Portland State University
PDXScholar

Dissertations and Theses Dissertations and Theses

1-1-2011

Low-latency Estimates for Window-Aggregate Queries over Data Streams

Amit Bhat
Portland State University

Let us know how access to this document benefits you.
Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Recommended Citation
Bhat, Amit, "Low-latency Estimates for Window-Aggregate Queries over Data Streams" (2011). *Dissertations and Theses.* Paper 161.

10.15760/etd.161

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

المنارة للاستشارات

www.manaraa.com

Low-latency Estimates for Window-Aggregate Queries over Data Streams

by

Amit Bhat

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
David Maier, Chair
Leonard Shapiro
Kristin Tufte

Portland State University
© 2011

المنارة للاستشارات

www.manaraa.com

ABSTRACT

Obtaining low-latency results from window-aggregate queries can be critical to certain data-stream processing applications. Due to a DSMS's lack of control over incoming data (typically, because of delays and bursts in data arrival), timely results for a window-aggregate query over a data stream cannot be obtained with guarantees about the results' accuracy. In this thesis, I propose a technique, which I term *prodding*, to obtain early result estimates for window-aggregate queries over data streams. The early estimates are obtained in addition to the regular query results. The proposed technique aims to maximize the contribution to a result-estimate computation from all the stateful operators across a multi-level query plan. I evaluate the benefits of prodding using real-world and generated data streams having different patterns in data arrival and data values. I conclude that, in various DSMS applications, prodding can generate low-latency estimates to window-aggregate query results. The main factors affecting the degree of inaccuracy in such estimates are: the aggregate function used in a query, the patterns in arrivals and values of stream data, and the aggressiveness of demanding the estimates. The utility of the estimates obtained using prodding should be optimized by tuning the aggressiveness in result-estimate demands to the specific latency and accuracy needs of a business, considering any available knowledge about patterns in the incoming data.

المنارة للاستشارات

www.manaraa.com

DEDICATION

To my mother and father.

المنارة للاستشارات

www.manaraa.com

ACKNOWLEDGMENTS

I am grateful to my adviser, Dr. David Maier, for his guidance and encouragement throughout the work on this thesis. After each meeting with Dave, I walked away with a combination of ideas and motivation to keep me going till we met next. I also thank my thesis committee members Dr. Leonard Shapiro and Dr. Kristin Tufte for reviewing and giving feedback on my work.

I was fortunate to get proactive help and guidance from Rafael Fernandez at every level of this work. This thesis would not have been possible without Rafael's support.

Brainstorming on many implementation ideas with the streams research group (the *Latte* group) at Portland State University (PSU) played an important role in this work. The Latte group also trained me in presentation, especially before my mid-way thesis talk. I also found greatly helpful the feedback I received, from many of my fellow graduate students at PSU, on certain aspects of my presentation of this work.

I thank everyone who helped me with the logistics — especially, Dave for providing me a computer and a cubicle, James Whiteneck for making available traffic data required in part of my experiments, and Dr. Andrew Black for lending me a monitor.

Last but not the least, I thank all my friends who frequently encouraged me by telling me that I was doing "something important".

www.manaraa.com

CONTENTS

Abstract . . . . **i**

Dedication . . . . **ii**

Acknowledgments . . . . **iii**

List of Tables . . . . **vi**

List of Figures . . . . **vii**

1 Introduction **1**
1.1 Data Stream Processing 1
1.2 Unbounded Data Arrivals and Window Queries 3
1.3 Punctuations 5
1.4 Result Latency for Window Queries 7
1.4.1 Challenges in Getting Timely Results 9
1.4.2 Desire for Early Results 10
1.5 Thesis Contributions 11

2 The NiagaraST Data Stream Management System **14**
2.1 NiagaraST: Technical Overview 14
2.2 Window Aggregates 17
2.3 Paned Implementation 20

3 The Prodding Technique **24**
3.1 The Prodding Technique 24
3.2 Sources of Prods 29
3.3 Prodding a Query Plan 30
3.4 Accuracy of Early Results 38

المنارة للاستشارات

www.manaraa.com

4 Evaluation **41**
4.1 Environment and Instrumentation 42
4.2 Overheads in Prodding 43
4.3 Latency Gains 44
4.3.1 Latency Due to Data Delivery Delays 44
4.3.2 Latency Due to Bursts in Data 48
4.4 Prodding Aggressiveness 52
4.4.1 Uniform Distribution 54
4.4.2 Normal Distribution 55
4.5 Summary 60

5 Related Work **61**
5.1 Stream-Relational and Stream-Warehousing Systems 61
5.2 Systems Supporting Retractions or Negative Tuples 63
5.3 Load Shedding 64
5.4 Online Aggregation 65

6 Conclusion and Future Work **67**
6.1 Challenges 67
6.2 Future Work 70
6.3 Concluding Remarks 74

References **75**

www.manaraa.com

LIST OF TABLES

4.1 Overheads of Prodding: Comparison of Result Latencies with and without Prodding over 3 Different Data Streams 43
4.2 Latency Gains over Delayed Data Streams for Various Lengths . . . 47
4.3 Latency Gains over Bursty Data Substreams with Various Burst Lengths 52

www.manaraa.com

LIST OF FIGURES

1.1 Data processing models 2
1.2 Window Queries 4
1.3 Punctuated Data Stream 6
1.4 Tuple and Punctuation Arrival Patterns 8
1.5 Bursts in Data in a Wireless Network 10

2.1 NiagaraST operator – a logical view 14
2.2 Main objects in NiagaraST's inter-operator communication 15
2.3 Window-Sum query plan in NiagaraST 18
2.4 60 second windows subdivided into 20 second panes 21
2.5 NiagaraST query plan for paned implementation of Window-Sum . 22

3.1 Prodding a Window-Sum Query 26
3.2 Sources of Prods 29
3.3 Refresh GUI for the Latest Result Estimates 31
3.4 Prod Processing in the Paned Implementation of Window-Max . . . 33

4.1 Prodding a Sliding Window with Different Degrees of Prodding Aggressiveness (P_{Aggr}) 44
4.2 Latency Gains and Accuracy Curves for Q4-1 over Delayed Data Prodded at 90 Seconds in a 2-Minute Slide 45
4.3 Prodder and Expensive Operator in Burst Experiment Query Plan . 48
4.4 Data Arrival Bursts in a Wireless Network 50
4.5 Q4-2 over Bursty Data Prodded at 1 Second in a 1-Second Slide . . 51
4.6 Effects of Prodding Aggressiveness on Average Early-result Accuracy and Latency Gains over a Stream with Uniform Data Values . 54
4.7 Effects of Prodding Aggressiveness on Average Early-result Accuracy and Latency Gains over Stream Having Normally Distributed Data Values with Mean=500 and SD=100 55

www.manaraa.com

4.8 Effects of Prodding Aggressiveness on Average Early-result Accuracy over Stream Having Normally Distributed Data Values with Mean=500 and SD=400 57
4.9 Effects of Prodding Aggressiveness on Average Early-result Accuracy over Stream Having Normally Distributed Data Values with Mean=500 and SD=1000 57
4.10 Worst-case Accuracies for Early Results for 30-second Sliding Windows over Data Values with SD=400 and Mean=500 58
4.11 Worst-case Accuracies for Early Results over 30-Second Tumbling Windows over Data Values with SD=400 and Mean=500 58

6.1 Operator-level Prodding 67
6.2 Sidelining Portions of Bursty Data 71
6.3 Burst Processing Strategies 71
6.4 Monitoring Running Accuracy 73

المنارة للاستشارات
www.manaraa.com

Chapter 1

INTRODUCTION

When users base their time-critical decisions on results of queries in a data stream processing application, producing timely responses for such queries becomes essential. In Sections 1.1 to 1.4 of this chapter, I examine the issues and the techniques that relate to latency in computing window-query results over data streams. In Section 1.5, I introduce my proposed technique for getting low-latency results for window-aggregate queries, along with the summary of contributions of this thesis.

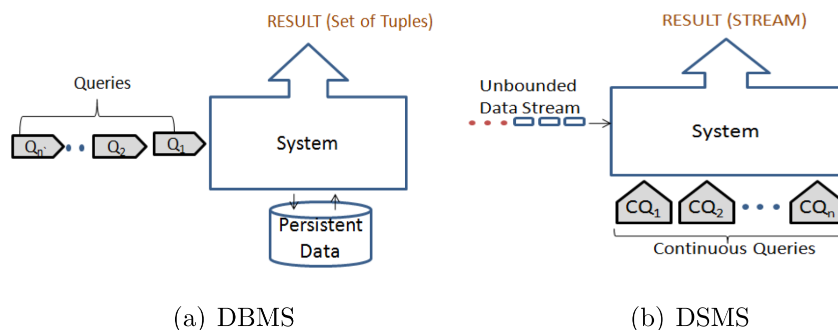
1.1 DATA STREAM PROCESSING

A wide range of applications that have been developed in the past three decades use *database management systems* (DBMSs) for data processing. In a DBMS (Figure 1.1(a)), data are stored on persistent storage devices [21]. Whenever a user application needs to extract some information, the application submits one or more queries to the DBMS. To process these queries, the system fetches the required data from persistent storage to memory, processes the data as dictated by the queries, and produces results as a set of data tuples. For example, a stock trader interested in obtaining market trends over a month will use a DBMS query against stored market data for the month.

To facilitate rapid data lookup, DBMSs associate indexes with the data, or cache pre-computed query results called materialized views. Furthermore, to

المنارة للاستشارات
www.manaraa.com

achieve faster query executions, DBMSs also perform optimizations by selecting the most efficient among alternative query plans to evaluate a given query [22].



(a) DBMS (b) DSMS

Figure 1.1: Data processing models

A class of applications (hereafter, *stream applications*), which has emerged in the past ten years, involves continuous processing of incoming data feeds that are together termed a *data stream*. Typically, in stream applications, data processing is followed by some time-critical decisions by the application users. For example, a stock trader interested in making immediate sale or purchase decisions can use stream applications to continuously analyze market data.

Traditional DBMSs are inadequate to fulfill data processing needs of stream applications [5]. First, the latency introduced by the DBMS data-processing model — which involves storage, loading, and indexing — is unacceptable. Second, using materialized views of data to quickly answer queries is not helpful, because, in most stream applications, including the most recently arrived data in the query result calculations is crucial to the subsequent decision making. For example, in a temperature-monitoring application, the latest readings of temperatures reported by the sensors must be included in the query processing to ensure proper subsequent re-adjustments in cooler settings. A materialized view, which is a pre-computed query result, will not include the very latest readings. Finally, DBMS query-optimization goals, such as minimizing I/O or disk accesses, may not be

المنارة للاستشارات
www.manaraa.com

suitable for processing data streams.

The systems that have been developed to enable stream applications to process data streams are called *data stream management systems* (DSMSs) (Figure 1.1(b)). In contrast to the *ad hoc* queries required by the DBMS applications, stream applications need queries to run continuously over unbounded data streams. Therefore, the DSMS queries are sometimes called *continuous queries* (CQs). The output of a CQ is a new data stream that contains results of query computations.

1.2 UNBOUNDED DATA ARRIVALS AND WINDOW QUERIES

In theory, data feeds never cease to arrive at a DSMS. Such potential unending data arrival is problematic for two reasons. First, an unending input data stream can cause a *stateful operator* (e.g., Join, DupElim) in a CQ to keep accumulating state and eventually run out of memory. The DupElim operator, for example, produces output by eliminating duplicate elements from its input. This operator would need to maintain in its state each unique data-stream element. Such unbounded state maintenance may cause the DSMS to run out of memory.

Second, a *blocking operator* (e.g., aggregate operators such as SUM, COUNT, MAX) cannot produce output until it sees the entire input data. For example, a CQ such as "Count the number of transactions for the stock symbol IBM" cannot produce the correct transaction count until it has seen all the data feeds (at least for IBM).

To overcome the aforementioned problems arising from the potential unending nature of data streams, window CQs fragment an unbounded data stream into bounded groups of tuples called *windows* [14, 18]. Grouping of tuples into windows is usually based on a progressing data attribute such as timestamp or tuple-count. Window CQs produce results for each window.

Typically, a DSMS user customizes windows in a CQ by specifying parameters

المنارة للاستشارات
www.manaraa.com

such as the window duration (also called the window *range*) and frequency for creating a new window (also called the window *slide*). Such parameterized windowing is apt for stream applications, because these applications usually involve analyzing recent events over specific time spans — e.g., "temperature readings over the last few hours", "user clicks during the last one minute", and "traffic speed readings over the last 30 minutes". Figure 1.2 shows examples of window queries.

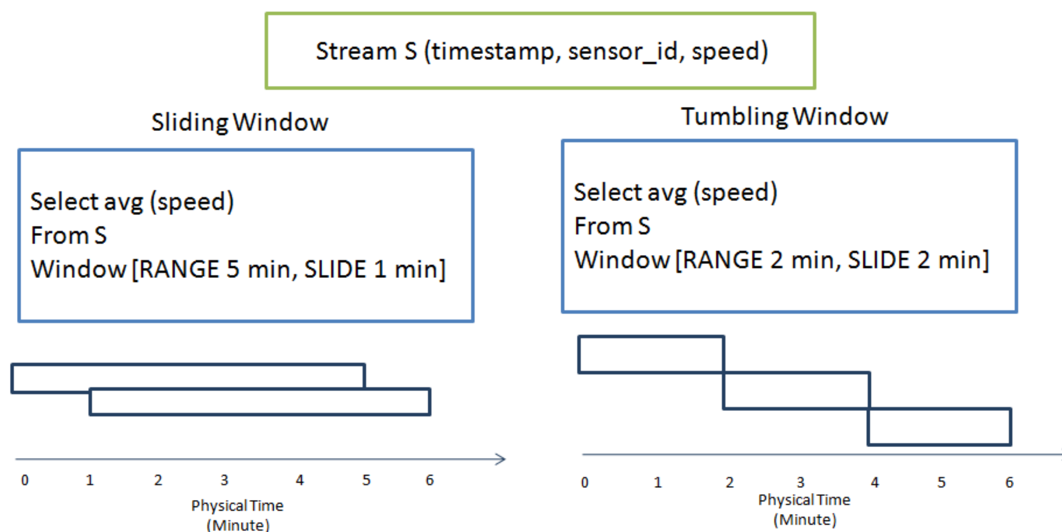


Figure 1.2: Window Queries

A query window that is specified with the three parameters RANGE, SLIDE and WINDOWING-ATTRIBUTE (e.g., timestamp, tuple-count) is called a *sliding window*, and a query that involves a sliding window specification is called a *sliding-window query*. See Figure 1.2. The sliding-window query Q1-1 asks a DSMS to compute the average of vehicle speed readings from a traffic data stream S over the most recent 5 minutes, and update the result every 1 minute (by considering the new *current* 5-minute window). Notice the WINDOW clause in Q1; the parameter RANGE specifies the span for a window, and the parameter SLIDE specifies the window-advancement stride. The WINDOWING-ATTRIBUTE is not explicit in query Q1-1; the unit "min" for RANGE and SLIDE hints a DSMS

المنارة للاستشارات
www.manaraa.com

that window range and slide are defined over timestamp. A DSMS may require that a WINDOWING-ATTRIBUTE is explicitly specified in a query. For a given WINDOWING-ATTRIBUTE, each window can be uniquely specified by the pair *window start time - window end time*. For example, Figure 1.2 depicts two consecutive sliding windows — Window 0-5 and Window 1-6 — corresponding to query Q1-1. (Another way to uniquely specify a window for a given WINDOWING-ATTRIBUTE is to use window start time and RANGE, which together determine window end time.)

A special case of sliding windows is *tumbling windows*, in which the parameters SLIDE and RANGE have equal values. Query Q1-2 in Figure 1.2 is an example of a tumbling window query. Note that all the tumbling windows are disjoint; e.g., Window 0-2, Window 2-4, and Window 4-6 shown in Figure 1.2 are all disjoint.

1.3 PUNCTUATIONS

Despite using window queries to fragment a potentially unbounded data stream, DSMSs face a challenge in computing query results over windows. It is difficult for a DSMS to ensure that all data that belong to a particular window have arrived at the system. The difficulty arises due to the following observed phenomena pertaining to stream data, which typically travel across networks. First, data in stream applications may arrive at a DSMS out of order of their generation timestamps. Such disorder in data occurs due to various reasons such as data rerouting across a network, combining data from sources that are skewed in data generation time, combining data that are delayed by different amounts, or disordered output from some query operators (e.g., MULTI-JOIN [10]). Second, due to inherent delays in networks, it is difficult to distinguish a *lull* (i.e., period of no input) in a data stream from delays in data arrival. Third, multiple tuple attributes (e.g., start_time and end_time in a Netflow record) may qualify to be a timestamp. Data may be generated at a source in order according to one such attribute, while a query uses

المنارة للاستشارات
www.manaraa.com

another qualifying attribute as a timestamp, on which data may be out of order.

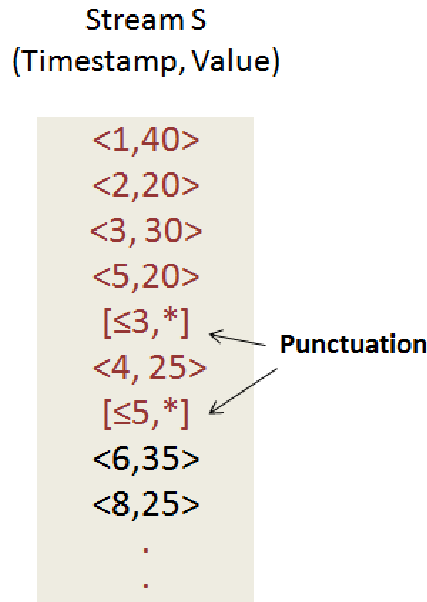


Figure 1.3: Punctuated Data Stream

To overcome the aforementioned difficulty, several DSMSs use *punctuations* [4, 13, 23, 28]. A *punctuation* is a special tuple in a data stream that marks the end of a substream. The substream corresponding to a punctuation is identified by predicate(s) and pattern(s) contained in the punctuation. Figure 1.3 depicts a *punctuated data stream* S. S has schema (Timestamp, Value), and has tuples ⟨Timestamp=t,Value=v⟩, or, for short, ⟨t,v⟩. Notice the punctuations, which are of the form [Predicate:(Timestamp≤t), Pattern:(Value=*)], or for short, [≤t,*]. A punctuation [≤t,*] informs the DSMS that tuples with Timestamps less than or equal to t and with any (indicated by the regular expression "*") Value will no longer arrive at the system. Suppose a user registers a window CQ with RANGE=5 and SLIDE=1. When the DSMS sees the punctuation [≤5,*], the system has a guarantee that all the tuples belonging to Window 0-5 (and also, any other window that ends at a Timestamp less than 5) have been seen. Therefore, the system can

المنارة للاستشارات
www.manaraa.com

close Window 0-5 (and any other open window that ends at a Timestamp less than 5) by producing the results, and removing corresponding state for the window. Tucker, et al. [28] comprehensively discuss punctuation semantics.

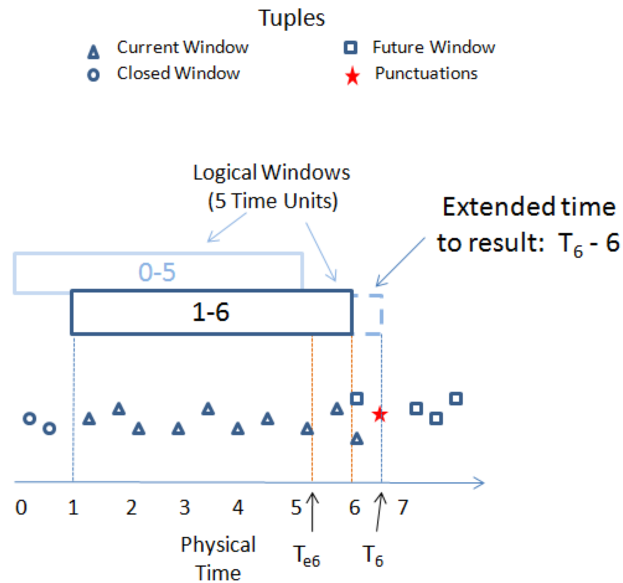
Note that punctuation mechanisms vary slightly from one system to another. Also, punctuations are known by different names in different systems; e.g., CTIs in StreamInsight [4], and Heartbeats in Gigascope [13] and STREAM [23]. Further, there are other DSMSs, such as Aurora [2], that do not use punctuations. These systems either expect data to arrive in order, or sort data after they arrive at the DSMS. Such systems, in case of order-sensitive operators, require an additional argument, such as a *slack*, which indicates the degree of possible disorder in data.

