

Punctuated Data Streams

Peter A. Tucker

A dissertation submitted to the faculty of the
OGI School of Science & Technology at
Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

August 2005

© Copyright 2005 by Peter A. Tucker
All Rights Reserved

The dissertation "Punctuated Data Streams" by Peter A. Tucker has been examined and approved by the following Examination Committee:

David Maier
Professor
Portland State University
Thesis Research Adviser

Lois M. L. Delcambre
Professor
Portland State University

Tim Sheard
Associate Professor
Portland State University

Jennifer Widom
Professor
Stanford University

Mark P. Jones
Associate Professor
OGI School of Science and Engineering
at OHSU

Dedication

To my grandfather.

Acknowledgements

I first thank my advisor, David Maier. His guidance has been invaluable. I enjoyed our discussions over the past six years on many topics, including data management, teaching techniques, and Mariners baseball. I am very fortunate to have Dave as my advisor. I also want to thank the other members of my thesis committee: Lois Delcambre, Tim Sheard, Jennifer Widom, and Mark Jones. Their comments and encouragement throughout this work have been wonderful. Lois has been on my committee with Dave from the beginning, and I enjoyed our many discussions, particularly those regarding our interest in teaching. Tim brought of interest from the functional programming perspective, and has helped to formulate much of the framework we developed. Jennifer's support for our work and comments throughout have been really appreciated. Mark showed interest in this work from an outside perspective and was often available to discuss ideas.

The "Niagara West" team in Portland was a wonderful group to work in, particularly Vassilis Papadimos, Kristin Tufte, and Jin Li. I enjoyed our visits to the Mad Greek Deli and our discussions of topics including and beyond our research.

The "Niagara East" team, including David DeWitt, Jeff Naughton, Raghu Ramakrishnan, Leonidas Galanis and Stratis Viglas offered many suggestions and much support for this work. I also thank Leonidas Fegaras and Johannes Gehrke for their suggestions and ideas throughout this work.

The people at Whitworth College gave me time and grace as I completed this work during my first two years there. Particularly, I want to thank Kent Jones, Susan Mabry, Lyle Cochrane, and Donna Pierce for their encouragement and understanding.

Finally, I want to thank my wife, my children, and my parents for all of their support and love. I could not have completed this without their help.

Portions of this work were originally published in shortened form in IEEE Conference

of Data Engineering [TM02], IEEE Data Engineering Bulletin [TM03a], IEEE Transactions of Knowledge and Data Engineering [TMSF03], the Workshop on Management and Processing of Data Streams [TM03b], and Stream Data Management [MTG05]. Other portions have been or will be submitted for consideration. Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF ITR award IIS0086002.

Contents

Dedication	iv
Acknowledgements	v
Abstract	xvi
1 Introduction	1
1.1 Motivating Examples	2
1.1.1 Warehouse Monitoring System	3
1.1.2 Network Traffic Monitor	4
1.1.3 Online Auction	5
1.1.4 Issues Raised from the Example Queries	7
1.2 Advantages of Using DBMS-Style Queries over Data Streams	8
1.3 Our Proposed Solution — Punctuated Data Streams	10
1.3.1 Punctuation Behaviors for Query Operators	11
1.3.2 Effects of Punctuation for Entire Queries	11
1.4 Contributions of this Dissertation	12
1.5 Structure of this Document	14
2 Background on Relational Database Management Systems and Data Stream Processing	15
2.1 Brief Overview of Query Operators	15
2.1.1 Unary Query Operators	15
2.1.2 Binary Query Operators	17
2.1.3 Building Complex Queries from Individual Operators	19
2.2 Approaches to Querying Over Continuous Data Streams	19
2.2.1 Positional and Ordered Data	22
2.2.2 Windows	23
2.3 Using Various Stream Approaches in Specific Applications	24
2.3.1 Warehouse	24
2.3.2 Network	27

2.3.3	Auction	28
2.4	Summary	30
3	Initial Investigation of Punctuated Streams	31
3.1	Initial Definition of Punctuation Rules	32
3.1.1	Pass Rules	33
3.1.2	Purge Rules	35
3.1.3	Propagation Rules	37
3.2	Implementation Details	39
3.2.1	Format for Punctuation	39
3.2.2	Punctuation Patterns	40
3.2.3	Definition of <i>match</i>	41
3.2.4	Implementation of Rules in Query Operators	42
3.3	Initial Results	45
3.4	Research Directions	49
4	Overview of Punctuation Semantics	51
4.1	Stream Iterators	51
4.2	Representation Issues	53
4.3	Enhancing Our Model of Streams and Stream Iterators	54
4.3.1	An Enhancement for Streams	54
4.3.2	An Enhancement for Stream Iterators	55
4.4	Including Punctuation Semantics in our Stream Iterator Model	57
4.5	Punctuation Representation and Manipulation	58
4.6	Punctuation-Aware Stream Iterators	61
5	A Framework for Stream Iterators	64
5.1	A Formulation for Stream Iterators	66
5.2	Implementation of Punctuations	71
5.3	Implementation of Punctuation-Aware Stream Iterators	74
5.4	Evaluation of Our Framework	80
5.5	Summary	88
6	Theory of Punctuated Streams	89
6.1	Faithfulness and Propriety	90
6.2	Pass Invariants	93
6.3	Propagation Invariants	94
6.4	Keep Invariants	96

6.5	The Minimality Condition	97
6.6	Proving Correctness of Stream Iterators	97
6.6.1	Stream Iterator for Duplicate Elimination	97
6.6.2	Stream Iterator for Difference	100
6.7	Discussion	110
7	Implementation Issues Regarding Punctuation Semantics	111
7.1	Embedding Punctuations in Data Streams	112
7.2	Overview of the Niagara Query Engine	113
7.2.1	Representation of Data Items	114
7.2.2	Query Operators	116
7.3	General Enhancements to Niagara	119
7.3.1	New Punctuation Class	120
7.3.2	Base Class Modifications	121
7.4	Enhancements to Specific Niagara Operators	121
7.5	The Describe Operator	127
7.5.1	Defining Describe and its Punctuation Invariants	128
7.5.2	Implementation of Describe	129
7.6	The Punctuate Operator	131
7.7	Dealing with Disorder	134
7.7.1	Querying Over “Nearly-ordered” Attributes	136
7.7.2	Maintaining Order	137
7.8	Summary	139
8	Performance Results	140
8.1	Scenario	140
8.1.1	Streams for the Online Auction Monitoring System	141
8.1.2	Queries for the Online Auction Monitoring System	141
8.2	Generating Stream Contents	145
8.2.1	Data Generation	146
8.2.2	Generating Punctuations	148
8.3	Performance Results	148
8.3.1	Test Configurations	149
8.3.2	Discussion of Performance Results	151
8.3.3	Comparison with Slack	154
8.4	Summary	157

9	Taking Punctuations Beyond Single Query Operators	161
9.1	Motivating Example	163
9.1.1	Example Queries	164
9.1.2	Solving the Example Queries with Punctuations	165
9.2	Groupings and Punctuation Schemes	166
9.2.1	Groupings for Dataspaces and Groupings of Interest	167
9.2.2	Punctuation Schemes	168
9.2.3	Overview of Our Approach	169
9.3	Unblocking Query Operators with Punctuation Schemes	171
9.3.1	Punctuation Scheme Assignments	172
9.3.2	Unblocking Unary Query Operators	173
9.3.3	Enabling Punctuations From Unary Operators Using Input Punctuation Schemes	175
9.3.4	Unblocking Binary Query Operators	178
9.3.5	Enabling Punctuations From Binary Operators Using Input Punctuation Schemes	179
9.4	Cleansing Query Operators with Punctuation Schemes	181
9.4.1	Modelling State for Algorithms for Traditional Query Operators	182
9.4.2	Cleansing Unary Operators with Punctuation Schemes	184
9.4.3	Cleansing Binary Operators with Punctuation Schemes	187
9.5	Using Punctuation Schemes to Benefit Specific Queries	190
9.5.1	Verifying Punctuation Schemes Benefit Example Queries	190
9.5.2	Analyzing Queries to Generate Punctuation Scheme Assignments	193
9.6	Summary	195
10	Related Work	197
10.1	Other Kinds of Stream Semantics	197
10.2	Approaches for Handling Large or Unbounded Data Inputs	198
10.2.1	Redefining Blocking Operators to Have Non-Blocking Definitions	198
10.2.2	Ordered Input	199
10.2.3	Windows	200
10.2.4	Approximation	201
10.2.5	Load Shedding	202
10.2.6	Bounded Memory Queries	203
10.2.7	Incomplete Query Processing	203
10.3	Data Stream Query Implementation Issues	204
10.3.1	Operator Algorithms	204

10.3.2	Optimization Techniques	205
10.4	Data Stream Management Systems	206
10.4.1	Models for Data Stream Processing	207
10.4.2	Data Stream Management Languages	210
10.4.3	Benchmarks for Data Stream Management Systems	211
11	Conclusions	212
11.1	Punctuation Semantics	213
11.2	Framework and Theory for Punctuations	214
11.3	Implementation and Performance	215
11.4	Benefiting Entire Queries	215
11.5	Future Work	216
11.5.1	Query Optimization Issues	216
11.5.2	Query Execution Issues	218
11.5.3	Applying Punctuations in Real-World Application Domains	219
11.5.4	Other Semantics for Punctuations	220
11.5.5	Other Methods that Benefit Queries	221
11.6	Concluding Remarks	222
Bibliography		223
A	Proofs of Correctness for Various Relational Operators	236
A.1	Stream Iterator for Select	236
A.1.1	Faithfulness of the Invariants for Select	237
A.1.2	Propriety of the Invariants for Select	237
A.1.3	Conformance of the Implementation of Select to its Invariants	238
A.2	Stream Iterator for Project	238
A.2.1	Faithfulness of the Invariants for Project	240
A.2.2	Propriety of the Invariants for Project	240
A.2.3	Conformance of the Implementation of Project to its Invariants	240
A.3	Stream Iterator for Group-By	242
A.3.1	Faithfulness of the Invariants for Group-By	244
A.3.2	Reducing State Required for Group-By	245
A.3.3	Propriety of the Invariants for Group-By	246
A.3.4	Conformance of the Implementation of Group-By to its Invariants	246
A.4	Stream Iterator for Sort	249
A.4.1	Faithfulness of the Invariants for Sort	250
A.4.2	Reducing State Required for Sort	251

A.4.3	Propriety of the Invariants for Sort	252
A.4.4	Conformance of the Implementation of Sort to its Invariants	252
A.5	Stream Iterator for Merge	255
A.5.1	Faithfulness of the Invariants for Merge	256
A.5.2	Propriety of the Invariants for Merge	257
A.5.3	Conformance of the Implementation of Merge to its Invariants	257
A.6	Stream Iterator for Intersect	259
A.6.1	Faithfulness of the Invariants for Intersect	260
A.6.2	Propriety of the Invariants for Intersect	262
A.6.3	Conformance of the Implementation of Intersect to its Invariants	262
A.7	Stream Iterator for Join	265
A.7.1	Faithfulness of the Invariants for Join	267
A.7.2	Propriety of the Invariants for Join	268
A.7.3	Conformance of the Implementation of Join to its Invariants	269
B	Source Code for the Online Auction Management System	272
B.1	Data Types for the Auction Scenario	272
B.2	Implementation of Queries in the Online Auction Scenario	273
B.2.1	Query 1 — Currency Conversion	273
B.2.2	Query 2 — Specific Bid Ranges	274
B.2.3	Query 3 — Bid Counts	275
B.2.4	Query 4 — Closing Price for Auctions in Specific Categories	275
B.2.5	Query 5 — Union of Bid Counts	276
B.3	Implementation of Online Auction Streams	277

List of Tables

4.1	Initial definitions for stream iterators	53
4.2	State function (r) definitions of stream iterators. Note that now we will use a duplicate-preserving implementation of union.	57
4.3	Output function (q) definitions of stream iterators.	57
4.4	Punctuation behaviors for the stream iterator dupelim.	58
4.5	Helper functions used on list of data items and punctuations.	61
6.1	Pass invariants for traditional query operators.	93
6.2	Propagation invariant for traditional query operators.	95
6.3	Keep invariants for traditional query operators.	96
8.1	The data sets used in our performance tests.	149
8.2	Data set characteristics for Query 1.	151
8.3	Data set characteristics for Query 2 and Query 3.	151
8.4	Data set characteristics for Query 4.	153
8.5	Data set characteristics for Query 5.	156
9.1	TCP/IP header fields, as defined in the IP RFC and the TCP RFC.	163
9.2	Definitions for <i>preimage</i> for unary operators.	174
9.3	Pass invariants for stream operators.	176
9.4	Definitions for <i>preimage</i> for binary operators.	179
9.5	State models for various implementations of query operators.	184
9.6	Non-trivial keep invariants for traditional query operators.	185
9.7	Punctuation scheme assignments for Query 9.1.	191
9.8	Punctuation scheme assignments for Query 9.2.	193

List of Figures

1.1	Simple architecture for a system to monitor an online auction.	6
2.1	Possible query plan for network monitoring query.	20
2.2	System to monitor temperature reports within a warehouse.	25
2.3	Query plan to determine the maximum bid prices for items for auction. . .	29
3.1	Punctuation with a single constant value (<code>hour</code>) and wildcards.	41
3.2	Punctuation with a constant value (<code>hour</code>) and range (<code>temp</code>).	41
3.3	Punctuation with a range (<code>temp</code>) from the minimum value up to, but not including 75, and a constant (<code>hour</code>).	42
3.4	Punctuation with a list (<code>id</code>).	42
3.5	Definition of the function <i>match</i>	43
3.6	Matching and non-matching data items for the punctuation given in Figure 3.1.	43
3.7	Matching and non-matching data items for the punctuation given in Figure 3.3.	44
3.8	Initial query plan used to test punctuation rules.	46
3.9	Amount of time required to retrieve the first and last data items from the group-by operator.	48
3.10	Number of data items held in state for the union operator (with duplicate elimination) throughout query execution.	49
4.1	Model of how output and state are threaded between iterations of <i>q</i> and <i>r</i> . .	56
5.1	Number of data items held in state for <code>groupbyS</code> during query execution, with and without punctuations.	86
7.1	Query plan in Niagara XML format to convert bids from US Dollars to Euros.	114
7.2	Sample bid stream contents.	115
7.3	Sample bid stream contents including a punctuation.	116
7.4	Partial Niagara query plan to show queues between query operators. . . .	117
7.5	Example of an unnest operation.	119

7.6	Pseudocode for converting tuple values from US Dollars to Euros.	120
7.7	Pseudocode for converting punctuations patterns from US Dollars to Euros.	123
7.8	Behavior of the Symmetric Hash Join.	125
7.9	Example punctuations on bids that describe different attributes.	127
7.10	Use of the describe operator in a Niagara query plan.	131
7.11	Source code for handling punctuations in the describe operator.	132
7.12	Query plan to determine the number of bids for items each hour.	135
7.13	Query subplan to handle packet reconstruction with a punctuate operator.	138
8.1	Architecture for the On-line Auction System	142
8.2	Plan for Query 3 using the describe operator.	146
8.3	Plan for Query 4 using the describe operator.	147
8.4	The XML structure for items from stream sources.	148
8.5	Example punctuations for bid data items.	149
8.6	Results for Query 1.	152
8.7	Results for Query 2.	153
8.8	Results for Query 3.	154
8.9	Results for Query 4.	155
8.10	Number of bid data items stored and data items output.	159
8.11	Accuracy and expansion of results for various values of slack.	160
9.1	Data items that match punctuations for a specific destination port and hour.	162
9.2	Architecture for the network monitoring system.	164
9.3	A possible query plan for Query 9.1.	173
9.4	The query plan for Query 9.2.	192
9.5	The query plan for Query 9.2.	194

Abstract

Punctuated Data Streams

Peter A. Tucker

Supervising Professor: David Maier

As most current query processing architectures are already pipelined, it seems logical to apply them to data streams. However, two classes of query operators are impractical for processing long or unbounded data streams. Unbounded stateful operators maintain state with no upper bound on its size, and so eventually run out of memory. Blocking operators read the entire input before emitting a single output, and so might never produce a result. We believe that a priori semantic knowledge of a data stream can permit the use of such operators in some cases. We explore a kind of stream semantics called *punctuated streams*. Punctuations in a stream mark the end of substreams, allowing us to view a non-terminating stream as a mixture of terminating streams. We introduce three kinds of invariants to specify the proper behavior of query operators in the presence of punctuation. *Pass invariants* unblock blocking operators by defining when such an operator can pass results on. *Keep invariants* define what must be kept in local state to continue successful operation. *Propagation invariants* define when an operator can pass punctuation on. We then present a strategy for proving that implementations of these invariants are faithful to their finite table counterparts.

In practice, it is important to answer the following question: “How much additional overhead is required when using punctuations?” We use the scenario of a monitoring

system for an online auction. Streams of bids, new items, and new users are sent to an online auction system. There are many interesting queries that can be posed over these auction streams. We define queries for this scenario, and execute them with different kinds and amounts of punctuations embedded in the input streams. We show that, for a reasonable ratio of punctuations to data items, the overhead is minimal. Additionally, we compare the behavior of a query using punctuations with the behavior of the same query using slack over data streams with disorder.

Clearly, not all punctuations are useful to a particular query, and it would be useful to make a determination of when they are. That is, we would like to answer the question “Can stream query Q benefit from a particular set of punctuations?” To that end, we first define *punctuation schemes* to specify the collection of punctuations that will be presented to a query on a particular data stream. We show how both punctuations and query operators induce groupings over the items in the domain of the input(s). We show that a query benefits from an input punctuation scheme (in terms of being able to produce a given output scheme), if each set in the groupings induced by the operators of the query is covered by a finite number of punctuations in the scheme — a kind of *compactness*.

We conclude with discussion on possible future directions of research related to punctuations and data streams. These directions focus on a variety of questions, ranging from issues in query optimization to other possible semantics that can be expressed using punctuations.

Chapter 1

Introduction

Data presented in the form of a stream is becoming more commonplace. Data streams originate from a variety of sources. For example, they may arise from a wireless or wireline sensor node, from customers over the Internet (e.g., customer purchase orders), from a router or firewall in the form of raw network data, or from an appliance such as a mobile phone. As the number of kinds of data presented on a stream increases, the need to be able to filter, transform, aggregate, and combine streaming data efficiently and generically becomes more and more important.

Often, applications that process data streams are designed to handle a specific kind of data. We will explore three example application domains: An application that monitors temperature data streamed from environment sensors to track temperature; an application that monitors the usage and performance of a network, as well as looking for intrusion or inappropriate usage; and an online auction application that monitors bids and new items for auction. It would be useful to develop a general system for processing data streams with which we could implement specific data stream applications, in a manner similar to how a Data Base Management System (DBMS) is used to implement multiple applications over finite data.

As data presented on a stream are often structured, it is appealing to use a DBMS to execute queries over them. There are several advantages to using a DBMS: First, data manipulation can be expressed in SQL, a well-known, domain-specific language. Second, applications can take advantage of DBMS technology, including query optimization. Third, a DBMS is already very good at retrieving stored data. Combining stored data with data from a stream would be useful.

However, two properties of data streams are difficult for a traditional DBMS to handle: First, data streams may present an unbounded amount of data. Many query operators in a DBMS expect an end to the input, which is not always the case for data stream inputs. Second, data from a stream is presented sequentially. Much of DBMS technology relies on random or repeated access to individual data items, which is not always possible when reading from a high-volume data stream.

A number of suggestions and approaches have been proposed to process data streams with DBMS technology. We will discuss approaches closely related to our approach in Chapter 2; see Babcock et al. [BBD⁺02] for a more complete list. Our approach is to embed special items in the data stream we call *punctuations* which improve the behavior of many query operators on data streams. A punctuation can be seen as a predicate over data in the stream, marking the end of a subset of data. Any data item that satisfies the predicate defined by the punctuation must precede the punctuation in the input stream. We call streams that contain punctuations *punctuated streams*. Operators that assume there will be an end to the data can use punctuations to determine the end of specific subsets of data. Often, knowing the end of a subset of data has arrived is enough to improve the behavior of an operator when operating over a non-terminating stream of data.

1.1 Motivating Examples

The following subsections present details of some motivating example applications that process streams of data. Each example application points out specific issues that must be addressed for the application to run effectively. We present each application in SQL for concreteness. We will discuss in Chapter 2 how our punctuated stream approach, as well as other approaches when possible, can be used to solve the problems posed by the examples.

1.1.1 Warehouse Monitoring System

A warehouse containing temperature-sensitive merchandise may deploy temperature sensors to report the current temperature regularly to a monitoring system. Each sensor is assigned its own identifier (ID), and presents its own data stream. A data item from a sensor contains the sensor ID, the current time and the current temperature. Suppose the monitoring system must report the count of distinct sensors that report temperatures over 75F each hour.

We would like to implement this application using a traditional DBMS. The monitoring system might be implemented as a query that reads the sensor ID, hour and temperature from each data item, unions all temperature reports, filters out temperature reports less than or equal to 75, groups all reports by hour eliminating duplicate sensor IDs, and finally outputs the number of distinct sensors in each group that report temperatures greater than 75 degrees. In the following SQL, we refer to each sensor by name, as `sensor1`, `sensor2`, and so on up to `sensorN`:

```
SELECT hour, COUNT(DISTINCT id) AS sensors
FROM (SELECT * FROM sensor1
      UNION
      SELECT * FROM sensor2
      UNION
      ...
      UNION
      SELECT * FROM sensorN)
WHERE temp > 75
GROUP BY hour;
```

Unfortunately, there are two reasons why this query may fail using a standard DBMS: First, duplicate elimination may require an unbounded amount of state. Thus, if state required for the query is kept in memory, the system will ultimately run out of memory. An alternative implementation might write state to disk when there is no room left in memory, but this approach would slow down the overall performance of the query. Second, a traditional implementation of group-by must wait until all data from its input have been

read before it can output results. Since the input is a non-terminating stream, the query will never output a result.

1.1.2 Network Traffic Monitor

Streams of raw network data also contain structured items. Common network protocols define structure over data in the form of *packets*. For example, an IP packet contains the source IP address, the destination IP address, the fragment offset (if fragmented), flags, and a packet identifier [Pos81a]. A TCP packet includes everything in an IP packet as well as source port number, destination port number, and other information [Pos81b]. Many applications process streams of network packets to monitor usage, analyze performance, and detect security issues such as intrusions.

Consider the following example: Network packets may be fragmented after they leave their source, to be reassembled at the destination. A network packet is assigned an identifier by its source so that the receiver can know which fragments belong to which packet. Each fragment contains the identifier of the packet, an offset specifying in bytes where it belongs in the reconstructed packet, and a flag specifying if it is the last fragment for a packet. One known attack to bypass a network firewall takes advantage of fragmented network packets [Coh96]. Network firewalls are designed to only allow network traffic for specific ports to pass through. For example, a firewall might allow packets destined for port 80 (HTTP) to pass through, but block packets destined for port 23 (TelNet). One known strategy for bypassing a firewall from a remote system in order to send packets to a blocked port proceeds as follows:

- The first fragment for a network packet is output with fragment offset set to 0 and with the destination port for the entire packet set to 80.
- The firewall allows this fragment to pass through, along with any subsequent fragments with the same network packet identifier, to be reassembled by the host inside the firewall.
- One of the subsequent fragments has a fragment offset of 0 (again), this time with a destination port of 23.

- The packet is reassembled inside the host and sent to port 23 instead of port 80, and the hacker has reached the blocked port.

Database-style queries over network traffic have already been shown to be a useful tool for network analysts — the Gigascope system [JCSS03] is an example. We can use a database-style query to detect intrusion attempts using the scenario above by counting the number of fragments that arrive for a particular ID with an offset equal to zero. If we get more than one fragment for a specific ID with an offset of zero, then an attack may be underway. The SQL for this query is as follows:

```
SELECT seqno, sourceip, sourceport COUNT(*) as NumOfZeroFrgs
FROM ippacketsource
WHERE fragmentoffset = 0
GROUP BY seqno, sourceip, sourceport
HAVING COUNT(*) > 1;
```

As in the warehouse monitoring system, this query requires a blocking operator, the group-by operator. However, the grouping attributes (`seqno`, `sourceip`, `sourceport`) over network packets are not non-increasing or non-decreasing. In the warehouse example, we might have been able to alter the blocking operator slightly to output data for the current group when a data item arrived with a new value for hour, if hour is non-decreasing. As mentioned earlier, network packet identifiers are assigned by the packet source. Since packets will arrive from many sources, we cannot assume that the identifier values will arrive in order. Further, fragments from many sources (with different identifiers) may arrive at the same time. Therefore, fragments with different identifiers will be mixed together in a data stream. We cannot assume that fragments for the current packet have all arrived just because a fragment with a new identifier has arrived.

1.1.3 Online Auction

There are a number of examples of commercial and research systems for processing and monitoring online auctions, including EBay [EBA], Yahoo! Auctions [YAH], the Fishmarket [RNSP97], and the Michigan Internet AuctionBot [WWW98]. Software agents may

be used to represent humans in the auction to bid on or sell items. A user first registers with the system through an agent, then participates in auctions as a buyer or seller. We model an online auction with three kinds of stream sources (which can be implemented as software agents) that supply data to an auction monitoring system. This architecture is shown in Figure 1.1. Bids for items currently for sale arrive on the **Bid** stream. New items for sale arrive on the **Auction** stream. New users arrive on the **Person** stream. A relational database is used to store auction history. However, systems that try to use a database to track active auctions face two issues: First, because all auctions read and store information in the same database, the database itself becomes a bottleneck [Wur03]. Second, synchronization issues arise when multiple system agents try to further the progress of an auction [WWW98]. For example, bids may arrive for an auction that has expired but not yet marked as closed by the system. When the system does mark the auction closed, care must be taken to ensure that only those bids that arrived while the auction was active should be considered. We can avoid these two issues by processing active auction data as it arrives.

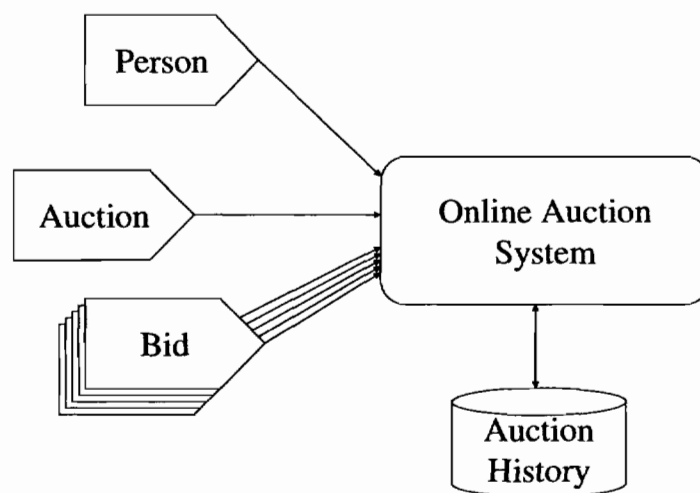


Figure 1.1: Simple architecture for a system to monitor an online auction.

Auction items may be organized in a hierarchical catalog of categories. Sellers are able to place items in an existing category, or insert an appropriate subcategory. Further, users may choose not to participate in the catalog. Consider an application that tracks winning bids for items in a particular set of categories. This application requires combining data from the *Bid* stream and the *Auction* stream based on the item ID, and it outputs the maximum bid when the auction is complete. The SQL for this query is:

```
SELECT B.a_id, MAX(B.price)
FROM Auction A, Bid B
WHERE A.a_id=B.a_id AND A.category IN {92, 136, 208, 294}
GROUP BY B.a_id;
```

A relational database system might execute this query with a select operator to filter out unwanted categories, a join operator to combine data items from the auction and bid streams, and a group-by operator to retrieve the maximum price. Since select does not maintain state and does not block, it can process data streams unaided. The group-by operator will have difficulties similar to those in the network example above. However, the join operator poses new challenges. There are many algorithms for implementing join. Algorithms that block over one or both inputs clearly cannot be used over data streams (e.g., paged nested loops). Further, implementations that require random access to the input, such as through an index (e.g., index nested loops), are also inappropriate for high-volume streams as buffering and then re-scanning the input is undesirable (or infeasible). The symmetric hash join algorithm [WA91] does not block on its inputs, so it is more appropriate for processing data streams. However, symmetric hash join maintains an unbounded amount of state, using a hash table for each input. As the query executes, hash tables used for the join grow without bound, so the system eventually runs out of memory.

1.1.4 Issues Raised from the Example Queries

Data streams are often *non-terminating* and *unbounded*. By non-terminating, we mean that, at any finite time t , there exists some data item that will arrive from the stream

after time t . By unbounded, we mean that, for any volume v , there is some finite prefix of the stream with more than v data items. We have already seen that many query operators implemented in a database management system (DBMS) are inappropriate for data streams. *Blocking operators*, such as group-by and sort, cannot output results until all input data have been read. Over non-terminating input, a blocking operator will never output a result. *Unbounded stateful operators*, such as duplicate elimination and join, maintain state that increases in size as more data arrives. Over unbounded data streams, an unbounded stateful operator will eventually consume any finite amount of memory. For example, consider a duplicate elimination operator that utilizes a hash table to remove duplicates. Each unique data item that arrives is added to the hash table. If the stream contains an unbounded number of unique data items, then the hash table will eventually run out of space to hold new items.

We have presented three queries that are important for different applications over data streams. Not one of the queries discussed will execute successfully in a traditional DBMS without help. Using a DBMS to implement applications over data streams would be beneficial for a number of reasons, which we will discuss in more detail in the next section.

1.2 Advantages of Using DBMS-Style Queries over Data Streams

Each application listed in Section 1.1 can be implemented using a general-purpose language, such as C++ or Java. In this section, we discuss three advantages of using a DBMS to implement applications that process data streams: First, applications written in general-purpose languages can be difficult for application developers to maintain. Instead, implementing stream-processing applications using a domain-specific language will often make implementing such applications more efficient. Second, many stream processing applications require data manipulation functionality commonly found in a DBMS, such as filtering and aggregation. Third, many stream-processing applications require data from finite sources (e.g., auction categories) in addition to streaming data.

Domain-specific Languages A domain-specific language is designed to improve the implementation process of applications in a specific application domain. There are many examples: The PERL language targets text and file processing, the OpenGL and Direct3D languages target three-dimensional graphics rendering, and the T_EX and L^AT_EX languages target document layout. A domain-specific language often makes implementing applications in its target domain easier than implementing them in a general purpose language, reducing the development effort of building more complex applications in terms of implementation as well as maintenance.

As data in a stream are often structured, a logical domain-specific language for implementing applications that process data streams is SQL, commonly found in relational DBMS's. The widespread use of SQL makes it appealing for processing data streams — users already familiar with SQL will not have to learn a new language for writing queries over data streams.

Common Data Manipulation Functionality Many stream-processing applications require functionality commonly found in a DBMS. A stream-processing application must read from its input streams and parse each data item, buffer data items as needed, and output results. Results may be computed using operations such as computing aggregate functions over the data, joining and merging data from multiple inputs, and filtering data. More complex applications may perform arbitrary combinations of such operations.

A DBMS already supports many of these operations. Further, generally speaking, it has multiple algorithms to choose from for each operation, techniques to decide on particular algorithms, and methods to determine which order to execute each operation.

Stored Data Many stream applications require data that has previously been stored to disk to compute their results. For example, a network monitoring application might want to compare the number of packets arriving at each hour, to see how the current volume of network packets compares with the same hour on previous days. A DBMS already supports retrieval and manipulation of stored data.

Current DBMS's support the functionality listed above for finite inputs that are files of structured data, but are generally inappropriate for processing data streams. By using a DBMS to process data streams, we can take advantage of features such as parsing structured data, query optimization techniques, and producing results (e.g., forms and reports). Further, stream-processing applications may be able to make use of data management research on multi-query optimization [DSRS03, RSB00, Sel88], sequential databases [SLR94, SLR95] and temporal databases [Soo91].

1.3 Our Proposed Solution — Punctuated Data Streams

Our solution builds on the following observation: The operators we are focusing on (blocking and unbounded stateful operators) all benefit from reaching the end of their inputs. A blocking operator outputs results when the end of its inputs have been reached, because no further data items will arrive to affect its output. An unbounded stateful operator purges state when the end of its inputs have been reached, because data kept in the operator's state is no longer required. Suppose an operator knew that no more data items for a specific subset of the input domain will ever arrive. A blocking operator might be able to output results for that subset, and an unbounded stateful operator might be able to purge state corresponding to that subset.

This observation is the basis for punctuated data streams. Punctuations describe a subset of the data in the input domain. We embed punctuations into a data stream to denote that no more data items will arrive that belong to the subset described by that punctuation. (Note that “end-of-input” can itself be considered a punctuation.) We call a punctuated stream *grammatical* if each punctuation is correctly placed: It appears somewhere after the last data item from the subset it represents. We will give a formal definition of grammatical in Section 4.5.

For example, suppose stream S contains pairs of integers (we use the notation $[[\dots]]$ to denote stream contents):

$$S = [(1, 2), (3, 3), (5, 8), (4, 3), \dots].$$

Now suppose we know after $(5, 8)$ that no more data items will appear in S where the

first value is less than 4. We could embed into the stream a punctuation p after the value $(5,8)$, where p matches the following data items:

$$\{(a, b) | (a, b) \in \mathbb{Z} \times \mathbb{Z} \wedge a < 4\}.$$

Embedding p results in a grammatical stream S' :

$$S' = [(1, 2), (3, 3), (5, 8), p, (4, 3), \dots].$$

1.3.1 Punctuation Behaviors for Query Operators

A *punctuation behavior* defines an action an operator should take in the presence of punctuations. We have identified three different kinds of punctuation behaviors for a query operator: First, an operator may output some results before encountering the end of the stream. This behavior applies mostly to blocking operators. Second, an operator may reduce the size of its state. This behavior applies mostly to unbounded stateful operators. Third, an operator may be able to emit punctuation for use by subsequent operators. We will later give formal definitions for these behaviors for many traditional relational query operators.

1.3.2 Effects of Punctuation for Entire Queries

It is not enough to say that punctuations can help individual query operators. Most meaningful queries require multiple operators working together over the input data. We must also show how punctuations affect entire queries. There are two parts to this analysis: First, we want to investigate the performance overhead of embedding punctuations into data streams for query execution. Clearly, punctuations take up bandwidth and processing resources. We want to find out if the improvement in behavior we get for blocking and unbounded stateful operators is worth the increased bandwidth and additional processing overhead, and to see what the penalty is for queries that do not require punctuations.

The second part of this analysis is more theoretical. There are two questions that we want to answer: “Can a given query Q benefit from particular punctuations?” and “What kinds of queries benefit from punctuation?” To address these questions, we first define *punctuation schemes* to specify the collection of punctuations that will be presented to a query on a particular data stream. We show how both punctuations and query operators

induce groupings over the items in the domain of the input(s). Further, we show that a query benefits from an input punctuation scheme (in terms of being able to produce a given output scheme), if each set in the grouping is covered by a finite number of punctuations in the input scheme — a kind of *compactness*.

1.4 Contributions of this Dissertation

The contributions of this dissertation are fivefold:

1. We define the semantics of punctuated data streams in Chapter 4. As stated before, a punctuation denotes the end of a subset of data. We formally define punctuations and important functions for working with punctuations and data items.
2. We present a general implementation framework for query operators and how they process data streams using a functional programming language in Chapter 5. In Chapter 6 we use this framework to prove that the implementations of query operators conform to the punctuation behaviors.
3. We define behaviors that query operators should adhere to in the presence of punctuations in Chapter 6. We define what correctness means for an operator, and then prove that the punctuation behaviors we define for each operator are correct.
4. We show that the performance of queries in the presence of punctuated data streams is comparable to that of queries over non-punctuated data streams in Chapter 8 (for terminating input). Embedding punctuations into a data stream takes up bandwidth. We show that query execution time does not increase significantly over punctuated streams, even for queries that do not require punctuations (such as filter queries). Further, we show the positive effects of punctuations for queries that can take advantage of the punctuations.
5. We demonstrate a way to determine the kinds of queries that can benefit from a particular set of punctuations, and to determine a set of punctuations that can help a given query in Chapter 9. We borrow notions from topology to define covers in terms of punctuations, and compact sets of data items based on our covers.

In addition to the contributions described above, we address a number of related issues, as follows:

- We present a new representation of data streams as a “sliced list” of data items, to overcome many issues with the standard singleton-list representation for streams. For example, it is difficult to model a “lull” or a burst of data with a singleton list. A sliced list representation can easily model lulls by emitting empty slices, and bursts by emitting slices with many data items.
- We define a particular class of punctuations using a schema that corresponds to the schema defined for the data stream. Our representation is simple, but has the useful closure property that we can logically “and” two punctuations together to get a new punctuation.
- We discuss ways to handle disorder in data streams using punctuations. Many stream systems assume order in a stream. For example, a stream system might rely on each data item in the stream to arrive sorted on time values, so that it can output results over the past hour when it sees the first data item for the next hour. If data items arrive slightly out-of-order, the assumptions made on sorted data can cause incorrect results. We show how punctuations can be used to handle this situation.
- We introduce two new query operators for handling punctuation. The *punctuate* operator embeds punctuations into a data stream based on known stream constraints. For example, if the timestamp value for data items is monotonically increasing, then we can use the punctuate operator to output punctuations periodically based on the timestamp value. The *describe* operator interprets and filters punctuations in a data stream. Data items are output from the describe operator as they arrive. For punctuations, the describe operator has two functions: First, it filters out punctuations that are unnecessary for the query. Second, it constructs new punctuations from incoming punctuations, where possible, that are of more use to downstream query operators.

1.5 Structure of this Document

The remainder of this dissertation is structured as follows: First, we give a background on processing data streams with DBMS technology in Chapter 2. We then discuss our initial efforts to explore the feasibility of punctuated data streams and list questions raised from that investigation that will guide much of the rest of this thesis in Chapter 3. In Chapter 4, we introduce our formal semantics for punctuated data streams and, in Chapter 5, we present our framework for modelling query operators and their behaviors over punctuated streams. We give our formal theory of punctuation behaviors in Chapter 6. We present details of enhancing query operators in a real query engine to support punctuations in Chapter 7, and discuss our performance results in Chapter 8. In Chapter 9, we analyze how punctuated streams can benefit entire queries. We discuss related work in Chapter 10, and conclude in Chapter 11.

Chapter 2

Background on Relational Database Management Systems and Data Stream Processing

There are several approaches for solving the difficulties of processing non-terminating data streams using query operators found in a DBMS. We discuss and compare these approaches in this chapter. First, we give a brief overview of query operators, and then give background on other techniques used to process data streams in other systems. Finally, we attempt to apply these techniques to the examples discussed in Section 1.1.

2.1 Brief Overview of Query Operators

Much of our work focuses on the behavior of individual query operators in a DBMS. Therefore, it is appropriate to give a brief discussion in this section of the query operators we will use. For each operator, we will discuss some traditional implementations and their applicability to unbounded data streams. The query operators we are concerned with in this work are listed below (along with the traditional symbols used to represent those operators).

2.1.1 Unary Query Operators

select (σ_p) The select operator filters data items from a single input based on some predicate p . Data items that evaluate to *true* for p are output. The select operator

does not store data items in state, and it outputs data items that pass p as they arrive, so it is not blocking. Therefore, select can be applied over data streams.

project (π_A) The project operator outputs only the subset of attributes A for each data item from a single input that arrives. Our version of project does not eliminate duplicates. The project operator does not store data items in state, and it outputs data items as they arrive. Therefore, project can be applied over data streams.

sort ($\mathcal{S}_A$) The sort operator outputs the input data from a single input in sorted order based on the values for attributes in A . Traditional implementations for sort require reading all data items before outputting results and so it is a blocking operator. Therefore, sort cannot be applied over data streams without additional knowledge.

duplicate elimination (δ) The duplicate elimination operator, referred to as *dupelim*, outputs unique items from its input. There are two traditional implementations: In one implementation, the input is sorted, and then data items are output. All duplicates are grouped together and only the first of each group is output. Since this implementation requires sorting, it is a blocking implementation, and cannot be applied over data streams without help.

The other implementation of *dupelim* uses a hash table. The hash key is generated from all values in a data item. If the hash key for a data item from the input already exists, then each item for that key is compared to the current item. If there are no matches, then the data item is output and added to the list of items for that hash key. If the hash key does not exist, then the item is added to the hash table and output. Since items are output as they arrive (if a duplicate has not arrived already), this implementation is not blocking. However, for unbounded input it does require an unbounded amount of state. Therefore, *dupelim* cannot be applied over data streams without help.

group-by ($\mathcal{G}_A^{agg}$) The group-by operator groups all data items from a single input based on values of the attributes in A . Once all data items have been read, then the aggregate function *agg* will be executed over each group. The group values for A

and the result of *agg* will be output for each group. Since all data items must be read before any output is generated, group-by is a blocking operator, and cannot be applied over data streams without help.

2.1.2 Binary Query Operators

join ($\bowtie_P$) The join operator reads from two inputs, and outputs data items in the cross-product of those two inputs that pass some predicate P . We limit our discussion to equality joins (equi-joins). This limitation increases the number of join algorithms we can consider by allowing us to include hash-based join algorithms. In the online auction example, bids and auctions were joined together based on the predicate *Auctions.a_id* = *Bids.a_id*.

A naïve implementation for join is to first compute the cross-product of the two inputs, then evaluate the predicate for each data item in the cross-product. Clearly, this implementation is inefficient. A great deal of research has been directed at implementing more efficient join algorithms. Some implementations must read in the entire input of at least one relation before producing an output or read from an index (e.g., paged nested loops and index nested loops [RG03]). Clearly these implementations are not appropriate for non-terminating data streams, though they may be appropriate for a join of a stream and a stored relation.

We discuss four join implementations that are not blocking. First, if the input data streams are already sorted on attributes participating in the join predicate, then the merge-join implementation [RG03] may be a good candidate. When a data item arrives from one of the inputs, we first check to see if its join values equal join values for data items from the other input, and output those that do. Any data items from the other input that precede the newly arrived item according to the sort order can be removed from state. Note that, if at least one of the inputs presents an unlimited number of constant values, then this implementation will require an unbounded amount of state.

The second non-blocking implementation of join is a generalization of the merge

join, called the symmetric hash join [WA91]. This implementation maintains a hash table for each input. Unlike merge join, the inputs do not need to be sorted. When a data item arrives from one input, we first probe the hash table for the other input to find matching data items from that input. Any matches are output, then the data item is added to the hash table for its own input. This implementation is not blocking, but does require an unbounded amount of state.

The third non-blocking implementation of join is extension join [Hon80]. This implementation is similar to symmetric hash join, but makes the assumption that the join is a one-to-many join. For non-terminating data streams, we add the constraint that data items from the one-side will arrive before items on the many-side. In this case, we only need to maintain a single hash table for data items from the one-side. This implementation also requires an unbounded amount of data, and therefore is not appropriate for unbounded data streams without help.

The fourth non-blocking implementation of join is XJoin [UF00]. XJoin is an extension to the symmetric hash join, where data items in either hash table can be partitioned to disk. Like the symmetric hash join, XJoin requires an unbounded amount of state (though since it can use secondary storage it can handle more data), and therefore is inappropriate for unbounded data streams without help.

union ($\cup$) The union operator reads from two inputs and outputs all data items that arrive. Union requires that both inputs have the same number of attributes, and that each corresponding attribute is of the same type. When two inputs have this property, they are called *union compatible*. Since union does not add data items to state, and since it is not blocking, union is appropriate for non-terminating data streams. (Note that we do not remove duplicates in our version of union.)

intersect ($\cap$) The intersect operator reads from two inputs, and outputs data items that appear on both inputs. Like union, intersect also requires that the inputs be union compatible. Intersect can be implemented using join and project, where the join predicate matches all attributes of data items from one input with the other, and the project operator outputs only the attributes from one input. As with join,

intersect requires an unbounded amount of state, as data items from one input must be held in state as they arrive to determine if they intersect with data items from the other input (and vice versa).

difference (–) The difference operator reads from two inputs and outputs data items in the first input that do not appear in the second. Thus, the difference operator must read the entirety of the second input before producing an output. Since it is a blocking operator, difference is inappropriate for data streams without help.

2.1.3 Building Complex Queries from Individual Operators

When a DBMS receives a query to be executed (using SQL or some other query language) it breaks the query into individual query operators. The query operators are formed into a tree (also called a query plan), where the results of one query operator are the input to the next operator for every operator except the root of the tree. The query in the network monitoring example (Section 1.1.2) was given in SQL as follows:

```
SELECT seqno, sourceip, sourceport, COUNT(*) as NumOfZeroFrag
FROM Router
WHERE fragmentoffset = 0
GROUP BY seqno, sourceip, sourceport
HAVING COUNT(*) > 1;
```

One possible tree for this query is shown in Figure 2.1. In general, a query can have multiple operator trees that produce equivalent results.

2.2 Approaches to Querying Over Continuous Data Streams

Certainly, one solution to using blocking operators for processing data streams is to redefine them in a non-blocking fashion. For example, if we have group-by with the aggregate *max* to output the maximum value over the entire input, the query will block until all the input has been read before producing a result. We could redefine *max* to output a new value every time the maximum value has changed. Given the input [1, 6, 4, 3, 8, 5], the output for the new *max* would be [1, 6, 8]. We call this implementation an *incremental*

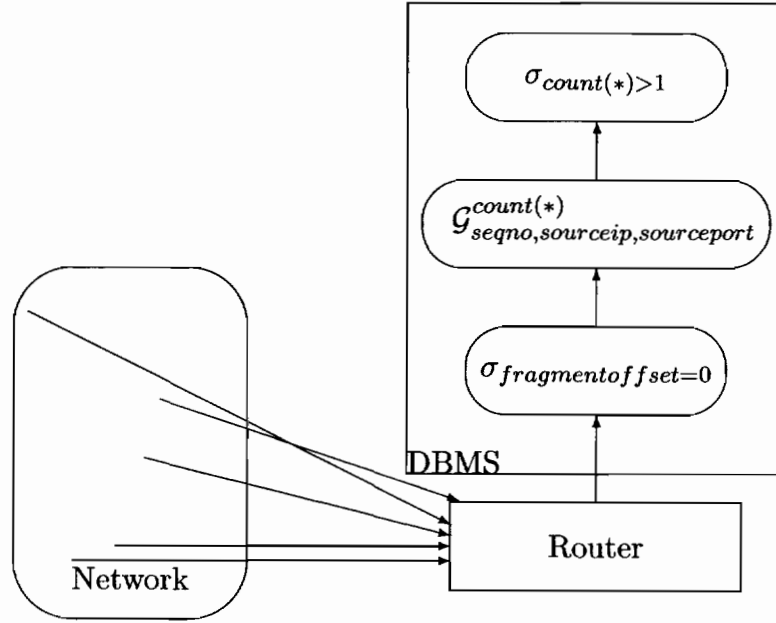


Figure 2.1: Possible query plan for network monitoring query from Section 2.1.3. Ovals represent individual query operators

implementation of *max*. There are interesting queries that can be solved using aggregates defined in this way. While this approach is one way to solve the issue of blocking operators, it does not solve the issue of unbounded stateful operators. Even by outputting incremental results, operators must still maintain an unbounded amount of state.

Law et al. [LWZ04] take the approach of restricting the kinds of query operators allowed in a query involving one or more data streams to those that are non-blocking. For example, they define $\mathcal{NB}$ -RA as those operators in relational algebra that are non-blocking ($\mathcal{NB}$ -RA = $\{\cup, \bowtie, \sigma, \pi\}$). They introduce the notion of $\mathcal{NB}$ -completeness to describe those query languages that can express all functions computable by non-blocking procedures. However, they do not consider operators that require an unbounded amount of state for unbounded input.

In their model, they only consider the effects of operators themselves, without consideration for the semantics of the input data. Operators are considered as either monotone

or not. Punctuation semantics allow us to consider operators that are traditionally considered non-monotonic as at least partially monotone. Punctuations in a data stream stabilize subsets of the input, allowing an operator to behave in a monotone fashion for those subsets. That is, for those subsets of the input, the operator's results will not be influenced by further input, and thus those results can be output.

Online query processing [HH99, HHW97, RH02] uses operators defined similar to an incremental group-by operator to output approximate results for long-running queries over large (but terminating) data sources. Users see updated results as data items arrive, along with an estimate of how accurate the results are. However, the approaches used in online query processing require terminating inputs to determine the accuracy of the current aggregate value. Further, it is unclear how query operators above an online aggregate operator in a query plan should handle output from an incremental aggregate.

One solution to the problem of unbounded stateful operators over data streams is to simply write state to disk. Disk is essentially unbounded, and can therefore hold all data items from a stream as they arrive. This approach will work for a number of queries. However, disk access is slow compared to the possible arrival rates of data streams (e.g., network packet data). When data items from the stream arrive faster than they can be written to disk, data may be lost or, worse, the system could fail. In these situations, data items must be processed upon arrival.

There are at least two other approaches for executing queries over data streams: First, systems can employ techniques from sequence database systems [SLR94, SLR95] and rely on data arriving in a meaningful order to compute results. Second, many systems define *windows* in queries over subsets of data in the stream. We will discuss these two approaches in some detail, then show how they can be used to execute the example queries defined in Section 1.1 when appropriate. We will also discuss how punctuated streams can be used to execute the example queries.

Another notable approach for processing data streams is to use approximations. This approach is less related to punctuated streams, and so we mention it only briefly. Many systems that process data streams maintain approximations of a stream's contents, rather than trying to store the entire stream [DGR03, DH00, GKS01, GKMS01]. Approximations

do not help blocking operators — we can use window techniques in addition to approximations to help blocking operators. However, using certain data structures to approximate the stream contents allows operators to bound the amount of required state while providing reasonable approximations for results. Approximation techniques do benefit unbounded stateful operators.

2.2.1 Positional and Ordered Data

Sequence database systems rely on meta-information about input sequences to optimize the execution of sequence queries. Many of these techniques can be applied to systems that process non-terminating data streams. Blocking operators can be implemented to take advantage of input that arrives in an interesting order. For example, the query in the warehouse example uses a group-by operator to group data items by hour. If the input arrives sorted on hour, then, when a new hour arrives to the group-by operator, results for the previous hour can be output. Unbounded stateful operators can also be implemented to take advantage of input that arrives in an interesting order. We have already discussed how merge join can remove data items from state during execution.

Other systems rely on data arriving in a specific order to process streams of data. The Gigascope system [JCSS03] processes streams of network packet data. Two kinds of operators in Gigascope rely on data that is monotonically increasing: First, the join operator must use a predicate that contains an attribute from each input that is monotonically increasing. The join implementation uses this information to determine when a data item will no longer join with data items from the other join input, and can therefore be removed from state (as in the merge join). Second, group-by requires that the grouping attributes contain an attribute that is monotonically non-decreasing. When a data item arrives whose ordered attribute is greater than any current group, then the group-by operator can deduce that no more data items will arrive that will contribute to that group. The results for that group can then be output.

2.2.2 Windows

A very common approach used by query engines to process data streams is to break the data into contiguous subsets, called *windows* [ABW03, CF02, GKS01, SH98, ZS02]. There have been a number of kinds of windows proposed in the literature. We discuss many of them in detail in Section 10.2.3. The two most common kinds of windows discussed in the literature, using terminology from Sullivan and Heybey [SH98], are:

- *Fixed windows* that break the stream into successive, non-overlapping subsets of data. For example, report the average temperature every ten minutes of all reports from a temperature sensor over the past ten minutes.
- *Moving windows* that break the stream into successive, overlapping subsets of data. For example, report the average temperature every minute of all reports from a temperature sensor over the past ten minutes.

By breaking up the stream input into bounded windows, a query operator processes bounded data sets. As each new window arrives, the query is restarted for that new window. Since the window’s content is bounded, blocking operators are able to output results before reaching the end of the stream, and unbounded stateful operators are able to purge state before the end of the stream.

For example, we could redefine our warehouse query to use fixed windows as, “Output the maximum temperature reported by any sensor every 60 minutes.” The current system time is stored when the query is started. The group-by operator stores the maximum temperature as data arrives. Every 60 minutes, group-by outputs its maximum value, then clears out its state and starts over.

In effect, a window system can be implemented using sequence database system techniques. Each data item is assigned a non-decreasing window identifier (*win-id*). We can then use the technique for group-by over sequences as discussed above — when the win-id value increases, the results for the previous win-id can be output and then removed from state.

2.3 Using Various Stream Approaches in Specific Applications

In Section 1.1 we introduced three scenarios and interesting queries over each scenario. In this section we show how each of the scenarios can be implemented using the approaches discussed in Section 2.2, and how they can be implemented using punctuations.

2.3.1 Warehouse

Of the three scenarios, the warehouse example is the easiest to implement. Recall the SQL for the warehouse example is:

```
SELECT hour, COUNT(DISTINCT id) AS sensors
FROM (SELECT * FROM sensor1
      UNION
      SELECT * FROM sensor2
      UNION
      ...
      UNION
      SELECT * FROM sensorN)
WHERE temp > 75
GROUP BY hour;
```

One possible query plan for this query is shown in Figure 2.2. This query uses union, select, dupelim, and group-by. As discussed earlier, executing this query using traditional DBMS query operators will fail over unbounded input for two reasons: First, duplicate elimination requires an unbounded amount of state. Second, group-by is a blocking operator.

One approach to solving the warehouse example query is to rely on data being sorted on the grouping attribute — time. Suppose we assume that the time attribute for data items from each sensor is non-decreasing. We will use an order-preserving implementation of the union operator. When data items for a new hour are output from union operator into dupelim, then data corresponding to the old hour can be released from dupelim's state. Further, group-by can output its result for the old hour.

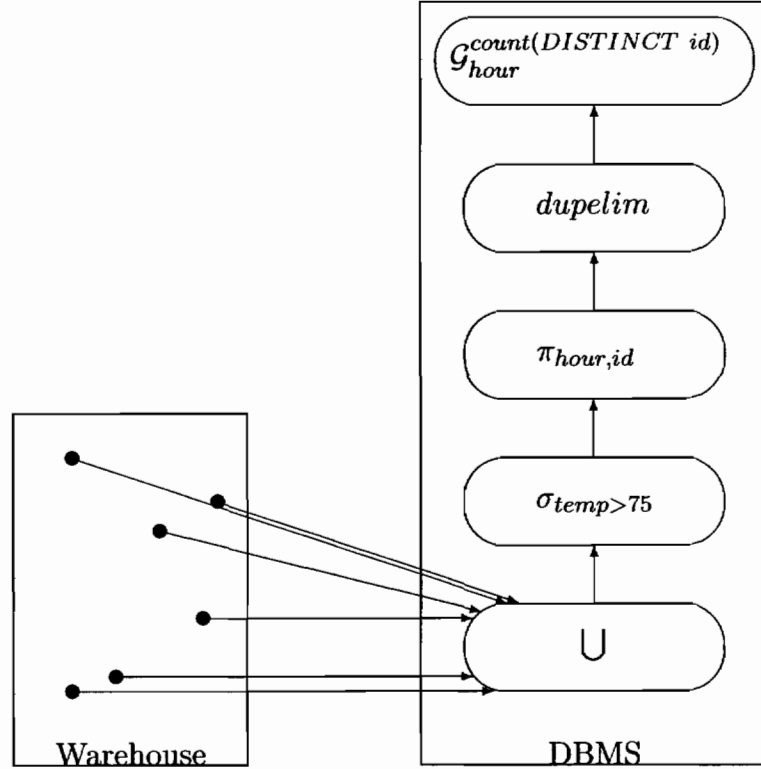


Figure 2.2: System to monitor temperature reports within a warehouse.

This approach will break down if data items arrive out of order. Data items that arrive out of order are either ignored or applied to calculations for the wrong hour. Data items may arrive out of order for a number of reasons, such as data items from the same stream taking different network paths with different latencies. We will address approaches to handling disorder in Section 7.7.

We can also use window queries to address the warehouse example query, by defining the query window to be one hour. We will express window queries using the CQL language [ABW03], an enhancement to the standard SQL language. Specifically, we will use two constructs from CQL: *Istream* and *Range*.

Before we can discuss the CQL constructs, we need to first give some basic definitions from Arasu et al. [ABW03]. Given a discrete, ordered time domain $\mathcal{T}$, there are two data

types: *relation* and *stream*. A relation R is a mapping from T to a finite but unbounded bag of data items belonging to the schema of R . A stream S is a possibly non-terminating bag of elements $\langle s, \tau \rangle$, where s is a data item from the schema of S and $\tau \in T$.

Given these two data types, we need functions to convert between them. Arasu et al. discuss three such functions: *Istream*, *Rstream*, and *Dstream*, that convert a relation input into a stream. We will only use *Istream* here, as *Istream* most closely matches the functionality of our approach. For each time τ , *Istream* outputs data items from the input relation R that are in $R(\tau)$ but not in $R(\tau - 1)$. Assume that $R(-1) = \emptyset$. Then $Istream(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \tau)$.

We also need functions to convert streams to relations. In CQL, all stream-to-relation functions are based on windows. The *Range* and *Rows* constructs are used to convert streams to windows based on time and data items, respectively. We focus on *Range* here. The *Range* function takes a stream S and a time interval T , denoted $R = S \text{ Range } T$, where:

$$R(\tau) = \{s | \langle s, \tau' \rangle \in S \wedge (\tau' \leq \tau) \wedge \tau' \geq \max(\tau - T, 0)\}.$$

That is, the relation returned for some range T at time τ contains all data items $\langle s, \tau' \rangle$ in the stream S where τ' is less than or equal to τ but greater than $\tau - T$.

To implement the warehouse example query, the timestamp for the first data item can be stored in memory, and, when a data item arrives whose timestamp is more than 60 minutes greater than the first timestamp, we know that the window is complete and results for that window can be output. The group-by operator will output results every hour, and duplicate elimination can purge all of its state every hour. The query in CQL is as follows:

```
SELECT Istream(hour, COUNT(DISTINCT id) as sensors)
FROM (SELECT * FROM sensor1
      UNION
      SELECT * FROM sensor2
      UNION
      ...
      UNION
```

```
SELECT * FROM sensorN) [Range 1 hour]
WHERE temp > 75
```

This approach will work well if the system times for all sensors are synchronized. If all systems are synchronized on time, then the data items will all arrive in order. Once a data item arrives that is 60 minutes beyond any data item currently in state, then the system can output the maximum temperature for the previous hour.

This approach becomes less straightforward if data arrives out-of-order. Results for a particular hour might not include data that should have participated in that window.

The issues in the warehouse example query can also be addressed by embedding punctuations in the sensor reports, and enhancing the union, select, dupelim, and group-by operators to handle punctuations. We enhance the sensors to embed punctuations at the end of each hour, stating that all reports have been emitted for that hour. When punctuations from all sensors for a particular hour have been received by union, we know that there will be no further reports from any sensor for that hour. (Note that we are assuming that data will arrive in the order it is emitted from the source. We will discuss handling disorder in data streams in Section 7.7.) Union can then output its own punctuation for that hour. Select outputs these punctuations as they arrive. When dupelim receives punctuation for a particular hour, data items for that hour can be removed from state, and that punctuation can be output. When group-by receives punctuation for a particular hour, it can output the results for groups that match the punctuation. Additionally, state relating to those groups is no longer needed.

By embedding punctuations, we automatically address the issue of data items arriving out of order. The union operator does not act on the punctuations it has received until matching punctuations have arrived from all inputs. Thus, it will not output a punctuation until it has punctuations from all inputs that match a particular hour.

2.3.2 Network

Recall our goal for this query is to find fragmented network packets where the fragment offset is set to 0 for more than one fragment. The SQL for this query is:

```

SELECT seqno, sourceip, sourceport, COUNT(*) as NumOfZeroFrag
FROM Router
WHERE fragmentoffset = 0
GROUP BY seqno, sourceip, sourceport
HAVING COUNT(*) > 1;

```

Figure 2.1 showed one possible query plan for this query. We cannot rely on the identification value for network packet fragments to be ordered, so the ordered data approach is inappropriate here. Further, we cannot efficiently use windows to answer this query. The values for identification will not neatly fit into a time-based or tuple-based subset of the data.

Punctuations, however, can be used to enable this query to output results. We can embed punctuations into the network stream from the firewall as follows: Each IP packet header contains information about whether this is the last fragment for the packet or not. Fragments for a packet may not arrive in order, but we will know by reading the flags for each fragment whether or not this is the final fragment of the packet. When the last fragment for a particular packet does arrive, the firewall can emit a punctuation following that packet fragment denoting the end of all fragments for that particular sequence number, source IP address, and source port. The example query groups on sequence number, source IP address, and source port. When a punctuation arrives, all results that match that punctuation can be calculated and output, thus unblocking the group-by operator.

2.3.3 Auction

Our example query outputs the maximum bid prices for items in specific categories. Recall that the SQL for this query is:

```

SELECT B.a_id, MAX(B.price)
FROM auction A, bid B
WHERE A.a_id=B.a_id AND A.category IN {92, 136, 208, 294}
GROUP BY B.a_id;

```

We show one possible query plan for this query in Figure 2.3. As in the network example, this example is grouping on an attribute that is not arriving in order — the

auction identifier (`a_id`). Therefore, techniques used in sequential database systems will not help here. Further, there are no natural windows that arise from this scenario. Even if there were a maximum time period for an auction and we used sliding windows based on that time period, we have two difficulties: First, results for auctions that have not yet closed would be output (since the window(s) they belong to will have expired). Second, results for auctions that close before the maximum time span are not available until the maximum time span expires.

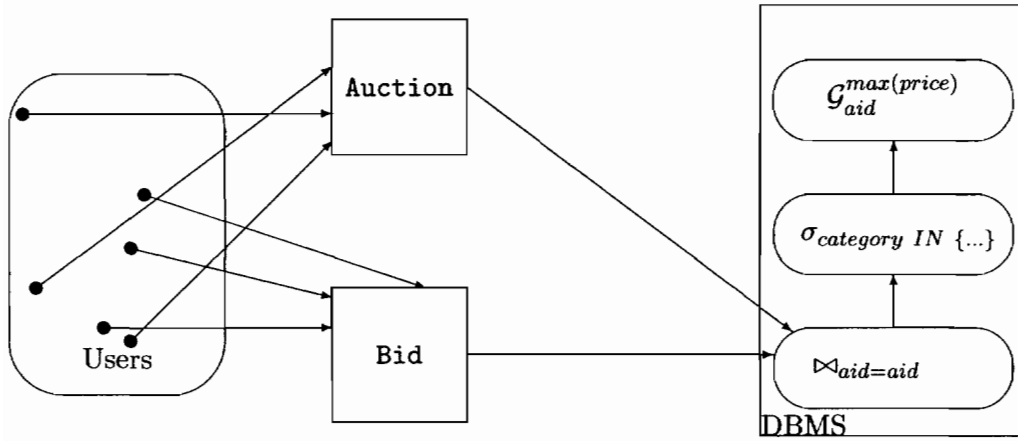


Figure 2.3: Query plan to determine the maximum bid prices for items for auction.

Punctuations can be used to help this query too. However, this query is more involved than the previous examples because it also involves a join operator. Punctuations cannot simply flow through a join operator to the group-by operator to unblock it, as they did through the select operator in the network example. We must make sure that punctuations emitted from the join operator are correct (that is, no data items will follow the punctuation that match it). A punctuation is only emitted from a join operator if it receives punctuations from each of its inputs that match all data items for a given value(s) of the join attribute(s).

Punctuations improve the behavior of this example query in the following manner: Items are assigned a unique auction identifier (`a_id`) when submitted to the Auction

stream source. Once emitted, a punctuation follows the item stating that no more items with that `a_id` will appear from that stream. We will use the symmetric hash algorithm discussed earlier for the join. Recall that this implementation of join maintains two hash tables: in this case, there will be one for bids and another for auction items. A punctuation from the `Auction` stream tells the join that it can remove bid items in its bid hash table with that `a_id` value, because all auction items with that `a_id` have been read. Further, when the auction has expired, a punctuation can be embedded into the bid stream, stating that no more bids for that auction item will appear from the bid stream. The join can use this information to discard state in the auction items hash table with that `a_id` value. Finally, once punctuations arrive from each input with the same `a_id` value, the join can emit its own punctuation, stating that it will not emit any more tuples with that `a_id` value. Punctuations from the join operator are propagated through the select operator. The group-by operator uses that punctuation to determine that no more bids will arrive for that auction item, and can emit its result for that auction item.

2.4 Summary

Some query operators found in a traditional DBMS can be used successfully over data streams. However, others are inappropriate without modification. We have discussed a number of approaches to deal with such operators, including defining incremental versions of operators, applying sequence database system techniques, and windows.

We have seen that sequence database techniques and windows are well-suited for some queries over data streams, but that there are queries for which these techniques fail. We have also seen that punctuations can be applied, even for queries where other techniques are not effective. In the next chapter, we present investigations regarding the feasibility of using punctuations in data streams. At the conclusion of the next chapter, we will present issues that guided the rest of our work.

Chapter 3

Initial Investigation of Punctuated Streams

In this chapter we discuss our initial investigations into how punctuated streams can be used to improve queries over data streams. These investigations were based largely on intuition. It seemed logical that different query operators could exploit punctuations in data streams, but there were three main questions we wanted to answer before pursuing a more extensive research program:

1. Is there a conceptual framework that can guide the implementation of operators that process punctuations? If there are only a few specific behaviors that a punctuation-aware operator can adhere to, then can we precisely specify those behaviors, giving implementors of query operators something concrete to follow?
2. What might punctuations embedded in a data stream look like? What guidelines should we follow to design punctuations so that an operator knows when it is dealing with a data item and when it is dealing with a punctuation.
3. What is the performance overhead for query operators when processing punctuations? Clearly, embedding punctuations into a data stream will take up bandwidth and processing time. Is the overhead substantial enough to make the approach of punctuated streams impractical? (Note at this point we were not after a full performance analysis. We simply wanted a general feel for the overhead required in processing punctuated streams.)

We address these questions in this chapter using the warehouse scenario and enhancements to various query operators in the Niagara Query Engine [NDM⁺00]. As we worked through this exercise, a number of other questions arose that we use to direct the rest of this dissertation. We will raise those questions at the end of this chapter.

3.1 Initial Definition of Punctuation Rules

In our early discussions on punctuated streams, it became clear that there are three types of actions that an operator can take to process punctuations. First, a blocking operator can output results, even in the presence of non-terminating input. Since a punctuation reports the end of a subset of data, it may be able to help a blocking operator output results even though more data may arrive from that input. Second, an unbounded stateful operator can reduce state during execution, even when reading from an unbounded input. Finally, other operators in the query tree might also block or maintain an unbounded amount of state. An operator should emit punctuations when appropriate so that other operators farther along in the query tree can also benefit from punctuations.

These three different kinds of actions that an operator can take in the presence of punctuations led us to define three classes of rules. We constructed a conceptual framework that contains definitions for each of these three rules for various operators, which we will call *punctuation rules* (which are analogous to *punctuation invariants* in our formal treatment later): A *pass rule* defines the data items that can be output early due to punctuations, a *purge rule* defines what state can be released by an operator due to punctuations, and a *propagation rule* defines the punctuations that can be emitted due to incoming punctuations. The definitions of these rules were used to guide the implementation of query operators for processing punctuations.

Our framework also uses a *match* function, which takes a data item and a punctuation and returns *true* if the data item belongs to the subset described by the punctuation. In addition, given a data item t and a set of punctuations P , we define the function $SetMatch(t, P) = true$ if there exists $p \in P$ such that $match(t, p)$.

3.1.1 Pass Rules

A *pass rule* defines when an operator can output part of its result based on punctuations and data items that have arrived. A pass rule takes the form $Pass(op, T_1, P_1, \dots, T_n, P_n) = T_o$, with arguments defined as follows: op is the operator along with any arguments, T_i is the set of data items that have arrived from the i th input, P_i is the set of punctuations that have arrived from the i th input, and T_o is an expression for the set of data items that can be output from the operator. (Throughout our rule definitions we will use the convention that E_i is the contents of the stream that has arrived on the i th input so far, T_i is the set of data items in E_i and P_i is the set of punctuations in E_i .) For non-blocking operators, such as select, the pass rule is trivial, with $op(T_1, \dots, T_n)$ for T_o . (That is, the pass rule is simply the normal behavior of the operator.) Note that pass rules define all data items so far that can be output. In practice, pass rules are implemented incrementally to calculate additional data items based on each new data item or punctuation received. We list pass rules for two example operators here.

Difference

The difference operator blocks on its second input. To compute $A - B$, which contains all data items from a relation A that are not found in relation B , an implementation of difference normally must read all of B before outputting any results so it can be sure which data items from A should be in the result. Difference can use punctuations, however, to know when it has seen enough data items from B to begin to output some data items from A early. The pass rule for difference is:

$$Pass(diff, T_1, P_1, T_2, P_2) = \{t | t \in T_1 \wedge t \notin T_2 \wedge SetMatch(t, P_2)\}$$

When difference receives a punctuation p from its second input, it knows that no more data items that match p will be seen. The operator can then output any data items that it has received from its first input that match p and are not in the set of data items from the second input. Using punctuations, the difference operator no longer has to wait for all data items from its second input before outputting results.

