when compared to the simple approach, suggesting that we can expect greater benefits for more selective range conditions.

5.8 Related Work

One class of techniques for resource sharing between different queries is based on detecting and exploiting common subexpressions. All of the multiquery optimization techniques for conventional one-time queries [93, 96] belong to this class. In the context of continuous queries, similar techniques have been used in the NiagaraCQ system [32]. Recent work in the TelegraphCQ project [29, 74] suggests using the Eddy operator [15] for sharing, and argues that since the Eddy operator does not fix a query plan, it exposes greater sharing possibilities.

The second class of techniques for resource sharing is based on collectively processing a set of structurally similar queries using a single query plan. Most previous work on this
class of resource sharing has focused on filters. All traditional pub-sub systems [3, 48, 60], and some continuous query processing systems [32, 74], use variants of predicate indexes for resource sharing in filters. Work on resource sharing for XML-based filters [5, 44, 62, 89], also belongs to this class. Recently, reference [45] considers the problem of sharing sketches for approximate join-based processing.

PSoup [30] briefly considers the problem of resource sharing in ASW operators. In particular, they propose using a ranked search tree, which has an $O(\log N_{max})$ cost for both update and lookup operations. All of our algorithms have an $O(1)$ cost for at least one of update or lookup. Also, PSoup does not consider ASW operators with quantiles, or SSF operators. Recently, Zhang et al. [116] have studied the resource sharing problem for aggregation queries with grouping. Their algorithms are based on the observation that $G_1 \mathcal{G}_{f(A)}(S)$ can be computed using $G_2 \mathcal{G}_{f(A)}(S)$ if $G_1 \subset G_2$. They do not consider queries with sliding windows.

5.9 Conclusion

In this chapter, we studied resource sharing for two classes of queries: aggregation over a sliding window (ASW) and substream filters (SSF). We provided a precise formulation of
the resource sharing problem for these two classes, indicated basic tradeoffs involved, and presented a suite of algorithms. All of our algorithms have precise theoretical guarantees and perform well in practice.

**ASW Operators**

For ASW operators, we presented algorithms that used generic properties of aggregation functions:

1. For distributive and algebraic aggregates, we presented two algorithms: B-INT and L-INT. B-INT has low update and space costs, while the L-INT has low lookup cost.

2. We identified a subclass of distributive and algebraic aggregates called subtractable aggregates, and presented an algorithm specific to this class called R-INT that is more efficient than the general B-INT and L-INT.

3. Since there is no common property for the class of holistic aggregates (they are defined as aggregates that are not algebraic), we cannot have a general algorithm for this class. However, we presented an algorithm B-INT-QNT for quantiles.

**SSF Operators**

For SSF operators, for specific aggregation functions (COUNT, SUM over positive values, MAX, MIN) and window types (suffix, time-based), we presented new algorithms for resource sharing. Our algorithms for COUNT and SUM are approximate: they slightly relax the range predicate. However the approximation is controlled by a parameter $\epsilon$, which can be made as small as desired at the expense of increased space and update time. Further, exact answers can be obtained by postprocessing.
Chapter 6

STREAM Prototype

Chapters 3–5 studied various research issues in processing continuous queries over data streams. Most of our results and techniques were generic, and not specific to any particular system. In this chapter, we describe a specific data stream management system we helped develop, called STREAM.

STREAM is a comprehensive general-purpose DSMS prototype developed as part of the STREAM project [101], and it is one of the earliest such prototypes. STREAM supports declarative CQL queries over streams and relations. It is a fairly complete prototype, and it can handle complex queries and high input data rates. We have released STREAM as public, open-source software under the BSD license, and it has been used by a number of other research projects for application development, comparison, and study (see, for example [42]).

We begin this chapter with a brief description of the portion of the CQL query language from Chapter 2 supported by STREAM. In Section 6.2, we present an overview of query execution in STREAM. In Section 6.3, we describe a graphical visualizer that we have developed for a human user to interact with STREAM. The visualizer allows the human user to register queries, feed input, and view query plans and query output. In Section 6.4, we present some low-level details of the STREAM prototype, including some performance numbers. In Section 6.5, we describe how the algorithms and techniques of the previous chapters can be integrated into STREAM, and finally present our conclusions in Section 6.6.
6.1 Query Language

STREAM allows applications to register and run CQL queries over input streams and relations. STREAM supports a large subset of CQL including the basic window operators (tuple-based, time-based, and partitioned), the three relation-to-stream operators (Istream, Dstream, and Rstream), and a reasonable subset of SQL queries. (Recall from Chapter 2 that CQL uses SQL queries for its relation-to-relation operators.) Many of the more esoteric SQL constructs are not supported. In addition, the system does not support subqueries in a Where clause, or a Having clause in a group-by query. Finally, windows cannot have a slide parameter (recall Section 2.6.1).

STREAM supports views: queries whose output can be used in other queries just like an input stream or relation. Using views, we can express complex query logic, including those that might otherwise require the SQL constructs not supported by STREAM, e.g., subqueries and Having clauses.

