Fall 1-1-2012

A Data-Descriptive Feedback Framework for Data Stream Management Systems

Rafael J. Fernández Moctezuma
Portland State University

Recommended Citation
Fernández Moctezuma, Rafael J., "A Data-Descriptive Feedback Framework for Data Stream Management Systems" (2012). *Dissertations and Theses.* Paper 116.

10.15760/etd.116

A Data-Descriptive Feedback Framework for

Data Stream Management Systems

by

Rafael de Jesús Fernández Moctezuma

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

Doctor of Philosophy
in
Computer Science

Portland State University
2012

ABSTRACT

Data Stream Management Systems (DSMSs) provide support for continuous query evaluation over data streams. Data streams provide processing challenges due to their unbounded nature and varying characteristics, such as rate and density fluctuations. DSMSs need to adapt stream processing to these changes within certain constraints, such as available computational resources and minimum latency requirements in producing results. The proposed research develops an inter-operator feedback framework, where opportunities for run-time adaptation of stream processing are expressed in terms of descriptions of substreams and actions applicable to the substreams, called *feedback punctuations*. Both the discovery of adaptation opportunities and the exploitation of these opportunities are performed in the query operators. DSMSs are also concerned with state management, in particular, state derived from tuple processing. The proposed research also introduces the *Contracts Framework*, which provides execution guarantees about state purging in continuous query evaluation for systems with and without inter-operator feedback. This research provides both theoretical and design contributions. The research also includes an implementation and evaluation of the feedback techniques in the NiagaraST DSMS, and a reference implementation of the Contracts Framework.

DEDICATION

This dissertation is dedicated to my family: To my wife, Anya. To my parents, Rafael de Jesús and Rosa Imelda. And to my grandmother Rosita.

ACKNOWLEDGMENTS

I like to think of the Ph.D. process as a team effort. All these years spent working toward this thesis are marked by love and support from my family. I thank my wife, Anya, for going through the process with me, and for constantly reminding me of the importance to make progress a day at a time. I would also like to thank my parents, Rafael de Jesús and Rosa Imelda, for putting up with weird schedules, and for having invested so much toward my education, both inside and outside academia. Thanks!

I could not have finished this dissertation without David Maier's advice. He was very generous with his time, always willing to listen to whatever I had to say, and always teaching me new things about most subjects. In addition to guiding my work and also providing life advice when needed, he instilled in me a set of skills that will no doubt remain with me for the years to come. Thank you, Dave!

I am fortunate to have collaborated with Robert Bertini while doing research in transportation, and I wish to thank him for his support and guidance in the early stages of my Ph.D. work. Len Shapiro and Lois Delcambre offered me advice on both my work, my academic life, and generously made themselves available to listen to any concerns I had; I remain grateful for having worked with them. While working in transportation I also had the opportunity to work with Christopher Monsere, who in addition to offering insight into my early transportation work, also agreed to serve in my dissertation committee. I want to thank Jonathan Goldstein from Microsoft not only for agreeing to be part of my committee, but also for for constantly giving me new opportunities to think about in stream processing.

My research leverages years of work from the original NiagaraST team. I benefitted not only from their design but also from their insight into stream processing. Their advice and criticism at the early stages of my work was remarkably instructional. Thank you Jin Li, Kristin Tufte, and Vassilis Papadimos.

Through my Ph.D. I was fortunate enough to be surrounded by talented people in many disciplines. In more than one occasion, I abused their generosity by asking them to proof-read manuscripts, criticize my work, help me practice for a talk, look at code, or listen to whatever I had in my head. For all this, I wish to thank Dave Archer, Amit Bhat, Sharook Daruwalla, Tom Harke, Rashawn Knapp, Chuan-kai Lin, Emerson Murphy-Hill, Susan Price, Jeremy Steinhauer, and Jerzy Wieczorek. In his dissertation, James Terwilliger thanks me for motivating him to power through the final stages of his Ph.D. work; I want to thank him for helping me through the early ones. I also want to thank all the faculty in the Computer

Science department, as well as all faculty at the Intelligent Transportation Systems Lab in the Department of Civil and Environmental Engineering for their instruction, guidance, and support. I also want to thank all staff members in the Computer Science department.

Jesús Leyva Ramos encouraged me to pursue a Ph.D. in Computer Science years ago. I am grateful to have received his teachings and advice. Around the same time, David Antonio Lizárraga Navarro and Alejandro Ricardo Femat Flores provided me with sound advice for surviving and succeeding in grad school, and I remain grateful for their time and encouragement. Todd Leen, who advised me during my M.Sc. work, taught me skills without which pursuing my Ph.D. would have been even harder, and I thank him for his teaching and friendship.

Working with the *StreamInsight* team allowed me to expand my perspective on DSMS design and architecture. I thank all my colleagues and mentors for all their advice and *insight* (no pun intended).

My friends Thad Davidson, Chad Ginther, Gonzalo Damián Hernández Araujo, Umut Özertem, and Brian Kurle played a very important role during this period, and I thank them for their support.

I am grateful for the funding that allowed me to pursue my Ph.D. I wish to thank the Consejo Nacional de Ciencia y Tecnología (CONACyT) for their support (fellowship 178258). Additional funding for this research was provided by the National Science Foundation (grants IIS 00-86002, IIS 06-12311, IIS-0917349, NSF-CAREER award 0236567), and the Oregon Transportation Research and Education Consortium (OTREC grant 2007-64).

I learned valuable skills from a lot of talented people. My work could not have occurred without the support of my sponsors. I look forward to passing on what I learned and contributing to the advancement of my field.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The last decade has seen the emergence of a new paradigm in data processing. Database-backed applications, in which data is stored and managed by a Database Management System (DBMS), proved challenging to adapt to high-rate incoming data. Latency requirements on these monitoring applications started to challenge the DBMS model, in which one first stores the data and then issues queries against this static collection. Monitoring applications, as explained by Carney *et al.* [14], exposed a different paradigm to data processing for which no data management system support existed.