Sort

Sort is also a blocking operator, because it does not know where to place data items that have not yet been read in the sorted output. Since punctuations tell an operator when the end of a subset of data has arrived, the sort operator uses them to determine whether it has enough input to return a correct prefix of its results. Not all punctuations can help sort. In fact, the only punctuations that help sort are those that describe a subset of the input that is a prefix of the sorted output. The pass rule for sort is:

$$\begin{aligned}
 Pass(Sort_A, T, P) &= Sort(\{t \in T \mid SetMatch(t, Initial(Sort_A, P))\}) \\
 Initial(Sort_A, P) &= \text{maximal contiguous subset } P' \text{ of } P \text{ such that } \neg \exists t, u \text{ where} \\
 &\quad (SetMatch(t, P') \wedge \neg SetMatch(u, P') \wedge (u <_A t))
 \end{aligned}$$

The first argument to *Pass* is the sort operator with its list of sort attributes *A*. *Pass* returns all data items that have arrived that match the punctuation set returned by *Initial*. *Initial* takes the sort operator (so that it can determine the sort order) and a set of punctuations as input, and returns a subset of those punctuations, defined so that no data item can be received later that should appear before data items that match this initial punctuation set. That is, data items that match punctuations returned by *Initial* are a prefix of the complete sorted result. As these data items form a prefix of the final sorted output, they can be output by sort.

For example, data in the warehouse example contains $\langle id, hour, temp \rangle$. Suppose we want to sort on *hour*. A punctuation that denotes the end of data items for a specific *id* value will not unblock sort, because we do not know that all data items for a prefix of the sorted output have arrived. Even a punctuation that denotes the end of all data items for a specific *hour* value does not unblock sort. We need a punctuation (or a set of punctuations) that describe a subset containing all possible data items for a prefix of the sorted output. For sorting on *hour* (whose domain we assume is the set of non-negative integers), we need punctuations that describe a subset containing all data items with an *hour* value between 0 and some integer *n*, inclusive. One such punctuation is $p = \langle *, (, n), * \rangle$, which states that all possible data values with hours up to *n* that will arrive have arrived.

3.1.2 Purge Rules

A stateful operator can use punctuations to identify parts of its state that are no longer needed. If an operator is able to discard state earlier than it normally would, it is able to handle more data, making it more suitable for data stream processing. A *purge rule* defines which data items can be removed from state during execution. We define purge rules for every input of an operator, as state might be maintained for each input. Purge rules take the form $Purge_j(op, T_1, P_1, \dots, T_n, P_n) = \hat{T}_j$, where j is the position of the operator's input that is accumulating state that might be purged; op is the operator with any attributes; T_i and P_i are as with the pass rules; and T_j is an expression for the set of data items that can be discarded from state for input j . A purge rule for an operator can assume that the pass rule for that operator has been evaluated and that all possible data items have been passed on.

Certain query operators already purge some state. The purge rules we discuss are in addition to any purging that normally occurs. For operators that do not accumulate state, such as select, the purge rule returns the empty set. For many operators, the purge rules coincide with the pass rules. This behavior makes some sense — the results for data have been output, and therefore they are no longer required in state. This generalization does not always hold, but it will occur for a number of stateful operators.

In Chapter 6, we define analogous *invariants* to these punctuation rules, but we will use *keep* invariants (rather than purge invariants) to describe which data items must remain in state rather than which ones to purge. We list purge rules for two example operators here.

Sort

The pass rule for the sort operator states that results can output early if the punctuations that have arrived match all possible data items in some prefix of the output. Once those data items have been passed on, there is no reason to keep the state associated with them, so that part of state can be discarded. Thus, the purge rule for sort is the same as the pass rule, $Purge(Sort_A, T, P) = Pass(Sort_A, T, P)$. (Note that we drop the index on *Purge*

for unary operators.)

Symmetric Hash Join

The traditional hash join algorithm executes in two phases. First, data items from both inputs are read completely and hashed using the same hash function into partitions. Since the same hash function is used for both inputs, we know that data items from one input that hashed into some partition i will join only with those data items from the other input that also hashed into partition i . Thus, the second phase of execution is to join data items from both inputs that hashed into the same partition for all partitions. This algorithm blocks on both inputs, and is therefore inappropriate for non-terminating data streams.

Symmetric hash join [WA91] is an enhancement to the traditional hash join that outputs results as it reads from both inputs. It maintains a hash table for each input. A data item arriving on either input is stored in the corresponding hash table, and then compared with data items in the other hash table. All data items from the other hash table that satisfy the join predicate with the newly arriving item are combined with it, and then output. This algorithm has the desirable property that it does not block on either of its inputs. However, each hash table grows without bound for unbounded input. Note that, since this is a hash join implementation, we only consider equi-join.

We can enhance symmetric hash join to exploit punctuations to discard some of its state. The operator can use punctuations from one input to match data items in the hash table for the other input, to find data items that will not be joined anymore and can be discarded. As in sort, not all punctuations will help the join operator. We need to know that all possible data items for particular values of the join attributes have arrived. If S_1 and S_2 are the domains for each of the inputs, then the purge rules for the symmetric hash join are:

$$\begin{aligned}
 \text{Purge}_1(\text{Join}_A, T_1, P_1, T_2, P_2) = \\
 & \{t_1 \in T_1 \mid \forall t_2 \in S_2, \text{joinable}(A, t_1, t_2) \Rightarrow \text{SetMatch}(t_2, P_2)\} \\
 \text{Purge}_2(\text{Join}_A, T_1, P_1, T_2, P_2) = \\
 & \{t_2 \in T_2 \mid \forall t_1 \in S_1, \text{joinable}(A, t_1, t_2) \Rightarrow \text{SetMatch}(t_1, P_1)\}
 \end{aligned}$$

The *joinable* function takes the join arguments and two data items, and determines if the two data items pass the join predicate. Each purge rule takes the join operator with the join attributes A , and considers data items from its respective input. Focusing on $Purge_1$, a data item t_1 can be discarded from the first hash table if all possible data items in S_2 that could be joined with t_1 on A match appropriate punctuation from the second input.

3.1.3 Propagation Rules

Propagation rules define the punctuations that an operator can emit as part of its result. There may be another operator waiting for that output that can also take advantage of punctuations, so it is desirable to propagate punctuations whenever possible. The operator must make sure that any punctuation it emits will hold true for its output. That is, no data items output after a punctuation will match that punctuation. Propagation rules for an operator can assume that data items indicated by the pass rule have already been output.

Formally, propagation rules take the form $Propagation(op, T_1, P_1, \dots, T_n, P_n) = P_o$, where: op is the operator with any attributes; T_i and P_i are as before, and P_o is the set of punctuations that can be emitted from the operator. Note that there can be many correct propagation rules for a given operator. Propagation rules must be defined so that no data items output by the operator after a punctuation match that punctuation. The trivial propagation rule for any operator just returns the empty set, but is not very useful. We list propagation rules for two example operators here.

Sort

There are at least two interesting propagation rules for sort. One is to output punctuation that matches data items returned according to the pass rule for sort. In this case, the rule is simply $Propagation(Sort_A, T, P) = Initial(Sort_A, P)$, using the *Initial* function defined for the pass rule for sort (Section 3.1.1).

The second punctuation rule requires more overhead, but possibly outputs more punctuations. In addition to punctuation from *Initial*, the operator iterates through the set of

punctuations that it has read from the input. For each punctuation, if there are no data items that match it in state, then that punctuation can be emitted. This rule is defined as:

$$\begin{aligned} \text{Propagation}(\text{Sort}_A, T, P) = & \text{Initial}(\text{Sort}_A, P) \cup \\ & \{p \in P \mid \neg \exists t \in (T - \text{Purge}(\text{Sort}_A, T, P)) \text{ such that } \text{match}(t, p)\} \end{aligned}$$

This second variation of the propagation rule can emit a punctuation “out of order” relative to the sort order. For example, suppose we are sorting authors by last name, and a punctuation arrives marking the end of authors with last name starting with ‘G’. If we find that there are no authors currently in state with a last name starting with ‘G’, we can emit the punctuation for ‘G’ even though we may only have passed along data items for authors up to ‘D’.

Union

The union operator must combine punctuations from all inputs into one punctuation for the output. The propagation rule for binary union is defined as:

$$\text{Propagation}(\text{Union}, T_1, P_1, T_2, P_2) = \{p \mid p_1 \in P_1, p_2 \in P_2, p = p_1 \cap p_2\}$$

where $\cap$ has the property: $p_1 \cap p_2 = p$ such that $\text{match}(t, p_1) \wedge \text{match}(t, p_2)$ if and only if $\text{match}(t, p)$. That is, p matches all data items that match both p_1 and p_2 .

For example, consider a union on two inputs, each of which contains the attributes $\langle A, B, C \rangle$. Punctuation arrives from the first input stating that all data items with values for attribute A in the range $[5, 15]$ have been read. Unfortunately, it would be incorrect for union simply to emit that punctuation immediately, because it is possible for the second input to still have data items with values for A in that same range. Instead, union must store the punctuation. If punctuation arrives from the second input for the range $[10, 20]$ on A , then the operator can intersect the two ranges, and pass on punctuation for the range $[10, 15]$ for attribute A . If the next punctuation from the second input is for the range $[20, 25]$, then the union operator can intersect this range with the range from its first input. In this case, $[5, 15] \cap [20, 25]$ is the empty set, so no punctuation is passed.

The reader may wonder about state required for punctuations. Constructing purge rules for punctuations is an interesting research direction that we have not spent a lot of time on.

3.2 Implementation Details

At this point, we implemented and tested our punctuation rules with actual data streams. To describe our implementation, we first specify our punctuation format, and give a definition for the *match* function using that format. We then discuss specific implementations of punctuation rules for some operators based on our punctuation format. In Chapter 7 we will give a more thorough discussion of the implementation details. Our testing uses streams of data in XML format. XML provides a reasonable format for tabular data, and there has been a lot of research recently on how to query XML data. We used the Niagara query engine [NDM⁺00] developed jointly at the University of Wisconsin and OHSU to read XML streams, and enhanced certain operators in the query engine to process streams that contain punctuations as well as data.

3.2.1 Format for Punctuation

We have three goals for our punctuation format. First, the punctuation itself should be small, similar in size to a data item in the stream. Second, the punctuation should not affect the results of query operators that do not understand punctuations. Third, it should be easy for the query operator to determine what subset of data a punctuation is describing. The Namespaces in XML [BHL99] recommendation helps us to address these goals. We can define a namespace that mirrors the structure of the existing data items, keeping the punctuation size similar to that of a data item. Since the punctuations belong to a different namespace, query operators will not confuse them with actual data items. For example, if we have the following data format for a stream of temperature reports:

```
<SENSOR>
  <ID> </ID>
  <HOUR> </HOUR>
```

```

    <MINUTE> </MINUTE>
    <TEMP> </TEMP>
</SENSOR>

```

Then punctuation for data in a stream of temperature reports will have the following general structure, where `punct` is a new namespace that we introduce:

```

<punct:SENSOR>
  <ID> </ID>
  <HOUR> </HOUR>
  <MINUTE> </MINUTE>
  <TEMP> </TEMP>
</punct:SENSOR>

```

Namespaces are inherited by subelements. That is, if the parent of an element is in the `punct` namespace, then that element is also in the `punct` namespace.

3.2.2 Punctuation Patterns

A punctuation contains pattern elements that correspond to elements in the data item it is being matched with. A pattern element can take the form of a wildcard, constant, range, or list:

- A *wildcard* denoted by ‘*’ matches any data value.
- A *constant*, illustrated in Figure 3.1, matches values equal to that constant.
- A *range*, illustrated in Figures 3.2 and 3.3, matches values for data item elements contained in the given range. A range can be defined as inclusive (using square brackets), or exclusive (using parentheses), or mixed. An empty value in the first position means the minimum value for the attribute, and an empty value in the second position means the maximum value for the attribute.
- A *list*, illustrated in Figure 3.4, matches values for elements in the list.

A punctuation in our work contains a pattern in each XML element.

```

<!-- Constant: All reports for hour 5
have been sent -->
<punct:SENSOR>
  <ID> * </ID>
  <HOUR> 5 </HOUR>
  <MINUTE> * </MINUTE>
  <TEMP> * </TEMP>
</punct:SENSOR>

```

Figure 3.1: Punctuation with a single constant value (hour) and wildcards.

```

<!-- Range: All reports between 70 and 75
degrees (inclusive) for hour 7 have been sent -->
<punct:SENSOR>
  <ID> * </ID>
  <HOUR> 7 </HOUR>
  <MINUTE> * </MINUTE>
  <TEMP> [70,75] </TEMP>
</punct:SENSOR>

```

Figure 3.2: Punctuation with a constant value (hour) and range (temp).

There are certainly other possible forms for punctuations. Any predicate on the corresponding attribute domain is a candidate for a pattern element, and a punctuation could even relate multiple elements. This form has the useful property that punctuations are closed under intersection (though ranges make union more difficult).

3.2.3 Definition of *match*

Given the punctuation form and the definitions of patterns, we can now define the *match* function. The *match* function iterates through the elements of the data item, and matches each with the corresponding pattern elements of the punctuation. If all corresponding elements match then *match* will return true. Our definition of *match* is shown in Figure 3.5.

We list below matching and non-matching data items for the punctuation from Figure 3.1 in Figure 3.6 and matching and non-matching data items for the punctuation in Figure 3.3 in Figure 3.7 below.

```

<!-- Range with no lower bound: All
temperatures below 75 degrees have been sent for hour 8. -->
<punct:SENSOR>
  <ID> * </ID>
  <HOUR> 8 </HOUR>
  <MINUTE> * </MINUTE>
  <TEMP> [,75) </TEMP>
</punct:SENSOR>

```

Figure 3.3: Punctuation with a range (`temp`) from the minimum value up to, but not including 75, and a constant (`hour`).

```

<!-- List: all temperature reports from
sensors 1, 5, and 6 have been sent -->
<punct:SENSOR>
  <ID> {1,5,6} </ID>
  <HOUR> * </HOUR>
  <MINUTE> * </MINUTE>
  <TEMP> * </TEMP>
</punct:SENSOR>

```

Figure 3.4: Punctuation with a list (`id`).

3.2.4 Implementation of Rules in Query Operators

Note that the rules we have given are only specifications of desired behavior, and do not necessarily embody practical algorithms. (For example, they contain universal quantifications.) Thus there is still some work involved to develop operators that obey punctuation rules.

Sort

In our implementation of *Initial*, we only consider punctuations that have range or wildcard values for the sort attributes and wildcard values for the other attributes. Of those punctuations, one must contain a range that is missing its first element (e.g., `[,c]`) for ascending sort, or missing its second element (e.g., `[c,)`) for descending sort. That punctuation is added to P' . *Initial* then looks for other punctuations whose ranges intersect or are contiguous to a range in P' , and adds each one to P' . This process continues until

<punct:SENSOR>	<SENSOR>	<SENSOR>
<ID> * </ID>	<ID> 1 </ID>	<ID> 1 </ID>
<HOUR> 8 </HOUR>	<HOUR> 8 </HOUR>	<HOUR> 8 </HOUR>
<MINUTE> * </MINUTE>	<MINUTE> 20 </MINUTE>	<MINUTE> 20 </MINUTE>
<TEMP> [,75) </TEMP>	<TEMP> 63 </TEMP>	<TEMP> 75 </TEMP>
</punct:SENSOR>	</SENSOR>	</SENSOR>
(punctuation)	(matching)	(non-matching)

Figure 3.7: Matching and non-matching data items for the punctuation given in Figure 3.3.

output, it purges the data items that were passed, then emits the punctuations that were returned by *Initial*.

Symmetric Hash Join

We implemented the symmetric hash join operator to purge data items based on punctuations that have wildcards for all attributes not participating in the join. For other kinds of punctuations, data items are not purged whether they match those punctuations or not. When an acceptable punctuation arrives on one input, we search the hash table of the other input for join values that match the punctuation on the join element. If the punctuation's join element is a list or a constant, then finding the value is easy. If it is a range, we have to iterate through the values in the hash table one by one, and determine if each one fits in the range. If we find a match, we know we can remove it from the hash table, because there will be no other tuples that join with it from the other input.

The pass rule for join is trivial, because join is a non-blocking operator. Therefore, no changes were made to our implementation of join for the pass rule. Also, we did not implement the propagation rule at this time. In this initial investigation of punctuated streams, the query we intended to execute did not require punctuations to be output from the join. We will discuss our implementation of a propagation invariant for join in our more complete investigation of performance (see Chapter 8).

Union

The only non-trivial rule for the union operator is the propagation rule. To implement propagation, we also must implement a function that intersects two punctuations. Our initial implementation simply compares punctuations for equality. Such a simple implementation is not unrealistic. If multiple stream sources are outputting union-compatible data items, it is likely that they will also emit the same punctuations. When the same punctuation has arrived from both inputs of union, the union operator emits that punctuation.

Group-by

Our implementation of group-by only processes punctuations that have wildcard values for all attributes not participating in the grouping. That is, we only exploited punctuations that marked the end of data items for entire groups. Group data are stored in a hash table, where the values of the grouping attributes are used to generate the hash key. When an appropriate punctuation arrives, we find all groups whose data items match that punctuation in a manner similar to symmetric hash join. Those groups are output, and then removed from state.

Note that many operators have similar functionality. For example, join and group-by filter out punctuations that are not useful. In Chapter 7, we will discuss a new operator (called the *describe* operator, see Section 7.5) that is added to a query plan to output punctuation in the form needed by other operators in a query based on input punctuations.

3.3 Initial Results

We ran tests to verify the advantages and check the overhead of adding punctuations to (terminating) data streams using a variation on the warehouse scenario. In addition to using the Niagara query engine, we have implemented a stream generator called the Firehose, a customizable stream source that outputs XML data. The tests were run on a Pentium III PC running Red Hat Linux version 6.0.

The query we used outputs the maximum temperature reported from any sensor at

the end of each hour. The SQL was:

```
SELECT hour, MAX(temp)
FROM (SELECT * FROM sensor1
      UNION
      SELECT * FROM sensor2
      UNION
      ...
      UNION
      SELECT * FROM sensorN)
GROUP BY hour;
```

The query plan for this query is shown in Figure 3.8.

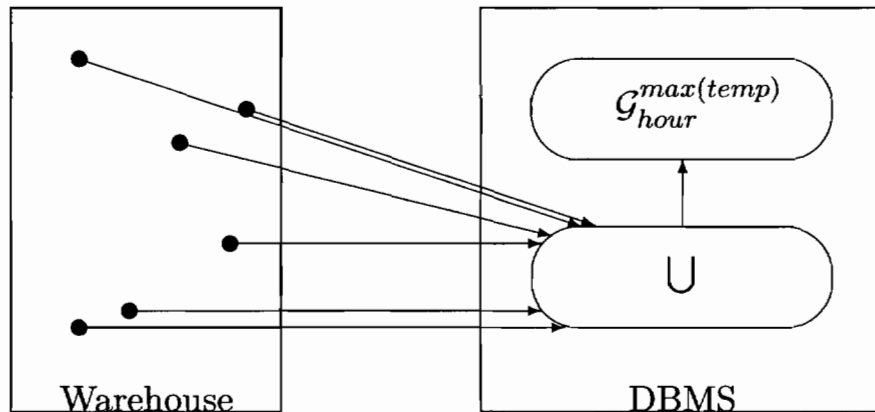


Figure 3.8: Initial query plan used to test punctuation rules.

In this case, our union operator performed duplicate elimination on its inputs. Thus, we could test all three kinds of punctuation rules: pass rules are tested with the group-by operator, and purge and propagation rules are tested with union.

Note that conducting benchmark tests on operators over data streams is not as direct as for operators over relations. Operators over data streams are intended to be able to

run continuously, but benchmarks are designed to run in finite time. Our approach is to benchmark with terminating input and report overall execution time.

We used the Firehose to simulate temperature sensors emitting regular temperature reports. These sensors were programmed to output data items reporting the current temperature at that sensor every minute. Our data generator (the Firehose) also embedded punctuations into the output streams. Each data stream contained one data item for each minute. For these tests, we varied the frequency of embedded punctuations from 0 to 30 per hour per sensor. By changing how far into the stream to embed the first punctuation (the *depth* of the punctuation), and by changing the amount of punctuations in the data stream (the *number* of punctuations), we could determine if levels existed where the cost of processing punctuations was greater than the benefit of punctuations. Since group-by can only process punctuations on the grouping attributes, punctuations emitted from the sensors denote the end of a specific hour. Thus, punctuations that denote the end of a particular minute were filtered out by group-by. We also included the case where punctuations were not emitted. To determine the performance overhead of processing punctuations, test data items were streamed out immediately, much faster than the timestamp value for each data item. The data was over the first 60 hours of temperature reports. The results are shown in Figures 3.9 and 3.10.

We see from Figure 3.9 that embedding punctuations into the stream unblocks the group-by operator, allowing it to output results at the conclusion of each hour. (Note that the “0 Punctuations per Hour” case represented the case where no punctuations exist in the input streams.) Without punctuations, the operator must wait until the input has been read completely before outputting results. We can also see that embedding punctuations does not significantly affect the overall performance of the query. In fact, when using a minimal number of punctuations, the performance improves slightly. We attribute this to the fact that the hash table used in the union operator (for duplicate elimination) maintains a reduced amount of state when punctuations exist in the data stream. If punctuations do not exist, then the hash table grows and grows, forcing it to allocate more and more memory (increasing the number of collisions and corresponding search overhead).

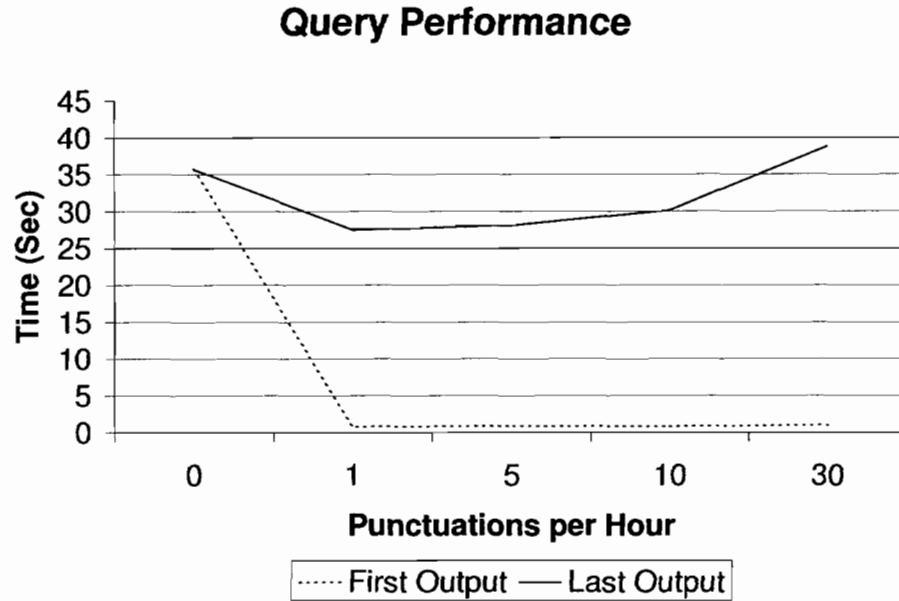


Figure 3.9: Amount of time required to retrieve the first and last data items from the group-by operator. The dashed line indicates the amount of time required to produce the first input, and the solid line indicates the amount of time required to retrieve the entire results.

We see from Figure 3.10 that the amount of memory used by the hash table in the union operator is greatly reduced. For streams that are not punctuated, the size of the hash table grows until the stream ends. If the stream is punctuated, the size of the hash table is in fact bounded (assuming the sensor stream rate is bounded), since each data item that arrives will match a punctuation within one hour.

This initial, ad hoc work showed that punctuations are a promising approach for helping query operators over non-terminating data streams. Separating the kinds of behaviors an operator can follow into three separate rules helped in enhancing operator implementations for handling punctuations. The three types of rules we use: pass, propagation, and

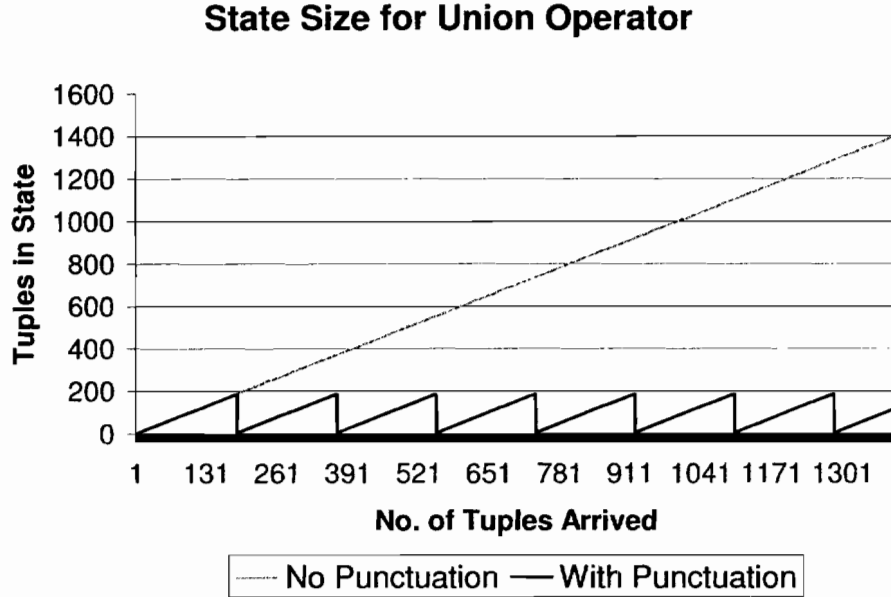


Figure 3.10: Number of data items held in state for the union operator (with duplicate elimination) throughout query execution. The gray line indicates the number of data items held in state when no punctuations are present, and the black line indicates the number of data items held in state when punctuations are present.

purge, address all the ways a punctuation may be handled by a query operator. Additionally, the overhead of processing punctuations is not prohibitive, at least for the query operators we tested.

3.4 Research Directions

After this initial investigation, there were still many questions related to punctuations to address. We discuss those questions in the remaining chapters of this dissertation:

- Could we construct a formal model for a system that processes queries over continuous data streams? A model would help us reason about the behavior and correctness

of stream operators. We present our model in Chapter 4.

- Is there a generic way to produce stream versions of query operators? A generic stream version of query operators would be very useful for easy operator implementation and reasoning about such operators. We will present our framework in Chapter 5.
- How do we know our operators are behaving reasonably in the presence of punctuations? That is, we need a formal way of showing that operators that have been enhanced to process punctuations still behave consistently with the operator's definition. We will address this issue in Chapter 6.
- How do punctuations get embedded into a data stream? Stream sources are programmed to emit data items. They can be enhanced to emit punctuations also, but are there other ways for punctuations to get into a data stream? Further, what issues arise when enhancing implementations of query operators to process punctuations? We address these questions in Chapter 7
- We have seen some initial tests that assess the performance overhead of processing punctuations for two operators (group-by and union). How do punctuations affect the performance of other query operators? Further, how do punctuations that do not help a query affect the performance of the overall query? We discuss the results of more thorough performance tests in Chapter 8.
- How do we know if a particular set of punctuations will “help” a given query over data streams? Users often know what queries they want to ask over a data stream. We want to analyze if available punctuations will unblock or otherwise benefit a query. We address this issue in Chapter 9.

Chapter 4

Overview of Punctuation Semantics

We have seen through our initial investigation that punctuations do indeed help in some instances. What we need now is a formal way to reason about punctuations and how operators behave when processing punctuated streams. We first need to define streams and stream iterators precisely. First, we define a data type for streams. For the moment, we represent a stream as a non-terminating sequence, like the usual *cons*-based formulation of lists, but with no *nil* list. Thus a stream over elements of type T can be specified as $Stream(T) = T \oplus Stream(T)$, where $\oplus$ is an infix constructor.

We will use $[[\dots]]$ to denote stream values to distinguish them from terminating lists. Thus, considering $Stream(Int)$, we can write $1 \oplus 3 \oplus 5 \oplus 7 \oplus \dots$ as $[[1, 3, 5, 7, \dots]]$. We need a function over streams that extracts the first i elements from a stream (and returns it as a *List* of the appropriate type), and so has the type $Stream(T) \times Int \rightarrow List(T)$. Given a stream S , we write $S[i]$ for the list of the first i elements. Thus, if S is as above, $S[3]$ is $[1, 3, 5]$. Further, for $n > i$, we use $S[i \rightarrow n]$ for the list of elements from $i+1$ to n . For S as above, $S[1 \rightarrow 3] = [3, 5]$. Note that $S[i] = S[0 \rightarrow i]$. We use $S@i$ to mean the i th element of the stream. We use $\otimes$ to construct streams from a terminating list and another stream. Thus $[2, 4, 6] \otimes S$ means $2 \oplus 4 \oplus 6 \oplus S$. Note also that $S[i] = S@1 \oplus S@2 \oplus \dots \oplus S@(i-1) \oplus S@i$.

4.1 Stream Iterators

We do not allow arbitrary stream-to-stream functions for operating on data streams. In particular, we want to avoid formulations that must access the entire stream at once. Thus, we introduce stream iterators that access inputs incrementally. Operator $f : Stream(T) \rightarrow$

$Stream(U)$ is a *stream iterator* if there exists some function $q : List(T) \rightarrow List(U)$ (where T is the type for input data items and U is the type for output data items) such that for any $S \in Stream(T)$, $f(S) = (q(S[1]) \otimes q(S[2]) \otimes \dots \otimes q(S[i]) \otimes \dots)$. That is, f is a stream iterator if it can be defined using repeated applications of q over all bounded prefixes of the input stream. For example, it is easily seen that the query operator *select* can be expressed as a stream iterator. If the selection predicate is p , then we define q by:

$$\begin{aligned} q(L ++ [a]) &= [a] \text{ if } p(a) \\ q(L ++ [a]) &= [] \text{ otherwise.} \end{aligned}$$

That is, q just looks at the last element of each prefix, and outputs it if it satisfies the predicate. (Here ‘ $++$ ’ is list concatenation.) The query operator for *dupelim* (duplicate elimination) is also a stream iterator. It uses q such that:

$$\begin{aligned} q(L ++ [a]) &= [a] \text{ if } a \notin L \\ q(L ++ [a]) &= [] \text{ otherwise.} \end{aligned}$$

That is, q outputs the final element in the prefix if it does not appear earlier in the prefix. Note that duplicate elimination requires some knowledge of all data items that have arrived so far. For binary operators such as cross-product and binary union (union over two inputs), we enhance q to take two inputs, so $q : List(T) \times List(V) \rightarrow List(U)$ (where T and U as before, and V is also a type for input data items). Binary union is a stream iterator using q such that:

$$q(L_1 ++ [a_1], L_2 ++ [a_2]) = [a_1, a_2]$$

We show these and other stream iterator definitions in Table 4.1. Note in our definition of union, we made the arbitrary decision to output data items from the left input before data items from the right input, and we assume that data items arrive “in-step” (that is, a_1 and a_2 are both available in sequence). Suppose the value 11 arrives on the left input, then the values 21, 22, and 23 arrive on the right input, and finally the values 12, and 13 arrive from the left input. By our definition, the prefix of the output would be [11, 21, 12, 22, 13, 23]. A more intuitive output would order items in the order they arrive, as in [11, 21, 22, 23, 12, 13]. In the next section, we will discuss other issues with this stream model, and, in Section 4.3, we will present an enhanced model for streams and stream iterators that addresses these issues.

Operator	Definition for q
select_p	$q(L ++ [a]) = \text{if } p(a) \text{ then } [a] \text{ else } []$
$\text{project}_{j_1, \dots, j_n}$	$q(L ++ [a]) = [a(j_1), \dots, a(j_n)]$
dupelim	$q(L ++ [a]) = \text{if } a \notin L \text{ then } [a] \text{ else } []$
union	$q(L_1 ++ [a_1], L_2 ++ [a_2]) = [a_1, a_2]$
intersect	$q(L_1 ++ [a_1], L_2 ++ [a_2]) = \text{if } a_1 \in (L_2 ++ [a_2]) \text{ then } [a_1] \text{ else } [] ++ \text{if } a_2 \in (L_1 ++ [a_1]) \text{ then } [a_2] \text{ else } []$
cross-product	$q(L_1 ++ [a_1], L_2 ++ [a_2]) = [a_1 \times b b \in (L_2 ++ [a_2])] ++ [c \times a_2 c \in (L_1 ++ [a_1])]$

Table 4.1: Initial definitions for stream iterators

It should be obvious that `sort` cannot be expressed as a stream iterator. `Sort`, even if well-defined (as over a stream of natural numbers), cannot output results after reading only a bounded prefix of the input. Similarly, the query operators `difference` and `group-by` also do not have stream-iterator analogues. We can never output the aggregate value for a group when reading from a non-terminating stream because there may still be members of the group coming later in the stream. (We are skirting an important issue in these examples, namely what is an appropriate extension of a table operator to non-terminating streams. We are assuming here that stream versions of `select` and `dupelim` should behave as described. In Chapter 6 will formally define what it means for a stream iterator to be *faithful* to a table operator.)

4.2 Representation Issues

The model of a stream as a non-terminating sequence has an attractive simplicity to it and is directly supported in languages with lazy evaluation such as Haskell. However, we find it has certain limitations for defining and implementing stream iterators. Those limitations concern both modelling capabilities and pragmatic programming issues.

Representation of terminating streams: A stream that actually has only a bounded number of elements in it is not directly representable as a non-terminating sequence. We considered various alternatives, such as “stalling” after the last input element, padding the sequence with an unbounded number of null values, or using both terminating and non-terminating sequences. All of these options tended to complicate

the reasoning and programming models for streams.

Independence of iterator and stream rates: We do not want to build any assumption into our model that stream iterators are synchronized with stream arrival. There may be multiple arrivals between iterator steps. More problematically, there may be no stream elements arriving at a given step. With a non-terminating sequence representation, if no input is available from an input stream, the operator must block waiting for input to arrive (creating problems for operators with multiple inputs) or require a change to the sequence interface.

Interleaving of multiple inputs: Just as we do not want to assume an input stream is synchronized with operator iterations, different inputs need not necessarily proceed “in step”. “Desynchronizing” multiple inputs requires some kind of extension to ensure fairness and avoid blocking on one stream when input is available on the other. We saw this issue in the discussion of union above.

4.3 Enhancing Our Model of Streams and Stream Iterators

We enhanced our model of streams and stream iterators to address the issues listed above.

4.3.1 An Enhancement for Streams

Our refined model for a data stream is as a non-terminating sequence of bounded lists of elements — sort of a “sliced list”. Thus, for example, the sequence $[[1, 2, 3, 4, 5, \dots]]$ might appear in sliced form as $[[[1, 2], [3], []], [4, 5], \dots]]$ or as $[[[1, 2, 3], [4, 5], \dots]]$ or in myriad other forms.

We still use the $S[i]$ to talk about a bounded prefix of a stream, but now it refers to the first i slices. So, for example, if $S = [[1, 2], [3], [], [4, 5], [6], \dots]]$, then $S[4] = [1, 2, 3, 4, 5]$. Stream iterators take a slice from each input in turn. Thus, an iterator with n inputs will have consumed $n * i$ slices after completing stage i . Similarly, $S@i$ refers to the i th slice. For example, $S@4 = [4, 5]$.

This sliced representation overcomes the problems mentioned above. It can model a terminating stream using a trailing sequence of empty slices: $[[1, 2], [3], [], [], [], \dots]]$.

The sliced representation also models variability in stream arrivals relative to iterator steps. If the input is arriving at a rate faster than the iterator can process, then the next slice will contain many data items. Finally, the sliced representation can capture different interleavings of multiple inputs with alternative slicings of the streams. We use this representation mainly for modelling stream iterators and for reasoning about them. In our prototype (see Chapter 7 for details), operators use non-blocking reads from buffers to retrieve data.

We did consider a variety of other representations, but they all seemed to add complexity without discernable improvement in modelling stream behavior or iterator implementation. Interleaving all inputs into a single sequence requires tagging elements with an identifier for the input they belong to, and development of a notion of “fair merge” of streams. We considered adding an extra input sequence to each iterator that could indicate which stream sequence to access next or where input was available, but that approach led to considerable complication in iterator implementation. Labelling stream elements with an arrival time proved to be essentially equivalent to sliced streams, but our sliced representation proved easier to manage through queries involving multiple operators. We also considered capturing input interleaving inside iterators rather than in the data stream representation. Strict element-at-a-time alternation was unrealistic. Deciding which input to read from next by a random choose function lead to problems in repeatability.

4.3.2 An Enhancement for Stream Iterators

The change to stream iterators is to provide less than the entire stream prefix to the operator at each iteration. Some operators, such as select, do not need anything but the most recent input element to figure out their next output. Other operators only need a summary of the prefix. For example, dupelim only needs the distinct values in the prefix. Thus, in our actual formulation, stream iterators keep state from iteration to iteration. That state might be empty, or all input seen to that point, or some summary. The explicit representation of state also points up that there are certain query operators with stream analogues, but where the stream iterator has to keep track of arbitrary amounts of the prefix. For example, the state in dupelim can grow without bound.

Keeping state does not change the set of stream functions that can be expressed. Any function in the prefix version can be converted to the stateful version by keeping prior input in the state. A function in the stateful version can be adapted to the prefix form by computing its state from the prefix on each iteration. (Obviously, a function might run with different efficiency in the different versions.)

We now redefine a stream iterator based on the changes discussed earlier. The first change is to the function q . It still takes a list input, but now the list is only the i th slice of the input, rather than the i th prefix of the input. (That is, for a stream S , q takes $S@i$, rather than $S[i]$.)

The second change is more involved. The q function is also enhanced to take some state value. The type of this value is different for different stream iterators; For now we will call the type ST . We need a function that returns the updated state given a slice of the input and the current state: An operator $f : Stream(T) \rightarrow Stream(U)$ is a *stream iterator* if, for some type ST , there exist two functions $q : List(T) \times ST \rightarrow List(U)$ and $r : List(T) \times ST \rightarrow ST$, and some initial state $st_0 \in ST$ such that, for any $S \in Stream(T)$, $f(S) = (q(S@1, st_0) \otimes q(S@2, st_1) \otimes \dots \otimes q(S@i, st_{i-1}) \otimes \dots)$, where each $st_i = r(S@i, st_{i-1})$. For a binary stream iterator the type of q is $List(T) \times List(V) \times ST \rightarrow List(U)$, and the type of r is $List(T) \times List(V) \times ST \rightarrow ST$. The “threading” of state for unary operators from one invocation of q to the next invocation of q is illustrated in Figure 4.1.

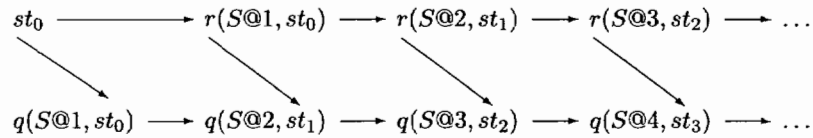


Figure 4.1: Model of how output and state are threaded between iterations of q and r , where q computes new output and r computes new state.

Now, we redefine particular stream iterators. The definition for r and the initial state is shown in Table 4.2 and the definition for q is shown in Table 4.3. Examining Table 4.2 reveals that the stream iterator definitions for *dupelim*, *intersect*, and *cross-product*

require potentially an unbounded amount of state for unbounded input.

Operator	Initial State	Definition for r
select_p	$st_0 = []$	$r(A, []) = []$
$\text{project}_{j_1, \dots, j_n}$	$st_0 = []$	$r(A, []) = []$
dupelim	$st_0 = \{\}$	$r(A, st) = A \cup st$
union	$st_0 = []$	$r(A_1, A_2, []) = []$
intersect	$st_0 = (\{\}, \{\})$	$r(A_1, A_2, (L, R)) = (L \cap A_1, R \cap A_2)$
cross-product	$st_0 = ([], [])$	$r(A_1, A_2, (L, R)) = (L ++ A_1, R ++ A_2)$

Table 4.2: State function (r) definitions of stream iterators. Note that now we will use a duplicate-preserving implementation of union.

Operator	Definition for q
select_p	$q(A, []) = [a a \in A \wedge p(a)]$
$\text{project}_{j_1, \dots, j_n}$	$q(A, []) = [<a(j_1), \dots, a(j_n)> a \in A]$
dupelim	$q(A, st) = [a a \in A \wedge a \notin st]$
union	$q(A_1, A_2, []) = [a a \in A_1 \vee a \in A_2]$
intersect	$q(A_1, A_2, (L, R)) = [a_1 a_1 \in A_1 \wedge a_1 \in (R ++ A_2)] ++ [a_2 a_2 \in A_2 \wedge a_2 \in (L ++ A_1)]$
cross-product	$q(A_1, A_2, (L, R)) = [a_1 \times b a_1 \in A_1 \wedge b \in (R ++ A_2)] ++ [c \times a_2 c \in (L ++ A_1) \wedge a_2 \in A_2]$

Table 4.3: Output function (q) definitions of stream iterators.

The enhancements we have discussed improve the implementation of many different stream iterators by only passing the current slice of input to the stream iterator and allowing the iterator to manage its own state. However, even with these enhancements, there still exist operators that cannot be implemented as stream iterators, including group-by, sort, and difference. We still need more enhancements before we can define a stream iterator for these operators.

4.4 Including Punctuation Semantics in our Stream Iterator Model

Our model of stream iterators helps us to recognize the two kinds of operators that are inappropriate for processing unbounded streams. First, there is no stream iterator analog

for a blocking operator such as sort or group-by. We cannot define a function q for a blocking operator such that, by repeatedly applying q to bounded prefixes of the input stream, we get some non-empty output. Second, operators whose state grows without bound are easily visible. Consider the definition of `dupelim` given in Table 4.3. For each application of r , the size of state is non-decreasing, so it clearly has potential to grow without bound.

We enhance our model of stream iterators by including functions for exploiting punctuations. In our original model, the functions q and r are called for each slice in the input stream(s). To include punctuation semantics, we add three new functions to describe each of the three punctuation behaviors (`pass`, `prop`, `keep`). Note that we use the function `keep` in place of the the purge rule from our previous investigation. We find that it is more intuitive to reason about what data should remain in state than what data should be removed from state. In Table 4.4, we give examples of the `pass`, `keep`, and `prop` functions for the stream iterator `dupelim`. Later, we will give more formal definitions of behaviors for `dupelim` as well as other operators. Now, for each slice in the input stream(s), five functions are called (q and r as before, as well as `pass`, `prop`, and `keep`). In the next section, we will discuss a more formal representation for punctuations. We will show how punctuation semantics are added to our model of stream iterators in Section 4.6.

<code>pass</code>	Nothing (the function q already output the appropriate data items)
<code>prop</code>	All punctuations as they arrive
<code>keep</code>	The set of data items that have arrive and do not match any punctuations that have arrived

Table 4.4: Punctuation behaviors for the stream iterator `dupelim`.

4.5 Punctuation Representation and Manipulation

We have presented punctuations as predicates on stream elements that should evaluate to *false* for all elements after the punctuation. Thus we might represent punctuations as “black box” Boolean functions. However, in our work we represent punctuations as

concrete data structures to allow their easy storage, searching and manipulation. A data item is a tuple of scalars, and a punctuation is a tuple of patterns, where each pattern corresponds to an attribute of a data item in a data stream.

To define the set of valid patterns we use for punctuations over a set A , we introduce two patterns $*$ and ϵ , where $\{*, \epsilon\} \cap A = \emptyset$. Thus, the set of patterns we use is defined as:

$$\Pi_A = \{*\} \cup \{\epsilon\} \cup \{a_1 | a_1 \in A\} \cup \{A' | A' \subseteq A \wedge A' \text{ is finite}\}$$

The *matchPat* function takes an element from A and a pattern from Π_A and returns *true* if the element matches the pattern. The function *matchPat* is defined for each kind of pattern, as follows:

$$\begin{aligned} \text{matchPat} &:: A \times \Pi_A \rightarrow \text{boolean} \\ \text{matchPat}(a, *) &= \text{true} \\ \text{matchPat}(a, \epsilon) &= \text{false} \\ \text{matchPat}(a, a_1) &= a == a_1 \\ \text{matchPat}(a, A') &= \text{true if } a \in A', \text{ false otherwise} \end{aligned}$$

If A is a totally ordered set with some relation $\leq$, then the set of valid patterns is enlarged. We introduce two new symbols $\perp$ and $\top$, where $\{\perp, \top\} \cap A = \emptyset \wedge \forall a \in A, \perp \leq a \wedge a \leq \top$. Now the set of valid patterns over A is defined as:

$$\begin{aligned} \Pi_A^+ &= \{*\} \cup \{\epsilon\} \cup \{a_1 | a_1 \in A\} \cup \{A' | A' \subseteq A\} \cup \\ &\quad \{(a_1, a_2) | a_1, a_2 \in A \wedge a_1 \leq a_2\} \cup \{(\perp, a_2) | a_2 \in A\} \cup \{(a_1, \top) | a_1 \in A\} \end{aligned}$$

For totally ordered sets, *matchPat* is enhanced as follows:

$$\begin{aligned} \text{matchPat}(a, (a_1, a_2)) &= a_1 \leq a \wedge a \leq a_2 \\ \text{matchPat}(a, (\perp, a_2)) &= a \leq a_2 \\ \text{matchPat}(a, (a_1, \top)) &= a_1 \leq a \end{aligned}$$

For example, for integer data items, the pattern 5 matches the integer value 5, and the pattern $(0, \top)$ matches all integer values greater than or equal to 0.

There are many possible families of patterns (or other representations) that can be used to make up a punctuation. The family we use has the important property that it is closed under intersection (and, therefore, punctuations are closed under intersection).

A *dataspace* D for a data stream R is $A_1 \times A_2 \times \dots \times A_n$, where each A_i is the domain of the i th attribute for data items in R . A *subspace* S over D is a subset of data items in D . For dataspace D , the *punctuation space* $\mathbb{P}_D = \Pi_{A_1} \times \Pi_{A_2} \times \dots \times \Pi_{A_n}$ is the set of possible punctuations that can be defined for dataspace D .

The *match* function determines when an item from the dataspace matches a punctuation from the punctuation space by comparing each attribute of the data item with each corresponding pattern in the punctuation. For D :

$$\begin{aligned} \text{match} &:: D \times \mathbb{P}_D \rightarrow \text{boolean} \\ \text{match}(d, p) &= \text{matchPat}(d(1), p(1)) \wedge \text{matchPat}(d(2), p(2)) \wedge \dots \\ &\quad \wedge \text{matchPat}(d(n), p(n)) \end{aligned}$$

Note that these definitions of *matchPat* and *match* correspond to the definition in Figure 3.5. Using the example patterns 5 and $(0, \top)$ from above, and the dataspace $\text{Int} \times \text{Int}$, we can form the punctuation $p = \langle 5, (0, \top) \rangle$. For the data items $\langle 5, 5 \rangle$, $\langle 6, 5 \rangle$, and $\langle 5, -1 \rangle$, we have that $\text{match}(\langle 5, 5 \rangle, p) = \text{true}$, but $\text{match}(\langle 6, 5 \rangle, p) = \text{false}$ and $\text{match}(\langle 5, -1 \rangle, p) = \text{false}$.

Some operators, such as union, need to be able to combine multiple punctuations. Given two punctuations, we need a function that outputs a punctuation p such that all data items that match both input punctuations also match p , and vice versa. We call this function *combine*, and specify it as follows:

$$\text{combine}(p_1, p_2) = p \mid \forall d, \text{match}(d, p_1) \wedge \text{match}(d, p_2) \Leftrightarrow \text{match}(d, p).$$

Note that $\text{combine}(p_1, p_1)$ may not exist for some arbitrary punctuation pattern scheme. It will exist for our pattern scheme.

For example, given $p = \langle 5, (0, 15) \rangle$ and $p' = \langle 5, (-10, 10) \rangle$, $\text{combine}(p, p') = \langle 5, (0, 10) \rangle$.

Table 4.5 lists helper functions we use to operate on lists of punctuations and data items.

Function	Return Value
$data(xs)$	data items in xs
$puncts(xs)$	punctuations in xs
$nomatch(t, p)$	$\neg match(t, p)$
$setMatch(t, ps)$	True if $\exists p \in ps$ such that $match(t, p)$
$setNomatch(t, ps)$	True if $\forall p \in ps, nomatch(t, p)$
$setMatchTs(ts, ps)$	$\{t t \in ts \wedge setMatch(t, ps)\}$
$setNomatchTs(ts, ps)$	$\{t t \in ts \wedge setNomatch(t, ps)\}$
$setNomatchPs(ts, ps)$	$\{p p \in ps \wedge \forall t \in ts, nomatch(t, p)\}$
$setCombine(ps_1, ps_2)$	$\{combine(p_1, p_2) p_1 \in ps_1 \wedge p_2 \in ps_2\}$

Table 4.5: Helper functions used on list of data items and punctuations, where t is a data item, ts is a bounded list of data items, p is a punctuation, ps is a bounded list of punctuations, and xs is a bounded list of containing both data items and punctuations.

We can now characterize the kinds of punctuated streams that are properly formed. A *punctuated stream* is a stream that contains both data items and punctuations. A punctuated stream S is *grammatical* if for all i and for every $j > i$, if $p \in S[i]$ and $d \in S[i \rightarrow j]$ then $nomatch(d, p)$. In this work, we assume all punctuated streams are grammatical. Thus it is important that all stream sources and stream iterators output grammatical streams. Note that a stream that does not contain punctuations is grammatical.

4.6 Punctuation-Aware Stream Iterators

In this section, we present how our model of stream iterators is enhanced to support punctuations. The first issue we must address is how to represent items in a stream. Data items are tuples of scalars, and punctuations are tuples of patterns. We will define a *Tuple* to be a pair, where the first item indicates if the tuple is a data item or a punctuation, and the second item is the actual value (data item or punctuation). So, given a dataspace D and a punctuation space $\mathbb{P}_D$, $Tuple(D, \mathbb{P}_D) : (N, V)$, where $N = \{I, P\}$. When $N = I$, we know that V is a data item, and so $V \in D$. When $N = P$, we know that V is a punctuation, and so $V \in \mathbb{P}_D$.

Recall the infix constructors $\oplus : T \times Stream(T) \rightarrow Stream(T)$ and $\otimes : List(T) \times Stream(T) \rightarrow Stream(T)$. We overload these to operate on data items as well as punctuations, as follows:

$$\begin{aligned}
\oplus &: D \times \text{Stream}(\text{Tuple}(D, \mathbb{P}_D)) \rightarrow \text{Stream}(\text{Tuple}(D, \mathbb{P}_D)) \\
\oplus &: \mathbb{P}_D \times \text{Stream}(\text{Tuple}(D, \mathbb{P}_D)) \rightarrow \text{Stream}(\text{Tuple}(D, \mathbb{P}_D)) \\
\otimes &: \text{List}(D) \times \text{Stream}(\text{Tuple}(D, \mathbb{P}_D)) \rightarrow \text{Stream}(\text{Tuple}(D, \mathbb{P}_D)) \\
\otimes &: \text{List}(\mathbb{P}_D) \times \text{Stream}(\text{Tuple}(D, \mathbb{P}_D)) \rightarrow \text{Stream}(\text{Tuple}(D, \mathbb{P}_D))
\end{aligned}$$

Without punctuation, a stream iterator is defined using two functions q and r that represent the output and state, respectively, when data items arrive. We add the punctuation functions $pass$, $prop$, and $keep$ to define how punctuations affect the behavior of the stream iterator as well. Note that these functions define *additional* behavior, so our earlier definitions of q and r in Tables 4.2 and 4.3 are still correct for punctuated streams (with minor adjustments to some state data types to also store punctuations).

Now we can define punctuation-aware stream iterators. A function $f : \text{Stream}(\text{Tuple}(D_I, \mathbb{P}_{D_I})) \rightarrow \text{Stream}(\text{Tuple}(D_O, \mathbb{P}_{D_O}))$ is a *punctuation-aware stream iterator* if, in addition to the existence of two functions $q : \text{List}(D_I) \times ST \rightarrow \text{List}(D_O)$ and $r : \text{List}(D_I) \times ST \rightarrow ST$, and some initial state $st_0 \in ST$, there also exist three functions $pass : \text{List}(\mathbb{P}_{D_I}) \times ST \rightarrow \text{List}(D_O)$, $prop : \text{List}(\mathbb{P}_{D_I}) \times ST \rightarrow ST$, and $keep : \text{List}(\mathbb{P}_{D_I}) \times ST \rightarrow \text{List}(\mathbb{P}_{D_I})$ such that, for any $S \in \text{Stream}(\text{Tuple}(D_I, \mathbb{P}_{D_I}))$:

$$\begin{aligned}
f(S) = & (q(\text{data}(S@1), st_0) \otimes pass(\text{puncts}(S@1), st_0) \otimes prop(\text{puncts}(S@1), st_0) \otimes \\
& q(\text{data}(S@2), st_1) \otimes pass(\text{puncts}(S@2), st_1) \otimes prop(\text{puncts}(S@2), st_1) \otimes \\
& \dots \otimes q(S@i, st_{i-1}) \otimes pass(\text{puncts}(S@i), st_{i-1}) \otimes prop(\text{puncts}(S@i), st_{i-1}) \otimes \dots)
\end{aligned}$$

where $st_i = keep(\text{puncts}(S@i), r(\text{data}(S@i), st_{i-1}))$. For unary operators such as group-by that do not have stream-iterator analogues we use $q(A, st) = []$ and $r(A, st) = A ++ st$. The q and r functions for binary stream iterators are unchanged. (For difference, we use $q(A, B, st) = []$ and $r(A, B, (st_A, st_B)) = (A ++ st_A, B ++ st_B)$). For each input of a binary stream iterator, we define $lpass$, $lprop$, and $lkeep$ functions for the left input and $rpass$, $rprop$, and $rkeep$ functions for the right input.

The three punctuation behaviors are manifested in our work in two ways. In Chapter 5, we present an implementation of our generic framework for punctuated-stream iterators using the functional programming language Haskell. We use the function `step` to emulate the q and r functions described above. Additionally, we have helper functions, usually

called `pass`, `prop`, and `keep` that plug in to the generic framework to customize it for a particular iterator. These functions are called repeatedly as the iterator works its way through the inputs. Through this framework, we see more concretely how our model for stream iterators is enhanced for punctuations.

In Chapter 6, we use these concepts to prove correctness of punctuated-stream iterators. There we define `pass`, propagation, and `keep` *invariants* that specify the cumulative behavior of an iterator enhanced for punctuated streams. We generally call the formal invariants *cpass*, *cprop*, and *ckkeep*. Our basic proof strategy for the correctness of an operator in our framework starts with defining appropriate invariants. We show that a stream iterator that obeys the invariants (and a general minimality condition) is correct according to the notions we define. We then show that a stream iterator defined in our framework (via `pass`, `prop`, `keep` functions) obeys the cumulative invariants. This two-step approach allows us to prove correctness for multiple implementations of a stream iterator by proving only that each implementation obeys the cumulative invariants.

Note that `pass` and propagate behavior can be defined at the logical level for an operator, while `keep` is particular to a specific operator implementation. (However, operator state is usually related to operator inputs.)

Note also that the correspondence between *cpass* and *ckkeep* rules and the `pass` and `keep` functions may not be quite direct. Our iterator framework is based upon the definitions for the non-punctuated case, where the behavior of *q* and *r* are combined into the `step` function. Thus the *cpass* and *ckkeep* definitions actually specify the desired behavior of the `pass` and `keep` functions in combination with the base `step` function.

Chapter 5

A Framework for Stream Iterators

As discussed in Chapter 3, we initially made ad hoc extensions to the Niagara Query Engine [NDM⁺00] to test the feasibility of our approach. After seeing encouraging results, we tried to abstract the behavior of stream iterators into a high-level model. The result is a common control structure for stream iterators, which is customized by plugging in helper functions for each operator. Our model was implemented using the programming language Haskell [Hud00], a *lazy evaluation* functional language. We chose Haskell for three reasons: First, lazy evaluation languages do not evaluate expressions until required. Thus, they are suitable for modelling systems that process continuous data streams, and have been used previously [HC98, LM89, PMC89] to model data stream processing. This behavior is similar to Graefe’s query iterator function *next* [Gra93], which retrieves only the next item from its input without having to materialize the entire stream. Second, because it is a functional language, we can formulate the general behavior of all stream iterators in a single function, and pass the specifics of each operator as function arguments to the general function. Finally, because it is a language with a cleaner semantics (at least the part we use) as compared to iterative languages such as Java, it helps us to prove our implementations conform to the appropriate punctuation behaviors.

We give a brief explanation of Haskell features used in our framework that are defined in the standard Haskell library (Prelude.hs). The `fst` function takes a tuple and returns the first value from the tuple. For example, `fst (1,2)` returns the value 1. The infix function `++` concatenates two lists. The infix function `\\` removes items from its first argument that exist in its second argument. The function `nub` eliminates duplicates from a list. We also use the infix `:` operator, which takes an item and a list and a new list

with the item to the head of the original list. We explain other Haskell features as they are encountered.

We also make extensive use of the pattern matching feature of Haskell when defining functions. For example, constants of type `List` are denoted with square brackets: `[]` is the empty list, `['a']` is a list with a single character element, and `[1,2,3]` is a list containing three integers. Suppose we want a function that takes a list and returns its length. We could define this function as follows:

```
mylength :: [a] -> Int
mylength [] = 0
mylength (x:xs) = 1 + mylength xs
```

(The function is defined as `mylength` to distinguish it from the predefined `length` function that is defined in `Prelude.hs`.)

The first line gives the function's type: `mylength` takes a list of items of some type `a` and returns an integer. The second line defines `mylength` for the empty list: the length of an empty list is 0. The third line defines `mylength` for non-empty lists. That is, for lists that contain some element `x` at the head of a list `xs`, the function `mylength` adds one to return value of the recursive call to `mylength` applied to `xs`.

We also take advantage of classes in Haskell. In Haskell, a *class* defines a set of operations, and a data type that supports those operations is called an *instance*. For example, the `Eq` class specifies the function `==`, and data types that want to support comparison must define what the function `==` means for that type. A second example is the `Show` class which defines the function `show` to convert objects of that data type to a string (generally for displaying).

We also use the `Maybe` data type defined in the library `Prelude.hs`. The `Maybe` data type has two constructors: The `Just` constructor takes a data value, and the `Nothing` constructor takes no arguments. This type is useful for computations that, based on their input, may or may not return a result. If the result is valid, then we use the `Just` constructor. If not, then we can use the `Nothing` constructor. For example, a function that returns the position of an integer in a list of integers will return a position value if

the item exists in the list, but no value if the item is not in the list. This function could be defined as:

```
findPos :: Int -> [Int] -> Maybe Int
findPos y [] = Nothing
findPos y (x:xs) = if (x==y) then Just 1
                    else incrMaybe (findPos y xs)
  where incrMaybe (Just x) = Just (x+1)
        incrMaybe Nothing = Nothing
```

A user-defined data type in Haskell can be declared using either the `type` or `data` keywords. The `type` keyword is used to declare type synonyms. For example, the type for a warehouse sensor report is a 4-tuple of integers (SensorID, Hour, Minute, Temperature) can be declared as: `type Report = (Int,Int,Int,Int)`. The `data` keyword is used to declare more complex data types, in particular, recursive data types. For example, in the network monitoring scenario we define the data type for IP addresses as: `data N_IPAddr = IPAddr(Int,Int,Int,Int)`, where `IPAddr` is the constructor for objects of type `N_IPAddr`. In our framework, `N_IPAddr` could have been declared using `type` as a type synonym. By making it a more complex type, we are able to take advantage of pattern matching on the constructor in function declarations. In this case, we define `N_IPAddr` as an instance of the `Show` class, so that we can display IP addresses not as 4-tuples, but in the more standard `XXX.XXX.XXX.XXX` format as follows:

```
instance Show N_IPAddr
  where show (IPAddr (n1,n2,n3,n4)) =
        show n1 ++ "." ++ show n2 ++ "." ++
        show n3 ++ "." ++ show n4
```

5.1 A Formulation for Stream Iterators

We first develop our framework without punctuations. A `Stream` object is defined as a list of lists for some type `a`:

```
type Stream a = [[a]]
```

Thus, our definition of the `Stream` type resembles the sliced-list representation in our stream iterator model. Inner lists of a `Stream` are terminating, but the outer list may be non-terminating. Stream iterators in our framework (without punctuations) closely resemble Parker’s stream transducers [Par90]. A unary stream iterator is a 3-tuple (`initial_state`, `step`, `final`) that contains an initial state and two functions, where:

- The value `initial_state` is the state of the iterator before any data items have arrived on the input stream. This value is the same as the initial state (st_0) in our enhanced definition of stream iterators (see Section 4.3).
- The function `step` is called when new data arrives from the input stream. It takes a slice from the input stream and the current state, and returns any new output data items and a modified state. The `step` function performs the functionality of the output function (q) and the state function (r) in our enhanced definition of stream iterators (again see Section 4.3).
- The function `final` is called when the stream ends (i.e., when the stream is `[]`). It takes only the current state, and returns any new data items and a modified state. That `final` returns a modified state may seem a bit strange at first. For binary operators, `final` is called when one of the inputs has been read completely. The updated state will be used as the iterator continues to read from the other input.

We encapsulate the stream iterator functions in a single data type, `Basic`, defined as follows:

```
data Basic state input output =
  B ([input] -> state -> ([output],state)) -- step function
    (state -> ([output],state))           -- final function
```

That is, the constructor for the `Basic` data type is `B`, which takes two arguments: the `step` function and the `final` function.

The general behavior of all unary stream iterators is modelled in the `unary` function, which takes the initial state, an object of type `Basic`, and an input stream, and produces

an output stream. Our definition of `unary` is:

```
unary :: state -> (Basic state input output) -> Stream input ->
      Stream output
unary st (B step final) [] = [fst(final st)]
unary st (B step final) (xs:rest) =
    [out] ++ (unary s2 (B step final) rest)
    where (out,s2) = step xs s
```

where `state` is the data type for the state maintained by the stream iterator, and `input` and `output` are the data types contained in the input and output streams. Note that, in our framework, a stream iterator is a stream-to-stream function. In this way, we can construct arbitrarily complex stream iterators using combinations of basic stream iterators.

The first two arguments to `unary` define some specific stream iterator (such as `select` or `dupelim`). The third argument is the actual input data stream. The first definition of `unary` handles the case where the input stream has ended (that is, the input stream equals the empty list `[]`). In that case, we simply call `final` with the current state. Any final data items that must be output are handled here. Note that `final` need never be called in some cases. For example, `final` will never be called when executing the iterator `select` on a non-terminating stream. The second case handles non-empty stream inputs. In this case, the next slice of input and the current state are passed on to the `step` function to get any new output and an updated state. Then `unary` is recursively called with the remainder of the stream input.

For example, the definition of the stream iterator for duplicate elimination is:

```
dupelimS :: Eq i => Stream i -> Stream i
dupelimS = unary [] (B step final)
    where step ts st = ((nub ts \\ st), union st ts)
          final st = ([], st)
```

The `Eq` clause in the first line indicates that items of type `i` must support comparison for equality. We maintain a list of all unique data items that have arrived from the input so

far in state (`st`), with the empty list as the initial state. The `step` function returns any input data items not currently in the state and a new state containing the union of the new data items and the data items in the original state. The `final` function simply clears the state.

Let us see how `dupelimS` works with an example. Suppose we have the input stream `s = [| [1,5], [3], [], [5,6,7], ...|]`. To evaluate `dupelimS s`, each slice of the stream and the current state will be passed to the `step` function as follows:

	function	input	state	output	newstate
	<code>step</code>	<code>[1,5]</code>	<code>[]</code>	<code>[1,5]</code>	<code>[1,5]</code>
	<code>step</code>	<code>[3]</code>	<code>[1,5]</code>	<code>[3]</code>	<code>[1,5,3]</code>
	<code>step</code>	<code>[]</code>	<code>[1,5,3]</code>	<code>[]</code>	<code>[1,5,3]</code>
	<code>step</code>	<code>[5,6,7]</code>	<code>[1,5,3]</code>	<code>[6,7]</code>	<code>[1,5,3,6,7]</code>
	...				

Thus, the output for `dupelimS s` is `[| [1,5], [3], [], [6,7], ...|]`.

A binary stream iterator is defined by its initial state and four functions: (`stepL`, `stepR`, `finalL`, `finalR`). The meanings of `stepL` and `stepR` are the same as the `step` function for a unary operator, but defined here for each of the binary iterator's inputs. Likewise, `finalL` and `finalR` are the same as `final` for a unary operator, defined for each input. The general operation of a binary stream iterator reads alternatively from each input. When new data arrives from one of the inputs, the appropriate `step` function is called. If one input ends, the appropriate `final` is called, and then the other input is read until completion, and then its `final` is called. The formal definition of `binary` is similar to `unary`, as follows:

```

binary :: state -> (Basic state inputL output) ->
              (Basic state inputR output) ->
              Stream inputL -> Stream inputR -> Stream output

binary st (B stepL finalL) (B stepR finalR) [] rs =
  [out] ++ (unary s1 (B stepR finalR) rs)

```

```

    where (out,s1) = finalL st
  binary st (B stepL finalL) (B stepR finalR) (ls:rest) rs =
    [out] ++ (binary s2 (B stepR finalR) (B stepL finalL) rs rest)
    where (out,s2) = stepL ls st

```

The first equation in the definition of `binary` handles the case where one of the inputs has ended. In this case, a binary stream iterator becomes a unary stream iterator, so we call `unary` with current state and the functions for the other input, as well as the other (possibly non-empty) input stream. The second equation handles a non-empty input stream. We call the `step` function for that input to get any output data items and an undated state. Then we recursively call `binary` with the input functions and streams switched, so that the other input is handled.

For example, consider the following binary stream iterator for the difference operator:

```

diffS x y = dupelimS (_diffS x y)
  where _diffS = binary (False, [], []) (B stepL finalL) (B stepR finalR)
    stepL ts (False, xs, ys) = ([], (False, ts ++ xs, ys))
    stepL ts (True, xs, ys) = (nub res, (True, [], ys))
      where res = snd (partition ('elem' ys) ts)
    stepR ts (fYs, xs, ys) = ([], (fYs, xs, ts ++ ys))
    finalL (f, xs, ys) = ([], (f, xs, ys))
    finalR (f, xs, ys) = (nub res, (True, [], ys))
      where res = snd (partition ('elem' ys) xs)

```

The infix function `elem` is a predicate that returns `true` if the first argument is an element in the second argument. Its counterpart function `notElem` returns `true` if the first argument is not in the second argument.

The state contains a boolean indicating if the negative input stream has ended, and a list of the data items that have arrived for each input. The `stepL` and `stepR` functions add the incoming data to the appropriate list. Additionally, if the negative input has ended, the definition of the `stepL` function (when the boolean value in state is `True`) outputs any new data items that are not equal to data items from the negative input. The `finalR` function outputs any data items from the positive input that are not equal to any data

items from the negative input, and modifies state by setting the flag indicating the end of the negative input to `true` and clearing out the list for the positive input (since those data items were already output).

This example illustrates two points: First, it is clear that, in general, each input must have its own implementation of `step` and `final`. Often the tasks for these functions are different for each input. Second, `final` must return a state value, because that value may be needed for processing the other input.

5.2 Implementation of Punctuations

We now enhance our stream-iterator framework to process punctuated data streams. We will first discuss how punctuations are represented in our framework. Then, in the next section, we discuss how stream iterators in our framework are enhanced. A punctuation is a tuple of patterns. In our representation, we support five patterns: constant, wildcard, range, list, and empty. Thus, given some type `a`, we define the `Pattern` data type with five constructors:

```
data Pattern a = Literal a
               | Wildcard
               | Range (a, a)
               | ListPat [a]
               | Empty
```

where the `match` function is defined for each pattern. Given some value `v` and a pattern value over `v`, `match` is defined as follows:

```
match :: Eq a => a -> Pattern a -> Bool
match v (Literal b)    = v == b
match v (Range (b,c)) = b <= v && v <= c
match v (ListPat xs)   = (any (v ==) xs)
match _ Wildcard       = True
match _ Empty          = False
```

where the function `any` takes a predicate and a list and returns `True` if any elements in the list pass the predicate. (Note that we only implement closed ranges in our framework. It is

trivial to implement open and mixed ranges as well, but adding them makes the example needlessly more complex without contributing toward the goals of our framework.)