6.2 Query Execution

When a CQL query is registered with STREAM, a query plan is compiled from it. Query plans are composed of operators, which perform the actual data processing; queues, which buffer data as it moves between operators; and synopses, which hold operator state. Figure 6.1 shows an example query plan: the “ladders” denote queues, the ellipses denote operators, and the rectangles denote synopses. We discuss the plan shown in Figure 6.1 in more detail in Section 6.2.4.

6.2.1 Operators

Operators are the basic data processing units in a query plan. An operator takes one or more relations and or streams as input, and produces a relation or a stream as output. As in a traditional DBMS, a plan for a query connects a set of operators in a tree\(^1\): The output of a child operator forms an input of its parent operator, the input streams and relations of the query form the input of the leaf operators, and the output of the root operator forms the output of the entire query. For example, in Figure 6.1, the two input streams \(S_1\) and

\(^1\)More generally, a directed acyclic graph.
Figure 6.1: Plan for the query in Section 6.2.4

$S_2$ form the input of the leaf operators $O_1$ and $O_2$, and the output of these two operators forms the input of their common parent, operator $O_3$.

For purposes of operator input and output, a stream is represented as a sequence of timestamped tuples and a relation as a sequence of timestamped updates (insertions and deletions); the sequences are always in increasing timestamp order. We slightly abuse terminology and call both timestamped stream tuples and timestamped relation updates as elements. An operator conceptually runs forever in an infinite loop: It reads the sequences of elements corresponding to its input streams and relations and produces the sequence of elements corresponding to its output stream or relation.

Our representation of relations as sequences of timestamped updates is an important design feature, which enables incremental processing for relational operators such as join and group-by and aggregation. We illustrate incremental processing using a detailed example in Section 6.2.4.
### Table 6.1: Operators used in STREAM query plans

<table>
<thead>
<tr>
<th>Operator</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>Relation → Relation</td>
<td>Filters tuples based on predicates</td>
</tr>
<tr>
<td>project</td>
<td>Relation → Relation</td>
<td>Duplicate-preserving projection</td>
</tr>
<tr>
<td>join</td>
<td>Relation × Relation → Relation</td>
<td>Joins two input relations</td>
</tr>
<tr>
<td>str-join</td>
<td>Stream × Relation → Stream</td>
<td>Joins a stream with a relation</td>
</tr>
<tr>
<td>group-aggr</td>
<td>Relation → Relation</td>
<td>Performs grouping and aggregation</td>
</tr>
<tr>
<td>distinct</td>
<td>Relation → Relation</td>
<td>Performs duplicate elimination</td>
</tr>
<tr>
<td>union</td>
<td>Relation × Relation → Relation</td>
<td>Computes duplicate preserving union of two relations</td>
</tr>
<tr>
<td>except</td>
<td>Relation × Relation → Relation</td>
<td>Computes difference of two relations</td>
</tr>
<tr>
<td>range-win</td>
<td>Stream → Relation</td>
<td>Implements a time-based window</td>
</tr>
<tr>
<td>row-win</td>
<td>Stream → Relation</td>
<td>Implements a tuple-based window</td>
</tr>
<tr>
<td>partn-win</td>
<td>Stream → Relation</td>
<td>Implements a partitioned window</td>
</tr>
<tr>
<td>istream</td>
<td>Relation → Stream</td>
<td>Implements Istream semantics</td>
</tr>
<tr>
<td>dstream</td>
<td>Relation → Stream</td>
<td>Implements Dstream semantics</td>
</tr>
<tr>
<td>rstream</td>
<td>Relation → Stream</td>
<td>Implements Rstream semantics</td>
</tr>
</tbody>
</table>

Table 6.1 lists the operators used in STREAM, their input-output signatures, and a brief description. All of the operators except str-join correspond to some logical operator in our algebra of Section 2.7. (Nevertheless there is a distinction between physical operators in a query plan and logical operators used in query specification. In this chapter, by default, operators refer to physical operators.) The str-join operator over a stream $S$ and a relation $R$ implements the semantics of the following CQL query:

```
Select Rstream(*)
From S [Now], R
Where <join-predicate>
```

Recall from Section 2.6.7 that this is a very useful query construct in CQL.
6.2.2 Queues

A queue buffers the sequence of elements flowing from one operator to another. The operator producing the sequence simply enqueues the elements of the sequence into the queue, and the operator consuming the sequence reads off the elements from the queue. (There is exactly one writer and one reader for a queue.) Queues decouple operators from one another. An operator does not directly interact with other operators: It simply reads off elements from its input queues and writes elements to its output queue. In the example query plan shown in Figure 6.1, operator $O_1$ writes its output elements to queue $q_3$, which are subsequently read off the queue by operator $O_3$.

This decoupling enables a scheduler-based architecture for STREAM. A single global scheduler schedules currently active operators for specific durations of activity. The duration is controlled by a parameter $\text{maxElements}$, which denotes the maximum number of input elements that a scheduled operator can process. When an operator is scheduled it checks if there are any available elements in its input queues. If there are, it processes up to $\text{maxElements}$ of them, enqueues any resulting output elements in its output queue, and returns control to the scheduler.