The database community took interest in this problem. Projects such as Aurora [9], FireStream [52], Gigascope [20], Hancock [19], NiagaraST [47, 64], PIPES [34], and STREAM [3] studied this new problem space. Over time, models based on the relational model emerged to address the requirements of this new class of systems. Under these models, one can express *continuous queries* against a stream of incoming data, producing a stream of results. The resulting systems

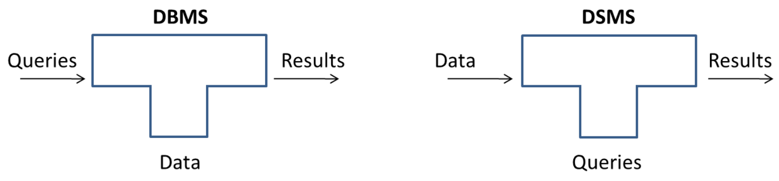


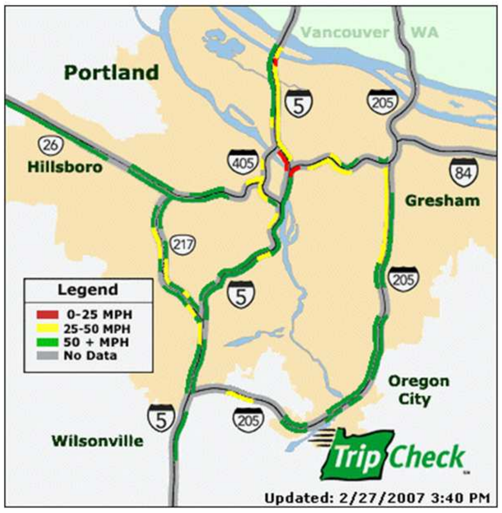
Figure 1.1: Conceptual difference between DBMSs and DSMSs.

are commonly known as Data Stream Management Systems (DSMSs) or Stream Processing Engines (SPEs). A high-level representation of the difference between DBMSs and DSMSs is presented in Figure 1.1. In DBMSs, the data is relatively static, queries are issued against it, and query results are obtained. In DSMSs, data is *pushed through* standing queries, which produce a stream of results.

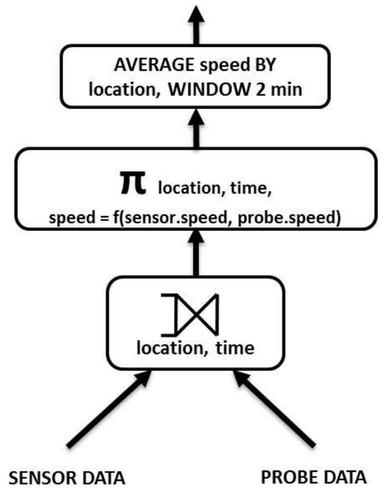
Borrowing from years of experience in DBMS design, DSMSs tend to represent queries internally as directed acyclic graphs of interconnected operators [9]. Operators in DSMSs perform functions that are equivalent to their DBMS counterparts such as filtering, joining, and windowing. Operator semantics in DSMSs have been defined to cope with three main concerns in data stream processing: (1) to maintain low latency in result production, (2) to manage available computational resources efficiently, and (3) to handle the unbounded and out-of-order nature of data streams. The following example illustrates these three main challenges:

Example 1. A speed map is a visual macroscopic representations of speed conditions in a road network. For example, TripCheck (`www.tripcheck.com`) provides a speed map of the Portland metropolitan area, where each segment of the freeway is tagged by a color indicating average speed – green for above 50 mph, red for under 20 mph, and yellow otherwise. Road conditions are estimated from two sources: probe vehicles and fixed sensors. A probe vehicle is equipped with a GPS unit and reports its average speed for the last 20 seconds plus its current location. Fixed sensors are deployed along the various corridors, placed about 1 mile apart from each other, and report the average speed of observed vehicles every 20 seconds. Speed maps are refreshed every 2 minutes. One way to support a speed-map application is to execute a continuous query over the sensor and probe data streams, outer-join by location and time, combine probe and sensor data by applying a function, and average the 20-second reports per sensor location into 2-minute estimates. Once the 2-minute estimates are computed, one can project these aggregated estimates to determine the appropriate color tag. An illustration of a speed map and a query plan to support it is presented in Figure 1.2.

Processing the continuous query in Example 1 can be challenging. The various operators involved require *contextual information* about the status of other operators (or buffers), not only to perform their stream-relational operation, but also to correctly manage their states and result production. The `AVERAGE` operator cannot issue its result correctly without knowing that all events for a 2-minute period have been seen, hence, it is a *blocking* operator. The `JOIN` operator needs to hold on to state related to the events it is processing, and cannot safely purge



(a) Speed map


AVERAGE speed BY location, WINDOW 2 min
π location, time, speed = f(sensor.speed, probe.speed)
⋈ location, time
SENSOR DATA
PROBE DATA

(b) Query plan to support speed map

Figure 1.2: A speed map application and a continuous query to support it. Image from `http://www.tripcheck.com/`.

state content unless it has definite knowledge that an element in state will not participate in future result production. Although the `JOIN` operator does not block result production, it is *stateful*, as it needs to remember information about past inputs to produce later results. In general, blocking operators contribute to latency concerns, and stateful operators contribute to resource-management concerns.

Various approaches have been proposed to cope with these two challenges, of which we mention two. A first approach assumes ordered input or bounded disorder [3, 9]. Under this approach, a blocking operator need only keep track of the current high watermark on a time attribute to decide when it is safe to produce a result. Similarly, a stateful operator can release partial state once a particular point in time is reached. When disorder is bounded, blocking and stateful operators are enhanced with techniques that account for the bounded disorder. One such technique is *slack*, where operators buffer and sort a fixed number or duration of events in the input prior to processing [2].

A second approach embeds *markers* in the stream to signal the passage of time. These markers are called *CTIs* in CEDR [27], *heartbeats* in Gigascope [33], and *punctuations* in FireStream [52] and NiagaraST [62]. In this approach, operators are asked to be aware of these markers, and exploit them to trigger result production and state-management techniques.

Disorder can occur not only at the input, but also in the streams produced by each operator, as noted by Hammad *et al.* [28]. Not only is obtaining prior knowledge of this disorder to accomodate slack challenging, but it is also hard to translate a value on the inputs into meaningful values for internal parts of the query to use. Handling the unbounded nature and disorder of streams is the third main concern in stream processing. By leveraging the embedded markers approach, disorder-tolerant, unbounded-handling architectures have been proposed, such as the OOP architecture by Li *et al.* [42].

The current paradigms in DSMS design have modified the functionality of operators, when compared to their relational counterparts. First, these operators are prepared to have data *pushed* through them, instead of *pulling* data from antecedent sources. Second, they are being asked to handle elements in their inputs beyond just events: they now consume, exploit, and produce embedded *markers* to manage stream progress, and, when appropriate, purge state and unblock.

1.1 DEFINITIONS

Various terms and syntax are used in the literature to refer to key concepts in Stream Processing. I present terminology and syntax for this domain that I use in this document, which is close to the syntax and definitions provided by Tucker [62], and terminology employed by Goldstein *et al.* [27].