1.4 RESULT LATENCY FOR WINDOW QUERIES

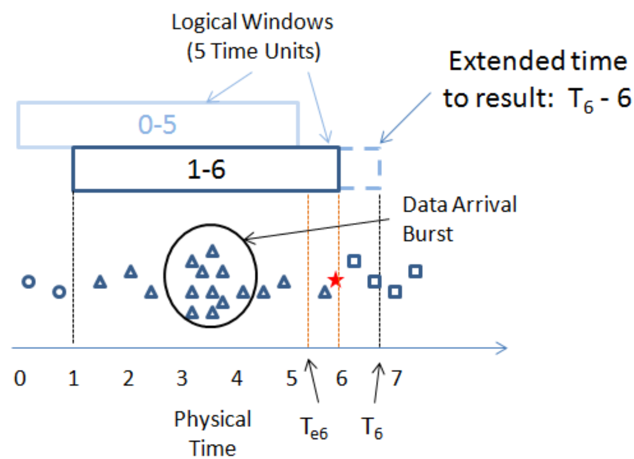
A typical data-stream tuple contains a *timestamp* (which indicates the tuple's generation time at its source, also called tuple's *application time*) and a *payload* (which is the tuple's data, and can be comprised of multiple fields). The length of time between the application time of the oldest tuple belonging to a window and the time of the result production for the window is the *result latency* for the window.

Another important notion related to time in stream applications is that of *system time*, which represents the DSMS clock. I say that a query result for a window is obtained in *timely* manner if the system time at which a DSMS generates the result is the same as the window end time (after considering possible time-zone skews between the data-stream source and the DSMS). A query result for a window that is produced by a DSMS before the window end time is an *early* result. The result that is produced by a DSMS on receiving an appropriate punctuation is the *final* result. In this thesis, I denote application time with *"t"* and system time with *"T"*.

المنارة للاستشارات
www.manaraa.com



(a) Delays in Data Arrival



(b) Bursts in Data Arrival

Figure 1.4: Tuple and Punctuation Arrival Patterns

I discuss the challenges in obtaining timely results for window CQs in Subsection 1.4.1. Then, in Subsection 1.4.2, I describe why early results to window queries are desirable in DSMS applications.

المنارة للاستشارات
www.manaraa.com

1.4.1 Challenges in Getting Timely Results

Despite using punctuations to close windows, DSMSs do not have control over the latency in producing window query results, for two reasons. One reason is that tuples or punctuations may arrive late at the system. Figure 1.4(a) depicts a snapshot of a sliding window (RANGE=5, SLIDE=1), and the corresponding data arrivals at a DSMS. Tuples belonging to Window 1-6 arrive even after physical time 6 has elapsed. The punctuation closing this window arrives at a physical time T_6, which lies between 6 and 7. Thus, the DSMS cannot produce a query result for Window 1-6 before time T_6, which means the result for Window 1-6 is not produced in timely manner. In other words, the logical boundary for Window 1-6 is extended in physical time from 6 to T_6 (shown with dashed lines in the Figure 1.4(a)).

The other reason for high latency is arrival of data in high-volume bursts. Such bursts of data can overwhelm a DSMS's processing capacity temporarily, thereby causing delays in computation of window query results. Delays are possible despite arrival of tuples and punctuations within the appropriate physical time boundaries. Figure 1.4(b) depicts this situation. A high-volume data burst is symbolized by a higher density of tuples (triangles) between physical times 3 and 4.

Figure 1.5 shows graphically a sample bursty arrival pattern in data consisting of the records collected at wireless access points in the Dartmouth college campus [15]. The graph in the figure shows the tuple count in each time bucket of 1 second. Notice that some of these buckets, e.g., 30, 53, 54, 62, see a high count (400 to 800) of tuple arrivals, whereas a majority of the other buckets see a moderate tuple count (0 to 300). A high-density portion (or simply, a burst) of data arrival will cause a DSMS to see a temporary spike in the load of tuples to be processed. A higher processing load may lead to longer processing time, and subsequently, to a high result latency. Therefore, when a data stream with variations in data arrival densities is input to a DSMS window query, I expect to see the corresponding variations in the result latency for different windows: higher latency for windows

المنارة للاستشارات
www.manaraa.com

that encompass high data-arrival counts, and lower latency for the other windows.

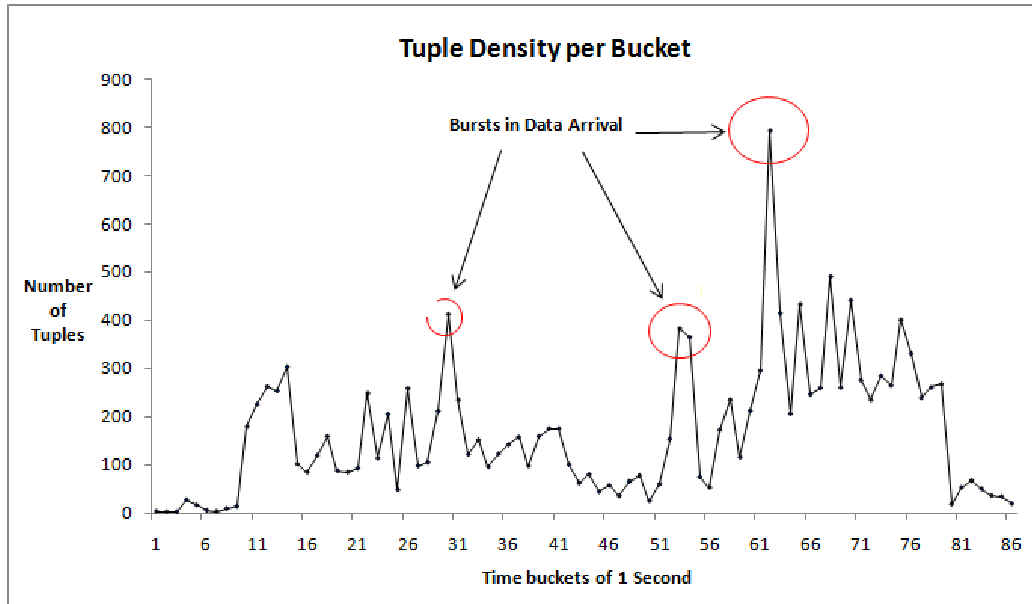


Figure 1.5: Bursts in Data in a Wireless Network

1.4.2 Desire for Early Results

Applications using a DSMS may involve time-critical decision making [4, 24]. Therefore, such applications would need low-latency responses for CQs. Examples of such applications include network traffic monitoring, financial analysis, sensor networks, and industrial process monitoring.

Thus, for a CQ over Window 1-6 in Figure 1.4, application users would like to see the results close to time 6. Or, to gain advantage by getting *early results*, users might desire to see the query results for window 1-6 early, say at T_{e6}, which is a time earlier than 6. For example, in stock trading, early query results might mean early detection of market trends. An example of a window query that tracks stock market trends is: "Report the stock symbols for which the difference between high price and low price in a 5-minute window is greater than 10".

Latency-accuracy tradeoff: An early window query result that is produced —

المنارة للاستشارات
www.manaraa.com

despite the challenges described in Section 1.4.1 — to satisfy the stream application needs may be inaccurate, since a DSMS may need to compute such an early result for a window without considering all the tuples that belong to the window. Furthermore, the lower the desired latency in an early result for a window, (potentially) the lower the portion of data (belonging to the window) that is considered for the result computation, and hence, the higher the potential inaccuracy in the early result. Moreover, the tolerable degree of such inaccuracy (incurred in exchange of low result latency) might vary from one stream application to another. Therefore, it would be desirable to enable stream applications to control the tradeoff between the latency gains in early window query results and the accuracy of such results.

1.5 THESIS CONTRIBUTIONS

A technique to provide DSMSs control over the latency-accuracy tradeoff in window-aggregate query results is the core contribution of this thesis. This technique, which I call *prodding*, involves injection of special tuples in a data stream. I call these injected tuples *prods*. A prod describes a subset of the data stream (or simply, a substream). A DSMS operator interprets a prod as a demand for an estimate of the query result over the substream described by the prod.

The work in this thesis consists of the following four components:

1. The prodding technique

   Prodding deals with the latency issues discussed in Section 1.4 by allowing a DSMS to produce *early estimates* of window query results. Such an estimate is generated by a DSMS when it encounters a prod in a data stream. A system enabled with prodding still produces a *final* window query result on processing the appropriate punctuation.

2. Semantics for prods

المنارة للاستشارات
www.manaraa.com

I describe how prods are processed at different operators in a window-aggregate query plan. More specifically, I explain how prods are interpreted, translated, and propagated by a windowing operator, and how prods affect *states* at an aggregation operator.

3. Implementation

   I have implemented a working prototype of prodding in NiagaraST DSMS. In this prototype, I enable NiagaraST's windowing and aggregate operators with prod processing.

4. Evaluation

   The usefulness of prodding is determined by latency gains achieved with early result estimates for window-aggregate queries, togetherwith the accuracy of these estimates. Result accuracy is not easy to define; the meaning of accuracy varies with data sets that are queried, and with aggregate functions in the queries.

   In my evaluation of prodding, I propose a broad definition for accuracy of a result estimate. I measure latency gains and accuracy drops in such estimates using the NiagaraST prodding prototype to process real-world and synthetic data streams. I study the effects of varying *aggressiveness in prodding* (i.e. how early a system asks a window for results) on the overall result latency gains and result accuracy, using input data streams having different patterns in data arrival and values.

   It is also important to ensure that the implementation of prodding does not impose significant overheads on a DSMS. I examine the possibility of any such overheads by comparing latencies of final query results obtained with and without use of prodding.

   Based on my observed experimental results for different combinations of data

المنارة للاستشارات
www.manaraa.com

sets and aggregate functions, I discuss the effectiveness of the prodding technique for real-world stream applications.

المنارة للاستشارات
www.manaraa.com

Chapter 2

THE NIAGARAST DATA STREAM MANAGEMENT SYSTEM

The NiagaraST data stream management system, developed at Portland State University (PSU), is a data-stream–processing derivative of Niagara [19], which is an Internet query engine developed jointly by the University of Wisconsin-Madison, Oregon Graduate Institute, and PSU. NiagaraST can process window queries over data streams, and can handle data that arrive out of order. I implement and evaluate a prototype of the prodding technique in the NiagaraST DSMS.

This chapter describes the technical features of NiagaraST that are relevant to the work in this thesis. Section 2.1 gives a technical overview of NiagaraST. Section 2.2 describes in detail the working of window-aggregates in NiagaraST. Section 2.3 describes the paned implementation of window-aggregates.

2.1 NIAGARAST: TECHNICAL OVERVIEW

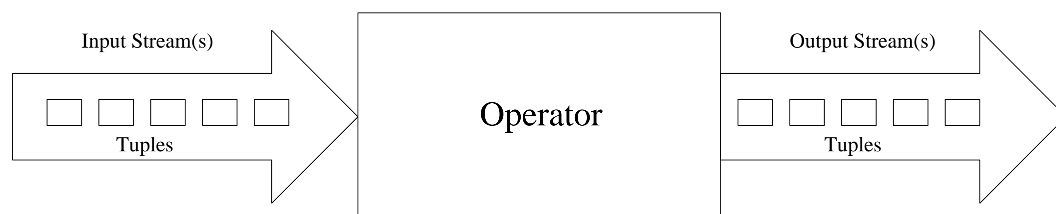


Figure 2.1: NiagaraST operator – a logical view

Figure 2.1 shows a high-level view of a typical NiagaraST operator (or simply an *operator*, for this discussion). An operator receives tuples from its input

www.manaraa.com

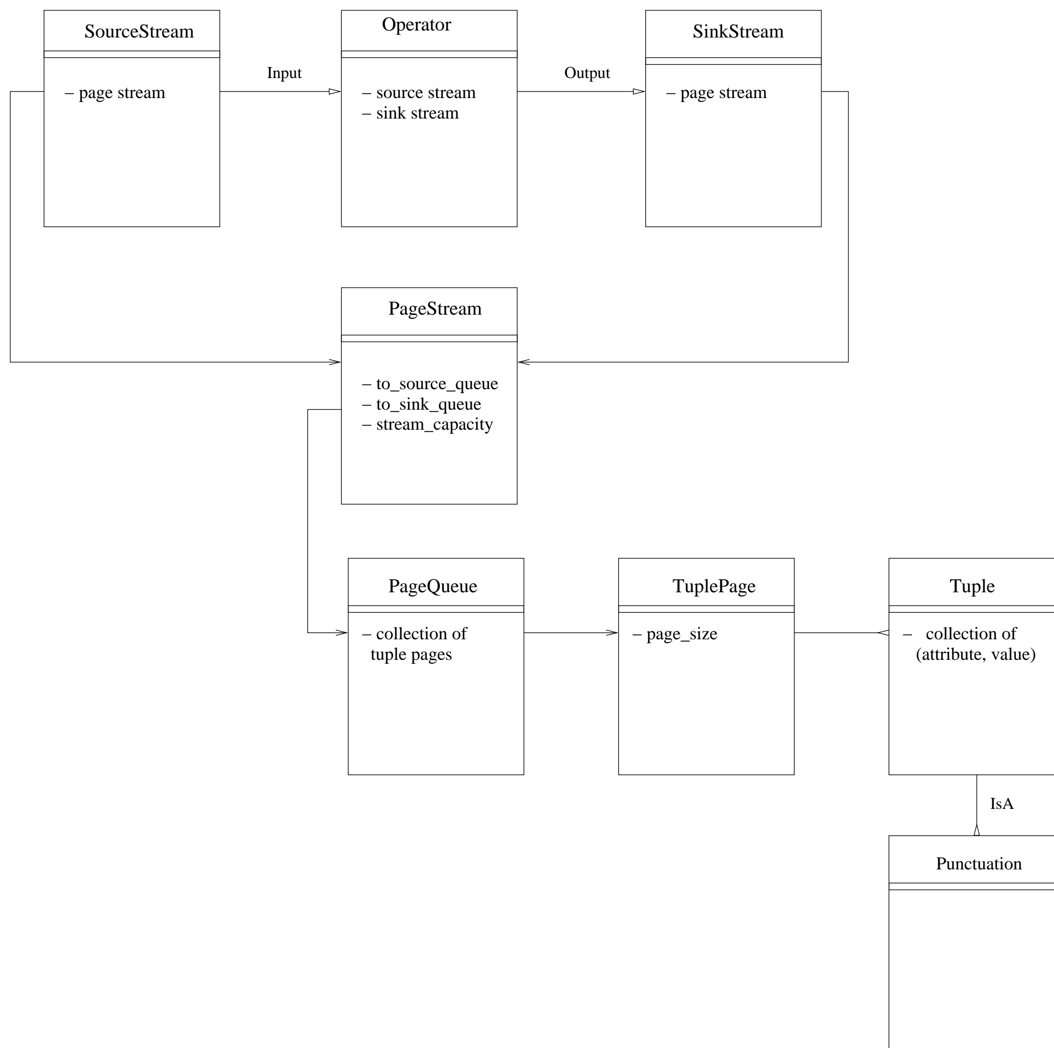


Figure 2.2: Main objects in NiagaraST's inter-operator communication

stream(s), processes the received tuples according to the operator's function (e.g., Sum, Count, Select), and sends the resulting tuples to its output stream.

A query plan in NiagaraST is a directed graph of operators. Typically, a plan will have a Scan operator at the bottom for data input, and a Construct operator at the top for formatting result output. I omit Scan and Construct operators from all the query plans depicted in this thesis, because these two operators do not affect query-execution logic. An example of a NiagaraST query plan is shown in Figure

www.manaraa.com

2.3. We shall examine this query plan in detail when I discuss the execution of query Q2-1 in Section 2.2.

Figure 2.2 shows a typical NiagaraST operator, and the objects that are involved in NiagaraST inter-operator communication. An operator has associated with it one or more *source streams*, which are its input streams, and *sink streams*, which are its output streams. A source stream for one operator may be a sink stream for some other operator.

Source and sink streams provide each operator a tuple-oriented interface for input and output. Underneath, the implementation passes pages of tuples between operators in a query plan. Such a page-oriented implementation avoids synchronizing operators at per-tuple granularity. The pages of tuples that are in flight between query operators are temporarily stored in the *inter-operator buffers*. An operator may also have local state to store intermediate results.

Both source and sink streams contain a *page stream* object, and each page stream object contains two *page queues* — a *to_source_queue* and a *to_sink_queue* — which serve as inter-operator buffers for downstream (towards a sink) and upstream (towards a source) tuples respectively. (Upstream transmission of tuples is useful to send interoperator feedback.) A page stream object also contains the parameter *stream_capacity*, which determines the number of *tuple-pages* in a page queue. A tuple-page is a sequence of tuples. The parameter *page_size* in a tuple-page object determines the maximum number of tuples that can fit per page.

A tuple object is a collection of attribute values. The attributes that belong to a tuple are determined by its schema. Each operator has one or more *input schemas*, plus an *output schema*. A punctuation is a special data-stream element whose values are predicates or patterns. For example, a tuple that has schema S(timestamp, sensor_id, speed) will have values such as ⟨10,2,55⟩ (which means timestamp=10, sensor_id=2, and speed=55). A punctuation with the same schema, S, will have values such as [≤10,2,*] (no more tuples with timestamps less than or equal to 10,

المنارة للاستشارات
www.manaraa.com

and sensor_id equal to 2 will be seen by the operator that has encountered this punctuation), or [≤10,*,*] (no more tuples with timestamps less than or equal to 10 will be seen by the operator).

2.2 WINDOW AGGREGATES

When the prodding technique is used to demand early result estimates from window-aggregate queries, two types of NiagaraST operators are mainly impacted — the Bucket operator and an Aggregate operator. Bucket performs windowing (i.e., mapping of each tuple to one or more windows) and an Aggregate operator, such as WindowSum or WindowMax, that follows Bucket (in a typical window-aggregate query plan) computes aggregates over windows [18].

Consider the sliding-window query Q2-1 (Figure 2.3), which computes sum of volumes for 60-second windows over a vehicle-traffic data stream S, and updates results every 20 seconds. Figure-2.3 also shows the NiagaraST query plan for Q2-1. The input data stream S has schema (timestamp, sensor_id, speed, volume) where timestamp is in seconds. The Bucket operator maps tuples to windows by mapping the tuple timestamps to *window-ids*. The range of windows to which a tuple belongs is determined by the two attributes *wid_from* and *wid_to*. The conversion from timestamp to window-id takes two simple calculations:

```
wid_from = timestamp / SLIDE,
wid_to = (timestamp + RANGE) / SLIDE − 1
```

where "/" is an integer division. For example, the tuple t1⟨211,1,54,25⟩ (Figure-2.3) gets wid_from $= 211/20 = 10$, and wid_to $= (211{+}60)/20 - 1 = 12$. Thus, t1 contributes to the sum for windows (or more precisely, windows with window-ids) 10, 11, and 12. As another example, tuple t4, which has timestamp 230, belongs to (and therefore, contributes to the sum calculation for) windows 11, 12, and 13. Bucket processes the timestamps in punctuations slightly differently. Wid_from is calculated as *timestamp/SLIDE − 1*, and wid_to is set to a "*". For example,

المنارة للاستشارات
www.manaraa.com

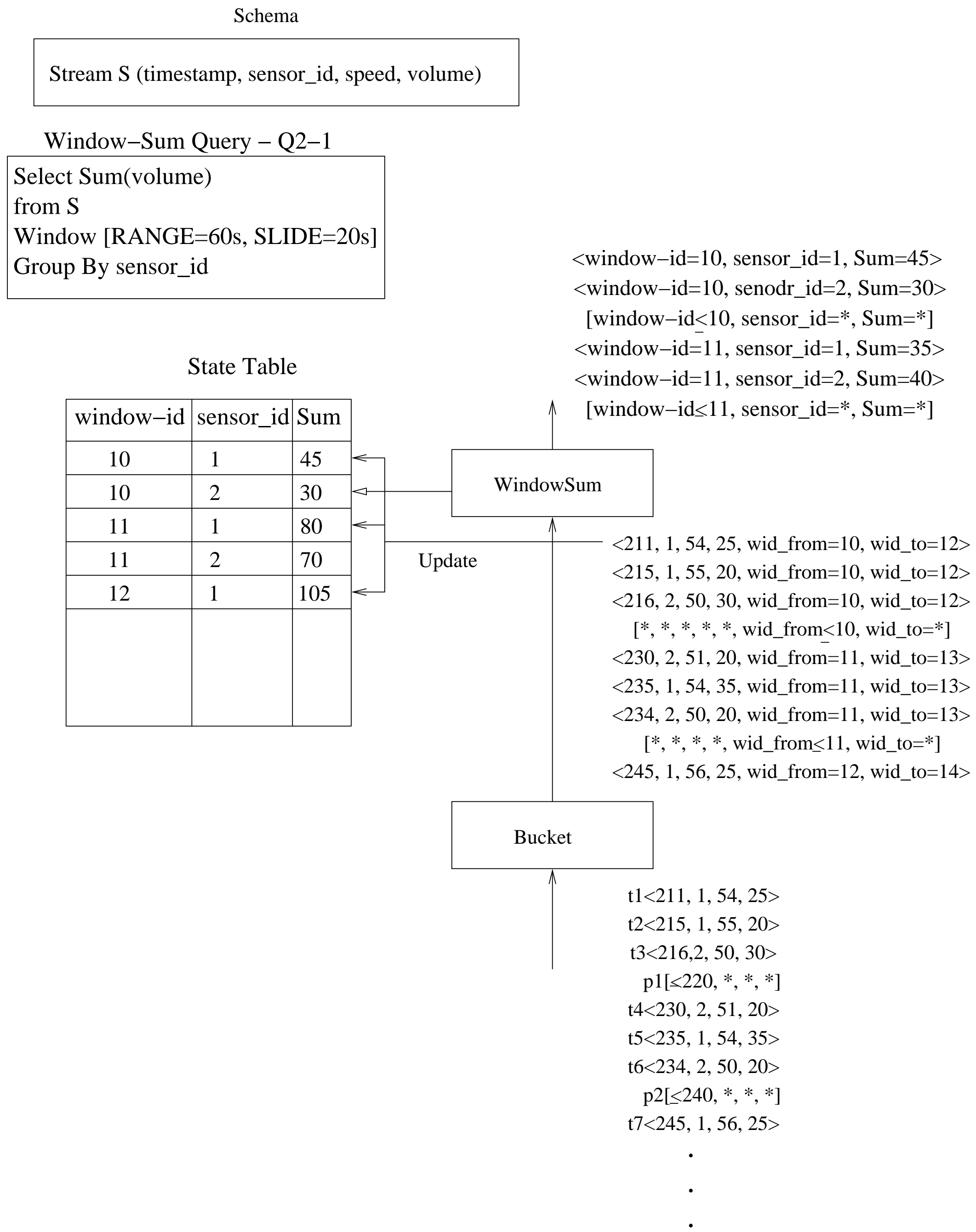


Figure 2.3: Window-Sum query plan in NiagaraST

Bucket translates the punctuation [timestamp≤240,*,*,*] into the punctuation [*,*,*,*,wid_from≤11,wid_to=*]. Such translation of timestamps to window-ids

www.manaraa.com

by Bucket ensures that the subsequent window-aggregate operator correctly closes windows. As with tuples, Bucket leaves the *non-timestamp* attributes in a punctuation untouched. Group-wise punctuations on timestamps are translated by Bucket to the group-wise punctuations on window-ids. For example., if the Bucket operator in Figure 2.3 encounters a punctuation [timestamp≤240,sensor_id=1,*,*,*], it would translate this punctuation to [*,sensor_id=1,*,*,wid_from≤11,wid_to=*], and send the translated punctuation to the subsequent Window-Sum. Such translation retains the punctuation predicate on group for use by the window-aggregate operator that follows Bucket in the query plan.