Given a definition for patterns and an operation over those patterns, we now need to know what kinds of operations we can do on a punctuation. The `Punc` class defines valid operations over data types that support punctuations, as follows:

```
class Punc p where
  isValid :: p -> Bool
  isEOD :: p -> Bool
  getEOD :: p
  combine :: p -> p -> p

  setCombine :: [p] -> [p] -> [p]
  setCombine ps1 ps2 =
    [combine p1 p2 | p1 <- ps1, p2 <- ps2, isValid (combine p1 p2)]
  covered :: p -> p -> Bool

  isCovered :: p -> [p] -> Bool
  isCovered pCheck ps = any (pCheck 'covered') ps

  findCovered :: [p] -> [p] -> [p]
  findCovered ps1 ps2 = [p | p <- ps1, p 'isCovered' ps2]

  findNotCovered :: [p] -> [p] -> [p]
  findNotCovered ps1 ps2 = [p | p <- ps1, not (p 'isCovered' ps2)]
```

An invalid punctuation is one that will never match anything (for any possible input). Based on the definition of `match` for patterns, an invalid punctuation is a tuple that contains the `Empty` pattern value for at least one of its attributes. So `isValid` returns `True` if the given punctuation does not contain the `Empty` pattern. The `isEOD` function returns `True` if the punctuation indicates the end of data items in the stream — that is, if the given punctuation contains wildcard values for all attributes, and the `textttgetEOD` returns a punctuation containing all wildcards, indicating the end of data items. The `combine` function takes two punctuations, and returns a punctuation that is the logical

“and” of those two punctuations. (Note that for our set of patterns, it is always possible to compute combine.) The `setCombine` function takes two lists of punctuations, and returns all valid pairwise combinations of the punctuations from each list.

Now we want streams that can contain data items or punctuations — two different kinds of tuples. Therefore, we redefine streams to take two types. In Haskell, all items in a list must be of the same type. The data type `Either` defined in `Prelude.hs` (the standard library for Haskell) gives us a way to put objects of two different data types into a single data type. The `Either` data type defines two constructors, where `Left` takes objects of one type, and `Right` takes objects of another type. We will store data items using the `Left` constructor and store punctuations using the `Right` constructor. We use the functions `norm` and `punct` to convert values into data items and punctuations, respectively. Thus, a `Stream` object is defined as follows:

```
type Stream a b = [[Either a b]]
```

```
norm :: a -> Either a b
```

```
norm x = Left x
```

```
punct :: b -> Either a b
```

```
punct x = Right x
```

Finally, we need to define functions that can be applied to tuples in a stream. Given a data type `t` for data items and a data type `p` for punctuations, the `Tuple` class is:

```
class Punc p => Tuple t p where
```

```
  match :: t -> p -> Bool
```

```
  nomatch :: t -> p -> Bool
```

```
  nomatch t p = not (match t p)
```

```
  splitPunc :: [Either t p] -> ([t],[p]) -> ([t],[p])
```

```
  splitPunc [] (ts,ps) = (reverse ts,reverse ps)
```

```
  splitPunc (Left t : xs) (ts,ps) = splitPunc xs (t:ts,ps)
```

```
  splitPunc (Right p : xs) (ts,ps) = splitPunc xs (ts,p:ps)
```

```

setMatch :: t -> [p] -> Bool
setMatch t ps = any (t 'match') ps
setNomatch :: t -> [p] -> Bool
setNomatch t ps = not (setMatch t ps)

setMatchTs :: [t] -> [p] -> [t]
setMatchTs ts ps = [t | t <- ts, setMatch t ps]

setNomatchTs :: [t] -> [p] -> [t]
setNomatchTs ts ps = [t | t <- ts, setNomatch t ps]

setNomatchPs :: [t] -> [p] -> [p]
setNomatchPs ts ps = [p | p <- ps, all ('nomatch' p) ts]

```

The `match` function takes a data item and a punctuation, and returns `True` if the data item matches the punctuation. The `splitPunc` function takes an input slice from the stream which may contain both data items and punctuations, and outputs a pair of lists where the first list contains all data items from the slice and the second list contains all punctuations from the slice. The `setXXX` functions build from the `match` function and operate on sets of data items or punctuations.

Now that we know what streams and punctuations look like in our framework, we are ready to consider implementing punctuation-aware stream iterators.

5.3 Implementation of Punctuation-Aware Stream Iterators

To enhance stream iterators to support punctuated streams, we first define three new functions: `pass`, `prop`, and `keep`. We then redefine a unary stream iterator as a 5-tuple `(initial_state, step, pass, prop, keep)`, where:

- `initial_state` has the same meaning as before.
- `step` has the same meaning as before.

- **pass** takes new punctuations and the current state, and returns any additional data items that can be output based on the punctuation.
- **prop** takes new punctuations and the current state, and returns punctuations that can be output.
- **keep** takes new punctuations and the state, and returns a new state based on the punctuations.

Note that **final** was removed. The **final** function was called when the input stream ended. It passed on any new data items, cleared out its state, and propagated the end of stream marker to the next operator. We can see that **final** simply performs the same tasks as **pass**, **prop**, and **keep** at the end of stream.

The order these functions are executed is important. The functions **step** and **pass** should be executed first, ensuring all appropriate data items are output. The function **prop** should follow **step** and **pass** in case any emitted punctuations that match output data items follow those output data items. Finally, **keep** should be executed to ensure that data items and punctuations are output before they are removed from state.

Thus, the new punctuation-aware Basic data type is as follows:

```
data Basic state input inputp output outputp =
  B {step :: [input] -> state -> ([output],state),      --Step function
     pass :: [inputp] -> state -> [output],              --Pass function
     prop :: [inputp] -> state -> [outputp],            --Prop function
     keep :: [inputp] -> state -> state                 --Keep function
  }
```

where **state**, **input**, and **output** are data types defined as before, and **inputp** and **outputp** are data types for input and output punctuations, respectively.

The changes to **unary** and **binary** for punctuated streams are similar, so we focus on **unary**. The punctuation-aware version of **unary** is as follows:

```
unary :: (Tuple it ip,Tuple ot op) => s ->
  (Basic s it ip ot op) -> Stream it ip -> Stream ot op
```

```

unary st (basic @ (B step pass prop keep)) [] =
  [map Left tsExtra ++ map Right psOut] ++ (unary stNew' basic [])
  where tsExtra      = pass [getEOD] st
        psOut       = prop [getEOD] st
        stNew'      = keep [getEOD] st
unary st (basic @ (B step pass prop keep)) (xs:rest) =
  [map Left tsOut ++ map Left tsExtra ++ map Right psOut] ++
  (unary stNew' basic rest)
  where (ts,ps)      = splitPunc xs ([],[])
        (tsOut,stNew) = step ts st
        tsExtra      = pass ps stNew
        psOut       = prop ps stNew
        stNew'      = keep ps stNew

```

That is, `unary` requires that the input and output streams of data items and punctuations adhere to the `Tuple` class. If the input stream has ended, then we call the three punctuation functions `pass`, `prop`, and `keep` in that order. All three punctuation functions are passed a punctuation that contains all wildcard patterns (using `getEOD`). If there are still items in the input stream, then they must be processed. New input may contain data items and punctuation, so they are first separated. Then `step` is called with new data items and the current state, followed by calls to `pass`, `prop`, and `keep` with new punctuations and the state. The execution order of the punctuation functions is important: The `prop` function could output punctuation that matches data items output by `pass`, so `pass` must be executed before `prop`. The `keep` function may not retain state related to data items that would be returned by `pass`, or punctuations that would be returned by `prop`, so `keep` must follow `pass` and `prop`.

There are cases where stream iterators do not need to implement particular punctuation functions. For example, the duplicate elimination iterator is not a blocking operator. It does not need to implement the `pass` function. We define *trivial punctuation functions* for such cases:

```

passT ps st = [];
propT ps st = [];

```

```
keepT ps st = st
```

We redefine `dupelimS` to exploit punctuations as follows:

```
dupelimS :: (Eq a, Tuple a b) => Stream a b -> Stream a b
dupelimS = unary [] (B step passT prop keep)
  where step ts tsSeen = (tsOut, tsSeen ++ tsOut)
        where tsOut = [t | t <- (nub ts), t `notElem` tsSeen]
        prop ps _ = ps
        keep ps tsSeen = setNomatchTs tsSeen ps
```

To eliminate duplicates in the original version, we kept the set of all data items that had arrived from the input stream. However, punctuations tell us what data items will never again appear in the stream. If there are data items in state that we know have no more duplicate values in the stream, they can be removed from state. We can remove any data items from state that match any punctuations that have arrived, and that is what the `keep` function does. The `prop` function propagates any punctuation as it arrives.

Let us see how `dupelimS` works with the same example as before, but enhanced with punctuations. We will add a punctuation to `s` to state that there will be no more data items between 0 and 4. Our new value for `s` is `[1,5], [3], [P(0,4)], [5,6,7], ...]`. In the following illustration, we omit calls to `pass` for space consideration. The `dupelimS` iterator uses `passT`, which returns `[]`.

	function	input	state	output	newstate
	step	[1,5]	[]	[1,5]	[1,5]
	prop	[]	[1,5]	[]	
	keep	[]	[1,5]		[1,5]
	step	[3]	[1,5]	[3]	[1,5,3]
	prop	[]	[1,5,3]	[]	
	keep	[]	[1,5,3]		[1,5,3]
	step	[]	[1,5,3]	[]	[1,5,3]

```

prop    [P(0,4)] [1,5,3]  [P(0,4)]
keep    [P(0,4)] [1,5,3]          [5]

step    [5,6,7] [5]      [6,7]  [5,6,7]
prop    []      [5,6,7]  []
keep    []      [5,6,7]          [5,6,7]
...

```

For the first two slices of input, execution is pretty much as it was for the non-punctuated case. The output is determined immediately, and the state grows with each input. Since no punctuations arrive during the first two slices, the `prop` and `keep` functions do not receive any punctuations for input, and therefore do not affect execution. When a punctuation arrives, however, we see the new behavior. The `prop` function receives the new punctuation and returns it, per its definition. The `keep` function receives the punctuation, and keeps in the new state only data items that do not match the punctuation. In this example, we keep all values greater or equal to 4 (1 and 3 are removed from state).

For completeness, we show here the definition of `binary`:

```

binary :: (Tuple itL ipR, Tuple itL ipR, Tuple ot op) =>
    s -> (Basic s itL ipL ot op) -> (Basic s itR ipR ot op) ->
    (Stream itL ipL, Stream itR ipR) -> Stream ot op
binary st basicL @ (B stepL passL propL keepL)
    basicR @ (B stepR passR propR keepR) ([], xsR) =
    (binary st basicR basicL (xsR, []))
binary st basicL @ (B stepL passL propL keepL)
    basicR @ (B stepR passR propR keepR) ((xsL:restL), xsR) =
    [map Left tsOut ++ map Left tsExtra ++ map Right psOut] ++
    (binary stNew' basicR basicL (xsR, restL))
where (ts,ps)      = splitPunc xs ([],[])
    (tsOut,stNew) = step1 ts st
    tsExtra      = pass1 ps stNew
    psOut        = prop1 ps stNew
    stNew'       = keep1 ps stNew

```

Now consider the enhanced version of `diffS` with punctuation functions:

```
--difference operator
diffS :: (Eq a,Eq b,Tuple a b) => (Stream a b, Stream a b) -> Stream a b
diffS = dupelimS . _diffS
_diffS :: (Eq a,Eq b,Tuple a b) => (Stream a b, Stream a b) -> Stream a b
_diffS = binary ([], [], [], []) (B lstep passT propT lkeep)
      (B rstep rpass rprop rkeep)
  where lstep ts (lts,rts,lps,rps) = ([],(ts ++ lts,rts,lps,rps))
        rstep ts (lts,rts,lps,rps) = ([], (lts,ts ++ rts,lps,rps))
        rpass ps (lts,rts,lps,rps) = setMatchTs (lts \\ rts) (ps ++ rps)
        rprop ps (lts,rts,lps,rps) = setCombine lps (ps ++ rps)
        lkeep ps (lts,rts,lps,rps) = (lts,rts,ps ++ lps,rps)
        rkeep ps (lts,rts,lps,rps) = (ltsNew,rtsNew,lps,rps)
      where ltsNew = nub(setNomatchTs (lts \\ rts) (ps ++ rps))
            rtsNew = nub(setNomatchTs rts lps)
```

Our data structure for state has changed from the non-punctuated case. We no longer need a Boolean value to tell us when we have reached the end of the negative input because punctuations tell us this. However, we do need to maintain two additional lists that hold the punctuations from each input. We can see that the `lstep` and `rstep` functions do not produce output (since we no longer track the end of the negative input). However, `rpass` outputs data items before the end of either stream. Data items can be emitted from the positive input that match punctuation from the negative input and have not appeared so far in the negative input. Thus, `diffS` is no longer necessarily blocking. The `rprop` function outputs certain punctuation. We cannot simply output punctuations as they arrive, as for `dupelimS`. Instead, we output punctuations that are valid combinations of punctuations received from each input. Finally, `rkeep` reduces the amount of state required when punctuations arrive. We only keep data items that do not match punctuations that have arrived from the other input, decreasing the amount of data items required in state. Also, we only keep punctuations that are not covered by punctuations in `psOut`, decreasing the amount of punctuations in state.

These are just two examples of the stream iterators defined in our framework. The complete listing of our stream iterators can be found in Appendix A.

5.4 Evaluation of Our Framework

We tested our framework using the warehouse example discussed in Chapter 2. Recall that there are many sensors that report temperatures periodically. The query we used reported the maximum temperature reported from any sensor each hour. The SQL for this query is:

```
SELECT hour, MAX(temp) AS MaxTemp
FROM (SELECT hour, temp FROM sensor1
      UNION
      SELECT hour, temp FROM sensor2
      UNION
      ...
      UNION
      SELECT hour, temp FROM sensorN)
GROUP BY hour;
```

We first implemented code to output streams of temperature reports from simulated sensors. Each sensor report contains the unique name of the sensor, the temperature being reported, and the hour and minute of the report. The code for our sensors is as follows:

```
type S_Name = String
type S_Temp = Int
type S_Hour = Int
type S_Minute = Int
type S_Seed = Int

-- Data schema: (sensorName, temperature, hour, minute)
type S_Report = (S_Name, S_Temp, S_Hour, S_Minute)
type S_ReportPunc =
    (Pattern S_Name, Pattern S_Temp, Pattern S_Hour, Pattern S_Minute)
```

```

sensor :: S_Name -> Int -> Stream S_Report S_ReportPunc
sensor a timeIncr = punctSensor (sensorValues a timeIncr)
  where punctSensor ((n1,t1,h1,m1) : x@(n2,t2,h2,m2) : xs) =
    if (h1 == h2) then [norm (n1,t1,h1,m1)] : punctSensor (x:xs)
    else [norm (n1,t1,h1,m1), punct (wc,wc,lit h1,wc)] :
      punctSensor (x:xs)

sensorValues :: S_Name -> Int -> [S_Report]
sensorValues a timeIncr = (a,70,1,0) :
  map newValues (sensorValues a timeIncr)
  where newValues :: S_Report -> S_Report
        newValues (n,t,h,m) = (n,newT,newH,newM)
          where seed = 27063 + (ord (n !! 0) * t) + ((h*60) + m)
                newT = t + (randoms seed 1 (-1)) !! m
                newM = (m + timeIncr) `mod` 60
                newH = if (m+timeIncr >= 60) then h + ((m+timeIncr) `div` 60)
                       else h

--Returns a list of random integers in [min,max].
randoms :: S_Seed -> Int -> Int -> [Int]
randoms n max min = map (incr max min) (iterate random n)
  where incr max min a = (a `mod` (max - min + 1)) + min

```

The `sensor` function outputs a stream of temperature readings and punctuations given a sensor name and a timer increment value (in minutes). We use the `sensorValues` function to generate a list of sensor reports. We start `sensorValues` with values for its name (from the input parameter `a`), the current temperature (70), the current hour (1), and the current minute (0). The `sensor` function reads the list of temperature reports and adds punctuations at the end of each hour.

The `sensorValues` function uses the infix operator `!!` that takes a list and an integer, and returns the item from the list specified by the integer (using zero-based indexing). For example, `[1,3,5] !! 2` will return 5. Additionally, the functions `'mod'` performs the modulus function, and `'div'` performs integer division. Finally, we have defined

the function `randoms` that constructs a list of random integers given a seed value within the given range. The random number generator function `random` used is based on the algorithm of Park and Miller [PM88].

Thus, the output of `take 25 (sensor "A" 5)` is (where the function `take` returns the first `n` items of the given list):

```
[[Left ("A",70,1,0)], [Left ("A",71,1,5)], [Left ("A",71,1,10)],
 [Left ("A",70,1,15)], [Left ("A",70,1,20)], [Left ("A",71,1,25)],
 [Left ("A",72,1,30)], [Left ("A",73,1,35)], [Left ("A",72,1,40)],
 [Left ("A",73,1,45)], [Left ("A",74,1,50)],
 [Left ("A",73,1,55), Right (*,*,{1},*)], [Left ("A",72,2,0)],
 [Left ("A",71,2,5)], [Left ("A",72,2,10)], [Left ("A",71,2,15)],
 [Left ("A",70,2,20)], [Left ("A",70,2,25)], [Left ("A",70,2,30)],
 [Left ("A",71,2,35)], [Left ("A",72,2,40)], [Left ("A",72,2,45)],
 [Left ("A",73,2,50)], [Left ("A",72,2,55), Right (*,*,{2},*)],
 [Left ("A",71,3,0)]]
```

We first implement the subquery using the `unionSensors` function, which takes as its parameters the number of sensors to union and the time increment for the sensors. It uses the stream iterator `unionS` to merge reports from the list of sensors into a single output stream. The function `simSensors` outputs a list of stream inputs, one for each sensor. The name for each i th sensor is simply the i th letter of the alphabet. That is, the first sensor has the name “A”, then second “B”, and so on. The union of sensor streams is therefore as follows:

```
unionSensors :: Int -> Int -> Stream S_Report S_ReportPunc
unionSensors n incr = unionS (simSensors 1 n incr)
simSensors _ 0 _ = []
simSensors i 1 incr = [sensor [(chr (i+64))] incr]
simSensors i n incr = (sensor [(chr (i+64))] incr) :
                      simSensors (i+1) (n-1) incr
```

Thus, a prefix of output of `unionSensors` for five input sensors is:

```
[[Left ("A",70,1,0)], [Left ("B",70,1,0)], [Left ("C",70,1,0)],
```

```

[Left ("D",70,1,0)], [Left ("E",70,1,0)], [Left ("A",71,1,5)],
[Left ("B",69,1,5)], [Left ("C",70,1,5)], [Left ("D",71,1,5)],
[Left ("E",69,1,5)], [Left ("A",71,1,10)], [Left ("B",68,1,10)],
[Left ("C",69,1,10)], [Left ("D",72,1,10)], [Left ("E",69,1,10)],
[Left ("A",70,1,15)], [Left ("B",68,1,15)], [Left ("C",69,1,15)],
[Left ("D",72,1,15)], [Left ("E",69,1,15)], [Left ("A",70,1,20)],
[Left ("B",68,1,20)], [Left ("C",68,1,20)], [Left ("D",71,1,20)],
[Left ("E",69,1,20)], [Left ("A",71,1,25)], [Left ("B",68,1,25)],
[Left ("C",68,1,25)], [Left ("D",70,1,25)], [Left ("E",69,1,25)],
[Left ("A",72,1,30)], [Left ("B",68,1,30)], [Left ("C",69,1,30)],
[Left ("D",70,1,30)], [Left ("E",69,1,30)], [Left ("A",73,1,35)],
[Left ("B",68,1,35)], [Left ("C",68,1,35)], [Left ("D",71,1,35)],
[Left ("E",70,1,35)], [Left ("A",72,1,40)], [Left ("B",69,1,40)],
[Left ("C",68,1,40)], [Left ("D",70,1,40)], [Left ("E",69,1,40)],
[Left ("A",73,1,45)], [Left ("B",68,1,45)], [Left ("C",69,1,45)],
[Left ("D",69,1,45)], [Left ("E",70,1,45)], [Left ("A",74,1,50)],
[Left ("B",68,1,50)], [Left ("C",68,1,50)], [Left ("D",68,1,50)],
[Left ("E",69,1,50)], [Left ("A",73,1,55)], [Left ("B",69,1,55)],
[Left ("C",68,1,55)], [Left ("D",67,1,55)], [Left ("E",68,1,55)],
[Left ("A",72,2,0), Right (*,*,{1},*)], [Left ("B",68,2,0)],
[Left ("C",69,2,0)], [Left ("D",68,2,0)], [Left ("E",69,2,0)],
[Left ("A",71,2,5)], [Left ("B",67,2,5)], [Left ("C",68,2,5)],
[Left ("D",68,2,5)], [Left ("E",68,2,5)]
]

```

Our first query, called `qryMaxTemp`, outputs the maximum temperature from any output each hour. In addition to the subquery `unionSensors`, we use the stream iterator `aggregateS`, which internally uses the stream iterator `groupbyS`. The stream iterator `groupbyS` takes two functions as input parameters: The first function takes a data item and returns the values of attributes from that data item to group on. The second function takes punctuations and returns `True` if the punctuation describes the grouping attributes.

The stream iterator `aggregateS` takes three functions: The first and second functions are as for the first two functions for `groupbyS`. Again, we want a function that will


```

[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],[],[Left (3,77),Right ({3},*)],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],[],[Left (4,77),Right ({4},*)],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],[],[Left (5,80),Right ({5},*)],[],
]

```

In this evaluation of our framework, we want to verify that three actions occur: First, that queries are not blocked. Second, that the state maintained by operators in the query is occasionally reduced. Third, that punctuations are passed through the query operators. We see from the output that the query is not blocked — output is produced before the end of the input arrives. Additionally, punctuation follows each result data item indicating that all reports for a particular hour have been output.

A bit more work is required to verify the amount of state used during execution. In our framework, we do not output information about the current state of our operators. Thus, we instrumented the code in `groupbyS` to track the size of state as data items arrive using the Hood debugger [HOO]. Without punctuations, the amount of state required by `groupbyS` grows without bound. However, with punctuation, the amount of state is reduced as punctuations arrive. For the query with five sensor inputs, the number of data items in state for `groupbyS` grows as data items arrive. When a punctuation arrives the state drops back to 0 data items. The size of state for `groupbyS` during execution is shown in Figure 5.1.

Finally, we see that punctuations are output, and that the state for `groupbyS` is reduced periodically during execution. From this behavior we can infer that punctuations are propagated through each stream iterator in the query.

We also implemented a query in the warehouse scenario to determine the count of distinct sensors reporting temperatures greater than 75 degrees from distinct sensors each hour, as follows:

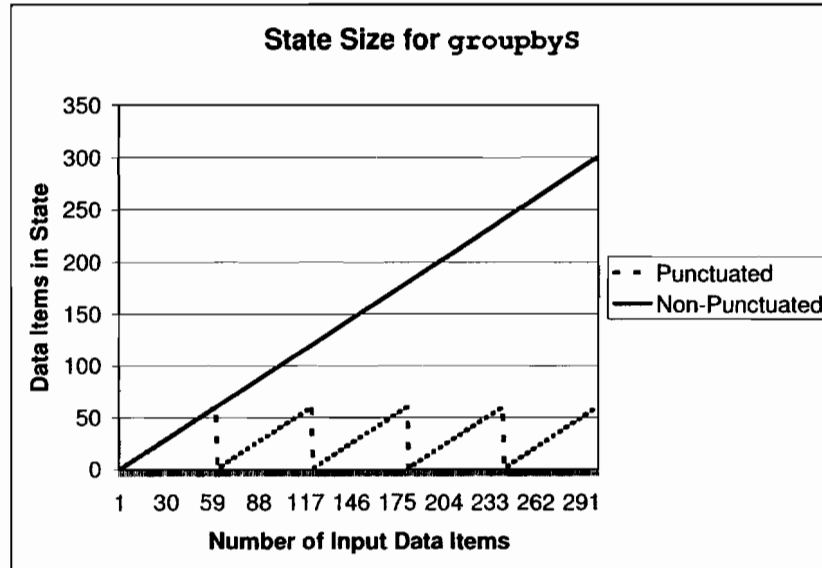


Figure 5.1: Number of data items held in state for `groupByS` during query execution, with and without punctuations.

```

qryHighDistinct :: Int -> Stream (S_Hour,Int) (Pattern S_Hour,Pattern Int)
qryHighDistinct n = aggregateS attr attrP countS (dupelimS highReports)
  where highReports = projectS prj prj pfilter (selectS pred (unionSensors n 5))
        attr (a,hr) = hr
        attrP (Wildcard,hr) = Just hr
        attrP _ = Nothing
        prj (a,tmp,hr,min) = (a,hr)
        pfilter (a,tmp,hr,min) = tmp == Wildcard && m == Wildcard
        pred (a,tmp,hr,min) = tmp > 75

```

For this query, the `aggregateS` stream iterator uses the aggregate function `countS` to count the number of data items each hour. In addition, we use three other stream iterators: `selectS`, `projectS`, and `dupelimS`. We have already discussed `dupelimS`. The `selectS` stream iterator outputs data items that pass the given predicate. For this query,

in Java (see Chapter 8) and queries defined in XML.

5.5 Summary

We have shown a framework using the functional programming language Haskell for a system that processes punctuated streams. We use this framework to model implementations of the three punctuation behaviors discussed in Chapter 4. Individual implementations of stream iterators implemented four functions to describe their behavior: The `step` function describes how to handle input without punctuation. The `pass` function describes what data items can be output due to incoming punctuation. The `prop` function describes what punctuations can be output due to incoming punctuation. The `keep` function describes what data items must remain in state due to punctuations. Our framework has proven general enough to support the operators we have encountered.

Our evaluation of this framework involved using the warehouse scenario described in Section 1.1.1. We implemented two queries in that scenario using our framework, and showed that three goals of exploiting punctuations were met: First, data items are output even when operators that are traditionally blocking were involved in the query. The `group-by` operator was used in both queries. By exploiting punctuations in our stream iterator implementation of `group-by`, results were output. Our second goal is that state is not constantly increasing. We saw using the Hood debugger for Haskell that state maintained in the stream iterator for `group-by` was decreased when punctuations arrived. Our final goal is that punctuations are propagated through each stream iterator in the query. Since punctuations are contained in the results, and state maintained by the stream iterator for `group-by` periodically decreased, we can infer that punctuations were propagated through the query.

In the next chapter we will address the question of whether or not our stream iterators are “reasonable” counterparts to their relational operator counterparts. Another benefit we will get from this framework is to prove that our implementations are correct according to formal definitions of punctuation behaviors. Since we used a language in our framework with formal semantics, we can use formal techniques to structure our proofs of correctness.

Chapter 6

Theory of Punctuated Streams

There is a significant issue we have not yet addressed, namely: When is a stream iterator a “reasonable” counterpart of its corresponding table operator? We hope the examples presented thus far seem sensible, but we have not yet shown that our implementations behave reasonably. In this chapter, we formulate our notions of correctness for stream iterators relative to their finite table counterparts. As noted in Chapter 4, we use three kinds of invariants to specify how a stream iterator should process punctuations: *pass invariants*, *propagation invariants*, and *keep invariants*. These are based on the punctuation rules discussed in Chapter 3. First, we will define what it means for a stream iterator to behave correctly in relation to a table operator. Then we define these invariants in terms of the allowable action on any given prefix of the input streams.

Punctuation invariants are cumulative. For some stream input S , they consider the prefix $S[i]$ (recall that $S[i]$ means the first i slices of S). In contrast, stream iterators are incremental. They consider individual slices $S@i$ (recall that $S@i$ returns the i -th slice of S). Invariant definitions are cumulative to line up with set-based definitions of table operators. Stream iterators are incremental to line up with their pipelined implementation. To be clear, we prefix invariant names with a ‘c’ to reflect their cumulative behavior.

We want to prove that a specific implementation of a stream iterator behaves in a manner consistent with the definition of its corresponding table operator (we later call this property “correctness”). Our proof strategy has two parts: First, we prove that any iterator implementation that adheres to the cumulative invariants is consistent with its corresponding table operator. Then, we prove that our implementations adhere to the invariants. The benefit of this approach comes up when an iterator has more than one

implementation, such as join. We only have to prove the invariants for join are consistent with the table definition one time, and then that each implementation adheres to the invariants. We give two such proofs in this chapter. Proofs for other stream iterators appear in Appendix A.

6.1 Faithfulness and Propriety

We base our notions of correctness of a stream iterator (which we term “faithfulness” and “propriety”) upon its series of partial outputs after processing each slice of the input(s). We want the output of our iterator at any point to be consistent with any possible further input.

We first define faithfulness for streams without punctuation. Let f be a unary iterator over non-punctuated data streams from $Stream(T)$ to $Stream(U)$, and let g be a table operator from $List(T)$ to $List(U)$. We say f is *faithful* to g if the following two conditions are met:

Safety For every S in $Stream(T)$, for all i , and for every bounded addition A in $List(T)$, $f(S)[i] \subseteq g(S[i] ++ A)$ ¹. That is, we never emit output unless we can be sure it will not conflict with any possible later input.

Completeness For every S in $Stream(T)$, for all i , and for all M , if $M \subseteq g(S[i] ++ A)$ for all bounded additions A in $List(T)$, then $M \subseteq f(S)[i]$. That is, we always emit an output if it will necessarily be generated by the table operator under any additional input, including no input.

The corresponding conditions for binary operators are similar: The safety condition is: $f(S_1, S_2)[2i] \subseteq g(S_1[i] ++ A_1, S_2[i] ++ A_2)$ for all A_1 and A_2 . The completeness condition is: $M \subseteq f(S_1, S_2)[2i]$ if $M \subseteq g(S_1[i] ++ A_1, S_2[i] ++ A_2)$. (Note the $2i$ index is because f will emit a slice of output for the i th slices of both S_1 and S_2 .)

¹Our meaning for “contained-in” ($\subseteq$) is determined by what g returns: a set, a bag or a list. When we use $\subseteq$ between a list and a set, we intend for the list to be viewed as a set and “contains” to mean subset. For a list and a bag, we intend for the list to be viewed as a bag, and $\subseteq$ to be interpreted as bag containment (per Albert [Alb91]). For a list and another list, $\subseteq$ is interpreted as prefix.

Theorem 6.1 *Every monotone table operator has a faithful stream counterpart.*

Proof: We prove the theorem for unary and binary iterators in turn. Let g be a monotone table operator, and f be a stream iterator.

case unary iterators: Let S be a stream. Define f as $f(S)@i = g(S[i]) - g(S[i-1])$ for $i > 0$. That is, the stream iterator f maintains all data items that have ever been output in state. We construct this proof using induction on i :

Base case ($i = 1$):

$$\begin{aligned} f(S)[1] &= f(S)@1, \text{ by the definitions of } [] \text{ and } @. \\ &= g(S[1]) - g(S[0]), \text{ by the definition of } f. \\ &= g(S[1]) - [], \text{ because } S[0] = []. \\ &= g(S[1]) \end{aligned}$$

Induction step:

For $i > 0$, assume $f(S)[i-1] = g(S[i-1])$. Prove $f(S)[i] = g(S[i])$.

$$\begin{aligned} f(S)[i] &= f(S)[i-1] \oplus f(S)@i, \text{ by the definitions of } [] \text{ and } @. \\ &= g(S[i-1]) \oplus f(S)@i, \text{ by the induction hypothesis} \\ &= g(S[i-1]) \oplus (g(S[i]) - g(S[i-1])), \text{ by the definition of } f. \\ &= g(S[i]), \text{ because } g \text{ is monotone} \end{aligned}$$

Thus, for monotone unary iterators, because $f(S)[n] = g(S[n])$, both conditions for faithfulness are satisfied.

case binary iterators: Let S_1 and S_2 be streams. Define f as $f(S_1, S_2)@2i \oplus f(S_1, S_2)@(2i-1) = g(S_1[i], S_2[i]) - g(S_1[i-1], S_2[i-1])$ for $i > 0$. Again, the stream iterator f maintains all data items that have ever been output in state. We will again construct this proof using induction:

Base case ($i=1$):

$$\begin{aligned} f(S_1, S_2)[2i] &= f(S_1, S_2)@1 \oplus f(S_1, S_2)@0 \\ &= g(S_1[1], S_2[1]) - g(S_1[0], S_2[0]) \end{aligned}$$

$$\begin{aligned}
&= g(S_1[1], S_2[1]) - [] \\
&= g(S_1[1], S_2[1])
\end{aligned}$$

Induction step:

For $i > 0$, assume $f(S_1, S_2)[2(i-1)] = g(S_1[i-1], S_2[i-1])$.

Prove $f(S_1, S_2)[2i] = g(S_1[i], S_2[i])$.

$$\begin{aligned}
f(S_1, S_2)[2i] &= f(S_1, S_2)@1 \oplus f(S_1, S_2)@2 \oplus \dots \oplus f(S_1, S_2)@(2i-1) \oplus f(S_1, S_2)@2i \\
&= f(S_1, S_2)[2(i-1)] \oplus f(S_1, S_2)@(2i-1) \oplus f(S_1, S_2)@2i \\
&= g(S_1[i-1], S_2[i-1]) \oplus f(S_1, S_2)@(2i-1) \oplus f(S_1, S_2)@2i \\
&= g(S_1[i-1], S_2[i-1]) \oplus (g(S_1[i], S_2[i]) - g(S_1[i-1], S_2[i-1])) \\
&= g(S_1[i], S_2[i]), \text{ because } g \text{ is monotone.}
\end{aligned}$$

Thus, for monotone binary iterators, because $f(S_1, S_2)[2n] = g(S_1[n], S_2[n])$, both conditions for faithfulness are satisfied.

End of proof.

For punctuated data streams, we modify the conditions of faithfulness such that any possible additional input is constrained to obey any punctuation seen already. Let f be a unary iterator over punctuated data streams from $Stream(T)$ to $Stream(U)$, and let g be a table operator from $List(T)$ to $List(U)$. We say f is *faithful* to g if the following two conditions are met:

Safety For every S in $Stream(T)$, for all i , and for every bounded addition A in $List(T)$, if $setMatchTs(A, puncts(S[i])) = []$, then $data(f(S)[i]) \subseteq g(data(S[i]) ++ A)$. Here the function *puncts* returns all punctuations in its input and the function *data* returns all data items in its input. The condition states that we never emit output unless we are sure it will not conflict with any later input A , where no data items in A match punctuations in $S[i]$.

Completeness For every S in $Stream(T)$, for all prefixes of S of length i , and for all M , if for all possible bounded additions A in $List(T)$ such that $setMatchTs(A, puncts(S[i])) = []$ and $M \subseteq g(data(S[i]) ++ A)$, then $M \subseteq data(f(S)[i])$. That is, we always emit

an output if it will necessarily be generated by the table operator under any additional input A (including no input), where no data items in A match punctuations in $S[i]$.

The definition for binary operators is modified in a similar way.

The other condition we want to enforce on our stream operators is that they are well-behaved with respect to punctuation. We say that a stream iterator f is *proper* if $f(S)$ is guaranteed to be grammatical whenever S is grammatical. Note that propriety is not a very strong condition. An iterator that emits no punctuation will always be proper. A stronger notion regarding punctuation would be to require an iterator to emit the *maximal* amount of punctuation that can be inferred from the input punctuation, but the original definition will serve for our purposes as the invariants we list later will emit punctuations as appropriate.

6.2 Pass Invariants

A *pass invariant* specifies the allowed output for a stream iterator after seeing a prefix of the input. A pass invariant has the form: $cpass(T_1, P_1, \dots, T_n, P_n) = T_O$, where T_O represents data items that can be output, given data items T_i and punctuations P_i that have arrived from the i th input. Table 6.1 lists some non-trivial pass invariants.

Op	Pass Invariant
group-by ($\mathcal{G}_A^F$)	$cpass(T_1, P_1) = \{t :: \langle f_i(U_t) \rangle \mid$ $t \in setMatchTs(\pi_A(T_1), groupP(A, P_1)) \wedge$ $f_i \in F \wedge U_t = \{u \mid u \in T_1 \wedge t[A] = u[A]\}\}$
sort ($\mathcal{S}_A$)	$cpass(T_1, P_1) = setMatchTs(T_1, init(Sort_A, P_1))$
difference ($-$)	$cpass(T_1, P_1, T_2, P_2) = \{t \mid t \in T_1 \wedge t \notin T_2 \wedge setMatch(t, P_2)\}$

Table 6.1: Pass invariants for traditional query operators. Here T_i represents data items and P_i represents punctuations from the i th input.

Two of the pass invariants require further explanation. The group-by iterator can output results for a group when all data items have been received for that group. The function *groupP* outputs punctuations in the output schema that match an entire group

when we have received enough input punctuations to know that all data items for that group have arrived. The set U_t contains all data items that have arrived with the same values (namely $t[A]$) for the grouping attributes. Thus we can compute and output the results for the set U_t . Using the warehouse example, suppose we have a query that reports the maximum temperature grouped by the hour attribute. If we receive the punctuation $\langle *, 4, [0, 30], * \rangle$ (the schema is $\langle \text{sensorid}, \text{hour}, \text{minute}, \text{temperature} \rangle$), then we cannot output results for any group. If we later receive the punctuation $\langle *, 4, [30, 59], * \rangle$, then we can output results for the hour 4 since no more data items will arrive that contribute to the hour 4.

In order for the sort iterator to output correct results early, we need to know when the values that appear first in sort order have arrived. Punctuations that match a prefix of the final sorted output give us this information. The *init* function returns punctuations based on the sort order and the input punctuations such that no data item can still arrive that would be sorted before data items that match those punctuations. Again using the warehouse example, suppose we have a query that sorts its input on the hour attribute in ascending order. If we receive punctuation that reads $\langle *, [2, 4], *, * \rangle$, then we cannot output anything. If, however, we also receive punctuation that reads $\langle *, [0, 3], *, * \rangle$, we can output results through the fourth hour. We know by the attribute's type that 0 is the first value for the sorted output. The *init* function, given the two punctuations shown above, will return $\langle *, [0, 4], *, * \rangle$.

6.3 Propagation Invariants

Propagation invariants specify what punctuations can be emitted by a stream iterator. Each iterator has a propagation invariant, of the form: $cprop(T_1, P_1, \dots, T_n, P_n) = P_O$, where P_O represents punctuations that can be output given T_i and P_i . Propagation invariants assume the output for the corresponding pass invariant has already been emitted. Table 6.2 lists propagation invariants.

Note that the *cprop* definitions we use here do not depend at all on the input data items (T_i) that have arrived. One might wonder, “Why include the input data items at

Op	Propagation Invariant
select (σ_q)	$cprop(T_1, P_1) = P_1$
project (π_A)	$cprop(T_1, P_1) = \{p p \in groupP(A, P_1)\}$
dupelim (δ)	$cprop(T_1, P_1) = P_1$
group-by ($\mathcal{G}_A^{agg}$)	$cprop(T_1, P_1) = \{p :: \langle w_i \rangle \mid p \in groupP(A, P_1) \wedge \forall f_i \in F, w_i = '*' \}$
sort ($\mathcal{S}_A$)	$cprop(T_1, P_1) = init(Sort_A, P_1)$
join ($\bowtie_c$)	$cprop(T_1, P_1, T_2, P_2) = P_1 \bowtie P_2$
union ($\bigcup$)	$cprop(T_1, P_1, T_2, P_2) = setCombine(P_1, P_2)$
intersect ($\bigcap$)	$cprop(T_1, P_1, T_2, P_2) = setCombine(P_1, P_2)$
difference ($-$)	$cprop(T_1, P_1, T_2, P_2) = setCombine(P_1, P_2)$

Table 6.2: Propagation invariant for traditional query operators. Here T_i represents data items and P_i represents punctuations from the i th input.

all in the definition?” Alternative definitions of *cprop* may indeed take advantage of data that have arrived. For example, consider join. If a punctuation arrives on one of the inputs that does not match any data items that have arrived from that input, then it can immediately be output with wildcard values for all attributes in the other input. Thus, we include input data items in our definition to accommodate these alternative definitions.

The propagation invariants for project, group-by, and join require explanation. Output punctuations must have the same structure as output data items. If the stream iterator is projecting out attributes, then the same attributes must be projected out of punctuations. However, we cannot simply modify all punctuations. We must make sure that all data items have arrived that match the projected attributes (indicated as A in Table 6.2). We use the function *groupP* to return only those punctuations that indicate all data items for particular values of the projected attributes have arrived. The invariant for group-by uses *groupP* in the same manner, but also uses wildcard values for the aggregate function attributes.

The propagation function for join simply joins punctuations based on the pattern values of the join attributes. For example, consider the input schemas $S_L(A, B)$ and $S_R(B, C, D)$. Suppose punctuation $lp = \langle 5, [8, 10] \rangle$ arrives on S_L and the punctuations $rp_1 = \langle [5, 7], 100, * \rangle$ and $rp_2 = \langle [7, 10], *, 50 \rangle$. Since the pattern $[8, 10]$ and $[5, 7]$ do not overlap, there can be no output from joining lp and rp_1 . However, the patterns $[8, 10]$ and

[7, 10] do overlap, and so lp and rp_2 can be joined, and the result of joining them would be $\langle 5, [8, 10], *, 50 \rangle$.

Note that these are not the only propagation invariants possible. There is a tradeoff between complexity of the operator implementation and how much punctuation can be emitted at a point. For example, in the case of finite domains, duplicate elimination could emit punctuation based solely on data items it has seen, at the expense of tracking the space of all possible data items.

6.4 Keep Invariants

Keep invariants specify what state a stream iterator must preserve, which will generally be a subset of what is retained in the non-punctuated case. Unlike pass and propagation invariants, we usually specify keep invariants on each input of an iterator (which we denote with the subscript j). They have the form: $ckee_p_j(T_1, P_1, \dots, T_n, P_n) = \hat{T}_j$, where T_i and P_i defined as before, and $\hat{T}_j$ is an expression for the data items that must remain in state that has arrived from the j th input. Table 6.3 lists non-trivial keep invariants. Keep invariants assume all data items and punctuations specified by the corresponding pass and propagation invariants have already been output.

Op	Keep Invariant
dupelim (δ)	$ckee_p_1(T_1, P_1) = \text{setNomatchTs}(T_1, P_1)$
group-by ($\mathcal{G}_A^F$)	$ckee_p_1(T_1, P_1) = [t t \in T_1 \wedge \text{setNomatch}(\pi_A(t), \text{groupP}(A, P_1))]$
sort ($\mathcal{S}_A$)	$ckee_p_1(T_1, P_1) = \text{setNomatchTs}(T_1, \text{init}(\text{Sort}_A, P_1))$
join ₁ ($\bowtie_c$)	$ckee_p_1(T_1, P_1, T_2, P_2) = [t t \in T_1 \wedge \text{setNomatch}(\pi_J(t), \text{groupP}(J, P_2))]$
join ₂ ($\bowtie_c$)	$ckee_p_2(T_1, P_1, T_2, P_2) = [t t \in T_2 \wedge \text{setNomatch}(\pi_J(t), \text{groupP}(J, P_1))]$
intersect ₁ ($\cap$)	$ckee_p_1(T_1, P_1, T_2, P_2) = \text{setNomatchTs}(T_1, P_2)$
intersect ₂ ($\cap$)	$ckee_p_2(T_1, P_1, T_2, P_2) = \text{setNomatchTs}(T_2, P_1)$
difference ₁ ($-$)	$ckee_p_1(T_1, P_1, T_2, P_2) = [t t \in T_1 \wedge t \notin T_2 \wedge \text{setNomatch}(t, P_2)]$
difference ₂ ($-$)	$ckee_p_2(T_1, P_1, T_2, P_2) = \text{setNomatchTs}(T_2, P_1)$

Table 6.3: Keep invariants for traditional query operators. Here T_i represents data items and P_i represents punctuations from the i th input, and J is the set of join attributes.

Note the “cross-over” aspect of the definitions of *ckee_p* for the binary operators listed. The data items that remain in state for one input are based on punctuation that have

arrived from the other input.

6.5 The Minimality Condition

In addition to the specific behavior rules for each kind of operator, we assume an additional condition on iterator implementations. The *minimality condition* states that an iterator never produces more output data items than needed to satisfy its behavior rules. This condition is necessary to prevent an operator from gratuitously repeating an output data item after punctuation matching that data item has been emitted. We note that it is generally straightforward to prove that iterators defined in our framework observe the minimality condition. We assume henceforth that all stream iterators satisfy the minimality condition.

6.6 Proving Correctness of Stream Iterators

We include here proofs that our stream versions of duplicate elimination and difference are faithful and proper. Our general strategy for such proofs is to specify cumulative invariants for an operator and to show that any stream iterator that obeys these cumulative rules is faithful and proper, and then to show that our particular iterator implementation indeed obeys the cumulative invariants.

6.6.1 Stream Iterator for Duplicate Elimination

Theorem 6.2 *The stream iterator `dupelimS` is faithful and proper to the table operator `duplicate elimination`.*

Proof: Consider the cumulative invariants for `dupelimS`:

$$cpass(T, P) = \delta(T)$$

$$cprop(T, P) = P$$

$$ckeeper(T, P) = setNomatchTs(T, P)$$

We use δ to represent duplicate elimination over some finite input, per Albert [Alb91]. We will first show that any stream iterator that conforms to these invariants and the minimality condition is faithful and proper to the table operator `dupelim`. We then show that the

particular stream iterator implementation of `dupelimS` given in Section 5.3 conforms to the invariants. For this proof, it is useful to denote the data items and punctuations present in the first i slices of input. We use the notation $ts_i = \text{data}(S[i])$ and $ps_i = \text{puncts}(S[i])$, where S is the input stream. Also, for $j > i$, let $ts_{ij} = \text{data}(S[i \rightarrow j])$.

As shown in Theorem 6.1, any monotone table operator such as `dupelim` has a faithful stream counterpart. If $\text{ckeep}(ts, ps) = ts$, we would have the “standard” faithful version defined in the proof of that theorem. We need to show for faithfulness that retaining less state does not change the output. Consider what is output between two points i and j (assuming minimality):

$$\begin{aligned} \text{cpass}(ts_j, ps_j) - \text{cpass}(ts_i, ps_i) &= \delta(ts_j) - \delta(ts_i), \text{ by the definition of } \text{cpass}. \\ &= \delta(ts_j - ts_i), \text{ by the definition of } \delta. \\ &= \delta(ts_{ij} - ts_i), \text{ because } ts_j = ts_i ++ ts_{ij} \end{aligned}$$

This equivalent expression indicates how state is used in an iterator implementation: past input is kept and used to filter subsequent input. So the question is, what is output if our state is per `ckkeep`:

$$\begin{aligned} &\delta(ts_{ij} - \text{ckkeep}(ts_i, ps_i)) \\ &= \delta(ts_{ij} - \text{setNomatchTs}(ts_i, ps_i)) \\ &= \delta(ts_{ij} - (ts_i - \text{setMatchTs}(ts_i, ps_i))) \\ &= \delta((ts_{ij} - ts_i) \cup (ts_{ij} \cap ts_i \cap \text{setMatchTs}(ts_i, ps_i))) \\ &\quad [\text{since } A - (B - C) = (A - B) \cup (A \cap B \cap C)] \\ &= \delta((ts_{ij} - ts_i) \cup \emptyset) \\ &\quad [\text{since } ts_{ij} \text{ and } \text{setMatchTs}(ts_i, ps_i) \text{ can have no data items in common} \\ &\quad \text{when the input is grammatical}] \\ &= \delta(ts_{ij} - ts_i). \end{aligned}$$

This value is the same as the standard version, so reducing state per `ckkeep` does not affect faithfulness.

For propriety, we see that punctuations emitted by stage i are all the punctuations received, namely ps_i . Can a data item t such that $\text{setMatch}(t, ps_i)$ be output after stage i ? If so, it is emitted by some stage $j > i$. So it must be that $t \in \text{cpass}(ts_j, ps_j)$,

and hence $t \in ts_j$. However, $t \notin ts_{ij}$, by grammaticality of the input. Hence $t \in ts_i$ and $t \in cpass(ts_i, ps_i)$. So t must already be output at stage i , and, by the minimality condition, will not be output again later. Thus the output of any iterator satisfying *cpass* and *cprop* is grammatical given grammatical input, and is therefore proper.

Conformance: We show that *dupelimS* (as defined in Section 5.3) conforms to the invariants, and hence is faithful. First, we show that the output at each iteration i (call it *incrts_i*) is equivalent to the pass rule at i minus the pass rule for the previous iteration $i - 1$. That is, $incrts_i = cpass(ts_i, ps_i) - cpass(ts_{i-1}, ps_{i-1})$. Note the output data items at any iteration are data items returned by *step* and *pass*. The state value at iteration i in our proof is denoted as st_i . New data items and punctuations arriving at slice i are denoted as ts^i and ps^i , therefore $ts_{i-1} \cup ts^i = ts_i$ and $ps_{i-1} \cup ps^i = ps_i$.

Pass:

$$\begin{aligned}
 incrts_i &= step(ts^i, st_{i-1}) ++ pass(ps^i, st_{i-1}) \\
 &= \delta(ts^i - st_{i-1}) ++ [] \\
 &= \delta(ts^i - (ts_{i-1} - setMatchTs(ts_{i-1}, ps_{i-1}))) \\
 &= \delta((ts^i - ts_{i-1}) \cup (ts^i \cap ts_{i-1} \cap setMatchTs(ts_{i-1}, ps_{i-1}))) \\
 &\quad [\text{since } A - (B - C) = (A - B) \cup (A \cap B \cap C)] \\
 &= \delta((ts^i - ts_{i-1}) \cup \emptyset) \\
 &\quad [\text{since } ts^i \text{ and } setMatchTs(ts_{i-1}, ps_{i-1}) \text{ have no common data items}] \\
 &= \delta(ts^i - ts_{i-1}) \\
 &= cpass(ts_i, ps_i) - cpass(ts_{i-1}, ps_{i-1}) \\
 &\quad [\text{from the equality shown earlier}]
 \end{aligned}$$

Showing conformance to *cprop* is trivial since *dupelimS* outputs punctuations as they arrive, and so is omitted.

Keep:

To prove conformance to *ckeeep*, we show by induction that at any stage i , $keep(ps^i, st_{i-1}) = ckeep(ts_i, ps_i)$.

Base case: $i = 0$, so $ts_i = ps_i = []$, and the initial state $st_0 = []$. Then:

$$\begin{aligned}
 &keep(ps^1, (ts^1 ++ st_0)) \\
 &= keep(ps^1, (ts^1 ++ []))
 \end{aligned}$$

$$\begin{aligned}
&= \text{keep}(ps_1, ts_1) \\
&= ts_1 - \text{setMatchTs}(ts_1, ps_1) \\
&= \text{ckeep}(ts_1, ps_1)
\end{aligned}$$

Induction step: Assume $st_{i-1} = \text{ckeep}(ts_{i-1}, ps_{i-1})$. Show that $\text{keep}(ps^i, st_{i-1}) = \text{ckeep}(ts_i, ps_i)$.

$$\begin{aligned}
&\text{keep}(ps^i, (ts^i ++ st_{i-1})) \\
&= \text{setNomatchTs}(ts^i ++ st_{i-1}, ps^i) \\
&= \text{setNomatchTs}(ts^i ++ \text{ckeep}(ts_{i-1}, ps_{i-1}), ps^i) \\
&\quad [\text{by the induction hypothesis}] \\
&= \text{setNomatchTs}(ts^i ++ \text{setNomatchTs}(ts_{i-1}, ps_{i-1}), ps^i) \\
&= \text{setNomatchTs}(ts^i ++ [t | t \in ts_{i-1} \wedge \text{setNomatch}(t, ps_{i-1})], ps^i) \\
&= \text{setNomatchTs}([t | t \in ts^i ++ ts_{i-1} \wedge \text{setNomatch}(t, ps_{i-1})], ps^i) \\
&\quad [\text{since } \forall t \in ts^i, \text{setNomatch}(t, ps_{i-1}) \text{ by grammaticality of the input}] \\
&= \text{setNomatchTs}([t | t \in ts_i \wedge \text{setNomatch}(t, ps_{i-1})], ps^i) \\
&= [u | u \in [t | t \in ts_i \wedge \text{setNomatch}(t, ps_{i-1})] \wedge \text{setNomatch}(u, ps^i)] \\
&= [u | u \in ts_i \wedge \text{setNomatch}(u, ps_{i-1}) \wedge \text{setNomatch}(u, ps^i)] \\
&= [u | u \in ts_i \wedge \text{setNomatch}(u, ps_{i-1} ++ ps^i)] \\
&= [u | u \in ts_i \wedge \text{setNomatch}(u, ps_i)] \\
&= \text{setNomatchTs}(ts_i, ps_i) \\
&= \text{ckeep}(ts_i, ps_i)
\end{aligned}$$

End of proof.

6.6.2 Stream Iterator for Difference

We first show that any stream iterator that conforms to *cpass* is faithful to table difference. Second, we show that we can drop the data items indicated by the keep invariants without impairing future computation, as long as the data items in *cpass* have been output already (so dropping these data items will not compromise faithfulness). Third, we show that emitting any of the punctuation in *cprop* after the data items in *cpass* will be grammatical. Finally, we show that our implementation of *diffS* (defined in Section 5.3) satisfies the *cpass* invariant, and hence that it is faithful and proper.

Theorem 6.3 *The stream iterator diffS is faithful and proper to the table operator $\mathit{difference}$.*

Proof: We start with the following invariants for the behavior of a stream iterator for difference. Let L and R be any two input streams, and assume we are at slice i in both streams. Let $lts_i = \mathit{data}(L[i])$, $lps_i = \mathit{puncts}(L[i])$, $rts_i = \mathit{data}(R[i])$ and $rps_i = \mathit{puncts}(R[i])$. The invariants we use for difference are:

$$\begin{aligned} \mathit{cpass}(lts_i, lps_i, rts_i, rps_i) &= \{t \mid t \in lts_i \wedge t \notin rts_i \wedge \mathit{setMatch}(t, rps_i)\} \\ \mathit{cprop}(lts_i, lps_i, rts_i, rps_i) &= \mathit{setCombine}(lps_i, rps_i) \\ \mathit{ckeeperLT}(lts_i, lps_i, rts_i, rps_i) &= \{t \mid t \in lts_i \wedge t \notin rts_i \wedge \mathit{setNomatch}(t, rps_i)\} \\ \mathit{ckeeperRT}(lts_i, lps_i, rts_i, rps_i) &= \mathit{setNomatchTs}(rts_i, lps_i) \end{aligned}$$

The pass invariant says what data items should have been emitted to the output at this point, namely data items that have arrived from the left (positive) input, have not arrived from the right (negative) input, and will not appear in the right input in the future. The propagation rule says what punctuation can be emitted. In this case all valid combinations (conjunctions) of punctuations that have arrived from each input may be emitted. The two keep invariants say which data items must still remain from the left and right inputs at this point without impairing our ability to process later inputs correctly. From the left input we can remove from state any data item that has appeared in the right input (it will never be output), or for which no matching data item will ever appear in the right input (so it has already been output). From the right input we discard any data item where there will be no further matching data items on the left input (so it will not be needed to “cancel” any further inputs).

Faithfulness (safety): We need to show that

$$\mathit{cpass}(lts_i, lps_i, rts_i, rps_i) \subseteq (lts_i ++ ls) - (rts_i ++ rs)$$

for any lists ls and rs where $\mathit{setMatchTs}(ls, lps_i) = \emptyset$ and $\mathit{setMatchTs}(rs, rps_i) = \emptyset$. Suppose t is a data item in the left-hand side above. It must be that $t \in lts_i$, $t \notin rts_i$, and $\mathit{setMatch}(t, rps_i)$. Therefore $t \in lts_i ++ ls$. We have $t \notin rs$ since it matches some punctuation in rps_i . So $t \notin rts_i ++ rs$. Thus t is in $(lts_i ++ ls) - (rts_i ++ rs)$ and the containment is proved.

Faithfulness (completeness): Suppose $t \in (lts_i ++ ls) - (rts_i ++ rs)$ for every ls and rs where $setMatchTs(ls, lps_i) = setMatchTs(rs, rps_i) = \emptyset$. We must show that $t \in cpass(lts_i, lps_i, rts_i, rps_i)$. Since completeness is shown for all possible bounded additions to lts_i , we can choose ls to be $[]$, and thus we must have $t \in (lts_i ++ []) - (rts_i ++ rs)$, hence $t \in lts_i ++ []$ and thus $t \in lts_i$. We know $t \notin rts_i$, otherwise it would be in $rts_i ++ rs$. Now consider any data item s where $setMatch(s, rps_i) = false$. We must have $t \in (lts_i ++ []) - (rts_i ++ [s])$, hence $t \notin rts_i ++ [s]$, so $t \neq s$. Thus $setMatch(t, rps_i) = true$. Since $t \in lts_i$, $t \notin rts_i$ and $setMatch(t, rps_i)$, we have $t \in cpass(lts_i, lps_i, rts_i, rps_i)$, as required for completeness.

Keep: Consider some later point j , and define lts_j , lps_j , rts_j and rps_j analogously to lts_i , lps_i , rts_i and rps_i . Further, define $lts_{ij} = data(L[i \rightarrow j])$ and $rts_{ij} = data(R[i \rightarrow j])$. Let $\overline{lts_i} = ckeepLT(lts_i, lps_i, rts_i, rps_i)$ and $\overline{rts_i} = ckeepRT(lts_i, lps_i, rts_i, rps_i)$. We want to show that:

$$cpass(lts_j, lps_j, rts_j, rps_j) = cpass(lts_i, lps_i, rts_i, rps_i) \cup (cpass(lts_{ij} \cup \overline{lts_i}), lps_j, (rts_{ij} \cup \overline{rts_i}), rps_j).$$

That is, if we have emitted all of $cpass(lts_i, lps_i, rts_i, rps_i)$ at time i , then we can produce the correct additional output at time j from only those data items arriving after time i combined with the reduced state held in $\overline{lts_i}$ and $\overline{rts_i}$.

left $\subseteq$ right: If $t \in cpass(lts_j, lps_j, rts_j, rps_j)$, then $t \in lts_j$, $t \notin rts_j$ (hence $t \notin rts_i$ and $t \notin rts_{ij} \cup \overline{rts_i}$) and $setMatch(t, rps_j)$. There are two cases here, depending on whether t matches punctuations in rps_i . (Case 1) Assume $setMatch(t, rps_i)$. If $t \in lts_i$, then $t \in cpass(lts_i, lps_i, rts_i, rps_i)$. If $t \notin lts_i$, then $t \in lts_{ij}$, and $t \in lts_{ij} \cup \overline{lts_i}$. Since $t \notin \overline{rts_i}$ and $setMatch(t, rps_j)$, it must be that $t \in cpass((lts_{ij} \cup \overline{lts_i}), lps_j, (rts_{ij} \cup \overline{rts_i}), rps_j)$. (Case 2) Assume $setMatch(t, rps_i) = false$ (hence $setNomatch(t, rps_i)$). Thus, $t \notin cpass(lts_i, lps_i, rts_i, rps_i)$. Recall that $t \notin rts_{ij} \cup \overline{rts_i}$ and $setMatch(t, rps_j)$. If $t \in lts_i$, then $t \in \overline{lts_i}$. Otherwise, because $t \in lts_j$, $t \in lts_{ij}$. In either case, $t \in lts_{ij} \cup \overline{lts_i}$ and therefore $t \in cpass((lts_{ij} \cup \overline{lts_i}), rps_j, (rts_{ij} \cup \overline{rts_i}), rps_j)$.

right $\subseteq$ left: There are two cases, depending on which side of the union t is in. (Case 1) Assume $t \in cpass(lts_i, lps_i, rts_i, rps_i)$. So $t \in lts_i$ (hence $t \in lts_j$), $t \notin rts_i$ and $setMatch(t, rps_i)$. As t is not in rts_i and cannot appear after point i , $t \notin rts_j$. As

$setMatch(t, rps_j)$ follows from $setMatch(t, rps_i)$, we have $t \in cpass(lts_j, lps_j, rts_j, rps_j)$. (Case 2) Assume $t \in cpass((lts_{ij} \cup \overline{lts_i}), lps_j, (rts_{ij} \cup \overline{rts_i}), rps_j)$. Then $t \in lts_{ij} \cup \overline{lts_i}$ (hence $t \in lts_j$), $t \notin rts_{ij} \cup \overline{rts_i}$, and $setMatch(t, rps_j)$. (Case 2.1) If $t \in lts_i$, it must be that $t \in \overline{lts_i}$ (since $t \in (lts_{ij} \cup \overline{lts_i})$). Hence $t \notin rts_i$ by the definition of $\overline{lts_i}$. It follows that $t \notin \overline{rts_i}$. Since $t \notin rts_{ij} \cup \overline{rts_i}$, it must be $t \notin rts_j$. Thus we have all the conditions for $t \in cpass(lts_j, lps_j, rts_j, rps_j)$. (Case 2.2) If $t \notin lts_i$, but $t \in lts_j$, we know $setMatch(t, lps_i) = false$. Thus, because $t \notin \overline{rts_i}$, it must be that $t \notin rts_i$ by the definition of $\overline{rts_i}$, and therefore $t \notin rts_j$. That fact, together with $t \in lts_j$ and $setMatch(t, rps_j)$, says that $t \in cpass(lts_j, lps_j, rts_j, rps_j)$.

Thus, we have satisfied both conditions of faithfulness, and so any stream iterator that adheres to the pass and keep invariants defined for difference is faithful to the table version. Next, we show that, given grammatical inputs, a stream iterator that adheres to the propagation invariant defined above for difference is proper.

Propriety: Suppose we emit the punctuation in $cprop(lps_i, rps_i)$ at point i , and that all the data items in $cpass(lts_i, lps_i, rts_i, rps_i)$ have appeared. We want to make sure that any data item t emitted later does not match this punctuation. Consider a tuple $t \in cpass(lts_j, lps_j, rts_j, rps_j)$ where $setMatch(t, (cprop(lps_i, rps_i)))$. It must then be that $setMatch(t, lps_i)$ and $t \notin rts_i$ since t had to match some $lp \& rp$ where $lp \in lps_i$ and $rp \in rps_i$. Since $t \in lts_j$ but matches lps_i , t must already have appeared in lts_i . Since $t \notin rts_j$, also $t \notin rts_i$. Since $t \in lts_i$, $t \notin rts_i$ and $t \notin rts_j$, we see that t was already emitted as part of $cpass(lts_i, lps_i, rts_i, rps_i)$. (Even if the inputs have duplicates, t will not be emitted again, because $t \notin lts_j - lts_i$.)

Finally, we want to show that our implementation `diffS` of the stream iterator for difference conforms to the invariants, and thus is faithful and proper to its corresponding table operator.

Conformance: To show that `diffS` conforms to the appropriate punctuation invariants, we prove that the results output at each step are equivalent to the cumulative rule at that point in the input minus the cumulative rule for the previous point in the input. Recall our implementation of the stream iterator for difference (from Section 5.3):

--difference operator

```

diffS :: (Eq a, Eq b, Tuple a b) => (Stream a b, Stream a b) -> Stream a b
diffS = dupelimS . _diffS
_diffS :: (Eq a, Eq b, Tuple a b) => (Stream a b, Stream a b) -> Stream a b
_diffS = binary ([], [], [], []) (B lstep passT propT lkeep)
      (B rstep rpass rprop rkeep)
where lstep ts (lts, rts, lps, rps) = ([], (ts ++ lts, rts, lps, rps))
      rstep ts (lts, rts, lps, rps) = ([], (lts, ts ++ rts, lps, rps))
      rpass ps (lts, rts, lps, rps) = setMatchTs (lts \\ rts) (ps ++ rps)
      rprop ps (lts, rts, lps, rps) = setCombine lps (ps ++ rps)
      lkeep ps (lts, rts, lps, rps) = (lts, rts, ps ++ lps, rps)
      rkeep ps (lts, rts, lps, rps) = (ltsNew, rtsNew, lps, rps)
      where ltsNew = setNomatchTs (lts \\ rts) (ps ++ rps)
            rtsNew = setNomatchTs rts lps

```

We show that the output from each step is equivalent to the cumulative pass invariant for that step minus the cumulative pass invariant for the previous step. In the proofs that follow, we use the following notation in addition to ts_i and ts_{ij} defined earlier:

$$\begin{aligned}
cpass_i &= cpass(lts_i, lps_i, rts_i, rps_i) \\
cprop_i &= cprop(lts_i, lps_i, rts_i, rps_i) \\
ckkeepLT_i &= ckkeepLT(lts_i, lps_i, rts_i, rps_i) \\
ckkeepRT_i &= ckkeepRT(lts_i, lps_i, rts_i, rps_i) \\
ts^i &= data(T@i) \\
ps^i &= punct(T@i) \\
osl^i &= fst(lstep(lts^i, (ckkeepLT_i, ckkeepRT_i, lps_i, rps_i))) \\
opl^i &= passT(lps^i, (lts^i ++ ckkeepLT_i, ckkeepRT_i, lps_i, rps_i)) \\
osr^i &= fst(rstep(lts^i, (ckkeepLT_i, ckkeepRT_i, lps_i, rps_i))) \\
opr^i &= rpass(rps^i, (lts^i ++ ckkeepLT_i, rts^i ++ ckkeepRT_i, lps_i, rps_i))
\end{aligned}$$

The values osl^i , opl^i , osr^i , and opr^i represent the output of each component at the i th step. For now, we assume that state at any given state i is $(ckkeepLT_{i-1}, ckkeepRT_{i-1}, lps_i, rps_i)$. We will establish this assumption in the proof for the keep functions. Thus, we want to show that $osl^i ++ opl^i ++ osr^i ++ opr^i = cpass_i - cpass_{i-1}$.

Proof:

$$\begin{aligned}
& osl^i ++ opl^i ++ osr^i ++ opr^i \\
&= [] ++ [] ++ [] ++ opr^i \text{ [By definition of } lstep, lpass, \text{ and } rstep.] \\
&= opr^i \\
&= setMatchTs((lts^i ++ ckeepLT_{i-1}) \setminus (rts^i ++ ckeepRT_{i-1}), rps_i) \\
&= [t | t \in (lts^i ++ ckeepLT_{i-1}) \setminus (rts^i ++ ckeepRT_{i-1}) \wedge setMatch(t, rps_i)] \\
&= [t | t \in (lts^i ++ ckeepLT_{i-1}) \wedge t \notin (rts^i ++ ckeepRT_{i-1}) \wedge setMatch(t, rps_i)] \\
&= [t | (t \in lts^i \vee t \in ckeepLT_{i-1}) \wedge t \notin (rts^i ++ ckeepRT_{i-1}) \wedge setMatch(t, rps_i)] \\
&= [t | (t \in lts^i \vee t \in (lts_{i-1} - cpass_{i-1})) \wedge t \notin (rts^i ++ ckeepRT_{i-1}) \wedge setMatch(t, rps_i)] \\
&\quad \text{[By Lemma 6.1, below.]} \\
&= [t | (t \in lts^i \vee t \in (lts_{i-1} - cpass_{i-1})) \wedge t \notin rts_i \wedge setMatch(t, rps_i)] \\
&= [t | (t \in lts_i - cpass_{i-1}) \wedge t \notin rts_i \wedge setMatch(t, rps_i)] \\
&= [t | t \in lts_i \wedge t \notin cpass_{i-1} \wedge t \notin rts_i \wedge setMatch(t, rps_i)] \\
&= [t | t \in lts_i \wedge t \notin rts_i \wedge setMatch(t, rps_i)] - [t | t \in cpass_{i-1}] \\
&= cpass_i - cpass_{i-1}
\end{aligned}$$

End of proof.

Lemma 6.1 $ckeepLT_i \subseteq lts_i - cpass_i$

Proof:

Suppose $t \in ckeepLT_i$

$$\begin{aligned}
&\Rightarrow t \in [u | u \in lts_i \wedge u \notin rts_i \wedge setNomatch(u, rps_i)] \\
&\Rightarrow t \in lts_i \wedge t \in [u | u \in lts_i \wedge u \notin rts_i \wedge setNomatch(u, rps_i)] \\
&\Rightarrow t \in lts_i \wedge t \notin [u | u \in lts_i \wedge u \notin rts_i \wedge setMatch(u, rps_i)] \\
&\Rightarrow t \in lts_i \wedge t \notin cpass_i \\
&\Rightarrow t \in lts_i - cpass_i
\end{aligned}$$

End of proof.

Keep: We show by induction that, at every step i , the first member of state is equivalent to $ckeepLT_i$ and the second member of state is equivalent to $ckeepRT_i$.

Base case ($i = 1$): Show that $st^1 = (ckeepLT_1, ckeepRT_1, lps_1, rps_1)$

$$\begin{aligned}
& st^1 \\
&= rkeep(rps^1, snd(rstep(rts^1, lkeep(lps^1, snd(lstep(lts^1, st^0)))))) \\
&= rkeep(rps^1, snd(rstep(rts^1, lkeep(lps^1, snd(lstep(lts^1, ([], [], [], []))))))) \\
&= rkeep(rps^1, snd(rstep(rts^1, lkeep(lps^1, snd([], (lts^1 ++ [], [], [], [])))))) \\
&= rkeep(rps^1, snd(rstep(rts^1, lkeep(lps^1, snd([], (lts_1, [], [], [])))))) \\
&= rkeep(rps^1, snd(rstep(rts^1, lkeep(lps^1, (lts_1, [], [], [])))))) \\
&= rkeep(rps^1, snd(rstep(rts^1, (lts_1, [], lps^1 ++ [], [])))) \\
&= rkeep(rps^1, snd(rstep(rts^1, (lts_1, [], lps_1, [])))) \\
&= rkeep(rps^1, snd([], (lts_1, rts^1 ++ [], lps_1, []))) \\
&= rkeep(rps^1, snd([], (lts_1, rts_1, lps_1, []))) \\
&= rkeep(rps^1, (lts_1, rts_1, lps_1, [])) \\
&= (setNomatchTs((lts_1 \setminus rts_1), (rps^1 ++ [])), \\
&\quad setNomatchTs(rts_1, lps_1), lps_1, rps^1 ++ [])) \\
&= (setNomatchTs((lts_1 \setminus rts_1), rps_1), setNomatchTs(rts_1, lps_1), lps_1, rps_1)) \\
&= [t|t < -lts_1 \setminus rts_1, setNomatch(t, rps_1)], [t|t < -rts_1, setNomatch(t, lps_1)], lps_1, rps_1) \\
&= ([t|t \in lts_1 - rts_1 \wedge setNomatch(t, rps_1)], [t|t \in rts_1 \wedge setNomatch(t, lps_1)], lps_1, rps_1) \\
&= (ckeepLT_1, ckeepRT_1, lps_1, rps_1)
\end{aligned}$$

Induction step: Assume that $st^i = (ckeepLT_i, ckeepRT_i, lps_i, rps_i)$.