Schema A *schema* s is a named, ordered list of attribute-domain mappings. To represent three attributes, `time`, `id` and `temperature`, whose domains are `timestamp`, `integer` and `float`, respectively, we can write a schema named *sensor* as

```
sensor(time:timestamp, id:integer, temperature:float).
```

Event An *event* e over schema s is an ordered list of values that adheres to s. The order in which the values appear in e matches the order of s. To represent a measurement taken on ‘April 1st 2010, at 10:00:00’ by a sensor with `id = 4` and a temperature reading of `76.5` in schema `sensor`, we write

```
<‘2010-04-01 10:00:00’, 4, 76.5>.
```

Tuple is a synonym for event, and I will use these terms interchangeably.

Data Stream A *data stream* over schema s is a potentially unbounded sequence of events, with each event adhering to s. For example, consider the stream DS which conforms to the aforementioned `sensor` schema. The following collection of events might be an initial prefix of the DS data stream:

```
<‘2010-04-01 10:00:00’, 4, 76.5>
<‘2010-04-01 10:00:00’, 3, 75.9>
<‘2010-04-01 09:59:00’, 7, 76.6>,
```

while the following list is not:

```
<4, 76.5>
<‘2010-04-01 10:00:00’, 3, 75.9>
<‘2010-04-01 10:00:00’, 7, hot>.
```

Notice that data streams can be *physically disordered*. In the first DS example, we observe an event for time ‘09:59:00’ after events for time ‘10:00:00’.

Punctuation A punctuation is a predicate that describes a subset of events in a data stream. Such a predicate is a list of restrictions, ordered according to the data stream's schema. Each restriction refers to the domain of the attribute it is positioned on. Each restriction can contain comparators with a value (such as $> 10, \leq 9, <$‘10:00:00 a.m.’), a specific value (equivalent to using “=”), or the wildcard “*”, which refers to any value. For example, to describe all events in the sensor stream related to the sensor with `id = 4`, we write

$$[*, 4, *].$$

To describe all readings up to "April 1st 2010, at 10:00:00" from sensor with `id = 4`, we write

$$[\leq\text{‘2010-04-01 10:00:00’}, 4, *].$$

Given an event e, we can say it *matches* a punctuation p if e is in the set of events described by p. For example, the event `<‘2010-04-01 9:00:00’,4, 79>` matches the previous punctuation, as it is in the set of all events up to "April 1st 2010, at 10:00:00" from sensor with `id = 4`.

Punctuations can be used to communicate context. When an operator emits a punctuation, it is informing its subseqent operator(s) that it will not emit further events that match that punctuation.

Punctuated Data Stream A punctuated data stream is a sequence of events and punctuations with a grammaticality constraint. For any punctuation p that describes a set of events $\mathcal{E}$, no event $e \in \mathcal{E}$ is present in the stream after p. The following sequence is an example of a punctuated data stream over the `sensor` schema, where the topmost element is the earliest:

```
<‘2010-04-01 10:00:00’, 4, 76.5>
<‘2010-04-01 10:00:00’, 3, 75.9>
<‘2010-04-01 09:59:00’, 7, 76.6>
[<‘2010-04-01 10:00:00’, *, * ]
<‘2010-04-01 010:01:00’, 7, 76.6>.
```

The following stream is not a punctuated data stream, since a tuple that matches a punctuation appears after it:

```
<‘2010-04-01 10:00:00’, 4, 76.5>
<‘2010-04-01 10:00:00’, 3, 75.9>
<‘2010-04-01 09:59:00’, 7, 76.6>
[*, 7, *]
<‘2010-04-01 010:01:00’, 7, 76.6>.
```

1.2 IMPROVING STREAM PROCESSING WITH MORE CONTEXT

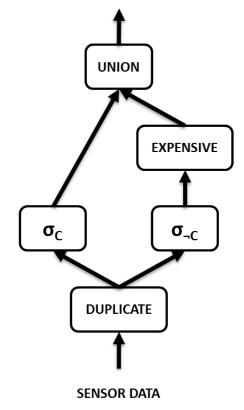
Contextual information embedded in the stream, specifically punctuations, enables DSMSs to cope with latency, resource management, and disorder. To illustrate this paradigm, consider the following input streams for the query discussed in Example 1, with schemas `sensor(time, location, speed)`, and `probe(time, location, speed)`:

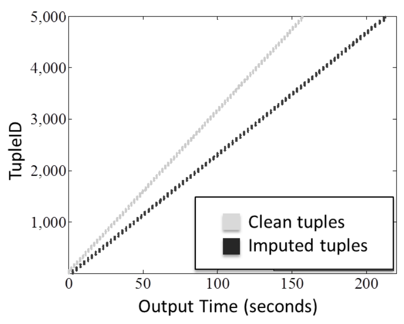
sensor	probe
`<‘10:00:00’, 1, 55>`	
`<‘10:00:00’, 2, 55>`	`<‘10:00:00’,2,65>`
`<‘10:01:00’, 1, 55>`	`[≤‘10:00:00’,2,*]`
`<‘10:01:00’, 2, 55>`	
`[≤‘10:01:00’,*,*]`	

Recall we are joining these streams on `time` and `sensor id`. A symmetric hash join implementation can leverage the punctuation from the `probe` stream to purge matching state from the hash table for the `sensor` stream, and vice-versa. In this example, the tuple `<‘10:00:00’, 2, 55>` in the `sensor` side can be purged once the `[≤‘10:00:00’,2,*]` punctuation is received from the `probe` side.

Now suppose we also see punctuation `[≤‘10:01:00’,*,*]` on the `probe` side. `JOIN` can output the punctuation `[≤‘10:01:00’,*,*,*]` once it has seen punctuation from both inputs. This punctuation can then be used by the aggregate to unblock and produce output for all complete aggregate groups up to the specified timestamp. The projection does not exploit punctuation.

Punctuation is conveying context downstream, and is being leveraged by various operators to process the stream. In this research, I extend this contextual paradigm to consider context which is conveyed *upstream* during processing. To motivate discussion, consider Example 2.





(a) Imputation Query Plan (b) Divergence in the output time of clean and imputed tuples

Figure 1.3: Imputation query plan and divergence among branches.

Example 2. Consider an input stream of sensor data, where a sensor experiences an intermittent failure that causes it to report null values for speed. Input events are filtered and split into two disjoint streams: clean and dirty. An operator called `IMPUTE` processes events for the *dirty* stream and uses a computationally expensive method to replace each missing value with an acceptable estimate. Both substreams are then input to `UNION` to produced an imputed result stream. Figure 1.3(a) shows an illustration of the query plan.