The window-aggregate operators in NiagaraST are *stateful*; they typically contain a *state table* (hereafter, just *state*) that holds a running aggregate per group per window. For each tuple that arrives at a window-aggregate operator, the window-aggregate updates the running aggregate values in its state for the tuple's group for all the windows that the tuple belongs to. For example, as shown in Figure-2.3, Window-Sum stores in its state group-wise *Sum* values for each sensor_id per window.

On receiving a punctuation, a window-aggregate produces aggregate results that are covered by that punctuation. For example, punctuation [*,*,*,*,wid_from ≤10,*], which indicates that the Window-Sum operator will encounter no more tuples with window-ids less than or equal to 10, allows Window-Sum to output the *Sum* for all the open windows that have ids less than or equal to 10. Further, Window-Sum purges from its state the aggregate values that the operator has output. In this thesis, I refer to a punctuation, such as [*,*,*,*,wid_from≤10,*], that allows a window-aggregate to output all the aggregate results for a window, as a *window-closing punctuation*. On receiving a group-wise punctuation, a window-aggregate outputs results, and purges state for the groups that are covered by the punctuation. For example, the Window-Sum operator shown in Figure 2.3, on receiving the punctuation [*,sensor_id=1,*,*,wid_from≤10,*], purges state and

المنارة للاستشارات
www.manaraa.com

emits results only for sensor_id=1 for all the open windows with ids less than or equal to 10.[1]

Grammatical data streams: Although punctuations allow the NiagaraST operators to handle out-of-order data, the operators expect their input punctuated data streams to be *grammatical*. A punctuated data stream S is grammatical if, in S, no tuple t that matches the predicate(s) of a punctuation p follows p [28]. Every NiagaraST operator guarantees that its output data stream is grammatical, given that the operator's input data stream(s) are grammatical.

2.3 PANED IMPLEMENTATION

A time-efficient (and in some specific cases, space-efficient) way to implement window-aggregates, as proposed by Li *et al.* [17], is to subdivide windows into *panes*, calculate aggregates per pane, and roll up the *pane-level aggregates* into window-aggregates. Figure-2.4 illustrates the *paned approach*.

Consider query Q2-2. With the paned approach, the DSMS subdivides each 60-second window into three 20-second panes, calculates the sum for each pane, and uses the pane-level result to update the sum for all the windows that the pane belongs to. For query Q2-2, each 20-second pane belongs to three consecutive windows, and therefore, each pane-level result updates the sum for three windows.

Figure 2.5 shows the NiagaraST query plan for computing the Window-Sum query Q2-2 using the paned implementation. Notice that panes are tumbling windows, therefore each tuple affects just a single pane. A pane-level result is emitted by the pane-aggregate operator when it receives a corresponding *pane-closing* punctuation. Only the pane-level results (rather than every tuple in the stream) update multiple window-level aggregates. As illustrated in Figure 2.5,

[1]NiagaraST generally expects punctuations to arrive often enough that there is one punctuation per window. Thus, each such punctuation generally causes output of results from one window, since earlier windows were covered by previous punctuations.

المنارة للاستشارات
www.manaraa.com

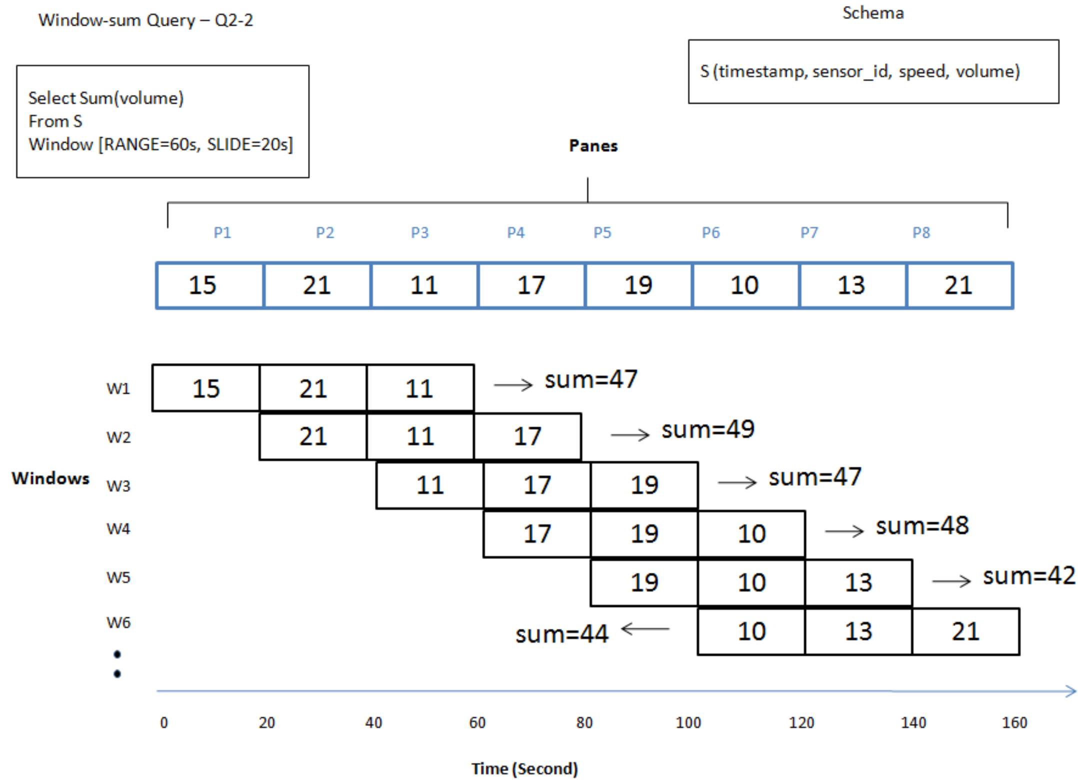


Figure 2.4: 60 second windows subdivided into 20 second panes

each tuple updates just one state entry at the Pane-Sum operator, while each pane-level result updates three entries in the window-level state. For example, the tuples ⟨211,1,54, 25⟩, ⟨215,1,55,20⟩, and ⟨216,2,50,30⟩ update Pane-Sum's single state-table entry that corresponds to the pane- id 10. Only the pane-level sum result for these tuples, i.e., the tuple ⟨pane-id=10, sum=75⟩, updates three state table entries in the Window-Sum operator. Such savings in the number of state updates per tuple result in improved execution time. Li *et al.* [17] demonstrate the performance gains obtained using the paned approach to implement window-aggregates. In their work, Li *et al.*, also describe the space benefits in the paned approach when this approach is used to compute *sub-aggregation* similar to that in Gigascope [6]. They observe that for sub-aggregation implemented using panes, the amount of buffering needed is significantly reduced.

المنارة للاستشارات
www.manaraa.com

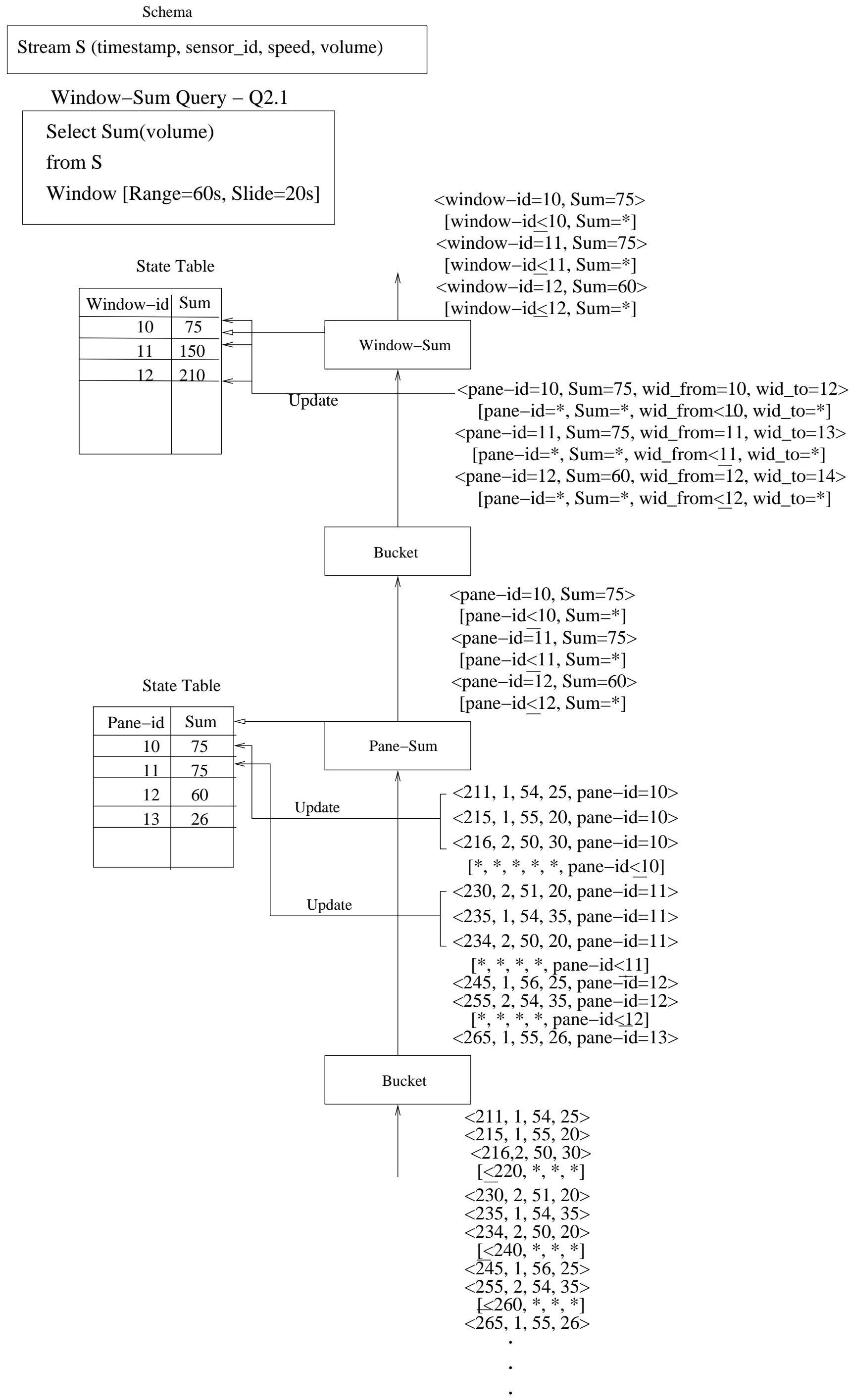


Figure 2.5: NiagaraST query plan for paned implementation of Window-Sum

www.manaraa.com

In this thesis, I shall use the paned implementation of window-aggregates as an example of a scenario wherein state is distributed across multiple operators in a query plan. The prodding technique becomes particularly conducive to such a scenario, because it becomes important to convey the urgency for a query-result estimate over a particular window to all the operators that can contribute to the estimate calculation. Such a result, whose production is triggered by prodding, is a *prodded* or *early* result. In Section 3.3, using a paned implementation example, I shall demonstrate how prodding synchronizes partial-result generation at multiple stateful operators in a query plan to maximize the contributions to a prodded result.

للاستشارات المنارة
www.manaraa.com

Chapter 3

THE PRODDING TECHNIQUE

Low-latency results for window-queries are important in various application domains. In the case of financial trading, early results may give a strategic advantage by allowing a user to quickly detect market trends. In the case of sensor networks, such as temperature-monitoring networks, delays in detecting high-temperature conditions may be costly (or sometimes fatal). In the case of traffic monitoring, early detection of traffic conditions such as congestion may allow controllers to take quick decisions such as rerouting the traffic.

In this chapter I describe my proposed technique, which I term *prodding*, to obtain such low-latency results. Prodding offers DSMSs control over the latency-accuracy balance in window-query results. I introduce the prodding technique in Section 3.1. In Section 3.2, I discuss various sources that can enable prodding. I describe the application of the prodding technique to a multi-level window-aggregate query plan in the Section 3.3. Finally, in Section 3.4, I discuss accuracy of the early results produced by prodding.

3.1 THE PRODDING TECHNIQUE

A *prod* is a special stream element that signals a DSMS to produce output (which may be an approximate result) that corresponds to a specific application time. To specify which application time, a prod contains at least one predicate on application time, although, a prod may also contain predicates or patterns on other tuple attributes. A typical prod would look like {Timestamp≤t1,*,*,*...}, where t1 is a

المنارة للاستشارات
www.manaraa.com

valid timestamp, and "*" matches any value.

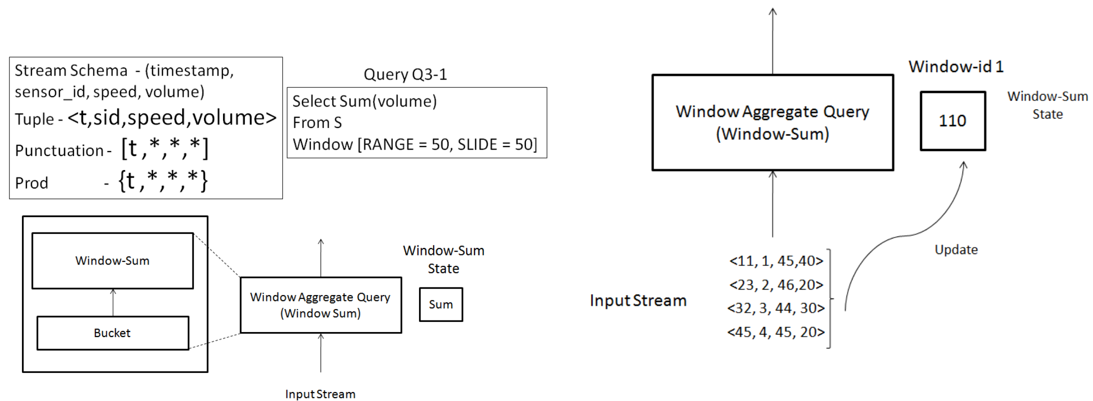
The results produced by a DSMS in response to a prod are *early* results. Such early results are a quick (and perhaps an incomplete) response produced by a DSMS despite any possible delays (described in Section 1.4.1) in producing *final* results, which are triggered by a punctuation. Early results may be produced with a low latency, but such results may be compromised in their accuracy. I shall discuss accuracy of early results in Section 3.4.

Although prods are structurally similar to punctuations — a prod triggers output over a substream that is specified by the prod's predicates or patterns — there are important differences between these two types of data-stream elements. First, prod processing may not (necessarily) result in purging of state at a (stateful) DSMS operator. And, second, tuples matching a prod's predicate may follow the prod in a data stream. Contrast such a data stream with the grammatical punctuated data streams described in the Section 2.2. A data stream that contains both prods and punctuations is a *prodded punctuated data stream*.

Consider the window-sum query Q3-1 (Figure 3.1(a)) over the prodded punctuated data stream S. Although I discuss prodding considering the NiagaraST-style use of a Bucket operator followed by a Window-Aggregate operator (as shown in Figure 3.1(a)), the proposed technique is applicable to window-query implementations in DSMSs other than NiagaraST where the bucketing functionality may be implemented inside window-aggregate operators. Figure 3.1(b) shows the query state after the query has processed the tuples ⟨11,1,45,40⟩, ⟨23,2,46,20⟩, ⟨32,3,44,30⟩, and ⟨45,4,45,20⟩ of S. The Window-Sum operator maintains in its state table the running sum for window id 1. Figure 3.1(c) shows the query state after the DSMS has processed the prod {50,*,*,*}. A Bucket operator processes a prod the same way as it does a punctuation. The Bucket operator in the query plan

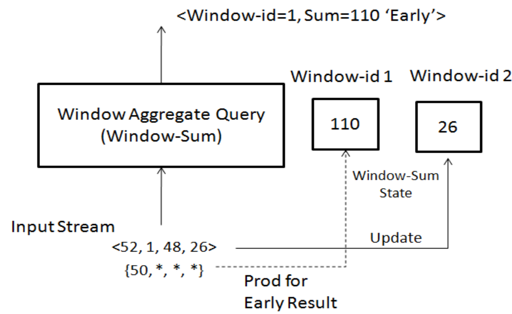
المنارة للاستشارات

www.manaraa.com

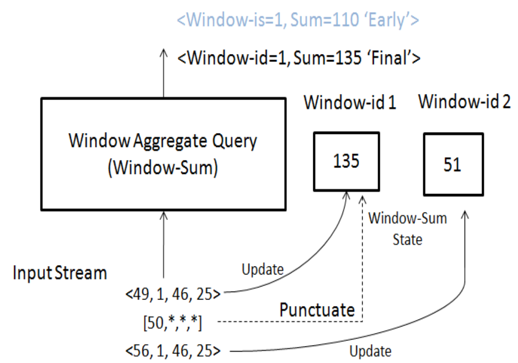


(a) Window-Sum Query Plan Main Operators: Bucket and Window-Sum

(b) Window-Sum Tuple Processing



(c) Window-Sum Prod Processing



(d) Window-Sum Punctuation Processing

Figure 3.1: Prodding a Window-Sum Query

translates the prod {50,*,*,*} into the prod {*,*,*,*,wid_from=1,wid_to=*} (remember the window id computation described in Section 2.2), and passes the translated prod to the Window-Sum operator. On receiving such a prod, the Window-Sum operator emits results corresponding to the predicates in the prod. The NiagaraST prodding prototype marks the result produced in response to a prod as 'Early'. As shown in Figure 3.1(c), on receiving the prod {*,*,*,*,wid_from=1, wid_to=*}, Window-Sum produces the result '110' for window-id 1. Window-Sum, after producing the corresponding results, outputs the prod without modifying it.

المنارة للاستشارات
www.manaraa.com

The DSMS forms a new window on arrival of the first tuple belonging to that new window. As shown in Figure 3.1(c), the tuple ⟨52,1,48,26⟩ causes the system to form Window 2.