Prove $st^{i+1} = (ckeepLT_{i+1}, ckeepRT_{i+1}, lps_{i+1}, rps_{i+1})$.

$$\begin{aligned}
& st^{i+1} \\
&= rkeep(rps^{i+1}, snd(rstep(rts^{i+1}, lkeep(lps^{i+1}, snd(lstep(lts^{i+1}, st^i)))))) \\
&= rkeep(rps^{i+1}, snd(rstep(rts^{i+1}, lkeep(lps^{i+1}, snd(lstep(lts^{i+1}, \\
&\quad (ckeepLT_i, ckeepRT_i, lps_i, rps_i)))))) \\
&= rkeep(rps^{i+1}, snd(rstep(rts^{i+1}, lkeep(lps^{i+1}, snd([], \\
&\quad (lts^{i+1} ++ ckeepLT_i, ckeepRT_i, lps_i, rps_i)))))) \\
&= rkeep(rps^{i+1}, snd(rstep(rts^{i+1}, lkeep(lps^{i+1}, \\
&\quad (lts^{i+1} ++ ckeepLT_i, ckeepRT_i, lps_i, rps_i)))))) \\
&= rkeep(rps^{i+1}, snd(rstep(rts^{i+1}, (lts^{i+1} ++ ckeepLT_i, ckeepRT_i, lps^{i+1} ++ lps_i, rps_i)))) \\
&= rkeep(rps^{i+1}, snd(rstep(rts^{i+1}, (lts^{i+1} ++ ckeepLT_i, ckeepRT_i, lps_{i+1}, rps_i))))
\end{aligned}$$

$$\begin{aligned}
&= rkeep(rps^{i+1}, snd([], (lts^{i+1} ++ ckeepLT_i, rts^{i+1} ++ ckeepRT_i, lps_{i+1}, rps_i))) \\
&= rkeep(rps^{i+1}, (lts^{i+1} ++ ckeepLT_i, rts^{i+1} ++ ckeepRT_i, lps_{i+1}, rps_i)) \\
&= (setNomatchTs((lts^{i+1} ++ ckeepLT_i \setminus rts^{i+1} ++ ckeepRT_i), (rps^{i+1} ++ rps_i)), \\
&\quad setNomatchTs(rts^{i+1} ++ ckeepRT_i, lps_{i+1}), lps_{i+1} ++ lps_i, rps^{i+1} ++ rps_i) \\
&= (setNomatchTs((lts^{i+1} ++ ckeepLT_i \setminus rts^{i+1} ++ ckeepRT_i), rps_{i+1}), \\
&\quad setNomatchTs(rts^{i+1} ++ ckeepRT_i, lps_{i+1}), lps_{i+1}, rps_{i+1})) \\
&= (ckeepLT_{i+1}, ckeepRT_{i+1}, lps_{i+1}, rps_{i+1}) \text{ [by Lemmas 6.2 and 6.3, below.]}
\end{aligned}$$

End of proof.

Lemma 6.2

$$setNomatchTs((lts_{i+1} ++ ckeepLT_i) \setminus (rts^{i+1} ++ ckeepRT_i), (rps^{i+1} ++ rps_i)) = ckeepLT_{i+1}.$$

Proof:

$$\begin{aligned}
&setNomatchTs((lts^{i+1} ++ ckeepLT_i) \setminus (rts^{i+1} ++ ckeepRT_i), (rps^{i+1} ++ rps_i)) \\
&= setNomatchTs((lts^{i+1} ++ ckeepLT_i) \setminus (rts^{i+1} ++ ckeepRT_i), rps_{i+1}) \\
&= [t | t \in (lts^{i+1} ++ ckeepLT_i \setminus rts^{i+1} ++ ckeepRT_i) \wedge setNomatch(t, rps_{i+1})] \\
&= [t | t \in (lts^{i+1} ++ ckeepLT_i \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \wedge setNomatch(t, rps_{i+1})] \\
&= [t | (t \in (lts^{i+1} \vee t \in ckeepLT_i) \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \wedge setNomatch(t, rps_{i+1})] \\
&= [t | (t \in (lts^{i+1} \vee t \in [u | u \in lts_i \wedge u \notin rts_i \wedge setNomatch(u, rps_i)]) \wedge \\
&\quad t \notin (rts^{i+1} ++ ckeepRT_i)) \wedge setNomatch(t, rps_{i+1})] \\
&= [t | (t \in lts^{i+1} \vee (t \in lts_i \wedge t \notin rts_i \wedge setNomatch(t, rps_i)) \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \\
&\quad \wedge setNomatch(t, rps_{i+1})] \\
&= [t | (t \in lts^{i+1} \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \vee \\
&\quad (t \in lts_i \wedge t \notin rts_i \wedge setNomatch(t, rps_i) \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \\
&\quad \wedge setNomatch(t, rps_{i+1})] \\
&= [t | (t \in lts^{i+1} \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \vee \\
&\quad (t \in lts_i \wedge t \notin rts_i \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \wedge setNomatch(t, rps_{i+1})] \\
&= [t | (t \in lts^{i+1} \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \vee \\
&\quad (t \in lts_i \wedge t \notin rts_i \wedge t \notin rts^{i+1} \wedge t \notin ckeepRT_i) \wedge setNomatch(t, rps_{i+1})] \\
&= [t | (t \in lts^{i+1} \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \vee
\end{aligned}$$

$$\begin{aligned}
& (t \in lts_i \wedge t \notin rts_i \wedge t \notin rts^{i+1}) \wedge setNomatch(t, rps_{i+1}) \\
& \quad [\text{Since } ckeepRT_i \subseteq rts_i] \\
& = [t | (t \in lts^{i+1} \wedge t \notin (rts^{i+1} ++ ckeepRT_i)) \vee \\
& \quad (t \in lts_i \wedge t \notin rts_{i+1}) \wedge setNomatch(t, rps_{i+1})] \\
& = [t | (t \in lts^{i+1} \wedge t \notin (rts^{i+1} ++ rts_i)) \vee \\
& \quad (t \in lts_i \wedge t \notin rts_{i+1}) \wedge setNomatch(t, rps_{i+1})] \\
& \quad [\text{Recall that } ckeepRT_i = [t | t \in rts_i \wedge setNomatch(t, lps_i)]. \text{ Since } \\
& \quad setMatchTs(lts^{i+1}, lps_i) = \emptyset, \text{ we can safely remove that predicate} \\
& \quad \text{as a condition of list membership, and thus we have } [t | t \in rts_i].] \\
& = [t | (t \in lts^{i+1} \wedge t \notin rts_{i+1}) \vee (t \in lts_i \wedge t \notin rts_{i+1}) \wedge setNomatch(t, rps_{i+1})] \\
& = [t | (t \in lts^{i+1} \vee t \in lts_i) \wedge t \notin rts_{i+1}) \wedge setNomatch(t, rps_{i+1})] \\
& = [t | t \in lts_{i+1} \wedge t \notin rts_{i+1}) \wedge setNomatch(t, rps_{i+1})] \\
& = ckeepLT_{i+1}
\end{aligned}$$

End of proof.

Lemma 6.3 $setNomatchTs(rts^{i+1} ++ ckeepRT_i, lps_{i+1}) = ckeepRT_{i+1}$

Proof:

$$\begin{aligned}
& setNomatchTs(rts^{i+1} ++ ckeepRT_i, lps_{i+1}) \\
& = [t | t \in rts^{i+1} ++ ckeepRT_i \wedge setNomatch(t, lps_{i+1})] \\
& = [t | (t \in rts^{i+1} \vee t \in ckeepRT_i) \wedge setNomatch(t, lps_{i+1})] \\
& = [t | (t \in rts^{i+1} \wedge setNomatch(t, lps_{i+1})) \vee (t \in ckeepRT_i \wedge setNomatch(t, lps_{i+1}))] \\
& = [t | (t \in rts^{i+1} \wedge setNomatch(t, lps_{i+1})) \vee \\
& \quad (t \in rts_i \wedge setNomatch(t, lps_i) \wedge setNomatch(t, lps_{i+1}))] \\
& = [t | (t \in rts^{i+1} \wedge setNomatch(t, lps_{i+1})) \vee (t \in rts_i \wedge setNomatch(t, lps_{i+1}))] \\
& = [t | (t \in rts^{i+1} \vee t \in rts_i) \wedge setNomatch(t, lps_{i+1})] \\
& = [t | t \in rts_{i+1} \wedge setNomatch(t, lps_{i+1})] \\
& = ckeepRT_{i+1}
\end{aligned}$$

End of proof.

Propagation: Finally, we show that our implementation of `rprop` conforms to the propagation rule for difference, showing, by induction, that $cprop_i = \bigcup_{j=0 \dots i} rprop(rps^j, st_j)$.

We do not need to consider the output from the propagation function for the left input, because it is defined as `propT`, which does not output anything.

Base Case: Show $cprop_0 = \bigcup_{j=0\dots 0} rprop(rps^j, st_j)$

$$\begin{aligned}
 \bigcup_{j=0\dots 0} rprop(rps^j, st_j) &= rprop(rps^0, st_0) \\
 &= rprop([], (ckeepLT_0, ckeepRT_0, lps_0, rps_0)) \\
 &= setCombine(lps_0, ([] ++ rps_0)) \\
 &= setCombine(lps_0, rps_0) \\
 &= cprop_0
 \end{aligned}$$

Induction Step: Assume $cprop_{i-1} = \bigcup_{j=0\dots i-1} rprop(rps^j, st_j)$.

Show $cprop_i = \bigcup_{j=0\dots i} rprop(rps^j, st_j)$.

$$\begin{aligned}
 \bigcup_{j=0\dots i} rprop(rps^j, st_j) &= \bigcup_{j=0\dots i-1} rprop(rps^j, st_j) \cup rprop(rps^i, st_i) \\
 &= cprop_{i-1} \cup rprop(rps^i, st_i) \\
 &= cprop_{i-1} \cup rprop(rps^i, \\
 &\quad (lts^i ++ ckeepLT_{i-1}, rts^i ++ ckeepRT_{i-1}, lps^i ++ lps_{i-1}, rps^i ++ rps_{i-1})) \\
 &= cprop_{i-1} \cup rprop(rps^i, (lts^i ++ ckeepLT_{i-1}, rts^i ++ ckeepRT_{i-1}, lps_i, rps_i)) \\
 &= cprop_{i-1} \cup setCombine(lps_i, rps_i) \\
 &\quad \text{[By the definition of } rprop\text{]} \\
 &= setCombine(lps_{i-1}, rps_{i-1}) \cup setCombine(lps_i, rps_i) \\
 &\quad \text{[By the definition of } cprop_{i-1}\text{]} \\
 &= setCombine(lps_i, rps_i) \\
 &= cprop_i
 \end{aligned}$$

End of proof.

The reader may notice that, though we remove data items from state due to punctuations, we do not discard punctuations, hence state may grow without bound. It would be desirable to remove punctuations from state as well as data items. We have investigated removing any punctuations that are output from state, because outputting the same punctuation repeatedly is wasteful. Intuitively, this action is appropriate, though we have not constructed a formal proof that the resulting stream iterator remains proper.

6.7 Discussion

We have shown that our implementations of stream iterators for `dupelim` and `difference` are faithful to their table counterparts. We first defined punctuation invariants for a stream iterator. Our proof for faithfulness was accomplished in two parts: First, we showed that any stream iterator that obeys the punctuation invariants were faithful to their counterpart table operator. Second, we showed that the particular implementation for that stream iterator defined in our framework obeyed the invariant. Appendix A provides proofs for other operators in the same style.

In the next chapter, we will give details of implementation issues that we encountered as we enhanced a real query engine to take advantage of punctuations.

Chapter 7

Implementation Issues Regarding Punctuation Semantics

In order to gain a deeper understanding of the complexity of implementing punctuations and their effect on performance, we made enhancements to the Niagara Query Engine [NDM⁺00] to support punctuation behaviors. Niagara is implemented in the Java programming language [Jav] to execute queries over XML data. In Section 3.2.1, we presented our format for punctuations in XML, using the namespace `punct` to distinguish punctuations from regular data items. Here we discuss general implementation issues and details in enhancing Niagara to support punctuated streams. Performance testing details using our enhanced version of Niagara are given in the next chapter.

We point out here that the implementation issues we address are related to the Niagara Query Engine. These issues are separate from the framework implementation (in Haskell) discussed in Chapter 5.

We first present thoughts on how punctuations might be embedded into a stream in Section 7.1. We follow with a general overview of Niagara and its query operators in Section 7.2. We will then divide our discussion of enhancements to Niagara into three parts: First, in Section 7.3, we discuss general enhancements to Niagara. Second, enhancements to specific operators are discussed in Section 7.4. Third, we introduce two new operators specific to punctuated streams in Sections 7.5 and 7.6. In Section 7.7, we will discuss how punctuations can be used to tolerate disorder.

7.1 Embedding Punctuations in Data Streams

We have discussed how punctuations embedded into a data stream can assist query operators. So far, we have generally ignored questions of how punctuations get into a data stream in the first place. To this point, we have assumed punctuations are emitted at the stream source. However, there are other reasonable ways to embed punctuations into a data stream. Let us suppose a logical operator existed that embedded punctuations in its output. Such an operator could be located in a variety of places: at the stream source, at the edge of the query processor, or after particular query operators within the query itself. We will call this operator the *insert punctuation* operator.

There are many different schemes we could use to implement the insert punctuation operator. Which scheme to choose depends on where the information resides for generating the punctuation. We list several alternatives below:

- **Source or sensor intelligence:** A data stream source may know enough to emit a punctuation. For example, stream sources in the warehouse scenario output data with time values specifying when the report was created. When an hour ends, the sensor is enhanced to emit punctuation that all reports for that hour have been output. This approach is related to the work by Gigascope [SJSM05].
- **Knowledge of access order:** A scan or fetch operator may know something about its source and generate punctuations based on that knowledge. For example, if scan is able to use an index to read a source, it may use information from that index to tell when data items for a given attribute value have been read.
- **Knowledge of stream or application semantics:** An insert punctuation operator may know something about the semantics of its source. In the warehouse example, the temperature sensors likely have temperature limits. Suppose the limits are 20F and 95F. An insert punctuation operator can output punctuations immediately on startup: One that says there will not be any temperature reports above 95F and another that says there will not be any reports below 20F.
- **Auxiliary information:** Punctuation may be generated from sources other than

the input stream, such as relational tables or other files. In the warehouse example, we might store a list of all the sensor units in a local relation. An insert punctuation operator could use that information to determine when all sensors have output all results for a particular hour, and embed the punctuation itself. Such an approach might be used instead of adding logic to sensors to embed their own punctuations.

- **Stream iterator semantics:** Some stream iterators impose semantics on output data. The select operator, for example, can embed additional punctuation that no data items will appear in the stream that would fail the selection predicate. Additionally, the sort operator can embed punctuations based on its sort order. When it emits a data item, it can follow it with a punctuation stating that no more data items will appear that precede that data item according to the sort order.

In our performance testing (see Chapter 8), we use the source intelligence alternative.

7.2 Overview of the Niagara Query Engine

As stated earlier, Niagara executes queries over data in XML format and produces results in XML format. Query plans in Niagara are also described in XML format, with parameters for each query operator contained in separate attributes or elements. For example, one query we will use in performance testing (see Chapter 8) from the online auction scenario converts prices in data items from the Bid stream from US Dollars to Euros, as follows:

```
SELECT bidder, hour, DOLTOEUR(price)
FROM bids;
```

One possible Niagara query plan for this query is shown in XML format in Figure 7.1.

In this query plan, we can see that each query operator is represented by an XML element. Each operator has an attribute called `id` which uniquely identifies that operator in the query plan. Operators that receive data from other query operators in the plan specify their input operator(s) with the `input` attribute, containing the `id` of their input query operator(s). The query plan in Figure 7.1 uses operators such as `unnest` and

```

<?xml version="1.0"?> <!DOCTYPE plan SYSTEM "queryplan.dtd">
<!--Convert prices for bids from Dollars to EUROs
      SELECT bidder, hour, DOLTOEUR(price)
      FROM bids
-->

<plan top="cons">
  <firehosescan id="scan" host="localhost" port="5000" rate="0"
    datatype="pauction_stream" num_gen_calls="55000"
    desc="bid" desc2="" num_tl_elts="1" prettyprint="no"/>

  <unnest id="bid" type="element" regexp="bid" input="scan"/>
  <unnest id="price" type="element" regexp="price" input="bid"/>
  <unnest id="bidder" root="$bid" regexp="bidder"
    type = "element" input="price"/>
  <unnest id="hour" root="$bid" regexp="hour"
    type = "element" input="bidder"/>
  <expression id="doltoeur" variables="price"
    class="niagara.query_engine.DOLTOEURExpression"
    input = "hour"/>

  <construct id="cons" input="doltoeur">
    <![CDATA[
      <bid>$bidder $hour $doltoeur</bid>
    ]]>
  </construct>
</plan>

```

Figure 7.1: Query plan in Niagara XML format to convert bids from US Dollars to Euros.

construct which are specific to querying tree-based data formats such as XML. In Section 7.2.2 we will discuss further the query operators in Niagara.

7.2.1 Representation of Data Items

XML data are often modelled using tree-like structures. When an XML document arrives, Niagara uses a `StreamTupleElement` class object to store the contents of the document. The `StreamTupleElement` class has a vector member to store elements of an XML document, and supports a number of operations over its data, including copying and accessing specific XML documents in the object. In this discussion, we will refer to each

XML document stored in the vector of a `StreamTupleElement` as an *attribute* of that `StreamTupleElement`. (Note, this usage is different from an XML attribute.)

Niagara reads XML data from two kinds of sources: First, Niagara can read from XML files stored locally or on the Internet. Second, Niagara can read from XML stream sources, using the Firehose streaming system discussed in Section 3.3. We will generally focus on data from Firehose streams in this discussion. In our system, a stream of XML data contains many small XML documents, where each document represents a single data item. For example, the Bid stream contains bids for various items up for auction. A sample bid stream is given in Figure 7.2. We shorten the bid component names to reduce the size of each data item (e.g., we use `<H>` instead of `<HOUR>`).

<code><bid></code>	<code><bid></code>	<code><bid></code>
<code><A>40</code>	<code><A>10</code>	<code><A>42</code>
<code>11</code>	<code>42</code>	<code>34</code>
<code><H>2</H></code>	<code><H>2</H></code>	<code><H>2</H></code>
<code><M>0</M></code>	<code><M>0</M></code>	<code><M>0</M></code>
<code><S>30</S></code>	<code><S>30</S></code>	<code><S>32</S></code>
<code><P>66.00</P></code>	<code><P>73.00</P></code>	<code><P>133.00</P></code>
<code></bid></code>	<code></bid></code>	<code></bid></code>

Figure 7.2: Sample bid stream contents. Each bid element is a data item in the stream, and the stream contents are arriving from left to right (in time order). The components for the bid stream are as follows: A is the id of the auction bid upon, B is the id of the bidder, H is the hour, M is the minute, S is the second, and P is the price for that particular bid.

Recall from Chapter 3 that we defined the XML namespace `punct` to differentiate punctuations from data items. During that initial investigation we used the XML parser Xerces [XER] to parse XML data into an internal tree structure. During further testing, we found that this implementation required a great deal of memory when processing a stream containing many small XML documents. As Java implements garbage collection, we found that using the Xerces implementation caused garbage collection to be invoked very often, dramatically affecting the performance of Niagara. We therefore implemented

our own parser for XML data to improve on the management of memory for XML nodes. However, to conserve the amount of memory required for each XML node, we did not implement support for namespaces. Therefore, punctuations in our XML format use the prefix `PUNCT_` instead of using a namespace. The overall behavior remains the same. Figure 7.3 shows a sample bid stream with a punctuation in this form.

<code><bid></code>	<code><bid></code>	<code><PUNCT_bid></code>	<code><bid></code>
<code><A>40</code>	<code><A>10</code>	<code><A>*</code>	<code><A>42</code>
<code>11</code>	<code>42</code>	<code>*</code>	<code>34</code>
<code><H>2</H></code>	<code><H>2</H></code>	<code><H>1</H></code>	<code><H>2</H></code>
<code><M>0</M></code>	<code><M>0</M></code>	<code><M>*</M></code>	<code><M>0</M></code>
<code><S>30</S></code>	<code><S>30</S></code>	<code><S>*</S></code>	<code><S>32</S></code>
<code><P>66.00</P></code>	<code><P>73.00</P></code>	<code><P>*</P></code>	<code><P>133.00</P></code>
<code></bid></code>	<code></bid></code>	<code></PUNCT_bid></code>	<code></bid></code>

Figure 7.3: Sample bid stream contents including a punctuation, using the same data items from Figure 7.2.

We point out here that, though we use XML data in this work, our focus is on “flat” XML elements that represent data items (such as those in Figure 7.3). In this way, we can apply our work to more traditional relational data models. An interesting area of future work, which we will discuss further in Chapter 11, is how to handle and take advantage of more nested XML data.

7.2.2 Query Operators

In general, query operators in Niagara process data in a “push-based” fashion, as opposed to the “pull-based” model used in the stream iterator framework discussed in Chapter 5. In a pull-based model, a parent operator in the query tree reads the output of its child operator(s) directly by invoking the operator to produce its next data item. Data flows through the query tree when a request for data has been made from the top operator in the tree (i.e., when the parent operator pulls data from its child operator(s)).

In a push-based model, a query operator reads data from input queue(s) and outputs

its results to output queues. A query operator does not read the output of its child operator(s) directly. Instead of reacting to data requests from its parent operator, a query operator is directed to process data by a separate scheduler module. No scheduler module is required in a pull-based model. As data from a stream are pushed into a stream processing system (and not simply waiting to be read as in a file), the push-based model is more appropriate for data stream processing. For the example query in Figure 7.1, a partial query tree (with inter-operator queues) is shown in Figure 7.4. Generally, query plans will be shown without queues for readability.

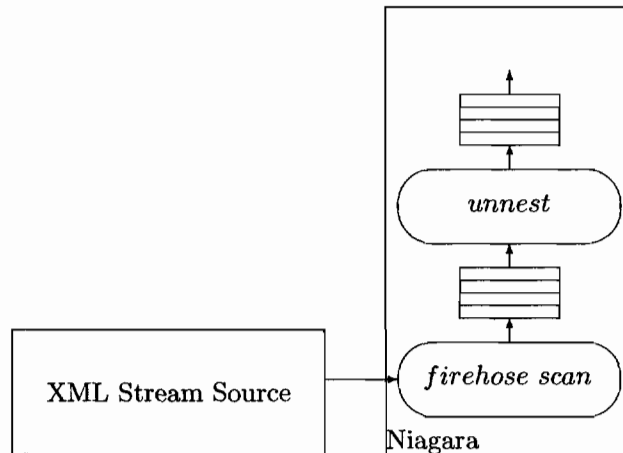


Figure 7.4: Partial Niagara query plan to show queues between query operators. Note, this figure only represents a part of the plan from Figure 7.1.

In Niagara, push-based query operators inherit from the `PhysicalOperator` class. The `PhysicalOperator` class implements generic query operator behavior, including responding to scheduler directives, reading data from the query operator's input queues, and outputting results to the query operator's output queues, similar to the `unary` and `binary` functions in our stream iterator framework. The `PhysicalOperator` class also specifies methods to be implemented by a subclass query operator to define the specific operator's behavior, similar to the `step` function in our stream iterator framework.

Niagara has implementations for many of the traditional query operators discussed in

Section 2.1, including select, group-by and various aggregates, join and union. In addition, Niagara supports the query operators construct, dup, expression, and unnest. These new operators are discussed below:

Unnest and Predicated Unnest The unnest operator takes a regular expression and reads data from a single source queue. Our internal representation of an XML document is modelled in a tree-like structure. We need an operator that can read data from the children of a node in the tree. The unnest operator searches through the immediate children of the input `StreamTupleElement` and finds those child labels that match the given regular expression. (Note that the regular expression can be written to search further descendants of the given tree.) All matches are then unnested into a separate tree structure, which becomes a new attribute in the `StreamTupleElement`. An example is shown in Figure 7.5.

The predicated unnest operator performs the same behavior as unnest, but is also given a predicate. The predicated unnest operation only outputs child nodes that pass the predicate.

Dup The dup operator reads data items from a single source queue and outputs a copy of each data item to one or more output queues.

Expression The expression operator takes a custom Java object that supports the method `processTuple`. For each data item that arrives from its single source, the `processTuple` method is called with that tuple, and a new tuple is returned. In the query plan shown in Figure 7.1, we used an expression operator to convert bid values from US Dollars to Euros. Pseudocode for this expression function is given in Figure 7.6.

Construct The construct operator is similar to the traditional project operator. It takes a list of `StreamTupleElement` attributes, reads data from a single source queue, and builds a new XML document from input `StreamTupleElements`. In the query plan shown in Figure 7.1, the construct operator uses the attributes `bidder`, `hour`, and `doltocur` from each input `StreamTupleElement` object to generate a new XML document with those attributes as elements.

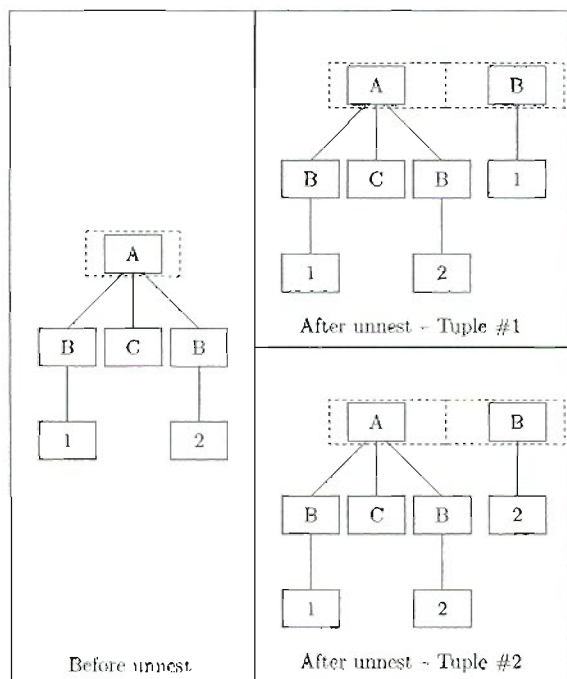


Figure 7.5: Example of an unnest operation. The input tuple is unnested on ‘B’. Before the unnest operation, the `StreamTupleElement` holds a single XML document in its vector. The unnest operator outputs two `StreamTupleElement` objects (one for each ‘B’ child), and each has two XML documents in its vector. The dashed boxes represent the vector for each `StreamTupleElement`.

7.3 General Enhancements to Niagara

In this section we discuss general enhancements made to the Niagara system to support punctuations in input streams. We first discuss a new class that models punctuations and contains the implementations for punctuation functions (such as *match*). Then we discuss modifications made to the `PhysicalOperator` class implementation. In the next section, we will focus on enhancements made to existing query operators.

```

//input : tuple t
function processTuple(t)
    //Get the value for the "price" attribute
    price := t.getAttributeValue("price")
    p.setAttributeValue("price", dollarToEuro(price))
    return t
end function

```

Figure 7.6: Pseudocode for converting tuple values from US Dollars to Euros.

7.3.1 New Punctuation Class

To support processing punctuations in Niagara, we first had to decide how to model punctuations themselves. As in the framework (see Chapter 5), we want to treat punctuations as data items. Specifically, we want to be able to pass punctuations between operators using the same queues that hold data items, and we want to be able to perform the same operations on data items as punctuations, such as copying punctuations. Thus, it makes sense to implement a class to model punctuations that inherits from the class that models data items. Our new class is called `StreamPunctuationElement`, and it inherits from the `StreamTupleElement` class.

The `StreamPunctuationElement` class has an implementation of the `match` function to determine if a given data item matches a punctuation. The implementation of `match` takes a `StreamTupleElement` as its only argument, and returns `true` if the `StreamTupleElement` matches the punctuation. Our implementation first checks if the input argument is a `StreamPunctuationElement`. If so, then `match` returns `false`, as punctuations do not match other punctuations. Otherwise, we match corresponding pattern values in the punctuation with data values in the `StreamTupleElement` object for each child node in the punctuation. We first retrieve text data from each child node for the punctuation and the data item. Then we parse the punctuation node value to determine what kind of pattern it is, and then determine if the data item value matches the pattern. If all data item nodes match all punctuation nodes, then we return `true` from `match`.

Our implementation of `match` brings up an interesting issue. To this point we have focused on handling “flat” relational data, but Niagara processes tree-structured XML

data. Applying punctuations to tree-structured data is an area for future work. In this work we will assume each XML document in the input stream has a root node and leaf nodes, but no interior nodes (i.e., the document is a tree with a height of 1). In this way, we can treat XML data as if they were flat relational data.

7.3.2 Base Class Modifications

A Niagara query operator must implement one of two methods defined by

`PhysicalOperator` for processing `StreamTupleElement` objects:

`blockingProcessSourceTupleElement` or `nonblockingProcessSourceTupleElement`.

Data items read from an input queue are passed to one of these two functions, based on whether the operator blocks on that input or not.

We also need a way for a query operator to process punctuations. In our framework, we separate data items from punctuations. Data items are passed to the `step` function, and punctuations are passed to the `pass`, `prop`, and `keep` functions. The default behavior for these three functions is to ignore punctuations. Similarly, we added a method to `PhysicalOperator` to implement punctuation behaviors, called `processPunctuation`. When a `PhysicalOperator` object reads a tuple from an input queue, it determines whether the tuple is a `StreamTupleElement` or a `StreamPunctuationElement`. The former are handled as before, while the latter are passed to `processPunctuation`. As for `pass`, `prop`, and `keep`, the default behavior for `processPunctuation` is to ignore punctuation. A subclass of `PhysicalOperator` can implement its own version of `processPunctuation` to implement punctuation behaviors appropriate for that operator.

7.4 Enhancements to Specific Niagara Operators

We address two kinds of issues in enhancing query operators to support punctuations: First, in this section we discuss enhancements to existing operators to exploit punctuations. Second, in Sections 7.5 and 7.6, we will discuss the design and implementation of two new operators specifically designed for processing punctuations. The following operators were enhanced to process punctuations: `scan`, `unnest`, `predicated unnest`, `expression`, `select`,

dup, group-by, union, and join. The enhancements for each operator are discussed below.

Scan The scan operator reads a raw XML document and stores its contents in a `StreamTupleElement`. To handle punctuations, it checks the name of the top-level element. If the name has the prefix `PUNCT_`, then the tuple is a punctuation. Its contents are stored in a `StreamPunctuationElement` object, and that object is put in the output queue. If the name does not have the `PUNCT_` prefix, then the tuple is a data item and its contents are stored in a `StreamTupleElement` object, and that object is put in the output queue. Thus, the only punctuation behavior implemented for scan is a propagation behavior to emit punctuations as they arrive.

Unnest and Predicated Unnest Both unnest and predicated unnest find child nodes that match the given regular expression and unnest them. For punctuations, the enhancement to both of these operators is to also unnest child nodes of punctuations as they arrive. Punctuations have the same structure as the data items they match, so the unnest operators are able to unnest punctuations as they do data items. Note that the predicate in predicated unnest is not applied to patterns in punctuations.

Expression In addition to implementing `processTuple` to handle data items, a custom object must also implement the method `processPunctuation` to handle punctuations. As for `PhysicalOperator`, the default implementation of `processPunctuation` is to ignore punctuations (and not output them).

In the query given in Figure 7.1, we used a custom object to convert monetary values from US Dollars to Euros. When a punctuation arrives, the `processPunctuation` method is invoked. This method should handle the different kinds of patterns that can occur for US Dollars. Pseudocode for `processPunctuation` for this example is shown in Figure 7.7.

Ideally, a system enhanced for punctuations could use the `processTuple` function to handle applying the custom expression to punctuations as well. However, this approach will not always work. For a simple example, consider the range pattern and the function that converts numbers to strings. The range pattern `[100, 200]`

```

//input : punctuation p
function processPunctuation(p)
  //Get the value for the "price" attribute
  pricePattern := p.getAttributeValue("price")
  switch pricePattern:
    //wildcards can simply be output
    case "*": p.setAttributeValue("price", "*")
    //Need to convert constant value
    case "a": p.setAttributeValue("price", DollarToEuro(a))
    //Need to convert each value in the range
    case "[a,b]": newA = DollarToEuro(a)
                  newB = DollarToEuro(b)
                  p.setAttributeValue("price", "[" + newA + ", " + newB + "]")
    //Need to convert each value in the list
    case "{as}": pattern := "{"
                  foreach a in as
                    pattern = pattern + DollarToEuro(a)
                    if not last then pattern = pattern + ","
                  pattern = pattern + "}"
                  p.setAttributeValue("price", pattern)
  end switch
  return p
end function

```

Figure 7.7: Pseudocode for converting punctuations patterns from US Dollars to Euros.

matches all numbers between 100 and 200, but not the number 1111. If we blindly convert the range pattern to a string (i.e., ["100", "200"]), then that pattern will match the value "1111". If the value 1111 follows the punctuation in the stream, then after the conversion the value "1111" will match a punctuation that exists before it, violating our requirement of grammatical streams. For this reason, custom objects must process punctuations separately from data items in the general case.

Select and dup The select operator outputs only those data items that pass its predicate. The enhancement to select is to output all punctuations as they arrive. As in the case for predicated unnest, we do not try to apply the selection predicate to punctuations. The dup operator outputs data items from its input to one or more outputs. Like the select operator, dup outputs all punctuations to all output queues

as they arrive. Thus, select and dup implement only propagation behaviors.

Group-by The group-by operator maintains state for each data group as data items for that group arrive. Data for each group are stored in a hash table, where the hash key is generated by the values in the grouping attributes. Recall from the group-by invariants that only punctuations that match all possible data items for one or more specific groups can be processed by group-by. When a punctuation arrives stating that all data items that will contribute to a group have arrived, then results for that group are output and the hash entry for that group is removed. Our implementation only considers single punctuations that cover a group (or groups). We rely on the describe operator (discussed in Section 7.5) to “build up” several punctuations into a single punctuation.

Union The union operator outputs data items from each of its inputs. In this case, union does not perform any duplicate elimination, nor does it attempt to maintain order in its output. The union operator does not store data items in its state, so when a punctuation arrives there is no state to reduce nor are there data waiting to be output. However, we do want union to output punctuations, so we implemented a simple version of the propagation behavior for union. Our enhancement to union is to output matching punctuations from all inputs. Thus, once a given punctuation has been received from all inputs, it is output. Therefore, union must store punctuations in state. We keep an array of vectors, one vector for each input. When a punctuation arrives, we scan the vectors of the other inputs. If punctuations equal to the incoming punctuation exist in each vector, then the punctuation can be output and all equal punctuations can be removed from state.

Join We consider two join implementations here. First, we consider the symmetric hash join [WA91]. As data items from each input stream arrive, they are stored in the hash table associated with that stream. We then probe the other hash table looking for matches and output the join of those matching data items. The two hash tables will grow without bound as data items arrive, and so this implementation of join is unbounded. This behavior is shown in Figure 7.8.

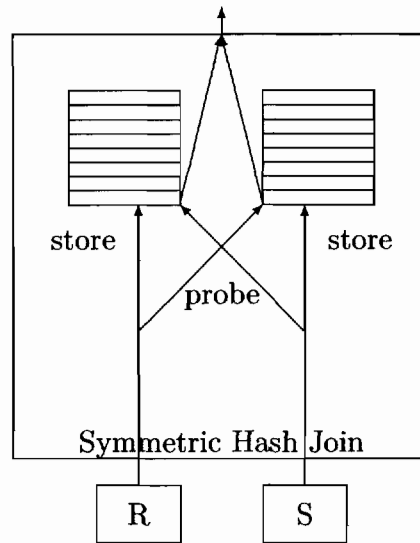


Figure 7.8: Behavior of the Symmetric Hash Join.

In our enhancement for symmetric hash join, we create two additional hash tables for storing punctuations, one for each input stream. Punctuations that arrive are added to the appropriate hash table. A punctuation that matches all possible data items for specific join values is used in two ways: First, it is used to purge data items in the hash table for the opposite stream. We use the values in the punctuation to derive a hash key for the opposite hash table. Since no more data items will arrive that will join with data items for that hash key, those data items can be discarded. Second, the punctuation is joined with punctuations from the opposite stream and emitted in the same way as data items are joined, providing punctuations to other operators further along in the query plan. Thus, we implement keep and propagation behaviors for symmetric hash join.

For example, consider the online auction example from Section 1.1.3. The query to get the final bid for auctions in specific categories requires a join between the Auction and Bid streams on `id=auctionid`. Since data for each auction ends at some point in each stream, we can embed punctuations denoting the end of each auction. When

punctuations arrive from the Auction stream for the end of a particular auction, all data items in the hash table for bids for that auction can be removed. Similarly, when punctuations arrive from the Bid stream, all data items in the hash table corresponding to that auction can be removed.

Note that bids may arrive for an auction after a punctuation denoting the end of that auction arrives on the Auction stream. As bids arrive, we check the auction punctuation hash table for matches. If a match exists, then bid does not need to be stored in the bid hash table.

Note also that the hash tables for the punctuations might grow without bound. In our implementation, we remove a punctuation from the hash table when it joins with a punctuation from the other stream to minimize the amount of additional memory required. Even so, this additional overhead can be significant, especially when an input stream contains a large number of punctuations that are not useful to the join operator. We can reduce the memory requirements by filtering out punctuations that will not benefit the join operator. We present an operator to do so in Section 7.5.

The second join implementation we enhanced was our variation of the extension join [Hon80]. In our implementation, we assume the join attributes for one side of the join forms a key for that input (we call this the *key input*). We also assume that key values arrive before their counterparts in the other input. In this way, our implementation is simply a specialization of the symmetric hash join, where we only need a hash table to store values from the key input. Our enhancement to the extension join follows the enhancement to the symmetric hash join.

At this point, we have made sufficient enhancements to execute the queries from the motivating examples in Section 1.1. Blocking operators are unblocked and stateful operators require less state. There are more sophisticated approaches that could be used in some cases. For example, group-by could combine several punctuations to cover a group. To this end, we have made two additional enhancements that further improve query behavior over non-terminating data streams. First, we have seen that a number of operators only

process certain kinds of punctuations (e.g., group-by). In the next section, we introduce the *describe* operator to ensure that only punctuations that benefit operators in the query are passed through the query tree. Second, we have no way to insert punctuations when a source hits a “lull” or fails. In Section 7.6, we introduce the *punctuate* operator to embed punctuations into the data stream.

7.5 The Describe Operator

We say that a punctuation *describes* a set of attributes if, for a specific set of values for those attributes, the punctuation matches all possible data items for those values. In our punctuation format described in Section 3.2.1, given a set of attributes A in a schema R , a punctuation describes A if the pattern for every attribute in $R - A$ is the wildcard. Consider the punctuations in Figure 7.9. One punctuation describes the `hour` attribute (as well as the hour and minute attributes, and any other combination of attributes containing hour), and the other punctuation describes the `a_id` attribute.

<pre> <PUNCT_bid> <A>* * <H>15</H> <M>*</M> <S>*</S> <P>*</P> </PUNCT_bid> (a) </pre>	<pre> <PUNCT_bid> <A>1023 * <H>*</H> <M>*</M> <S>*</S> <P>*</P> </PUNCT_bid> (b) </pre>
--	--

Figure 7.9: Example punctuations on bids that describe different attributes. The first punctuation (a) describes the hour attribute (H), and the second punctuation (b) describes the auction id attribute (A).

In Section 7.4, we discussed a number of query operators that are limited to processing only certain kinds of punctuations. We use the notion of punctuations that describe certain attributes to help in discussing these limitations. Join, for example, can only remove a data item from its hash tables when it gets a punctuation that describes the set of join

attributes. Similarly, group-by can only be unblocked given a punctuation that describes the set of grouping attributes.

Rather than having operators determine if a given punctuation is useful, we factor this functionality into a new operator, called the *describe* operator. The describe operator outputs data items as they arrive and filters out incoming punctuations that will not help query operators further along in the query tree. In factoring this functionality into a separate operator, we avoid implementing separate code in operators such as join and group-by to verify each incoming punctuation. Further, by taking advantage of equivalences with describe and other operators, we are able to “push” the describe operator down in the query plan in order to filter out punctuations earlier during query execution. Note that the describe operator can be used to filter out all punctuations when the query does not require punctuations for proper execution (for example, filter queries).

Another function that can optionally be performed by the describe operator is to “build up” new punctuations from incoming punctuations when possible. For example, data items in our Bid stream have an hour attribute and a minute attribute. Suppose a punctuation arrives that matches all possible data items for hour 1 between minutes 0 and 15 ($\langle *, *, *, 1, [0, 15] \rangle$), then another arrives that matches all possible data items for hour 1 between minutes 15 and 45 ($\langle *, *, *, 1, [15, 45] \rangle$), and then a third arrives that matches all data items for hour 1 between 45 and 59 ($\langle *, *, *, 1, [45, 59] \rangle$). From these punctuations, we can infer that all data items for hour 1 have arrived, even though we have not explicitly received that punctuation. In this example the describe operator will output a new punctuation that matches all data items for hour 1 ($\langle *, *, *, 1, * \rangle$).

7.5.1 Defining Describe and its Punctuation Invariants

The formal definition for describe is straightforward. We denote the describe operator as $\mathcal{D}_A(S)$, where A is the list of attributes that output punctuations should describe and S is the input stream. Any data item that arrives is output. Therefore, the behavior of describe is:

$$\mathcal{D}_A(S) = S, \text{ if } S \text{ contains only data items.}$$

Now we define the punctuation invariants for describe. The pass and propagation

invariants for `describe` are simple. `Describe` is not a blocking operator. Therefore, the pass invariant for `describe` does not specify any additional data items to be output due to punctuations. Similarly, because `describe` does not store data items in state, the keep invariant is trivial (though it might store punctuations in state).

The main purpose for the `describe` operator is to propagate punctuation, so the propagation invariant is more complex. Recall that `describe` manipulates punctuations in one of two ways: First, only those punctuations that will help the query are emitted. Second, new punctuations are built up from incoming punctuations when possible. The first function of `describe` is relatively easy to define. If a punctuation arrives that describes the desired attributes, then we can pass it on. If not, then ignore it. Given an input schema R :

$$cprop_{\mathcal{D}_A}(ts, ps) = \{x | x \in ps \wedge (\forall a \in (R - A), x(a) = *)\}. \quad (7.1)$$

That is, only output punctuations that have a wildcard value (*) for attributes that are not in the set of described attributes A .

Punctuations that alone do not describe the desired attributes may be combined into a new punctuation that does describe the desired attributes. In the following alternate definition of `cprop` for `describe`, we use the function `setCoalesce` that takes a set of punctuations and builds new valid punctuations to build up output punctuations from input punctuations:

$$cprop_{\mathcal{D}_A}(ts, ps) = \{x | x \in setCoalesce(ps) \wedge (\forall a \in (R - A), x(a) = *)\}. \quad (7.2)$$

This definition outputs all punctuations that describe the desired attributes, as well as the punctuations that can be derived from the punctuations received. Both definitions of `cprop` are valid. Definition 7.2 comes at an implementation cost. We must keep punctuations in state as they arrive to be able to derive new punctuations when later punctuations arrive. Definition 7.1 does not have that added cost.

7.5.2 Implementation of Describe

In our implementation in Niagara, the `describe` operator takes three parameters: the `attributes-to-describe` parameter is required, and the `watch-attributes` and

`watch-patterns` parameters are optional. The `attribute-to-describe` parameter specifies what punctuations are meaningful. In the warehouse query, the meaningful punctuations are those that match all data items for a given hour. Therefore, in that case `attributes-to-describe` is set to `hour`. The `watch-attributes` parameter lists the attributes that can be used to build up punctuations that describe the attributes listed in the `attributes-to-describe` parameter. Each attribute in the `watch-attributes` parameter has a pattern in the `watch-patterns` parameter value defining a range that must be covered by input punctuations in order to generate the new punctuation. Again using the warehouse query as an example, we can watch for punctuations that describe the `minute` attribute for a specific hour, and if they cover the range `[0,59]` for a particular hour, new punctuation can be generated that describes that hour. Therefore, we set `watch-attributes` to `minute`, and `watch-patterns` to the range `[0,59]`.

The implementation for Equation 7.1 is easy — simply walk through the attributes of a punctuation and, for all non-describe attributes, check that the value is the wildcard (*). Our implementation for Equation 7.2 is somewhat simplified. We limit the `watch-attributes` parameter to a single attribute. In order to generate new punctuations, we only handle punctuations with range-type values for the `watch-attributes` and wildcard values for all other non-describe attributes. As punctuations arrive, our goal is to combine ranges for the `watch-attributes` values from multiple punctuations that produce a cover the `watch-patterns` value. When a cover for the `watch-patterns` has arrived, a new punctuation can be output with wildcard values for attributes not listed in the `attributes-to-describe` parameter value. An example of declaring the describe operator in a Niagara query plan is shown in Figure 7.10.

We store incoming punctuations in a hash table. The hash key is built by concatenating values for each non-describe attribute. When a punctuation arrives, we build the hash key and search for existing data. If it does not exist, we use the current range value for the `watch-attribute` as the hash value. If it does exist, we calculate the union of the existing range value with the range value from the punctuation. If that range covers the `watch-pattern`, then we build a punctuation with the wildcard value for `watch-attribute`. Otherwise, we rehash the range back into the hash table and continue.

```

<?xml version="1.0"?>
<!DOCTYPE plan SYSTEM "queryplan.dtd">

<!--Output the number of bids that arrive each hour
SELECT COUNT(*), hour
FROM bids
GROUP BY hour; -->

<plan top="cons">
<firehosescan id="scan" host="localhost" port="5000" rate="0"
  datatype="pauction_stream" num_gen_calls="40000" desc="bid"
  desc2="minute.15" num_tl_elts="1" prettyprint="no"/>

<describe id="descBid" attrsDescribe="H" watchAttr="M"
  watchPattern="[0,59]" input="scan"/>

<unnest id="bid" regexp="PUNCT_bid|bid" input="descBid"/>
<unnest id="bidHour" regexp="H" input="bid"/>

<count id="cnt" groupby="$bidHour" countattr="$bidHour" input="bidHour"/>

<construct id="cons" input="cnt">
  <![CDATA[<res><CNT>$cnt</CNT> $bidHour</res>]]>
</construct>
</plan>

```

Figure 7.10: Use of the describe operator in a Niagara query plan to output punctuations on the hour attribute (H), building up from punctuations on the minute attribute (M).

The pseudocode for this algorithm is given in Figure 7.11.

7.6 The Punctuate Operator

Thus far, we have assumed that data stream sources embed punctuations. This assumption is often reasonable. However, embedding punctuations into a data stream at the source is not always straightforward. There are many factors that make embedding punctuations difficult or inappropriate, including:

- We may not have access to the source code for the stream source, and therefore we cannot enhance it to take advantage of known data constraints.

```

handlePunct(DataItem d)
    fDescribe = true
    fWatch = (watch-attributes.length != 0)

    for each Attribute a in d
        if a.Name not in attrs-to-describe then
            if a.Value != '*' then
                fDescribe = false

                if fWatch AND a.Name != watch-attr then
                    if a.Value != '*' then
                        fWatch = false
                        hashCode = hashCode + a.Value + ";"

    if fDescribe then outputPunct(d)
    else if fWatch then
        checkWatch(d, hashCode)

checkWatch(DataItem d, HashCode hashCode)
    rCurr = hashTable.get(hashCode)
    rNew = Range(d.watch-attr.Value)

    if rCurr == null then
        hashTable.put(hashCode, rNew)
    else
        rNew = union(rNew, rCurr)
        if intersect(rNew, watchRange) == watchRange then
            createAndOutputPunct(d)
            hashTable.remove(hashCode)
        else
            hashTable.put(hashCode, rNew)

```

Figure 7.11: Source code for handling punctuations in the describe operator.

- We may need to embed punctuations within a query plan, rather than simply have them embedded at the source. For example, a system that appends a timestamp to data items as they arrive may also want to embed punctuations for those timestamps (assuming timestamp values are monotonically increasing).
- We may want to enforce rules in our system, and use punctuations to enforce those rules over the input stream. For example, suppose data items are given a timestamp value at the stream source. We want to impose a limit such that data items arriving over five minutes late are invalid. That is, if the timestamp of a data item set at the source is more than five minutes older than the current system time upon arrival, then that data item is invalid. We could periodically embed punctuations at the leaf operator in a query tree that denote the end of data items with a timestamp of the current time minus five minutes, and additionally filter out any data items that arrive more than five minutes late.

We have implemented the *punctuate* operator in Niagara to embed appropriate punctuations into an input data stream at the leaves of the query tree. The *punctuate* operator takes two inputs: The first is the data stream that we want to add punctuation to. Tuples from the first input are generally output immediately. The second contains values output at regular intervals (such as a timer stream). Values in the second input are related to values in the input data stream (for example, timestamps from the source and system time), and are used to build punctuations for the output stream. We use the second stream in a manner similar to *heartbeats* described by Arasu et al. [ABW03]. Since the second stream outputs values at regular intervals, we can be sure that punctuations are also output at regular intervals, even when there is a “lull” in the input data stream.

For example, suppose a system-generated timestamp is added to every data item on arrival. We might add a *punctuate* operator to the query plan to take input data items (with the added timestamp attribute) as well as a system time stream that outputs the current time every minute. The *punctuate* operator would output bids as they arrive. When a value t from the system timer stream arrives, the *punctuate* operator would emit a punctuation, using the schema of the bid stream, stating that all bids with timestamp

less than or equal to t have appeared on the stream.

Optionally, the punctuate operator can enforce its punctuations. That is, a data item that arrives from the input stream is compared with all punctuations that have been emitted by the punctuate operator. If the data item matches any of those punctuations, it is not output, thus enforcing the punctuations that have already been emitted. In the presence of unreliable inputs, we must enforce any punctuations that are inserted.

Consider a query in the online auction scenario that outputs the number of bids that arrive each hour. We would like to emit punctuations from the Bid stream source when all bids for a particular hour have been output. In a perfect world, this behavior is acceptable. However, suppose our system runs over the Internet. Different bids may take different paths, and therefore may not arrive in the order they were output. There is no guarantee that a punctuation emitted after a bid will arrive after that bid. Therefore, we cannot send punctuations from the bid stream source and rely on the arriving stream to be grammatical.

We can handle this issue by making some changes to the online auction architecture, including the use of a timer stream and the punctuate operator. We first modify our stream sources so that they do not emit punctuation. The plan is shown in Figure 7.12. Tuples from the bid stream feed into the punctuate operator along with periodic input from the timer. The punctuate operator builds punctuations for the bid schema based on values from the timer stream. Any late-arriving bids that violate punctuations that have been emitted from the punctuate operator are filtered out, enforcing our punctuations. The group-by operator uses those punctuations to know when all valid bids for a particular hour have arrived, and outputs results for that hour.

7.7 Dealing with Disorder

Order can be important in processing data streams. The order of items in a stream may carry information such as temporal sequence of events or measurements. Preserving such information may require query operators to maintain order of stream elements. Query specifications may depend on order, such as selecting all local maxima or windowing

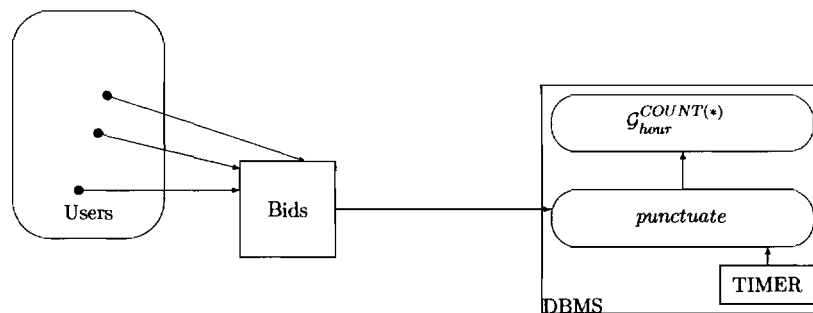


Figure 7.12: Query plan to determine the number of bids for items each hour, using the punctuate operator to embed punctuation.

based on position of items in order (for example, sliding average over the last n items). Knowledge of stream order can benefit certain operators, such as aggregates, that might otherwise block or retain state until the end of the input.

Order can be implicit (for example, arrival sequence of items) or explicit (for example, via timestamps or sequence numbers). There may be multiple explicit orders of interest on a single stream [JCSS03]. Obviously, implicit order can be converted to explicit order by augmenting data items with an additional attribute. Converting explicit order to implicit order (that is, sorting on an ordering attribute) is not always so simple with data streams, nor is it always desirable.

Disorder arises for several reasons: Data items may take different routes, with different delays, from their source; the stream might be a combination of many sources with different delays; the ordering attribute of interest (event start time) may differ from the order in which items are produced (event end time).

There have been several approaches suggested to deal with disordered streams. A common approach is to sort data items into the order of interest as they arrive. That approach has several problems. One is that it introduces delay — a received item cannot be released for processing until it is known that all items earlier in the order have been received. Another is that sorting requires space for buffering out-of-order tuples. A third is that it may conflict with policies (such as quality of service) that want to give certain data items higher priority for processing. Finally, we may not know a priori that all data items up to a certain point in the order have been received.

A second class of approaches leaves the stream disordered, but uses (or enforces) constraints on the amount of disorder so that data items may be processed as they arrive. One such approach is the use of a *slack* parameter for operators of the Aurora system [CQC⁺02] that defines how far displaced (in terms of number of data items) any data item can be from its correctly ordered position. Slack can be used both for ordered data approaches and window approaches. For example, let S be the list of integers [1, 2, 4, 5, 3, 6]. Notice that, with the exception of the value 3, the data items are sorted. Suppose an operator O were given S as input. If we define the slack parameter for O as 3, then O could process S as if its data items were arriving in order. It could simply buffer data in a cache the size of the slack parameter (actually *slack* + 1), and always output the next result in the sort order when the cache is full. In Aurora slack is defined on individual query operators.

7.7.1 Querying Over “Nearly-ordered” Attributes

In the online auction system, bids should be processed in the order that they are posted, not the order they arrive. We cannot assume that individual items arrive in order. However, in most cases we can assume items will arrive within some constant bounds of where they belong in the sort order. Slack can be set here to the bound on how out-of-order the input can become. Operators can use this bound to reorder the input.

We use punctuations to address this issue. Suppose, for any data item in the stream, we know that no data items will arrive that precede it in the sort order after s seconds. We set up a timer stream to output timer values of the current system time minus s seconds at regular intervals. The punctuate operator receives these values and outputs punctuations stating that no more data items will appear in the stream with the datetime value less than the value from the timer. The advantage of this approach is that other operators in the query can take advantage of punctuations. Slack, in comparison, is defined for each operator.

In the online auction example, suppose we want to know how many bids arrive each hour, and we know that bids may be delayed up to five minutes. We can set up a timer operator to output every minute the system time minus five minutes. When each timer value arrives in the punctuate operator, a punctuation is generated in the same schema

as data items in the Bid stream, using the timer value in the `time` attribute and wildcard values for all other attributes. Bids do not need to be put back in order. In such an approach, late-arriving bids must be filtered out.

7.7.2 Maintaining Order

Even if the input is sorted, we often want to maintain the sort order throughout the query. For many operators, such as `select`, this requirement is trivial. However, other operators must take extra care to ensure their output is in order. For example, the order-preserving version of `union` must get data items from all inputs before outputting a result. In a traditional DBMS, this behavior is common (for example, the merge behavior in the sort-merge implementation of `join`). However, data streams complicate processing. We cannot guarantee that data items will be available from all input streams. One stream input may be intermittent or down, causing merge to block. It cannot output a result until it has a data item from every input.

The Gigascope developers [JCSS03] address maintaining order in an order-preserving union. Gigascope executes queries over network packet data, ordered by timestamp. Fragmented packets are split from the stream and reconstructed, then merged back into the original stream. The stream of newly reconstructed packets is lower volume than the original stream, often causing merge to block for a reconstructed packet. They insert tokens (that can be considered a special form of punctuations) denoting a minimum timestamp into the reconstructed packet stream when it has no data items to process. Data items from the original stream with a timestamp less than the token are output.

The punctuate operator gives us the same benefits as the token approach used in Gigascope but in a more general manner. We can set up a query plan as shown in Figure 7.13 where network packets and a timer can be inputs to the punctuate operator. Punctuations will be emitted from the punctuate operator when the timer outputs a data item. In this query plan, we show the split operator. The split operator sends the punctuations it receives to two outputs, where items that pass the given predicate go to one output and items that fail go to another output. We implement split using a combination of `dup` and two `select` operators.

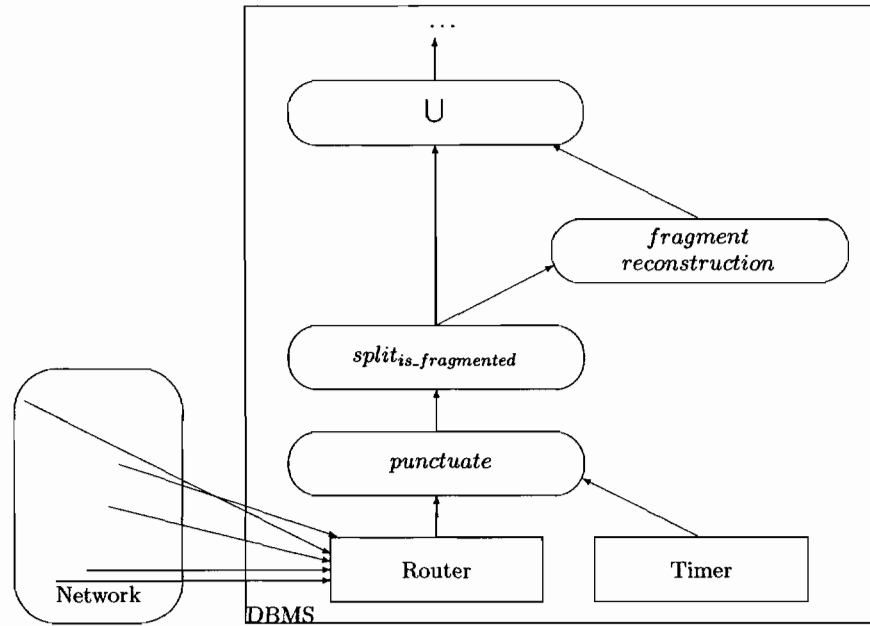


Figure 7.13: Query subplan to handle packet reconstruction with a punctuate operator.

The packet reconstruction operation must be enhanced to handle punctuations. We take an approach similar to Gigascope, where the packet reconstruction operation emits timestamp tokens when there is no work to do. A punctuation cannot be output until its timestamp is less than any timestamp for any packet currently under construction. Once the union operator receives punctuations from each input, it can output all data items in order that have timestamps less than the punctuations, and then its punctuation. In this way, the output data items remain in order based on timestamp. As usual, the advantage of using punctuations is that the punctuations emitted from merge can be used by other operators in the query (such as joins or aggregates).

Punctuations give us another advantage in this example. Embedding punctuations in the stream allows us to remove the order-preserving functionality of this implementation of merge, and simply output data items as they arrive. Operators further along in the query tree can use the punctuations to process the data stream without having to rely on order. This approach allows data items to be emitted sooner, because they do not have to wait to get back in order.

7.8 Summary

We have discussed the various enhancements made to the Niagara Query Engine to support punctuated streams, including general enhancements, enhancements to specific existing operators, and new query operators designed specifically for handling punctuations. The next step is to test these enhancements using one of the stream applications discussed in Section 1.1. In the next chapter, we will present results from our performance testing.

Chapter 8

Performance Results

We have made the enhancements discussed in the previous chapter to the Niagara Query Engine. At this point, we want to address questions related to the performance overhead of executing queries over punctuated streams. In Chapter 3, we described a preliminary evaluation to get a general idea of the overhead of punctuations. Here, we report a more thorough investigation of how query execution performance is affected by punctuated streams. The general performance question we address is, “What is the overhead required by punctuations on entire queries?” Embedding extra information, such as punctuations, in a data stream requires extra bandwidth. How do punctuations influence the rate at which data flow through a query engine? Also, what effect does processing punctuations have on overall memory usage, in terms of data items as well as punctuations? Further, query operators have extra work to do in the presence of punctuations. What is the processing overhead for entire queries? Finally, many useful queries over non-terminating data streams execute successfully without any kind of help (e.g., queries that filter data streams based on some simple predicate). How do punctuations affect the performance of such queries?

8.1 Scenario

We will use an extension of the online auction system discussed in Section 1.1.3 as the scenario for our performance tests. There are a number of examples of commercial and research systems for processing and monitoring online auctions, including eBay [EBA], Yahoo! Auctions [YAH], the Fishmarket [RNSP97], Emporias [RW01], and the Michigan

Internet AuctionBot [WWW98]. In an online auction, humans must first register with the auction monitoring system. Once registered, they can participate in auctions as buyers or sellers through software agents. New items are added to the system from a seller to be sold in new auctions. Each item can be placed in a hierarchical category to make it easier for potential buyers to find. Bids continuously arrive from potential buyers for items open for auction. We describe below three kinds of data streams arising from this scenario.

8.1.1 Streams for the Online Auction Monitoring System

We model an online auction with three kinds of stream sources (which can be implemented as software agents) that supply data to an auction monitoring system, as shown in Figure 8.1. A relational database may be used to maintain category information. Each kind of source represents different interactions users have in the online auction, as follows:

- The Person stream source contains registration information for new users who want to participate in auctions. Users may participate in auctions as buyers or sellers. Note the Person stream is only listed here for completeness of the scenario. It is not involved in any of our test queries.
- The Auction stream source contains information about items for sale, including seller id, time period for the auction, and category that the item belongs to. The item is open for bidding at the instant it arrives on the Auction stream.
- The Bid stream sources contain bids on items for sale. Each bid contains the bidder id, the item being bid on, the time the bid was placed, and the amount of the bid. There may be multiple bid streams in our system. Each Bid source is ordered on time. However, when multiple bid sources are used, they are not synchronized.

8.1.2 Queries for the Online Auction Monitoring System

There are many kinds of queries an online auction monitoring system might use to track the progress and usage patterns of auctions on behalf of both users and administrators of the system. The query set we use in our performance tests contains five queries, which are

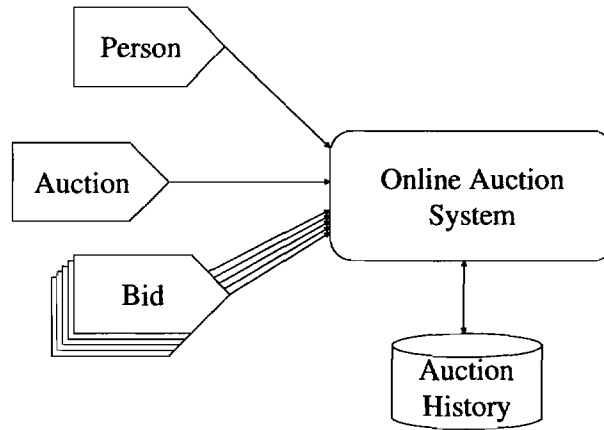


Figure 8.1: Architecture for the On-line Auction System

intended to be realistic for our scenario. It is also designed so that we can evaluate the performance overhead of embedding punctuations into data streams. The first two queries use operators that are already appropriate for data streams — that is, they do not block and do not accumulate an unbounded amount of state. We use these queries to test the performance overhead of punctuated streams on queries that do not require punctuations. The other three queries use operators that cannot be executed over non-terminating data streams without help, to show how punctuations affect queries that contain operators that require punctuations. As it is difficult to compare queries over non-terminating input streams, we use terminating input streams in our tests. We are interested in the amount of time that it takes to execute queries, and in the memory requirements during query execution. For each query, we vary the frequency of punctuations in data streams, including the case where no punctuations are embedded. Punctuations are embedded by the stream source.

We express our queries in SQL, due to its familiarity. The first three queries read from a single bid source stream, rather than multiple bid sources, so that bids arrive in order based on time. In doing so, we are able to analyze the results of each test without having

to account for disorder in the stream. For the same reason, the fourth query uses only one bid stream along with the auction stream. The fifth query contains a union of multiple bid streams, so that we can specifically test how disorder is handled.

1. Currency conversion

```
SELECT bidder, hour, DOLTOEUR(price)
FROM bid1;
```

This query reads from one bid source and converts the bid price value from US Dollars to Euros. Each bid item passes through an “expression” operator, which processes each data item as it arrives with a user-defined function. This query does not benefit from punctuations, and so will show the overhead of embedding punctuations in a data stream for a query not requiring punctuations.

2. Specific bid ranges

```
SELECT a_id, price
FROM bid1
WHERE 350<=price AND price<=450;
```

This query filters out bids from one bid source that are not within a price range. Since it is a simple query using select and project operators, it will not benefit from punctuations. We again will assess the overhead of embedding punctuations when they are not required.

We point out that there are certain punctuations that select can take advantage of. In this query, for example, if a punctuation arrives that states that no more data items will arrive with a price less than \$500, then select will know that no more data items will pass its predicate, and can therefore stop reading from the bid stream, and send an all-wildcard punctuation (essentially end-of-stream). While these are interesting possibilities, they are not relevant to our performance testing.

3. Bid counts

```
SELECT hour, COUNT(*)  
FROM bid1  
GROUP BY hour;
```

This query outputs the number of bids posted from one bid source during one hour time periods. Its results can be used to determine peak and low-traffic periods. Since group-by is a blocking operator, it cannot be executed over a non-terminating data stream without help. Punctuations that describe data items for a particular hour will unblock this query, allowing data to be output before the entire contents of the stream have been read.

4. Closing price for auctions in specific categories

```
SELECT B.a_id, MAX(B.price)  
FROM auction A, bid1 B  
WHERE A.a_id=B.a_id AND A.category IN {92, 136, 208, 294}  
GROUP BY B.a_id;
```

This is the query described in Section 1.1.3. It determines the winning bid for auctions in particular categories. It requires a join between the auction stream and one bid stream, thus enabling us to measure the growth of state during query execution and how punctuations help reduce the amount of state. This query also uses a group-by operator to determine the maximum bid price for each auction. Since this query uses both an unbounded stateful operator and a blocking operator, it also cannot be executed over non-terminating data streams without help. Punctuations should reduce the amount of state for join and unblock group-by.

5. Union of bid counts

```
SELECT hour, COUNT(*)  
FROM (SELECT * FROM bid1  
      UNION ALL
```

```

SELECT * FROM bid2
UNION ALL
SELECT * FROM bid3
UNION ALL
SELECT * FROM bid4
UNION ALL
SELECT * FROM bid5)
GROUP BY hour;

```

As with Query 3, this query outputs the number of bids posted during one hour time periods. However, in this case we combine inputs from five separate bid streams before determining the count. Punctuations unblock the group-by operator. For our testing purposes, the bid sources are not synchronized on a global time. Bids from each stream still arrive in order based on bid time. However, the result of simply merging the bid streams is not guaranteed to be ordered on time. We use this query to show how disorder is handled using punctuations versus using slack when the input does not arrive to the group-by operator in order.

For Queries 1, 2, 3, and 4, we tested the effects of using the describe operator introduced in Section 7.5 to filter out unwanted punctuations. Additionally, we “built up” punctuations on the hour attribute based on patterns of punctuations on the minute attribute for Query 3. The query plans for Queries 3 and 4 are shown in Figure 8.2 and Figure 8.3 using the describe operator. The describe operator is shown for both plans in a dashed box, because it is not required for query execution. In our performance tests, we compare performance results using versions of these queries with and without the describe operator.