When the density of events that require imputation is high and when the need to impute occurs frequently, the relative timestamps in the output will diverge considerably over time. This observation is illustrated in Figure 1.3(b), where every other event in the input requires imputation. In this figure, tuples are assigned an arrival id at the input and tagged to differentiate which side of the union they came from on the output. This divergence per se may not be a problem if the result stream's destination is a data archive – but what if this were the "sensor data" input for the query plan in Figure 1.2? The speed map application has low *utility* for tuples that are too late. Knowing the conditions of the freeway an hour ago defeats the purpose of a "live" speed map. The remark "you would have been so much better off if you had left before lunch" does not make one feel better about the recently experienced stop-and-go traffic.

Detecting that events about to be output have lost their utility can be done by modifying `UNION` to be aware of this divergence. I call this divergence-aware union operator `PACE`. Detection alone is insufficient, as one needs to communicate this discovery to adapt the query processing and avoid handling events that no longer have output utility, as will be explained in detail in Section 7.4. Various approaches to on-line adaptation of query processing have been proposed, such as STREAMoN [8], which monitors properties of the output to adjust parameters of operators involved in a query, and control-theoretic approaches that view stream processing as a control problem, such as work by Carminati *et al.* [13]. These approaches have a common property, which is collecting some measurement in the output, analyzing these measurements in a separate control unit (or monitor), and using the analysis results to adjust operator properties.

The aforementioned approaches can certainly be applied to the divergence problem, but I question the need to construct separate machinery to monitor and handle the adaptation. `IMPUTE` would certainly benefit from receiving a description of

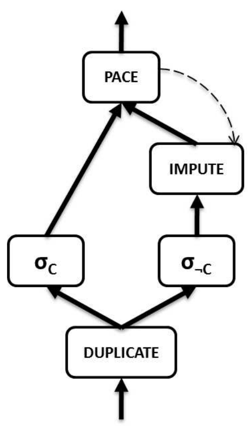


Figure 1.4: `PACE` communicates context information upstream to `IMPUTE`.

which events have lost utility from `PACE`, and it could immediately eliminate events that match this description from its processing queue. This approach would allow `IMPUTE` to resume work on tuples that still have utility. `PACE` already detected the divergence, and it could send this *downstream* context upstream. Moreover, it could communicate it as a description of a set of events. Figure 1.4 illustrates this communication flow.

In this research, I provide a framework that enables detection, communication, and exploitation of downstream context to *improve* query processing, be it by *avoiding* the processing of events whose utility has expired, or by avoiding processing events that do not contribute to the query output, as it is done in sideways information passing [32]. I also identify other options for improvement, namely priorization of subsets of the event stream and causing partial results to be produced.

The approach consists in modifying operators to: (1) detect optimization opportunities, (2) express these opportunities in terms of both a set of events via punctuation and an action to be performed on that set, (3) propagate these *feedback punctuations* upstream to antecedent operators, and (4) exploit incoming feedback to improve local processing. The proposed model is illustrated in Figure 1.5 (b), and contrasted to an alternative centralized approach shown in Figure 1.5 (a).

1.3 ACCOUNTING FOR FEEDBACK-RELATED STATE

One of the main concerns in DSMS design is the accumulation of event-related state. Punctuated data-stream processing has enabled DSMSs to cleanse event-related state as the stream progresses over time. While developing the feedback framework, I noticed that I was introducing context-related state into the operators. The specifics of these new pieces of state are detailed in Chapter 3. *Grosso modo*, operators can hold every description received via feedback, and avoid emitting events described by that feedback. As query processing advances, this feedback-related state can continue growing. Concerned about unbounded feedback-related state, I developed a framework to guarantee such state's correct disposal.

The intuition is that this state can be cleansed by exploiting the same mechanism used to cleanse event-state: punctuation. For example, if an operator is

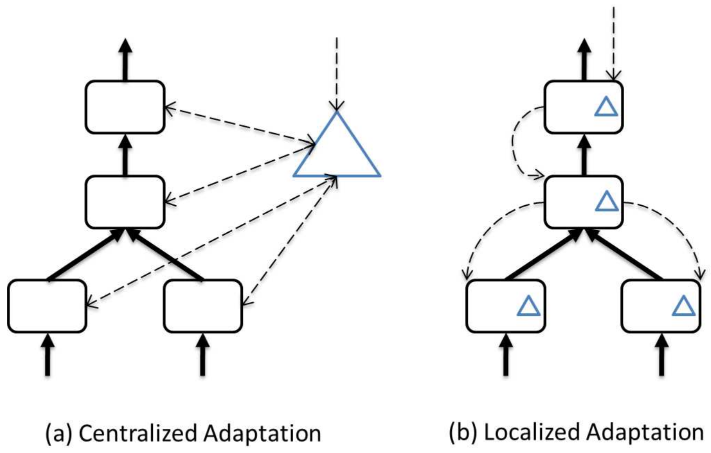


Figure 1.5: Centralized Optimization vs. Localized Optimization. Thick arrows indicate stream flow. Dashed arrows indicate feedback communication. The triangle represents where optimization opportunities are discovered and interpreted. In (a), contextual information is sent to a centralized adaptation artifact, which in turn analyzes the context, determines adaptation opportunities, and communicates these opportunities to relevant parts of the query. In (b), the adaptation machinery is embedded in the operators, which need only communicate opportunities upstream.

holding feedback that describes events whose timestamps are before "11:00 a.m.", incoming punctuation that asserts the current time past "11:00 a.m." can correctly trigger the removal of this feedback. The problem is if one wants to make a global statement about managing state, one needs to know about future punctuation. In some DSMSs, such as CEDR [27], punctuations (CTIs in CEDR) only occur in the time-related portion of the events. This limitation is not true in NiagaraST, or in general when dealing with arbitrarily punctuated streams, complicating progress-related execution guarantees.

The proposed *Inter-Operator Contracts Framework* not only annotates streams with the expected form of incoming punctuations, but also enables pre-evaluation analysis of a query to reason about its execution safety. With these annotations we can perform an analysis before the query is executed and assert certain guarantees (or lack thereof) about result production and state management.

1.4 CONTRIBUTIONS

This research makes the following contributions:

1. An extension to the state of the art in Data Stream Management System architectures, where information that may be used to improve processing is sent counter to the stream direction, from subsequent to antecedent operators.

2. Theoretical contributions, in the form of precise definitions of correctness and operator characterizations, to exploit downstream context to avoid processing

subsets of the stream that do not participate in the output of a continuous query, hence avoiding unnecessary computations.

3. A theoretical extension to punctuated stream processing where whole-query analysis is performed on continuous queries and punctuation descriptions to avoid executing queries that do not eventually produce all correct output or do not eventually release all pieces of state.