On receiving further tuples that belong to Window 1, Window-Sum updates the state for window-id 1. Punctuations are processed as before: Bucket translates the punctuation [50,*,*,*] to the punctuation [*,*,*,*,wid_from=1,wid_to=*]. Window-Sum, on receiving a punctuation on window-id, releases results for the window-id indicated in the punctuation (and in case of group-wise punctuations, for the entries that also match the predicates on groups), and purges state entries for all the emitted results. The system marks result produced in response to a punctuation as 'Final'. Thus, as shown in Figure 3.1(d), the Window-Sum on receiving the punctuation [50,*,*,*], emits the final windowed-sum of 135 for window 1, and purges the corresponding entry from its state table.

Operator state: In general, a window-aggregate may or may not purge state on receiving a prod depending on where the operator is placed in a query plan. The rule to determine whether a window-aggregate should purge state entries corresponding to a prod is as follows:

```
State Purging Rule, R1:  If a window-aggregate is the topmost among
the stateful operators in a query plan, the operator does not purge
state entries corresponding to the prods that it receives.
Otherwise, the window-aggregate purges the state entries for any
prods that it receives.
```

In NiagaraST, as described in Chapter 2, the regular (i.e., non-paned) implementation of a window-aggregate query has a single stateful operator, namely, the window-aggregate operator. For example, in the query plan shown in Figure 3.1(a),

المنارة للاستشارات
www.manaraa.com

the Window-Sum is the only (and hence the topmost) stateful operator. Therefore, according to the rule R1, after emitting the early Sum of 110 for window id 1, Window-Sum does not purge this Sum (entry) from its state. The rationale for purging in rule R1 for the otherwise case — i.e., when the operator is not the topmost stateful operator in the query plan — will become clearer when I discuss prod processing in the paned implementation of window-aggregates (Section 3.3), wherein there exists a window-aggregate operator that is not the topmost stateful operator in the query plan.

Note that Rule R1 is applicable to the aggregate operators in the query plans that include regular or paned implementation of window-aggregates. R1 may not be more generally applicable to a query plan that includes more complex combinations of stateful operators — for example, query plans with more than two aggregates, and query plans that include other stateful operators such as Join. I discuss state purging in general query plans in Section 3.3.

Variety of prods: If a window query has a group-by clause, multiple early results (possibly, one per group) may be produced by the query after receiving a prod. Or, query operators may receive group-wise prods — for example, the data stream S in the following query Q3-2 may contain a prod {50,1,*,*}, which signals the DSMS to produce early results for the application time 50 and sensor_id 1.

```
Query3-2:  ''Select Sum(volume) from S Window [RANGE=50, SLIDE=50]
Group by sensor_id''.
```

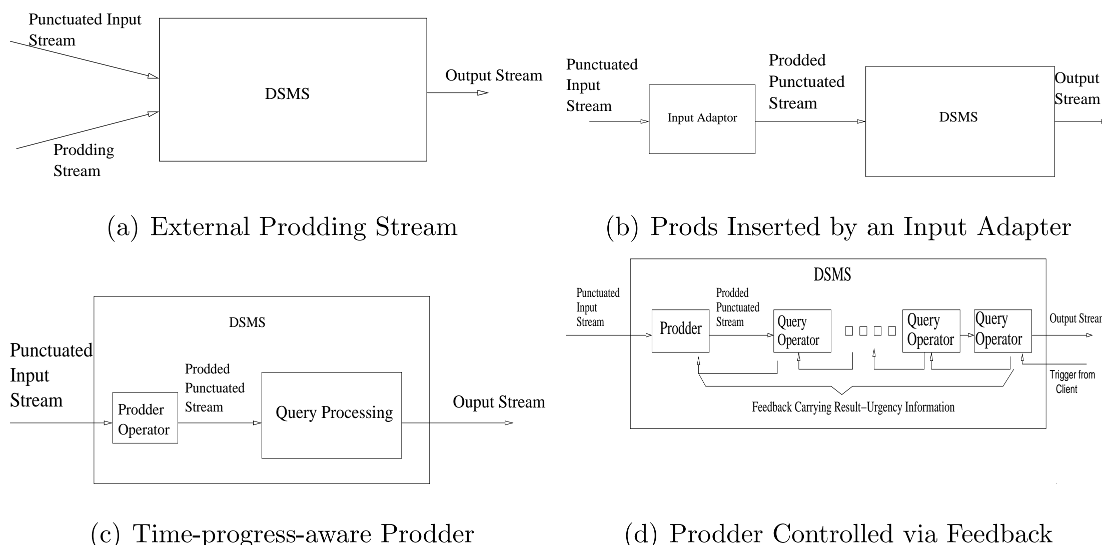
In theory, a prod, like a punctuation, may contain any combination of predicates or patterns; even the requirement to include a predicate on application time may be relaxed. Thus, a prod can be used to trigger query results over particular substreams. In this thesis, I shall mainly consider prods that have at least one application-timestamp predicate of the form "timestamp=t" or "timestamp≤t",

المنارة للاستشارات
www.manaraa.com

such that t is a valid value for time.

I call the technique of inserting prods in a data stream to signal a DSMS to produce results corresponding to the substreams described in the prods the *prodding technique*, or simply *prodding*.

3.2 SOURCES OF PRODS



(a) External Prodding Stream

(b) Prods Inserted by an Input Adapter

(c) Time-progress-aware Prodder

(d) Prodder Controlled via Feedback

Figure 3.2: Sources of Prods

The prodding technique is agnostic to the sources of prods. Figure 3.2 shows some possible ways for inserting prods in a data stream.

A source of prods can come from outside a DSMS: (1) A seperate stream, in addition to the input stream, may issue prods to a DSMS (Figure 3.2(a)) thereby controlling query-result generation over the input stream. This separate stream need not be a stream of prods; it can be a normal stream whose tuples are used to generate prods. (2) Commercial data stream management software, such as Microsoft's StreamInsight [26], use a special program called *input adapter* to read input streams from various external sources, and convert these streams into the

www.manaraa.com

format readable by the DSMS. Such an input adapter may be used to monitor time progress, and, after every advance by certain time period, insert prods in the input stream (Figure 3.2(b)).

Alternatively, the source of prods may be within a DSMS: (1) A DSMS query plan itself may include an operator to monitor time and insert prods in a data stream. I call such an operator a Prodder (Figure 3.2(c)). (2) The topmost operator in a query plan may have more information regarding query-progress needs. So, such an operator may pass this progress requirement to its upstream operators in the form of Feedback Punctuations [8] (Figure 3.2(d)). When the upstream-propagating feedback reaches the prodding operator (the Prodder) in the query plan, the Prodder inserts prods (having appropriate predicates) in the data stream. Such upstream feedback may be triggered by a DSMS user pressing a GUI button such as *Refresh*, which cause the DSMS to generate the latest estimates of results for currently open window(s) (Figure 3.3). Note that aggressively refreshing the GUI may cause the DSMS to prod the same window multiple times, each time generating more recent estimates for the window.

3.3 PRODDING A QUERY PLAN

Letting the prods flow together with the data stream allows prods to sweep before themselves partially computed results that are accumulated at operator states across a query plan. Such sweeping allows DSMSs to maximize the portion of data available in the system that will contribute to early result computation. In this section, I describe prodding a query plan that has state accumulated at more than one level in the plan. I use the paned implementation of a window-aggregate (described in Section 2.3) as an example of such a query plan.

Consider the Window-Max query Q3-3 over traffic data stream S shown in Figure 3.4(a). Figure 3.4(b) shows the state of execution of Q3-3 after the query has processed elements of S up to the tuple $\langle 125,2,46,45 \rangle$. Notice the paned-max values

المنارة للاستشارات
www.manaraa.com

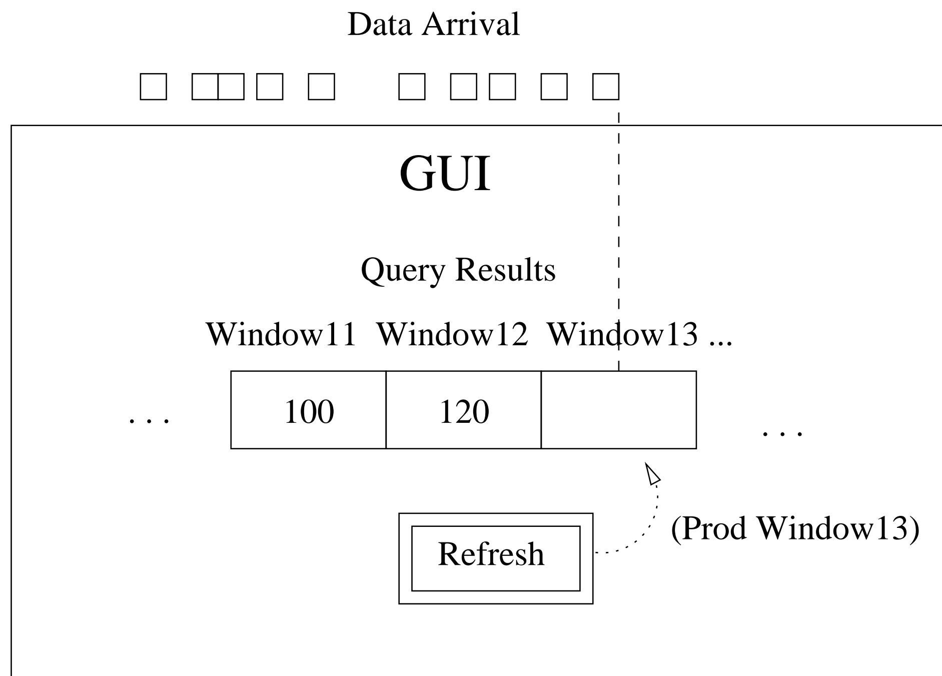


Figure 3.3: Refresh GUI for the Latest Result Estimates

accumulate in the state of the Pane-Max operator. At this stage in the execution, Pane-Max cannot release the paned-max values to the downstream operators, because these values may be updated by the subsequent tuples in S. The paned-max values can be released by Pane-Max only when it sees punctuations that close the panes. The state of the query plan after processing the punctuations [110,*,*,*] and [120,*,*,*] is shown in Figure 3.4(c). Bucket1 translates these punctuations on timestamps to punctuations on pane-ids, and the Pane-Max operator upon receiving the translated punctuations releases downstream the paned-max values for the panes covered by these punctuations. For example, the punctuation [120,*,*,*] is translated by Bucket1 into the punctuation [*,*,*,*,pid_from=11,pid_to=*], and the Pane-Max operator on receiving this translated punctuation sends to its downstream operator Bucket2 the paned-max results for the pane-id 11. On receiving punctuations [*,*,*,*,pid_from=10,pid_to=*] and [*,*,*,*,pid_from=11,pid_to=*], Pane-Max also purges the accumulated paned-max results for the panes 10 and

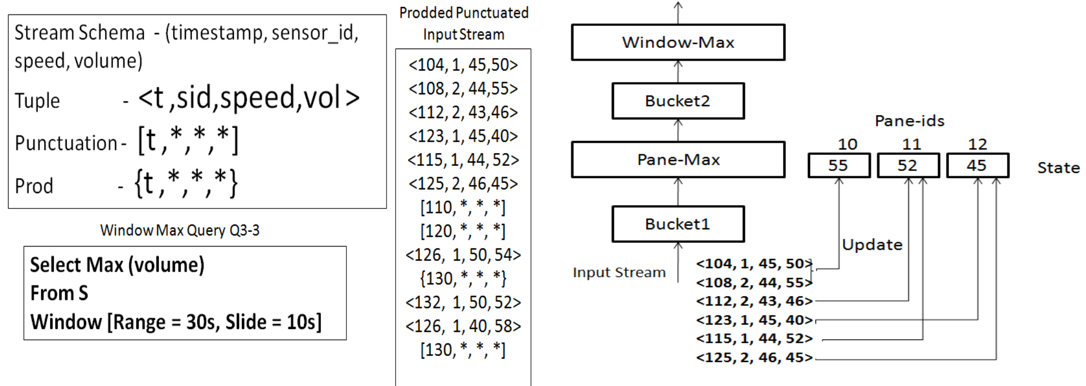
www.manaraa.com

11 from its state table (per Rule R1), and sends the punctuations downstream without modifying them.

Bucket2 assigns window-ids to the paned-max tuples that come out of the Pane-Max operator. For Q3-3, the RANGE of 30 seconds translates to a range over 3 panes, and the 10-second SLIDE becomes a slide by 1 pane. Thus, the range of window-ids affected by the pane-id 10 is given by wid_from=10 and wid_to=12. At the Window-Max operator, the paned-max 55 is used to update the windowed-max values held in the Window-Max state for the window-ids 10 to 12. Similarly, the paned-max of 52 from pane 11 updates the windowed-max values for the window-ids 11 to 13. Notice that because 52 is smaller than the previously computed windowed-max of 55 for the windows 11 and 12, the states for these two windows are unaffected by the update. Bucket2 translates the punctuations coming from Pane-Max that are on pane-ids to punctuations on window-ids. E.g., Bucket2 translates the punctuations [*,*,*,*,pid_from=10,pid_to=*] and [*,*,*,*,pid_from=11,pid_to=*] on pane-id to the window-id punctuations [*,*,*,*,*,*,wid_from=11,wid_to=*] and [*,*,*,*,*,*,wid_from=12,wid_to=*]. Window-Max, on receiving these punctuations, releases their corresponding windowed-max values (55 in each case here), and purges the state for windows 10 and 11.

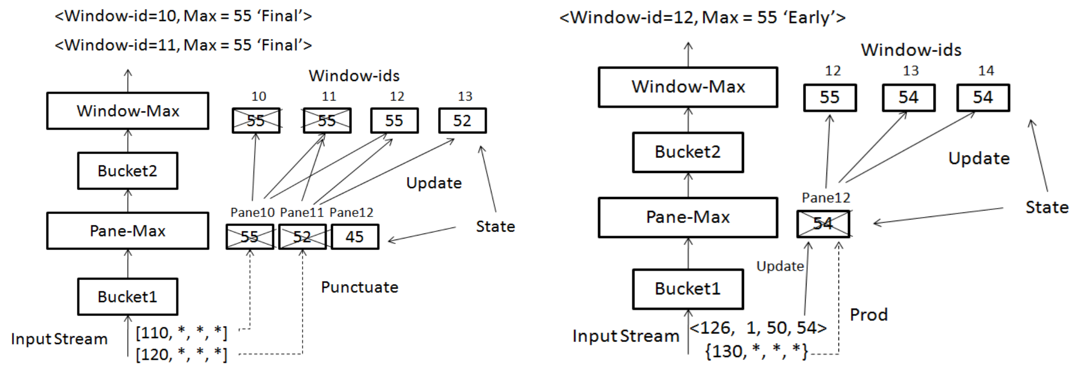
The subplan — Bucket1 followed by Pane-Max — next processes another tuple ⟨126,1,50,54⟩, which updates the paned-max value for Pane 12 from 45 to 54. Bucket1 next sees the prod {130,*,*,*}, which is an indication to the query that it should now produce results up to time 130 using whatever computations the query has performed so far. Bucket1 translates the prod {130,*,*,*} to {*,*,*,*,pid_from=12,pid_to=*}. This translated prod on pane-id 12 causes the Pane-Max operator to release the paned-max of 54 calculated for Pane 12 to Pane-Max's downstream operators, and to purge this paned-max from its state. Note that such purging of state on receiving a prod complies with rule R1 described in Section 3.1, because Pane-Max is not the topmost stateful operator in the query

www.manaraa.com



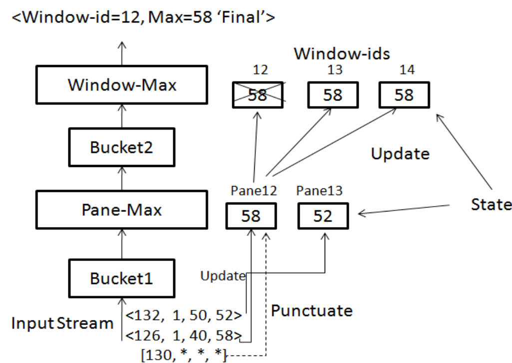
(a) Window Max Query Q3-3

(b) Building Panes



(c) Paned Results Emitted On Receiving Punctuation

(d) Prod Processing in Pane-Max



(e) Final Result by Punctuation

Figure 3.4: Prod Processing in the Paned Implementation of Window-Max

www.manaraa.com

plan. Pane-Max also passes on the prod {*,*,*,*,pid_from=12,pid_to=*} unmodified to Bucket2.

Bucket 2 assigns window-ids to the paned-max tuples that come out of the Pane-Max operator, and translates the pane-id prod {*,*,*,*,pid_from=12,pid_to=*} to the window-id prod {*,*,*,*,*,*,wid_from=12,wid_to=*}. At the Window-Max operator, the partial max 54 is used to update the current windowed-max for windows 10 to 12. On receiving the prod {*,*,*,*,*,*,wid_from=12,wid_to=*}, Window-Max emits the windowed-max result (marked as 'Early') for Window 12. As per Rule R1, Window-Max does not purge state for Window 12, because Window-Max in this query plan is the topmost stateful operator. Retaining state here is necessary: Because S is not expected to be grammatical with respect to prods, in S, more tuples for Window 12 may follow the prod {*,*,*,*,*,*,wid_from=12,wid_to=*}. If the largest value for volume among the tuples that arrive between the prod {*,*,*,*,*,*,wid_from=12,wid_to=*} and its corresponding punctuation [*,*,*,*,*,*,wid_from=12,wid_to=*] is smaller than the *early* windowed-max that Window-Max has output, then the *final* windowed-max for Window 12 will be same as its early windowed-max. To accommodate such a case where the early windowed-max is the overall result, retaining the early result in the Window-Max state is necessary. A similar argument holds for the use of Rule R1 in case of the paned implementation of the Window-Min operator.

R1 for *cumulative* aggregates: The state purging rule, R1, ensures correct operation not only for Window-Max and Window-Min, but also for cumulative aggregate functions such as Count, Sum, or Average computed over windows. For example, in case of the paned implementation of Window-Sum, purging state on receiving a prod at the Pane-Sum operator will ensure that the same subset of data does not contribute more than once to the results calculated at Window-Sum. Retaining state after processing prods at the Window-Sum ensures that the *final* punctuated results are calculated correctly by including the contributions

المنارة للاستشارات
www.manaraa.com

from all the tuples in a window.

Regardless of the use of prodding, a query eventually processes all the tuples or punctuations as normal, i.e., as detailed in Section 2.3. Each tuple contributes to the single pane that the tuple belongs to. Each pane-level result updates 3 (or, in general RANGE/SLIDE) windows. A punctuation allows window-aggregates at the pane and window level to emit windowed results and purge the result entries from operator states. Because a prod purges state at the pane that it covers, the first tuple belonging to the purged pane that follows the corresponding prod recreates the state for the pane. For example, as shown in Figure 3.4(e), the tuple ⟨126,1,40,58⟩ creates a state entry with value 58 for the paned-max for pane 12.

Note that Bucket and window-aggregate operators are oblivious to such recreation of a pane. What essentially happens is that a prod breaks a pane into multiple sub-panes (call them *pane-fragments*). Whereas, in theory, a window has fixed number of panes, the NiagaraST implementation of panes relaxes this requirement, because a NiagaraST bucket relies solely on the pane-id in a tuple to assign it to a range of windows. Furthermore, the existence of panes is transparent to a NiagaraST window-aggregate operator, which can accept any number of inputs with the same window-id range.

Punctuations as usual let window-aggregates produce results and purge state. The punctuation [130,*,*,*] causes Pane 12 to release the paned-max of 58 (marked as 'Final'), which updates windowed-max for the windows 12, 13, and 14. The new paned-max 58 replaces the windowed-max of 54 previously held in the state of window 12. On receiving the punctuation on Window 12, Window-Max outputs the correct final windowed-max of 58. After emitting the final result, Window-Max purges the state entry for Window 12.

Mis-timed prods: In Section 3.2, I discussed the sources of prods. It is possible that a source of prods injects a prod too early or too late in a data stream. If a prod arrives at a stateful operator when the prod's corresponding window has

المنارة للاستشارات
www.manaraa.com

not even been formed at the operator (i.e., if a prod reaches the operator before any of its corresponding tuples), then the prod is too early for the operator. Such premature prodding may occur for various reasons such as data being delayed too long, or bursty data arrivals causing tuple-processing backlogs at some operators, or a user trying to refresh a particular window too aggressively. A prod is too late for a stateful operator if by the time the prod arrives at the operator, the state at the operator for the window corresponding to the prod has been already purged (by a punctuation). Such belated prodding may occur for reasons such as feedback propagation delay, a late user refresh, or a high user-specified prodding offset to a prodder.