Our scheduler-based architecture is significantly different from the traditional pull-based architectures, where an operator directly invokes “getNext()” to get data from its child operators [50]. The scheduler-based architecture is more suitable for data streams than the traditional architecture, especially if the streams are bursty, as shown by Babcock et al. [16].

6.2.3 Synopses

All of the operators listed in Table 6.1, except select and project, need to maintain some state during execution. For example, the join operator needs to store the current bag of tuples for each of its input relations. A Synopsis is an entity that holds operator state. It encapsulates and hides the details of how the state is physically stored from its “owning” operator, and allows the operator to access and modify its state using a simple interface.

In the query plan of Figure 6.1, the synopses are shown as rectangles. The join operator $O_3$ owns two synopses, $Syn_4$ and $Syn_5$, while operators $O_1$ and $O_2$ own synopses $Syn_1$ and $Syn_2$, respectively. We describe the contents of these synopses in Section 6.2.4.

Specifically, a synopsis stores a time-varying bag of tuples, and exports an interface
to update or access the bag. There are different kinds of synopses that differ in the interface that they provide. Table 6.2 lists the kinds of synopses used in STREAM with a brief description of their interface. A relation synopsis allows an operator to insert a new tuple into the bag of tuples that it stores, delete an existing tuple from the bag, and scan the contents of the bag; the scan can be an indexed scan that returns all tuples satisfying a given predicate. Relation synopses are used by various relation-to-relation operators such as join, group-aggr, and distinct. A window synopsis allows an operator to insert a new tuple and delete the oldest tuple from the bag of tuples that it stores, and it is used by time-win and row-win operators. A partition-window synopsis is similar to window synopsis, except that it deletes the oldest tuple of a given partition (or group), and it is used by the partn-win operator.

The details of how a synopsis physically stores its bag of tuples is closely related to our memory management scheme, and we describe them in Section 6.2.5.

6.2.4 Example

In this section, we illustrate query execution in STREAM for the following very simple example CQL query:

```sql
Select *  
From S1 [Rows 100], S2 [Rows 200]  
Where S1.A = S2.A
```
CHAPTER 6. STREAM PROTOTYPE

170

The query is a sliding-window join over two streams $S_1$ and $S_2$; both the sliding windows are tuple-based windows.

**Query Plan**

Figure 6.1 in Page 166 shows a query plan used in STREAM for executing the query above. The query plan contains three operators ($O_1, O_2, O_3$), five queues ($q_1$–$q_5$), and four synopses ($Syn_1$–$Syn_4$). The window operators $O_1$ and $O_2$ apply a tuple-based sliding window over input streams $S_1$ and $S_2$, respectively, and the join operator $O_3$ joins the relations output by $O_1$ and $O_2$. Queues $q_1$ and $q_2$ buffer elements of input streams $S_1$ and $S_2$, respectively. Queue $q_3$ buffers elements flowing from operator $O_1$ to operator $O_3$, and $q_4$ does the same for elements flowing from operator $O_2$ to operator $O_3$. Queue $q_5$ buffers the elements output by $O_3$. The window operators each contain a window synopsis ($Syn_1$ and $Syn_2$), while the join operator contains two relational synopses ($Syn_3$ and $Syn_4$).

**Query Execution**

Recall from Section 6.2.1 that a relation is represented as a sequence of timestamped updates. For a given relation $R$, we use $\langle +, \tau, t \rangle$ to denote an insertion of tuple $t$ into $R$ with timestamp $\tau$, and $\langle -, \tau, t \rangle$ to denote a deletion of tuple $t$ with timestamp $\tau$. In other words, using the terminology from Chapter 2, there is one $\langle +, \tau, t \rangle$ for every $t \in R(\tau) - R(\tau - 1)$, and one $\langle -, \tau, t \rangle$ for every $t \in R(\tau - 1) - R(\tau)$.

Operator $O_1$ performs the following steps when a new element $\langle \tau, t \rangle$ arrives on stream $S_1$: It writes the element $\langle +, \tau, t \rangle$ on its output queue $q_3$ to reflect the addition of $t$ to the window, and inserts tuple $t$ in its synopsis $Syn_1$. If at least 100 elements have arrived on $S_1$, operator $O_1$ deletes the oldest tuple (say $t'$) from $Syn_1$ and writes the element $\langle -, \tau, t' \rangle$ on its output queue $q_3$. Operator $O_2$ takes similar steps to process a new element of $S_2$.