4. An evaluation of the proposed feedback framework in the NiagaraST system.

1.5 ORGANIZATION

The rest of this dissertation is organized as follows: In Chapter 2 I present a review of the literature relevant to both the feedback work and the contracts work. Chapter 3 contains the proposed feedback model, and details both our view of where feedback may come from and how it could be exploited, and defines correctness for feedback propagation and exploitation. I characterize operators found in the NiagaraST algebra in Chapter 4, proving a subset of correct responses of feedback, and detailing the procedure one may follow when extending operators to be feedback-compatible. The contracts framework is introduced in Chapter 5, where I detail its derivation, explain its application in general, and further extend it to cover feedback. Chapter 6 includes a summary of the implementation of the feedback framework, detailing the changes made to the NiagaraST architecture. I present an experimental evaluation of the feedback framework in Chapter 7. Chapter 8 contains my conclusions on this work, and a series of open questions and possibilities for future work.

Chapter 2

BACKGROUND AND LITERATURE REVIEW

Data Stream Management Systems emerged from a series of observations around a particular type of application that was both under-served by database management systems and required ad-hoc solutions to be designed from the ground-up. Applications of this type had a theme in common: monitoring. Schreier *et al.* describe a new architecture for turning a DBMS from a passive system, waiting for queries to be issued in order to evaluate them, into an active system, in which the acquisition of data would itself trigger computations over it. They called their system Alert [53]. Systems contemporary to Alert, such as HiPAC [21] and Starburst [65], exploited the relational model in various ways, and focused on updating the state of relations as data entered the system. Another key commonality is the use of rules as the central entity in these systems.

As streaming needs became better understood, language constructs and innovative architectures were proposed to better address the common requirements of monitoring applications. Essentially, it was well understood that challenges included the unbounded nature of data streams, requirements on expressibility of interesting queries, scalability, and efficiency in state management. The bulk of the work in this second stage of streaming systems focused on optimizing the design and architecture of a server that could host continuous queries, often expressed in a SQL-like language. From this period, notable systems include STREAM [3] and Aurora [9]. Seminal work in defining languages to express continuous queries includes Hancock [19] and CQL [4]. Systems that focused on addressing out-of-order processing, the unbounded nature of streams, and characterizing temporal semantics include NiagaraST [40, 63], CEDR [27], and Gigascope [20]. The notions I adopted in this research, and described in Chapter 1, come directly from the work done in this era. At the time of this writing, I am tempted to state that the requirements, characteristics, and challenges of systems that serve continuous queries are well understood – so well understood that we are seeing commercial implementations of stream engines being offered today. Examples include IBM's InfoSphere Streams [12, 31], Microsoft StreamInsight [46], Oracle CEP [51], Streambase [55], and Truviso [58].

My work arrives to the field on top of these foundations, and at the beginning of a new stage in stream processing. For the most part of its research years, the dominating paradigm of streaming systems was a client-server model. The continuous queries being evaluated, in most scenarios, were the primary concern of event-driven solutions. Much work went into making sure the queries were evaluated as efficiently as possible in these systems, as well as into enriching the semantics and behavior of the stream-relational operators involved. Some work also went into resiliency of the queries, as was the case with replication in Borealis-R [30]. This thesis, even though it concerns extending the internal capabilities of the system, is mostly focused on the edges of the continuous query. Increasingly, streaming applications are now a part of larger systems, and are merely a component interacting in larger systems. We may be at the beginning of the next era of stream processing. Consider, for example, one of the supported models of deployment of Microsoft StreamInsight. A streaming engine, capable of hosting multiple queries, can be created on demand inside another process – as opposed to the classical client-server model, where the server would run in its own process, and more often than not in a different machine. The type of streaming applications that are being built using this *embedded* model is opening up interesting opportunities to define and interact with streaming engines.

This research looks at a streaming query as a component that can receive external stimuli from the output side in the form of contextual information, as well as contextual information generated within the query. Not only are input-stream properties defining the behavior of the streaming query, but also the consumer of a query's output can express different needs. My exemplar application is a query whose evaluation is not keeping up with a service-level agreement expected from the consumer, and the consumer expresses a desire for the query to catch up. The runtime dynamic implications are the propagated as inter-operator feedback through the query. As an embedded component, it is reasonable to expect some execution guarantees from the streaming query. Specifically, the second aspect of this work looks at guaranteeing that a query's execution will remain bounded and well-behaved with respect to event-related state management and result production. These guarantees will be evaluated before the query is executed.

In this Chapter, I present several areas related to my research. I have organized related work in four sections, each of which matches an aspect of this thesis' contributions. First, I discuss how contextual information is used to evaluate continuous queries. Second, I give an overview of how other areas of research have incorporated notions of feedback, both adaptively and reactively. Third, I discuss how the static analysis of a continuous query helps determine runtime properties to guarantee its successful execution. Last, I bring to the reader's attention an emerging design pattern in stream engines in which an operator is responsible for activities other than its functional description.

2.1 EXPLOITING CONTEXTUAL INFORMATION

The design and implementation of stream systems responded to a series of challenges present in the nature of data streams. Babcock *et al.* identify several important challenges and considerations this domain, mostly due to the unbounded nature of streams, data density, and processing models [7]. In addition to these challenges, Golab *et al.* also discuss the latency requirements of applications consuming data streams, prominently the need to quickly react to novelties in the input streams [26]. Continuous queries, as opposed to traditional database queries, are often evaluated over a subset of the stream, as opposed to all of the data in the stream. This partitioning of the stream, called windowing, proved very popular in streaming systems [35, 40], however, understanding when state could be correctly purged, as well as when a result was deemed to be correct, proved challenging from an implementation perspective, especially for disordered streams. Both problems disappear when a notion of stream progress is present.

The Aurora system [9] did not require ordered arrival of stream events, neither did the evaluation of aggregates require sorting, as strongly stated in Abadi *et al.* [2]. The system exploited two notions to cope with disorder: first, if events arrive out of order, they can simply be dropped at the input. However, the authors recognize this restriction is too strong for most scenarios; they also offered the concept of a *slack* parameter that bounded disorder. The problem of determining whether all events for a particular window have arrived remains when elements are out of order beyond the slack parameter, or when there are lulls in transmission. Aurora addressed this issue by specifying a timeout parameter, which would cause window results to be emitted as correct and subsequent, too-late tuples to be ignored. Notice this system-oriented approach is well within the spirit of stream systems, where approximate results may be desired over complete results with unacceptable latency. In the Aurora model, stream progress is determined not only as a function of incoming events, but also as a function of the system's clock, as evidenced by the timeout parameter.