All the stateful operators pass on a mis-timed prod downstream without updating their own state or performing any computation. Similarly, all stateless operators pass on the delayed or premature prods downstream without any processing; the exception is the Bucket operator, which performs the appropriate prod-translation regardless of when the operator encounters a prod.

Note that it is important that all the operators pass on the prods downstream, because a prod that had no effect on one stateful operator's results may still go on to trigger early results from some other downstream operator. For example, a prod that is too early for a pane may still affect the window encompassing the pane. Or, a prod that is too late for a lower level aggregate may still go on to trigger result generation at a higher level aggregate, which could not produce output earlier in response to a timely prod, say, because an in-between multiple-input operator (such as a Union or a Join) was awaiting data from some other input stream.

State purging in general: As described in Section 3.1, Rule R1 is applicable only to the query plans that include regular or paned implementation of window-aggregates. The decision regarding operator state purging in case of more complex query plans may not be straightforward. For example, consider a sliding-window query, call it Q3-4, that first computes Sum(volume) per sensor over data stream

المنارة للاستشارات
www.manaraa.com

S in Figure 3.4(a), and then computes a Max of the sums over all the sensors. The query plan for the paned implementation of Q3-4 will include a Pane-Sum followed by a Window-Sum followed by a Window-Max. In such a query plan, Pane-Sum can purge its state on receiving a prod, and emit incremental results to the Window-Sum operator. However, Window-Sum needs to accumulate these increments. The operator cannot purge its state unless it receives a punctuation, because the topmost operator Window-Max maintains only the maximum of the sums it receives, and loses all other results it receives from Window-Sum.

In general, there can be multiple operators, in a query plan, that need to accumulate incremental results received from below in the query plan. Each operator can be parameterized to denote whether the operator should act as an accumulator of increments (by retaining its state after emitting a prodded result). Enabling a DSMS to deduce the accumulating operators in a given query plan can be a subject of future investigation.

Prodding alternatives: Note that the prodding approach, as described in this chapter, involves a prod that flows across a query plan with the data stream. Having a prod flow across the entire query plan leads to some latency even in the early results. An alternative approach to prodding is to independently signal one or more query operators to emit early results. However, in such approach to prodding, achieving synchronization is complicated: we must first prod a lower operator (in the query plan), then ensure that results from the lower operator have arrived at the next higher operator, then prod the higher operator, and so forth. A key challenge is that the time for the interoperator transfer of results is unpredictable. I explain this challenge in more detail in Section 6.1.

A simpler but suboptimal alternative to get early query results is to prod a single operator — probably the topmost aggregate operator in a query plan — to produce results. Such operator-level prodding can be achieved by sending an external signal to the operator being prodded, or by injecting a prod in a data

المنارة للاستشارات
www.manaraa.com

stream just before the operator. However, operator-level prodding will not be optimal when a multi-level query plan includes state distributed across multiple query operators, and the state from more than one operator contributes to the query-result computation. The issue in prodding any single operator in such a multi-level query plan is as follows: If we prod just the topmost stateful operator, the produced early result will not include contributions from the stateful operators lower in the query plan. If we prod just an operator that is lower in a multi-level query plan, such prodding will not cause higher-level operators to emit early results. For example, in the plan for query Q3-3 (Figure 3.4), prodding only Window-Max will cause the operator to emit early results that do not include contribution of Pane-Max, whereas, prodding only Pane-Max will cause the pane-level contribution to reach up to Window-Max but Window-Max will not be triggered to produce window-level early results.

3.4 ACCURACY OF EARLY RESULTS

Despite using the strategy of prodding an entire query plan to maximize the contribution that goes in early result computation, the early results may not be accurate, because they do not include contribution by any tuples that arrive between the prod and the punctuation for the window. The degree of deviation of an early result from the corresponding final result (which is accurate given that the input stream is grammatical with respect to punctuations) depends on where in the data stream the prod occurs with respect to its corresponding punctuation (i.e., on what percentage of the window's tuples lie between the prod and the punctuation for the window). The two other factors that affect the degree of inaccuracy in early results are: (a) the pattern of data values, and (b) the aggregate function being used. In the rest of this section, I discuss accuracy of early results for different aggregate functions assuming *uniform* distribution of the "values being aggregated" (or, aggr-values for short) across data in a window. Note that when the distribution

www.manaraa.com

in the aggr-values is known to be different than uniform, the expected accuracy in the early results will need a different treatment.

Given the uniform distribution of aggr-values in a data stream, the smaller the percentage of *trailing* tuples — i.e., the tuples in a window that arrive between the prod and the punctuation for the window — the smaller the degree of inaccuracy in the early results. Let us examine this claim as it applies to different aggregate functions used for windows in a data stream query: For aggregates such as Average, Sum, or Count, each trailing tuple in a window potentially adds to the degree of inaccuracy in the early result for the window. For Average, if the early result is produced after processing substantial portion of the tuples in a window, the deviation of the early results from true average will be slight. Such small inaccuracy may be tolerable in application domains such as vehicle-traffic congestion control (e.g., an approximate average of vehicle speeds over a time window can be a sufficient indicator of the congestion condition). In the case of Count, each trailing tuple adds one to the error. For Sum, the inaccuracy in the early results will be directly proportional to the percentage of the trailing tuples for a window (again because the values contributing to the Sum are uniformly distributed). For aggregate functions such as Max and Min, the trailing tuples might contain the actual maximum or minimum value. So, the early Max or Min may deviate arbitrarily from the correct final maximum or minimum. Or, the true maximum or minimum may have arrived at the system before the corresponding prod, thus giving accurate early results. For the assumed uniform distribution, the smaller the percentage of trailing tuples, the more the chances of getting the actual maximum or minimum (or, if not the actual, a value that is close to the actual maximum or minimum) included in the early result.

In general, it is difficult to give a precise measure of the accuracy of the early results: The distribution of the aggr-values in a data stream may not be known. Even when such distribution is known, it is hard to compute the percentage of

www.manaraa.com

window data that contributes to the early result, because this percentage depends on the combination of the time a window is prodded and the time-wise distribution of the data arrival. Also, despite possible prior knowledge of the aforementioned factors for a data stream, because of variability, the accuracy measure for a specific early result can be only approximate. However, it is straightforward to compute the accuracy of an early result once the corresponding final result is obtained. This way, a stream application can monitor the early-result accuracy of a running system over a period, giving users a practical measure of the usefulness of prodding with a given degree of aggressiveness.

المنارة للاستشارات
www.manaraa.com

Chapter 4

EVALUATION

It is important to evaluate the tradeoffs in prodding in terms of the prodded-result latency gains together with the degree of accuracy of the early results. In this chapter, I describe my experiments, conducted to evaluate various aspects of prodding.

I discuss the run-time environment for the experiments and the query instrumentation I have used in Section 4.1. In Section 4.2, I evaluate whether prodding imposes any significant overheads to a DSMS. Section 4.3 describes my experiments to evaluate latency gains and accuracy of early results from queries over real-world data streams that are subject to delivery delays (Section 4.3.1) or to bursty data arrivals (Section 4.3.2). The prodded results, while obtained with low latency, may compromise on accuracy. Further, the more aggressively you prod a window, the lower should be the latency in getting early results, and, potentially, the greater would be the degree of inaccuracy of the results. I study the effects of *prodding aggressiveness* on the latency gains in early results and on the accuracy of early results in Section 4.4. With increasing prodding aggressiveness, early-result accuracy may degrade at different rates depending on the data (value) patterns and the window-aggregate function being used. I conduct the prodding aggressiveness experiments over two different types of patterns exhibited in the data values being aggregated — uniform distribution (Section 4.4.1) and normal distribution (Section 4.4.2) — and for the aggregate functions Average, Count, Max, and Sum. I close the chapter in Section 4.5 with a summary of the experiment observations and a conclusion, based on these observations, about the applicability of prodding.

المنارة للاستشارات

www.manaraa.com

4.1 ENVIRONMENT AND INSTRUMENTATION

I evaluate prodding using my implementation of the technique in NiagaraST. I have performed all the experiments that I describe in this chapter on a computer having an Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00 GHz with 4.00 GB RAM and running a 64-bit Windows 7 Operating System. I have run the experiments with the Java Virtual Machine's maximum heap size of 192 MB. I set the inter-operator buffers to contain 5 pages of tuples with each tuple page having the capacity to contain 30 tuples per page.

I define the *latency* in producing a query result for a window as the difference between the arrival time of the first tuple for the window at the first Bucket operator in a query plan and the time of producing the result for the window. For example, for a 5 minute window, say 5-10, if the first tuple arrives at time 5 and the query result for the window is produced at time 11, then result latency is 6. Similarly, if the first tuple for the window 5-10 arrives at time 7, and the query result is produced at time 11, then the result latency is 4.

In the prototype used for my experiments, for each window, I tag the first tuple belonging to the window with the system timestamp at which the tuple arrives at the first Bucket operator in a query plan. Later, at the window-level aggregate operator, I record: (a) the system time tagged (at the first Bucket operator) to the first tuple of the window and (b) the system time of the production of aggregate output for the window. The absolute difference between these two recorded times for a window is its query result latency.

To simulate delivery of tuples as a data stream, I have designed a new operator, which I call Streamer, that resides at the beginning of a query plan and delivers tuples to the first query-plan operator according to the tuples' application timestamps. In some of my experiments, to simulate possible data-delivery delays on a network, I configure the Streamer to introduce random delays (up to a certain

المنارة للاستشارات
www.manaraa.com

Table 4.1: Overheads of Prodding: Comparison of Result Latencies with and without Prodding over 3 Different Data Streams

Data Stream Id	Final Result Latency With Prodding (Seconds)	Final Result Latency Without Prodding (Seconds)
Uniform_1500	11	11
Burst_2	6	6
Burst_4	6	6

degree) in the delivery of tuples. Such simulation of tuple delivery allows for correlating application timestamps on a tuple to the system clock, and to demand results corresponding to application time 10 at system (or user) time, say, 10 (or, aggressively, at an earlier user time, say, 9).

4.2 OVERHEADS IN PRODDING

The benefits of prodding in terms of latency gains will be of interest to a DSMS developer only if such benefits are not nullified by possible processing overheads incurred by the prodding technique. Therefore, I first describe the experiments that examine whether use of prodding has any significant overheads. Table 4.1 compares the latency of the final results obtained with and without employing the prodding mechanism for three different data streams. The data streams and the corresponding queries used for the comparison experiments in this section are from among those I use in the Delay and Burst experiments in Sections 4.3.1 and 4.3.2, respectively. (Compare the data stream ids in Table 4.1 to those in the Tables 4.2 and 4.3.)

As the Table 4.1 shows, the latencies for final results obtained with and without prodding are very close. Getting such similar result latencies is reasonable, because

المنارة للاستشارات
www.manaraa.com

prods are sparsely present in a data stream, and are processed in constant time. The closeness of latencies in getting final results with and without prodding shows that the latency gains obtained through prodding are true gains, and are not obtained at the expense of delaying the final results, thereby enlarging the gap between early and final results.

4.3 LATENCY GAINS

In this section, I discuss the results for the experiments that I have run to study the latency gains and accuracy of early (interchangeably, *prodded*) results for window-aggregate queries over data streams. I describe my experiments to evaluate the prodding technique in case of data streams with delivery delays in Section 4.3.1, and in case of data streams with bursty data arrivals in Section 4.3.2.

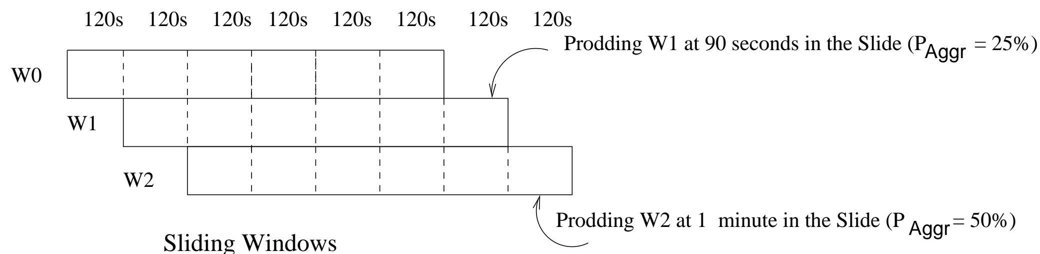


Figure 4.1: Prodding a Sliding Window with Different Degrees of Prodding Aggressiveness (P_{Aggr})

4.3.1 Latency Due to Data Delivery Delays

Delays in data arrival at a DSMS can lead to latency in computing window query results over data streams. In a situation where data delivery delays cause such latency, I expect prodding to provide timely estimates to window query results. Figure 4.2 shows the result latencies (defined above in Section 4.1) against window-ids for the paned implementation of the window query

المنارة للاستشارات
www.manaraa.com

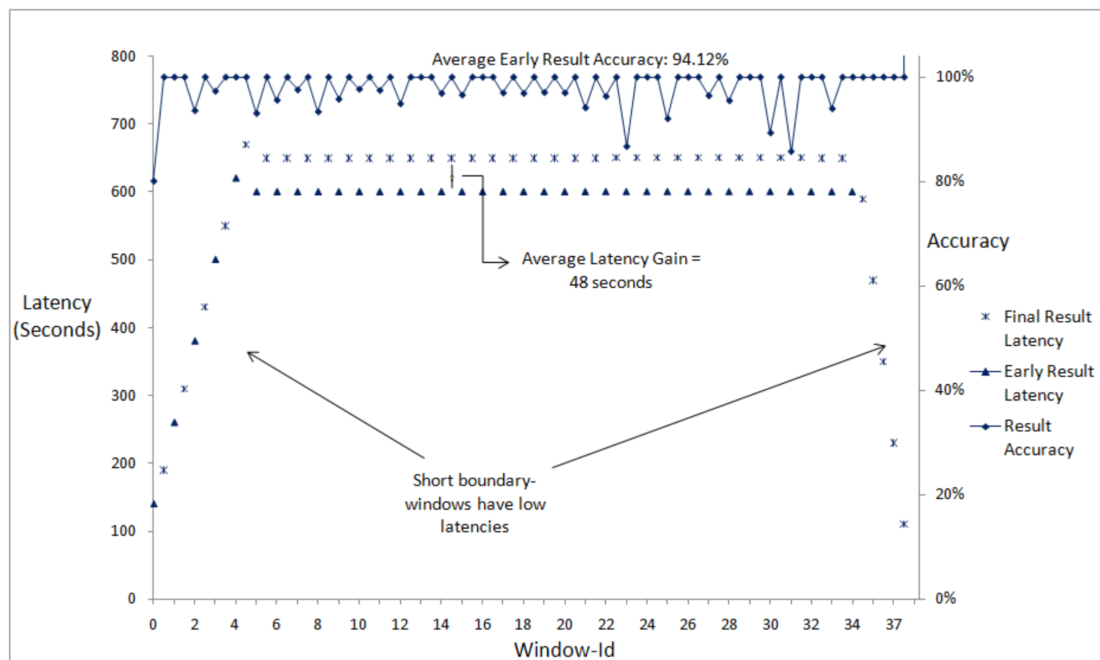


Figure 4.2: Latency Gains and Accuracy Curves for Q4-1 over Delayed Data Prodded at 90 Seconds in a 2-Minute Slide

```
Q4-1:  Select Sum(volume) from S
       Window[Range 10min, Slide 2min]
```

where S is a prodded punctuated vehicle traffic data stream [1] with the schema (starttime, volume, speed, occupancy, status). The traffic-data records are delivered to the system on the fixed reporting schedule of every 20 seconds. Therefore, on average, 6 records are delivered to the system in a given period of 2 minutes. Due to the delays in the sensor networks, the arrival gaps between two successive tuples may be larger than 20 seconds. I use a Streamer operator (as described in Section 4.1) configured to simulate data delivery at 20-second intervals with possible (further) delays of up to 20 seconds. As one part of preprocessing for this experiment, I insert, in the stream S, prods with timestamps such that each window is prodded for a result at 90 seconds from the beginning of a (120 second) slide (see Figure 4.1). As another part of preprocessing, I insert an appropriate

www.manaraa.com

punctuation after every increment by 120 seconds in the application timestamps on the tuples.

A slide of 120 seconds implies that each new sliding window will end 120 seconds further in time from its previous window. Therefore, a DSMS user will expect to see a new Window-Sum every 120 seconds. Note that with the system configuration described in Section 4.1, processing of 6 new tuples to compute a Sum aggregate takes a negligible amount of time — the cause of result latency here can only be the schedule of data delivery to the system.

Notice in Figure 4.2 that the latency gains in the early results are represented by the gap between the latency graphs for the final (shown by *'s) and the early (shown by solid triangles) results. The average gain in the result latency of the early results over the latency in the final results is $616 - 568 = 48$ seconds. Thus, for a slide of 2 minutes, we get early results in $120 - 48 = 72$ seconds on average.

See the accuracy curve at the top in the Figure 4.2. Final results are (100%) accurate, since the streams are all grammatical with respect to punctuations. Let F_v denote the final aggregate value for a window and E_v denote the corresponding early value. I compute the *average percentage accuracy* (or, simply, *accuracy* for the rest of this chapter) of early results as the average of $[(F_v - E_v)/F_v] * 100$ over all the results. The average accuracy of the early results for Q4-1 is 94.12%. Such a degree of accuracy may be "good enough" in vehicle-traffic–monitoring applications to indicate a congestion condition (by correlating the obtained early volume estimates with the corresponding speed readings). In general, to improve on the result accuracy, a user may choose to prod less aggressively (by tuning the source of prodding to prod the DSMS for results after, say, 120 or 125 seconds from the start of a slide). I study the relationship among prodding aggressiveness, result accuracy, and latency gains in the early results in Section 4.4.

Computation of the average window result latencies and accuracies in this experiment also includes the readings taken for the windows at the beginning and

المنارة للاستشارات
www.manaraa.com

the end of the stream S. Such boundary-windows in a data stream are shorter than the window range specified in a query, and hence do not represent a typical window in the query over the given stream. To reduce the contribution of boundary windows in the experimental results, I observed latency gains of the early results over longer data streams. I generated data streams of different lengths — spanning 1000, 1500, and 2000 seconds — with data values uniformly distributed. Each of these three data streams contains about 500 tuples per 30-second window, and is prodded at 7 seconds into a (10-second) slide during preprocessing. For these three experimental runs, a Streamer simulates delivery of tuples according to their application timestamps with added delays of up to 0.5 seconds for every tuple. Over the generated streams, I executed the paned implementation of the query

```
Select Max(speed) from S
Window[Range=30 Second, Slide=10 Second].
```

As shown in Table 4.2, the system produces results with average early result accuracy maintained at 98 to 99%, and a steady average latency gain of near 3 seconds.

Table 4.2: Latency Gains over Delayed Data Streams for Various Lengths

Data Stream Id	Description	Average Accuracy	Average Latency Gain (Seconds)
Uniform_1000	Uniformly distributed data stream 1000 seconds long	99.04	2.80
Uniform_1500	Uniformly distributed data stream 1500 seconds long	99.48	3.02
Uniform_2000	Uniformly distributed data stream 2000 seconds long	98.11	2.98

المنارة للاستشارات
www.manaraa.com

4.3.2 Latency Due to Bursts in Data

Another important reason for high latencies in producing results for window queries is data-arrival bursts that can temporarily overwhelm the processing capacity of a system. To study prodding in case of such bursty data arrivals, I use the query

```
Q4-2:  Select Max(counter) from S
       Window[Range=5 Second, Slide=1 Second]
```

where S is a punctuated data stream of wireless network records obtained from the Dartmouth college campus [15]. S has schema (timestamp, access_point_name, counter, message). The values for *counter* in S are neither monotonically increasing nor monotonically decreasing. There may be short runs of counter values within a window that having increasing values of counter.

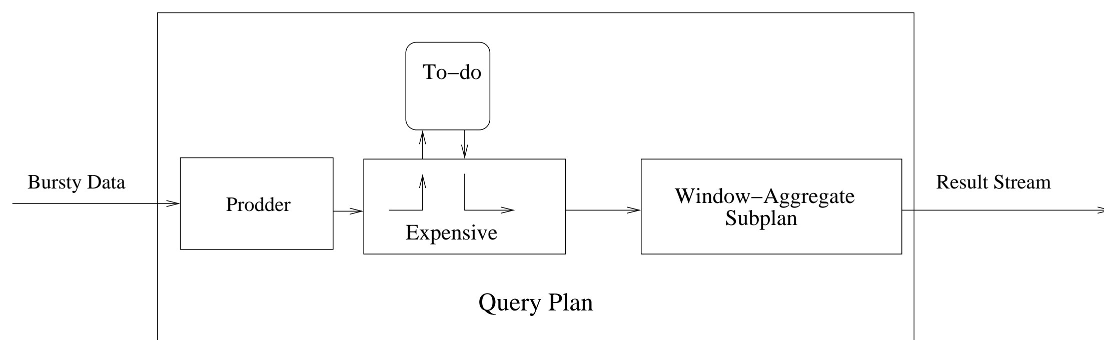


Figure 4.3: Prodder and Expensive Operator in Burst Experiment Query Plan

The Expensive operator: For the experiments in this subsection, it is important to ensure that the bursty portions of S temporarily overwhelm the system's processing capacity, while the non-bursty portions of S are processed in a timely manner. To simulate such effects, I include in my query plan an operator that increases the per-tuple processing cost by performing some dummy expensive work on each tuple in the data stream. I call this operator Expensive[1] (see Figure 4.3). With

[1]Rafael Fernandez has introduced, and used the Expensive operator in the experiments for his PhD thesis.