The join operator $O_3$ maintains two synopses: $Syn_3$ contains the current bag of tuples of its left relation, i.e., the current window over $S_1$, and $Syn_4$ contains the current bag of tuples of its right relation, i.e., the current window over $S_2$. Operator $O_3$ repeatedly performs the following sequence of steps when it is scheduled: It picks the next element in its input queues with the smallest timestamp. Without loss of generality, we assume that

---

2There are three more “system” operators in the query plan that are not shown in Figure 6.1: two “input” operators that convert elements of $S_1$ and $S_2$ into the internal representation of STREAM and one “output” operator that pushes the results out of STREAM.
this element belongs to queue $q_3$. If the next element in $q_3$ is an insertion $\langle +, \tau, t \rangle$, $O_3$ first inserts tuple $t$ into $Syn_3$. Next, it scans all tuples of $Syn_4$ that have a value $t.A$ in their $A$ attribute. For every tuple $t'$ returned by the scan, $O_3$ writes the element $\langle +, \tau, t \cdot t' \rangle$ on its output queue $q_5$. (The notation $t \cdot t'$ denotes the concatenation of tuples $t$ and $t'$.) Intuitively, the insertion of $t$ into the left relation causes the insertion of the tuple $t \cdot t'$ in the join. If the next element in $q_3$ is a deletion $\langle -, \tau, t \rangle$, $O_3$ deletes the tuple $t$ from $Syn_3$, and for every tuple $t'$ in $Syn_4$ with a value $t.A$ in its $A$ attribute, writes the element $\langle -, \tau, t \cdot t' \rangle$ onto its output queue $q_5$. Intuitively, deletion of $t$ from the left relation causes the deletion of the tuple $t \cdot t'$ from the join.

**Synopsis Sharing**

Both synopses $Syn_1$ and $Syn_3$ store bags of tuples corresponding to some window of $S_1$, but at any given point in time, the position of the window for $Syn_1$ is slightly different from the position of the window for $Syn_3$. We can show this difference is exactly half the number of elements currently in queue $q_3$, so if there are a few elements in $q_3$, the overlap between the bags stored by $Syn_1$ and $Syn_3$ is large. Synopses $Syn_2$ and $Syn_4$ are similarly related.

**Example 6.1** Let the sequence elements of $S_1$ be $\langle \tau_1, t_1 \rangle, \langle \tau_2, t_2 \rangle, \langle \tau_3, t_3 \rangle, \ldots$. Consider a point in time, when operator $O_1$ has processed 500 elements of $S_1$, and let $q_3$ contain 6 elements at this point. $Syn_1$ currently contains the bag of tuples $\{t_{401}, t_{402}, \ldots, t_{500}\}$. From our description of $O_1$, we can infer that the 6 elements enqueued currently in $q_3$ are: $\langle +, \tau_{498}, t_{498} \rangle, \langle -, \tau_{498}, t_{398} \rangle, \langle +, \tau_{499}, t_{499} \rangle, \langle -, \tau_{499}, t_{399} \rangle, \langle +, \tau_{500}, t_{500} \rangle, \langle -, \tau_{500}, t_{400} \rangle$. Operator $O_3$ has processed all input elements on queue $q_3$ with timestamps up to $\tau_{497}$, so $Syn_3$ contains the last 100 tuples of $S_1$ at application time 497, i.e., the tuples $\{t_{398}, \ldots, t_{497}\}$. Therefore, $Syn_1$ and $Syn_3$ have 97 tuples in common. □

**STREAM** exploits this redundancy and uses a common store for tuples belonging to related synopses. Figure 6.1 schematically illustrates such state sharing, and we provide more details in the next section.

### 6.2.5 Memory Management

Designing an effective memory management scheme for a dynamic system like STREAM, where thousands of tuples can enter and leave the system every second, is challenging. A
naive approach that uses a generic, standard memory-management library such as \texttt{malloc} is neither space- nor time-efficient: It incurs the overhead of a large number of library calls, and memory gets fragmented and poorly used. We have designed and implemented our own memory management subsystem for \textsc{Stream}. Our design is more efficient than the naive approach, it understands and exploits detailed knowledge of tuple-usage patterns, and it enables the synopsis sharing we discussed in the previous section.

Memory management in \textsc{Stream} works at two levels. First, there is a global, system-wide \textit{memory manager} that gets a large chunk of memory from the operating system. The memory manager distributes this memory across several \textit{stores}, which manage memory at the level of individual (intermediate) streams and relations. A store is a complex object that supports two functions: First, it allocates and deallocates memory for tuples belonging to a single stream or relation. Second, it holds the state for one or more related synopses, enabling synopsis state sharing. A synopsis itself does not store any tuples: It is just a thin interface, and it forwards all its operations—inserts, deletes, and scans—to its associated store. A store handles these forwarded operations and ensures that it provides a consistent view to each synopsis that it supports. The exact details of how stores are implemented involves many low-level details that are not presented here. We refer an interested reader to the \textsc{Stream} system documentation [102] and the \textsc{Stream} user manual [103]. We illustrate some basic ideas using the following example.

\textbf{Example 6.2} Consider the query plan of Figure 6.1. As Figure 6.1 suggests, a single store holds the state for both Syn$_1$ and Syn$_3$. At any point in time, the store stores the smallest window of $S_1$ that subsumes the current windows of both Syn$_1$ and Syn$_3$. Assume that the current state of the system is as described in Example 6.1: Syn$_1$ logically holds the window of tuples $\{t_{398}, \ldots, t_{497}\}$ and Syn$_3$ logically holds the window $\{t_{401}, \ldots, t_{500}\}$. At this point in time, the store stores the window $\{t_{398}, \ldots, t_{500}\}$. In addition, for each of Syn$_1$ and Syn$_3$, the store maintains two pointers that identify the left and right ends of the window for the synopsis. For example, the left pointer for Syn$_3$ points to tuple $t_{401}$ and the right pointer to tuple $t_{500}$.  