8.2 Generating Stream Contents

The issues we considered in developing these performance tests fall in two categories: data generation and implementation. The implementation issues have been discussed already in Chapter 7. Here we address data-generation issues specific to the online auction and

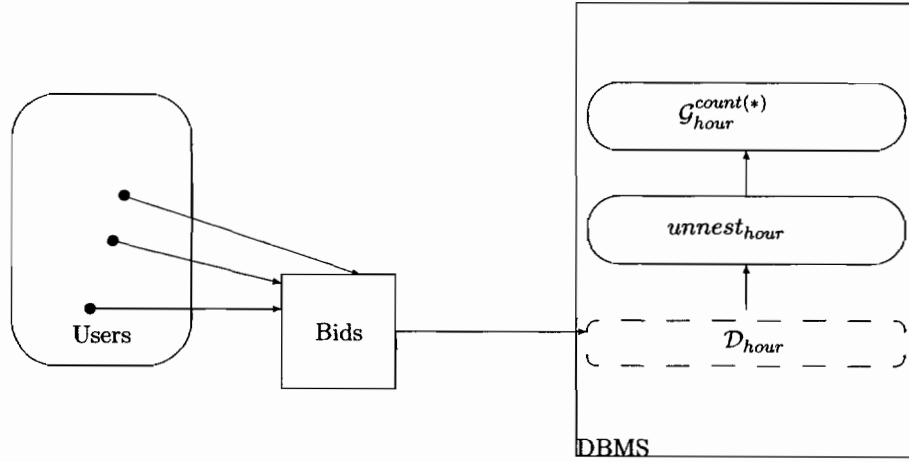


Figure 8.2: Plan for Query 3 using the describe operator. The describe operator is shown in a dashed box since it is not used in all query configurations.

punctuated streams. We developed a system to stream auction data in XML format into the Niagara Query Engine. All three kinds of data items (bids, auctions, and persons) may be contained in a single stream. In each test, the stream only includes data relevant to the query being tested. For example, Query 1 requires only bid data, and Query 4 requires bid and auction data.

8.2.1 Data Generation

Data for the auction are generated as follows: For person data, the name, email, city, and state values are selected randomly from a list of acceptable values (as defined in the data generator for XMark [SWK⁺01]). For auction data, the seller is a random ID value for a person that has already registered with the system. Note that, though our queries do not use the Person stream, we assume that some process exists that is actively registering users. That is, we assume that the bidder id values for bids and seller id values for auctions are valid persons that registered through the Person stream. The queries in our test set do not verify the validity of person identification values. The auction category is an integer value randomly distributed between 0 and 302. The auction expiration time is a time period value randomly distributed between two hours and twenty-four hours. An auction starts at the time it enters the system. Bids are generated for a valid auction (an

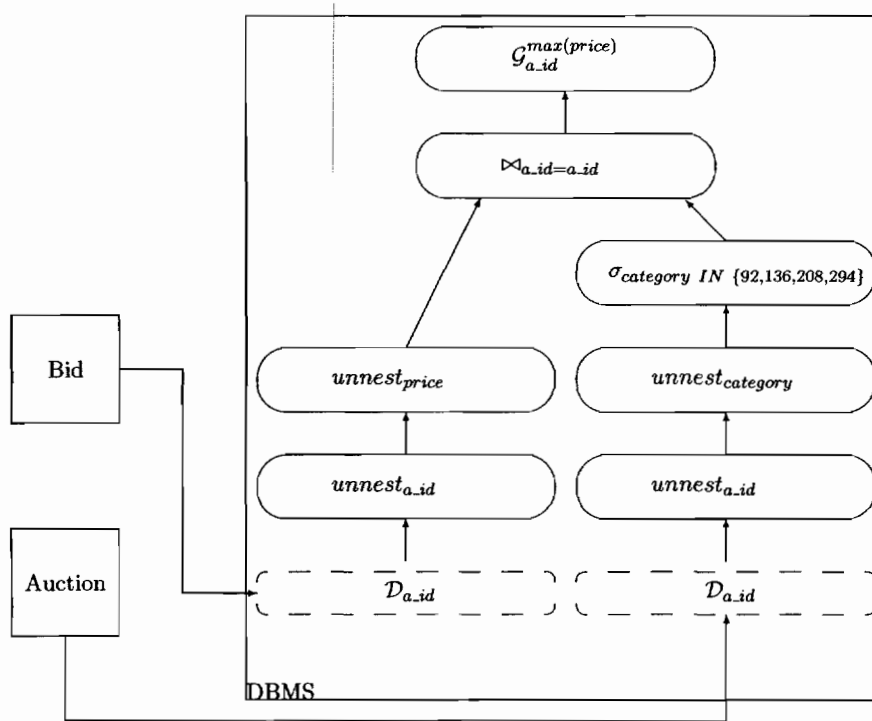


Figure 8.3: Plan for Query 4 using the describe operator. The describe operators are shown in dashed boxes since they are not used in all query configurations.

auction that has already been sent to the system), with a valid person, time and a price that exceeds the current price for that auction. The price increase is a value randomly distributed between \$1 and \$25. On average, for every person in the stream there are ten auctions and one hundred bids in the stream.

Time values in the data are simulated. We use an internal object to simulate a clock, which is updated on average every ten bids. Time advances by 30 seconds on average.

Since the Niagara Query Engine operates over XML data, our input stream contains flat XML documents for the auction data. That is, each XML document consists only of a root node and leaf nodes. Root nodes in the XML structure are full stream names. As we discussed earlier, in order to reduce the size in bytes of the XML data, names of the child nodes are abbreviated to a single character. The structure for the different auction data items is shown in Figure 8.4.

<pre> <person> <P>pid</P> <N>name</N> <E>email</E> <C>city</C> <S>state</S> </person> </pre>	<pre> <auction> <A>a.id <E>expires</E> <S>seller</S> <C>category</C> </auction> </pre>	<pre> <bid> <A>a.id bidder <H>hour</H> <M>minute</M> <S>second</S> <P>price</P> </bid> </pre>
(a)	(b)	(c)

Figure 8.4: The XML structure for items from each stream source: (a) Person data items (b) Auction data items (c) Bid data items.

8.2.2 Generating Punctuations

Punctuations are embedded in the stream at the source. In our testing, a stream will contain one of the following kinds of punctuations: a) punctuations that denote the end of an auction, b) punctuations that denote the end of some range of hours, c) punctuations that denote the end of some range of minutes, or d) punctuations that denote the end of some range of seconds. Any combination of these is also acceptable. End-of-auction punctuations can be embedded into both the auction and bid stream when the time period for an auction has elapsed, indicating that no more data items will arrive from either stream for a particular auction id. Punctuations marking the end of a time period (such as the end of an hour) are added to the bid stream after a bid for a new time period has been output. Punctuations over time periods are only added to the bid stream.

In Figure 8.5, we show two punctuations. The first matches all bids between the hours of 1 and 5, and the second matches all bids for auction id 6.

8.3 Performance Results

Our tests were run on a Pentium 4 with a 2.8 GHz processor and 512Mb RAM, running Red Hat Linux 9.0, kernel 2.4.20-20.9. Niagara and the Firehose were run in Java 1.4.1, using a maximum of 324 MB of memory. For each query, we generated at least two data sets. The types of data sets are listed in Table 8.1.

Our tests were run at “full throttle.” That is, we do not attempt to output stream

<PUNCT_bid> <A>* * <H>[1,5]</H> <M>*</M> <S>*</S> <P>*</P> </PUNCT_bid> (a)	<PUNCT_bid> <A>6 * <H>*</H> <M>*</M> <S>*</S> <P>*</P> </PUNCT_bid> (b)
---	---

Figure 8.5: Example punctuations for bid data items: (a) Punctuation that matches all bids between the hours of 1 and 5 inclusive, and (b) punctuation that matches all bids for auction id 6.

Configuration	Punctuations	Queries
data-NP	None	1, 2, 3, 4, 5
data-A	Ending each auction	1, 2, 3, 4
data-H1	Ending each hour	1, 2, 3, 4, 5
data-M15	Ending 15 minute intervals	1, 2, 3, 4
data-M1	Ending each minute	1, 2, 3, 4
data-S30	Ending 30 second intervals	1, 2, 3
data-S15	Ending 15 second intervals	1, 2, 3

Table 8.1: The data sets used in our performance tests. The Configuration column names the different data sets used. The Punctuations column reports the kinds of punctuations embedded in that data set, and the Queries column lists the queries that were executed over that data set. Each data set was run through each listed query five times.

items at a given time, rather, the queries run as fast as possible — the goal being to see what the maximum rate that a query can process at (as reflected by the total elapsed time).

8.3.1 Test Configurations

For the first four queries, we used three configurations of Niagara queries: First, we ran with no punctuation enhancements for any Niagara query operators. Thus, the leaf query operators in the query plans filtered out all punctuations. In the second configuration we enabled the punctuation enhancements described in Chapter 7. All punctuations embedded in the input stream were processed by each operator according to the appropriate

punctuation behaviors for that operator. Finally, we added the describe operator to each query at the leaf level to filter out punctuations that are not useful, and to build up useful punctuations where possible. The parameters for the describe operator were different for each query. For Query 1 and 2, there are no useful punctuations, so the `attributes-to-describe` parameter was empty. Thus, all punctuations are filtered out by the describe operator. For Query 3, the group-by operator is unblocked by punctuations on hour, so we set `attributes-to-describe` to H. Further, we set `watch-attributes` to M, and `watch-patterns` to [0,59], to build up punctuations on hour when sufficient coverage was made by punctuations over the minute attribute. For Query 4, the join and group-by operators can both take advantage of punctuations over the auction id attribute, so we set `attributes-to-describe` to A. We cannot build up punctuations on `a_id` from other kinds of punctuations in this case, however, so the other two parameters are empty. Further, all other kinds of punctuations (e.g., punctuations on hour) are filtered out. Since Query 5 was used to compare the handling of disorder between slack and punctuations, there were only two configurations used: First, we ran the test using data without punctuations and various settings for the slack parameter. Second, we ran the test using data with punctuations at the end of each hour.

For Query 1, we generated 522,032 data items¹. The stream contained only bids and punctuations on bids as specified. We used seven different stream configurations, listed in Table 8.2. The results for Query 1 are shown in Figure 8.6. For Query 2 and Query 3, we generated 379,736 data items. Again, the stream contained only bids and punctuations on bids when specified, and we used seven different stream configurations. The data set configurations for Query 2 and Query 3 are listed in Table 8.3. The results for Query 2 are shown in Figure 8.7, and the results for Query 3 are shown in Figure 8.8. For Query 4, we generated 264,613 data items. The stream contained bid and auction data items, and punctuations on bids and auctions where specified. We used five data configurations, listed in Table 8.4. The results for Query 4 are shown in Figure 8.9. Finally, for Query

¹In our data generation routine, we specify a certain number of passes. Since the number of auctions and bids are determined randomly based on the number of passes, the final number of data items in the stream will seem somewhat odd.

5, the data configurations are shown in Table 8.5. We discuss the results for Query 5 in Section 8.3.3.

Configuration	Size (bytes)	Gen Time (s)	Punctuations
data-NP	38,362,561	16.404	0
data-A	38,739,100	16.498	2461
data-H1	38,431,582	16.379	941
data-M15	38,708,883	16.574	4405
data-M1	42,552,049	16.849	66186
data-S30	51,201,054	17.622	162031
data-S15	60,122,269	18.354	274028

Table 8.2: Data set characteristics for Query 1. Size contains the data set size in bytes. Gen Time lists the number of seconds to put data in the data stream. Punctuations lists the number of punctuations in the data set. Each row specifies the kinds of punctuations: data-NP means no punctuations in the stream, data-A means punctuations on auction_id are embedded in the stream, data-H1 means punctuations are embedded every hour in the stream, data-M15 means punctuations are embedded every 15 minutes in the stream, data-M1 means punctuations are embedded every minute in the stream, data-S30 means punctuations are embedded every 30 seconds in the stream, and data-S15 means punctuations are embedded every 15 seconds in the stream.

Configuration	Size (bytes)	Gen Time (s)	Punctuations
data-NP	27,790,296	9.068	0
data-A	28,058,852	9.186	1776
data-H1	27,840,483	9.077	688
data-M15	28,042,332	9.095	3205
data-M1	30,836,112	9.321	41326
data-S30	37,123,510	9.88	117845
data-S15	43,609,399	10.347	199311

Table 8.3: Data set characteristics for Query 2 and Query 3. See Table 8.2 for meaning of columns and configurations. Note that data-S15 was run only for Query 2.

8.3.2 Discussion of Performance Results

From the results for the first four queries, we can see that when the ratio of punctuations to data items is below approximately 15% in the data stream, the overhead of processing punctuations on the performance of the query is not significant. The size in bytes of the input for that ratio of punctuations does not increase significantly, suggesting that

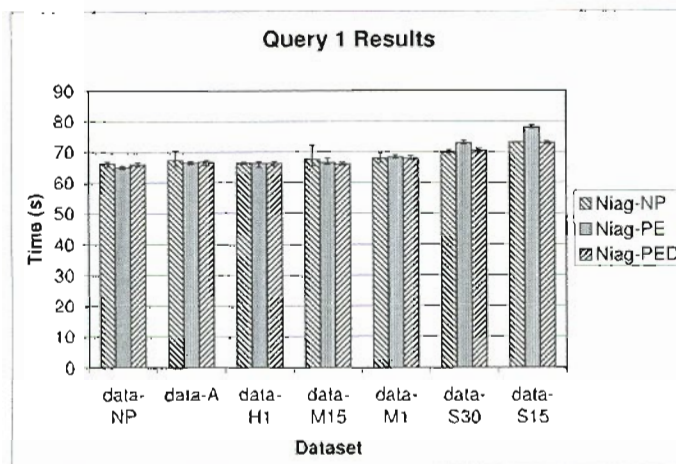


Figure 8.6: Results for Query 1 and 2. For each data set, we ran three different configurations of Niagara: Niag-NP is the Niagara Query Engine without any punctuation enhancements. Niag-PE is the Niagara Query Engine with punctuation enhancements. Niag-PED is the Niagara Query Engine with punctuation enhancements, and queries containing the describe operator.

bandwidth is not noticeably affected. Even for higher ratios of punctuations to data items, where the data set contains one punctuation for every two data items, the time to execute the queries does not increase dramatically, with the exception of Query 4. For cases where unnecessary punctuations do significantly increase the overall execution time, adding the describe operator to the query decreases execution time to close to the level when the operators are not punctuation-aware. By filtering out unnecessary punctuations when they enter the system, we reduce the processing overhead for those punctuations. This effect is most evident in Query 4 (see Figure 8.9, data set data-M1), where every punctuation for a one minute period is added to the hash table in the join operator. These punctuations are not going to benefit the join operator, and they use up space. By filtering those punctuations out with the describe operator, we reduce the overall processing overhead.

In addition, we are able to see the benefits of punctuations in Query 4 in Figure 8.10.

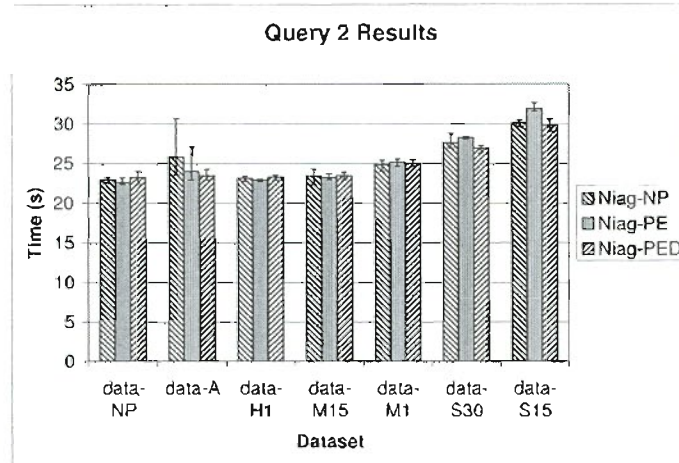


Figure 8.7: Results for Query 2. See Figure 8.6 for explanation of the legend.

Configuration	Size (bytes)	Gen Time (s)	Punctuations
data-NP	19,065,680	4.295	0
data-A	19,534,468	4.605	1570
data-H1	19,097,033	4.344	431
data-M15	19,223,143	4.389	2007
data-M1	20,970,102	4.451	25875

Table 8.4: Data set characteristics for Query 4. See Table 8.2 for meaning of columns and configurations.

Query 4 uses join and group-by operators. Punctuations embedded to denote the end of auctions allow join to reduce the amount of state required and group-by to output results for specific groups early. To test the size of state throughout query execution, we instrumented join to track the number of data items held in state with and without punctuations. As expected, the number of data items held in state for join was bounded when punctuations were embedded in the stream to denote the end of an auction. When a punctuation arrived denoting the end of an auction, the state for that auction was purged. Without punctuations, the number of data items held in state for join grew steadily as more data items arrived.

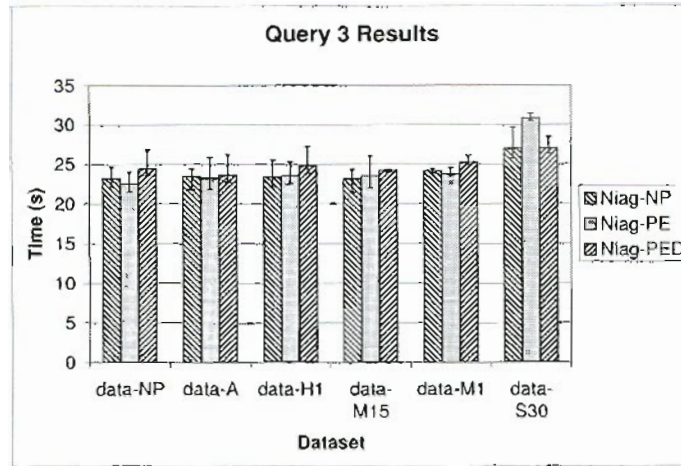


Figure 8.8: Results for Query 3. See Figure 8.6 for explanation of the legend.

We see in Figure 8.10 that, without punctuations, the group-by operator holds all output until the end of the input is reached, whereas punctuations allow data items to be output much sooner. That results are output before the end of the input arrives shows that punctuations flow through each operator in the query plan, all the way to the group-by operator. In the case of non-terminating data streams, no results would ever be output. That case requires some other approach to generating output, such as modifying the query by defining windows.

Further, by adding the describe operator to the query, we are also able to unblock the group-by operator with punctuations on different ranges of minutes as well. When all minutes are covered by punctuations for a specific hour, the describe operator outputs a single punctuation for that hour, and the group-by operator is able to take advantage of the punctuation in the same way as above.

8.3.3 Comparison with Slack

The developers of the Aurora project [ACQ⁺03] define two kinds of query operators: *Order-agnostic* operators process data items as they arrive (e.g., union). *Order-sensitive*

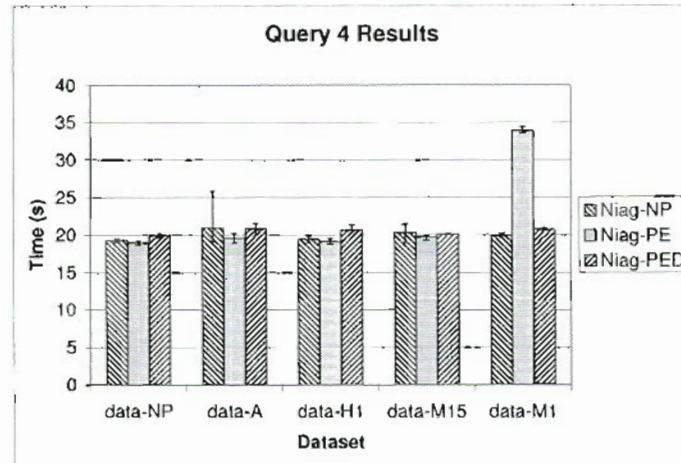


Figure 8.9: Results for Query 4. See Figure 8.6 for explanation of the legend.

operators must assume some kind of ordering in order to execute with bounded buffer space and in finite time. Group-by is an example of an order-sensitive operator. If the set of grouping attributes contains an ordering attribute, then group-by can output results before reaching the end of the stream. (Note that, in Aurora, group-by is included in the aggregate operator.) To account for disorder in input streams, order-sensitive operators have a *slack* parameter that users can set. Slack defines a bound on how disordered an input can become. (Note that data items that arrive out of order by more than the bound defined by slack are discarded.)

In our testing, Query 3 could be executed using order-sensitive versions of group-by. Query 3 groups data on hour and outputs the count of bids for each hour. Since data arrives ordered on hour, an order-sensitive implementation of group-by will output results for an hour when values for the next hour begin to arrive. However, when we have to union data items from multiple stream sources, as in Query 5, the result of that union has the potential to become disordered. Disorder can occur if the stream sources are not synchronized on time, or if the stream sources have different network delays to the system reading the streams.

Configuration	Stream	Size (bytes)	Punctuations
data-NP	1	18,303,854	0
	2	18,460,538	0
	3	18,456,289	0
	4	18,555,052	0
	5	18,419,441	0
data-H1	1	18,322,502	332
	2	18,479,186	332
	3	18,474,937	332
	4	18,573,700	332
	5	18,438,809	332

Table 8.5: Data set characteristics for Query 5. The Stream column lists each individual bid stream. See Table 8.2 for the meaning of other columns and configurations.

We executed tests using Query 5 to compare the behavior of queries using punctuations with queries using slack. We enhanced our group-by operator to be order-sensitive with tolerance for bounded disorder using a slack parameter (in addition to the enhancements discussed earlier for punctuations). The times for each stream source were set to slightly different values — compared to `bid1`, `bid2` was two minutes ahead, `bid3` was four minutes ahead, `bid4` was six minutes ahead, and `bid5` was eight minutes ahead.

We are interested in two different kinds of comparisons here between punctuations and slack. First, we want to know the *accuracy* of the results by comparing results from our tests with the actual values when executed over a terminating input (without punctuations). Second, we want to know the *expansion* of each result by comparing the percentage of data items that arrived after the first data item that contributed to a result when that result was output. An expansion of 100% indicates that the result for a group was output at the point when the final data item for that group arrived. An expansion greater than 100% indicates that more data items were processed than were required before results for a group were output, and an expansion less than 100% indicates that data items belonging to a group had not yet arrived when the result was output. To that end, we performed two comparisons using multiple values for slack. First, we compared the accuracy of results using slack with the expected results. Second, we compared the expansion of the group-by operator using slack with the expansion using punctuations.

Accuracy and expansion results are shown in Figure 8.11.

As can be seen from these results, there is a tradeoff of accuracy and expansion when using slack. The more accurate the results have to be, the more data items that must arrive before those results are output. Punctuations always report accurate results (assuming input streams are grammatical). Further, if punctuations are emitted as soon as they can be, then the number of data items that arrive that do not contribute to a group will be kept at a minimum for each group. In our testing, punctuations were emitted from stream sources immediately, so the expansion for all groups is 100%.

To get completely accurate results using slack, we must generally wait for more data items to arrive for the result to be output than with punctuations. Even when we set slack such that only two-thirds of the results are accurate, we still must wait for more data items on average before results are output than is necessary with punctuations. Punctuations adapt to bursts and lulls in the stream. Slack, however, is a fixed value for all groups, so the accuracy and expansion of the result value for each group is dependent on a single slack value. Note that, if punctuations embedded in the stream are not “tight” for a particular group, then the expansion for that group will increase. (By “tight”, we mean that each punctuation appears in the stream immediately following the end of the subset it matches.)

8.4 Summary

Punctuations embedded in data streams can be used to improve the behavior of many traditional query operators over data streams. However, embedding punctuations into a data stream requires extra stream bandwidth, and handling punctuations in operators requires extra processing overhead. We extended the online auction monitoring scenario discussed in Section 1.1.3 and evaluated five realistic queries for that scenario. Each query is tested using streams with varying amounts and kinds of punctuations, to determine the cost of processing punctuated streams. For these queries, we saw that punctuations do not significantly affect the overall performance of many kinds of queries. The performance of queries that do not require punctuations is not significantly hurt by reasonable ratios

of punctuations to data items. The behavior of queries that require blocking operators and unbounded stateful operators is improved using specific kinds of punctuations.

We have at this point considered how punctuations affect the performance of queries over data streams. In the next chapter, we consider another issue of punctuated streams and entire queries. That is, what kinds of punctuations can help a given query?

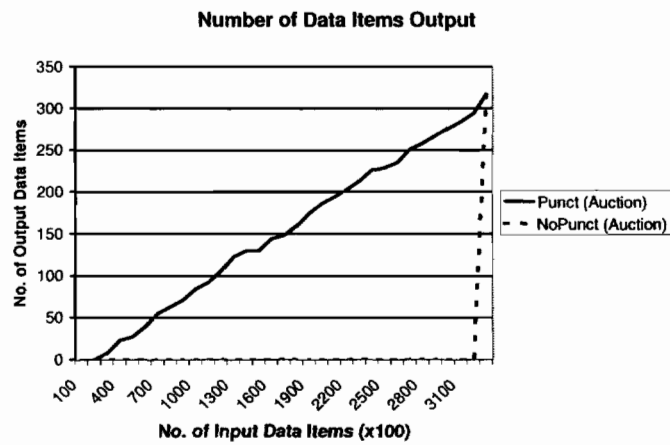
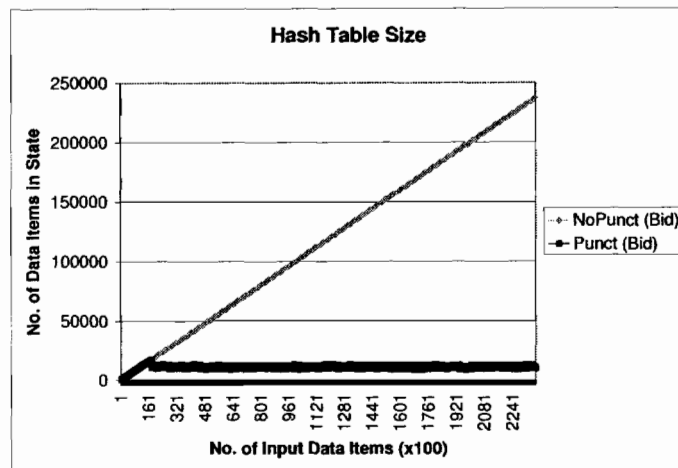


Figure 8.10: Number of bid data items stored in the hash table for the join operator and number of data items output as data items arrive, with and without punctuations on bids, for Query 4.

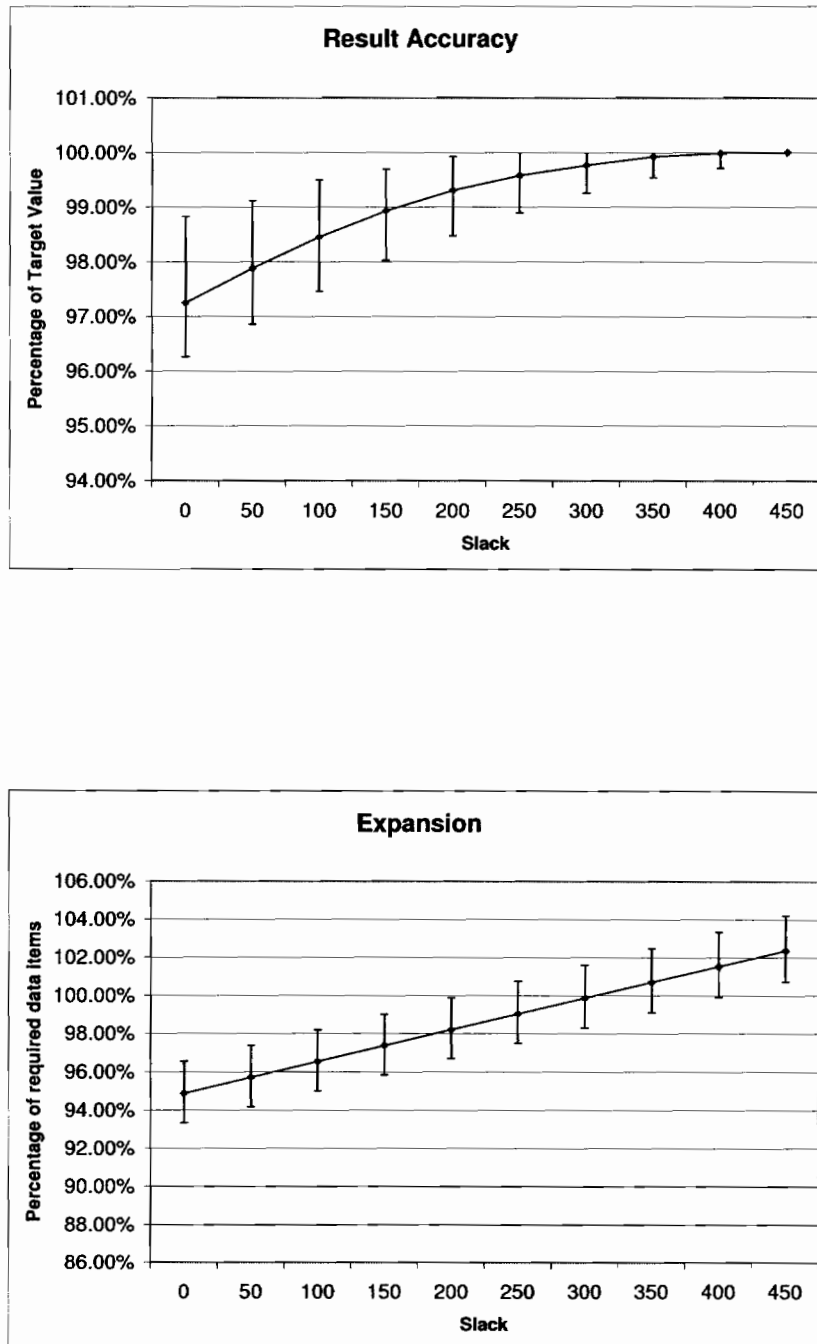


Figure 8.11: Accuracy and expansion of results for various values of slack. The bars indicate the range of accuracy for group results.

Chapter 9

Taking Punctuations Beyond Single Query Operators

To this point we have focused mainly on the behaviors of individual operators in the presence of punctuations. We know that punctuations can help various query operators, and we have seen how they can be used throughout execution of a query. In our example queries, we have used punctuations to produce results and reduce state but we have avoided an important question, namely, “How do we determine what kinds of punctuations will help a given query?”

Not all kinds of punctuations will help a given query. Indeed, there are queries that cannot be improved with any kind of punctuation. The warehouse example queries group data items on values of the hour attribute, and so have a natural *grouping of interest* based on the hour attribute. A grouping is a collection of groups in the data domain based on values of the grouping attribute. Many operators have natural groupings of interest. We will use groupings to decide if a particular set of punctuations will benefit a query. We will discuss groupings of interest in more detail in Section 9.2.

Punctuations that match all possible data items in a grouping of interest may benefit a query, but it must also be that the number of punctuations needed to match all data items for each specific group is finite. (Note that we do not expect that the total number of punctuations in a stream to be finite.) To illustrate, consider the network-traffic monitoring system from Section 1.1.2, and suppose that the incoming network stream of TCP/IP packets contains punctuations marking the end of packets for a specific destination port

for a particular hour. Figure 9.1 shows the data items that match each punctuation. Using this set of punctuations, we can match the group of packets from every destination port for each hour using a finite number of punctuations. Therefore, queries that group solely on the hour attribute will be able to output results without having to read the entire stream by taking advantage of the punctuations in the stream. However, we cannot match packets for every hour from each destination port with a finite number of punctuations, since there is no end for those subsets in the input stream. Therefore, queries that group on destination port will not be able to output results by taking advantage of this set of punctuations.

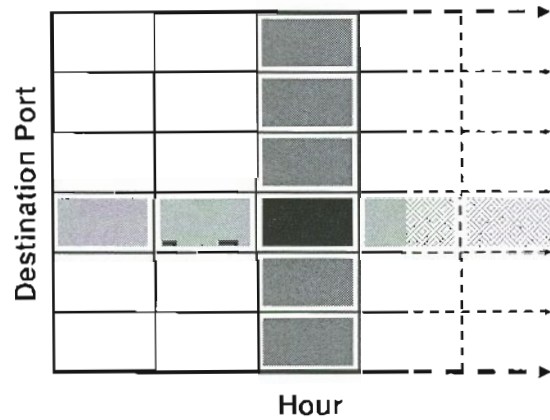


Figure 9.1: Data items that match punctuations for a specific destination port and hour. Each square in the grid represents all packets that have arrived from a particular source IP during a specific hour. The darker area, containing packets from all destination ports for a particular hour, can be covered by a finite number of punctuations. The lighter area, containing all packets from a specific destination port, cannot be covered by a finite number of punctuations.

That the number of input punctuations that match all data in a group must be finite suggests a notion similar to *compact sets* [Kas71, Rud64] from topology. We will use analogues to compactness to formally determine if a specific set of punctuations will benefit

a particular query.

9.1 Motivating Example

We will expand on the network monitoring system from Section 1.1.2 as a running example. The architecture for our monitoring system is shown in Figure 9.2. Two streams contain data to be processed by the monitoring system. The first stream (which we will refer to as “inbound”) contains network packets sent from other systems on the network to the server system, and the second stream (which we will refer to as “outbound”) contains network packets from the server system sent to other systems. Both streams contain structured data as defined in the IP RFC [Pos81a] and TCP RFC [Pos81b]. Table 9.1 lists the fields we will use in our discussion. Data from each stream flow through the network monitoring system unmodified, so that the destination system can process the packets. In addition, packets are passed through a network monitoring system. This system executes queries that track the performance of the system as well as check for possible network attacks against the server.

Name	Abbreviation	RFC
Source IP Address	<i>si</i>	IP
Destination IP Address	<i>di</i>	IP
Control Bits	<i>c</i>	IP
Fragment Offset	<i>fo</i>	IP
Source Port	<i>sp</i>	TCP
Destination Port	<i>dp</i>	TCP
Sequence Number	<i>sn</i>	TCP
Acknowledgment Number	<i>an</i>	TCP
Flags	<i>f</i>	TCP

Table 9.1: TCP/IP header fields, as defined in the IP RFC [Pos81a] and the TCP RFC [Pos81b]. Name is as given in the IP RFC and TCP RFC, Abbreviation is how we will refer to that field, and RFC lists which RFC each field is defined in.

For monitoring purposes, network packets are enhanced with a timestamp value *ts* in the network monitoring system for the benefit of queries. These timestamp values will be used in one of the performance monitoring queries defined in the next section.

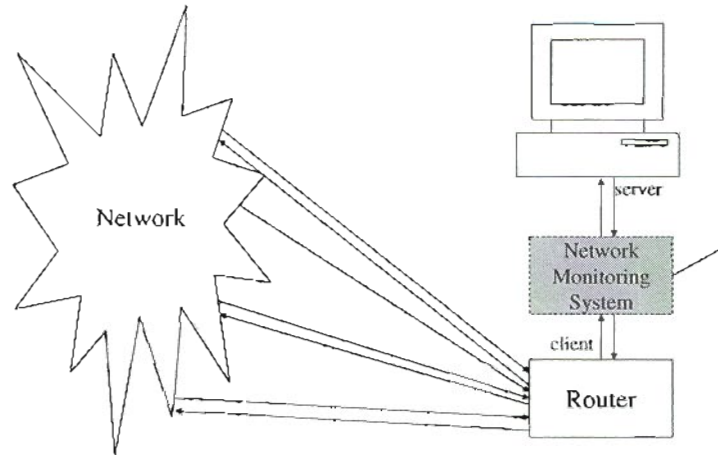


Figure 9.2: Architecture for the network monitoring system. Packets are streamed through the monitoring system (in gray), and results from monitoring queries are output to the network administrators.

9.1.1 Example Queries

Recall the query presented in Section 1.1.2. We want to check for a known network attack where more than one fragment for a packet contains the offset 0, possibly allowing a hacker to send data to a port blocked by the firewall [Coh96]. One query that can help check for this outputs the source IP address and the count of fragments where the offset is 0 when that count is greater than 1, as follows:

Query 9.1

```

SELECT sn, si, sp, COUNT(*) as NumOfZeroFrag
FROM inbound
WHERE fo = 0
GROUP BY sn, si, sp
HAVING COUNT(*) > 1;

```

This query cannot be executed over an unbounded data stream without help. Since it contains a blocking aggregation operator, results will not be output until the end of the input has been read.

Our second query, taken from a presentation by Spatscheck [CJS03], determines the response time for a server to return a SYN-ACK response to a client, per the “three-way handshake” described in the TCP RFC. The SQL for this query is:

Query 9.2

```
SELECT sn, si, sp, (O.ts - I.ts) AS response
FROM outbound O, inbound I
WHERE O.an=I.sn AND O.di=I.si AND O.dp=I.sp AND
      isSYN(I.c) AND isSYNACK(O.c)
```

We filter out all data items from the client that are not SYN requests and all data items from the server that are not SYN-ACK responses. We use the functions `isSYN()` and `isSYNACK()` to represent this filtering. Both functions simply check the flags of the packets for the appropriate values. We want to match up server responses with client requests based on the client’s initial sequence number, IP address, and port. This query uses join, which is an unbounded stateful operator. Therefore, when posed over an unbounded data stream, this query needs help to reduce the amount of state required during execution.

9.1.2 Solving the Example Queries with Punctuations

As discussed in Section 2.3, punctuations can be used to help Query 9.1. Each IP packet header contains information about whether or not it is the last fragment for the packet. The network monitor is enhanced such that, when the last fragment for a particular packet does arrive, the network monitor will emit punctuation following that packet fragment denoting the end of all fragments for that particular combination of sequence number and source IP address. The example query is grouping on those two attributes. When a punctuation arrives, all results that match that punctuation can be calculated and output, thus unblocking the aggregate operator.

Punctuations from both the client and server stream that mark the end of sequence numbers for particular IP addresses can also help Query 9.2. We can enhance the network

monitor to embed punctuation in the inbound stream following a SYN request that no more data items will arrive on the inbound stream containing the given sequence number and source IP address. Similarly, we can embed punctuation in the outbound stream following a SYN-ACK response that no more data items will be sent on the outbound stream containing the given acknowledgement number and destination IP address. As these are the attributes that make up the join, the state for join can be reduced due to punctuations.

In practice, the combination of sequence number and source IP address may not be unique throughout the entire network stream. According to the TCP RFC, the sequence number value at a TCP source will cycle approximately every 4.55 hours. The TCP RFC sets the maximum lifetime for a TCP packet (called the Maximum Segment Lifetime, or MSL) at two minutes, quite a bit less than the sequence number value cycle time. Therefore, it is correct to consider duplicate sequence number values from the same source IP as unique if they arrive sufficiently far apart in the stream. Thus, it is semantically acceptable to embed punctuations as described above.

9.2 Groupings and Punctuation Schemes

We want to be able to determine the utility of a given set of punctuations for a query. To this end, we introduce several new concepts. A *dataspace* represents the set of all possible data items that may appear on a stream, similar to a domain for a relation. For example, a simplified dataspace for the TCP stream might only contain source IP address (si), destination IP address (di), control bits (c), fragment offset (fo), source port (sp), destination port (dp), sequence number (sn), acknowledgement number (an), flags (f), and timestamp (ts). The dataspace for this simplified schema is:

$$D_T = \{ \langle si, di, c, fo, sp, dp, sn, an, f, ts \rangle \mid si \in [0, 2^{32} - 1], di \in [0, 2^{32} - 1], c \in [0, 2^6 - 1], \\ fo \in [0, 2^{13} - 1], sp \in [0, 2^{16} - 1], dp \in [0, 2^{16} - 1], sn \in [0, 2^{32} - 1], an \in [0, 2^{32} - 1], \\ f \in [0, 2^3 - 1], ts \in \mathbb{Z} \}.$$

We use the term *subspace* to mean any subset of a dataspace.

9.2.1 Groupings for Dataspace and Groupings of Interest

A *grouping* (which we will denote as $\mathbf{G}$) for a dataspace (or subspace) is a collection of subspaces on that dataspace (or subspace) where each subspace in the collection is made up of items that have equal values for specified attributes, and the union of the grouping equals the dataspace (or original subspace). Note that the specified attributes are the same for all subspaces (groups) in a grouping. Using a simple example, consider a dataspace of integer pairs, where $D = \{ \langle a, b \rangle \mid a \in \mathbb{Z}, b \in \mathbb{Z} \}$. A grouping can be formed on D using a as $\mathbf{G}_a = \{ \{ \langle a, b \rangle \mid b \in \mathbb{Z} \} \mid a \in \mathbb{Z} \}$, and another grouping can be formed on D using b as $\mathbf{G}_b = \{ \{ \langle a, b \rangle \mid a \in \mathbb{Z} \} \mid b \in \mathbb{Z} \}$. For the subspace $S = \{ \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 2 \rangle, \langle 2, 1 \rangle \}$, $\mathbf{G}_a = \{ \{ \langle 1, 1 \rangle, \langle 1, 2 \rangle \}, \{ \langle 2, 2 \rangle, \langle 2, 1 \rangle \} \}$, and $\mathbf{G}_b = \{ \{ \langle 1, 1 \rangle, \langle 2, 1 \rangle \}, \{ \langle 1, 2 \rangle, \langle 2, 2 \rangle \} \}$.

For a more complex example using D_T above, a grouping of D_T can be formed using values of sequence number (sn) as:

$$\mathbf{G}_{sn} = \left\{ \left\{ \langle si, di, c, fo, sp, dp, sn, an, f, ts \rangle \mid \begin{aligned} &si \in [0, 2^{32} - 1], di \in [0, 2^{32} - 1], \\ &c \in [0, 2^6 - 1], fo \in [0, 2^{13} - 1], sp \in [0, 2^{16} - 1], dp \in [0, 2^{16} - 1], \\ &an \in [0, 2^{32} - 1], f \in [0, 2^3 - 1], ts \in \mathbb{Z} \end{aligned} \right. \right. \\ \left. \mid sn \in [0, 2^{32} - 1] \right\}$$

For the subspace $S = \{ \langle 1, 1, 0, 0, 50, 80, 15, 0, 0, 125 \rangle, \langle 1, 1, 0, 0, 50, 80, 25, 0, 0, 131 \rangle, \langle 1, 1, 0, 0, 70, 21, 15, 0, 0, 132 \rangle \}$, the grouping on sn would be (sn values in **bold**):

$$\mathbf{G}_S = \left\{ \left\{ \langle 1, 1, 0, 0, 50, 80, \mathbf{15}, 0, 0, 125 \rangle, \langle 1, 1, 0, 0, 70, 21, \mathbf{15}, 0, 0, 132 \rangle \right\}, \right. \\ \left. \left\{ \langle 1, 1, 0, 0, 50, 80, \mathbf{25}, 0, 0, 131 \rangle \right\} \right\}$$

A *grouping of interest* for a query operator is a grouping that arises naturally from the definition or implementation of the operator. Many query operators have groupings of interest. For example, an aggregate operator's grouping of interest is the one defined by values of the group-by attributes. The join operator has two groupings of interest, one for each input, based on the join attributes. We will discuss groupings of interest for specific operators in Section 9.4.1.

For attributes defined over a totally ordered set, we enhance our definition of a grouping where subspaces in the grouping are defined over ranges of attribute values, rather than

specific attribute values. Note that, with this enhancement, subspaces in a grouping may overlap. For example, we could define a grouping over D_T where each subspace in the grouping is a prefix of timestamp values (ts). That is:

$$\mathbf{G}'_{sn} = \left\{ \begin{array}{l} \{ \langle si, di, c, fo, sp, dp, sn, an, f, ts' \rangle \mid si \in [0, 2^{32} - 1], di \in [0, 2^{32} - 1], \\ c \in [0, 2^6 - 1], fo \in [0, 2^{13} - 1], sp \in [0, 2^{16} - 1], dp \in [0, 2^{16} - 1], sn \in [0, 2^{32} - 1] \\ an \in [0, 2^{32} - 1], f \in [0, 2^3 - 1], ts' \in \mathbb{Z} \wedge ts' \leq ts \} \\ \mid ts \in \mathbb{Z} \end{array} \right\}.$$

This enhanced definition for groupings will be useful when we discuss groupings of interest for certain operators, particularly sort.

9.2.2 Punctuation Schemes

Recall from Section 4.5, the set of valid punctuation patterns over a set A is denoted as:

$$\Pi_A = \{*\} \cup \{\epsilon\} \cup \{a_1 \mid a_1 \in A\} \cup \{A' \mid A' \subseteq A \wedge |A'| \text{ is finite}\}$$

Further, if A is a totally ordered set, then the set of valid patterns is enlarged as:

$$\begin{aligned} \Pi_A = \{*\} \cup \{\epsilon\} \cup \{a_1 \mid a_1 \in A\} \cup \{A' \mid A' \subseteq A \wedge |A'| \text{ is finite}\} \cup \\ \{(a_1, a_2) \mid a_1, a_2 \in A \wedge a_1 \leq a_2\} \cup \{(\perp, a_2) \mid a_2 \in A\} \cup \{(a_1, \top) \mid a_1 \in A\} \end{aligned}$$

As may be expected, the range pattern could be inclusive or exclusive on either side of the range, using parentheses or square brackets as usual.

For dataspace D (over $A_1 \times A_2 \times \dots \times A_n$), the *punctuation space* $\mathbb{P}_D = \Pi_{A_1} \times \Pi_{A_2} \times \dots \times \Pi_{A_n}$ is the set of possible punctuations that can be defined for D . A *punctuation scheme* $\mathcal{P}_D$ for D is the set of punctuations that will be emitted from a stream source¹. Clearly, $\mathcal{P}_D \subseteq \mathbb{P}_D$. We will specify particular punctuation schemes using set notation. In Section 9.2 we defined a simplified dataspace for TCP packets as D_T . Suppose the router is enhanced to embed punctuations in the client stream at the end of each sequence

¹Note that every punctuation in $\mathcal{P}_D$ is guaranteed to eventually appear. Punctuations that will not appear on the data stream are not in $\mathcal{P}_D$. However, it is possible that punctuations not in $\mathcal{P}_D$ may also appear on the data stream.

number (sn) for each source address (si). Then the punctuation scheme for the source is $\mathcal{P}_{D_T} = \{ \langle si, *, *, *, *, *, sn, *, *, * \rangle | si \in [0, 2^{32} - 1], sn \in [0, 2^{32} - 1] \}$.

Given a grouping of interest for a particular query operator, we want to know when all data items for one or more members of that grouping have arrived. We will use punctuations to determine this condition. The *interpretation* of a punctuation p is the subspace of data items that match that punctuation, denoted $\mathcal{I}(p)$. Formally, given a dataspace D and a punctuation $p \in \mathbb{P}_D$, $\mathcal{I}(p) = \{d | d \in D \wedge match(d, p)\}$. For a given punctuation scheme $\mathcal{P}_D$, $\mathbf{S}_{\mathcal{P}_D} = \{\mathcal{I}(p) | p \in \mathcal{P}_D\}$ is the collection of interpretations for punctuations in $\mathcal{P}_D$. When D is understood from context, we will write $\mathcal{P}$ for $\mathcal{P}_D$ and $\mathbf{S}_{\mathcal{P}}$ for $\mathbf{S}_{\mathcal{P}_D}$.

A punctuation scheme $\mathcal{P}$ is said to be *complete* for D if, for every data item $d \in D$, there exists a punctuation $p \in \mathcal{P}$ such that $match(d, p) = true$. Equivalently, $\mathcal{P}$ is complete for D if for all $d \in D$, there exists $G \in \mathbf{S}_{\mathcal{P}}$ such that $d \in G$. Generally, if a punctuation scheme is not complete, then blocking operators cannot be completely unblocked. The punctuation scheme $\mathcal{P}' = \{ \langle si, *, *, *, *, *, sn, *, *, * \rangle | si \in [0, 2^{32} - 1], sn \in [0, 2^{32} - 1] \wedge sn \% 2 = 0 \}$ emits punctuations that match data items for every even source port value, and so is not complete. The punctuation scheme $\mathcal{P}_{D_T}$ given earlier is complete.

For a given query operator, we will want a complete punctuation scheme where the collection of interpretations of that scheme forms a cover for the grouping of interest for that operator. For example, the group-by operator in Query 9.1 groups on values of the sequence number and source IP address attributes. The collection of interpretations of the punctuation scheme $\mathcal{P}_{D_T}$ forms a cover for the group-by operator. We will show in the next sections that this kind of punctuation scheme can “help” particular queries.

9.2.3 Overview of Our Approach

A query is said to *benefit* from its input punctuation schemes if the following two conditions hold:

Enables: All result data items for a query will eventually be output.

Cleanses: Every data item that resides in state for any operator in the query will eventually be removed. Note that it may not be the case that state will be completely empty, only that each data item that exists in state will eventually be removed.

By *eventually*, we mean that, for some non-terminating stream, there exists some finite time j where the event will occur (e.g. outputting results). Note that, even if a set of punctuation schemes cleanses a query, the state may be non-empty throughout execution of the query².

Our goal is to determine whether the punctuation schemes emitted by stream sources benefit all of the operators in a query. Given a query tree for a query and the punctuation schemes for each stream source, we can use propagation invariants for operators in the query tree to determine the output punctuation scheme for each operator. If the root operator in the tree emits a complete punctuation scheme, we know that the query is enabled, and all result data items will eventually be output.

We will use a concept with an analog in topology to help us determine whether a given set of punctuation schemes benefits a query. A collection $\mathbf{S}$ forms a *cover* for a set T if $\bigcup \mathbf{S} = T$. Further, T is *compact* relative to S if every cover for T contains a finite subcollection that is also a cover for T . Given a group G in a grouping $\mathbf{G}$ over dataspace D and a punctuation scheme $\mathcal{P}_D$, $\mathcal{P}_D$ is said to be a *cover* for G if every data item in G matches some punctuation in $\mathcal{P}_D$. G is said to be *compact* relative to $\mathcal{P}_D$ if there exists some finite subset of punctuations P' in $\mathcal{P}_D$ such that P' forms a cover for G . (Note these are slightly different notions. The second does not require finite subcovers of every cover, on that some finite cover exists.) If all groups in $\mathbf{G}$ are compact relative to $\mathcal{P}_D$, we say that the grouping $\mathbf{G}$ is compact relative to $\mathcal{P}_D$.

Further, we will define a function *preimage* for each stream operator that takes a punctuation in the output punctuation space as an argument and returns a subspace of the input. For binary operators, *preimage* returns a pair of subspaces, one for each input dataspace. Our intention is, given a punctuation p , the function *preimage* will return

²Other techniques, such as querying over data with an assumed order (e.g. merge join), may also be shown to benefit a query. Investigating how other techniques may benefit certain kinds of queries is an area for future work.

all values in $\mathcal{I}(p)$. Given a query, we calculate the *preimage* for every punctuation p in the output punctuation scheme for each operator. Then, given a set of input punctuation schemes, we determine if that set will benefit the query by checking that the subspace output by *preimage* is compact relative to the input punctuation scheme(s) for that operator for all p in the output punctuation scheme.

9.3 Unblocking Query Operators with Punctuation Schemes

We want to analyze the effects of given input punctuation schemes for a given query. In this section, we will focus on enabling individual query operators with punctuation schemes. In the next section, we will address cleansing individual operators using punctuation schemes, and finally in Section 9.5 we will discuss how to determine if a given query benefits from particular punctuation schemes.

It is awkward to reason about a query or subquery emitting all possible data items eventually. Specifically, consider a data item d_o in the output domain of a given query. Suppose at some point during execution, d_o has not been output. We need some way of determining whether or not d_o will ever be output. So, we focus on the output punctuation schemes of each operator in the query. If an operator correctly emits a punctuation, then we can be sure that any data item in the output domain that has not been output and matches that punctuation will never be output. For d_o , if it matches some punctuation that has arrived and has not been output, then we know d_o will never be output. Thus, we need to know what kinds of punctuations will be emitted by an operator. If an operator emits a complete punctuation scheme, we know that it will emit all possible result data items.

Note that we are making an assumption about stream operators. A stream operator could just emit the punctuations in its output scheme, and no data items. The output is grammatical, but clearly the operator is not unblocked. In this work we assume that query operators are *faithful* and *proper* counterparts to the traditional table-based relational operators as defined in Section 6.1. That is, we assume that all data items that can be output due to punctuation will be output before the punctuation is output.

9.3.1 Punctuation Scheme Assignments

Let us define a *scheme assignment* $\mathcal{A}$ to be a mapping from each operator O in a query tree to the input and output punctuation schemes. That is, for unary operator O_u , $\mathcal{A}[O_u] = (\{\mathcal{P}_I[O]\}, \mathcal{P}_R[O])$. The first item in the mapping ($\mathcal{P}_I[O]$) is the input punctuation scheme. The second item ($\mathcal{P}_R[O]$) is the output punctuation scheme. For binary operator O_b , $\mathcal{A}[O_b] = (\{\mathcal{P}_I^1[O_b], \mathcal{P}_I^2[O_b]\}, \mathcal{P}_R[O_b])$. Note that the output punctuation scheme in an operator's scheme assignment will be a member of the input punctuation scheme for the parent of that operator in the query tree. We want to determine if a punctuation in the output punctuation scheme will be emitted due to punctuations arriving from given input punctuation schemes. Given a scheme assignment for all operators in the tree, then the problem reduces to checking that, for each operator O in the tree, each p in the output punctuation scheme can be emitted given some finite subset from each of the input punctuation schemes for O . The ultimate goal is to find punctuation schemes for each input stream to the query as a whole that allows the root operator in the query to emit all punctuations in some complete punctuation scheme.

Recall in Query 9.1, we output sequence number, source IP address, source port, and count of fragments where the fragment offset is 0 for all suspect packets. Therefore, the output dataspace for Query 9.1 is $D_o = \{ \langle sn, si, sp, cnt \rangle \mid sn \in [0, 2^{32} - 1], si \in [0, 2^{32} - 1], sp \in [0, 2^{16} - 1], cnt \in \mathbb{N} \}$, and we want the root operator in the query to output a punctuation scheme that is complete for D_o . Suppose the network monitor is enhanced to emit punctuations in the punctuation scheme $\mathcal{P}_{D_T} = \{ \langle si, *, *, *, sn, *, *, * \rangle \mid si \in [0, 2^{32} - 1], sn \in [0, 2^{32} - 1] \}$ defined earlier. We will use the following relational algebra formula for Query 9.1: $\sigma_{COUNT(*) > 1}(\mathcal{G}_{sn, si, sp}^{COUNT(*)}(\sigma_{fo=0}(inbound)))$. The query plan is shown in Figure 9.3.

We can use the propagation invariants for select and group-by to determine appropriate scheme assignments for the operators in this query. Let $\mathcal{P}_R[\sigma_{COUNT(*) > 1}] = \{ \langle sn, si, *, * \rangle \mid sn \in [0, 2^{32} - 1], si \in [0, 2^{32} - 1] \}$. Clearly, $\mathcal{P}_R[\sigma_{COUNT(*) > 1}]$ is complete for D_o . Let us set the scheme assignments for this query as: $\mathcal{A}[\sigma_{fo=0}] = (\{\mathcal{P}_{D_T}\}, \mathcal{P}_{D_T})$, $\mathcal{A}[\mathcal{G}_{sn, si, sp}^{COUNT(*)}] = (\{\mathcal{P}_{D_T}\}, \mathcal{P}_R[\sigma_{COUNT(*) > 1}])$, and $\mathcal{A}[\sigma_{COUNT(*) > 1}] =$

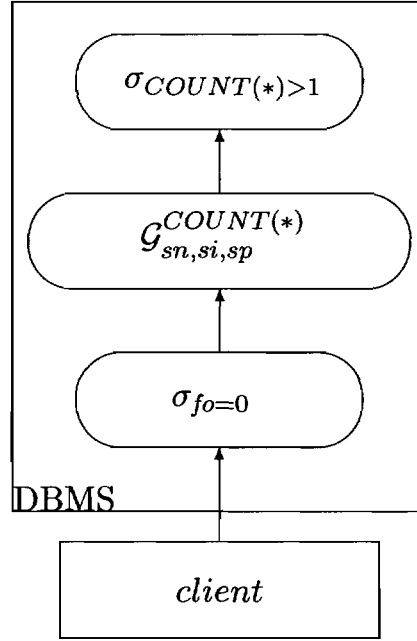


Figure 9.3: A possible query plan for Query 9.1.

$(\{\mathcal{P}_R[\sigma_{COUNT(*)>1}]\}, \mathcal{P}_R[\sigma_{COUNT(*)>1}])$. Therefore, the output punctuation scheme for this query is $\mathcal{P}_R[\sigma_{COUNT(*)>1}]$. If we could show that each operator in this query produces its output punctuation scheme given its input punctuation scheme, we would know the query is unblocked. So the ultimate question is, “How do we decide if a query can produce a given output scheme?” We will address this question in the next section. We will later turn to the question of, “How can we come up with an appropriate scheme assignment initially?”

9.3.2 Unblocking Unary Query Operators

We want to first show the kinds of punctuation schemes that unblock unary operators. For a unary operator O , we will denote the input punctuation scheme as $\mathcal{P}_I[O]$ and the output punctuation scheme as $\mathcal{P}_R[O]$. (We address binary operators in Section 9.3.4.) We say that an input punctuation scheme $\mathcal{P}_I[O]$ *enables* an output punctuation $p \in \mathcal{P}_R[O]$ if p can be emitted after a finite number of punctuations in $\mathcal{P}_I[O]$ have arrived. Further, an

input punctuation scheme $\mathcal{P}_I[O]$ *enables* $\mathcal{P}_R[O]$ if, for every $p \in \mathcal{P}_R[O]$, $\mathcal{P}_I[O]$ enables p . If $\mathcal{P}_I[O]$ enables $\mathcal{P}_R[O]$ for a given operator, then we know that all result data items that exist in interpretations of $\mathcal{P}_R[O]$ will be output, and therefore O is enabled.

It appears logical to use the definitions of compactness and enables to derive the following statement: Given $\mathcal{P}_I[O]$ and $\mathcal{P}_R[O]$ as input and output punctuation schemes for an operator O , let $\mathbf{S}_{\mathcal{P}_R[O]}$ be the collection of interpretations of $\mathcal{P}_R[O]$ (that is, $\mathbf{S}_{\mathcal{P}_R[O]} = \{\mathcal{I}(p) | p \in \mathcal{P}_R[O]\}$). If $\mathbf{S}_{\mathcal{P}_R[O]}$ is compact relative to $\mathcal{P}_I[O]$, then $\mathcal{P}_I[O]$ enables $\mathcal{P}_R[O]$. Unfortunately, such a statement is too simple. The output schema for many query operators is different than the input schema. Project and group-by, for example, generally have a different output schema from their input schema. The statement above also does not address binary operators (which we will discuss in Section 9.3.4).

We need a map from a punctuation p in $\mathcal{P}_R[O]$ to a subspace T of the input dataspace D_I such that if T is compact relative to $\mathcal{P}_I[O]$, then p can be emitted. We call this map *preimage*, defined for specific unary operators in Table 9.2. Intuitively, for some punctuation $p \in \mathcal{P}_R[O]$, $\text{preimage}[O](p)$ tells us what subspace must be covered by input punctuations before an operator can safely emit p . Put another way, $\text{preimage}[O](p)$ gives us that subset of data items in the input dataspace that contributes to output data items that match p .

select	$\text{preimage}[\sigma](p) =$	$\mathcal{I}(p)$
dupelim	$\text{preimage}[\delta](p) =$	$\mathcal{I}(p)$
project	$\text{preimage}[\pi_A](p) =$	$\mathcal{I}(p(A) : *)$
group-by	$\text{preimage}[\mathcal{G}_A^F](p) =$	$\mathcal{I}(p(A) : *)$
sort	$\text{preimage}[\mathcal{S}_A](p) =$	let $a = \text{primary}(A)$ in $\mathcal{I}((\perp, \max(p.a)) : *)$ where $\text{primary}(A)$ returns the primary sorting attribute

Table 9.2: Definitions for *preimage* for unary operators for punctuations $p \in \mathcal{P}_R[O]$. We use the notation “: *” to mean * values for each attribute of the schema not already listed. The function *max* returns the maximum of the given pattern value. For example, given a range pattern value, the maximum range value is returned. Given a list of pattern values, *max* returns the maximum pattern value in the list.

There are definitions for *preimage* other than those defined in Table 9.2 for various operators. For example, for some predicate q , we could also define $\text{preimage}[\sigma_q](p)$ as

$\sigma_q(\mathcal{I}(p))$. That is, *preimage* returns only those data items from the input that pass the predicate. Whichever definition of *preimage* we choose for a given operator, it is important that Theorem 9.1 (given in the next section) holds.

9.3.3 Enabling Punctuations From Unary Operators Using Input Punctuation Schemes

We want to be able to determine if an input punctuation scheme enables an output punctuation scheme for a unary operator. First, we give a theorem specifying when a given punctuation in the output punctuation scheme can be emitted by an operator. We will use that result to show when an output punctuation scheme is enabled by an input punctuation scheme.

Theorem 9.1 *For a unary operator O , given a grammatical input stream S , an input punctuation scheme $\mathcal{P}_I[O]$, and an output punctuation scheme $\mathcal{P}_R[O]$, $p_r \in \mathcal{P}_R[O]$ can be emitted if $\text{preimage}[O](p_r)$ is compact relative to $\mathcal{P}_I[O]$.*

Proof (for *preimage* defined in Table 9.2):

For each unary operator O , suppose $\text{preimage}[O](p_r)$ is compact relative to $\mathcal{P}_I[O]$. Therefore, there exists a finite $P \subseteq \mathcal{P}_I[O]$ that is a cover for $\text{preimage}[O](p_r)$. As P is finite, all punctuations in P will eventually arrive. Consider a point after all punctuations in P have arrived. Let $\mathbf{S}_P = \{\mathcal{I}(p) | p \in P\}$. Since the input stream S is grammatical, it must also be that any data items contained in $S \cap (\bigcup \mathbf{S}_P)$ have also arrived. For the proof below, let $\hat{\mathbf{S}} = \bigcup \mathbf{S}_P$.

At this point, we prove that p_r can be emitted for each individual operator using case analysis. We will refer to the pass invariants for various operators here. We list formal definitions for pass invariants for various operators in Table 9.3 for reference.

Case select (σ): By the pass invariant for select, data items that have arrived and pass the predicate for select will be output, and those that do not pass the predicate will never be output. Therefore, all data items in $S \cap (\bigcup \mathbf{S}_P)$ that pass the predicate have been output. Since P covers $\text{preimage}[\sigma](p_r)$, all data items in $\mathcal{I}(p_r)$ are also

Op	Pass Invariant
Select (σ)	$\sigma(T_I)$
Dupelim (δ)	$\delta(T_I)$
Project (π_A)	$\pi_A(T_I)$
Group-by ($\mathcal{G}_A^F$)	$\{t :: \langle f_i(U_i) \rangle t \in \text{setMatchTs}(\pi_A(T_I), \text{groupP}(A, P_I)) \wedge f_i \in F \wedge U_i = \{u u \in T_I \wedge t[A] = u[A]\}\}$
Sort ($\mathcal{S}_A$)	$\text{setMatchTs}(T_1, \text{init}(\text{Sort}_A, P_1))$
Union ($\cup$)	$T_I^1 \cup T_I^2$
Intersect ($\cap$)	$T_I^1 \cap T_I^2$
Join ($\bowtie$)	$T_I^1 \bowtie T_I^2$
Difference ($-$)	$\text{setMatchTs}((T_I^1 - T_I^2), P_I^2)$

Table 9.3: Pass invariants for stream operators. D_I is the input domain, sets T_I and P_I represent the input data items and punctuations, respectively, that have arrived so far. For binary operators, superscripts refer to the first and second input streams.

contained in $\widehat{S}$ (by the definition of $\widehat{S}$, and therefore all data items in $\mathcal{I}(p_r) \cap S$ that pass the predicate have been output. Thus, p_r can also be output.

Case duplicate elimination (δ): By the pass invariant for duplicate elimination, exactly one element of each data item that has arrived can be output. Therefore, one copy of each data item in $S \cap (\bigcup \mathbf{S}_P)$ has been output. Since P covers $\text{preimage}[\delta](p_r)$, all data items in $S \cap \mathcal{I}(p_r)$ are also contained in $\widehat{S}$, and have been output. Thus, p_r can also be output.

Case project (π_A): By the pass invariant for project (duplicate-preserving), the projection on A of any data item that has arrived is output. Therefore, the projection on A of any data item in $S \cap \widehat{S}$ has been output. Since P covers $\text{preimage}[\pi_A](p_r)$, we know that all data items contained in $\mathcal{I}(p_r(A) : *)$ have arrived, and their projection on A has therefore been output. Thus, p_r can also be output.

Case group-by ($\mathcal{G}_A^F$): The pass invariant for group-by allows results for a group to be output when enough punctuations have arrived stating that all data items that will contribute to that group have arrived. We use the function groupP to get only those punctuations that indicate all data items for particular values of the projected attributes have arrived, and then emit punctuations with only patterns for the projected attributes. Since P is a cover for $\text{preimage}[\mathcal{G}_A^F](p_r)$, we know that

all data items in $S \cap \widehat{\mathbf{S}}$ have arrived, hence $S \cap (\mathcal{I}(p_r(A) : *))$ have arrived. Therefore, all data items belonging to groups generated using values of A that match $p_r(A) : *$ have arrived, and those results can be calculated. By the pass invariant for group-by, results for those groups have been output. Therefore, p_r can also be output.

Case sort (S_A): By the pass invariant for sort, in order for some data item d to be output from sort, enough punctuations must have arrived stating that all data items in the prefix of the output up to d that will arrive have arrived. Since P is a cover for $\text{preimage}[S_A](p_r)$, we know that all data items in $S \cap \widehat{\mathbf{S}}$ have arrived, hence $S \cap (\mathcal{I}((\perp, \max(p_r.a)] : *))$ has arrived (where $a = \text{primary}(A)$). Thus, it must be that d has arrived, and that all data items in the prefix of the output before d have also arrived. By the pass invariant for sort, all data items in the prefix up to and including d have been output. Since no more data items will be output that would match p_r , p_r can also be output.

End of proof.

We use this result to show the required properties of input punctuation schemes that will enable an output punctuation scheme in the following theorem.

Theorem 9.2 *For a unary operator O , let $\mathbf{S}_g = \{\text{preimage}[O](p_r) \mid p_r \in \mathcal{P}_R[O]\}$. If $\mathbf{S}_g$ is compact relative to $\mathcal{P}_I[O]$, then $\mathcal{P}_I[O]$ enables $\mathcal{P}_R[O]$.*

Proof:

Suppose $\mathbf{S}_g$ is compact relative to $\mathcal{P}_I[O]$. Then:

$$\begin{aligned}
 & \forall G \in \mathbf{S}_g, G \text{ is compact relative to } \mathcal{P}_I[O] \\
 & \Rightarrow \forall G \in \{\text{preimage}[O](p_r) \mid p_r \in \mathcal{P}_R[O]\}, G \text{ is compact relative to } \mathcal{P}_I[O] \\
 & \Rightarrow \forall p_r \in \mathcal{P}_R[O], \text{preimage}[O](p_r) \text{ is compact relative to } \mathcal{P}_I[O] \\
 & \Rightarrow [\text{by Theorem 9.1}] \forall p_r \in \mathcal{P}_R[O], p_r \text{ will be emitted} \\
 & \Rightarrow \forall p_r \in \mathcal{P}_R[O], \mathcal{P}_I[O] \text{ enables } p_r \\
 & \Rightarrow \mathcal{P}_I[O] \text{ enables } \mathcal{P}_R[O]
 \end{aligned}$$

End of proof.

That is, if the collection of preimages for each punctuation in the output punctuation scheme is compact relative to the input punctuation scheme, then the output punctuation scheme is enabled by the input punctuation scheme. In this case we know that all punctuations in the output punctuation scheme will be emitted, and therefore, all result data items will eventually be output.

9.3.4 Unblocking Binary Query Operators

Now we consider unblocking binary operators. A punctuation can be output from a binary operator O only when O has received enough information from both of its inputs concerning the data items that have arrived. We first need a pairwise version for the definition of enable. Throughout this chapter, we will use superscript to denote each input to a binary operator. For example, we will write $\mathcal{P}_I^1[O]$ and $\mathcal{P}_I^2[O]$ for each input punctuation scheme to a binary operator O . For an output punctuation scheme $\mathcal{P}_R[O]$, we say that $\mathcal{P}_I^1[O]$ and $\mathcal{P}_I^2[O]$ *pairwise enable* an output punctuation $p_r \in \mathcal{P}_R[O]$ if p_r can be emitted after a finite number of punctuations in $\mathcal{P}_I^1[O]$ and $\mathcal{P}_I^2[O]$ have arrived. Further, input punctuation schemes $\mathcal{P}_I^1[O]$ and $\mathcal{P}_I^2[O]$ *pairwise enable* $\mathcal{P}_R[O]$ if, for all $p_r \in \mathcal{P}_R[O]$, $\mathcal{P}_I^1[O]$ and $\mathcal{P}_I^2[O]$ pairwise enable p_r .