المنارة للاستشارات
www.manaraa.com

Expensive being part of my query plan, the windows that contain bursts of data take longer to produce aggregate results. Expensive operates as follows: (a) It puts all the tuples and punctuations it reads in a separate *todo* list, (b) spawns a thread that reads tuples one by one from the todo list, performs an expensive operation on each tuple read, and emits the tuple to the Expensive's downstream operator in the query plan, and (c) processes any prod it reads immediately, thereby letting a prod bypass the todo list. Note that such operation of Expensive rescues a prod from being delayed due to a burst in the data stream with which the prod is flowing. In a real-world business application, such an Expensive operator may be some expensive operation such as a Database table access (say, for a join), or an expensive sub-plan in a query, or a pattern matching operation in case of String data. For example, in a query like such as

```
Select count(word) from WordStream
where word foundIn(SomeText)
Window[Range=5 min, Slide=1 min]
```

where "foundIn" is an expensive search function.

For the experiments with bursty data sets, instead of inserting the prods in the data stream as a part of preprocessing (like in the delay experiments described in the previous subsection), I use a Prodder operator, which resides at the beginning of the query plan (see Figure 4.3), and injects a prod into the data stream at every 1-second interval of system time. Each prod triggers the prodded-result computation for a new (1-second sliding) window. Thus, prodding here is used mainly to get timely results even when the system encounters a data burst.

Figure 4.4 shows the data-arrival patterns in S as the number of tuples per second. The circled parts of the curve are some of the regions that exhibit spikes in data arrivals. Data-arrival spikes lead to temporarily high processing loads for a DSMS Therefore, I expect query Q4-2 to produce results for the windows corresponding to these spike-affected regions with a higher latency. We may also

المنارة للاستشارات
www.manaraa.com

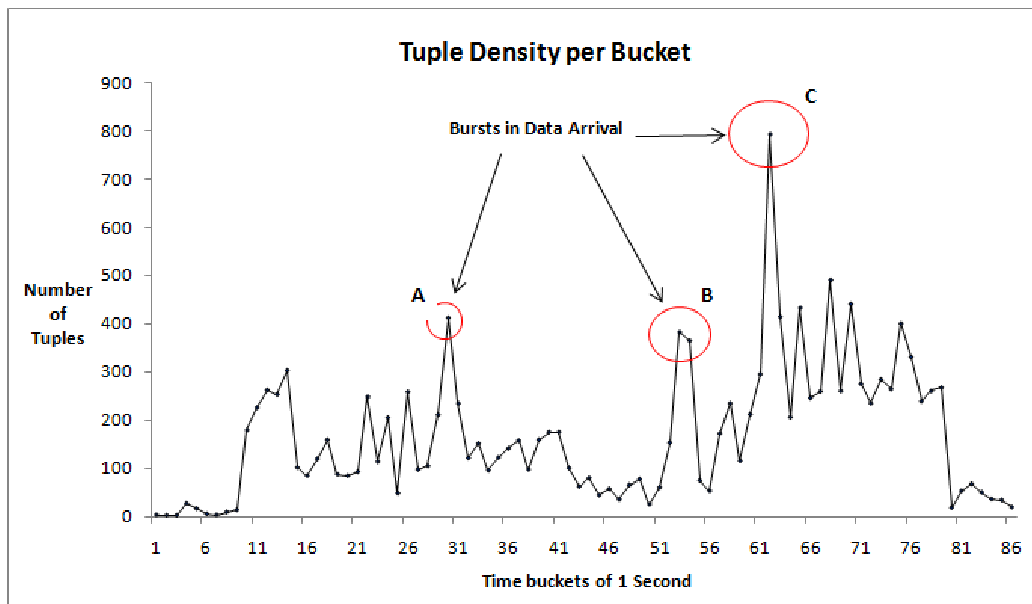


Figure 4.4: Data Arrival Bursts in a Wireless Network

see some prodded results for these windows. Another aspect to notice in Figure 4.4 is that the data burst at the circled region A is followed by a period of low-density data arrivals, which may allow a DSMS to recover from possible data-processing backlogs caused by spikes at A. In contrast, the bursts at the circled regions B and C are followed by medium- to high-density data arrivals. In other words, regions B and C represent more sustained bursts. Such a sustained burst may preclude a DSMS from quickly eliminating the data-processing backlog, thereby causing delays in the result production of some subsequent windows having low data density.

For the windows that do not contain a data burst, the DSMS finishes the result computation well before 1 second. For such windows, only the final results are produced, because a prod (which comes after 1 second from the beginning of a slide) for these windows is processed by the query operators after the final results for the prodded window have been already produced. Therefore, while measuring the latency gains in the burst experiments, I average latencies for only the windows

للاستشارات
www.manaraa.com

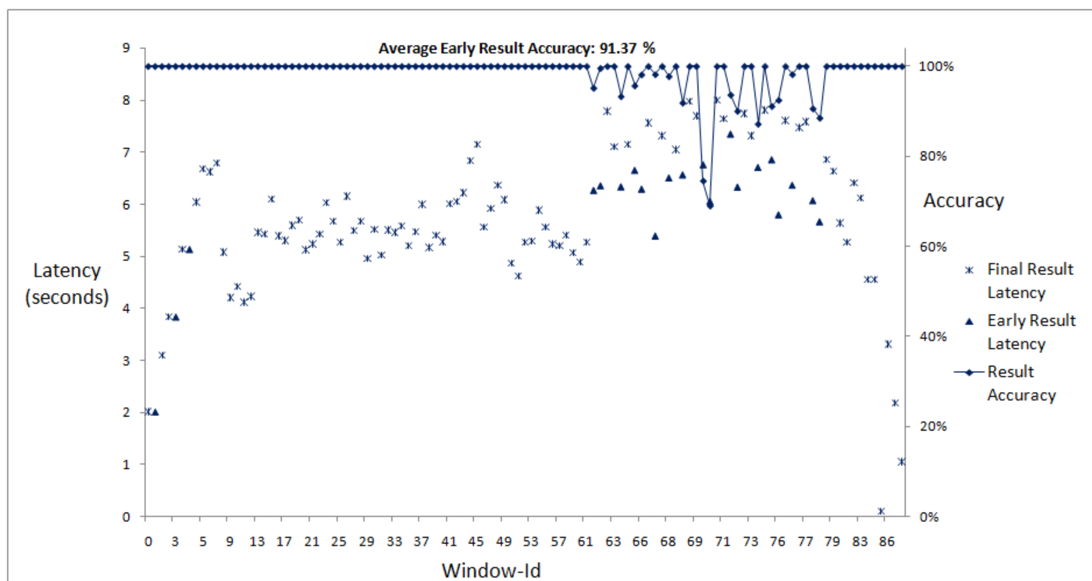


Figure 4.5: Q4-2 over Bursty Data Prodded at 1 Second in a 1-Second Slide

that have an early result.

As we see in Figure 4.4, sustained data bursts occur between windows 62 and 75. (Notice that the chosen bucket size of 1 Second in the figure equals the 1 Second slide in the query Q4-2.) Therefore, between these two windows, final results are produced with a high latency (see the latency plot in Figure 4.5). In this burst-affected region of the stream, the outcomes of prodding become visible in the form of early results. The effects of query-result-computation delays caused by data bursts are cumulative, i.e., windows subsequent to the burst-affected windows are also affected thereby incurring delays in their own result generation. Therefore, we continue to see the early results until the bursts recede (which happens past window 80).

Again, the accuracy curve is superimposed on the latency graph. Average accuracy for the early results is 91.37%. Once again, to counter the contribution of the boundary window results in the accuracy and latency calculations, I executed query Q4-2 over twice and four times longer data streams with bursts in them.

www.manaraa.com

The configuration for the Prodder is unchanged, that is, it prods every second for a result over a new 1-second slide. The results listed in Table 4.3 show that for various lengths of bursty data streams, prodding continues to produce sustained latency gains with average early-result accuracy of more than 97%.

Table 4.3: Latency Gains over Bursty Data Substreams with Various Burst Lengths

Data Stream Id	Description	Average Accuracy	Average Latency Gain (Seconds)
Burst_1	Bursty data stream with ˜14000 records	97.62	1.33
Burst_2	Bursty data stream with ˜28000 records	98.38	1.27
Burst_4	Bursty data stream with ˜56000 records	98.65	1.18

4.4 PRODDING AGGRESSIVENESS

Obtaining the right balance between window-query result latency and accuracy is key in effectively using prodding. In this section, I define prodding aggressiveness, and describe my experiments to study how prodding aggressiveness affects the latency-accuracy balance of early results. In particular, I study how prodding aggressiveness affects latency gains and result accuracy for the window aggregate functions Average, Max, Sum, and Count. I expect that result accuracy will vary with the distribution of data values in a stream. I conduct two sets of experiments

المنارة للاستشارات
www.manaraa.com

related to prodding aggressiveness: One using data generated with a uniform distribution of aggregate-attribute values (Section 4.4.1), and the other using data generated with a normal distribution of the aggregate-attribute values (Section 4.4.2).

For a source that prods a DSMS to produce results over a window with slide w at the system time that is p prior to the end (application) time of the window, I define prodding aggressiveness (P_{Aggr}) as the percentage of w covered by p. For example, for a window that is a result of a 120-second slide w, say from 12.00.00 to 12.02.00PM, if a source prods for results over the window with end application time of 12.02.00PM at the system time 12.01.30 (that is, 30 seconds prior to the window end time), then the P_{Aggr} of the source is $30/120 * 100 = 25\%$. An alternative way to describe prodding aggressiveness is: if a source prods a window 90 seconds into a (120-second) slide, then P_{Aggr} is equal to $(120 - 90)/120 * 100 = 25\%$ (see window W1 in Figure 4.1).

For a given experiment, I measure average percentage latency gain (or, simply *latency gain*, for this section) as $[(F_L - E_L)/F_L] * 100$ where E_L is the average latency of all early results and F_L is the average latency of all the corresponding final results. I measure average percentage early-result accuracy (or, simply *accuracy*, for this section) as the average of $[(F_V - |F_V - E_V|)/F_V] * 100$ over all pairs of early and final results, where F_V is the value of a final result and E_V is the value of a corresponding early result.

The data that I use in both the sets of experiments are 95% dense with tuples distributed more or less uniformly (timestampwise) across a window. By 95% dense, I mean that the density parameter in my data generator is equal to 95, which indicates that after generating a tuple with timestamp t, the data generator produces the next tuple with timestamp t with probability 95%, and with timestamp $t + 1$ second with probability 5%. In the experiments in this section, a window ranging over 30 seconds typically contains about 540 tuples, and an input data stream runs from 0 to 2000 seconds. Using sparser windows would be less useful for two reasons: (a) Sparse windows do not represent the case of system capacity being overwhelmed by high data volumes. (b) The early-result accuracy for a sparse window may be largely dominated by the data

المنارة للاستشارات

www.manaraa.com

values of the small number of tuples in a window that are processed after a corresponding prod. The query I use for the experiments in this section is of the form

```
Select AGGREGATE(value) from S
Window [Range 30 Second, Slide 10 Second]
```

where AGGREGATE is Average, Max, Sum, or Count.

4.4.1 Uniform Distribution

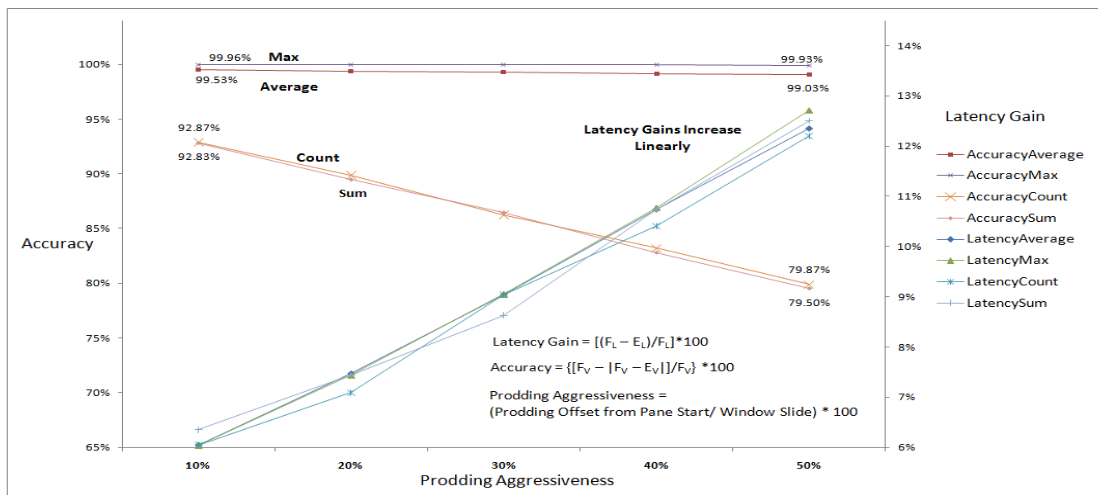


Figure 4.6: Effects of Prodding Aggressiveness on Average Early-result Accuracy and Latency Gains over a Stream with Uniform Data Values

Figure 4.6 shows the average window-aggregate query result latency gains (hereafter, latency gains) and the average result accuracies (hereafter, result accuracies) against prodding aggressiveness for the aggregate functions Average, Max, Sum, and Count over a data stream S with the aggregate-attribute values uniformly distributed in the range 0 to 999. See that for all the aggregate functions in the graph, the latency gains increase about linearly with the increase in the prodding aggressiveness. Such a linear increase in latency gains is expected, because, the higher the P_{Aggr}, the earlier the aggregate computation gets triggered for each window.

Notice how different degrees of P_{Aggr} affect the early result accuracies for the different aggregate functions. After sampling sufficient data values, Average produces a close to

المنارة للاستشارات
www.manaraa.com

correct answer. The 99.53% accuracy for Average when P_{Aggr} is equal to 10% decreases only slightly to 99.03% when P_{Aggr} is increased to 50%. More remarkable is the flatness of the Max accuracy curve — the average accuracy drops from 99.96% to only 99.93% when P_{Aggr} changes from 10 to 50%. Note that to maintain high accuracy for Max, it suffices that a single value — the true Max for the window or a value close to the true Max — is covered in the early result computation. Therefore the early result accuracy for Max can frequently be close to 100% (given that the data are not biased) as we can observe in the figure for different degrees of prodding aggressiveness. In the Section 4.4.2, we shall see that the early Max-result-accuracy graph over normally distributed data values shows similar behavior in terms of maintaining high accuracy despite aggressive prodding.

For Sum and Count, as expected, the accuracy goes down with the increase in P_{Aggr}, because fewer tuples are included in the early result computation. At 50% P_{Aggr}, the average result accuracy for Sum is 79.5% , and that for Count is 79.87%.

4.4.2 Normal Distribution

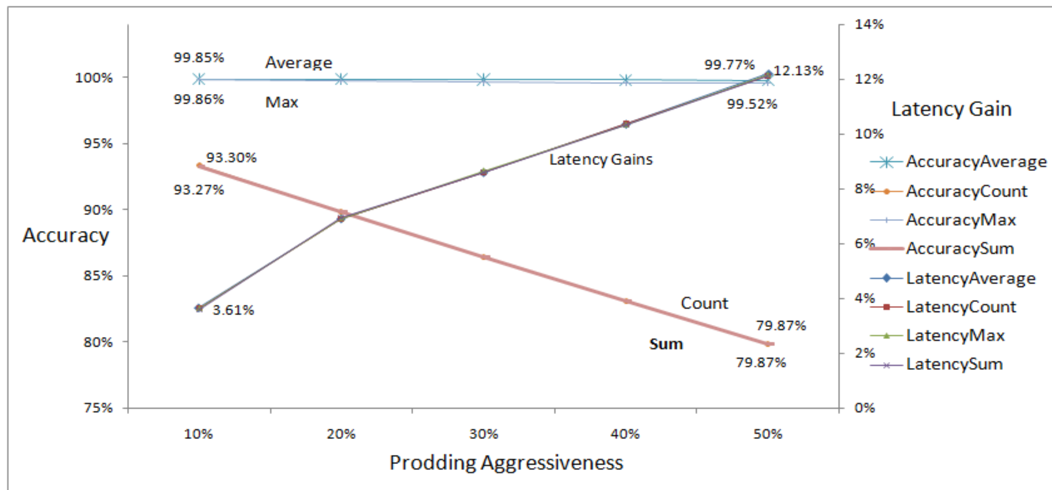


Figure 4.7: Effects of Prodding Aggressiveness on Average Early-result Accuracy and Latency Gains over Stream Having Normally Distributed Data Values with Mean=500 and SD=100

The relationship among prodding aggressiveness, result accuracy, and latency gains

www.manaraa.com

for window-queries looks similar (to that in case of a uniform distribution) with a normal distribution of aggregate-attribute values. Figure 4.7 shows the corresponding plots obtained for a normal distribution of data values with mean equal to 500 and standard deviation (SD) equal to 100. The accuracy graph for Max once again looks similar to the Average graph. With SD equal to 100, the true (i.e., final) Max will be relatively (with respect to higher SDs) close to the mean (because, as per the 68-95-99.7 rule for normal distribution [7], nearly all values lie within three standard deviations). Therefore, the early results have a high probability of including the true Max or a close-to–true Max value in their computation. I executed the same query with normally distributed data values for larger SDs — viz., SD = 400 and SD = 1000. Figures 4.8 and 4.9 show the accuracy graphs corresponding to the two experiments. Plots for SD = 400 look similar to those for SD = 100. For SD = 1000, the Average degrades more quickly, albeit slightly, than the Max curve, which suggests that a higher variance in data values has more adverse effects on the accuracy of early Averages than on the early Maxes. The other two aggregate functions — Sum and Count — once again linearly degrade in the early result accuracy with increasing P_{Aggr}, because proportionately fewer tuples contribute to the computation.

The average-accuracy graphs for Average and Max in case of both uniformly and normally distributed data values demonstrate that prodding, in case of these distributions, can produce low-latency answers for sliding-window Average queries or Max queries over data streams, without significantly compromising result accuracy. For sliding-window Sum and Count queries, result accuracy degrades linearly with increase in prodding aggressiveness.