\textbf{6.3 \textsc{Stream} Visualizer}

We have developed a sophisticated graphical visualizer to interact with \textsc{Stream} and inspect the state of the system while it is executing. Specifically, the visualizer allows a user
1. Register and run queries, feed streams and relations into the system, and view query answers.

2. View query plans generated by STREAM for registered queries. A displayed query plan contains details of how operators in the plan are connected to each other, the synopses used by the operators, and the stores used to allocate and hold tuples.

3. Monitor various dynamic properties of the system such as the rate of flow of elements within a queue, the current number of tuples in a synopsis, the memory used by a store, and selectivity of a join operator. These properties are plotted using graphs as time-varying values.

Figure 6.2 shows a screenshot of the visualizer. The large pane on the left shows the plan for a registered query, which happens to be a sliding-window join query similar to the
example query used in Section 6.2.4. The legend in the bottom of the right pane describes the meaning of various images used in the query plan. For example, the rectangles with text within them are operators, the solid lines are queues connecting the operators, the circles are synopses, and so on. The left pane (query plan) of Figure 6.2 also shows three monitoring graphs. For example, the topmost monitoring graph shows the rate of flow of elements in the output queue of the join operator.

Our technique for implementing the monitoring graphs shown in Figure 6.2 is based on introspection queries over a special system-managed stream called SysStream. When a specific dynamic monitoring task is desired, e.g., monitoring recent join selectivity, the relevant entity writes its statistics periodically on SysStream. Then a standard CQL query, typically a windowed aggregation query, is registered over SysStream to compute the desired continuous result, which is fed to the monitoring graph in the visualizer.

6.4 Prototype Details

In this section, we present some further details and mechanics of the STREAM prototype, and briefly describe its performance.

Programming Language and Environment

STREAM is a fairly large system: The STREAM server is about 56,000 lines of C++ code. The code for the main execution engine of STREAM which includes all the components of the architecture described in Section 6.2 such as operators, queues, synopses, and stores is relatively small, only about 18,000 lines of code. The remainder of the code deals with metadata management, query parsing, and plan generation. The visualizer code is written in Java and is about 13,000 lines. STREAM runs only on the Linux platform, while the visualizer runs on any system with a Java runtime environment.

Usage

The core of STREAM is designed as a library with a relatively simple interface, and therefore can be linked to arbitrary C++ programs. Further, this core is input-output agnostic: By

---

3We ran a simple wc command against the source files to get this number, so it includes comments and empty lines.
implementing simple interfaces we can input data into system from arbitrary data sources (e.g., network feeds, local files) and output results to arbitrary destinations.

We have implemented two standalone programs using the library: The first is a networked server, which allows any client to connect to it and use the STREAM system over a network. The STREAM visualizer is one such client. The second is a thin command-line wrapper around the library that is useful mainly for debugging and measuring performance. The details of using the STREAM library and the programs above can be found in the STREAM user manual [103].

### Stability and Performance

STREAM is an academic prototype, and it has not been extensively tested and debugged. However, the system is fairly stable: We have run it it without any problem for extended periods of time (up to a few days) and for fairly complex input queries such as those of the Linear Road Benchmark [11].

We now present the performance of STREAM on five representative queries, denoted $Q_1$–$Q_5$. Queries $Q_1$–$Q_3$ are listed in Table 6.3: Query $Q_1$ is a simple selection query, $Q_2$ is a sliding-window aggregation query, and $Q_3$ is a sliding-window join query. Queries $Q_4$ and $Q_5$ are complex queries specified using several subqueries or views (recall Section 6.1). Query $Q_4$ is the Linear Road query: We presented a simplified version of this query in Chapter 2, Section 2.6.5; however, here we present our results for the complete query without simplifications. The complete query can be found in the Stream Query Repository [104]. Finally, Query $Q_5$, shown in Figure 6.3, was used by Demers et al. [42] to compare the performance of STREAM against Cayuga, a system designed specifically for identifying patterns in event streams. Informally, the query identifies all stocks whose price fluctuation roughly resembles the letter “M” in the input stock stream.

| $Q_1$ | Select * 
| From S where A = 1 |
| $Q_2$ | Select Sum(A) 
| From S [Rows 10] |
| $Q_3$ | Select * 
| From S1 [Rows 10], S2 [Rows 5] 
| Where S1.A = S2.A |

Table 6.3: Queries used in STREAM performance measurements
CHAPTER 6. STREAM PROTOTYPE

Figure 6.3: The M-Pattern Query \((Q_5)\) in CQL [42].