In distributed systems, Aguilera *et al.* introduced a powerful construct called Heartbeats, a "periodic *I-am-alive* signal" sent by each node in a distributed system. Different from time-outs, which could incorrectly mark a node as unavailable, a system need only pay attention to the cumulative number of heartbeats sent by a node to asses node health. In STREAM, Srivastava *et al.* extended this notion to address the liveliness problem of pushing tuples through the system [54]. First, they make strong assumptions of deterministic bounds in skew and disorder. Second, their heartbeat signal carries application-time information. Third, events are not sent to the query until a heartbeat arrives with a later application-time than the buffered events. A heartbeat h sent to a query Q causes all events buffered at the input of Q whose application time is less than h to be application-time ordered and pushed through the query. An operator O in query Q will emit a heartbeat h once it has processed all events with application time prior to h.

Johnson *et al.* [33, 42] further extended the idea of heartbeats not only to address liveliness, but also to deal with disorder, unblock operators, and propagate temporal updates downstream through the query. A key aspect of their implementation is that the temporal update is not generated ad-hoc inside the operator, but rather *carried forward* as the heartbeat propagates through the query plan. In this respect, Gigascope's heartbeats, its application and uses are very similar to the uses and applications of Niagara's punctuations [62, 63], which we detailed in

Chapter 1. Both heartbeats in Gigascope and punctuations are conveying a specific contextual meaning, that all events described by a specific punctuation have been seen, and that no event described by the punctuation should be expected after it.

Hammad *et al.* proposed a different approach to understanding stream progress. *Time Probing* is a technique used to request progress information from antecedent operators [29]. In search for guarantees on when it is correct for a given operator O to purge state, O issues a probe to antecedent operators to obtain their maximum-seen timestamp at the time of the probe. If an antecedent operator is still working on events prior to the ones O is planning to evict, purging is not correct. This approach relies on operators having to keep track of progress and respond to probe signals.

There is a common theme in all the techniques I have described: The state of progress upstream (be it data sources or antecedent operators) is communicated downstream. There is no doubt that context detection, specifically progress, has proved advantageous to stream processing. It has enabled operators to be designed to unblock and purge state correctly. In some systems, context detection is propagated as part of the stream, while in others, it is discovered or generated on demand. Regardless of the origin of context, its application to low-latency result-production and optimal resource usage is evident.

My research is also exploiting contextual information to benefit stream processing. My work distinguishes itself from current contextual work in terms of where discovery of context occurs and how it is communicated to other operators. In my work, this context originates downstream, not upstream, and is communicated counter to the stream direction, not in the regular stream direction. This downstream context is communicated by a feedback punctuation, following the theme of systems that rely on descriptions of sets of tuples. Unlike punctuation, feedback punctuations are not part of the stream, rather, they flow outside of the data queues. Similar to time probing, there is a notion of communicating with antecedent operators, but these communications are not requests, rather, they are the delivery of contextual information.

2.2 RUNTIME ADAPTATION

During query execution, changing properties of the input streams, as well as resource limitations, can affect the evaluation of a continuous query. A central contribution of my thesis is the ability to use feedback as a trigger for avoiding unnecessary work in a continuous query. In this section, I will situate the work with respect to two main areas in stream engine design: scheduling and shedding load, and the use of feedback mechanisms.

2.2.1 Adaptation of operators and plans

When evaluating a continuous query, operator scheduling becomes critical. The obvious reason is the unbounded nature of streams – one cannot schedule an operator and wait until it finishes consuming *all* of its input, as the end of the input may not happen. Moreover, since resources are bounded, there is a limit in how many events can be waiting on an operator's input queue. Unsurprisingly, most stream-system descriptions mention this resource limitation as a motivation for operator scheduling. While it is possible to delegate scheduling to the operating system, by assigning one operator per thread (as it is done in NiagaraST), a more common approach consists in reasoning about state to schedule operator execution. Carney *et al.* describe how operators in Aurora are scheduled in an attempt to maintain an expected quality-of-service level [15]. The Aurora visual query language represents a workflow-level abstraction, in which operators (typically represented as boxes) are interconnected via arrows (queues). Various consecutive operators can also be grouped in a *superbox*, and the superbox itself can be scheduled for execution as a single unit. During execution, there is a notion of tuple-processing cost, as well as latency of tuples waiting to be processed. The objective is to estimate which box (or superbox) can yield the maximum benefit in aggregate quality of service, if scheduled next.

A similar scheduling technique, consisting of grouping chains of operators, was proposed by Babcock *et al.* for the STREAM system. They observed that simple scheduling techniques, such as first-input, first-output, or greedy approaches based on number of input elements to an operator, did not present significant advantages until combined toward the goal of maximizing throughput [3, 6].

Schedulers evolved to include runtime-derived statistics. Avnur *et al.* introduced the construct of Eddies, mechanisms that can adaptively re-order operators during query execution [5, 17]. An eddy will dynamically choose where to send a particular tuple to be processed based on policies, such as operator cost, selectivity, or queue lengths. Eddies were shown to be beneficial in the presence of multi-way joins, where selectivity and other derived statistics helped route an event to one join over the other (when possible) in an attempt to reduce query-processing latency. Babu *et al.* added an adaptive monitor to STREAM, called StreaMon, to re-order joins based on greedy policies, again, in an attempt to minimize resource consumption and improve throughput [8].

Nehme *et al.* introduced a "middle-ground" variation to adaptive query processing, called *Query Mesh* [48]. Their system acknowledges data distribution can shift over time, and uses a classifier to detect dominance of a particular distribution at any given time. When a pattern shift occurs, tuples are routed to one of many alternate query plans whose physical evaluation is better tuned for the incoming distribution. The system computes the set of possible query plans and the classifier based on a training data set.

Works *et al.* introduced an adaptation technique that favors (*i.e.*, prioritizes) processing of a subset of the input stream [66]. Their *Proactive Promotion Engine* makes decisions on scheduling based on *priority punctuations*, which are elements in the stream describing a set of tuples and their relevant characteristics with respect to a potential preferential switch. Priority punctuations are injected based by classifier operators, which evaluate runtime properties against a trained classifier.

Notice the systems I have described attempt to collect runtime information to adapt during query execution. Runtime information includes event-derived properties, such as which input queue has the tuple that has been waiting to be processed the longest, as well as learning the selectivity of operators, or collecting statistics on memory usage. All these concerns, both system- and query-related, have been thrown into the mix with the goal of maintaining optimal continuous query evaluation. However, scheduling enough is not sufficient in some scenarios. Consider the motivating example in Figure 1.4. Imagine a particular scheduling technique that realizes `EXPENSIVE` takes considerably more time processing events than all other operators, and therefore schedules it to be executed most of the time. Such a scheduling policy, in the most optimistic case, could minimize the divergence in application-time as observed by the `PACE` operator. While scheduling has minimized the divergence, the query consumer (the speedmap) will receive all tuples late if resources are insufficient. The policy would minimize divergence at the expense of hurting latency and throughput. Assume scheduling tries to maximize throughput and schedules all other operators more frequently than `EXPENSIVE`.