We also need compactness over pairs. First, given collections $\mathbf{S}^1$ and $\mathbf{S}^2$, we say that the pair $(\mathbf{S}^1, \mathbf{S}^2)$ forms a *pairwise cover* for the pair of sets (T^1, T^2) if each $\mathbf{S}^i$ is a cover for T^i . Further, (T^1, T^2) are *pairwise compact* relative to $(\mathbf{S}^1, \mathbf{S}^2)$ if each T^i is compact relative to $\mathbf{S}^i$. As we did with cover and compact, we extend our definitions of pairwise cover and pairwise compact to include punctuation schemes. Given a group G^1 in a grouping over dataspace D^1 , a group G^2 in a grouping over dataspace D^2 , and punctuation schemes $\mathcal{P}_{D^1}$ and $\mathcal{P}_{D^2}$, $(\mathcal{P}_{D^1}, \mathcal{P}_{D^2})$ is a *pairwise cover* for (G^1, G^2) if each $\mathcal{P}_{D^i}$ is a cover for G^i . The pair (G^1, G^2) is *pairwise compact* relative to $(\mathcal{P}_{D^1}, \mathcal{P}_{D^2})$ if each G^i is compact relative to $\mathcal{P}_{D^i}$. Finally, if all pairs of groups from a pair of groupings are pairwise compact relative to $(\mathcal{P}_{D^1}, \mathcal{P}_{D^2})$, then the pair of groupings is pairwise compact relative to $(\mathcal{P}_{D^1}, \mathcal{P}_{D^2})$.

For some binary operator O , $\text{preimage}[O](p)$ must return two subspaces, one for each input. For $p_r \in \mathcal{P}_R[O]$, $\text{preimage}(p_r)$ must be pairwise compact relative to the interpretations of punctuations from each input before p_r can be output. Table 9.4 lists our

definitions of $preimage[O]$ for several binary stream operators.

union	$preimage[\cup](p) = (\mathcal{I}(p), \mathcal{I}(p))$
intersect	$preimage[\cap](p) = (\mathcal{I}(p), \mathcal{I}(p))$
difference	$preimage[-](p) = (\mathcal{I}(p), \mathcal{I}(p))$
equi-join	$preimage[\bowtie](p) = (\mathcal{I}(p(A) : p(B)), \mathcal{I}(p(B) : p(C)))$ where the input schemas are $A \cup B$ and $B \cup C$

Table 9.4: Definitions for $preimage[O]$ for binary operators for punctuations $p \in \mathcal{P}_R[O]$.

9.3.5 Enabling Punctuations From Binary Operators Using Input Punctuation Schemes

There are analogues of Theorems 9.1 and 9.2 for binary operators, as follows:

Theorem 9.3 *For a binary operator O , given grammatical inputs S^1 and S^2 and an output punctuation scheme $\mathcal{P}_R[O]$, $p_r \in \mathcal{P}_R[O]$ can be emitted if $preimage[O](p_r)$ is pairwise compact relative to the pair of input punctuation schemes $(\mathcal{P}_I^1[O], \mathcal{P}_I^2[O])$.*

Proof (for $preimage$ defined in Table 9.4):

For each binary operator O , suppose S^1 and S^2 are grammatical and for $p_r \in \mathcal{P}_R[O]$, $preimage[O](p_r)$ is pairwise compact relative to $(\mathcal{P}_I^1[O], \mathcal{P}_I^2[O])$. Therefore, there exist finite subsets $P^1 \subseteq \mathcal{P}_I^1[O]$ and $P^2 \subseteq \mathcal{P}_I^2[O]$ such that (P^1, P^2) is a pairwise cover for $preimage[O](p_r)$. As P^1 and P^2 are finite, all punctuations in P^1 and P^2 will eventually arrive. Consider a point after all punctuations in P^1 and P^2 have arrived from S^1 and S^2 , respectively. Let $\mathbf{S}_{P^1} = \{\mathcal{I}(p) | p \in P^1\}$ and $\mathbf{S}_{P^2} = \{\mathcal{I}(p) | p \in P^2\}$. Further, let $(T^1, T^2) = preimage[O](p_r)$. Since the input streams S^1 and S^2 are grammatical, it must also be that all data items contained in $S^1 \cap (\bigcup \mathbf{S}_{P^1})$ have arrived, and all data items contained in $S^2 \cap (\bigcup \mathbf{S}_{P^2})$ have arrived. Similar to before, let $\widehat{\mathbf{S}}^1 = \bigcup \mathbf{S}_{P^1}$ and $\widehat{\mathbf{S}}^2 = \bigcup \mathbf{S}_{P^2}$.

At this point we must prove that p_r can be emitted for each individual operator using case analysis.

Case union ($\cup$): By the pass invariant for union (duplicate preserving), data items that have arrived from either input have been output. Therefore, all data items contained in $S^1 \cap \widehat{\mathbf{S}}^1$ have been output and all data items contained in $S^2 \cap \widehat{\mathbf{S}}^2$ have been output.

Since (P^1, P^2) pairwise cover $\text{preimage}[\cup](p_r)$ all data items in T^1 are also in $\widehat{\mathbf{S}}^1$, and all data items in T^2 are also in $\widehat{\mathbf{S}}^2$, and therefore all data items in $S^1 \cap T^1$ and $S^2 \cap T^2$ have arrived and been output. Thus, p_r can also be output.

Case intersect ($\cap$): By the pass invariant for intersect, data items that have arrived on both inputs can be output. Therefore, all data items in $(S^1 \cap \widehat{\mathbf{S}}^1) \cap (S^2 \cap \widehat{\mathbf{S}}^2)$ have been output. Since (P^1, P^2) pairwise cover $\text{preimage}[\cap](p_r)$, all data items in T^1 are also in $\widehat{\mathbf{S}}^1$, and all data items in T^2 are also in $\widehat{\mathbf{S}}^2$, and therefore all data items in $(S^1 \cap T^1) \cap (S^2 \cap T^2)$ have been output. Thus, p_r can also be output.

Case difference ($-$): By the pass invariant for difference, all data items in $\text{setMatchTs}((S^1 \cap \widehat{\mathbf{S}}^1) - (S^2 \cap \widehat{\mathbf{S}}^2), P^2)$ have been output. Since (P^1, P^2) pairwise cover $\text{preimage}[-](p_r)$, all data items in T^1 are also in $\widehat{\mathbf{S}}^1$, and all data items in T^2 are also in $\widehat{\mathbf{S}}^2$, and therefore all data items in $\text{setMatchTs}((S^1 \cap T^1) - (S^2 \cap T^2), P^2)$ have been output. Thus, p_r can also be output.

Case join ($\bowtie$): By the pass invariant for join, data items from each input that pass the join predicate can be combined and output. Therefore, all data items in $(S^1 \cap \widehat{\mathbf{S}}^1) \bowtie (S^2 \cap \widehat{\mathbf{S}}^2)$ have been output. Since (P^1, P^2) pairwise cover $\text{preimage}[\bowtie](p_r)$, all data items in T^1 are also in $\widehat{\mathbf{S}}^1$, and all data items in T^2 are also in $\widehat{\mathbf{S}}^2$, and therefore all data items in $(S^1 \cap T^1) \bowtie (S^2 \cap T^2)$ have been output. Thus, p_r can also be output.

End of proof.

We use this result to show the required properties of input punctuation schemes to enable binary operators. As for unary operators, for a binary operator O and an output punctuation scheme $\mathcal{P}_R[O]$, let $\mathbf{S}_g = \{\text{preimage}(p_r) \mid p_r \in \mathcal{P}_R[O]\}$.

Theorem 9.4 *For a binary operator O , if $\mathbf{S}_g$ is pairwise compact relative to $(\mathcal{P}_I^1[O], \mathcal{P}_I^2[O])$ then $\mathcal{P}_I^1[O]$ and $\mathcal{P}_I^2[O]$ pairwise enable $\mathcal{P}_R[O]$*

Proof:

Suppose $\mathbf{S}_g$ is pairwise compact relative to $(\mathcal{P}_I^1[O], \mathcal{P}_I^2[O])$. Then:

$$\begin{aligned}
& \forall (P^1, P^2) \in \mathbf{S}_g, P^1 \text{ is compact relative to } \mathcal{P}_I^1[O] \text{ and } P^2 \text{ is compact relative to } \mathcal{P}_I^2[O] \\
& \Rightarrow \forall (P^1, P^2) \in \{\text{preimage}(p_r) | p_r \in \mathcal{P}_R[O]\}, P^1 \text{ is compact relative to } \mathcal{P}_I^1[O] \\
& \quad \text{and } P^2 \text{ is compact relative to } \mathcal{P}_I^2[O] \\
& \Rightarrow \forall p_r \in \mathcal{P}_R[O], \text{preimage}[O](p_r) \text{ is pairwise compact relative to } \mathcal{P}_I^1[O] \text{ and } \mathcal{P}_I^2[O] \\
& \Rightarrow [\text{by Theorem 9.3}] \forall p_r \in \mathcal{P}_R[O], p_r \text{ will be emitted} \\
& \Rightarrow \forall p_r \in \mathcal{P}_R[O], \mathcal{P}_I^1[O] \text{ and } \mathcal{P}_I^2[O] \text{ pairwise enable } p_r \\
& \Rightarrow \mathcal{P}_I^1[O] \text{ and } \mathcal{P}_I^2[O] \text{ pairwise enable } \mathcal{P}_R[O]
\end{aligned}$$

End of proof.

Now we have the required properties for input punctuation schemes in order to enable the output punctuation scheme of unary and binary operators. Our next goal is to show how a set of input punctuation schemes cleanses a query operator. That is, we must show that any data item that resides in the state maintained for a query operator will eventually be removed.

9.4 Cleansing Query Operators with Punctuation Schemes

In order to reason about the state of a query operator, we must first construct a model of state for each operator. We use groupings to model state for various operators. We will show that if the input punctuation schemes are complete, and if all groupings for the state for an operator are compact relative to the input punctuation schemes, then the operator will be cleansed.

In our discussion of enabling a query operator, we focused on the logical definition of that operator and did not need to consider the implementation of that operator. However, because we must model the state maintained during execution in order to determine whether an operator is cleansed, we must now include the implementation of a query operator. We limit our discussion to well-known implementations of query operators that are suitable for processing non-terminating data streams. For example, the implementation of join that we consider does not block on either input and does not require indexes.

One data structure used in many implementations of query operators is the hash table. Values for a subset of the attributes of a data item make up the key for the hash table.

Clearly, hash table structures conform neatly into groupings, where the attributes that are used to make up the hash key are used as the grouping attributes. Note that in practice different hash key values may be placed in the same hash bucket. While our model does not exactly match the hash buckets, it is sufficient for modelling how state is maintained. For example, given a data item with the schema $\langle A, B, C, D \rangle$ in some stream S where the hash key is C , the grouping is defined as: $\{\{ \langle a, b, c, d \rangle \mid a \in A, b \in B, d \in D \} \mid c \in C\}$.

9.4.1 Modelling State for Algorithms for Traditional Query Operators

Here we present models for the state maintained by the implementation of operators we consider. We are concerned only with data items from the input that are stored in state. Other possible items that may be maintained, such as overall quality of service data, are not considered. Operators that do not maintain state, such as select, project (duplicate-preserving), and union (duplicate-preserving) are not listed as they are trivially cleansed.

Duplicate Elimination: Duplicate elimination can be implemented using a hash table, where an input data item's hash key can be generated from values of all attributes. When a new data item arrives, we build the hash key and then check the hash table to see if the data item is already there. If not, the data item is output and then added to the hash table. If it does exist, then the data item is not output. Since the hash key is built from values of all attributes, the grouping for this implementation is defined as a collection of singleton sets. Using (A, B, C, D) as an example input schema, the grouping is defined as $\mathbf{G}[\delta] = \{\{ \langle a, b, c, d \rangle \} \mid a \in A, b \in B, c \in C, d \in D\}$.

Group-by: We implement group-by using a hash table, where the hash key is generated using values of the grouping attributes. Therefore, we will model state maintained by group-by using a grouping where the grouping attributes are the group-by attributes. Again using (A, B, C, D) , if the group-by attribute is A , then the grouping is defined as $\mathbf{G}[\mathcal{G}_A^F] = \{\{ \langle a, b, c, d \rangle \mid b \in B, c \in C, d \in D \} \mid a \in A\}$. In practice, full data items will not be held in state. Instead, only information required to generate the resulting data items are held. Our model here using groupings of full data items follows a naïve implementation, but can be adapted to a more common implementation by

considering groupings as required state maintained to produce results for each group.

Sort: We implement sort by maintaining a sorted list of all data items that have arrived in state. To determine what can be output (per the pass invariant for sort), we need to track prefixes of the sorted output. Thus, we model state using a grouping based on ranges, where each range starts at the minimum for the domain of the primary sort attribute. Using (A, B, C, D) for the input schema where the primary sorting attribute is A , our model for sort is: $\{\{ \langle a', b, c, d \rangle \mid b \in B, c \in C, d \in D, a' \in A \wedge a' \leq a \} \mid a \in A\}$. Since we must have covers for prefixes of the input, data items in this model may reside in more than one grouping, in contrast with other state models we have presented.

Join: We use the symmetric hash join [WA91] implementation for the join operator. Consider the following two example input streams: S^1 with attributes (A, B, C, D) and S^2 with attributes (D, E, F) , where the join condition is $S^1.D = S^2.D$. The symmetric hash join maintains one hash table for each input, where the hash keys are built using values of the join attributes for each input data item. We model the state required for the join with two groupings, one for each input, where the grouping attributes are the join attributes for that input. Using the example schema above, the pair of groupings is $(G^1[\bowtie], G^2[\bowtie])$, where:

$$G^1[\bowtie] = \{\{ \langle a, b, c, d \rangle \mid a \in A, b \in B, c \in C \} \mid d \in D\}$$

$$G^2[\bowtie] = \{\{ \langle d, e, f \rangle \mid e \in E, f \in F \} \mid d \in D\}$$

Note that, for binary operators, we model state as a pair of groupings. This model is in contrast to the *preimage* function, which returned a collection of pairs.

Intersect: Intersect can be implemented as a special case of join where all attributes participate in the join condition. Thus, we can use the model for state as for join. Using two inputs with the same schema, the hash keys for join will be generated from values of all attributes in the data item. Thus, the grouping attributes in our model of state are all attributes. Using (A, B, C, D) for the schema for both inputs, the pair of groupings is $(G[\cap], G[\cap])$, where:

$$\mathbf{G}[\cap] = \{\{ \langle a, b, c, d \rangle \} | a \in A, b \in B, c \in C, d \in D\}$$

Difference: We implement difference using a hash table for each input, where hash keys for each hash table are built using values from all attributes, as in intersect. When some input t arrives from the positive side, we first probe the hash table for the negative side. If t is there, then it is discarded. Otherwise, t is added to the hash table for the positive side. When an input arrives from the negative side, we first probe the hash table on the positive side and remove all matching data items. We then add the data item to the hash table for the negative side. We define groupings for each input based on the two hash tables as before. Using input streams S^1 and S^2 with attributes (A, B, C, D) , the two groupings are defined as $(\mathbf{G}[-], \mathbf{G}[-])$, where:

$$\mathbf{G}[-] = \{\{ \langle a, b, c, d \rangle \} | a \in A, b \in B, c \in C, d \in D\}$$

For reference, Table 9.5 lists simplified models for states of operators discussed above. Note that all groupings cover the input data scheme(s).

Operator	State Grouping Model
dupelim (δ)	$\mathbf{G}[\delta] = \{\{d\} d \in D_I\}$
group-by ($\mathcal{G}_A^F$)	$\mathbf{G}[\mathcal{G}_A^F] = \{\{d \forall r \in R - A, d.r \in D_I(r)\} \forall a \in A, d.a \in D_I(a)\}$
sort ($\mathcal{S}_A$)	$\mathbf{G}[\mathcal{S}_A] = \{\{ \langle a', d.r \rangle \forall r \in R - \{primary(A)\}, d.r \in D_I(r), a' \in D_I(primary(A)) \wedge a' \leq a \} a \in D_I(primary(A))\}$
join ($\bowtie_A$)	$(\mathbf{G}^1[\bowtie_A], \mathbf{G}^2[\bowtie_A])$ where $\mathbf{G}^1[\bowtie_A] = \{\{d \forall r \in R^1 - A, d.r \in D_I^1(r)\} \forall a \in A, d.a \in D_I^1(a)\}$ $\mathbf{G}^2[\bowtie_A] = \{\{d \forall r \in R^2 - A, d.r \in D_I^2(r)\} \forall a \in A, d.a \in D_I^2(a)\}$
intersect ($\cap$)	$(\mathbf{G}[\cap], \mathbf{G}[\cap])$ where $\mathbf{G}[\cap] = \{\{d\} d \in D_I\}$
difference ($-$)	$(\mathbf{G}[-], \mathbf{G}[-])$ where $\mathbf{G}[-] = \{\{d\} d \in D_I\}$

Table 9.5: State models for various implementations of query operators. We use R to represent the input schema. By $D_I(a)$, we mean the domain of the attribute a in D_I , where D_I is the input domain. The function $primary(A)$ returns the primary sorting attribute in A .

9.4.2 Cleansing Unary Operators with Punctuation Schemes

By defining models for state maintained by various implementations of query operators, we can now describe the kinds of punctuation schemes that cleanse those operators. As

in our presentation of enabling query operators, we first present two theorems and proofs for unary operators, and then present a theorem and proof for binary operators.

Theorem 9.5 *Given a grammatical stream S , a unary operator O that discards state per the keep invariant for O at the earliest possible instant, a model for state for O represented as a grouping $G[O]$, and an input punctuation scheme $\mathcal{P}_I[O]$, if $G \in \mathbf{G}[O]$ is compact relative to $\mathcal{P}_I[O]$, then all data items held in state for O that also exist in G will eventually be removed.*

Proof (using the state models of Table 9.5): Let d be a data item currently held in state such that $d \in G$. Since G is compact relative to $\mathcal{P}_I[O]$, there must exist a finite $P \in \mathcal{P}_I[O]$ that covers G . Since P covers G , there exists $p \in P$ such that $\text{match}(d, p)$. We will refer to the keep invariants for many operators. Formal definitions for keep invariants are listed in Table 9.6 for reference.

Op	Keep Invariants
$\text{dupelim } (\delta)$	$\text{setNomatchTs}(T_I, P_I)$
$\text{group-by } (\mathcal{G}_A)$	$[t t \in T_I \wedge \text{setNomatch}(\pi_A(t), \text{groupP}(A, P_I))]$
$\text{sort } (\mathcal{S}_A)$	$\text{setNomatchTs}(T_I, \text{init}(\text{Sort}_A, P_I))$
$\text{join}^1 (\bowtie)$	$[t t \in T_I^1 \wedge \text{setNomatch}(\pi_J(t), \text{groupP}(J, P_I^2))]$
$\text{join}^2 (\bowtie)$	$[t t \in T_I^2 \wedge \text{setNomatch}(\pi_J(t), \text{groupP}(J, P_I^1))]$
$\text{intersect}^1 (\cap)$	$\text{setNomatchTs}(T_I^1, P_I^2)$
$\text{intersect}^2 (\cap)$	$\text{setNomatchTs}(T_I^2, P_I^1)$
$\text{difference}^1 (-)$	$[t t \in T_I^1 \wedge t \notin T_I^2 \wedge \text{setNomatch}(t, P_I^2)]$
$\text{difference}^2 (-)$	$\text{setNomatchTs}(T_I^2, P_I^1)$

Table 9.6: Non-trivial keep invariants for traditional query operators. Sets T_I and P_I represent the data items and punctuations, respectively, that have arrived so far. Set D_I is represents the input dataspace. For binary operators, the superscripts refer to the first and second input. J refers to the join attributes for the join operator.

Case $\text{dupelim } (\delta)$: By the keep invariant for dupelim , because d is in state, no punctuations p such that $\text{match}(d, p)$ have arrived. Suppose at some later time all punctuations in P have arrived. Therefore, p has now arrived. By the keep invariant for dupelim , when p arrives, d will no longer be maintained in state. By the model for state for dupelim , d is the only member of G . Therefore, all data items contained in G that exist in state will be removed.

Case group-by ($\mathcal{G}_A^F$): Since P covers G , and since G is grouped on values of A , $P' = \text{group}P(A, P)$ also covers G . Recall from Section 6.2 that the function $\text{group}P$ outputs punctuations that match an entire group when we have received enough input punctuations to know that all data items for that group have arrived. By the keep invariant for group-by, because d is still in state, not all punctuations in P have arrived. Since $P \in \mathcal{P}_I[\mathcal{G}_A^F]$ is finite, all of P will eventually arrive. By the keep invariant for group-by, when P has completely arrived, all data items $d' \in G$ such that $\text{setMatch}(d', P')$ will be removed from state. Therefore, all data items contained in G that exist in state will be removed.

Case sort ($\mathcal{S}_A$): Let $\hat{P}$ be the punctuations that have arrived so far. By the keep invariant for sort, because d is currently in state, $\text{setNomatch}(d, \text{init}(\hat{P}))$. That is, we have not yet seen enough punctuation to know that all data items in the output that come before d have arrived. Suppose at some later time all punctuations P have arrived. Since P covers G , it must be that, for all d' that sort before d on A , there exists $p \in P$ such that $\text{match}(d', p)$. That is, $\text{setMatch}(d', \text{init}(\text{Sort}_A, P))$ and $\text{setMatch}(d, \text{init}(\text{Sort}_A, P))$. Therefore, $\text{setMatchTs}(G, \text{init}(\text{Sort}_A, P))$, and all data items currently held in state that also exist in G will be removed.

End of proof.

Now that we can show that a group can be removed from state based on the input punctuation scheme, we extend that to show that each group in a grouping will eventually be removed from state, and therefore the operator is cleansed by the input punctuation scheme.

Theorem 9.6 *Given a grammatical stream S , a unary operator O , and a complete punctuation scheme $\mathcal{P}_I[O]$, if the state model for O is a grouping that is compact relative to $\mathcal{P}_I[O]$, then $\mathcal{P}_I[O]$ cleanses O .*

Proof:

We need to show that, for a unary operator, every data item that at some point resides in state will eventually be removed. Let $d \in D_I$ be some data item that resides in state

at some point during query execution (clearly, data items that do not ever reside in state are not a concern for operator cleansing), and let $\mathcal{P}_I[O]$ be the input punctuation scheme. Since $\mathcal{P}_I[O]$ is complete, there must be a punctuation $p \in \mathcal{P}_I[O]$ such that $\text{match}(d, p)$. Since S is grammatical, it must be that any $p \in \mathcal{P}_I[O]$ such that $\text{match}(d, p)$ must arrive after d in S .

Suppose that the model for state is a grouping $\mathbf{G}[O]$, which is compact relative to $\mathcal{P}_I[O]$. Since d is in state, there must be a group $G_d \in \mathbf{G}[O]$ such that $d \in G_d$. Since $\mathbf{G}[O]$ is compact relative to $\mathcal{P}_I[O]$, G_d is also compact relative to $\mathcal{P}_I[O]$. By Theorem 9.5, all data items in state that are also in G_d can be removed. Therefore, $\mathcal{P}_I[O]$ cleanses O . End of proof.

We have shown the properties for an input punctuation scheme that are required to cleanse a unary operator. In the next section, we will address cleansing binary operators.

9.4.3 Cleansing Binary Operators with Punctuation Schemes

As in our discussion of enabling binary operators, we must modify our definition of cleansing to accommodate binary operators. Given a binary operator O with input streams S^1 and S^2 , a pair of punctuation schemes $(\mathcal{P}^1, \mathcal{P}^2)$ *pairwise cleanse* O if every data item $d_1 \in D_I^1$ and every data item $d_2 \in D_I^2$ that will reside in the state for O will eventually be removed due to punctuations in $\mathcal{P}^1$ and $\mathcal{P}^2$.

Theorem 9.7 *Given grammatical streams S^1 and S^2 , a binary operator O that discards state per the keep invariants for O at the earliest possible instance, and complete input punctuation schemes $\mathcal{P}_I^1[O]$ and $\mathcal{P}_I^2[O]$ for O , if the state model for O is a grouping that is pairwise compact relative to $(\mathcal{P}_I^1[O], \mathcal{P}_I^2[O])$, then $(\mathcal{P}_I^1[O], \mathcal{P}_I^2[O])$ pairwise cleanse O .*

Proof (using the models from Section 9.4.1):

As for cleansing unary operators, we must show that any data item held in state for O will eventually be removed.

Case join ($\bowtie$): Let $\mathbf{G}^1[\bowtie]$ be the grouping of data items from D^1 on the join attributes, and $\mathbf{G}^2[\bowtie]$ be the grouping of data items from D^2 on the join attributes. We must show that any data item contained in $\bigcup \mathbf{G}^1[\bowtie]$ or $\bigcup \mathbf{G}^2[\bowtie]$ will eventually be removed

from state. We construct the proof for $\mathbf{G}^1[\bowtie]$; the proof for $\mathbf{G}^2[\bowtie]$ is similar. Suppose $d \in \bigcup \mathbf{G}^1[\bowtie]$. Then there must be some group $G_d \in \mathbf{G}^1[\bowtie]$ such that $d \in G_d$. For join attributes A , G_d is defined as $G_d = \{x | x \in D_I^1 \wedge \forall a \in A, x(a) = d(a)\}$. As both $\mathbf{G}^1[\bowtie]$ and $\mathbf{G}^2[\bowtie]$ are defined on the same grouping attributes A , there must be a group $\hat{G} \in \mathbf{G}^2[\bowtie]$ such that $\hat{G} = \{x | x \in D_I^2 \wedge \forall a \in A, x(a) = d(a)\}$. That is, all data items that will join with d are in $\hat{G}$. Since $\mathbf{G}^2[\bowtie]$ is compact relative to $\mathcal{P}_I^2[\bowtie]$, $\hat{G}$ must also be compact relative to $\mathcal{P}_I^2[\bowtie]$. Therefore, there must exist a finite subset $\hat{P} \subseteq \mathcal{P}_I^2[\bowtie]$ that covers $\hat{G}$. Let $P' = \text{groupP}(J, \hat{P}) : *_{D_I^1 - J}$. Since $\hat{P}$ covers $\hat{G}$, it must be that P' covers G_d . By the keep invariant for join, when all punctuations $\hat{P}$ arrive, all data items in state that are also contained in G_d will be removed from $\mathbf{G}^1[\bowtie]$ (using P'). Therefore, d will be removed from state. As the proof is similar for $\mathbf{G}^2[\bowtie]$, all data items in state will eventually be removed. Therefore, $(\mathcal{P}_I^1[\bowtie], \mathcal{P}_I^2[\bowtie])$ pairwise cleanse join.

Case intersect ($\cap$): Suppose $(\mathbf{G}[\cap], \mathbf{G}[\cap])$ is pairwise compact to $(\mathcal{P}_I^1[\cap], \mathcal{P}_I^2[\cap])$. Let $\mathbf{G}^1$ be the collection of groups for data items that have arrived from the first input, and $\mathbf{G}^2$ be the collection of groups for data items that have arrived from the second input. Note that $\bigcup \mathbf{G}^1 \subseteq \bigcup \mathbf{G}[\cap]$ and $\bigcup \mathbf{G}^2 \subseteq \bigcup \mathbf{G}[\cap]$. We must show that any data item contained in $\bigcup \mathbf{G}^1$ or $\bigcup \mathbf{G}^2$ will eventually be removed from state. As in the previous case, we construct this proof for the first input; the proof for the second input is similar. Suppose d arrives on the first input, and is contained in $\bigcup \mathbf{G}^1$. Then there must be some group $G_d \in \mathbf{G}^1$ such that $d \in G_d$. Since both inputs have the same grouping definition, and since $\mathbf{G}[\cap]$ is compact relative to $\mathcal{P}_I^2[\cap]$, G_d must also be compact relative to $\mathcal{P}_I^2[\cap]$, and therefore there exists a finite subset $P \in \mathcal{P}_I^2[\cap]$ such that P covers G_d . By the keep behavior for intersect, because d is in state, P has not completely arrived. When P does completely arrive, d will be removed from state. Thus, punctuations from $\mathcal{P}_I^2[\cap]$ will eventually remove all data items from $\mathbf{G}^1$. In similar fashion it can be shown that punctuations from $\mathcal{P}_I^1[\cap]$ will eventually remove all data items in $\mathbf{G}^2$. Therefore, $(\mathcal{P}_I^1[\cap], \mathcal{P}_I^2[\cap])$ pairwise cleanse intersect.

Case difference ($-$): Suppose $(\mathbf{G}[-], \mathbf{G}[-])$ is pairwise compact relative to $(\mathcal{P}_I^1[-], \mathcal{P}_I^2[-])$.

Let $\mathbf{G}^1$ be the collection of groups for data items that have arrived from the first input, and $\mathbf{G}^2$ be the collection of groups for data items that have arrived from the second input. Again, note that $\bigcup \mathbf{G}^1 \subseteq \bigcup \mathbf{G}[-]$ and $\bigcup \mathbf{G}^2 \subseteq \bigcup \mathbf{G}[-]$. Therefore, we must show that any data item contained in $\bigcup \mathbf{G}^1$ or $\bigcup \mathbf{G}^2$ will eventually be removed from state. **Case 1:** Suppose d is contained in $\bigcup \mathbf{G}^1$ (state for the positive input). Therefore, there must exist some $G_d \in \mathbf{G}^1$ such that $d \in G_d$. Since the grouping definition for both inputs is the same, and since $\mathbf{G}^1$ is compact relative to $\mathcal{P}_I^1[-]$, it follows that $\mathbf{G}^1$ is compact relative to $\mathcal{P}_I^2[-]$. Further, G_d must also be compact relative to $\mathcal{P}_I^2[-]$. Therefore there exists a finite subset $P \in \mathcal{P}_I^2[-]$ such that P covers G_d . By the keep behavior for difference, because d is in state, P has not completely arrived. When P does completely arrive, d will be removed from state. Either it will be removed by a punctuation in P or it will have been “cancelled out” by a data item from S^2 . Thus, punctuations from $\mathcal{P}_I^2[-]$ will eventually remove all remaining data items that reside in $\mathbf{G}^1$. **Case 2:** Suppose d is contained in $\bigcup \mathbf{G}^2$ (state for the negative input). The proof follows as for case 1, with the exception that no data items from the positive input will cancel out d . Since in either case all data items will eventually be removed from state, $(\mathcal{P}_I^1[-], \mathcal{P}_I^2[-])$ pairwise cleanse difference.

End of proof.

With this result, we can now determine if a given collection of punctuation schemes will benefit (enable and cleanse) individual query operators. We will use these results in the next section to show if a given collection of punctuation schemes can benefit a given query. Note, however, we do not address cleansing an operator of punctuation. For example, though the union operator does not maintain data items in state, it must maintain punctuations in state in order to propagate punctuations correctly. Cleansing punctuations is an area for future work.

9.5 Using Punctuation Schemes to Benefit Specific Queries

We have tools to determine if given input punctuation schemes benefit a given query operator. Now, we want to be able to determine if particular input punctuation schemes benefit an entire query plan. By a query plan, we mean an arbitrary combination of query operators. First, we will use the example queries brought up in Section 9.1 to determine if the suggested input punctuation scheme benefits each query under a particular punctuation scheme we provide. Then, we will discuss one way to analyze a query to generate an input punctuation scheme to benefit that query.

To test whether given punctuation schemes will benefit a given query plan, our approach is to define scheme assignments for each operator in the plan, where the output punctuation scheme for an operator is an input punctuation scheme for that operator's parent. We will prove that each operator in the query plan benefits from its input punctuation(s). If all operators benefit, then the query will benefit from the input punctuation schemes.

9.5.1 Verifying Punctuation Schemes Benefit Example Queries

First, consider Query 9.1. As before, we express this query in relational algebra as: $Q1 = \sigma_{COUNT(*) > 1}(\mathcal{G}_{sn, si, sp}^{COUNT(*)}(\sigma_{fo=0}(inbound)))$. A possible query plan was given earlier in Figure 9.3.

Suppose the client network stream (which we will call C) was enhanced to output the following punctuation scheme: $\mathcal{P}_C = \{ \langle si, *, *, *, *, *, sn, *, *, * \rangle \mid si \in [0, 2^{32} - 1], sn \in [0, 2^{32} - 1] \}$, and that C emits a grammatical stream. This input punctuation scheme will benefit Query 9.1 if all operators in that query benefit from their respective input punctuation schemes. Let $\mathcal{P}_O = \{ \langle sn, si, *, *, * \rangle \mid sn \in [0, 2^{32} - 1], si \in [0, 2^{32} - 1] \}$ be the output punctuation scheme for the query. Thus, the scheme assignment for the query is $\mathcal{A}[Q9.1] = (\{\mathcal{P}_C\}, \mathcal{P}_O)$. The scheme assignments for each operator in Query 9.1 are given in Table 9.7.

We start with the leaf select operator. The input punctuation scheme is $\mathcal{P}_C$. Given $\mathcal{A}[\sigma_{fo=0}]$ (note that we are using $\mathcal{P}_C$ as both the input and output punctuation schemes),

Op	Punctuation Scheme Assignment
$\mathcal{A}[\sigma_{f_0=0}]$	$= (\{\mathcal{P}_C\}, \mathcal{P}_C)$
$\mathcal{A}[\mathcal{G}_{sn,si,sp}^{COUNT(*)}]$	$= (\{\mathcal{P}_C\}, \mathcal{P}_O)$
$\mathcal{A}[\sigma_{COUNT(*)>1}]$	$= (\{\mathcal{P}_O\}, \mathcal{P}_O)$

Table 9.7: Punctuation scheme assignments for each operator in Query 9.1.

recall for punctuation p that $preimage[\sigma](p) = \mathcal{I}(p)$. For each $p \in \mathcal{P}_C$, it is easy to show that $preimage[\sigma](p)$ is compact relative to $\mathcal{P}_C$, therefore this punctuation scheme enables select. Further, because select does not maintain data items in state, select is trivially cleansed. Therefore, the scheme assignment $\mathcal{A}[\sigma_{f_0=0}]$ benefits select.

In order to show that group-by is enabled by the input punctuation scheme $\mathcal{P}_C$, we must show that for some output punctuation p to be emitted, $preimage[\mathcal{G}](p)$ must be compact relative to $\mathcal{P}_C$. Given $\mathcal{A}[\mathcal{G}_{sn,si,sp}^{COUNT(*)}]$, then $preimage[\mathcal{G}_{sn,si,sp}^{COUNT(*)}](p) = \mathcal{I}(\langle p(si), *, *, *, *, *, p(sn), *, *, * \rangle)$. Since $p(si) \in [0, 2^{32} - 1]$ and $p(sn) \in [0, 2^{32} - 1]$, it must be that there exists $p_c \in \mathcal{P}_C$ such that $\mathcal{I}(\langle p(si), *, *, *, *, *, p(sn), *, *, * \rangle)$ is covered by p_c . Therefore, $\mathcal{I}(\langle p(si), *, *, *, *, *, p(sn), *, *, * \rangle)$ is compact relative to $\mathcal{P}_C$, and so $\mathcal{P}_C$ enables group-by.

To show that group-by is cleansed by $\mathcal{P}_C$, recall that the model for state is a grouping based on the group-by attribute. In this case, the grouping is defined on the grouping attributes sn, si, sp , and so:

$$\begin{aligned} \mathbf{G}[\mathcal{G}_{sn,si,sp}^{COUNT(*)}] = \Big\{ \{ \langle si, di, c, fo, sp, dp, sn, an, f, ts \rangle \mid & di \in [0, 2^{32} - 1], c \in [0, 2^6 - 1], \\ & fo \in [0, 2^{13} - 1], dp \in [0, 2^{16} - 1], an \in [0, 2^{32} - 1], f \in [0, 2^3 - 1], ts \in \mathbb{Z} \} \mid \\ & si \in [0, 2^{32} - 1], sp \in [0, 2^{16} - 1], sn \in [0, 2^{32} - 1] \Big\}. \end{aligned}$$

Given some group $G \in \mathbf{G}[\mathcal{G}_{sn,si,sp}^{COUNT(*)}]$, there must exist some $p \in \mathcal{P}_C$ that covers G . Therefore, $\mathbf{G}[\mathcal{G}_{sn,si,sp}^{COUNT(*)}]$ is cleansed by $\mathcal{P}_C$, and we have that the scheme assignment $\mathcal{A}[\mathcal{G}_{sn,si,sp}^{COUNT(*)}]$ benefits group-by.

It is trivial to show that the root select operator benefits from $\mathcal{A}[\sigma_{COUNT(*)>1}]$. Therefore, because all operators in Query 9.1 benefit from the input punctuation scheme $\mathcal{P}_C$, we have that Query 9.1 benefits from its input punctuation scheme.

We can approach Query 9.2 in the same fashion. For this query, we read from two network streams: Inbound (I) and outbound (O). A relational algebra formula for Query 9.2 is: $Q2 = \pi_{sn,si,sp,(O.ts-I.ts)}((\sigma_{isSYNACK(O)}(O.c)) \bowtie_{O.an=I.sn \wedge O.di=I.di \wedge O.dp=I.sp} (\sigma_{isSYN(I)}(I.c)))$. One possible query plan is shown in Figure 9.4.

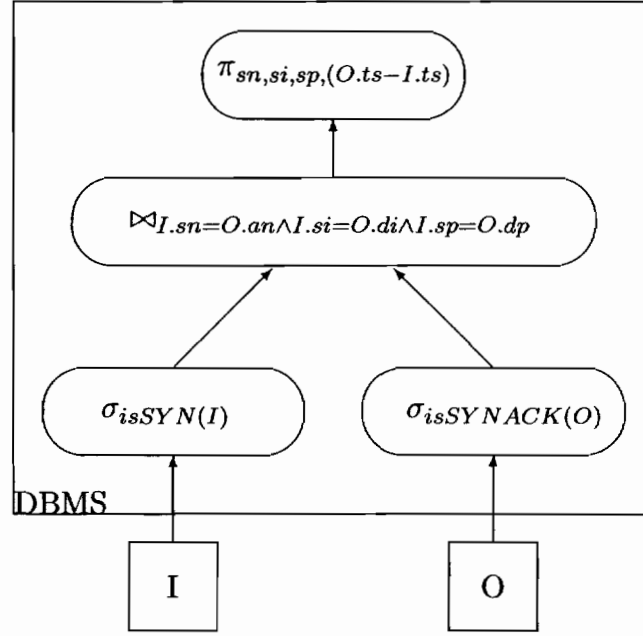


Figure 9.4: The query plan for Query 9.2.

Suppose we enhance the inbound network stream as before, using $\mathcal{P}_C$, and the outbound network stream with $\mathcal{P}_S = \{ \langle *, di, *, *, *, *, *, an, *, * \rangle \mid di \in [0, 2^{32} - 1], an \in [0, 2^{32} - 1] \}$. The leaf select operators benefit from $\mathcal{P}_C$ and $\mathcal{P}_S$ using an argument similar as in Query 9.1. We will try the output punctuation scheme as $\mathcal{P}_O = \{ \langle sn, si, *, * \rangle \mid si \in [0, 2^{32} - 1], sn \in [0, 2^{32} - 1] \}$. In addition, we will also use the scheme assignment $\mathcal{P}_J = \{ \langle si, *, *, *, *, *, sn, *, *, * \rangle \mid si \in [0, 2^{32} - 1], sn \in [0, 2^{32} - 1] \}$. We give punctuation scheme assignments for each operator in Table 9.8. We discuss one way to choose such an assignment in Section 9.5.2.

It is trivial to show that the leaf select operators ($\sigma_{isSYN(I)}$ and $\sigma_{isSYNACK(O)}$) benefit from their respective scheme assignments. We must now show that the join operator

Op	Punctuation Scheme Assignment
$\mathcal{A}[\sigma_{isSYN(I)}]$	$= (\{\mathcal{P}_C\}, \mathcal{P}_C)$
$\mathcal{A}[\sigma_{isSYNACK(O)}]$	$= (\{\mathcal{P}_S\}, \mathcal{P}_S)$
$\mathcal{A}[\bowtie_{O.an=I.sn \wedge O.di=I.di \wedge O.dp=I.sp}]$	$= (\{\mathcal{P}_S, \mathcal{P}_C\}, \mathcal{P}_J)$
$\mathcal{A}[\pi_{sn,si,sp,(S.ts-C.ts)}]$	$= (\{\mathcal{P}_J\}, \mathcal{P}_O)$

Table 9.8: Punctuation scheme assignments for each operator in Query 9.2.

benefits from $\mathcal{A}[\bowtie_{O.an=I.sn \wedge O.di=I.di \wedge O.dp=I.sp}]$. To show that $\mathcal{P}_S$ and $\mathcal{P}_C$ pairwise enable join, it must be that for all $p \in \mathcal{P}_J$, each item in $preimage[\bowtie](p)$ is pairwise compact relative to $(\mathcal{P}_S, \mathcal{P}_C)$. That is, we must show that

$\{\mathcal{I}(p) | p \in \{ \langle *, di, *, *, *, *, *, an, *, * \rangle | di \in [0, 2^{32} - 1], an \in [0, 2^{32} - 1] \} \}$ is compact relative to $\mathcal{P}_S$, and that $\{\mathcal{I}(p) | p \in \{ \langle si, *, *, *, *, *, sn, *, * \rangle | si \in [0, 2^{32} - 1], sn \in [0, 2^{32} - 1] \} \}$ is compact relative to $\mathcal{P}_C$. It can be shown that $preimage[\bowtie](p)$ is compact relative to $\mathcal{P}_S$ and $\mathcal{P}_C$ (since p describes a subset of the join attributes, and $\mathcal{P}_S$ and $\mathcal{P}_C$ describe the same subset of join attributes), and therefore $(\mathcal{P}_S, \mathcal{P}_C)$ pairwise enable the join. Now, because the state model for join is a pair of groupings based on the join attributes, and $\mathcal{P}_S$ and $\mathcal{P}_C$ are also based on the join attributes, it is easy to show that $\mathcal{P}_S$ and $\mathcal{P}_C$ cleanse join. Therefore, $\mathcal{P}_S$ and $\mathcal{P}_C$ benefit the join operator.

Finally, it is easy to see that $\mathcal{P}_J$ enables the project operator. As project is trivially cleansed, we have that $\mathcal{P}_J$ benefits project. Since all operators in this query benefit when the input punctuation scheme is $\mathcal{P}_S$ and $\mathcal{P}_C$, $\mathcal{P}_S$ and $\mathcal{P}_C$ benefit Query 9.2.

9.5.2 Analyzing Queries to Generate Punctuation Scheme Assignments

In the previous examples, we provided a punctuation scheme assignment for analysis. More generally, given a query, we would like to determine if there is a scheme assignment that enables that query (in particular, we want to find appropriate punctuation schemes for the input streams). One approach is to start at the root of the query plan, and work our way down operator by operator using the *preimage* functions for each operator to determine the input punctuation schemes required to output a given punctuation.

For example, consider again Query 9.2, which outputs the amount of time taken between a SYN request from a client to the ACK response from the server. The query plan

and our approach using *preimage* to determine the required subspaces is shown in Figure 9.5.

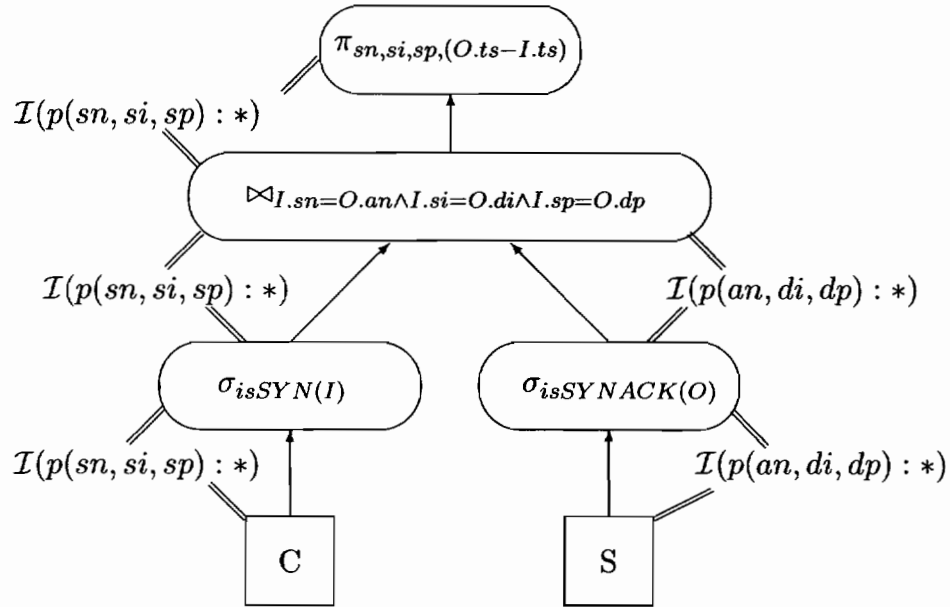


Figure 9.5: The query plan for Query 9.2. Double lines lead to collections of subspaces that must be compact relative to the input punctuation scheme.

In order for project to output a punctuation p , $preimage[\pi](p) = \mathcal{I}(p(sn, si, sp) : *)$ must be compact relative to the input punctuation scheme. We can ensure this if the input punctuation scheme is $\mathcal{P}_J = \{ \langle sn, si, sp, *, *, *, *, *, *, * \rangle \mid sn \in [0, 2^{32} - 1], si \in [0, 2^{32} - 1], sp \in [0, 2^{16} - 1] \}$. For some punctuations $p \in \mathcal{P}_J$, we need that $preimage[\bowtie] = (\mathcal{I}(p(sn, si, sp) : *), \mathcal{I}(p(an, di, dp) : *))$ be pairwise compact relative to punctuations emitted from each select operator. The subspace $\mathcal{I}(p(sn, si, sp) : *)$ is covered by the select on the client stream if the output punctuation scheme is $\mathcal{P}_C$, and the subspace $\mathcal{I}(p(an, di, dp) : *)$ is covered by the select on the server stream if the output punctuation scheme is $\mathcal{P}_S$. Continuing further, we get that the output punctuation scheme from the client source should be $\mathcal{P}_C$ and that the output punctuation scheme from the server source

should be $\mathcal{P}_S$.

Note that this approach has not yet been refined into a complete algorithm. (For example, it needs a means to select the punctuation scheme at the top of the query.) Nor have we formally proved that the resulting scheme assignment always benefits the query. Further, there may be other input punctuation schemes that are better suited for a query than the ones that are produced with this approach. For example, we might want a set of input punctuation schemes with finer granularity in order to purge state more frequently. One final issue with our approach is that it may derive an input punctuation scheme that cannot realistically be generated (for example, requiring a wildcard pattern in a timestamp field). All of these are directions of future work.

9.6 Summary

Understanding how punctuations improve the behavior of individual query operators is interesting. However, it is only a building block for a more important problem, namely, whether a given query can benefit from punctuations, and if so, what kinds of punctuations can benefit that query.

To this end, we first need a way to describe the kinds of punctuations that may appear on a stream. We use punctuation schemes to define the kinds of punctuations that may appear from a stream source. Punctuation schemes are defined using sets, and define a natural grouping. Further, we have seen that various query operators also have natural groupings of interest. When the groupings defined by punctuation schemes cover the groupings of interest for a query, then the punctuation schemes may benefit the query.

However, it is not enough to just be a cover for a grouping of interest. It must also be that the number of punctuations that form the cover is finite, and hence will arrive in finite time. That is, the grouping of interest must be compact relative to the punctuation scheme.

Our work discussed in this chapter is only a starting point. There is a great deal of additional related work that can be done. One possible direction is to come up with a more formal approach for determining necessary input punctuation schemes for a given

query. To be effective, such an approach should be able to list multiple input punctuation schemes as well as suggest an “optimal” set of punctuation schemes. By optimal, we could mean the set of punctuation schemes that output data items the fastest, the set of punctuation schemes that keep the amount of data held in state at a minimum, or the set of schemes that minimize the number of punctuations embedded into the data stream, or perhaps some other criterion.

Our notion of cleansing an operator could be strengthened. We say that an operator is cleansed if any data item that resides in state will eventually be removed from state. A stronger notion would be to give a bound for state. This notion might be captured in one of at least two related ways. First, a given data item will be removed within n data items that follow it (similar to one kind of k -constraints [BSW04]), or second, that there is a bound on how many data items will be held in state. Strengthening our notion of cleansing in this way could improve how an optimizer chooses a query plan.

In this chapter we discussed queries which could benefit from input punctuation schemes. There are a number of related issues, all of which are areas for future work. We would like to come up with an algorithm for determining if a given query cannot benefit from any scheme assignment. It would be very useful to know that a query cannot benefit from an assignment, rather than trying to work through possible scheme assignments and seeing that none help. Also, for a query that cannot benefit from any scheme assignment, we would like to determine if an alternative, equivalent query plan exists that can benefit from a scheme assignment. Finally, we would like to be able to include scheme assignment in the process of query optimization. For example, we would like for query plans that benefit from a scheme assignment to be ranked higher in the optimization cost model than those that do not benefit from any scheme assignments, so that those query plans are more likely to be chosen by the optimizer.

Chapter 10

Related Work

We discuss here work related to our work with punctuated streams. Stephens [Ste97] presents a general survey on stream processing. Chaudhry's text [MTG05] and other recent works [BBD⁺02, Geh03, GO03] give good surveys of issues in stream processing from the data management perspective. Further, Maier et al. [MLT⁺05] give a good overview of semantics of data streams and operators.

10.1 Other Kinds of Stream Semantics

Query operators can process data streams by exploiting a priori or run-time semantics of data streams. If a query operator knows about a stream's semantics, it can improve how it processes a data stream. Our work is based on the semantics of punctuated streams. We have identified other kinds of stream semantics that are potentially useful for query operators, as follows:

1. A *repeat stream* is a sequence of snapshots of some base data, produced in succession. Repeat streams are similar to broadcast disks by Acharya et al. [AAFZ95]. Query operators can take advantage of repeat streams by not saving all the data they read. Since the operator knows that it will see the data again, it need not keep it all in its local state.
2. A *refresh stream* is similar to a repeat stream. It consists mostly of deltas to a base data set, and occasionally the base data set itself. Bose et al. [BF04] use a semantics similar to a refresh stream in their work on executing continuous queries

over time-varying streamed XML data. Query operators can use this semantics to store only the changes of an interesting subset of the data in the stream. If it later needs data from another subset, it can wait for the stream to produce the base data again.

3. A *bounded-reconstitution stream* is similar to a refresh stream. It adds the guarantee that the deltas, when applied to the base data, will yield a result smaller than some fixed bound. Operators can use this semantics to allocate space one time, and know that they will never need to allocate more memory for data, improving efficiency. Additionally, an operator can simply store deltas, rather than applying them immediately. When the operator begins to run out of memory, it can apply the deltas and return the amount of memory used to a fixed level. Delaying the applications of deltas can be useful when deltas arrive quickly, and the operator does not have enough time to process each delta as it arrives.
4. A *constraint-adhering stream* is a stream where constraints in data items or arrival are observed throughout the stream. A stream whose contents arrive ordered on a timestamp value is one simple example. Babu et al. [BSW04] give ways in which constraint-adhering streams can be exploited by various query operators.

10.2 Approaches for Handling Large or Unbounded Data Inputs

In Section 2.2, we discussed various approaches to processing data streams, and how they might be applied to the motivating examples given in Section 1.1. Here we give a more complete list of approaches to processing data streams.

10.2.1 Redefining Blocking Operators to Have Non-Blocking Definitions

The simplest approach for making blocking operators suitable for processing non-terminating data streams is to simply redefine them in a non-blocking fashion. For example, we could redefine aggregate operators to output their current results every time a new data item arrives. This approach solves the problem of using blocking operators over data streams, but

has one major drawback. It does not solve the problem of unbounded memory. Unbounded stateful operators will still accumulate data items in state until memory is exhausted.

An additional problem with this approach has to do with aggregate operators (such as max or average). Often, an aggregate operator is used in a query plan to reduce the amount of data the user sees. By redefining aggregate operators to output current results as new data items arrive, the amount of data output to the user is not reduced at all.

10.2.2 Ordered Input

A query will often have an “order of interest” associated with it. For example, the query in the warehouse scenario groups on the hour attribute. If data arrive from the sensors in order based on hour, then the group-by operator can output results before reading the end of the input. Therefore, the interesting order for this query is on the hour attribute. If data items arrive from a stream in an interesting order relative to a query, then processing of that query can be accomplished using methods from sequence [SLR95, SLR94] and temporal [Soo91] database systems.

When some degree of disorder is introduced, complications in query processing arise. There have been three main approaches to handling disorder in data streams. In Section 7.7 we discussed handling data that arrives out of order using punctuations. The system described by Srivastava et al. [SBW03, SW04] requires that data items be ordered on time. All input streams filter through an input manager, which sorts all input data items on their timestamp. Srivastava et al. introduce the notion of a *heartbeat* to give queries that rely on time-based windows enough information to continue producing output, and introduce ways to generate heartbeats based on known stream properties. A heartbeat τ for a set of input streams provides a guarantee that all data items that arrive from those streams after the heartbeat will have a timestamp greater than or equal to τ . A heartbeat is a very simple form of punctuation. Once the heartbeat arrives, all data items in the input manager with a timestamp less than τ can be passed on to the query plans, ordered on timestamp.

Additionally, Abadi et al. [ACQ⁺03] define a notion of *slack* for queries on a per-operator basis. Slack defines how much disorder an operator will tolerate from an input

stream during execution. Slack is similar to k -constraints in STREAM [BSW04]. For example, we can define a query over a stream of bids for an online auction. The stream generally will be ordered on time, but some data items may arrive out of order. How much disorder we will tolerate in the input data stream for an operator is specified in terms of data items by setting the slack parameter for that operator. Aurora uses the slack parameter for an operator very much like we use punctuations. For example, given a slack value n , an aggregate operator can output results for an existing group after n data items have arrived from the beginning of the next group. Note that the slack may be a number of data items, as in the example above, or a range on some attribute value.

10.2.3 Windows

Window queries have been proposed as an effective way to process non-terminating data streams. Windows break up the input data into successive, contiguous, finite subsets, and queries only process those subsets. Thus, blocking operators can output results when the end of a window arrives. Additionally, data items maintained in state may be purged when they are no longer part of an active window. A number of variations of window queries have been suggested. First, *moving-window* queries were introduced in temporal database systems [OS95, SS87] and later applied to queries over data streams in the Tangram system [PMC89]. For example, a moving-window query over the auction bid stream might be, “At the end of each day, return the seven-day average bid increase for computer monitors.” This kind of query has also been called a *sliding-window* query in the literature [GKS01].

Next, *fixed-window* queries are a special case of moving window queries. A moving-window query returns results as data items in the stream advance. In the example above, results were output at the end of each day. For a fixed-window query, we alter the request slightly to only output results at the end of the moving-window span (seven days), as follows: “At the end of every seventh day, return the seven-day average bid increase for computer monitors.” This kind of query has also been called a *tumbling-window* query [ACQ⁺03] in the literature.

Landmark windows, as discussed by Gehrke et al. [GKS01], consider all data items in a stream from some landmark forward to calculate a result. For example, landmarks in

a data stream could be at the beginning of every hour or every day. Note that multiple windows may result for each landmark.

Damped-window queries were introduced by Zhu and Shasha [ZS02] as an extension of moving-window queries. A damped-window query evaluates each window along with previous windows together, where more recent windows make a greater contribution to the results for a window than older windows.

Finally, Chandrasekaran and Franklin [CF03] have proposed a more generalized window query model, where the user can specify in the query the start and end time for each window. They support two parameters, **BEGIN** and **END**. These parameters can contain constant values or values relative to the current clock, and can express moving windows or fixed windows. It is not clear, however, that damped windows can be expressed in this model.

10.2.4 Approximation

As the size of data from a non-terminating data stream is unbounded, many systems have proposed constructing approximations, or *synopses*, of the data that has arrived from the stream in a single pass. These synopses can then be queried over to get approximate results. Often these synopses are based on concepts from mathematics, and proofs can be made about the accuracy of the generated synopses.

Gilbert et al. [GKMS01] use wavelet transforms to answer point, range, and aged-aggregate queries over the data stream for a telecommunications network monitoring system. Gerhke et al. [GKS01] use histograms to approximate results for correlated aggregate queries over fixed and moving windows. Correlated aggregate queries compute results where two or more aggregate functions are used in determining result membership. For example, for each bidder, how often is the total number of bids per hour for that bidder within 10% of the maximum number of bids for any bidder in that hour. In this case, we require **COUNT** and **MAX** aggregate functions.

Approximations have been used to solve queries beyond aggregate functions as well. Dobra et al. [DGGR02] use pseudo-random sketch summaries on data from streams to process complex queries, including queries with aggregate functions and multiple join

operators. Das et al. [DGR03] use approximation techniques to answer moving window join queries. In both cases, part of the focus is on systems with limited memory. Johnson et al. [JMR05] use a sampling operator over a data stream to produce approximate results for continuous queries.

10.2.5 Load Shedding

As data stream management systems must handle push-based data sources, it is possible that data arrival rate may exceed system capacities. This situation can cause loss of data, increased latency, or even a system crash if not handled properly. One solution to addressing this situation is to *shed load*. That is, drop data items gracefully from the system in order to keep up with incoming data items.

The load shedding work by Tatbul et al. [TCZ⁺03] suggest two methods for load shedding. In the first method, a random fraction of data are dropped. In the second method, data items are dropped based on their perceived importance to the result. *Drop boxes* are placed in the query plan to implement load shedding within a query.

Babcock et al. [BDM04] also address load shedding in stream processing systems. In this work, load shedding is treated as an optimization problem. Load shedding operators are placed at various places in the query plan, based on known statistics about the input data streams. The decision on dropping individual data items from the input is made randomly.

Load shedding may be able to take advantage of punctuations to decide which data items to drop. For example, if a group may be nearly complete, then results for that group may be output and further data items that contribute to that group can be dropped. In the query from the warehouse example, we group on the `hour` attribute. Suppose punctuation arrived that stated all data items for the first 55 minutes for hour h have arrived. If the system becomes overloaded, then the punctuation could be converted to state all data items for hour h have arrived and then emitted. All future data items for hour h will be dropped, thus shedding load.

10.2.6 Bounded Memory Queries

Windows over the number of data items provide one way to ensure that only a bounded amount of memory is considered during query execution. Note that windows over time theoretically make no such guarantees, though realistically the window size is limited by stream bandwidth. Approximations also give some characterization on the amount of memory required to process a given query.

In addition, Arasu et al. [ABB⁺04] discuss methods to characterize the kinds of queries that can be computed over non-terminating data streams in bounded (worst-case) memory. For queries that can be evaluated in bounded memory, they produce an execution strategy. Their discussion is limited to the select, project (including duplicate elimination), join, and group-by operators, but does not address other operators such as intersect, and set difference.

10.2.7 Incomplete Query Processing

As users are often unwilling to wait for completely accurate results, it is often useful to report the results of a query on data seen up to a point. With such a framework, users in general will receive successive approximations. Thus, users can decide when the results are close enough, and cancel further execution of a query.

Online data processing is one approach to producing results to the user before the end of the input has been reached. According to Hidber [Hid99], an algorithm is *online* if the following three conditions hold: First, it gives continuous feedback to the user. Second, it is user-controllable during processing. Third, its result is deterministic and accurate. Online query processing [HH99, HHW97, LEHN02, RH02] is a good example. Online query processing systems report partial results at regular intervals, along with a user interface to report progress and allow users to cancel queries when the result is “good enough”. Users can determine the accuracy of a partial result from confidence level and interval values provided by the system. *A priori* knowledge of the data in the input is required to determine the confidence level and interval. Hidber [Hid99] presents an online algorithm for finding large itemsets for association rule mining. Users can adjust support

and confidence threshold values during query execution.

Another approach, called *partial results*, has been proposed by Shanmugasundaram et al. [STD⁺00]. In a system that supports partial results, query operators awaiting input from other, perhaps blocking, operators, can request a partial result (initiated by the query client). An operator satisfies a request for partial results by producing results based on data seen so far. These results are passed through the system, with a flag marking the data as partial.

10.3 Data Stream Query Implementation Issues

A number of issues have been discussed when implementing systems that query over data streams. These issues concern specific query operators as well as system-wide concerns.

10.3.1 Operator Algorithms

In Chapter 8 we presented our implementation of query operators to support punctuations, making them more appropriate for processing data streams. Much other work has been directed toward implementations of query operators that support non-terminating stream input.

Many traditional implementations of join are blocking, making them inappropriate for data streams. However, the symmetric hash join [WA91] and XJoin [UF00] implementations are all non-blocking. Other implementations, such as merge join and extension join [Hon80] are also non-blocking if certain arrival-order constraints hold. Even so, each requires an unbounded amount of memory and so cannot be used over data streams without help. In Chapter 8 we discuss how punctuations can be used to improve the behavior of symmetric hash join and extension join. The enhancements to XJoin for exploiting punctuations are similar.

Window queries give rise to algorithms specifically designed for window-based operators. Kang et al. [KNV03] evaluate numerous traditional algorithms in terms of a unit-time cost estimation technique, including hash-based algorithms and nested-loop algorithms. They show that, when the arrival rates of one of the input streams is faster than the other,

often an asymmetric combination of algorithms often outperforms a symmetric algorithm. An asymmetric combination of algorithms involves using one algorithm for one input and a different algorithm for the other (e.g., combining nested-loops join with hash join).

Madden et al. [MSHR02] introduce *SteMs* as a way to split the join operator into two unary operators that allow pipelined join computation and sharing of state between joins in different queries. A *SteM* simply encapsulates the in-memory index for a single input using the join attributes as key. The index could be a hash table for equality joins, or a B+-tree or other indexing structure for inequality joins. Separating a join operator into *SteMs* makes join operators more appropriate for use in the Eddy system [AH00]. Instead of data items flowing through a static query tree, an eddy operator continuously reorders the flow of data in a query plan, data item by data item.

Hammad et al. [HAE03] introduce two new window join algorithms, backward and forward evaluation of time-constrained window join, called *BEW-join* and *FEW-join*. The *BEW-join* is more appropriate for lower-rate streams, and *FEW-join* can support higher-rate streams before thrashing. Both algorithms are evaluated using nested loops and hash-based implementations.

10.3.2 Optimization Techniques

We presented *groupings of interest* in Section 9 as a way to determine if a given query could benefit from a particular punctuation scheme. The concept of groupings of interest is similar to the notion of *interesting orders* introduced by Sellinger et al. [SAC⁺79]. A data order is considered interesting if it can be exploited by an operator in the query plan. Simmen et al. [SSM96] give a framework for reasoning about interesting orders during the query optimization phase, and Neumann and Moerkotte [NM04] improve Simmen's framework with a more efficient algorithm. Optimization techniques for determining and reasoning about groupings of interest is an open issue. We discuss open optimization issues in Chapter 11.

10.4 Data Stream Management Systems

Data stream processing systems have been researched for a number of years, though interest has increased recently. There are many systems that could be discussed here. We limit the discussion to systems that address issues related to our work.

The Tangram stream query processing system described by Parker et al. [PMC89] is an early attempt to use data management technology on a data stream. They define a *stream transducer* in a manner similar to our stream iterator, and discuss a number of stream transducer implementations. They also mention that their system can handle moving-window queries, though do not address this point in detail.

Most stream data management systems use windows to handle blocking and unbounded stateful operators. The Tribeca system [SH98] is an early example of such a system. The developers of Tribeca introduce a query language for stream processing. They include analogues to the select and project operators, and add operators specifically for defining windows on a stream. They do not include join, as it was not required in their motivating application (network traffic monitor).

Recently, data stream processing has attracted new interest from the data management community. We have already discussed some related work from the STREAM project [STR]. In addition, Babu et al. [BW01] describe an architecture for queries that process data streams. A query has one or more input streams, an output stream, a *store* for holding tuples that might be part of the result, and a *scratch* for holding state. Punctuations can be used in this architecture in two ways: First, implementing pass behavior minimizes the size of store by moving tuples to the output stream sooner. Second, implementing keep behavior minimizes the size of scratch.

Research in the TelegraphCQ project [Tel] has made a number of contributions to data stream research. In early work, a new query-plan data structure is introduced, called a *fjord* [MF02], to efficiently combine data from static (file-based) and streaming sources. Two initial systems were developed: The CACQ system [MSHR02] applied research on adaptive query processing systems [AH00] to support queries over data streams, and the PSoup system [CF03] presented a model for windows over data streams and discussed the

advantages of using a model that treats queries and data as duals. More recently, work in the TelegraphCQ project has adapted the PostgreSQL system [Pos] so that it can support data streams, incorporating many of the contributions discussed above. An interesting related research direction is to determine if and how punctuations can be used in systems that use eddies. For example, punctuations should be able to be exploited in SteMs to decrease state.

Aurora [Aur] is a dataflow system, where data flow through a loop-free directed graph represented in a “boxes and arrows” network [ACQ⁺03]. One interesting aspect of this system is its focus on perceived quality of service (QoS). An application administrator defines QoS in terms of graphs depicting the importance of such parameters as timeliness and accuracy of results. For example, if the system begins to get overloaded, Aurora uses these graphs to determine how many data items to drop. The more data items dropped, results may be presented to the user faster but less accurately. Another interesting aspect is on load shedding, which we discussed earlier in Section 10.2.5.

Gigascope [JCSS03, CJS03] is a data stream management system especially designed for monitoring network traffic. The focus of this system is to monitor high-speed data streams where the incoming data is assumed to be ordered on time. They use a combination of punctuations and heartbeats to unblock order-preserving stream operators such as union.

10.4.1 Models for Data Stream Processing

We presented in Chapter 4 our model for streams (as a “sliced list”) and stream iterators, and our framework in Haskell in Chapter 5. Other research generally follows the standard stream model as a sequence of data items, but there have been several models for operators that process streams. In this discussion, we will use the stream iterator `select` as a running example. For comparison purposes, in our framework we define a stream iterator as a three-tuple containing (`initial_state`, `step`, and `final`), and the stream iterator `select` is defined as:

```
select :: Tuple a b => (a -> Bool) -> Stream a b -> Stream a b
select pred = unary pred (B step passT prop purgeT)
```

```

where step ts pred = ([t | t <- ts, pred t], pred)
prop ps _ = ps

```

Parker discusses a model for stream transducers [Par90] that is very similar to stream iterator in our framework (and also similar to the model in ATLaS [WZL03]). A *stream transducer* is a 4-tuple, as:

$(InitialState, Transduction, NewState, FinalTransduction),$

where *InitialState* is as our `initial_state`, *Transduction* and *NewState* are encapsulated in our `step` function, and *FinalTransduction* is as our `final` function. Parker limits his discussion to unary stream transducers, and does not address how they interact with non-terminating data streams.

Parker uses the language Prolog to implement his framework. The general form for a stream transducer *StreamTrans* in Prolog is:

```

StreamTrans(Stream)
=> StreamTrans(Stream, InitialState)
StreamTrans([], State)
=> FinalTransduction(State)
StreamTrans([Input|Stream], State)
=> append(Transduction(Input, State),
          StreamTrans(Stream, NewState(Input, State)))

```

Thus, given predicate *pred*, a stream transducer for `select` is $([], pred, newstate, final),$ where:

```

newstate(input, state) = []
final(state) = []

```

Carlsson and Hallgren [HC98] use a streaming model to describe the communication between a user and graphical user interface (GUI) objects (e.g. buttons and text boxes). The stream model is the commonly-used model of a potentially infinite sequence of values. A *stream processor* is a process that consumes input streams and produces output streams. In Haskell, given an input type *i* and an output type *o*, a stream processor type is:

```
data SP i o
```

Stream processors are defined using three basic actions in the Fudget library:

```
putSP  :: output -> SP input output -> SP input output
getSP  :: (input -> SP input output) -> SP input output
nullSP :: SP input output
```

where `putSP` puts data items in the output stream, `getSP` gets input from the input stream, and `nullSP` terminates the stream processor. We can implement the stream processor `select` as follows:

```
select :: (a -> Bool) -> SP a a
select p = getSP $ \ x -> if p x
                        then putSP x $ select p
                        else select p
```

The model proposed by Wang et al. [WZL03] in the ATLaS system [Atl] is an extension built into SQL for user-defined aggregates. They define three blocks within SQL: `INITIALIZE`, `ITERATE`, and `TERMINATE`. The `INITIALIZE` block initializes any required state, `ITERATE` accepts input and updates the state, and `TERMINATE` clears out any required state. The `INITIALIZE` block is similar to our `initial_state`, the `ITERATE` is similar to our `step` function, and `TERMINATE` is similar to our `final` function. Nalamwar [Nal03] shows one method to process punctuations in the ATLaS framework.

As ATLaS is an extension of SQL:1999 [ANS99], the keyword `RETURN` is used to output results. For a stream of integers, we define `select` to only output data items greater than 5, as follows:

```
AGGREGATE select(Next Int): Int
{
    TABLE temp(n Int);
    INITIALIZE: {
        INSERT INTO temp VALUES (Next);
```

```

        INSERT INTO RETURN
            SELECT n
            FROM temp
            WHERE n > 5;
    }
    ITERATE: {
        UPDATE temp
            SET n=Next;
        INSERT INTO RETURN
            SELECT n
            FROM temp
            WHERE n > 5;
    }
}

```

10.4.2 Data Stream Management Languages

One reason to use a DBMS to process queries over data streams is to take advantage of writing applications in SQL. Since SQL is a domain-specific language, writing and maintaining data-intensive applications is often considerably easier. There has been other work to define domain-specific languages for stream processing. For example, researchers at AT&T labs have designed Hancock [CFP⁺00] and the PADS project [FG03] to process streams of cell-phone data. Hancock allows programmers to process streams of data using C-style user-defined structures. Hancock is embedded into the C compiler, allowing programmers who are familiar with the C programming language a powerful way to process data streams. The PADS project uses a domain-specific language, PADSL, to describe the structure of data in a stream. PADSL is compiled into a C library, which can be used to parse, manipulate, or summarize data in a stream.