Worst case accuracies: The results discussed so far speak about the average accuracies of the early aggregates over several windows. The usefulness of this measure of accuracy will be determined by whether an application is concerned more about the average accuracy of the results over several windows or is sensitive to the lower bound on the worst-case early-result accuracy. For example, in an application using query Q4-3: “Give the hourly count of the 5 minute windows having Max(Temperature) above 80 degrees”, it matters more to overall produce Max (Temperature) values that are close to

المنارة للاستشارات

www.manaraa.com

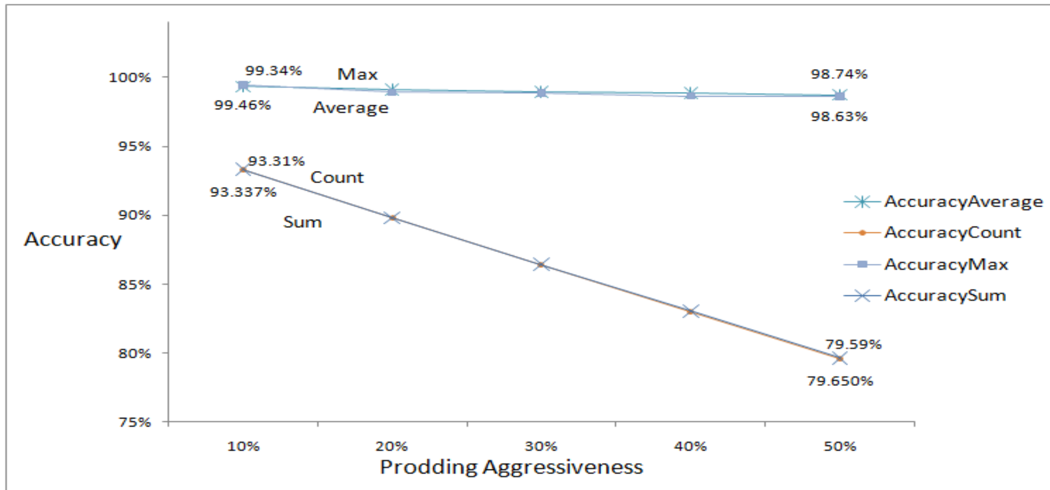


Figure 4.8: Effects of Prodding Aggressiveness on Average Early-result Accuracy over Stream Having Normally Distributed Data Values with Mean=500 and SD=400

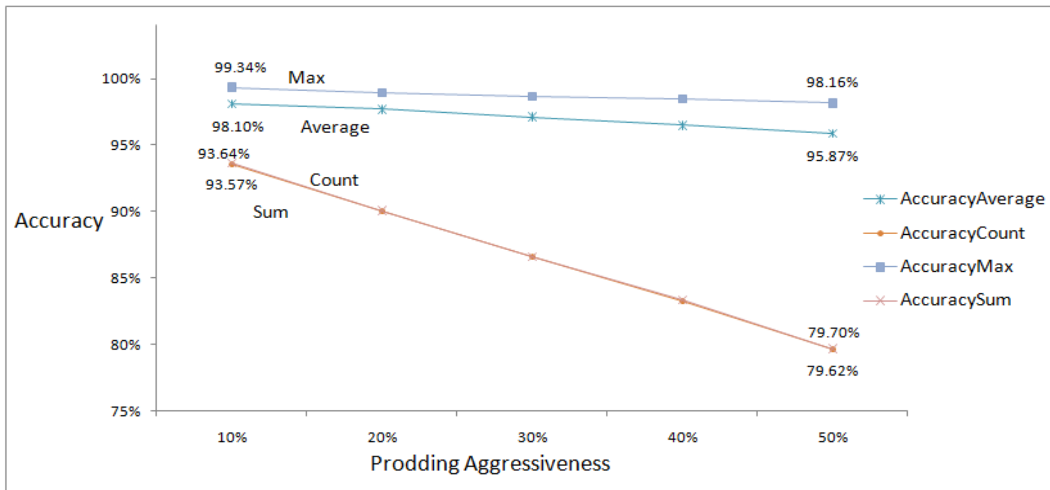


Figure 4.9: Effects of Prodding Aggressiveness on Average Early-result Accuracy over Stream Having Normally Distributed Data Values with Mean=500 and SD=1000

the true Max most of the time; an extremely inaccurate Max in an odd window affects the final query output (of hourly count) by just one. In other words, in query Q4-3, the degree of inaccuracy ceases to matter once the early estimate is below 80 degrees.

المنارة للاستشارات
www.manaraa.com

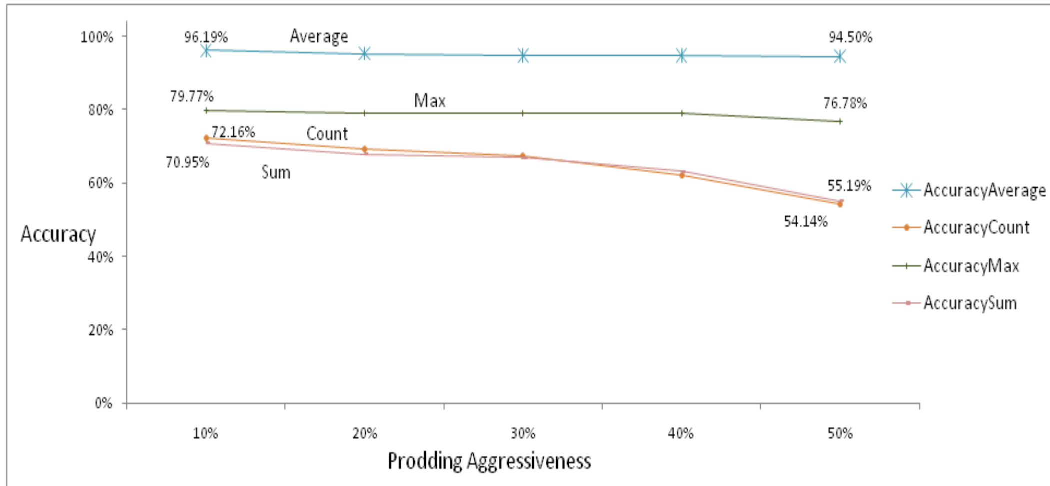


Figure 4.10: Worst-case Accuracies for Early Results for 30-second Sliding Windows over Data Values with SD=400 and Mean=500

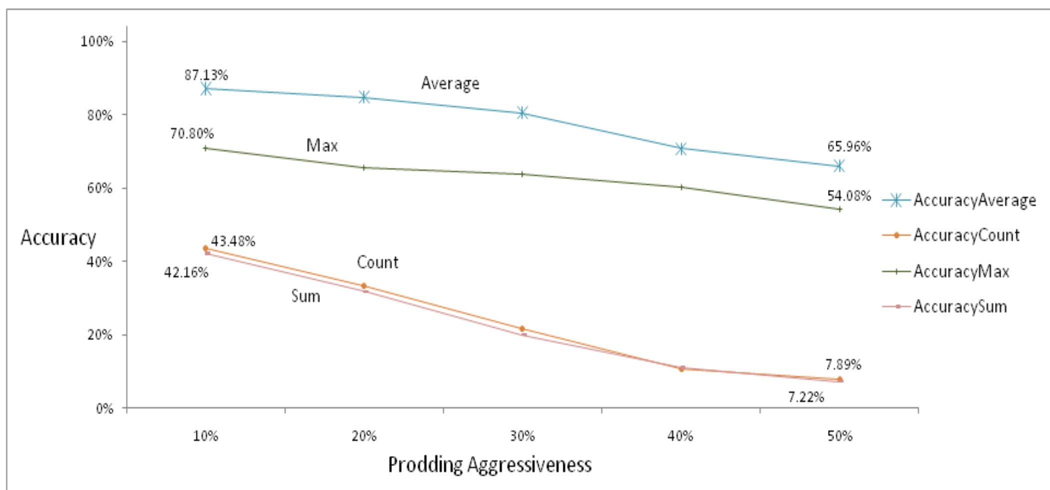


Figure 4.11: Worst-case Accuracies for Early Results over 30-Second Tumbling Windows over Data Values with SD=400 and Mean=500

On the other hand, the degree of inaccuracy may be critical for a query such as Q4-4: “Give the difference between a Maximum and a Minimum stock price for IBM every 5 minutes”. In such a query, a highly deviant early answer can affect how the market trends are perceived, thereby rendering the early results unreliable. Figure 4.10 shows the worst-case accuracy curves corresponding to the average accuracy curves we saw in

المنارة للاستشارات
www.manaraa.com

Figure 4.8 (SD = 400). Notice that Average still has high lower bounds (in the mid-90's) on the early estimate accuracy, whereas the worst-case accuracies for Max are significantly lower than their corresponding average accuracies that we saw in Figure 4.8. The worst-case accuracies for Count and Sum are lower yet — varying between low-70's and mid-50's for the P_{Aggr} varying between 10 and 50%.

Prodding sliding vs. tumbling windows: Notice that the average, and even the worst-case accuracies, studied so far in this section lie (often vastly) above 50%. Such high values for accuracy have been possible because of the overlapping computations that occur within sliding windows. Recall Figure 4.1. When a user (or some other source) prods a 12-minute window that slides by 2 minutes with a P_{Aggr} of 50%, i.e, when the user demands an early result at 1 minute into the slide, the (time) portion of the window that contributes to the early result computation is $10 + 1 = 11$ minutes, which is 91.66% of the entire window. It is this overlap of tuples between two successive windows that makes the sliding windows amicable to prodding. In case of tumbling windows, wherein every window computes results over a fresh set of tuples, the accuracies (both worst-case and average), at least in some cases, may degrade well below 50%. Figure 4.11 shows the worst-case accuracies for Average, Max, Count, and Sum against P_{Aggr} for a 30-second tumbling-window query. As in the earlier experiments, each window contains about 540 tuples and the input data stream runs for about 2000 seconds. As the curves in the figure indicate, Average maintains reasonable accuracies (degrading from 87.13 to 65.96% with increasing P_{Aggr}). For the other three aggregate functions, the (worst-case) accuracies of the early result estimates are fairly low for tumbling window queries.

Caution in using prodding: Despite the high early-result accuracies observed in many of the experiments described in this section, the prodding technique in general should be used with caution: (a) The tolerable degree of the result approximation for a given application should be taken into account. For example, getting 98% accurate (2-minute) sliding window query results a minute early may be a profitable tradeoff in vehicle-traffic monitoring or financial-trend-detection applications, but unacceptable in case of a fire-detection application (wherein slight inaccuracy could be intolerable). (b) Any available knowledge about the data patterns in a stream should be considered. For example, an

المنارة للاستشارات
www.manaraa.com

early Max can be particularly vulnerable if data values arrive in ascending order within a window (such data arrival patterns may be encountered when certain trends are building at a data source, e.g., increase in traffic volume at the onset of congestion conditions). (c) The formula to measure accuracy that I have defined in this section may not work for all types of aggregate operations, and hence may need to be suitably redefined. For example, some applications may be text-based, and need some other measure of accuracy such as the Hamming distance between two text strings. An example of an aggregate that returns a text value is a function that takes a stream of several strings and returns the closest match to a pre-specified pattern.

Accuracy trends: Although controlling prodding-aggressiveness lets one control the latency-accuracy balance, we cannot know beforehand how accurate query answers will be with a specific degree of prodding aggressiveness. But a running value of a DSMS's early-result accuracy can be maintained at the output (see Figure 6.4, and the corresponding discussion). Based on the running accuracy over a period, the prodding-aggressiveness can be adjusted. Such an adjustment could be made via an automated feedback, or the adjustment may be manual, i.e., a user might monitor the degree of accuracy over a period, and decide to prod less aggressively to improve result accuracy, or prod more aggressively to gain on latency when the result accuracy is already adequate.

4.5 SUMMARY

Prodding can produce timely result estimates for window aggregate queries over data streams. Some important factors that determine the usefulness of these estimates are: the tolerable degree of result approximation for an application, the aggregate function being used in a query, the use of sliding vs. tumbling windows in a query, and data-arrival and value patterns in a stream. The trick is to achieve the optimum balance between the latency gains in the early results and the accuracy of those results. Such a balance can be achieved by fine-tuning the prodding aggressiveness, keeping in mind the aforementioned factors as they apply to a given application.

المنارة للاستشارات
www.manaraa.com

Chapter 5

RELATED WORK

The work related to the idea of generating low-latency early estimates for window-query results followed by accurate final results can be broadly classified into the following categories: (a) stream-relational or stream-warehouse systems that handle low-latency streaming query answers while also maintaining historical integrity, (b) systems that emit low-latency results followed by possible retractions or negative tuples, (c) DSMSs that handle bursts through *load shedding*, and (d) the research in DBMSs on producing running aggregation-query–result estimates together with confidence intervals for the estimates. I consider these four categories of related work, one in each section of this chapter.

5.1 STREAM-RELATIONAL AND STREAM-WAREHOUSING SYSTEMS

Krishnamurthy *et al.* describe Truviso's [16] technique to handle very late data by initially computing partial results, and then consolidating the partials into final results. For continuous queries, the partial-to-final consolidation occurs immediately, whereas for static queries over tables, the partials are consolidated into final results lazily, i.e., when a query demands. Late consolidation allows Truviso to maintain *historical integrity* by accounting for substantially late data. Truviso processes all input data by placing them into appropriate partitions and computing partial results over individual partitions. Data that arrive after their target partition has closed are processed by the system in a new partition. The system processes all the partitions in parallel, using separate instances of a query plan. Therefore, Truviso can serve newly arriving data and simultaneously process very late data. To limit the total number of active partitions, a background

المنارة للاستشارات
www.manaraa.com

process periodically runs over partials to roll up several partial results into a single partial result.

What relates the prodding technique, which I have described in this thesis, to the Truviso approach is the similarity between early and final results produced in prodding and the partial and final results produced by Truviso. Whereas Krishnamurthy *et al.* go on to describe the Truviso system architecture, which emphasizes use of parallel processing wherever possible, the work in this thesis describes in detail the semantics of prod processing by the prodding-enabled aggregate operators. The proposed prodding semantics allow a source (of prods) to prod an entire query plan, thereby including contribution from partial states across a query plan in early-result computation. Moreover, the prodding technique also opens up ways to counter bursts, by triggering their sidelining inside an expensive part of a query plan. (Section 6.2 details the idea.) Further, the prodding technique does not require parallel-processing capabilities. A limitation in the prodding technique, and perhaps in all the systems that do not handle very late tuples as in Truviso, is that they need to rely on grammatical nature of data streams with respect to punctuations to ensure (100%) accuracy of the produced final results.

DataDepot [9], a stream-warehousing system, deals with bursty data arrivals by prioritizing the updates of (user-specified) important tables to keep them as fresh as possible. The system deals with delayed data by taking a conservative approach in query answering, i.e., by computing query answers by including only the data records that are below the low-water-mark, or the "safe trailing edge" as DataDepot refers to it. A safe trailing edge is determined by punctuations in a data stream. Note that with window queries, such a conservative approach would mean that a query cannot produce a result for a window if part of the window lies above the low-water-mark. In such a situation, use of prodding would allow the system to produce early estimates of query results. As we saw in Chapter 4, such early estimates in many cases can be substantially accurate, especially for sliding-window queries. Moreover, the accurate final results produced on arrival of corresponding punctuations would help in maintaining the historical integrity in the data archive.

www.manaraa.com

5.2 SYSTEMS SUPPORTING RETRACTIONS OR NEGATIVE TUPLES

In their work on CEDR [4], Barga *et al.* emphasize the need to produce DSMS query output without blocking for complete (pertinent) data to arrive. They propose producing such output, and, if needed, retracting the previous output and inserting the correct revised results. To model and handle such retractions, CEDR uses a tritemporal model consisting for each event a system time (also called CEDR time) in addition to the event's occurrence and validity times. Using the notion of *canonical history tables*, which are based on this tritemporal model, authors define three levels of consistency — strong consistency, middle consistency, and weak consistency — having (from strong to weak levels) decreasing result production latencies, but increasing chances of retractions. To allow DSMS users to achieve latency-consistency trade-offs, CEDR enables its operators with two components: (a) a consistency monitor and (b) an operational module. The consistency monitor determines for how long to block the input in the operator buffer so as to maintain the desired level of consistency, whereas the operational module performs the regular operator function. By adjusting the buffer memory and the blocking to specific lengths of time, CEDR users can obtain a spectrum of consistency levels. Prodding allows a DSMS to achieve latency-accuracy balance in a cleaner fashion, by having a prod specify the exact desired time to see the query results. Moreover, a prod can denote in its predicates the exact substream for which the system should emit a low-latency result. Such a fine level of control is not offered in CEDR. Another subtle difference between retractions in CEDR and prodding is that a retraction will occur only if there are late tuples that can change the previously produced results, whereas a prodded result will always have a corresponding final result even if the two results are the same.

Hammad *et al.* describe use of *negative tuples* to free sliding windows in a DSMS from relying for progress on external input tuples or punctuations [11]. The authors propose using a special operator, which they call Expire. The Expire operator resides at the beginning of a query plan, and for each tuple, generates a negative tuple to indicate when that tuple expires from a window. Because the use of negative tuples generated by the Expire operator is purely based on time, Expire frees the system from

www.manaraa.com

relying on external input (tuples or punctuations) to ensure progress of sliding windows. Although prods do not expire any tuples or close any windows, a source of prods does free the system from relying on the arrival of tuples or punctuations in a data stream for producing window query results. Furthermore, a user may choose to be aggressive in prodding, and ask for early estimates to window query results, possibly even before the application time at which a window should close. Accommodating such aggressive demands of early query results in the negative tuple approach is unintuitive, because it would require invalidating tuples before their appropriate expiration time.

5.3 LOAD SHEDDING

Tatbul *et al.* (using the Aurora DSMS) [25] and Babcock *et al.* (using the STREAM DSMS) [3] have proposed load-shedding techniques to handle data-arrival bursts in a DSMS. Aurora, on encountering a data burst that overwhelms the processing capacity of the system, drops a subset of windows, thereby producing a subset of the accurate results (also called subset results). The system uses a WindowDrop operator at the beginning of a window-query plan. In a bursty situation, WindowDrop probabilistically decides which (entire) windows to drop. To ensure subset results, WindowDrop marks each tuple with a flag indicating whether or not the tuple can form a new window. The goal of WindowDrop is to ensure generation of maximum subsets of original query answers when the system encounters overwhelming data bursts. Tatbul *et al.* contrast the subset-result approach with load shedding on the individual-tuple level. The authors point out that such tuple-level load shedding leads to production of erroneous results, whose impact may propagate downstream in the query plan.

The load-shedding technique in STREAM uses random-sampling operations over input data. To perform sampling, the system uses load-shedding operators, called *load-shedders*. Each load-shedder is parameterized with a sampling rate, which is the probability for including a tuple in a sample at the load-shedder. STREAM determines where to place the load-shedders and how to set the sampling rate at each of them based on

المنارة للاستشارات
www.manaraa.com

the statistics about the data streams such as the observed stream-arrival rates and operator selectivities. To compensate for the aggregates calculated after dropping tuples, the aggregate values are scaled appropriately to produce unbiased approximate answers.

Prodding currently handles bursts by providing early results using the part of the window data that the system can process in a timely manner. The technique produces accurate final answers with unavoidable delays that are caused by bursts. As future work, I propose sidelining a burst, and processing it at the time of lulls in the system so that the effects of busrts on the result latencies do not propagate to the subsequent non-bursty windows. I further describe the idea in Section 6.2.

5.4 ONLINE AGGREGATION

The database counterpart that the prodding technique resembles is the work on online aggregation by Hellerstein *et al.* [12]. The online aggregation technique enables users to control latency-accuracy tradeoffs in results for aggregate queries over large data sets by letting the users observe and control query execution through an online interface. The interface gives a running estimate of the aggregate along with a confidence interval and the percentage query completion. A user can choose to stop the query execution at any time, say based on whether aggregate estimates are produced within user's desired confidence intervals. Thus, users can control the latency-accuracy trade-off for the query answers on the fly. The online-aggregation interface starts displaying results after processing the first tuple and updates the results at a time interval comfortable for a human observer. Hellerstein *et al.* propose formulas to compute the confidence intervals that are applicable to a wide variety of aggregate functions. Online aggregation also achieves important goals, such as facilitating users' control of relative progress among groups and minimizing time to first "useful" estimates.

The online aggregation system does not compel users to pre-specify statistical measures, thereby eliminating the need for users to perform statistical computations in advance. The system obtains statistically meaningful estimates by ensuring random access to the data. Such random data access is ensured by using techniques such as heap scans,

المنارة للاستشارات
www.manaraa.com

index scans and sampling from indexes, which are feasible in case of stored data. To ensure fairness in groupwise result generation, online aggregation may employ techniques such as hybrid hashing and index striding, which again are available for use over database systems.

Such control over data access is not possible in case of data streams, wherein the query result computations may be running even before the entire data needed have arrived at the system. Therefore, in DSMSs, it is difficult to provide a statistically meaningful running confidence interval for aggregate-query result estimates. For future work, I propose displaying a running measure of a DSMS's overall accuracy of early results over several windows. Such a measure would allow a user to control the latency-accuracy balance by tuning prodding aggressiveness. I describe this idea in detail in the next chapter (Section 6.2). Although the prodding technique allows users to pre-specify the measure of prodding aggressiveness, such pre-specification by a human user is not mandatory, because a prodding-enabled DSMS is agnostic of the sources of prods.

المنارة للاستشارات
www.manaraa.com

Chapter 6

CONCLUSION AND FUTURE WORK

This research has addressed the low-latency result needs of data-stream applications. Although the focus of the work has been window-aggregate queries over data streams, the prodding technique introduced here is extensible to data stream queries in general. This concluding chapter contains both the summary of the work and possible take-home messages from this thesis.

In Section 6.1, I describe some key challenges I faced during this research. I believe that considering these challenges would be useful for a future researcher who takes up work similar to that in this thesis. I hint at some possible directions for further research in Section 6.2. Finally, in Section 6.3, I sum up the work in this thesis with some concluding remarks.