For Queries \(Q_1\)–\(Q_3\), we generated synthetic input streams. For Query \(Q_3\), the join selectivity was 1, meaning that each \(S_1\) tuple joined with exactly one \(S_2\) tuple, and vice-versa. For Query \(Q_4\), we used the data provided by the Linear Road Benchmark [11]. We did not measure the performance of \textsc{stream} for \(Q_5\) ourselves; we are merely presenting the performance measurements given in [42]. We used a four processor 2.8 GHz Pentium machine with 1 GB main memory for measuring the performance of Queries \(Q_1\)–\(Q_4\); Demers \textit{et al.} [42] used a 3 GHz Pentium machine with 1 GB of RAM for measuring the performance of \(Q_5\).

Table 6.4 shows the performance of \textsc{stream} in terms of the maximum rate of input tuples handled for the five queries \(Q_1\)–\(Q_5\). For simple selection queries \textsc{stream} is able to process input at a rate exceeding 15 million tuples/second. This number is lower for
<table>
<thead>
<tr>
<th>Query</th>
<th>Maximum input rate handled (tuples/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$ Selection</td>
<td>$1.67 \times 10^7$</td>
</tr>
<tr>
<td>$Q_2$ Sliding-Window Aggregation</td>
<td>$1.15 \times 10^6$</td>
</tr>
<tr>
<td>$Q_3$ Sliding-Window Joins</td>
<td>$5.75 \times 10^5$</td>
</tr>
<tr>
<td>$Q_4$ Linear Road [11]</td>
<td>$2.21 \times 10^4$</td>
</tr>
<tr>
<td>$Q_5$ M-Pattern Query [42]</td>
<td>$1.10 \times 10^5$</td>
</tr>
</tbody>
</table>

Table 6.4: Performance of STREAM for representative queries

aggregation and join queries, although it is still around half a million to one million tuples/second. The maximum rate for $Q_4$ and $Q_5$ is much lower, but these are exceedingly complex queries: $Q_4$ involves around 70 physical operators while $Q_5$ involves around 20 operators. Further, the performance of STREAM for $Q_5$ is within a factor 2 of the performance of Cayuga on the same query, although Cayuga is designed specifically to support pattern matching queries.

### 6.5 Integrating Thesis Techniques into the Prototype

Apart from the work on CQL (Chapter 2), which forms the query language for STREAM, the work presented in the remaining chapters (Chapters 3, 4, and 5) has not yet been integrated into STREAM. We now describe how we can incorporate the techniques in these chapters into the prototype, and discuss some implications of doing so.

**Memory Requirement of Queries (Chapter 3)**

The bulk of Chapter 3 deals with mathematically characterizing memory requirements of queries, and is not directly relevant to the prototype. However, the memory-efficient query evaluation strategy for bounded-memory computable queries presented in Section 3.6 is relevant, and can be incorporated easily into the prototype. In order to do so, we need to extend the relational synopsis (recall Section 6.2.3) to accept a filter and an attribute projection list as parameters. Any new tuple inserted into the synopsis that satisfies the filter is projected on the attributes of the projection list and stored; other tuples are discarded. We can show that the algorithms of Section 3.6 can be implemented using these extended relational synopses by using appropriate filters and projection lists.
These extended relational synopses can be used to reduce memory for a wider class of queries, not just bounded-memory computable queries, and to implement other state reducing techniques such as those based on stream constraints [24]. For example, consider the CQL query:

\[
\text{Select } * \text{ From } S1 \text{ [Range 5 Minutes]}, S2 \text{ [Range 10 Minutes]} \text{ Where } S1.A = S2.A \text{ and } S2.A > 5
\]

For this query, a regular relational synopsis stores all the tuples of S1 in the current window. However, for correctness, it is sufficient to store only those tuples of S1 in the current window satisfying \( S1.A > 5 \), which can be easily implemented using an extended synopsis.

**Approximate Statistics over Streams (Chapter 4)**

Currently, STREAM computes only exact answers for queries. In order to integrate the algorithms of Chapter 4, we need to first enhance the STREAM interface to accept an approximation parameter \( \epsilon \) for queries computing quantiles or frequent elements over sliding windows. (We could also use other interface alternatives such as specifying available memory for the query from which the best possible \( \epsilon \) can be inferred.) Also, the current prototype recognizes only MAX, MIN, SUM, COUNT, and AVG aggregation functions, so we need to modify the parsing module to recognize quantile and frequent elements statistics.

In the execution subsystem, we need to implement one new operator for each algorithm of Chapter 4. Note that, by our construction of the algorithms, these “approximate” operators subsume the functionality of both windowing and aggregation, so we do not need a separate window operator while using these operators. One important difference between these approximate operators and the exact operators is in their state: The state for the exact operators can be represented as bags of tuples, while it is hard to cast the state of the approximate operators in terms of bags of tuples. Therefore, we need to bypass the existing synopsis and memory-management machinery, which are designed for bags of tuples, for these operators. These operators need to directly interact with the memory manager, and perform memory management on their own.
Resource Sharing (Chapter 5)

There is a mismatch in the output semantics of STREAM and the output semantics under which the algorithms of Chapter 5 are designed. Currently, STREAM “pushes” out its output irrespective of whether it is a stream or a relation; if the output is a relation, STREAM pushes out a sequence of timestamped updates to the relation. In contrast, the algorithms of Chapter 5 assume that the answers of queries are “pulled” using an explicit lookup operation. Once we modify the output semantics of STREAM to include the pull model, the rest of the details of incorporating the algorithms of Chapter 5 are exactly identical to those for the algorithms of Chapter 4: We need to design new operators, one per algorithm; the operators again subsume the functionality of both windowing and aggregation; and, finally, the operators need to perform their own memory management.