The Spidle language [CHR⁺03] is a flow-based language that gives programmers a domain-specific language for specifying streaming applications. Like Hancock, Spidle uses syntax similar to the C language. Programmers use Spidle to define various stream tasks, such as filters and mergers, and then define how stream data are piped through each task.

StreamIt [TKA02] is a language and compiler for processing data streams. The designers of StreamIt argue that grid-based architectures are well-suited for stream processing, as they support multiple instruction streams and distributed memory banks. Since the C programming language is based on a single instruction stream and monolithic memory, it may be inappropriate for stream processing. Thus, the designers of StreamIt have two goals in mind in the design of a stream-processing language and compiler: First, like Spidle they provide high-level stream abstractions of stream structures and tasks. Second, they want a common language for grid-based architectures.

10.4.3 Benchmarks for Data Stream Management Systems

As research into querying data streams is still relatively new, there has not been extensive effort in formal benchmarks for systems that query over data streams. We have started work on a benchmark based on the scenario used in the XMark benchmark [SWK⁺01] for XML query engines. Our benchmark is called NEXMark [LMP⁺03]. The scenario is a monitor for an online auction. The data and the queries used in this performance discussion are based on the NEXMark benchmark.

Another effort towards defining a benchmark for data stream management systems is the Linear Road Benchmark [ACG⁺04]. This scenario monitors traffic on a highway system. Vehicles are charged a toll for segments they travel through, and average speed in each segment is tracked. The Linear road benchmark assumes that data arrives at the stream processor in timestamp order. However, in a more realistic model of the scenario the data may be slightly disordered, as data will arrive from many, widely distributed sources (the vehicles themselves). Thus, how well a system handles disorder in data is not addressed with this benchmark.

Chapter 11

Conclusions

A data stream is a possibly non-terminating sequence of data items. There are many examples of data streams where the contents of the stream are structured, including stock ticker streams, streams of environment reports, and network packets. As data in a stream are often structured, it is desirable to use traditional DBMS-style queries over a stream's contents. However, some traditional query operators are inappropriate for processing non-terminating data streams. A blocking operator, such as group-by, difference, or sort, will never output results when reading from non-terminating inputs. An unbounded stateful operator, such as duplicate elimination or join, will accumulate state throughout execution and eventually become memory-bound when reading from non-terminating inputs. These operators are required, however, for many meaningful queries over data streams. Thus, we need to find methods that improve how these kinds of operators process non-terminating data streams as input.

A number of methods have been proposed for querying over data streams, including redefining blocking operators to use non-blocking semantics, relying on ordered input, or redefining queries into window queries. Ordered input can be used by a blocking operator to produce results early. For example, if one of the grouping attributes for group-by is sorted, then a group is complete when new values arrive for those attributes, and that group's results can be output. Windows can also be used to define the end of a group, and results for that window can be output. For example, if one of the grouping attributes is a time period (e.g., hour), we can redefine our query to define windows every 60 minutes, and output results for each window.

We propose embedding punctuations into data streams, and enhancing query operators to exploit those punctuations. A punctuation is a tuple embedded into a data stream stating that no more data items will arrive that match that punctuation. That is, punctuations mark the end of subsets of data. With this approach, we are able to unblock blocking operators and reduce the amount of state required for stateful operators. Thus, we are able to increase the kinds of queries that can be executed over data streams.

11.1 Punctuation Semantics

After seeing encouraging results from initial ad-hoc investigations, we proceeded to define a more formal semantics for punctuations. We noticed in our initial work that there are three different kinds of behaviors an operator can take in the presence of punctuations. First, *pass behavior* defines what results can be output early due to punctuations. Second, *keep behavior* defines what data items must be held in state relative to punctuations (and indirectly what data items can be removed from state). Third, *propagation behavior* defines what punctuations can be emitted from an operator. We use these behaviors throughout our work.

Before we could address punctuation semantics, we first needed formal models of a stream and a stream iterator. The traditional definition of a stream is as a sequence of data items. We ran into problems with this definition having to do with modelling the unpredictability of many data streams, and so we decided to model a stream as a sequence of bounded lists (slices). This model gives us more flexibility in terms of burstiness of input and interleaving of multiple inputs. Then, we defined a stream iterator as a function that can be expressed as repeated applications of some other function over every bounded prefix of the data stream. This definition rules out all functions which must read the entire input before producing an output.

Given definitions for streams and stream iterators, we then enhanced our definitions to include punctuation semantics. We first gave a representation for punctuations, where each punctuation had a schema that resembled the schema for data items in the stream. Then we defined functions used to manipulate punctuations. The most important of these

functions is *match*, which takes a data item and a punctuation and indicates whether the data item belongs to the subset described by the punctuation. Finally, we enhanced our definition of stream iterators to exploit punctuations.

11.2 Framework and Theory for Punctuations

Given a semantics for punctuations, we then proceeded to develop a framework within which we implemented various stream iterators. As in our work on punctuation semantics, we first developed our framework without enhancements for punctuations, and then added in punctuations. We abstracted common behavior from stream iterators into a general function for unary iterators and another general function for binary iterators. The specific implementation of a stream iterator is defined in separate behavior functions, which are passed to the appropriate general function.

To add punctuations to our stream iterator framework, we introduced three new functions for defining the specific behavior of a stream iterator, one for each punctuation behavior. The general functions were enhanced to call these new punctuation behavior functions. Using this framework, we were able to implement the queries defined in both the warehouse and auction example scenarios.

In addition to our framework, we also developed a formal theory for punctuations. Punctuation behaviors for each operator were defined more formally, called punctuation invariants. Using traditional definitions of various query operators and a formalization of *correctness*, we were able to prove that a stream iterator that adheres to its punctuation invariants is faithful to its counterpart table operator. Further, we were also able to give proofs that various implementations of stream iterators in our framework adhere to the punctuation invariants, and therefore are faithful. This two-phase approach to faithfulness proofs allows us to prove that different implementations of the same stream iterator adhere to the punctuation invariants, but only have to prove one time that a stream iterator that adheres to the invariants is faithful to its counterpart query operator.

11.3 Implementation and Performance

Our theory of punctuation semantics seemed reasonable, but we also wanted to test punctuations in practice. We made enhancements to the Niagara Query Engine, which executes queries over XML data. We made three kinds of enhancements: First, we enhanced general classes to represent and generically handle punctuations for the entire system. Second, we made enhancements to specific query operators based largely on their punctuation invariants. Third, we designed and implemented two new classes of query operators specifically for processing punctuations, namely *punctuate* and *describe*. Throughout this discussion, we also addressed other questions we ran into while making these enhancements, including ways to embed punctuations into a data stream and how to handle disorder in a data stream.

We then conducted performance tests using the enhancements to Niagara using an online-auction monitoring scenario and defining five queries for that scenario. Each query was tested using streams with varying amounts and kinds of punctuations, to determine the cost of processing punctuated streams. For these queries, we show that the overhead required to process punctuations is not significant for many kinds of queries. The performance of simple queries that do not require punctuations at all is not significantly affected by reasonable amounts of punctuations. The behavior of queries that require blocking operators and unbounded stateful operators is improved using specific kinds of punctuations as well.

11.4 Benefiting Entire Queries

Our work to this point had focused on how the behavior of individual stream operators can be improved using punctuations. We apply this work to entire queries. We defined groupings of interest, based on properties of operators in a query. For example, the grouping of interest for the group-by operator is based on the grouping attributes. A stream operator benefits from a set of punctuations if all results for that operator will eventually be output, and if every data item that exists in the state for that operator will eventually be released. We showed that, if each stream operator in a given query is

faithful to its relational counterpart, and if the groupings of interest for each operator are covered by a finite set of input punctuations, then the query will be unblocked by the punctuation scheme. Finally, we showed how this theory can be applied to specific queries in the online auction scenario.

11.5 Future Work

Research in punctuation semantics, as well as querying over data streams in general, is still very young. There is a great deal more research that can be done related to punctuation semantics and data streams. The following discussion falls into the following categories: Query optimization and execution issues, real-world applications, alternative punctuation semantics, and other methods for benefiting queries. This discussion is not exhaustive, but these are ideas that we considered during this effort.

11.5.1 Query Optimization Issues

Traditional query optimization techniques must be revisited for the case of data stream inputs in at least two ways. First, the I/O cost model commonly used in a DBMS query optimizer may not be appropriate for data streams as it is for stored relations. Since data from a stream are typically pushed into the system, the cost of reading data from a stream is negligible. Other cost models may be more appropriate for stream inputs. One such model is CPU cost. For example, if a given query plan is estimated to process data at a rate slower than the data arrival rate from a stream, then that plan may not be a good choice for processing the stream.

Another interesting cost model for query optimization is a query's estimated memory requirement. Given two equivalent query plans, if one can be executed using a bounded amount of memory, then that plan may be more desirable than an equivalent plan whose memory requirement is unbounded. Arasu et al. [ABB⁺04] give a good start in this direction. Punctuations can play a role in this cost model. If the input punctuation schemes are known to the query operator, and if one query plan under consideration is shown to be cleansed by those punctuation schemes, then that query plan may be more

desirable than another equivalent query plan that is not cleansed by the input punctuation schemes.

A query optimization technique that must be revisited for data stream inputs is algorithm selection for operators in a query plan. In a traditional DBMS, many query operators have multiple implementations that can be considered for use in a query plan. Some implementations are generally more appropriate for data stream inputs than other implementations. We have discussed this issue briefly for duplicate elimination and join. Duplicate elimination can be implemented using a sort-based algorithm or a hash-based algorithm. As the sort-based algorithm is blocking, it is less appropriate for data streams than the non-blocking hash-based algorithm. Work by Kang et al. [KNV03] has shown some interesting related results. They show that an asymmetric combination of join algorithms (e.g., hash-join for one input, nested loops join for the other input) can outperform traditional symmetric algorithms for window join queries over data streams. Such findings may also apply to a join query where the inputs are a stored (bounded) relation and a data stream. Punctuations can also play a role in algorithm selection. Suppose for a query that joins two data streams that the input punctuation scheme denotes the end of prefixes of the input data sorted on the join attributes. Such information may make the merge-join algorithm viable for performing the join operation. Similarly, punctuations denoting the end of prefixes of input data in some sort order may make the sort-based algorithm of duplicate elimination suitable for data stream input.

In this work we have assumed a fixed query tree is being analyzed. We have observed cases, however, where logically equivalent query trees have different behavior under the same punctuation scheme as regards unblocking or purging state. If we determine a particular query tree does not benefit from a given punctuation scheme, a natural question is if there is any equivalent query that does benefit. Obviously, if one can analyze individual query trees and exhaustively generate all equivalent queries to a given query, then there is a brute-force approach to determining the answer. However, we would like to find more efficient searches by reasoning about the input punctuation schemes and properties of operators.

In Chapter 9 we defined benefits to be “all or nothing.” That is, either the given

input punctuation schemes benefit a specific query in its entirety or they do not. A more flexible approach would be to quantify how much of a given query is benefitted by its input punctuation schemes. For example, given a query plan, perhaps not all operators in that plan will benefit from its input punctuation schemes, but enough of the query can be unblocked and enough state removed such that the query will be execute successfully. Such an approach would be more meaningful for a query optimizer, allowing it to choose from two equivalent query plans based on how much each benefit from punctuations.

Also in Chapter 9, given some specific query, we gave an informal algorithm for determining candidate groupings that must be covered by input punctuation schemes in order that the punctuation schemes benefit the query. Our algorithm is not complete, nor has it been proven to always produce candidate groupings with the guarantee that, if covered, the query will benefit. For example, we do not discuss in our algorithm how to determine the output punctuation scheme for a given query. Further work is needed to refine our algorithm into one that can be proven effective, as well as one that can be implemented by a query optimizer.

11.5.2 Query Execution Issues

We have focused a lot of attention on the behavior of queries and query operators during execution due to punctuations. However, there are still more directions that can be taken regarding punctuations and query execution. For example, in our work we focus on punctuations flowing up a query tree, along with data items. However, there may be advantages to pushing punctuations down a query tree as well, when we know of certain subsets of data that are no longer of use. Let us suppose a dynamic version of the select operator existed whose predicate could be updated during execution. Now suppose we have a query involving a join that reads from two of these dynamic select operators. When a punctuation arrives from one input denoting the end of data items for specific values of the join attributes, and if there are no data items in memory for that input that match that punctuation, then that punctuation can be pushed down to the other dynamic select operator. The operator could combine the predicate in the punctuation with its own predicate, increasing its selectivity. In doing so, we may reduce the amount of state

required by the join operator, because more data items may be filtered out before they reach the join.

Other kinds of query execution engines may also benefit from punctuations. Online query processing systems [HH99, HHW97, RH02] can use punctuations in a number of ways. For example, online aggregate operators produce incremental results as well as an accuracy estimate for each result. Online aggregation operators can be enhanced to use punctuations to determine when the results for a particular group are completely accurate, and no longer an approximation. The user interface described by Hellerstein et al. [HHW97] could use these punctuations to mark a result for a particular group complete. As data items for new groups arrive, the interface may become cluttered and confusing. Users can be given the option to remove or hide results for completed groups from view.

Query execution engines that support *eddy* operators [AH00, MSHR02] may also be able to support punctuations. An eddy must track which operators have processed a given data item. Only when all operators in the query have processed the data item can it be removed from state. As eddy operators only operate on pipelined operators, the handling of punctuations in an eddy as well as the propagation of punctuations from an eddy to non-pipelined operators (such as group-by) is an interesting direction of research.

The Borealis system [AAB⁺05] supports revision and replay of results due to imperfect input or load-shedding. This revision support requires maintaining history information in state for many operators. Punctuations may be sent from lower-level query operators to those upstream in the query plan to reduce the amount of history required for operators.

11.5.3 Applying Punctuations in Real-World Application Domains

We used three simple scenarios for data stream processing to motivate our work. However, all three scenarios were simulations. Further, the queries we presented, though realistic, were only a subset of those used in complete systems. We would like to apply our work to real applications, with the goal of developing a usable, complete system. In developing a real application, we may be able to address some of the optimization and execution issues discussed above. Additionally, we can address ways to embed punctuations into

data streams with a more concrete example. In our simulations, disorder was “forced” into the input streams. Testing with in a real application would help us get a better idea of what real disorder looks like, and how well we actually handle it.

One very appealing application domain is network monitoring. Our network monitoring scenario used two queries, both of which required punctuations to run effectively. Clearly, there are many more queries that network administrators must execute in order to maintain and monitor their systems. For example, many techniques employed by hackers, such as email bombs and web sit-ins, take advantage of automated software [Den01]. Thus, in many cases a signature in the packet may be detectable, and if that signature arrives often during a short time period, an alert can be posted. Such a query would involve a group-by operator over time. We want to work closely with network administrators to implement a system that they can use. An initial monitoring system might first focus on queries for performance monitoring. We have taken some initial steps in this direction with some positive results.

11.5.4 Other Semantics for Punctuations

There are a number of other kinds of punctuations that may be used to improve the behavior of query operators. Our punctuations are strict — they say that no more data items will arrive matching a punctuation from the stream. A more relaxed punctuation might tell an operator that there will not be any matching tuples arriving after it for an extended time. Similarly, a *forward punctuation* might be used to tell operators something about the data items that will arrive (as opposed to those that will not arrive). Operators could use that information to enhance buffer eviction policies. Data items not needed for a while are good candidates for swapping to disk.

We have already seen one example of a relaxed punctuation. In the network monitoring application, punctuations marked the end of data items for a specific sequence number, source IP address, and source port. Since the values for sequence numbers from a specific IP address cycle every 4.55 hours, these punctuations are not strict. However, because they line up with the semantics of sequence numbers, they are still acceptable. Another example of a relaxed punctuation is in tracking moving objects. Consider a system that

monitors the movement of a platoon on a battlefield. One interesting query reports when every member of the platoon cross a particular boundary. A relaxed kind of punctuation could be embedded into a stream stating that no more reports will arrive from platoon members behind the boundary, even though it is entirely possible that some or all members may cross back at a later point.

Punctuations can also be used to declare information about the arriving data, rather than the data that has been seen. For example, Fegaras et al. [FLBC02] use annotations in the data stream to declare the incoming data structure, and whether the data fragment following the annotation is a repeat or an update. One could pursue this direction further to specify other constraints, such as a sort order over particular attributes or even attribute domains. Such information could be used by query engines in the same manner as system catalogs for query optimization.

To this point, we have focused on “flat” XML data. Punctuation semantics for nested XML data is also very interesting. For example, our implementation of the unnest operator discussed in Chapter 7 is limited due to our assumption on flat XML data. An interesting question is how should unnest handle more deeply nested data where punctuation patterns may or may not have child nodes.

11.5.5 Other Methods that Benefit Queries

In Chapter 9, we say that a query benefits from a set of punctuation schemes if all data results are eventually output (enables) and if all data items that reside in operator state are eventually removed (cleanses). Other techniques also fulfill this definition for certain kinds of queries. For example, ordered input benefits queries that rely on sorted input. Additionally, windows benefit queries that are defined with an appropriate window specification. Interesting future work would be to compare characteristics of various techniques that enable queries in order to evaluate which technique would best benefit a given query.

Further, it would be interesting to determine characteristics of queries that can benefit from any technique, in order to decide what kinds of queries are appropriate for executing over data streams. We have found that the existence of groupings of interest for a query is a good indicator of whether that query can benefit from punctuation schemes. Are

there other factors that can also help determine whether a given query can benefit from a particular technique?

11.6 Concluding Remarks

In this dissertation we have presented a semantics for punctuations embedded in data streams and we have shown how the semantics can be exploited by query operators to effectively execute queries over non-terminating data streams. As more and more kinds of data become available in the form of data streams, the need to be able to process it effectively and generically becomes more important. Generally speaking, a traditional DBMS is not equipped to handle non-terminating data streams, even though data in a data stream are often structured.

Exploiting punctuations embedded in a data stream increases the kinds of queries that can be executed over data streams. The usefulness of our approach has been confirmed in our own work in enhancing the Niagara query engine to support punctuations, as well as work in other systems that use punctuations [DMRH04, Nal03, SJSM05].

As the need for efficient, generic processing of data streams increases, more techniques for handling non-terminating inputs will be required. We believe that various kinds of punctuation semantics, based on the theories we have presented in this dissertation, will be exploited by a number of systems that process non-terminating inputs.

Bibliography

- [AAB⁺05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Linder, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research*, pages 277–289, Asilomar, CA, January 2005.
- [AAFZ95] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 199–210, San Jose, CA, May 1995.
- [ABB⁺04] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29(1):162–194, March 2004.
- [ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, Stanford University, October 2003.
- [ACÇ⁺03] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The International Journal on Very Large Data Bases*, 12(2):120–139, August 2003.
- [ACG⁺04] Arvind Arasu, Mitch Cherniak, Eduardo Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbets. Linear

- road: A stream data management benchmark. In *Proceedings of the International Conference on Very Large Data Bases*, pages 480–491, Toronto, Canada, August 2004.
- [AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 261–272, Dallas, TX, May 2000.
- [Alb91] Joseph Albert. Algebraic properties of bag data types. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 211–219, Barcelona, Catalonia, Spain, September 1991.
- [ANS99] ANSI/ISO/IEC. *International Standard (IS) Database Language SQL — Part 2: Foundation (SQL/Foundation)*, September 1999.
- [Atl] ATLaS project page. URL: <http://wis.cs.ucla.edu/atlas/> [Viewed Aug 03, 2005].
- [Aur] Aurora project page. URL: <http://www.cs.brown.edu/research/aurora/> [Viewed August 03, 2005].
- [BBD⁺02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, Madison, WI, June 2002.
- [BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregate queries over data streams. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 350–361, Boston, MA, March 2004.
- [BF04] Sujoe Bose and Leonidas Fegaras. Data stream management for historical XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 239–250, Paris, France, May 2004.

- [BHL99] Tim Bray, Dave Hollander, and Andrew Layman, editors. *Namespaces in XML*. World Wide Web Consortium, January 1999. URL: <http://www.w3.org/TR/REC-xml-names/> [Viewed August 03, 2005].
- [BSW04] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transaction on Database Systems*, 29(3):545–580, September 2004.
- [BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, September 2001.
- [CÇC⁺02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams — a new class of data management applications. In *Proceedings of the International Conference on Very Large Data Bases*, pages 215–226, Hong Kong, China, August 2002.
- [CF02] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proceedings of the International Conference on Very Large Data Bases*, pages 203–214, Hong Kong, China, August 2002.
- [CF03] Sirish Chandrasekaran and Michael J. Franklin. PSoup: A system for streaming queries over streaming data. *The International Journal on Very Large Data Bases*, 12(2):140–156, 2003.
- [CFP⁺00] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A language for extracting signatures from data streams. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–17, Boston, MA, August 2000.
- [CHR⁺03] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan

- Yu, and Calton Pu. Spidle: A DSL approach to specifying streaming applications. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 1–17, Erfurt, Germany, September 2003.
- [CJS03] Chuck Cranor, Ted Johnson, and Oliver Spatscheck. Gigascope: How to monitor network traffic 5 gbit/sec at a time. INVITED TALK at Workshop on Management and Processing of Data Streams, June 2003.
- [Coh96] Frederick B. Cohen. Internet holes — packet fragmentation attacks, 1996. URL: <http://www.all.net/journal/netsec/1995-09.html> [Viewed August 03, 2005].
- [Den01] Dorothy E. Denning. Activism, hacktivism, and cyberterrorism: The Internet as a tool for influencing foreign policy. In John Arquilla and David Ronfeldt, editors, *Networks and Netwars: The Future of Terror, Crime, and Militancy*, chapter 8. RAND Corporation, 2001.
- [DGGR02] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 40–51, Madison, WI, June 2002.
- [DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedwald. Approximate join processing over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 40–51, San Diego, CA, June 2003.
- [DH00] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 199–210, Boston, MA, August 2000.

- [DMRH04] Luping Ding, Nishant Mehta, Elke A. Rundensteiner, and George T. Heinemann. Joining punctuated streams. In *9th International Conference on Extending Database Technology*, pages 587–604, March 2004.
- [DSRS03] Nilesh N. Dalvi, Sumit K Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. *Journal of Computer and System Sciences*, 66(4):728–762, 2003.
- [EBA] eBay home page. URL: <http://www.ebay.com/> [Viewed Aug 03, 2005].
- [FG03] Kathleen Fisher and Robert E. Gruber. PADS: Processing arbitrary data streams. In *Workshop on Management and Processing of Data Streams*, San Diego, CA, June 2003.
- [FLBC02] Leonidas Fegaras, David Levine, Sujoe Bose, and Vamsi Chaluvasi. Query processing of streamed XML data. In *Proceedings of the 11th International Conference on Information and Knowledge Management*, pages 126–133, McLean, VA, November 2002.
- [Geh03] Johannes Gehrke. Special issue on data stream processing. *IEEE Data Engineering Bulletin*, 26(1), 2003.
- [GKMS01] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the International Conference on Very Large Data Bases*, pages 79–88, Roma, Italy, September 2001.
- [GKS01] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continuous data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–24, Santa Barbara, California, May 2001.
- [GO03] Lukasz Golab and M. Tamer Oszu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

- [Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
- [HAE03] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proceedings of the IEEE International Conference on Scientific and Statistical Database Management*, pages 75–84, Cambridge, MA, July 2003.
- [HC98] Thomas Hallgren and Magnus Carlsson. *Fudgets*. PhD thesis, Chalmers University of Technology, March 1998.
- [HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 287–298, Philadelphia, PA, June 1999.
- [HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 171–182, Tucson, AZ, June 1997.
- [Hid99] Christian Hidber. Online association rule mining. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 145–156, Philadelphia, PA, June 1999.
- [Hon80] Peter Honeyman. Extension joins. In *Proceedings of the International Conference on Very Large Data Bases*, pages 239–244, Montreal, Canada, October 1980.
- [HOO] The Haskell Object Observation Debugger home page. URL: <http://haskell.cs.yale.edu/hood/> [Viewed Aug 03, 2005].
- [Hud00] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.
- [Jav] Java home page. URL: <http://java.sun.com> [Viewed Aug 03, 2005].

- [JCSS03] Theodore Johnson, Chuck Cranor, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 647–651, San Diego, CA, June 2003.
- [JMR05] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12, Baltimore, MD, June 2005.
- [Kas71] Robert H. Kasriel. *Undergraduate Topology*. W. B. Saunders Company, 1971.
- [KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, pages 341–352, Bangalore, India, March 2003.
- [LEHN02] Gang Luo, Curt J. Ellmann, Peter J. Haas, and Jeffrey F. Naughton. A scalable hash ripple join algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 252–262, Madison, WI, June 2002.
- [LM89] Brian K. Livezey and Richard R. Muntz. ASPEN: A stream processing environment. In *PARLE '89: Parallel Architecture and Languages Europe, Volume II: Parallel Languages*, pages 374–388, The Netherlands, 1989.
- [LMP⁺03] Jin Li, David Maier, Vassilis Papadimos, Peter Tucker, and Kristin Tufte. NEXMark — a benchmark for queries over data streams, 2003. URL: <http://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf> [Viewed Aug 03, 2005].
- [LWZ04] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *Proceedings of the International Conference on Very Large Data Bases*, pages 492–503, Toronto, Canada, August 2004.

- [MF02] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 555–566, San Jose, California, February 2002.
- [MLT⁺05] David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. Semantics of data streams and operators. In *Proceedings of the 10th International Conference on Database Theory*, pages 37–52, Edinburgh, UK, January 2005.
- [MSHR02] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–60, Madison, WI, June 2002.
- [MTG05] David Maier, Peter A. Tucker, and Minos Garofalakis. Filtering, punctuation, windows, and synopses. In Nauman Chaudhry, Kevin Shaw, and Mahdi Abdelguerfi, editors, *Stream Data Management*, chapter 3. Springer, 2005.
- [Nal03] Anu Nalamwar. Blocking operators and punctuated data streams in continuous query systems. Master’s thesis, University of California Los Angeles, February 2003.
- [NDM⁺00] Jeffrey Naughton, David DeWitt, David Maier, Jianjun Chen, Leonidas Galanis, Kristin Tufte, Jaewoo Kang, Qiong Luo, Naveen Prakash, and Feng Tian. The Niagara query system. *The IEEE Data Engineering Bulletin*, 24(2):27–33, June 2000.
- [NM04] Thomas Neumann and Guido Moerkotte. An efficient framework for order optimization. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 461–472, Boston, MA, March 2004.
- [OS95] Gultekin Özsoyoglu and Richard T. Snodgrass. Temporal and real-time

- databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.
- [Par90] D. Stott Parker. Stream data analysis in Prolog. In Leon Sterling, editor, *The Practice of Prolog*, chapter 8. MIT Press, 1990.
- [PM88] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.
- [PMC89] D. Stott Parker, Richard R. Muntz, and Lewis Chau. The Tangram stream query processing system. In *Proceedings of the 5th IEEE International Conference on Data Engineering*, pages 556–563, Los Angeles, CA, February 1989.
- [Pos] PostgreSQL database management system. URL: <http://www.postgresql.org/> [Viewed Aug 03, 2005].
- [Pos81a] Jon Postel, ed. RFC 791: Internet protocol: DARPA Internet program protocol specification, September 1981.
- [Pos81b] Jon Postel, ed. RFC 793: Transmission control protocol: DARPA Internet program protocol specification, September 1981.
- [RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. The McGraw-Hill Companies, 2003.
- [RH02] Vijayshankar Raman and Joseph M. Hellerstein. Partial results for online query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 275–286, Madison, WI, June 2002.
- [RNSP97] Juan A. Rodríguez, Pablo Noriega, Carles Sierra, and Julian Padget. FM96.5 A java-based electronic auction house. In *Proceedings of the International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 207–224, London, England, April 1997.

- [RSB00] Prasan Roy, S. Seshadri S. Sudarshan, and Siddhesh Bhole. Efficient and extensible algorithms for multi-query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–260, Dallas, TX, May 2000.
- [Rud64] Walter Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, Inc., 1964.
- [RW01] Bernhard Rumpe and Guido Wimmel. A framework for realtime online auctions. In *Proceedings of IRMA International Conference on Managing Information Technology in a Global Economy*, pages 908–912, Toronto, May 2001.
- [SAC⁺79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, MA, May 1979.
- [SBW03] Utkarsh Srivastava, Shivnath Babu, and Jennifer Widom. Monitoring stream properties for continuous query processing. In *Workshop on Management and Processing of Data Streams*, San Diego, CA, June 2003.
- [Sel88] Timos K. Sellis. Multiple-query processing. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.
- [SH98] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX Annual Technical Conference*, pages 13–24, New Orleans, LA, June 1998.
- [SJSM05] Vladislav Shkapenyuk, Theodore Johnson, Oliver Spatscheck, and S. Muthukrishnan. A heartbeat mechanism and its application in Gigascope. In *Proceedings of the International Conference on Very Large Data Bases (to appear)*, Trondheim, Norway, August 2005.
- [SLR94] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query

- processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 430–441, Minneapolis, MN, May 1994.
- [SLR95] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A model for sequence databases. In *Proceedings of the IEEE International Conference of Data Engineering*, pages 232–239, Taipei, Taiwan, March 1995.
- [Soo91] Michael D. Soo. Bibliography on temporal databases. *SIGMOD Record*, 20(1):14–23, 1991.
- [SS87] Arie Segev and Arie Shoshani. Logical modeling of temporal data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 454–466, San Francisco, CA, May 1987.
- [SSM96] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 57–67, Montreal, Quebec, Canada, March 1996.
- [STD⁺00] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey Naughton, and David Maier. Architecting a network query engine for producing partial results. In *WebDB (Informal Proceedings)*, pages 17–22, May 2000.
- [Ste97] R. Stephens. A survey of stream processing. *ACTA Informatica*, 34(7):491–541, 1997.
- [STR] Stanford stream data manager. URL: <http://www-db.stanford.edu/stream> [Viewed Aug 03, 2005].
- [SW04] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 263–274, Paris, France, May 2004.
- [SWK⁺01] Albrecht Schmidt, Florian Waas, Martin Kersten, Daniela Florescu, Ioana Manolescu, Michael J. Carey, and Ralph Busse. The XML benchmark project.

- Technical Report INS-R0103, Centrum voor Wiskunde en Informatica, April 2001.
- [TÇZ⁺03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 309–320, Berlin, Germany, September 2003.
- [Tel] TelegraphCQ web page. URL: <http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/> [Viewed Aug 03, 2005].
- [TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, Grenoble, France, April 2002.
- [TM02] Pete Tucker and David Maier. Exploiting punctuation semantics in data streams. In *Proceedings of the IEEE International Conference on Data Engineering*, page 279, San Jose, California, February 2002.
- [TM03a] Peter A. Tucker and David Maier. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Engineering Bulletin*, 26(1):33–40, March 2003.
- [TM03b] Peter A. Tucker and David Maier. Dealing with disorder. In *Workshop on Management and Processing of Data Streams*, San Diego, CA, June 2003.
- [TMSF03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.
- [UF00] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *The IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

- [WA91] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the IASTED International Conference of Parallel and Distributed Information Systems*, pages 68–77, Miami, FL, December 1991.
- [Wur03] Peter R. Wurman. Online auction site management. In Hossein Bidgoli, editor, *The Internet Encyclopedia*. Wiley, 2003.
- [WWW98] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Proceedings of the 2nd International Conference on Autonomous Agents (Agents '98)*, pages 301–308, Minneapolis/St. Paul, MN, May 1998.
- [WZL03] Haixun Wang, Carlo Zaniolo, and Change Richard Luo. ATLaS: A small but complete SQL extension for data mining and data streams (demo). In *Proceedings of the International Conference on Very Large Data Bases*, pages 1113–1116, Berlin, Germany, September 2003.
- [XER] Xerces XML parser. URL: <http://xml.apache.org/> [Viewed Aug 03, 2005].
- [YAH] Yahoo! auctions home page. URL: <http://auctions.yahoo.com/> [Viewed Aug 03, 2005].
- [ZS02] Yunyue Zhu and Dennis Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the International Conference on Very Large Data Bases*, pages 358–369, Hong Kong, China, August 2002.

Appendix A

Proofs of Correctness for Various Relational Operators

In this appendix we provide proofs of correctness for additional stream iterators. Specifically, we present proofs for stream iterator implementations of select, project, group-by, union, intersect, and join. Note that proofs for dupelim and difference have already been presented in Section 6.6, and so they are not repeated here.

As with the proofs in Section 6.6, we want to denote the data items and punctuations present in the first i slices of input. We use the notation $ts_i = data(S[i])$ and $ps_i = puncts(S[i])$, where S is the input stream. Also, for $j > i$, let $ts_{ij} = data(S[i \rightarrow j])$ and $ps_{ij} = puncts(S[i \rightarrow j])$. Finally, we will use $ts^i = data(S@i)$ and $ps^i = puncts(S@i)$ (recall $S@i$ means the i^{th} slice of S).

A.1 Stream Iterator for Select

Our implementation for select takes a predicate function (`pred`) and the input stream. The select iterator does not maintain anything in state (the empty list is used for initial state, but it never changes). In select, the definition for `step` outputs any data items that satisfy the predicate, and the definition for `prop` outputs punctuations as they arrive. Since select does not block and does not maintain data in state, our stream iterator implementation of select uses the trivial functions for pass and keep (`passT` and `keepT`).

```
selectS :: Tuple a b => (a -> Bool) -> Stream a b -> Stream a b
selectS pred = unary [] (B step passT prop keepT)
```

where $\text{step } ts_ = (\text{filter } \text{pred } ts, [])$
 $\text{prop } ps_ = ps$

Theorem A.1 *The stream iterator $\text{select}S$ is faithful and proper for the table operator select .*

Proof: We use in our proofs the invariants for select defined below. Let q be the selection predicate and S be the input stream.

$$cpass(ts_i, ps_i) = \sigma_q(ts_i)$$

$$cprop(ts_i, ps_i) = ps_i$$

$$ckeep(ts_i, ps_i) = []$$

A.1.1 Faithfulness of the Invariants for Select

As shown in Theorem 6.1, every monotone table operator g has a faithful stream counterpart for a stream S . As $cpass(ts_i, ps_i) = \sigma_q(ts_i)$, if we use $ckeep(ts_i, ps_i) = ts_i$ we would have the version of select per Theorem 6.1. We now have to show faithfulness if state is maintained per $ckeep(ts_i, ps_i) = []$.

Consider what is output between two points i and $j > i$:

$$\begin{aligned} cpass(ts_j, ps_j) - cpass(ts_i, ps_i) &= \sigma_q(ts_j) - \sigma_q(ts_i) \\ &= \sigma_q(ts_j - ts_i) \\ &= \sigma_q(ts_{ij}) \end{aligned}$$

Thus, the output of select does not depend on holding any previous data items in state. Therefore, maintaining state per $ckeep(ts_i, ps_i) = []$ does not affect faithfulness.

A.1.2 Propriety of the Invariants for Select

For propriety, we see that punctuations emitted by stage i are all the punctuations received, namely ps_i . Can a data item t such that $\text{setMatch}(t, ps_i)$ be emitted after stage i ? If so, it is emitted by some stage $j > i$. So it must be that $t \in cpass(ts_j, ps_j)$. So $t \in ts_j$. However, $t \notin ts_{ij}$, by our assumption that the input is grammatical. Hence $t \in ts_i$, and $t \in cpass(ts_i, ps_i)$. So t must already be emitted at stage i , and by the minimality

condition, will not be output again later. Thus the output of any iterator satisfying *cpass* and *cprop* for select is grammatical given grammatical input, and is therefore proper.

A.1.3 Conformance of the Implementation of Select to its Invariants

Now that we have shown that our invariants for select are faithful and proper, we want to show that our implementation of `selectS` adheres to those invariants.

Pass: As in previous proofs, we want to show that the output at each iteration i is equivalent to the cumulative pass invariant at i minus the cumulative pass invariant at $i-1$. That is, $fst(step(ts^i, st_{i-1})) ++ pass(ps^i, st_{i-1}) = cpass(ts_i, ps_i) - cpass(ts_{i-1}, ps_{i-1})$.

$$\begin{aligned}
 &fst(step(ts^i, st_{i-1})) ++ pass(ps^i, st_{i-1}) \\
 &= fst(step(ts^i, [])) ++ pass(ps^i, []) \\
 &= filter(pred, ts^i) ++ [] \\
 &= [t | t \in ts^i \wedge pred(t)] ++ [] \\
 &= [t | t \in ts^i \wedge pred(t)] \\
 &= [t | t \in (ts_i - ts_{i-1}) \wedge pred(t)] \\
 &= [t | t \in ts_i \wedge pred(t)] - [t | t \in ts_{i-1} \wedge pred(t)] \\
 &= \sigma_{pred}(ts_i) - \sigma_{pred}(ts_{i-1}) \\
 &= cpass(ts_i, ps_i) - cpass(ts_{i-1}, ps_{i-1})
 \end{aligned}$$

Note that, for this proof we assume no duplicates. A similar proof works for duplicates using bag difference instead of difference.

Prop: The *prop* function for select simply outputs punctuations as they arrive. Therefore, the proof that the *prop* function conforms to the *cprop* invariant is trivial and is omitted.

Keep: Since the stream iterator for select does not maintain any data items in state and *ckeep*(*ts*, *ps*) for select is [], the proof for keep is trivial.

End of proof.

A.2 Stream Iterator for Project

The project iterator takes two functions and the input stream. The first function, `prjT`, is the projection function over data items. The second function, `prjP`, is the projection

function over punctuations. It returns punctuations in the `Maybe` data type (defined in the Haskell standard library `prelude.hs`). The state for `project` is an empty list.

The implementation in `project` for `step` simply applies the projection function `prjT` to each input data item. The implementation of `prop` applies the projection function `prjP`. It uses the utility function `justxs`, which removes all `Nothing` objects from a list of objects of type `Maybe`. Like `select`, `project` is not blocking and does not maintain data in state. Therefore, our stream iterator implementation of `project` uses the trivial punctuation functions `passT` and `keepT`.

```
projectS :: (Tuple a b, Tuple c d, Eq b, Eq d) =>
  (a -> c) -> ([b] -> [d]) -> Stream a b -> Stream c d
projectS prjT prjP = unary ([], []) (B step passT prop keep)
  where step ts st = (map prjT ts, st)
        prop ps (psI, ps0) = prjP (ps++psI) \\ ps0
        keep ps st@(psI, ps0) = (psI++ps, ps0 ++ (prop ps st))
```

Theorem A.2 *The stream iterator `projectS` is faithful and proper for the table operator `project`.*

Proof: We consider only the version of `project` that is duplicate preserving in this proof, using bag operators per Albert [Alb91]. In particular, we use ‘ $-$ ’ to denote bag difference. We have already shown that our stream iterator for duplicate elimination is faithful and proper (see Theorem 6.2). We use the following invariants for `project`, where A is the set of projected attributes, and ts_i and ps_i are as before:

$$\begin{aligned} cpass(ts_i, ps_i) &= \pi_A(ts_i) \\ cprop(ts_i, ps_i) &= groupP(A, ps_i) \\ ckeep(ts_i, ps_i) &= [] \end{aligned}$$

Given the input dataspace D , let $\mathbf{G}_A = \{\{d \mid d \in D \wedge \forall a_i \in A, d.a_i = c_i\} \mid \forall a_i \in A, c_i \in D(a_i)\}$. In Section 9.2.1 we call this a grouping. We define $groupP$ as follows:

$$groupP(A, P) = \{p \mid \forall G \in \mathbf{G}_A \text{ such that } setMatchTs(G, P) = G, \forall d \in G, \forall a_i \in A, p.a_i = pattern(d.a_i)\}.$$

The function `pattern` converts the data value into a pattern on that value. Thus, $groupP$ returns new punctuations for specific values of the projected attributes, where all data

items with those values must already have arrived (hence the condition $setMatchTs(G, P)$ in the definition of $groupP$).

A.2.1 Faithfulness of the Invariants for Project

By Theorem 6.1, every monotone table operator g has a faithful stream counterpart for a stream S . As $cpass(ts_i, ps_i) = \pi_A(ts_i)$, if we use $ckeeper(ts_i, ps_i) = ts_i$ we would have the version of project per Theorem 6.1. We now have to show faithfulness if state is maintained per $ckeeper(ts_i, ps_i) = []$.

Consider what is output between two points i and $j > i$:

$$\begin{aligned}
 cpass(ts_j, ps_j) - cpass(ts_i, ps_i) &= \pi_A(ts_j) - \pi_A(ts_i) \\
 &= \pi_A(ts_{ij} ++ ts_i) - \pi_A(ts_i) \\
 &= \pi_A(ts_{ij}) ++ \pi_A(ts_i) - \pi_A(ts_i) \\
 &= \pi_A(ts_{ij})
 \end{aligned}$$

Thus, the output of project does not depend on holding any previous data items in state. Therefore, maintaining state per $ckeeper(ts_i, ps_i) = []$ does not affect faithfulness.

A.2.2 Propriety of the Invariants for Project

Consider a punctuation p emitted by stage i . By the definition of $groupP$ and grammaticality of the input, it must be that all data items $t \in T_1$ such that $matchPat(t.a, p.a)$ for all $a \in A$ have arrived. By $cpass$, $\pi_A(t)$ has already been output. Thus, p can also be output, and any iterator that conforms to $cprop$ is proper.

A.2.3 Conformance of the Implementation of Project to its Invariants

Now that we have shown that our invariants for project are faithful and proper, we want to show that our implementation of `projectS` adheres to those invariants.

Pass: As before, we want to show that the output at each iteration i is equivalent to the cumulative pass invariant at i minus the cumulative pass invariant at $i - 1$. That is, $fst(step(ts^i, st_{i-1})) ++ pass(ps^i, st_{i-1}) = cpass(ts_i, ps_i) - cpass(ts_{i-1}, ps_{i-1})$. Let prj

be the function that converts a list of input data items to output data items, where $prj(t) = t[A]$.

$$\begin{aligned}
&fst(step(ts^i, st_{i-1})) ++ passT(ps^i, st_{i-1}) \\
&= fst(step(ts^i, [])) ++ passT(ps^i, []) \\
&= fst(step(ts^i, [])) ++ [] \\
&= fst(step(ts^i, [])) \\
&= map(prj, ts^i) \\
&= [prj(t) | t \in ts^i] \\
&= [t[A] | t \in ts^i] \\
&= [t[A] | t \in (ts_i - ts_{i-1})] \\
&= [t[A] | t \in ts_i] - [t[A] | t \in ts_{i-1}] \\
&\quad [\text{since } ts_i \subseteq ts_{i-1}] \\
&= \pi_A(ts_i) - \pi_A(ts_{i-1}) \\
&= cpass(ts_i, ps_i) - cpass(ts_{i-1}, ps_{i-1})
\end{aligned}$$

Prop: The stream iterator for project takes as its second input a function that takes a list of punctuations and returns punctuations that have been modified to match the output structure of data items. As we did for pass, we want to show that the output from the *prop* function at some step i is equivalent to the cumulative propagation invariant at i minus the cumulative propagation invariant at $i-1$. We construct this proof using induction. We assume in the following proof that the user-defined function *prjP* conforms to *groupP*. In addition, we will assume that punctuations are not emitted more than once (which holds in our implementation since we store all punctuations that have been emitted).

Base Case: Show $\bigcup_{k=0 \dots 0} prop(ps^j, st_j) = cprop(ts_0, ps_0)$

$$\begin{aligned}
&\bigcup_{k=0 \dots 0} prop(ps^k, st_k) \\
&= prop(ps^0, st_0) \\
&= prjP(ps^0 ++ []) \\
&= prjP(ps_0) \\
&= groupP(A, ps_0)
\end{aligned}$$

$$= cprop(ts_0, ps_0)$$

Induction step: Assume $\bigcup_{k=0 \dots i-1} prop(ps^k, st_k) = cprop(ts_{i-1}, ps_{i-1})$.

Show $\bigcup_{k=0 \dots i} prop(ps^k, st_k) = cprop(ts_i, ps_i)$.

$$\begin{aligned} & \bigcup_{k=0 \dots i} prop(ps^k, st_k) \\ &= (\bigcup_{k=0 \dots i-1} prop(ps^k, st_k)) \cup prop(ps^i, st_i) \\ &= cprop(ts_{i-1}, ps_{i-1}) \cup prop(ps^i, st_i) \\ & \quad [\text{by the induction hypothesis}] \\ &= cprop(ts_{i-1}, ps_{i-1}) \cup prjP(ps^i ++ ps_{i-1}) \\ &= cprop(ts_{i-1}, ps_{i-1}) \cup prjP(ps_i) \\ &= cprop(ts_{i-1}, ps_{i-1}) \cup groupP(A, (ps_i)) \\ &= cprop(ts_{i-1}, ps_{i-1}) \cup cprop(ts_i, ps_i) \\ &= cprop(ts_i, ps_i) \end{aligned}$$

Keep: Since the stream iterator for project does not maintain data items in state and *ckkeep* for project is `[ ]`, the proof for keep is trivial.

End of proof.

A.3 Stream Iterator for Group-By

The iterator below for group-by takes three functions: A grouping function `fGrp` takes a group of data items and returns the grouping value for that group. Data items are kept in state, arranged by values returned by the grouping function. The function `fGrpP` outputs punctuations based on input punctuations that describe the grouping attributes. The function `fAgg` is the aggregate function, which is applied to all groups that are output. The `step` function does not output data and adds new data items to state according to the group each data item belongs to. The `pass` function outputs groups that match punctuations from `fGrpP`. The `prop` function outputs any punctuations that exist in state. The `keep` function removes from state any data items and punctuations that have been output. We also list implementations of `fAgg` for traditional aggregate functions.

`--groupby operator`

`groupbyS :: (Tuple a b, Tuple (c, d) (Pattern c, Pattern d),`

```

Eq a, Eq b, Eq c, Eq d, Ord c) =>
([a] -> c) -> ([b] -> [Pattern c]) -> ([a] -> d) ->
Stream a b -> Stream (c,d) (Pattern c,Pattern d)
groupByS fGrp fGrpP fAgg = unary ([],[],[]) (B step pass prop keep)
  where step ts (gs,psI,ps0) = ([], (addNew fGrp gs ts, psI, ps0))
    pass ps (gs,psI,ps0) = map (\xs -> (fGrp xs,fAgg xs)) tsOut
      where tsOut = matchGroups gs (fGrpP (ps++psI))
    prop ps (gs,psI,ps0) = (map (\p -> (p, wc)) (fGrpP (ps++psI))) \\ ps0
    keep ps st@(gs,psI,ps0) =
      (filter (\ (g,ts) -> setNomatch g (fGrpP (ps++psI))) gs,
       psI++ps,ps0++(prop ps st))
    matchGroups gs ps = map snd (filter (\(g, ts) -> setMatch g ps) gs)
    addNew fGrp gs [] = gs
    addNew fGrp gs (x:xs) = addNew fGrp (addToGroups fGrp gs x) xs
    addToGroups fGrp [] t = [(fGrp [t], [t])]
    addToGroups fGrp ((g,ts):gs) t =
      if (g==(fGrp [t])) then ((g,(t:ts)):gs)
      else ((g,ts):(addToGroups fGrp gs t))

--traditional aggregate functions
sumS val xs = foldr (\z -> (\s -> s+(val z))) 0 xs
maxS val (x:xs) = foldr (\z -> (\s -> max s (val z))) (val x) xs
minS val (x:xs) = foldr (\z -> (\s -> min s (val z))) (val x) xs
countS xs = foldr (\z -> (\s -> s+1)) 0 xs
avgS val xs = (sumS val xs) / (countS xs)

```

Theorem A.3 *The stream iterator groupbyS is faithful and proper for the table operator group-by.*

Proof: Given a set of grouping attributes $A = \{a_1, a_2, \dots, a_n\}$ and a set of aggregate functions F , we define group-by as:

$$\mathcal{G}_A^F(R) = \{t :: \langle f_i(U_t) \rangle \mid t \in \pi_A(R) \wedge f_i \in F \wedge U_t = \{u \mid u \in R \wedge t[A] = u[A]\}\}.$$

That is, for each data item t in $\pi_A(R)$, we append the value of all aggregate functions in F to the set of data items with the same value as t for attributes in A .

We define the invariants for group-by as follows, where A is the set of grouping attributes, ts_i is the set of input data items, and ps_i is the set of input punctuations:

$$cpass(ts_i, ps_i) = \{t :: \langle f_i(U_t) \rangle \mid t \in setMatchTs(\pi_A(ts_i), groupP(A, ps_i)) \wedge$$

$$f_i \in F \wedge U_t = \{u \mid u \in ts_i \wedge t[A] = u[A]\}\}$$

$$cprop(ts_i, ps_i) = [p :: \langle w_i \rangle \mid p \in groupP(A, ps_i) \wedge \forall f_i \in F, w_i = '*']$$

$$ckeep(ts_i, ps_i) = [t \mid t \in ts_i \wedge setNomatch(\pi_A(t), groupP(A, ps_i))]$$

As for the project operator, we are using the function $groupP$ over the grouping attributes A . The pass invariant outputs results for groups where all data items that could possibly contribute to results for a given group have arrived according to input punctuations. The propagation invariant outputs punctuations that completely cover a group that has been output, with wildcard patterns for the aggregate attributes. The keep invariant retains in state those data items that belong to groups that are not completely covered by the input punctuations.

A.3.1 Faithfulness of the Invariants for Group-By

As group-by is not monotone, we must first prove that stream iterators that obey the $cpass$ invariant for group-by is faithful.

Faithfulness (safety): We need to show that, at any stage i , $cpass(ts_i, ps_i) \subseteq \mathcal{G}_A^F(ts_i ++ S)$ for any list S where $setMatchTs(S, ps_i) = []$. Let $t \in cpass(ts_i, ps_i)$ and let D be the dataspace such that $ts_i \subseteq D$. Further, let $L \subseteq D$ be defined as $\{u \mid u \in D \wedge u[A] = t[A]\}$. Since $t \in cpass(ts_i, ps_i)$, it must be that there exists some $p \in groupP(A, ps_i)$ such that $match(\pi_A(t), p)$. (Note that, if $match(\pi_A(t), p)$, then $match(u[A], p)$ for all $u \in L$.) By the definition of $groupP$, because $p \in groupP(A, ps_i)$, no more data items in L will arrive. For S such that $setMatchTs(S, ps_i) = \emptyset$, there cannot exist $s \in S$ such that $t[A] = s[A]$. Thus, there does not exist $s \in S$ such that $s \in L$. Therefore, $t \in cpass(ts_i, ps_i) \subseteq \mathcal{G}_A^F(ts_i ++ S)$, and we have containment.

Faithfulness (completeness) : Suppose $t \in \mathcal{G}_A^F(ts_i ++ S)$ for every S where $setMatchTs(S, ps_i) = \emptyset$. Note that it must be that $t \in \mathcal{G}_A^F(ts_i)$, because S can be $\emptyset$. We must show that $t \in cpass(ts_i, ps_i)$. Again let D be the input dataspace, such that $ts_i \subseteq D$. Further, let $L = \{u \mid u \in D \wedge t[A] = u[A]\}$ (that is, L is the set of data items

that could contribute to the result t). Note that, for any A -group, there can be only one resulting t . Suppose $setMatchTs(L, ps_i) \subset L$. Therefore, not enough punctuations have arrived to completely cover L . (If $setMatchTs(L, ps_i) = L$, then $t \in cpass(ts_i, ps_i)$.) Pick $S = [s]$ such that $setMatch(s, ps_i) = \emptyset$ and $s \in L$ (that is, $s[A] = t[A]$). Since ps_i does not completely cover L , such an s exists. Note that s can be chosen to change the aggregate function values corresponding to t . Since s contributes to the result t , it would not necessarily have been output by $\mathcal{G}_A^F(ts_i ++ S)$ under any additional input S (for example, $S = \emptyset$). This is a contradiction, so it must be that $setMatchTs(L, ps_i) = L$. Since L is a grouping on attributes A , it must be that $setMatchTs(\pi_A(L), groupP(ps_i)) = \pi_A(L)$, and therefore $t \in cpass(ts_i, ps_i)$. Thus, completeness is satisfied.

A.3.2 Reducing State Required for Group-By

Consider some point j after i ($j > i$). We want to show that $cpass(ts_j, ps_j) = cpass(ts_i, ps_i) \cup cpass(ts_{ij} \cup ckeep(ts_i, ps_i), ps_j)$.

left $\subseteq$ right: Let $t \in cpass(ts_j, ps_j)$, and let $U_t = \{u | u \in ts_i \wedge u[A] = t[A]\}$. Note that $\mathcal{G}_A^F(U_t) = t$. Then $U_t \subseteq ts_j$ and $setMatchTs(\pi_A(U_t), groupP(A, ps_j)) = \pi_A(U_t)$. (Case 1) Assume $setMatchTs(\pi_A(U_t), groupP(A, ps_i)) = \pi_A(U_t)$. Since the input is grammatical, $U_t \subseteq ts_i$. Therefore $t \in cpass(ts_i, ps_i)$. (Case 2) Assume $setMatchTs(\pi_A(U_t), groupP(A, ps_i)) \neq \pi_A(U_t)$. By Lemma A.1 (below), it must be that $U_t \cap ts_i = U_t \cap ckeep(ts_i, ps_i)$. That is, any data item that contributes to U_t that arrived as part of ts_i still remains in state as part of the keep invariant for group-by. Therefore, for $U_t \subseteq ts_j$, it must be that $U_t \subseteq ts_{ij} \cup ckeep(ts_i, ps_i)$. We know that $setMatchTs(\pi_A(U_t), groupP(A, ps_j)) = \pi_A(U_t)$. Therefore, $t \in cpass(ts_{ij} \cup ckeep(ts_i, ps_i), ps_j)$.

right $\subseteq$ left: (Case 1) Let $t \in cpass(ts_i, ps_i)$ and U_t be defined as before. Then $U_t \subseteq ts_i$ and $setMatchTs(U_t, groupP(A, ps_i)) = U_t$. Since $ps_i \subseteq ps_j$, $groupP(A, ps_i) \subseteq groupP(A, ps_j)$. Since $ts_i \subseteq ts_j$, $U_t \subseteq ts_j$. Further, because $groupP(A, ps_i) \subseteq groupP(A, ps_j)$, $setMatch(U_t, ps_j) = U_t$, and therefore $t \in cpass(ts_j, ps_j)$. (Case 2) Let $t \in cpass(ts_{ij} \cup ckeep(ts_i, ps_i), ps_j)$ and U_t be defined as before. Then $U_t \subseteq ts_{ij} \cup ckeep(ts_i, ps_i)$ and $setMatchTs(\pi_A(U_t), groupP(A, ps_j)) = \pi_A(U_t)$. Since $ts_{ij} \cup ckeep(ts_i, ps_i) \subseteq ts_j$, $U_t \subseteq ts_j$. Therefore $t \in cpass(ts_j, ps_j)$.

Lemma A.1 For $t \in cpass(ts_j, ps_j)$, $t \notin cpass(ts_i, ps_i)$, and $U_t = \{u | u \in ts_i \wedge u[A] = t[A]\}$, $U_t \cap ts_i = U_t \cap ckeep(ts_i, ps_i)$.

Proof:

By the definition above, note that $\mathcal{G}_A^F(U_t) = t$.

left $\subseteq$ right: Let $x \in U_t \cap ts_i$. Then $x \in U_t$ and $x \in ts_i$. Since $t \notin cpass(ts_i, ps_i)$, it must be that $setMatch(x, groupP(A, ps_i)) = false$. Therefore $setNomatch(x, groupP(A, ps_i)) = true$, and we have $x \in U_t \cap ckeep(ts_i, ps_i)$.

right $\subseteq$ left: Let $x \in U_t \cap ckeep(ts_i, ps_i)$. Then $x \in U_t$ and $x \in ckeep(ts_i, ps_i)$. Since $ckeeper(ts_i, ps_i) \subseteq ts_i$, it must be that $x \in ts_i$, and we have $x \in U_t \cap ts_i$.

End of proof.

A.3.3 Propriety of the Invariants for Group-By

Suppose we emit the punctuation $p \in cprop(ts_i, ps_i)$ at point i and all data items in $cpass(ts_i, ps_i)$ have been output. We want to show that any data item output at a later point does not match that punctuation. Let j be some value such that $j > i$, and let $t \in cpass(ts_j, ps_j)$ such that $match(t, p)$. Our goal then is to show that $t \in cpass(ts_i, ps_i)$ since $match(t, p)$. Let $U_t = \{u | u \in ts_j \wedge t[A] = u[A]\}$. Since $p \in cprop(ts_i, ps_i)$, it must be that there exists a $p \in groupP(A, ps_i)$ such that $setMatchTs(\pi_A(U_t), ps_i) = \pi_A(U_t)$. Since the input is grammatical, it must be that $U_t \subseteq ts_i$, and therefore $t \in cpass(ts_i, ps_i)$. Thus, we have that any iterator adhering to $cprop(ts_i, ps_i)$ is proper.

A.3.4 Conformance of the Implementation of Group-By to its Invariants

Now that we have shown that our invariants for group-by are faithful and proper, we want to show that our implementation of `groupbyS` adheres to the invariants.

Pass: As usual, we want to show that the output at each iteration i is equivalent to the cumulative pass invariant at i minus the cumulative pass invariant at $i - 1$. That is, $fst(step(ts^i, st_{i-1})) ++ pass(ps^i, st_{i-1}) = cpass(ts_i, ps_i) - cpass(ts_{i-1}, ps_{i-1})$. We represent state for `groupbyS` for the calls to `step` and `pass` (but before `keep` is called) at some stage i as: $gs_i = \{(t[A], \sigma_{A=t[A]}(S) | S = ts^i ++ ckeep(ts_{i-1}, ps_{i-1})), t \in S\}$. For the moment, we will assume that state is maintained by `groupbyS` per `keep`. We provide this

proof later. Note that we use the subscript $i - 1$ to denote that *keep* has not yet been called. In addition, because *step* has been called for the i th iteration, the data items ts^i also exist in state.

$$\begin{aligned}
&fst(step(ts^i, gs_i)) ++ pass(ps^i, gs_i) \\
&= [] ++ pass(ps^i, gs_i) \\
&= pass(ps^i, gs_i) \\
&= map((\backslash xs \rightarrow (fGrp(xs), fAgg(xs))), tsOut) \\
&= \{ \langle fGrp(xs), fAgg(xs) \rangle \mid xs \in tsOut \} \\
&= \{ \langle fGrp(xs), fAgg(xs) \rangle \mid xs \in matchGroups(gs_i, fGrpP(ps^i ++ ps_{i-1})) \} \\
&= \{ \langle fGrp(xs), fAgg(xs) \rangle \mid xs \in matchGroups(gs_i, fGrpP(ps_i)) \} \\
&= \{ \langle fGrp(xs), fAgg(xs) \rangle \mid xs \in matchGroups(gs_i, groupP(A, ps_i)) \} \\
&\quad [\text{by our assumption on } fGrpP] \\
&= \{ \langle fGrp(xs), fAgg(xs) \rangle \mid xs \in [snd(t, ts) \mid (t, ts) \in gs_i \wedge \\
&\quad \quad \quad setMatch(t, groupP(A, ps_i))] \} \\
&= \{ \langle fGrp(xs), fAgg(xs) \rangle \mid (x, xs) \in gs_i \wedge setMatch(x, groupP(A, ps_i)) \} \\
&= \{ \langle fGrp(xs), fAgg(xs) \rangle \mid S = ts^i ++ ckeep(ts_{i-1}, ps_{i-1}), x \in \pi_A(S) \wedge \\
&\quad \quad \quad setMatch(x, groupP(A, ps_i)), xs \in \sigma_{A=x}(S) \} \\
&= \{ \langle x, fAgg(xs) \rangle \mid S = ts^i ++ ckeep(ts_{i-1}, ps_{i-1}), x \in \pi_A(S) \wedge \\
&\quad \quad \quad setMatch(x, groupP(A, ps_i)), xs \in \sigma_{A=x}(S) \} \\
&\quad [\text{by the definition of } fGrp] \\
&= \{ \langle x, fAgg(xs) \rangle \mid S = ts^i ++ ckeep(ts_{i-1}, ps_{i-1}), x \in \pi_A(S) \wedge \\
&\quad \quad \quad setMatch(x, groupP(A, ps_i)), xs \in \{u \mid u \in S \wedge x = u[A]\} \} \\
&= \{ \langle x, fAgg(xs) \rangle \mid \\
&\quad \quad \quad S = ts_i - [v \mid v \in ts_{i-1} \wedge \pi_A(v) \in setMatchTs(\pi_A(ts_{i-1}), \\
&\quad \quad \quad groupP(A, ps_{i-1}))], x \in \pi_A(S) \wedge setMatch(x, groupP(A, ps_i)), \\
&\quad \quad \quad xs \in \{u \mid u \in S \wedge x = u[A]\} \} \\
&= \{ \langle x, fAgg(xs) \rangle \mid S = ts_i, x \in \pi_A(S) \wedge setMatch(x, groupP(A, ps_i)), \\
&\quad \quad \quad xs \in \{u \mid u \in S \wedge x = u[A]\} \} - \\
&\quad \{ \langle x, fAgg(xs) \rangle \mid
\end{aligned}$$

$$\begin{aligned}
& S = [v|v \in ts_{i-1} \wedge \pi_A(v) \in setMatchTs(\pi_A(ts_{i-1}), groupP(A, ps_{i-1}))], \\
& x \in \pi_A(S) \wedge setMatch(x, groupP(A, ps_i)), xs \in \{u|u \in S \wedge x = u[A]\} \\
& \text{[by Lemma A.2 below]} \\
& = \{<x, fAgg(xs)>|S = ts_i, x \in \pi_A(S) \wedge setMatch(x, groupP(A, ps_i)), \\
& \quad xs \in \{u|u \in S \wedge x = u[A]\}\} - \\
& \quad \{<x, fAgg(xs)>|S = [v|v \in setMatchTs(\pi_A(ts_{i-1}), groupP(A, ps_{i-1}))], \\
& \quad xs \in \{u|u \in S \wedge x = u[A]\}\} \\
& = cpass(ts_i, ps_i) - cpass(ts_{i-1}, ps_{i-1})
\end{aligned}$$

Lemma A.2 $ts^i ++ ckeep(ts_{i-1}, ps_{i-1}) =$
 $ts_i - [v|v \in ts_{i-1} \wedge \pi_A(v) \in setMatchTs(\pi_A(ts_{i-1}), groupP(A, ps_{i-1}))]$

Proof:

$$\begin{aligned}
& ts^i ++ ckeep(ts_{i-1}, ps_{i-1}) \\
& = ts^i ++ [v|v \in ts_{i-1} \wedge setNomatch(\pi_A(v), groupP(A, ps_{i-1}))] \\
& = [v|v \in ts^i \vee (v \in ts_{i-1} \wedge setNomatch(\pi_A(v), groupP(A, ps_{i-1})))] \\
& = [v|v \in ts^i ++ (ts_{i-1} \wedge setNomatch(\pi_A(v), groupP(A, ps_{i-1})))] \\
& \quad \text{[by grammaticality of input]} \\
& = [v|v \in ts_i \wedge setNomatch(\pi_A(v), groupP(A, ps_{i-1}))] \\
& = [v|v \in ts_i \wedge \neg setMatch(\pi_A(v), groupP(A, ps_{i-1}))] \\
& = [v|v \in ts_i \wedge v[A] \notin setMatchTs(\pi_A(ts_i), groupP(A, ps_{i-1}))] \\
& = ts_i - [v|v \in ts_i \wedge v[A] \in setMatchTs(\pi_A(ts_i), groupP(A, ps_{i-1}))] \\
& = ts_i - [v|v \in ts_{i-1} \wedge v[A] \in setMatchTs(\pi_A(ts_i), groupP(A, ps_{i-1}))] \\
& \quad \text{[By grammaticality of input]}
\end{aligned}$$

End of proof.

Prop:

$$\begin{aligned}
& prop(ps^i, gs_i) \\
& = (map(fGrpP, groupP(A, ps^i))) - cprop(ts_{i-1}, ps_{i-1}) \\
& = [fGrpP(p)|p \in groupP(A, ps_i)] - cprop(ts_{i-1}, ps_{i-1}) \\
& = [<p, '*'>|p \in groupP(A, ps_i)] - cprop(ts_{i-1}, ps_{i-1})
\end{aligned}$$

$$= cprop(ts_i, ps_i) - cprop(ts_{i-1}, ps_{i-1})$$

Keep:

$$\begin{aligned} keep(ps^i, gs_i) &= filter(nomatchGrp(groupP(A, ps^i), gs_i)) \\ &= [t | t \in ts_i \wedge setNomatch(t[A], groupP(A, ps_i))] \\ &= setNomatchTs(ts_i, ps_i) \\ &= ckeep(ts_i, ps_i) \end{aligned}$$

End of proof.

A.4 Stream Iterator for Sort

Our iterator for sort takes two functions as input. The function `comp` takes two data items and returns an `Ordering` value (as defined in `Prelude.hs`) determining if the first data item is sorted before the second. The second function takes a list of punctuations and returns punctuations that match all possible data items for some prefix of the sorted output.

```
sortS :: (Tuple a b, Eq a, Eq b) =>
  (a -> a -> Ordering) -> ([b] -> [b]) -> Stream a b -> Stream a b
sortS comp sinit = unary ([], [], []) (B step pass prop keep)
  where step ts (tsSeen, psSeen, psOut) =
    ([], ((ts ++ tsSeen), psSeen, psOut))
    pass ps (tsSeen, psSeen, psOut) =
      sortBy comp setMatchTs tsSeen (sinit (psSeen++ps))
    prop ps (tsSeen, psSeen, psOut) = (sinit (psSeen++ps)) \\ psOut
    keep ps st@(tsSeen, psSeen, psOut) =
      (setNomatchTs tsSeen (sinit (psSeen++ps)), psSeen++ps,
       psOut++(prop ps st))
```

Theorem A.4 *The stream iterator `sortS` is faithful and proper for the table operator `sort`.*

Proof: We use the following invariants for sort, where A is the set of sorting attributes, with ts_i and ps_i are defined as before:

$$\begin{aligned} cpass(ts_i, ps_i) &= setMatchTs(ts_i, init(A, ps_i)) \\ cprop(ts_i, ps_i) &= init(A, ps_i) \\ ckeep(ts_i, ps_i) &= setNomatchTs(ts_i, init(A, ps_i)) \end{aligned}$$

where $init$ is defined for some input dataspace D as:

$init(A, ps)$ = the maximal ps' such that if $D' = \bigcup_{p \in ps'} \mathcal{I}(p)$, there does not exist $d \in D - D'$ such that $\exists d' \in D'$ where $d \leq_A d'$.

That is, $init(A, ps)$ returns the maximal set of punctuations that cover some prefix of D sorted on A .

A.4.1 Faithfulness of the Invariants for Sort

We must show that a stream iterator is faithful to the table operator sort if it obeys the $cpass$ invariant. That is, we must show safety and completeness.

Faithfulness (safety): We need to show that $cpass(ts[i], ps[i]) \leq \mathcal{S}_A(ts[i] ++ S)$ for any list S such that $setMatchTs(S, ps[i]) = []$. Since order of output is important, we must modify what we mean by faithfulness. By ' $\leq$ ', we mean prefix. Let D be the dataspace of ts , and $x \in cpass(ts[i], ps[i])$. Then it must be that $setMatch(x, init(A, ps[i]))$. By definition of $init$, there does not exist $d \in D$ such that $d \leq x$. Therefore, because $S \subseteq D$, for all $s \in S$, $x \leq s$. Thus, $x \in \mathcal{S}_A(ts[i] ++ S)$, and we have that any stream iterator that satisfies $cpass(ts, ps)$ is safe.

Faithfulness (completeness): Suppose $t \in \mathcal{S}_A(ts_i ++ S)$ for every S where $setMatchTs(S, ps_i) = \emptyset$. Note that it must be that $t \in \mathcal{S}_A(ts_i)$, because S can be $\emptyset$. We must show that $t \in cpass(ts_i, ps_i)$. Suppose $t \notin cpass(ts_i, ps_i)$. Then $setNomatch(t, init(ps_i))$. By the definition of $init$, there must exist some s such that $s <_A t$ and $setNomatch(s, ps_i)$. By setting $S = [s]$, we have a contradiction. Therefore, $t \in cpass(ts_i, ps_i)$, and we have completeness.