6.1 CHALLENGES

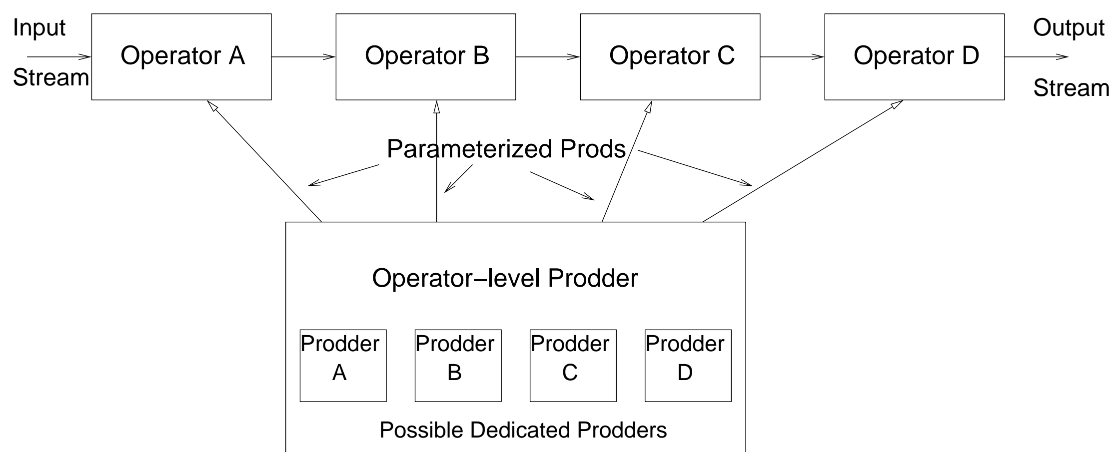


Figure 6.1: Operator-level Prodding

www.manaraa.com

Key aspects to consider in taking up work on "low-latency DSMS query-result estimates" are: synchronization among query operators, streaming data delivery, and measuring result-estimate accuracy.

Synchronization: In Chapter 3 (Sections 3.1 and 3.3), we saw why it is desirable to prod not just an operator but an entire query plan. We saw how the prodding technique lets a source (of prodding) prod an entire query plan — a prod travels across a query plan, and sweeps ahead of itself the contributions from states across multiple query operators. Here I describe an alternative approach to the design of prodding machinery, which I considered as a candidate.

The idea in the alternative approach is to prod all the operators separately. Figure 6.1 shows how such an architecture would look. The example query plan shown in the figure has four operators A, B, C, and D, and prodding machinery that prods each operator individually. Note that the important issue in such a design is to synchronize the prods in such a way that when an operator is prodded for early results, it has already received the collective early contribution from all the operators upstream of it. Achieving such synchronization would be challenging. Suppose all the prods are activated by the centralized prodding machinery at once, and these prods reach all the operators at the same time. In such a situation, none of the prodded operators will have any contribution available from its upstream operators (because all the operators are *just* prodded). A possible improvement would be to slightly delay the prods to each successive operator. But it would be hard to tell with any guarantee what the delay should be, because the time for interoperator transfer of tuples is not predictable. The main factors affecting such interoperator tuple-transfer time are operator scheduling and the interoperator buffer length.

Streaming data delivery: A typical way to test a DSMS's operation is to read an input data stream from a file. Such an input stream may be generated by a test-data generator, which timestamps and punctuates data according to the testing needs. For example, Microsoft's StreamInsight team uses an event generator to produce test data streams according to declarative specifications by a tester [20]. To accommodate a tester's desired temporal properties in a generated data stream, StreamInsight's event

www.manaraa.com

generator augments generated tuples with suitable application timestamps. The event generator ensures time progress in a generated data stream by injecting CTI (current time increment, similar to a punctuation) events in the data stream. Such an approach of relying on application timestamps and CTIs is useful to test the correctness of DSMS behavior. But when the tester's intention is also to study a DSMS's output latency, which may be influenced by data delivery aspects — viz., delays and bursts, it is essential to simulate data delivery to the DSMS in streaming fashion.

Observing a DSMS's output latency over data streams subject to delivery delays or bursts has been a central aspect of the work in this thesis. Therefore, it was important to simulate such aspects in the data delivery to a DSMS. The varying densities in application timestamps in my test data streams — obtained from real-world applications and programmatically generated — represented data bursts. Further, the effects of these data bursts were simulated by the use of a Streamer operator (introduced in Section 4.1), which fed the data streams to the NiagaraST prodding prototype according to the application timestamps of the tuples. To simulate the effects of data delivery delays, the Streamer takes the average tuple-delivery delay as a parameter and delivers tuples with the specified delays.

Accuracy of the estimates: Once the mechanism to obtain early estimates to DSMS window-query results is in place, the next step is to measure the accuracy of the estimates obtained. To measure the accuracy of a result estimate, the estimate needs to be compared with the true final result. Making such comparison at the time of early-result generation is not possible, because the final result is not yet known. Moreover, even if such comparison is made when the final result becomes available (as described in sections 3.4, 4.4.2 and 6.2), it is not straightforward to talk about an estimate's accuracy based on the early-final result comparison; aggregate result values and their interpretation typically will be application dependent. An approach might be to facilitate specification of a *user-defined function* that compares two values — an estimate and a final, and returns a quality level for the estimate. When computing the quality level, such a function can also consider the time gap between the production of the early and the final aggregate values being compared.

المنارة للاستشارات

www.manaraa.com

6.2 FUTURE WORK

This thesis has opened up a few ideas that can be taken up for research in the future. I describe three such ideas in this section.

(a) Deprioratized burst processing: After emitting an early query result for a window, a prodding-enabled DSMS continues processing the trailing portion of the data that belong to the window. In case of a data-burst–affected (or, simply, a burst-affected) windows, this trailing portion can be large, causing high-latency final-result production for the window. Such high latency in producing the final result for one window implies that the DSMS will start processing the subsequent window(s) late (assuming absence of DSMS-dedicated parallel processing capabilities), thereby possibly delaying their final result output even though these subsequent windows do not have high data density. To address this issue, prodding can be empowered with a new tuple-processing strategy that prevents (final result) delay effects of a data burst on one window from cumulatively spreading to subsequent non-burst-affected windows. The idea is to sideline the part of the data after the prod in a burst-affected window and start processing the data from the next window. The sidelined part of the burst can be processed when the system has excess processing capacity, say, during an input lull. Figure 6.2 shows a query plan similar to the one in Figure 4.3, which we saw in connection with the burst experiments described in Section 4.3.2. Figure 6.2 shows the added *burst handling logic*, which sidelines, at the *expensive* query subplan, the portion of data in the burst-affected window that follows a prod for the window.

Figure 6.3 shows early and final result curves for the query Q4-2 (from Section 4.3.2) run without (Figure 6.3(a)) and with (Figure 6.3(b)) the burst-handling logic. Notice the circled regions in the two graphs. When the trailing portion of data in a burst-affected window is sidelined, the latency in final results for the several following windows is low. In other words, the use of burst handling logic has flattened (or punctured) the burst. Moreover, because the final results in the windows subsequent to the burst-affected one are not cumulatively delayed, early results are needed less frequently (notice in the Figure 6.3(b) that the number of early results (solid triangles) is much lower than that in Figure

المنارة للاستشارات
www.manaraa.com

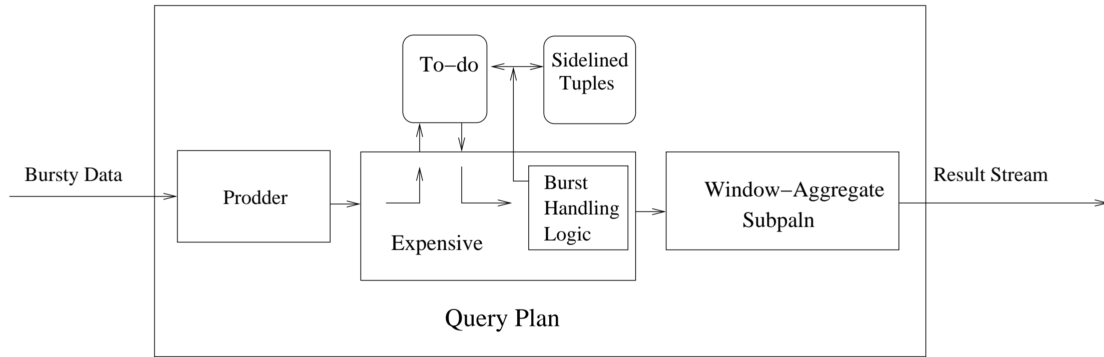
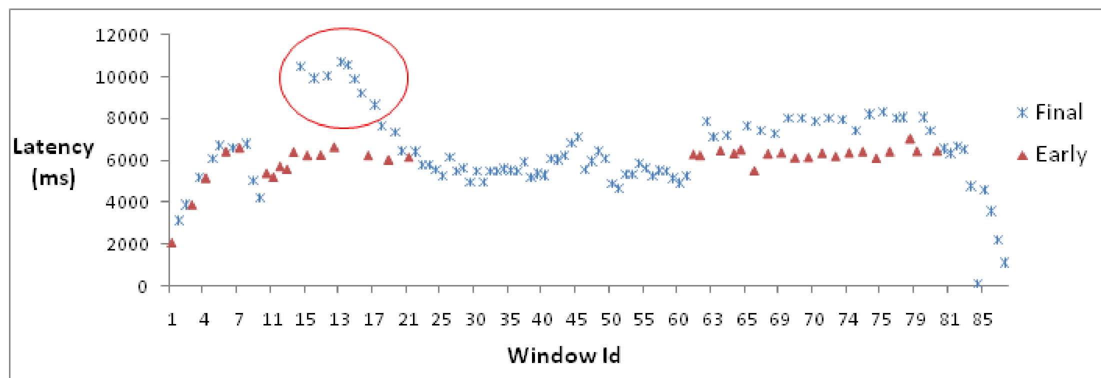
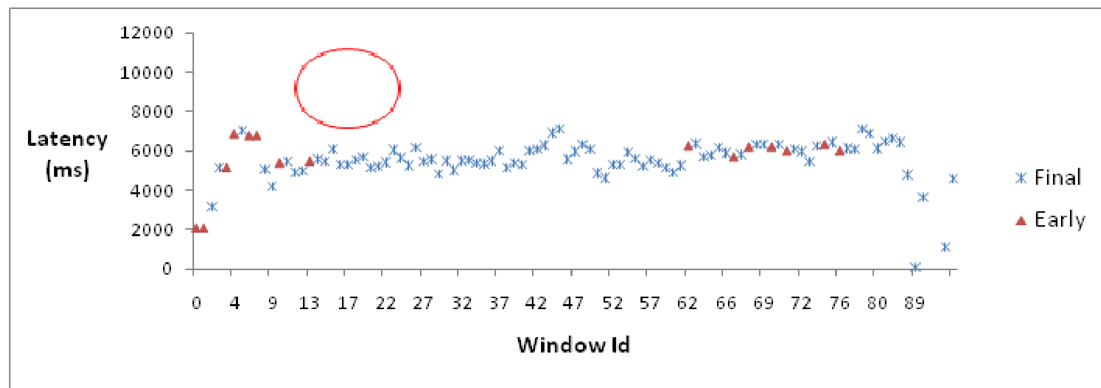


Figure 6.2: Sidelining Portions of Bursty Data



(a) Burst Processing Continued After Prod-Processing



(b) Burst Processing Canceled After Prod-Processing

Figure 6.3: Burst Processing Strategies

المنارة للاستشارات
www.manaraa.com

6.3(a)).

Several aspects of the aforementioned burst-processing strategy need further investigation. For one, Figure 6.3(b) shows the results of an implementation wherein the system never processes the sidelined portion of the bursts, in effect dropping the sidelined tuples; thus, the implemented burst-processing technique reduces to load-shedding. The algorithm to handle the sidelined portion of tuples at the expensive portion (or subplan) of a query seems straightforward. Whenever there is no other tuple to process at Expensive, pick the next tuple from the sidelined list and process it. But if the system is at least moderately busy for a decent length of time, the burst-affected windows can be starved for the production of their final results. Some strategy should be explored to avoid an indefinite hold on the sidelined portion of tuples, maybe by processing such tuples with a low priority. For example, the system could be parameterized to process a sidelined tuple after processing every 3 new tuples. Such a strategy may be termed *deprioratized burst processing*.

Another aspect to consider is the grammaticality of results (as discussed in the Section 2.2). Until the sidelined burst is processed, the corresponding punctuation cannot be released. Furthermore, to maintain grammaticality of the query output, none of the subsequent punctuations can be released, because tuples with older timestamps are yet to be processed. Therefore, even though the final results for the windows subsequent to a burst may be ready, these results cannot be emitted by the system. The challenge in maintaining the grammaticality of the query output may be handled by altering the interpretation of punctuations in the system to *range punctuations*, as described by Tucker in his dissertation [27]. A range punctuation on an attribute, say the timestamp attribute, describes a range (rather than just an upper bound) of timestamps between which no future tuples will be seen by the system. Use of range punctuations will allow a DSMS to emit final results (and punctuations) for a non-burst–affected window, while the system has held up data and punctuations for an earlier burst-affected window. The resulting out-of-order production of punctuations will not make the result stream ungrammatical, because a range punctuation covers only a certain range of (say) timestamps rather than all timestamps below a certain watermark.

المنارة للاستشارات

www.manaraa.com

Assessing the suitability of burst sidelining together with the use of range punctuations is a topic for future investigation. The burst-sidelining strategy will be very useful in applications where, once the early output for a window is used to answer a streaming query, the corresponding final result is not immediately needed, but is eventually required for the historical integrity of data archives.

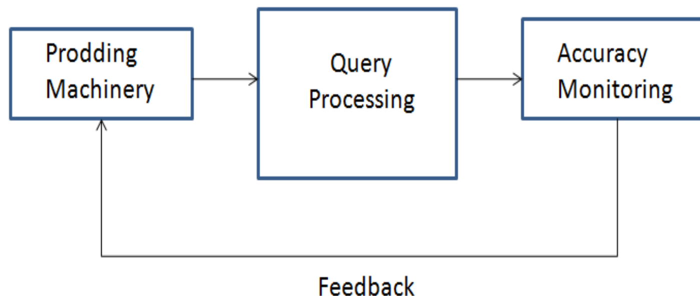


Figure 6.4: Monitoring Running Accuracy

(b) Prioratizing substream results: Application of prods for prioritizing early outputs of subgroups or substreams using the predicate field(s) inside a prod should be studied further. Although the semantics that I describe in Chapter 3 cover such use of prods, the benefits of prodding obtained by specifying complex patterns inside a prod predicate should be tested over data streams from different real-world applications.

(c) Accuracy Monitoring: As we saw in Section 5.4, in a DSMS, it is not possible to produce a confidence interval for early results using the statistical techniques described by Hellerstien *et al.* in their work on online aggregation [12]. In DSMS applications, such confidence intervals would let the users tune the prodding aggressiveness (P_{Aggr}) for a running query: increase P_{Aggr} if the result confidence is adequate and more latency gains are desirable or decrease P_{Aggr} if the confidence intervals are not satisfactory. An alternative technique that seems feasible for DSMSs is to compute the accuracies of early results (by comparing them with the corresponding final results when the finals are produced), and to monitor the overall running accuracy of the system for a given P_{Aggr}. An application user can then assess the ongoing overall accuracy, and if needed, adjust the P_{Aggr}. More ambitiously, such adjustment can be automated via a feedback

المنارة للاستشارات
www.manaraa.com

mechanism. Figure 6.4 gives a schematic of the feedback idea.

6.3 CONCLUDING REMARKS

Because of the inherent possibility of delays or bursts in data streams, a DSMS application cannot control the result latency of window queries over data streams without potentially compromising the accuracy of results. What a DSMS can control is the latency-accuracy tradeoff for the results. Using the prodding technique, a DSMS can generate window query result estimates — whose latency can be controlled by the system — followed by accurate final window query results.

The prod-processing mechanism ensures that partial states accumulated across a query plan contribute to the computation of early result estimates. In other words, the proposed prodding technique sweeps an entire query plan for a result, and not just the topmost aggregate operator (in the plan).

The prodding prototype developed in NiagaraST has been useful for evaluating the benefits of early results — I measured the latency gains and the accuracy of early query results over data streams having different data-arrival patterns and different patterns in the aggregate-attribute values.

Based on the results observed in my experiments, I believe that the prodding technique can be beneficial in various scenarios — the technique can produce early query-result estimates whose utility can be optimized by tuning the prodding aggressiveness. I remark that such tuning should be done considering the specific latency and accuracy needs of business, and if available, knowledge about patterns in the incoming data streams.

www.manaraa.com

REFERENCES

[1] Intelligent Transportation Systems Laboratory at Portland State University http://portal.its.pdx.edu.

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

[3] Babcock, B., Datar, M., and Motwani, R. Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the 20th International Conference on Data Engineering* (Washington, DC, USA, 2004), ICDE '04, IEEE Computer Society.

[4] Barga, R. S., Goldstein, J., Ali, M., and Hong, M. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *In CIDR* (2007), pp. 363–374.

[5] Chakravarthy, S., and Jiang, Q. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing.* Springer Publishing Company, Incorporated, 2009.

[6] Cormode, G., Johnson, T., Korn, F., Muthukrishnan, S., Spatscheck, O., and Srivastava, D. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2004), SIGMOD '04, ACM, pp. 35–46.

[7] Ethier, S. N. *The Doctrine of Chances: Probabilistic Aspects of Gambling*. Probability and Its Applications. Springer, Berlin and Heidelberg, 2010.

المنارة للاستشارات
www.manaraa.com

[8] Fernández-Moctezuma, R., Tufte, K., and Li, J. Inter-Operator Feedback in Data Stream Management Systems via Punctuation. In *CIDR* (2009).

[9] Golab, L., Johnson, T., Seidel, J. S., and Shkapenyuk, V. Stream warehousing with DataDepot. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data* (New York, NY, USA, 2009), ACM, pp. 847–854.

[10] Golab, L., and Özsu, M. T. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29* (2003), VLDB '2003, VLDB Endowment, pp. 500–511.

[11] Hammad, M., Aref, W. G., Franklin, M. J., Mokbel, M. F., and Elmagarmid, A. K. Efficient Execution of Sliding-Window Queries Over Data Streams, 2003.

[12] Hellerstein, J. M., Haas, P. J., and Wang, H. J. Online aggregation. *SIGMOD Rec. 26* (June 1997), 171–182.

[13] Johnson, T., Muthukrishnan, S., Shkapenyuk, V., and Spatscheck, O. A heartbeat mechanism and its application in gigascope. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 1079–1088.

[14] Kang, J., Naughton, J. F., and Viglas, S. D. Evaluating Window Joins over Unbounded Streams. In *In ICDE* (2003).

[15] Kotz, D., Henderson, T., Abyzov, I., and Yeo, J. CRAWDAD data set dartmouth/campus (v. 2009-09-09). Downloaded from http://crawdad.cs.dartmouth.edu/dartmouth/campus, Sept. 2009.

[16] Krishnamurthy, S., Franklin, M. J., Davis, J., Farina, D., Golovko, P., Li, A., and Thombre, N. Continuous analytics over discontinuous streams. In

المنارة للاستشارات

www.manaraa.com

SIGMOD '10: Proceedings of the 2010 international conference on Management of data (New York, NY, USA, 2010), ACM, pp. 1081–1092.

[17] Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record 34* (2005), 2005.

[18] Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2005), ACM, pp. 311–322.

[19] Naughton, J., Dewitt, D., Maier, D., Aboulnaga, A., Chen, J., Galanis, L., Kang, J., Krishnamurthy, R., Luo, Q., Prakash, N., Ramamurthy, R., Shanmugasundaram, J., Tian, F., Tufte, K., and Viglas, S. The Niagara Internet Query System. *IEEE Data Engineering Bulletin 24* (2001), 27–33.

[20] Raizman, A., Ananthanarayan, A., Kirilov, A., Chandramouli, B., and Ali, M. An extensible test framework for the Microsoft StreamInsight query processor. In *Proceedings of the Third International Workshop on Testing Database Systems* (New York, NY, USA, 2010), DBTest '10, ACM, pp. 2:1–2:6.

[21] Ramakrishnan, R., and Gehrke, J. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2003.

[22] Silberschatz, A., Korth, H. F., and Sudarshan, S. *Database Systems Concepts*. McGraw-Hill Higher Education, 2001.

[23] Srivastava, U., and Widom, J. Flexible time management in data stream systems. In *PODS '04: Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (New York, NY, USA, 2004), ACM, pp. 263–274.

المنارة للاستشارات
www.manaraa.com

[24] Tatbul, N., and Zdonik, S. Window-aware load shedding for aggregation queries over data streams. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases* (2006), VLDB Endowment, pp. 799–810.

[25] Tatbul, N., and Zdonik, S. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32nd international conference on Very large data bases* (2006), VLDB '06, VLDB Endowment, pp. 799–810.

[26] Torsten Grabs, Roman Schindlauer, R. K. J. G. Introducing Microsoft StreamInsight. White paper, Microsoft SQL Server 2008 R2 StreamInsight, September 2009. Available online (26 pages).

[27] Tucker, P. A. *Punctuated Data Streams.* PhD thesis, 2005. AAI3184366.

[28] Tucker, P. A., Maier, D., Sheard, T., and Fegaras, L. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering 15* (2003), 2003.

المنارة للاستشارات
www.manaraa.com