6.6 Conclusion

In this chapter, we presented an overview of the STREAM prototype DSMS. We described the query language supported by STREAM, and provided a high-level description of query execution. We then described the STREAM visualizer that provides a graphical interface to interact with the STREAM system and inspect the state of the system during its execution. We presented some low-level prototype details, including its performance for a representative set of queries. Finally, we described how the work presented in the remainder of the thesis can be integrated into the STREAM prototype. The STREAM prototype is available at http://www-db.stanford.edu/stream/code.
Chapter 7

Conclusion

7.1 Overview of Thesis

This thesis addresses a number of challenges in processing continuous queries over data streams. We began with query language design in Chapter 2, where we presented a new language for continuous queries called CQL. Our language design consists of a core abstract semantics for continuous queries that is based on generic black-box operators, and an instantiation of this core semantics using specific operators, mostly derived from the SQL language. An important feature of our language design is its extensibility: The two-step design enables easy extension of CQL or even derivation of new language (e.g., the algebra of continuous operators, ACO, that we presented in Section 2.7). CQL is one of the earliest continuous-query languages proposed, and it is empirically very expressive.

In Chapter 3, we studied the problem of characterizing the amount of memory required to process a continuous query. We presented a precise characterization for a large class of queries, roughly corresponding to single-block CQL queries. Interestingly, our characterization shows that all queries in this class either require a bounded amount of memory, or require memory that grows linearly with the input size.

For many queries, the problem of high memory requirement can be mitigated by settling for a lower result accuracy, and designing such memory-efficient, approximate algorithms was the subject of Chapter 4. We presented new algorithms for approximately maintaining two statistics—quantiles and frequent elements—over a sliding window. An exact computation provably requires memory that is as large as the window size, but our approximate algorithms require memory that is only logarithmic in the window size. Both
these statistics are used frequently to identify skew in data. Further, quantiles over sliding windows may prove useful in continuous-query optimization, just like quantiles over traditional relations are useful in traditional query optimization. Recent work [56] has implemented our algorithms of Chapter 4 over Graphics Processors (GPUs) for higher performance.

The next part of the thesis dealt with resource sharing: the sharing of resources such as memory and computation across different queries. Continuous queries provide rich opportunities for resource sharing, since there is often high temporal overlap among queries due to their long-running nature. In Chapter 5, we considered an important class of queries called sliding-window aggregates, and presented a suite of resource-sharing techniques that cover a wide range of possible scenarios: different classes of aggregation functions (algebraic, distributive, holistic), different window types (time-based, tuple-based, suffix, historical), and different input models (single stream, multiple substreams). Resource-sharing techniques are important in subscription-based applications where a large number of independent users monitor information of interest using continuous queries.

Finally, in Chapter 6, we presented STREAM, a prototype data stream management system developed as part of the STREAM project. STREAM supports continuous queries expressed in CQL over streams and relations. STREAM has been developed as a platform for research in continuous-query and data-stream processing, and we have released STREAM as open-source, public software.

7.2 Future Directions

We begin our suggestions for future work by reviewing the applications landscape for continuous queries, and suggest some broad directions for future work based on the mismatch between capabilities of current DSMSs and the requirements of applications. We describe more specific problems in Sections 7.2.2–7.2.4: Section 7.2.2 discusses the problem of designing distributed data stream management systems, Section 7.2.3 considers query language expressiveness issues, and Section 7.2.4 presents directions for future work in resource sharing.
7.2.1 Applications Landscape

Applications that perform continuous processing over data streams are quite diverse, and require a variety of functionality from a DSMS. Applications differ from each other along at least four different dimensions:

1. **Query Language**: Applications differ widely on how they process their input streams, and therefore in the kind of query language interface that they require. Some applications such as network health monitoring [37] require support for relational operations such as filters, joins, and aggregations, while others such as intrusion detection [72, 111] require the ability to perform complex pattern matching over their input. Note that CQL is designed for the former class of applications. There has been some recent work towards designing query languages for the latter class of applications [42].

2. **Data Rate**: There is a wide variation in the input data rates of applications. Some sensor-network applications need to process fewer than a hundred tuples per second, while network monitoring applications might have to process millions of tuples per second.

3. **Query Complexity and Number**: Some applications involve a small number of complex continuous queries, while others involve a large number of relatively simple queries. The hypothetical Linear Road application of Section 2.3 is an example of the former—it has a single complex query—while publish-subscribe systems, which process millions of simple filter queries, constitute examples of the latter type. Resource sharing techniques such as those described in Chapter 5 are important for the latter type of applications.