A.4.2 Reducing State Required for Sort

Consider some point j after i ($j > i$). We want to show that $cpass(ts_j, ps_j) = cpass(ts_i, ps_i) ++ cpass(ts_{ij} ++ ckeep(ts_i, ps_i), ps_j)$.

left $\subseteq$ right: Let $t \in cpass(ts_j, ps_j)$

$$\Rightarrow t \in setMatchTs(ts_j, init(A, ps_j))$$

$$\Rightarrow t \in setMatchTs(ts_{ij} ++ ts_i, init(A, ps_j))$$

Case 1: Suppose $t \in ts_i$

Case 1a: Suppose $setMatch(t, init(A, ps_i))$

$$\Rightarrow t \in setMatchTs(ts_i, init(A, ps_i))$$

$$\Rightarrow t \in cpass(ts_i, ps_i)$$

$$\Rightarrow t \in cpass(ts_i, ps_i) ++ cpass(ts_{ij} ++ ckeep(ts_i, ps_i), ps_j)$$

Case 1b: Suppose $setNomatch(t, init(A, ps_i))$

$$\Rightarrow t \in setNomatchTs(ts_i, init(A, ps_i))$$

$$\Rightarrow t \in ckeep(ts_i, ps_i)$$

$$\Rightarrow t \in ts_{ij} ++ ckeep(ts_i, ps_i)$$

$$\Rightarrow t \in cpass(ts_{ij} ++ ckeep(ts_i, ps_j))$$

[Since $ps_i \subseteq ps_j$.]

$$\Rightarrow t \in cpass(ts_i, ps_i) ++ cpass(ts_{ij} ++ ckeep(ts_i, ps_i), ps_j)$$

Case 2: Suppose $t \in ts_{ij}$

$$\Rightarrow t \in ts_{ij} ++ ckeep(ts_i, ps_i)$$

$$\Rightarrow t \in setMatchTs(ts_{ij} ++ ckeep(ts_i, ps_i), init(A, ps_j))$$

$$\Rightarrow t \in cpass(ts_{ij} ++ ckeep(ts_i, ps_i), ps_j)$$

right $\subseteq$ left: Let $t \in cpass(ts_{ij} ++ ckeep(ts_i, ps_i), ps_j) ++ cpass(ts_i, ps_i)$

Case 1: Suppose $t \in cpass(ts_{ij} ++ ckeep(ts_i, ps_i), ps_j)$

$$\Rightarrow t \in setMatchTs(ts_{ij} ++ ckeep(ts_i, ps_i), init(A, ps_i))$$

$$\Rightarrow t \in setMatchTs(ts_j, init(A, ps_j))$$

[Since $ts_{ij} ++ ckeep(ts_i, ps_i) \subseteq ts_j$ and $ps_i \subseteq ps_j$]

$$\Rightarrow t \in cpass(ts_j, ps_j)$$

Case 2: Suppose $t \in cpass(ts_i, ps_i)$

$$\begin{aligned}
&\Rightarrow t \in \text{setMatchTs}(ts_i, \text{init}(A, ps_i)) \\
&\quad [\text{Since } ts_i \subseteq ts_j \text{ and } ps_i \subseteq ps_j] \\
&\Rightarrow t \in \text{setMatchTs}(ts_j, \text{init}(A, ps_j)) \\
&\Rightarrow t \in \text{cpass}(ts_j, ps_j)
\end{aligned}$$

End of proof.

A.4.3 Propriety of the Invariants for Sort

We will take the same approach as for other operators. Suppose we emit the punctuation $p \in \text{cprop}(ts_i, ps_i)$ at point i and all data items in $\text{cpass}(ts_i, ps_i)$ have been output. We want to show that any data item output at a later point does not match that punctuation. Let j be some value such that $j > i$, and let $t \in \text{cpass}(ts_j, ps_j)$ such that $\text{match}(t, p)$. Our goal then is to show that $t \in \text{cpass}(ts_i, ps_i)$ since $\text{match}(t, p)$.

Since $p \in \text{cprop}(ts_i, ps_i)$, it must be that $p \in \text{init}(A, ps_i)$. Therefore, by the definition of init , $p \in ps_i$. Since $\text{match}(t, p)$ and grammaticality of input, $t \in ts_i$. Further, we know that $t \in \text{setMatchTs}(ts_i, ps_i)$. Thus, $t \in \text{cpass}(ts_i, ps_i)$.

A.4.4 Conformance of the Implementation of Sort to its Invariants

Now that we have shown that our invariants for sort are faithful and proper, we want to show that our implementation of `sortS` adheres to its invariants.

Pass: For sort we want to show that the output at each iteration i is equivalent to the cumulative pass invariant at i minus the cumulative pass invariant at $i - 1$. Additionally, we must ensure that the output is in sorted order. So we must show that, $\text{cpass}(ts_{i-1}, ps_{i-1}) ++ \text{fst}(\text{step}(ts^i, st_{i-1})) ++ \text{pass}(ps^i, st_{i-1}) = \text{cpass}(ts_i, ps_i)$. For the moment, we will assume that state is maintained by `sortS` per *keep*. We provide this proof later. We will also assume that the user-defined function *sinit* adheres to *init* as defined for the invariants. Finally, we assume that the user-defined function *comp* returns its input in sorted order on A .

$$\text{cpass}(ts_{i-1}, ps_{i-1}) ++ \text{fst}(\text{step}(ts^i, gs_i)) ++ \text{pass}(ps^i, gs_i)$$

$$\begin{aligned}
&= cpass(ts_{i-1}, ps_{i-1}) ++ [] ++ pass(ps^i, gs_i) \\
&= cpass(ts_{i-1}, ps_{i-1}) ++ pass(ps^i, gs_i) \\
&= cpass(ts_{i-1}, ps_{i-1}) ++ \\
&\quad sortBy(comp, \\
&\quad \quad setMatchTs(ts^i ++ ckeep(ts_{i-1}, init(A, ps_{i-1})), sinit(ps_{i-1} ++ ps^i))) \\
&\quad [where A is the set of sorting attributes.] \\
&= cpass(ts_{i-1}, ps_{i-1}) ++ \\
&\quad sortBy(comp, setMatchTs(ts^i ++ ckeep(ts_{i-1}, init(A, ps_{i-1})), sinit(ps_i))) \\
&= cpass(ts_{i-1}, ps_{i-1}) ++ \\
&\quad sortBy(comp, setMatchTs(ts^i ++ ckeep(ts_{i-1}, init(A, ps_{i-1})), init(A, ps_i))) \\
&\quad [by our assumption on sinit.] \\
&= cpass(ts_{i-1}, ps_{i-1}) ++ \\
&\quad sortBy(comp, setMatchTs(ts^i ++ setNomatchTs(ts_{i-1}, init(A, ps_{i-1})), init(A, ps_i))) \\
&= cpass(ts_{i-1}, ps_{i-1}) ++ \\
&\quad sortBy(comp, \\
&\quad \quad setMatchTs(ts^i ++ (ts_{i-1} \setminus \setMatchTs(ts_{i-1}, init(A, ps_{i-1}))), init(A, ps_i))) \\
&= cpass(ts_{i-1}, ps_{i-1}) ++ \\
&\quad sortBy(comp, \\
&\quad \quad setMatchTs(ts^i ++ (ts_{i-1} \setminus cpass(ts_{i-1}, ps_{i-1})), init(A, ps_i))) \\
&= sortBy(comp, setMatchTs(ts^i ++ ts_{i-1}, init(A, ps_i))) \\
&= sortBy(comp, setMatchTs(ts_i, init(A, ps_i))) \\
&= cpass(ts_i, ps_i)
\end{aligned}$$

Prop:

At any given stage i state for $sortS$ maintains three lists: First, the data items still required in state (which we have already assumed to be $ckeep(ts_i, ps_i)$). Second, all punctuations that have arrived (ps_i), and third, all punctuations that have been output up until that stage. For now, we will assume that this third item in state to be $cprop(ts_{i-1}, ps_{i-1})$, which we will prove later.

$$\begin{aligned}
& prop(ps^i, (ts_{i-1}, ps_{i-1}, psOut_{i-1})) \\
&= sinit(ps_{i-1} ++ ps^i) \setminus psOut_{i-1} \\
&= sinit(ps_{i-1} ++ ps^i) \setminus cprop(ts_{i-1}, ps_{i-1}) \\
&= sinit(ps_i) \setminus cprop(ts_{i-1}, ps_{i-1}) \\
&= init(A, ps_i) \setminus cprop(ts_{i-1}, ps_{i-1}) \\
&= cprop(ts_i, ps_i) - cprop(ts_{i-1}, ps_{i-1})
\end{aligned}$$

Keep:

We want to show that state at any stage i is $(ckeep(ts_i, ps_i), ps_i, cprop(ts_i, ps_i))$, as discussed above. We present this proof using induction. State held at any given stage i is the result of the *keep* function applied to the second of the pair return by the *step* function.

Base case ($i = 1$):

$$\begin{aligned}
& keep(ps^1, (snd(step(ts^1, ([], [], []))), [], [])) \\
&= keep(ps^1, (ts^1 ++ [], [], [])) \\
&= keep(ps^1, (ts^1, [], [])) \\
&= (setNomatchTs(ts^1, sinit(ps^1 ++ [])), ps^1 ++ [], [] ++ prop(ps^1, (ts^1, [], []))) \\
&= (setNomatchTs(ts^1, sinit(ps^1)), ps^1, prop(ps^1, (ts^1, [], []))) \\
&= (setNomatchTs(ts^1, sinit(ps^1)), ps^1, sinit(ps^1 ++ []) \setminus []) \\
&= (setNomatchTs(ts^1, sinit(ps^1)), ps^1, sinit(ps^1)) \\
&= (setNomatchTs(ts_1, sinit(ps_1)), ps_1, sinit(ps_1)) \\
&= (setNomatchTs(ts_1, init(A, ps_1)), ps_1, init(A, ps_1)) \\
&\quad \text{[where } A \text{ is the set of sorting attributes.]} \\
&= (ckeep(ts_1, ps_1), ps_1, cprop(ts_1, ps_1))
\end{aligned}$$

Induction step:

Assume state at stage $i - 1$ is: $(ckeep(ts_{i-1}, ps_{i-1}), ps_{i-1}, cprop(ts_{i-1}, ps_{i-1}))$.

Prove state at stage i is: $(ckeep(ts_i, ps_i), ps_i, cprop(ts_i, ps_i))$.

$$\begin{aligned}
& keep(ps^i, (snd(step(ckeep(ts_{i-1}, ps_{i-1}), ps_{i-1}, cprop(ts_{i-1}, ps_{i-1})))) \\
&= keep(ps^i, (ts^i ++ ckeep(ts_{i-1}, ps_{i-1}), ps_{i-1}, cprop(ts_{i-1}, ps_{i-1})))
\end{aligned}$$

$$\begin{aligned}
&= (\text{setNomatchTs}(ts^i ++ \text{ckeep}(ts_{i-1}, ps_{i-1}), \text{sinit}(ps^i ++ ps_{i-1})), ps^i ++ ps_{i-1}, \\
&\quad \text{cprop}(ts_{i-1}, ps_{i-1})) ++ \text{prop}(ps, st_{i-1}) \\
&\quad [\text{where } st_{i-1} \text{ is the state at stage } i - 1.] \\
&= (\text{setNomatchTs}(ts^i ++ \text{ckeep}(ts_{i-1}, ps_{i-1}), \text{sinit}(ps_i)), ps_i, \\
&\quad \text{cprop}(ts_{i-1}, ps_{i-1})) ++ \text{prop}(ps, st_{i-1})
\end{aligned}$$

Let us consider the first and third element in turn.

$$\begin{aligned}
&\text{setNomatchTs}(ts^i ++ \text{ckeep}(ts_{i-1}, ps_{i-1}), \text{sinit}(ps_i)) \\
&= \text{setNomatchTs}(ts^i ++ \text{setNomatchTs}(ts_{i-1}, \text{sinit}(ps_{i-1})), \text{sinit}(ps_i)) \\
&= \text{setNomatchTs}(\text{setNomatchTs}(ts^i ++ ts_{i-1}, \text{sinit}(ps_{i-1})), \text{sinit}(ps_i)) \\
&= \text{setNomatchTs}(\text{setNomatchTs}(ts_i, \text{sinit}(ps_{i-1})), \text{sinit}(ps_i)) \\
&= \text{setNomatchTs}(ts_i, \text{sinit}(ps_i)) \\
&= \text{setNomatchTs}(ts_i, \text{init}(A, ps_i)) \\
&= \text{ckeep}(ts_i, ps_i) \\
&\quad \text{cprop}(ts_{i-1}, ps_{i-1}) ++ \text{prop}(ps^i, st_{i-1}) \\
&= \text{cprop}(ts_{i-1}, ps_{i-1}) ++ \text{prop}(ps^i, (ts^i ++ \text{ckeep}(ts_{i-1}, ps_{i-1}), ps_{i-1}, \text{cprop}(ts_{i-1}, ps_{i-1}))) \\
&= \text{cprop}(ts_{i-1}, ps_{i-1}) ++ \text{sinit}(ps^i ++ ps_{i-1}) \setminus \setminus \text{cprop}(ts_{i-1}, ps_{i-1}) \\
&= \text{sinit}(ps^i ++ ps_{i-1}) \\
&= \text{sinit}(ps_i) \\
&= \text{init}(A, ps_i) \\
&= \text{cprop}(ts_i, ps_i)
\end{aligned}$$

Thus, we have that state is: $(\text{ckeep}(ts_i, ps_i), ps_i, \text{cprop}(ts_i, ps_i))$

End of proof.

A.5 Stream Iterator for Merge

We present a binary form of merge, where merge preserves duplicates. (Note that union can be implemented by combining merge and duplicate elimination.) In our performance experiments (see Chapter 8), we used an implementation that reads from n inputs.

```
bmergeS :: (Eq a, Eq b, Tuple a b) => (Stream a b, Stream a b) ->
```

```

Stream a b bmergeS (lxs, rxs) = binary ([], [], [])
    (B step passT propT lkeep) (B step passT rprop rkeep)
    (lxs, rxs)
where step ts (lps, rps, psOut) = (ts, (lps, rps, psOut))
    rprop ps (lps, rps, psOut) = (setCombine lps (ps ++ rps)) \\ psOut
    lkeep ps (lps, rps, psOut) = (ps ++ lps, rps, psOut)
    rkeep ps (lps, rps, psOut) =
        (lps, ps ++ rps, psOut ++ (rprop ps (lps, rps, psOut)))

```

Theorem A.5 *The stream iterator mergeS is faithful and proper for the table operator merge.*

Proof: We use the following invariants for merge, where S_L and S_R are the input streams. Let $lts_i = data(S_L[i])$, $rts_i = data(S_R[i])$, $lps_i = puncts(S_L[i])$, and $rps_i = puncts(S_R[i])$:

$$\begin{aligned}
 cpass(lts_i, lps_i, rts_i, rps_i) &= lts_i ++ rts_i \\
 cprop(lts_i, lps_i, rts_i, rps_i) &= setCombine(lps_i, rps_i) \\
 ckeep_1(lts_i, lps_i, rts_i, rps_i) &= [] \\
 ckeep_2(lts_i, lps_i, rts_i, rps_i) &= []
 \end{aligned}$$

A.5.1 Faithfulness of the Invariants for Merge

By Theorem 6.1, every monotone table operator has a faithful stream counterpart. As $cpass(lts_i, lps_i, rts_i, rps_i) = lts_i ++ rts_i$, if we use $ckeep(lts_i, lps_i, rts_i, rps_i) = lts_i ++ rts_i$ we would have the version of merge per Theorem 6.1. We now have to show faithfulness if state is maintained per $ckeep(lts_i, lps_i, rts_i, rps_i) = []$.

Consider what is output between two points i and $j > i$:

$$\begin{aligned}
 cpass(lts_j, lps_j, rts_j, rps_j) - cpass(lts_i, lps_i, rts_i, rps_i) \\
 &= (lts_j ++ rts_j) - (lts_i ++ rts_i) \\
 &= (lts_{ij} ++ lts_i ++ rts_{ij} ++ rts_i) - (lts_i ++ rts_i) \\
 &= (lts_{ij} ++ rts_{ij} ++ lts_i ++ rts_i) - (lts_i ++ rts_i) \\
 &= lts_{ij} ++ rts_{ij}
 \end{aligned}$$

Thus, the output of merge does not depend on holding any previous data items in state. Therefore, maintaining state per $ckeep(lts_i, lps_i, rts_i, rps_i) = []$ does not affect faithfulness.

A.5.2 Propriety of the Invariants for Merge

By the propagation invariant for merge, the set of punctuations emitted at stage i is $setCombine(lps_i, rps_i)$. Suppose a data item t is emitted at some point j where $j > i$ such that $setMatch(t, setCombine(lps_i, rps_i))$. Then t is in $cpass(lts_j, lps_j, rts_j, rps_j)$. Therefore, it must be that $t \in lts_j$ or $t \in rts_j$. Let us suppose $t \in lts_j$. Since $setMatch(t, setCombine(lps_i, rps_i))$, there must exist some $lp \in lps_i$ and $rp \in rps_i$ such that $match(t, combine(lp, rp))$, which implies $match(t, lp)$ and $match(t, rp)$. Since $lp \in lps_i$ and $rp \in rps_i$, it must be that $t \in lts_i$, and thus $t \in cpass(lts_i, lps_i, rts_i, rps_i)$, and by the minimality condition will not be emitted after stage i . A similar argument holds if $t \in rts_j$. Thus the output of any iterator satisfying $cpass$ and $cprop$ for merge is grammatical given grammatical input, and is therefore proper.

A.5.3 Conformance of the Implementation of Merge to its Invariants

Now that we have shown that our invariants for merge are faithful and proper, we want to show that our implementation of `merge` adheres to those invariants.

As the output for merge is a bag, we use some of the algebraic properties of bag types per Albert [Alb91] in our proofs. Specifically, we will use the following notation and properties:

$x \in\in B$ = the number of copies of x in the bag B

$A \subseteq B$ = $\forall x \in B, (x \in\in A) \leq (x \in\in B)$

$A \setminus B$ = the bag C such that $\forall x \in A$,

$$x \in\in C = \max((x \in\in A - x \in\in B), 0)$$

Pass: The output at any stage i is the concatenation of $fst(lstep(lts^i, st_{i-1}))$, $lpass(lps^i, st_{i-1})$, $fst(rstep(rts^i, st_{i-1}))$, and $rpass(rps^i, st_{i-1})$, so we start from there.

$$\begin{aligned} &fst(lstep(lts^i, st_{i-1})) ++ lpass(lps^i, st_{i-1}) ++ fst(rstep(rts^i, st_{i-1})) ++ rpass(rps^i, st_{i-1}) \\ &= lts^i ++ [] ++ rts^i ++ [] \end{aligned}$$

$$= lts^i ++ rts^i$$

So we need to show that $lts^i ++ rts^i = cpass(lts_i, lps_i, rts_i, rps_i) - cpass(lts_{i-1}, lps_{i-1}, rts_{i-1}, rps_{i-1})$.

case left $\subseteq$ right:

$$\begin{aligned} x \in (lts^i ++ rts^i) &= (x \in (lts_i \setminus lts_{i-1}) ++ (rts_i \setminus rts_{i-1})) \\ &= x \in ((lts_i \setminus lts_{i-1})) + (x \in (rts_i \setminus rts_{i-1})) \\ &= \max((x \in lts_i - x \in lts_{i-1}), 0) + \\ &\quad \max((x \in rts_i - x \in rts_{i-1}), 0) \end{aligned}$$

Now, because $lts_{i-1} \subseteq lts_i$ and $rts_{i-1} \subseteq rts_i$, it must be that

$(x \in lts_i - x \in lts_{i-1}) \geq 0$ and $(x \in rts_i - x \in rts_{i-1}) \geq 0$. So, the previous expression becomes:

$$\begin{aligned} &(x \in lts_i - x \in lts_{i-1}) + (x \in rts_i - x \in rts_{i-1}) \\ &= x \in lts_i - x \in lts_{i-1} + x \in rts_i - x \in rts_{i-1} \\ &= x \in lts_i + x \in rts_i - x \in lts_{i-1} - x \in rts_{i-1} \\ &= (x \in lts_i + x \in rts_i) - (x \in lts_{i-1} + x \in rts_{i-1}) \\ &= (x \in lts_i ++ rts_i) - (x \in lts_{i-1} ++ rts_{i-1}) \\ &= (x \in cpass_i) - (x \in cpass_{i-1}) \end{aligned}$$

case right $\subseteq$ left:

$$\begin{aligned} (x \in cpass_i) - (x \in cpass_{i-1}) &= (x \in lts_i ++ rts_i) - (x \in lts_{i-1} ++ rts_{i-1}) \\ &= (x \in lts_i + x \in rts_i) - (x \in lts_{i-1} + x \in rts_{i-1}) \\ &= x \in lts_i + x \in rts_i - x \in lts_{i-1} - x \in rts_{i-1} \\ &= x \in lts_i - x \in lts_{i-1} + x \in rts_i - x \in rts_{i-1} \\ &= (x \in lts_i - x \in lts_{i-1}) + (x \in rts_i - x \in rts_{i-1}) \end{aligned}$$

As before, because $lts_{i-1} \subseteq lts_i$ and $rts_{i-1} \subseteq rts_i$, it must be that

$(x \in lts_i - x \in lts_{i-1}) \geq 0$ and $(x \in rts_i - x \in rts_{i-1}) \geq 0$. So, the previous expression becomes:

$$\begin{aligned} &= \max((x \in lts_i - x \in lts_{i-1}), 0) + \\ &\quad \max((x \in rts_i - x \in rts_{i-1}), 0) \\ &= (x \in (lts_i \setminus lts_{i-1})) + (x \in (rts_i \setminus rts_{i-1})) \end{aligned}$$

$$\begin{aligned}
&= (x \in (lts_i \setminus lts_{i-1}) ++ (rts_i \setminus rts_{i-1})) \\
&\quad x \in (lts^i ++ rts^i)
\end{aligned}$$

Prop:

We show that the punctuations emitted at any step i is equivalent to

$cprop(lts_i, lps_i, rts_i, rps_i) - cprop(lts_{i-1}, lps_{i-1}, rts_{i-1}, rps_{i-1})$. For shorthand, we will use $cprop_i = cprop(lts_i, lps_i, rts_i, rps_i)$.

At some stage i , the output punctuations will be the results of the functions $propT$ and $rprop$, as:

$$\begin{aligned}
&propT(lps^i, (lps_{i-1}, rps_{i-1}, cprop_{i-1})) ++ rprop(rps^i, (lps_i, rps_{i-1}, cprop_{i-1})) \\
&= [] ++ rprop(rps^i, (lps_i, rps_{i-1}, cprop_{i-1})) \\
&= rprop(rps^i, (lps_i, rps_{i-1}, cprop_{i-1})) \\
&= setCombine(lps_i, (rps^i ++ rps_{i-1})) \setminus \setminus cprop_{i-1} \\
&= setCombine(lps_i, rps_i) \setminus \setminus cprop_{i-1} \\
&= setCombine(lps_i, rps_i) - cprop_{i-1} \\
&= cprop_i - cprop_{i-1}
\end{aligned}$$

Keep:

Since the stream iterator for merge does not maintain data items in state and *ckeep* for merge is $[]$, the proof for keep is trivial. Note, however, that the implementation does maintain punctuations, and that there is no purging of those punctuations.

A.6 Stream Iterator for Intersect

We use the following implementation of intersect:

```

intersectS :: (Eq a, Eq b, Tuple a b) =>
    (Stream a b, Stream a b) -> Stream a b
intersectS = binary ([], [], [], [], [], []) (B lstep passT propT lkeep)
    (B rstep passT rprop rkeep)
    where lstep ts (lts,rts,tsOut,lps,rps,psOut) =

```

```

([], (ts++lts, rts, tsOut, lps, rps, psOut))
rstep ts (lts, rts, tsOut, lps, rps, psOut) =
  (newOut, (lts, ts++rts, nub(tsOut++newOut), lps, rps, psOut))
  where newOut = nub (lts 'intersect' (ts++rts)) \\ tsOut
rprop ps (lts, rts, tsOut, lps, rps, psOut) =
  (setCombine lps (ps++rps)) \\ psOut
lkeep ps (lts, rts, tsOut, lps, rps, psOut) =
  (lts, rts, tsOut, ps++lps, rps, psOut)
rkeep ps st@(lts, rts, tsOut, lps, rps, psOut) =
  ((setNomatchTs lts (ps ++ rps)), (setNomatchTs rts lps),
   tsOut, lps, ps++rps, psOut++(rprop ps st))
  where psBoth = setCombine lps (ps++rps)

```

The state maintained in `intersect` is a six-tuple $(lts, rts, tsOut, lps, rps, psOut)$, where: lts and rts are data items that have arrived from the left and right inputs, respectively, $tsOut$ are data items that have been output, lps and rps are punctuations that have arrived from the left and right inputs, respectively, and $psOut$ are punctuations that have been output.

Theorem A.6 *The stream iterator `intersectS` is faithful and proper for the table operator `intersect`.*

Proof: We use the following invariants for `intersect`, where lts_i , rts_i , lps_i , and rps_i are defined as before:

$$\begin{aligned}
 cpass(lts_i, lps_i, rts_i, rps_i) &= lts_i \cap rts_i \\
 cprop(lts_i, lps_i, rts_i, rps_i) &= setCombine(lps_i, rps_i) \\
 ckeep_1(lts_i, lps_i, rts_i, rps_i) &= setNomatchTs(lps_i, rps_i) \\
 ckeep_2(lts_i, lps_i, rts_i, rps_i) &= setNomatchTs(lps_i, rps_i)
 \end{aligned}$$

A.6.1 Faithfulness of the Invariants for Intersect

By Theorem 6.1, every monotone table operator has a faithful stream counterpart. Therefore, `intersect` has a faithful stream counterpart. However, the “standard” stream iterator

for intersect must keep all data items in state. We must consider a stream iterator where state is maintained per *ckeepL* and *ckeepR*.

Consider what is output from a stream iterator for intersect at some stage j later than stage i (assuming minimality):

$$\begin{aligned}
& (lts_j \cap rts_j) - (lts_i \cap rts_i) \\
&= (lts_j - (lts_i \cap rts_i)) \cap (rts_j - (lts_i \cap rts_i)) \\
&= ((lts_j - lts_i) \cup (lts_j - rts_i)) \cap ((rts_j - lts_i) \cup (rts_j - rts_i)) \\
&= (lts_j \cap rts_j) \cap (lts'_i \cup rts'_i) \\
&\quad \text{[where } ts' \text{ denotes data items not in } ts.] \\
&= ((lts_j \cap rts_j) \cap lts'_i) \cup ((lts_j \cap rts_j) \cap rts'_i) \\
&= (lts_j \cap lts'_i \cap rts_j) \cup (lts_j \cap rts_j \cap rts'_i) \\
&= ((lts_j - lts_i) \cap rts_j) \cup ((lts_j \cap (rts_j - rts_i))) \\
&= (lts_{ij} \cap rts_j) \cup (lts_j \cap rts_{ij}) \\
&= (lts_{ij} \cap (rts_i \cup rts_{ij})) \cup ((lts_i \cup lts_{ij}) \cap rts_{ij}) \\
&= ((lts_{ij} \cap rts_i) \cup (lts_{ij} \cap rts_{ij})) \cup ((lts_i \cap rts_{ij}) \cup (lts_{ij} \cap rts_{ij})) \\
&= (lts_{ij} \cap rts_i) \cup (lts_{ij} \cap rts_{ij}) \cup (lts_i \cap rts_{ij}) \cup (lts_{ij} \cap rts_{ij}) \\
&= (lts_{ij} \cap rts_i) \cup (lts_{ij} \cap rts_{ij}) \cup (lts_i \cap rts_{ij})
\end{aligned}$$

Thus, to emit the correct results at stage j , the standard stream iterator for intersect keeps all data items up until stage i . Suppose instead the stream iterator state is per *ckeepL*($lts_i, lps_i, rts_i, rps_i$) instead. Since *ckeepL*($lts_i, lps_i, rts_i, rps_i$) = *setNomatchTs*(lts_i, rps_i), we know that no data items in rts_{ij} will intersect with $lts_i - \text{ckeepL}(lts_i, lps_i, rts_i, rps_i)$ (that is, $rts_{ij} \cap lts_i - \text{ckeepL}(lts_i, lps_i, rts_i, rps_i) = \emptyset$). Therefore, $lts_i \cap rts_{ij} = \text{ckeepL}(lts_i, lps_i, rts_i, rps_i) \cap rts_{ij}$. A similar argument can be made for rts_i and *ckeepR*($lts_i, lps_i, rts_i, rps_i$). Therefore, $(lts_{ij} \cap rts_i) \cup (lts_{ij} \cap rts_{ij}) \cup (lts_i \cap rts_{ij}) = (lts_{ij} \cap \text{ckeepR}(lts_i, lps_i, rts_i, rps_i)) \cup (lts_{ij} \cap rts_{ij}) \cup (\text{ckeepL}(lts_i, lps_i, rts_i, rps_i) \cap rts_{ij})$, and the enhanced version of the stream iterator for intersect with *ckeepL* and *ckeepR* is faithful.

A.6.2 Propriety of the Invariants for Intersect

By the propagation invariant for intersect, the set of punctuations emitted at stage i is $setCombine(lps_i, lps_i)$. Suppose a data item t is emitted at some point j where $j > i$ such that $setMatch(t, setCombine(lps_i, rps_i))$. Then t is in $cpass(lts_j, lps_j, rts_j, rps_j)$. Therefore, it must be that $t \in lts_j$ and $t \in rts_j$. Since $setMatch(t, setCombine(lps_i, rps_i))$, there must exist some $lp \in lps_i$ and $rp \in rps_i$ such that $match(t, combine(lp, rp))$, which implies $match(t, lp)$ and $match(t, rp)$. Since $lp \in lps_i$ and $rp \in rps_i$, $t \in lts_i$ and $t \in rts_i$, and thus $t \in cpass(lts_i, lps_i, rts_i, rps_i)$, and by the minimality condition, will not be emitted after stage i . Thus the output of any iterator satisfying $cpass$ and $cprop$ for intersect is grammatical given grammatical input, and is therefore proper.

A.6.3 Conformance of the Implementation of Intersect to its Invariants

Now that we have shown that our invariants for intersect are faithful and proper, we want to show that our implementation of `intersectS` adheres to those invariants.

Pass:

We show that the list of data items output at any step i is equivalent to $cpass(lts_i, lps_i, rts_i, rps_i) - cpass(lts_{i-1}, lps_{i-1}, rts_{i-1}, rps_{i-1})$. We assert that the third list in state (referred to as $tsOut$ above) is equivalent to $cpass(lts_{i-1}, lps_{i-1}, rts_{i-1}, rps_{i-1})$, which can be easily shown by induction. The initial contents of $tsOut_0$ is $[]$. Each iteration i , $tsOut_{i-1}$ is catenated with the result of intersecting lts_i and rts_i and then duplicates are removed. Thus, $tsOut_i$ is equivalent to $cpass_i$. By a similar argument, $psOut_i$ is equivalent to $cprop_i$.

We start with what is output at each step i , namely

$$\begin{aligned}
 & lstep ++ passT ++ rstep ++ passT, \text{ as follows:} \\
 & fst(lstep(lts^i, (lts_{i-1}, rts_{i-1}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1}))) ++ \\
 & \quad passT(lps^i, (lts_i, rts_{i-1}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1})) ++ \\
 & \quad fst(rstep(rts^i, (lts_{i-1}, rts_{i-1}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1}))) ++ \\
 & \quad passT(rps^i, (lts_i, rts_i, tsOut_i, lps_i, rps_{i-1}, psOut_{i-1})) \\
 & = [] ++ [] ++
 \end{aligned}$$

$$\begin{aligned}
&fst(rstep(rts^i, (lts_{i-1}, rts_{i-1}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1}))) ++ [] \\
&= fst(rstep(rts^i, (lts_{i-1}, rts_{i-1}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1}))) \\
&= fst(rstep(rts^i, (lts_{i-1}, rts_{i-1}, cpass_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1}))) \\
&= nub(lts_i \text{ 'intersect' } (rts^i ++ rts_{i-1})) \setminus \setminus cpass_{i-1} \\
&= nub(lts_i \text{ 'intersect' } rts_i) \setminus \setminus cpass_{i-1} \\
&= (lts_i \cap rts_i) - cpass_{i-1} \\
&= cpass_i - cpass_{i-1}
\end{aligned}$$

Prop:

We show that the punctuations emitted at any step i is equivalent to $cprop(lts_i, lps_i, rts_i, rps_i) - cprop(lts_{i-1}, lps_{i-1}, rts_{i-1}, rps_{i-1})$. For shorthand, we will use $cprop_i = cprop(lts_i, lps_i, rts_i, rps_i)$.

We start with what is emitted at each step i , namely $lprop ++ rprop$, as follows:

$$\begin{aligned}
&lprop(lps^i, (lts_{i-1}, rts_{i-1}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, cprop_{i-1})) ++ \\
&\quad rprop(lts_{i-1}, rts_{i-1}, tsOut_{i-1}, rps^i, (lps_i, rps_{i-1}, cprop_{i-1})) \\
&= [] ++ rprop(rps^i, (lts_{i-1}, rts_{i-1}, tsOut_{i-1}, lps_i, rps_{i-1}, cprop_{i-1})) \\
&= rprop(rps^i, (lts_{i-1}, rts_{i-1}, tsOut_{i-1}, lps_i, rps_{i-1}, cprop_{i-1})) \\
&= setCombine(lps_i, (rps^i ++ rps_{i-1})) \setminus \setminus cprop_{i-1} \\
&= setCombine(lps_i, rps_i) \setminus \setminus cprop_{i-1} \\
&= setCombine(lps_i, rps_i) - cprop_{i-1} \\
&= cprop_i - cprop_{i-1}
\end{aligned}$$

Keep:

For each iteration i , state goes through four phases, which we will denote with superscripts. We denote the data items held in state from the left input at stage i as $\overline{lts_i}$. Likewise, we denote data items held in state from the right input at stage i as $\overline{rts_i}$. Note that $\overline{lts_i} \subseteq lts_i$ and $\overline{rts_i} \subseteq rts_i$. Each phase of state at stage i is:

$$\begin{aligned}
st_i^1 &= snd(lstep(lts^i, (\overline{lts_{i-1}}, \overline{rts_{i-1}}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1}))) \\
&= snd([], (lts^i ++ \overline{lts_{i-1}}, \overline{rts_{i-1}}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1})) \\
&= (lts^i ++ \overline{lts_{i-1}}, \overline{rts_{i-1}}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1})
\end{aligned}$$

$$\begin{aligned}
st_i^2 &= lkeep(lps^i, st_i^1) \\
&= lkeep(lps^i, (lts^i ++ \overline{lts_{i-1}}, \overline{rts_{i-1}}, tsOut_{i-1}, lps_{i-1}, rps_{i-1}, psOut_{i-1})) \\
&= (lts^i ++ \overline{lts_{i-1}}, \overline{rts_{i-1}}, tsOut_{i-1}, lps^i ++ lps_{i-1}, rps_{i-1}, psOut_{i-1}) \\
&= (lts^i ++ \overline{lts_{i-1}}, \overline{rts_{i-1}}, tsOut_{i-1}, lps_i, rps_{i-1}, psOut_{i-1})
\end{aligned}$$

$$\begin{aligned}
st_i^3 &= snd(rstep(rts^i, st_i^2)) \\
&= snd(rstep(rts^i, (lts^i ++ \overline{lts_{i-1}}, \overline{rts_{i-1}}, tsOut_{i-1}, lps_i, rps_{i-1}, psOut_{i-1}))) \\
&= snd(tsOut^i, (lts^i ++ \overline{lts_{i-1}}, rts^i ++ \overline{rts_{i-1}}, \\
&\quad tsOut^i ++ tsOut_{i-1}, lps_i, rps_{i-1}, psOut_{i-1})) \\
&= (lts^i ++ \overline{lts_{i-1}}, rts^i ++ \overline{rts_{i-1}}, tsOut_i, lps_i, rps_{i-1}, psOut_{i-1})
\end{aligned}$$

$$\begin{aligned}
st_i^4 &= rkeep(rps^i, st_i^3) \\
&= rkeep(rps^i, (lts^i ++ \overline{lts_{i-1}}, rts^i ++ \overline{rts_{i-1}}, tsOut_i, lps_i, rps_{i-1}, psOut_{i-1})) \\
&= (setNomatchTs(lts^i ++ \overline{lts_{i-1}}, rps^i ++ rps_{i-1}), \\
&\quad setNomatchTs(rts^i ++ \overline{rts_{i-1}}, lps_i), tsOut_i, lps_i, rps^i ++ rps_{i-1}, psOut_{i-1})) \\
&= (setNomatchTs(lts^i ++ \overline{lts_{i-1}}, rps_i), setNomatchTs(rts^i ++ \overline{rts_{i-1}}, lps_i), \\
&\quad tsOut_i, lps_i, rps_i, psOut_{i-1}))
\end{aligned}$$

Thus, our goal is to show that the first member of st_i^4 is equivalent to $ckeepl_i$ and the second member of st_i^4 is equivalent to $ckeepr_i$. That is, $setNomatchTs(lts^i ++ \overline{lts_{i-1}}, rps_i) = ckeeepl_i$ and $setNomatchTs(rts^i ++ \overline{rts_{i-1}}, lps_i) = ckeeepr_i$. We will show these equivalences by induction. We show the proof for the left input. The proof for the right input is similar.

Base case ($i = 1$):

$$\begin{aligned}
setNomatchTs(lts^1 ++ \overline{lts_0}, rps_1) &= setNomatchTs(lts^1 ++ [], rps_1) \\
&= setNomatchTs(lts^1, rps_1) \\
&= setNomatchTs(lts_1, rps_1) \\
&= ckeeepl_1
\end{aligned}$$

Induction step: Assume $setNomatchTs(lts^{i-1} ++ \overline{lts_{i-2}}, rps_{i-1}) = ckeepL_{i-1}$.

Prove $setNomatchTs(lts^i ++ \overline{lts_{i-1}}, rps_i) = ckeepL_i$

$$\begin{aligned}
setNomatchTs(lts^i ++ \overline{lts_{i-1}}, rps_i) &= setNomatchTs(lts^i ++ ckeepL_{i-1}, rps_i) \\
&= setNomatchTs(lts^i ++ setNomatchTs(lts_{i-1}, rps_{i-1}), rps_i) \\
&= setNomatchTs(lts^i ++ [u | u \in lts_{i-1} \wedge setNomatch(u, rps_{i-1})], rps_i) \\
&= [t | t \in (lts^i ++ [u | u \in lts_{i-1} \wedge setNomatch(u, rps_{i-1})]) \wedge setNomatch(t, rps_i)] \\
&= [t | t \in (lts^i ++ [u | u \in lts_{i-1}]) \wedge setNomatch(t, rps_i)] \\
&\quad [\text{since } setNomatch(t, rps_i) \text{ is more restrictive than } setNomatch(u, rps_{i-1})] \\
&= [t | t \in (lts^i ++ lts_{i-1}) \wedge setNomatch(t, rps_i)] \\
&= [t | t \in lts_i \wedge setNomatch(t, rps_i)] \\
&= setNomatchTs(lts_i, rps_i) \\
&= ckeepL_i
\end{aligned}$$

End of proof.

A.7 Stream Iterator for Join

Our implementation of join is similar to a symmetric hash join, with the exception that data items are stored in lists rather than hash tables. Join takes six functions: The `combT` function joins a list of data items from each input, and `combP` joins a list of punctuations from each input. The functions `jOfL` and `jOfR` return the projection of the appropriate input data item on the join attributes. The functions `grpL` and `grpR` return punctuations that describe the join attributes based on punctuations that have arrived. The `lstep` and `rstep` functions output data items from one input that join with data items in state from the other input. The `lprop` and `rprop` functions emit the result of joining input punctuations. Each function joins punctuations in the current slice from its input with punctuations that have arrived from the other input. The `lkeep` and `rkeep` functions remove data items from one input that match punctuations that describe the join attributes from the other input. The state is a four-tuple that maintains data items and punctuations from each input.

```
joinS :: (Tuple a b, Tuple c d, Tuple e f, Tuple j k) =>
```

```

([a] -> [c] -> [e]) -> ([b] -> [d] -> [f]) ->
(a -> j) -> (c -> j) -> ([b] -> [k]) -> ([d] -> [k]) ->
(Stream a b, Stream c d) -> Stream e f
joinS combT combP jOfL jOfR grpL grpR =
  binary ([], [], [], []) (B lstep passT lprop lkeep)
    (B rstep passT rprop rkeep)
  where lstep ts (lts, rts, lps, rps) = (combT ts rts, (lts++ts, rts, lps, rps))
        rstep ts (lts, rts, lps, rps) = (combT lts ts, (lts, rts++ts, lps, rps))
        lprop ps (lts, rts, lps, rps) = combP ps rps
        rprop ps (lts, rts, lps, rps) = combP lps ps
        lkeep ps (lts, rts, lps, rps) = (ltsNew, rts, lps++ps, rps)
          where ltsNew = [t | t <- lts, setNomatch (jOfL t) (grpR rps)]
        rkeep ps (lts, rts, lps, rps) = (lts, rtsNew, lps, rps++ps)
          where rtsNew = [t | t <- rts, setNomatch (jOfR t) (grpL lps)]

```

Theorem A.7 *The stream iterator joinS is faithful and proper for the table operator join.*

Proof: Let the function *schema* take a stream as input and return the set of attributes for that stream. The punctuation invariants for join are listed below, where lts_i , rts_i , lps_i , and rps_i are defined as before. Let $I_L = \text{schema}(lts_i)$ and $I_R = \text{schema}(rts_i)$ are the schemas for the input streams, and J is the set of join attributes. For simplicity, we will assume that $J = I_L \cap I_R$:

$$\begin{aligned}
cpass(lts_i, lps_i, rts_i, rps_i) &= lts_i \bowtie rts_i \\
cprop(lts_i, lps_i, rts_i, rps_i) &= lps_i \bowtie rps_i \\
ckeepp_1(lts_i, lps_i, rts_i, rps_i) &= [t | t \in lts_i \wedge \text{setNomatch}(\pi_J(t), \text{groupP}(J, rps_i))] \\
ckeepp_2(lts_i, lps_i, rts_i, rps_i) &= [t | t \in rts_i \wedge \text{setNomatch}(\pi_J(t), \text{groupP}(J, lps_i))]
\end{aligned}$$

As in the case of group-by, we use the function *groupP* to decide which data items should remain in state. The propagation function for join simply joins punctuations based on the pattern values of the join attributes. For example, consider the input schemas $S_L(A, B)$ and $S_R(B, C, D)$. Suppose punctuation $lp = \langle 5, [8, 10] \rangle$ arrives on S_L and the punctuations $rp_1 = \langle [5, 7], 100, * \rangle$ and $rp_2 = \langle [7, 10], *, 50 \rangle$. Since the pattern $[8, 10]$ and $[5, 7]$ do not overlap, there can be no output from joining lp and rp_1 . However, the

patterns $[8, 10]$ and $[7, 10]$ do overlap, and so lp and rp_2 can be joined, and the result of joining them would be $\langle 5, [8, 10], *, 50 \rangle$.

A.7.1 Faithfulness of the Invariants for Join

By Theorem 6.1, every monotone table operator g has a faithful stream counterpart. Therefore, join has a faithful stream counterpart. However, the “standard” stream iterator for join must keep all data items that have arrived from each input in state. We must consider any stream iterator where state is maintained per $ckee_1$ and $ckee_2$.

First, consider the output of join at some stage k later than stage i ($k > i$):

$$\begin{aligned}
 & cpass(lts_k, lps_k, rts_k, rps_k) - cpass(lts_i, lps_i, rts_i, rps_i) \\
 &= (lts_k \bowtie rts_k) - (lts_i \bowtie rts_i) \\
 &= ((lts_{ik} ++ lts_i) \bowtie (rts_{ik} ++ rts_i)) - (lts_i \bowtie rts_i) \\
 &= ((lts_{ik} \bowtie rts_{ik}) ++ (lts_{ik} \bowtie rts_i) ++ (lts_i \bowtie rts_{ik}) ++ (lts_i \bowtie rts_i)) - (lts_i \bowtie rts_i) \\
 &= (lts_{ik} \bowtie rts_{ik}) ++ (lts_{ik} \bowtie rts_i) ++ (lts_i \bowtie rts_{ik})
 \end{aligned}$$

Thus, the state maintained for each input holds past data items from one input to see if they join with new data arrived from the other input. Now, suppose state is maintained per $ckee_1$ and $ckee_2$. Let $ckee_1[i] = ckee_1(lts_i, lps_i, rts_i, rps_i)$ and $ckee_2[i] = ckee_2(lts_i, lps_i, rts_i, rps_i)$:

$$\begin{aligned}
 & (lts_{ik} \bowtie rts_{ik}) ++ (lts_{ik} \bowtie ckee_2[i]) ++ (ckee_1[i] \bowtie rts_{ik}) \\
 &= (lts_{ik} \bowtie rts_{ik}) ++ (lts_{ik} \bowtie [t | t \in rts_i \wedge setNomatch(\pi_J(t), groupP(J, lps_i))]) \\
 &\quad ++ ([t | t \in lts_i \wedge setNomatch(\pi_J(t), groupP(J, rps_i))] \bowtie rts_{ik})
 \end{aligned}$$

Consider $[t | t \in rts_i \wedge setNomatch(\pi_J(t), groupP(J, lps_i))]$. As $groupP$ only returns those punctuations that describe the join attributes, it must be that $setMatchTs(\pi_J(lts_{ik}), groupP(J, lps_i)) = []$. That is, no data items will arrive after stage i from the left input with values for the join attributes that match punctuations from $groupP(J, lps_i)$. Thus, data items from the right input with join values that match $groupP(J, lps_i)$ will never join with data items from the left input that arrive after stage

i. That is, $lts_{ik} \bowtie [t|t \in rts_i \wedge setMatch(\pi_J(t), groupP(J, lps_i))] = []$. As this result is empty, it must be that:

$$\begin{aligned}
lts_{ik} &\bowtie [t|t \in rts_i \wedge setNomatch(\pi_J(t), groupP(J, lps_i))] \\
&= lts_{ik} \bowtie ([t|t \in rts_i \wedge setNomatch(\pi_J(t), groupP(J, lps_i))] ++ []) \\
&= lts_{ik} \bowtie ([t|t \in rts_i \wedge setNomatch(\pi_J(t), groupP(J, lps_i))] ++ \\
&\quad [t|t \in rts_i \wedge setMatch(\pi_J(t), groupP(J, lps_i))]) \\
&= lts_{ik} \bowtie [t|t \in rts_i \wedge (setNomatch(\pi_J(t), groupP(J, lps_i)) \vee \\
&\quad setMatch(\pi_J(t), groupP(J, lps_i)))] \\
&= lts_{ik} \bowtie [t|t \in rts_i \wedge true] \\
&= lts_{ik} \bowtie [t|t \in rts_i] \\
&= lts_{ik} \bowtie rts_i
\end{aligned}$$

A similar argument holds for $[t|t \in lts_i \wedge setMatch(\pi_J(t), groupP(J, rps_i))] \bowtie rts_{ik}$.

Thus:

$$\begin{aligned}
&(lts_{ik} \bowtie rts_{ik}) ++ (lts_{ik} \bowtie [t|t \in rts_i \wedge setNomatch(\pi_J(t), groupP(J, lps_i))]) \\
&\quad ++ ([t|t \in lts_i \wedge setNomatch(\pi_J(t), groupP(J, rps_i))] \bowtie rts_{ik}) \\
&= (lts_{ik} \bowtie rts_{ik}) ++ (lts_{ik} \bowtie rts_i) ++ (lts_i \bowtie rts_{ik})
\end{aligned}$$

Since this equation is equal to the value for the “standard” version of join, maintaining state per $ckep_1$ and $ckep_2$ does not affect faithfulness.

A.7.2 Propriety of the Invariants for Join

By the propagation invariant for join, the set of punctuations emitted at stage i is $lps_i \bowtie rps_i$ for join attributes J . Suppose a data item $t = lt \bowtie rt$ is emitted at some point k where $k > i$ such that $setMatch(lt, lps_i)$ and $setMatch(rt, rps_i)$. Then t is in $cpass(lts_k, lps_k, rts_k, rps_k)$. Therefore, it must be that $lt \in lts_k$ and $rt \in rts_k$. Since $setMatch(lt, lps_i)$ and $setMatch(rt, rps_i)$, there must exist some $lp \in lps_i$ and $rp \in rps_i$ such that $match(lt, lp)$ and $match(rt, rp)$. Since $lp \in lps_i$ and $rp \in rps_i$, it must be that $lt \in lts_i$ and $rt \in rts_i$ by grammaticality of the inputs. Therefore,

$t \in cpass(lts_i, lps_i, rts_i, rps_i)$, and by the minimality condition, t will not be emitted after stage i . Thus the output of any iterator satisfying *cpass* and *cprop* for join is grammatical given grammatical input, and is therefore proper.

A.7.3 Conformance of the Implementation of Join to its Invariants

Now that we have shown that our invariants for join are faithful and proper, we want to show that our implementation of `joinS` adheres to the invariants.

Pass:

We show that the list of data items output at any step i is equivalent to $cpass(lts_i, lps_i, rts_i, rps_i) - cpass(lts_{i-1}, lps_{i-1}, rts_{i-1}, rps_{i-1})$. We assume for now that all data items are held in state. Later we will show that our keep function conforms to *ckeeper*.

We start with the output at some step i , namely:

$lstep++lpass++rstep++rpass$, as follows:

$$\begin{aligned}
&fst(lstep(lts^i, (lts_{i-1}, rts_{i-1}, lps_{i-1}, rps_{i-1}))) ++ passT ++ \\
&\quad fst(rstep(rts^i, (lts_{i-1}, rts_{i-1}, lps_{i-1}, rps_{i-1}))) ++ passT \\
&= fst(lstep(lts^i, (lts_{i-1}, rts_{i-1}, lps_{i-1}, rps_{i-1}))) ++ [] ++ \\
&\quad fst(rstep(rts^i, (lts_i, rts_{i-1}, lps_i, rps_{i-1}))) ++ [] \\
&= fst(lstep(lts^i, (lts_{i-1}, rts_{i-1}, lps_{i-1}, rps_{i-1}))) ++ \\
&\quad fst(rstep(rts^i, (lts_i, rts_{i-1}, lps_i, rps_{i-1}))) \\
&= combT(lts^i, rts_{i-1}) ++ combT(lts_i, rts^i) \\
&= (lts^i \bowtie rts_{i-1}) ++ (lts_i \bowtie rts^i) \\
&= (lts^i \bowtie rts_{i-1}) ++ ((lts^i ++ lts_{i-1}) \bowtie rts^i) \\
&= (lts^i \bowtie rts_{i-1}) ++ (lts^i \bowtie rts^i) ++ (lts_{i-1} \bowtie rts^i) \\
&= cpass_i - cpass_{i-1} \text{ from the equality given earlier.}
\end{aligned}$$

Prop:

Punctuations output at some step i are the catenation of *lprop* and *rprop*:

$$lprop(lps^i, (lts_{i-1}, rts_{i-1}, lps_{i-1}, rps_{i-1})) ++ rprop(rps^i, (lts_i, rts_{i-1}, lps_i, rps_{i-1}))$$

$$\begin{aligned}
&= \text{comb}P(lps^i, rps_{i-1}) ++ \text{comb}P(lps_i, rps^i) \\
&= (lps^i \bowtie rps_{i-1}) ++ (lps_i \bowtie rps^i) \\
&= (lps^i \bowtie rps_{i-1}) ++ ((lps^i ++ lps_{i-1}) \bowtie rps^i) \\
&= (lps^i \bowtie rps_{i-1}) ++ (lts^i \bowtie rps^i) ++ (lts_{i-1} \bowtie rps^i) \\
&= \text{cprop}_i - \text{cprop}_{i-1} \text{ using a similar argument as for } \text{cpass} \text{ earlier.}
\end{aligned}$$

Keep:

As for intersect, we need to first work through the the four phases that state goes through during each step. We will use $\text{ckeeper}_1[i] = \text{ckeeper}_1(lts_i, lps_i, rts_i, rps_i)$ and $\text{ckeeper}_2[i] = \text{ckeeper}_2(lts_i, lps_i, rts_i, rps_i)$. We want to show that, after the four phases, the first member of state at any stage i is equivalent to $\text{ckeeper}_1[i]$ and the second member of state at any stage i is equivalent to $\text{ckeeper}_2[i]$. As before, we will denote each phase of state k for some stage i as st_i^k . Also, we denote the data items held in state from the left input at stage i as $\overline{lts_i}$. Likewise, we denote data items held in state from the right input at stage i as $\overline{rts_i}$.

$$\begin{aligned}
st_i^1 &= \text{snd}(\text{lstep}(lts^i, (\overline{lts_{i-1}}, \overline{rts_{i-1}}, lps_{i-1}, rps_{i-1}))) \\
&= (\overline{lts_{i-1}} ++ lts^i, \overline{rts_{i-1}}, lps_{i-1}, rps_{i-1}) \\
\\
st_i^2 &= \text{lkeep}(lps^i, st_i^1) \\
&= \text{lkeep}(lps^i, (\overline{lts_{i-1}} ++ lts^i, \overline{rts_{i-1}}, lps_{i-1}, rps_{i-1})) \\
&= (\overline{lts_{i-1}} ++ lts^i, \overline{rts_{i-1}}, lps_{i-1} ++ lps^i, rps_{i-1}) \\
&= (\overline{lts_{i-1}} ++ lts^i, \overline{rts_{i-1}}, lps_i, rps_{i-1}) \\
\\
st_i^3 &= \text{snd}(\text{rstep}(rts^i, st_i^2)) \\
&= \text{snd}(\text{rstep}(rts^i, (\overline{lts_{i-1}} ++ lts^i, \overline{rts_{i-1}}, lps_i, rps_{i-1}))) \\
&= (\overline{lts_{i-1}} ++ lts^i, \overline{rts_{i-1}} ++ rts^i, lps_i, rps_{i-1}) \\
\\
st_i^4 &= \text{rkeep}(rps^i, st_i^3) \\
&= \text{rkeep}(rps^i, (\overline{lts_{i-1}} ++ lts^i, \overline{rts_{i-1}} ++ rts^i, lps_i, rps_{i-1})) \\
&= ([t | t \in \overline{lts_{i-1}} ++ lts^i \wedge \text{setNomatch}(\pi_J(t), \text{group}P(J, rps_i))],
\end{aligned}$$

$$[t | t \in \overline{rts_{i-1}} ++ rts^i \wedge setNomatch(\pi_J(t), groupP(J, lps_i))], lps_i, rps_i)$$

Now, for $t \in lts_{i-1} - \overline{lts_{i-1}}$, it must be that $setNomatch(\pi_J(t), groupP(J, rps_{i-1})) = false$. Since $rps_{i-1} \subseteq rps_i$, it must also be that $setNomatch(\pi_J(t), groupP(J, rps_i)) = false$. Therefore, $[t | t \in \overline{lts_{i-1}} ++ lts^i \wedge setNomatch(\pi_J(t), groupP(J, rps_i))] = [t | t \in lts_i \wedge setNomatch(\pi_J(t), groupP(J, rps_i))]$, and therefore $[t | t \in \overline{lts_{i-1}} ++ lts^i \wedge setNomatch(\pi_J(t), groupP(J, rps_i))] = ckeep_1[i]$. We can use a similar argument for $[t | t \in \overline{rts_{i-1}} ++ rts^i \wedge setNomatch(\pi_J(t), groupP(J, lps_i))]$. Therefore, the first member maintained in state conforms to $ckeeper_1[i]$ and the second member maintained in state conforms to $ckeeper_2[i]$, and we have that our implementation of the stream iterator for join conforms to its punctuation invariants.

End of proof.

Appendix B

Source Code for the Online Auction Management System

In this appendix we give implementations of the online auction queries in Haskell. This implementation is a model of those used in the performance testing (see Chapter 8).

B.1 Data Types for the Auction Scenario

The first section of code contains the various types used for the incoming data items from each of the three streams (Person, Auction, Bid).

```
-----  
--Online Auction Monitoring Scenario  
--Auction data types  
-----  
  
type AuctionState = (A_Step,A_Seed,A_Hour,A_Minute,[A_Person],  
                    [(A_Auction,A_Bid,Bool)])  
  
type A_BidReport = (A_Hour,A_Minute,A_Auction,A_Person,A_Bid)  
type A_BidPunc = (Pattern A_Hour,Pattern A_Minute,Pattern A_Auction,  
                 Pattern A_Person,Pattern A_Bid)  
  
type A_PersonReport = (A_Person, A_Name, A_Name)  
type A_PersonPunc = (Pattern A_Person, Pattern A_Name, Pattern A_Name)  
type A_AuctionReport = (A_Auction, A_Category)  
type A_AuctionPunc = (Pattern A_Auction, Pattern A_Category)  
  
type A_Step = Int      --How many iterations have we gone through?  
type A_Seed = Int      --What was the seed from the Random call
```

```

type A_Hour = Int      --The current hour
type A_Minute = Int    --The current minute
type A_Person = Int    --Person identifier
type A_Auction = Int   --Auction identifier
type A_Bid = Int       --Price of the current auction bid
type A_Name = String   --Person's name in the auction
type A_Category = Int  --Category id for an auction

```

B.2 Implementation of Queries in the Online Auction Scenario

Now we discuss the Haskell implementation of the five queries used in the online auction scenario in our stream iterator framework. The implementation of the auction streams is discussed in the next section.

B.2.1 Query 1 — Currency Conversion

In this query we use only the project stream iterator to convert price values in bids from US Dollars to Euros. The function `dol2eur` converts prices for values of data items. The `dol2eurP` function converts price patterns in punctuations. Finally, `pfilter` filters out punctuations in a manner similar to the `describe` operator. Since the minute and auction id attributes are being projected away, the patterns for those attributes in a punctuation must be the wildcard.

```

-----
--Query 1: Currency conversion
--  SELECT bidder, hour, DOLTOEUR(price)
--  FROM bid1;
-----

qryCurrConv :: Stream (A_Person, A_Bid, A_Hour)
              (Pattern A_Person,Pattern A_Bid,Pattern A_Hour)
qryCurrConv = projectS dol2eur dol2eurP bidStream2
  where dol2eur (h,m,a,p,b) = (p,(b*2),h)
        dol2eurP ps =

```

```

map convP (filter \(ph,pm,pa,pp,pb) -> pm==Wildcard &&
                                pa==Wildcard) ps)
convP (ph,_,_,pp,Literal pb) = (pp,Literal (2*pb),ph)
convP (ph,_,_,pp,Range (pb1,pb2)) = (pp,Range (2*pb1,2*pb2),ph)
convP (ph,_,_,pp,Wildcard) = (pp,Wildcard,ph)
convP (ph,_,_,pp,ListPat pbs) = (pp,ListPat (map (2*) pbs),ph)

```

B.2.2 Query 2 — Specific Bid Ranges

In this query we again use the project stream iterator to project only the auction id and price attributes. In addition, the select iterator filters out bids outside a specific price range. In our performance tests, the price range was between 350 and 450.

The `aid_price` function projects out only the desired attributes, and the `pfilter` function behaves in a manner similar as before, keeping only those punctuations that describe the projection attributes. The `highprices` function is the predicate function passed to the select iterator to decide which data items to keep in the output stream.

```

-----
--Query 2: Specific Bid Ranges

```

```

-- SELECT a_id, price
-- FROM bid1
-- WHERE price>=350 AND price<=450;
-----

```

```

qryCategories :: Stream (A_Auction, A_Bid) (Pattern A_Auction,Pattern A_Bid)

```

```

qryCategories =

```

```

    projectS aid_price aid_priceP (selectS highprices bidStream2)
  where aid_price (h,m,a,p,b) = (a,b)
        aid_priceP ps =
          map aid_price
            (filter \(ph,pm,pa,pp,pb) -> ph == Wildcard &&
                                pm == Wildcard &&
                                pp == Wildcard) ps)
        highprices (h,m,a,p,b) = (b>=350 && b<=450)

```

B.2.3 Query 3 — Bid Counts

In this query, we want the number of bids each hour. We use the aggregate stream iterator to group data items on the hour attribute (`h`) using the function `grp`. The `grpP` function filters out punctuations that do not describe the grouping attribute. The aggregate function used is the built-in `countStream` function.

```
-----
--Query 3: Bid Counts
--    SELECT hour, COUNT(*)
--    FROM bid1
--    GROUP BY hour;
-----

qryBidCount :: Stream (A_Hour, Int) (Pattern A_Hour, Pattern Int)
qryBidCount = groupbyS grp grpP countS bidStream2
  where grp ((h,m,a,p,b):ts) = h
        grpP ps = map (\(ph,pm,pa,pp,pb) -> ph) (filter descHr ps)
        descHr (ph,pm,pa,pp,pb) = pm==Wildcard && pa==Wildcard &&
                                   pp==Wildcard && pb==Wildcard
```

B.2.4 Query 4 — Closing Price for Auctions in Specific Categories

In this final query, we use an aggregate iterator over the output of the join iterator. The join iterator joins data items from the bid stream with data items from the auction stream (with specific category values). The `comb` function defines how items from the auction stream and the bid stream should be joined together. The `grp` function defines the attribute that we are grouping on, and the `grpP` function is used to filter out those punctuations that do not describe the grouping attribute. Finally, the `val` function returns the value that is being aggregated over. The aggregate function is the built-in `maxStream` function.

```
-----
--Query 4: Closing price
--    SELECT B.a_id, MAX(B.price)
--    FROM auction A, bid1 B
```

```

-- WHERE A.a_id=B.a_id AND A.category IN {92, 136, 208, 294}
-- GROUP BY B.a_id;
-----

qryClosePrice :: Stream (A_Auction, Int) (Pattern A_Auction, Pattern Int)
qryClosePrice = groupbyS grp grpP (maxS val)
                (joinSL comb comb jOfL jOfR grpL grpR
                 ((selectS mycategory auctionStream2), bidStream2))
where mycategory (i,c) = (c==92 || c==136 || c==208 || c==294)
      comb ls rs =
        [(i,c,h,m,p,b) | (i,c) <- ls, (h,m,a,p,b) <- rs, i==a]
      jOfL (i,c) = i
      jOfR (h,m,a,p,b) = a
      grpL ps = map fst (filter (\(pi,pc) -> pc == Wildcard) ps)
      grpR ps = map getA
                (filter (\(ph,pm,pa,pp,pb) -> ph == Wildcard &&
                                                pm == Wildcard &&
                                                pp == Wildcard &&
                                                pb == Wildcard) ps)
      getA (ph,pm,pa,pp,pb) = pa
      grp ((i,c,h,m,p,b):ts) = i
      grpP ps = map (\(pi,pc,ph,pm,pp,pb) -> pi) (filter descrI ps)
      descrI (pi,pc,ph,pm,pp,pb) = pc==Wildcard && ph==Wildcard &&
                                     pm==Wildcard && pp==Wildcard &&
                                     pb==Wildcard

      val (i,c,h,m,p,b) = b

```

B.2.5 Query 5 — Union of Bid Counts

As with Query 3, we want the number of bids each hour. However, this time we are taking the union of five input bid streams, rather than reading from only one bid stream.

```

-----
--Query 5: Union of Bid Counts
-- SELECT hour, COUNT(*)
-- FROM (SELECT * FROM bid1

```

```

--          UNION ALL
--          ...
--          UNION ALL
--          SELECT * FROM bid5)
--      GROUP BY hour;
-----

qryUnionBidCount :: Stream (A_Hour, Int) (Pattern A_Hour, Pattern Int)
qryUnionBidCount =
    groupbyS grp grpP countS
        (unionS [bidStream2, bidStream2, bidStream2, bidStream2, bidStream2])
    where grp ((h,m,a,p,b):ts) = h
          grpP ps = map (\(ph,pm,pa,pp,pb) -> ph) (filter descHr ps)
          descHr (ph,pm,pa,pp,pb) = pm==Wildcard && pa==Wildcard &&
                                     pp==Wildcard && pb==Wildcard

```

B.3 Implementation of Online Auction Streams

Here we give the implementation for our online auction streams. The function `newstates` generates a stream of auction states. Each item in the stream contains an updated auction state. Each item contains the current iteration, a randomly generated number, an hour and minute, a list of registered persons, and a list of auctions with the current bid price for each auction and a flag on whether or not the auction is closed.

We use this stream of auction states to generate the three streams used in the auction. We generate data items in the person stream by comparing two consecutive auction state items to see if a new person was added to the list of registered persons. If a new person exists, his or her id value is taken from the state, all other information is generated randomly and output as a slice in the person stream. If not, then an empty slice is output. The items in the auction stream are generated in a similar fashion. The data items in the bid stream are generated by comparing the auction lists in consecutive states. Any items with updated bids are added to a slice, and output to the bid stream.

Punctuations are added to the auction stream by comparing the auction lists from two states for auctions whose closed flag has changed from `False` to `True`. For each item, a

punctuation with a constant value for the id is output, indicating that no more auctions with that auction id will exist in the stream. Similarly, we add punctuations to the bid stream when the hour value has changed in consecutive state items.

```
-----
--Auction State Stream
-----
```

```
newstates :: [AuctionState]
newstates = (0,27063,0,0,[5,4,3,2,1],[]) : (map newstate newstates)

newstate :: AuctionState -> AuctionState
newstate (s,d,h,m,ps,abs) = (s+1,d',h',m',newps,newabs)
  where m' = (m+mincr) 'mod' 60
        h' = if (m+mincr) > 59 then h+1 else h
        d' = random d
        mincr = if ((random d') 'mod' 5) == 0 then 1 else 0
        newps = if ((random d') 'mod' 10) == 0 then newperson : ps else ps
        newabs = if ((random (d'-17)) 'mod' (1 + length (openabs abs))) == 0 then
                    newab : abs
                    else (updateabs 0 ((random d') 'mod' length abs) abs)
        newperson = 1 + length ps
        openabs abs = filter (\(a,b,c) -> c == False) abs
        newab = (1+length abs, 20, False)
        updateabs i n [] = []
        updateabs i n ((a,b,c):abs') =
            if (i < n || c == True) then (a,b,c):updateabs (i+1) n abs'
            else if ((random d') 'mod' 17) == 0 then (a,b,True):abs'
                  else (a,b+1+((random d') 'mod' 3),c):abs'
```

```
-----
--Auction Data Streams
-----
```

```
--BID stream
-- BID(hour,minute,auction,person,bid)
bidStream2 :: Stream A_BidReport A_BidPunc
```

```

bidStream2 = addHourPuncts 0 (bid2 newstates)
  where addHourPuncts lasthour (xs:xss) =
    if (any (hourchanged lasthour) xs) then
      xs:[punct(lit lasthour,wc,wc,wc,wc)]:addHourPuncts (lasthour+1) xss
    else xs : addHourPuncts lasthour xss
    hourchanged lasthour (Left (h,m,a,p,b)) = h > lasthour
    hourchanged _ _ = False

bid2 :: [AuctionState] -> Stream A_BidReport A_BidPunc
bid2 ((s,d,h,m,ps,abs):a2@(s',d',h',m',ps',abs')):as) =
  if (a == -1) then [] : bid2 (a2:as)
  else if (c == False) then [norm (h',m',a,p,b)] : bid2 (a2:as)
  else [punct (wc,wc,lit a,wc,wc)] : bid2 (a2:as)
  where (a,b,c) = diff abs abs'
        p = ps' !! ((random s') 'mod' (length ps'))
  diff [] [] = (-1,-1,False)
  diff [] (x:xs) = x
  diff (x:xs) (y:ys) = if x /= y then y else diff xs ys

--PERSON stream
-- PERSON(id,FName,LName)
fnames :: [A_Name]
fnames = ["Dave","Lois","Juliana","Kristin","Shawn","Pete",
          "Vassilis","Mat","Bill","Sun","Jenny"]
lnames :: [A_Name]
lnames = ["Maier","Delcambre","Freire","Tufte","Bowers","Tucker",
          "Papadimos","Weaver","Howe","Murthy","Li"]

personStream2 :: Stream A_PersonReport A_PersonPunc
personStream2 = person2 newstates
person2 :: [AuctionState] -> Stream (A_Person,A_Name,A_Name)
                                   (Pattern A_Person,Pattern A_Name,Pattern A_Name)
person2 ((s,d,h,m,ps,abs):a2@(s',d',h',m',ps',abs')):as) =
  if (length ps == length ps') then [] : person2 (a2:as)

```

```

    else [norm ((ps' !! 0), getfname, getlname)] : person2 (a2:as)
  where getfname = fnames !! ((random d') 'mod' (length fnames))
        getlname = lnames !! ((random d'-17) 'mod' (length lnames))

--AUCTION stream
-- AUCTION(id, category)
auctionStream2 :: Stream A_AuctionReport A_AuctionPunc
auctionStream2 = auction2 newstates
auction2 :: [AuctionState] -> Stream A_AuctionReport A_AuctionPunc
auction2 (a1@(s,d,h,m,ps,abs):a2@(s',d',h',m',ps',abs')):as =
  if (length abs == length abs') then (newclosed abs abs'):auction2 (a2:as)
  else [norm (getid (abs' !! 0), (random d') 'mod' 302)] : auction2 (a2:as)
where getid (a,b,c) = a
      newclosed [] _ = []
      newclosed ((a,b,c):xs) ((a',b',c'):ys) =
        if (a==a' && c==False && c'==True) then [punct (lit a, wc)]
        else newclosed xs ys

```