No such scheduling policy could prevent the expensive branch from accumulating work and becoming progressively later. The techniques I introduce in this thesis are outside of scheduling by necessity not only because of the extreme case, but also because I argue for cases where detection of service-level agreement violations occur at the output edge (and possibly outside) of the query. However, the technique is not isolated from scheduling – operators encapsulating the feedback mechanism still need to be scheduled for execution.

The Aurora system has a technique, called load shedding, that drops work when query processing is not keeping up with an expected quality-of-service metric [56]. Data rates are input to a load shedder component, which is in charge of maintaining a load-shedding *roadmap* for the particular query. This roadmap has candidate locations where a drop operator can be instantiated (or removed, if not needed). Specification of an expected quality of service entails details about throughput and rates. The load shedder is effectively finding a new query plan that can maximize quality of service based on available resource by knowing how much processing it is avoiding. A drop location is favored over another based on whether the least amount of information is dropped for the largest throughput gain. In a later refinement, Tatbul *et al.* introduced an operator capable of dropping entire windows of data, requiring the operator to have knowledge of downstream window semantics [49].

My research is also dropping tuples, based not on rate or cost, but on necessity for the query output. Consider the speedmap example. Rather than dropping, say, every other tuple input to `EXPENSIVE`, I argue it is more meaningful to drop a specific subset of tuples that will yield not-particularly-useful results. The approach I propose has a declarative flavor to it, rather than an operational flavor, in the sense that I name the tuples to avoid – rather than saying I wished I saw fewer tuples. Notice that the proposed technique will also be applicable even when there is no resource contention in the system – the consumer of a query may be experiencing network congestion even though query execution is well within resources in the stream engine's process.

The approaches I have described rely on centralized, special components of the engine (*e.g.*, the load shedder, StreaMon, Eddies) to detect and initiate runtime adaptations of the query. Another key difference in my research is I do not impose a new construct – rather, I empower operators to detect, exploit, and propagate query-optimization opportunities.

2.2.2 Feedback mechanisms

Several stream systems send information contrary to stream direction during execution in order to adapt query processing. In NiagaraCQ, Shanmugasundaram *et al.* had the ability to request partial results from antecedent operators [36] by sending a control request via a control channel that interconnects operators in a query plan. As I discuss in Chapter 6, NiagaraST is a descendent of this architecture, and I extend this control mechanism to carry feedback punctuations as well as other messages.

Inter-operator control channels are not exclusive to Niagara. Borealis, the distributed version of Aurora has a mechanism called Control Lines [9]. These lines feed operators with functions that affect the operator's behavior, such as decreasing the selectivity of a filtering operator. Borealis maintains a centralized repository of user-defined applicable functions. An operator called `Bind` analyzes the incoming stream to select (or create) a new function from the repository. The results of `Bind` are sent through the control line to the operator, along with instructions on how to apply the changes. Control Lines, as well as instances of `Bind`, are explicitly created for each operator at query-design time. An application of feedback discussed in Borealis consists in enabling online modification of continuous queries to fulfill a quality-of-service specification [1].

Feedback loops are common in control theory, where a particular systems output is continuously analyzed, and the result of this analysis is used to alter some of the system's parameters to achieve a specific workflow. Yu *et al.* reconsider Aurora's load-shedding strategy with a control-theoretic point of view. Their work models stream processing as a feedback loop, where an actuator (the load shedder) sheds tuples at the edge of the query. They claim their actuation can also be placed inside the query. In any case, a very rigorous modeling and understanding of stream characteristics (such as drastic changes is incoming rates), as well as shifting processing costs, leads to some complications in designing the appropriate controller for a continuous query [59, 60]. My work is related to the notion of feedback in the sense that discovery of opportunities may occur downstream, and this knowledge is sent upstream to alter query performance. Unlike as in control-theoretic approaches, my architecture does not send this information through extra-operator channels, rather, it does so in an inter-operator fashion.

2.3 QUERY PROPERTY DECLARATION AND DERIVATION PRIOR TO EXECUTION

As I have mentioned before, stream processing is challenging primarily due to the unpredictable behavior of data streams. In the previous section, we primarily focused on runtime adaptation to dynamic properties of the stream. In this section, we will discuss a complimentary approach towards better predictability of streaming query evaluation, where properties of the query are described or derived prior to its execution. In particular, in this section I position the work detailed in Chapter 5.

Tatbul *et al.* consider the problem of hitting resource limits during query evaluation and introduce the concept of load shedding, where events are dropped in an attempt to keep the query running when the system is running out of resources.

Aurora makes some decision on-the-fly with respect to when and where in the query plan it should drop events, and it also determines how many events to drop. These decisions are the outcome of an optimization problem in which the main constraint, a quality of service specification, is stated prior to execution. In this respect, run-time properties of the query are derived based on the quality-of-service constraints stated for the query. In particular, these are the operating parameters for pre-computing the load-shedding roadmap discussed in Section 2.2.1. My research shares a similarity with this approach only in the sense of annotating the query on one of its ends, but with two important distinctions. First, I aim toward deriving run-time guarantees with respect to result production and state management, as opposed to maintaining a specific quality-of-service requirement on a query's output. Second, the annotations I will employ mention punctuation properties of input streams to a query, rather than runtime expectations of said query.

Towards the end of their seminal punctuation work, Tucker *et al.* started looking at characterizing the types of streams that individual operators could support [61]. Their framework clearly defines the notion of a punctuation scheme, which describes the type of punctuations present in a data stream. For each streaming operator in their query algebra, they show there exists a family of punctuations of interest depending on what the operator does. This characterization enables them to prove a query benefits from a given scheme, where the benefit entails enabling query output production as well as eventual cleansing of state. Our work in this area further extends these notions by describing specific instances of input and output stream punctuation styles, called contracts, offered by a given operator. Moreover, we support more than one option per logical operator. We take the approach of assuming physical implementations of the operators exist and offer specific actions in the context of the physical implementation. When specific operators are put together in a query, we determine the safety of executing this query given the set of offerings available. A second extension consists in distinguishing various different types of stream progress, specifically the "+" and "#" patterns. Furthermore, in my framework there is a notion of query annotation prior to execution, where the annotations serve as an abstraction of supported punctuation patterns in inputs and outputs of any given operator. I also extend the framework to account for feedback punctuations.

Li *et al.* also discuss annotating a query with punctuation styles expected to be in the stream [37]. In their work, they focus on the effect of different styles of punctuation in the JOIN operator. Specifically, they looked at re-arranging multi-joins to guarantee correct execution. My work attempts to validate queries with multiple kinds of operators, but does not consider query rearrangement in order to find a feasible query plan.