4. **Spatial Data Locality**: The input data of an application could be available in a centralized location (e.g., stock data in a financial application) or physically distributed (e.g., sensor data or network traffic data from different routers).

Current DSMSs support a small subset of the functionality listed above. For example, STREAM supports a relational query language (CQL), handles reasonably high data rates, is designed to support a small number of complex queries, and is a centralized system; TelegraphCQ [29] and Aurora [1] have a similar functionality. One broad direction for
future work is to identify useful combinations of functionality not supported by current DSMSs and design new DSMSs or augment existing ones to support them. There has been some recent work in this direction: For example, Cayuga [42] is a (centralized) system that supports a large number of event-pattern queries.

7.2.2 Distributed Data Stream Management Systems

As mentioned in the previous subsection (“Applications Landscape”), for many applications such as network monitoring and sensor data processing, the input data is physically distributed. Shipping all the data to a central location could potentially incur high communication cost and might not be always feasible. The observation above motivates the need for distributed data stream management systems. Distributed DSMSs can perform computation closer to the data source, and can therefore reduce communication cost. They can also exploit parallelism for greater efficiency. From an application perspective, distributed DSMSs hide details of physical distribution of data and its movement, simplifying application design.

The fundamental challenge introduced by distributed DSMSs is that of identifying communication-efficient query plans. Previous work suggests that this is nontrivial even for specialized query classes like Top-k statistics [21]. We do not know good algorithms for even the most basic query classes: For example, how do we maintain median for a dynamic bag of values distributed across several nodes? Generating plans for general queries would require us to heuristically search through the space of possible plans, which requires us to collect and meaningfully use information about the execution environment such as processing power of different nodes, and network connectivity and bandwidth between nodes. Further, unlike traditional distributed query processing, these parameters could change dynamically—nodes can become loaded, network links can become clogged, and so on—further increasing the problem complexity.

There has been some recent work on distributed DSMSs [33], but this work focuses mostly on infrastructural issues of a distributed DSMS, and not so much on query processing issues discussed above.
7.2.3 Query Language Expressiveness

There has been little work on formal study of continuous query languages and their expressiveness. Most of the “practical” query languages such as CQL and TelegraphCQ are unwieldy from the point of view of theoretical analysis, so designing a more formal language might be necessary for this exercise. There also seems to be interesting connections between continuous query languages and the work on incremental evaluation of Datalog queries [46], which shows that repeatedly evaluating Datalog queries on the updates to a database has more expressive power than regular nonrecursive Datalog queries. Continuous queries also have access to updates, suggesting that they might be related to incremental Datalog queries.

7.2.4 Resource Sharing

Current resource sharing techniques are limited to a small class of queries: filters, sliding window aggregates (which we covered in Chapter 5), and, more recently, for complex event patterns [42]. Many interesting and practical classes of queries remain to be explored. We mention two classes of queries:

1. \textbf{ASW + filters}: This class of queries is the class of sliding window aggregate queries (ASW) that we considered in Chapter 5 but with an optional filter predicate over the input stream. In ACO (recall Section 2.7) these queries are of the form \( G_F(\sigma_P(S)[W]_T) \) or of the form \( G_F(\sigma_P(S)[N]) \). The queries that are input to the resource sharing problem can now differ both in the window size and the filter predicate. The techniques of Chapter 5, which compute partial aggregates over various intervals of the input stream, are not directly applicable to this problem variant since different filters can remove different subset of elements from a given interval of stream. We can also study resource sharing for SSF queries with filters; our techniques of Chapter 5 do not work for the same reason as for ASW queries. Note that the two example Traderbot queries in Chapter 5 Page 134 apply filters over their input streams.

2. \textbf{ASW + Threshold-based Trigger}: These are ASW queries with an associated threshold. A query generates an output trigger when its ASW answer exceeds the threshold value. An example of such a query is “generate an alert when average value of Microsoft stock over the last 1 hour crosses 30”. In ACO, these queries are of the form
\[ \sigma_P(G_F(S[W]_T)) \text{ or of the form } \sigma_P(G_F(S[N])). \]

Note that there is a subtle difference in semantics between the output of regular ASW queries and the threshold-ASW queries: The output of the former class of queries is pulled using a lookup operation, while the output of the latter is pushed automatically whenever the aggregation value exceeds the threshold. The techniques of Chapter 5 are not directly applicable due to this difference: They can be used to verify if the aggregation value of a query exceeds the threshold, but not to determine when it crosses the threshold.

The algorithms for resource sharing presented in Chapter 5 are static schemes, i.e., they do not factor in the actual stream update and answer lookup rates. However, a truly optimal solution depends on these rates. For example, the “Materialize All” approach (described in Section 5.3) is optimal when the update rate is 0 and the lookup rate is high, while the “Materialize None” approach (again, described in Section 5.3) is optimal when the lookup rate is 0 and the update rate is high. One interesting direction for future work is to modify the algorithms of Chapter 5, making them more responsive to actual update and lookup rates.
Bibliography