2.4 MULTI-FACET OPERATOR DESIGN

A common design pattern I have observed in stream-processing systems consists in requiring an operator to do work beyond its temporal-relational definition. Aspects of such extended work include communicating context downstream or being responsible for other changes in the query. My research is also asking extra work from operators: they now are responsible of detecting, responding to, and producing feedback. In this section I bring attention to the ancillary activities operators are being asked to perform in various systems.

When dealing with temporal semantics, as it is the case in the CEDR system [27], operators are responsible for handling CTIs. This responsibility is not surprising, as these stream elements are used by operators to produce output and cleanse state, and CTIs need to be propagated downstream. This pattern is also present in Gigascope [33], NiagaraST [62], and STREAM [54].

In CEDR, an operator may issue speculative output, which is a result based on the stream seen so far. This speculation enables low-latency result production. In addition to handling time-carrying signals, operators in CEDR are also in charge of issuing retractions when they have revised estimates of a particular quantity. In contrast to CEDR speculation, partial-result estimation in the context of windowed-queries is possible in NiagaraST with an on-demand mechanism called a “prod”, as introduced by Bhat [11].

Beyond time- and result-related activities, operators have been asked to do more work in the stream world. In previous work, we asked operators to adapt and communicate schema changes during query execution when possible [23, 57]. In that scenario, we preceed evolutionary changes with a marker called an “accent”, which describes the evolution and the set of tuples it applies to. Operators are then responsible to adapt to the evolution when it is correct to do so, and propagate this contextual information downstream. Nehme *et al.* asked stream operators handle access control, in particular, responding to a type of punctuation called security punctuation. These security punctuations precede and describe a set of tuples over which a security description, declared in the punctuation, applies. In this respect, the work is similar to the schema-evolution approach, where a description and an intent precede a set of tuples. These two approaches differ from regular punctuation work in their preceding the set of tuples they describe instead of preceding it.

The feedback work described in this thesis has extended two interesting patterns in streaming system design: first, using a descriptive marker and an intent to refer to a set of events in the stream, and second, extending operator implementations to handle these markers. In the case of heartbeats, CTIs, and punctuations, the intent of the marker is implicit, that is, it communicates the passage of time or other progress. Operators then use this knowledge to perform some actions. In the case of accents and security punctuations, the intention is explicitly declared – either by describing a schema-evolution rule to be enacted or a security-policy to be enforced. Feedback punctuations also make their intent explicit, as we will examine in Chapter 3.

Chapter 3

FEEDBACK MODEL

In Chapter 1 we discussed how downstream context can be exploited to improve query processing, and sketched our working hypothesis. In this chapter, I formalize the problem statement, state the research hypothesis, and introduce a feedback model to test the hypothesis. I illustrate how the model covers both sources and types of downstream context, and how this information is expressed in terms of the data.

Problem statement. How can one identify, express, and exploit downstream context information to improve stream query processing?

Hypothesis. One can extend query operators to identify and exploit contextual information to improve query processing. The contextual information can be expressed as a description of a set of tuples and an intended action on that set of tuples. These descriptions can be communicated to antecedent operators, during runtime, to improve query processing.

I will pursue a design that empowers adaptation by the operators in two modes: One, when the opportunity to adapt is discovered within the operator (as is the case with adaptive windowing by Li *et al.* [41]), and second, when the opportunity to adapt is expressed as context from downstream. This context may have been in turn have been discovered by a subsequent operator, or sent explicitly by a client subscriber to the query.

3.1 TYPES AND SOURCES OF FEEDBACK

A feedback punctuation carries two pieces of information: the intent of the feedback and a predicate that describes the set of tuples associated with the feedback. Feedback punctuations flow from one operator to the next against the stream direction. From the motivating examples described in Chapter 1 one can observe that (1) discovery of optimization opportunities for an operator can occur downstream in the query plan, and (2) propagating feedback information upstream may benefit query processing efficiency by having antecedent operators exploit the discovered opportunities. In this section, I present different types of feedback punctuations and their potential applications, discuss potential sources of feedback, and present a range of responses that operators may exhibit when receiving feedback punctuations.

3.1.1 Types of Feedback Punctuations

I have hinted at having more than one intention associated with a feedback punctuation. In stream processing, one can observe the need to avoid unnecessary work (e.g., avoid processing tuples upstream if they are not part of a result downstream), the need to prioritize processing of some tuples, and the need for triggering on-demand partial result production. I have expressed these three needs as the following types of feedback punctuation.

Assumed. An *assumed punctuation* communicates a set of tuples that may be avoided. Consider an operator, `O`, that produces feedback stating that any tuples with timestamp prior to “10:00 a.m.” will be ignored. `O`’s intention is to inform antecedent operators that no tuple matching the feedback punctuation will contribute to `O`’s subsequent result production. This type of feedback punctuation has more of the flavor of a hint than a command – the issuing operator does not assume the receiving operator will react. The receiving operator then has latitude in responding to the feedback punctuation – it can suppress production of some or all tuples that match the description.

Desired. A desired punctuation has the intent of prioritizing production of the set of tuples described. Consider a scenario in which prioritized processing of subsets of tuples is required: A user drives through a freeway system and is interested in receiving the most up-to-date information available for her current location, but is willing to tolerate increased delay in data from other locations. I propose desired punctuation as a type of feedback that can enable prioritization.

To illustrate the use of desired punctuation, consider a new binary operator: `IMPATIENT JOIN`. This operator is eager to produce results. Consider joining probe vehicle data and sensor data. Potentially, there are considerably fewer vehicle-reported events than sensor-reported events, as instrumented vehicles tend to be an expensive means to collect data. `IMPATIENT JOIN` can send desired punctuation feedback to the sensor stream saying "I have vehicle data for segment #3 up to 10:00 a.m. today". Antecedent operators may wish to prioritize processing data for that segment and time period, since `IMPATIENT JOIN` can use such results to produce output. Unlike assumed punctuation, if this new type of feedback is exploited by an antecedent operator, event production time and order is altered, but not result contents. We have found similar uses for describing the need to prioritize computation of a set of tuples in the stream, most notable in the work of Works *et al.* [66]. In their model, the processing engine makes scheduling decisions based on priority punctuations. A priority punctuation describes the set of tuples of interest, but flows in the same direction as the stream.

Demanded. A demanded punctuation conveys the intent of being willing to accept an approximate result. A demanded punctuation carries the sentiment of "I need a result now, even if it is a partial one." Production of this feedback may be triggered by a utility-policy violation. For example, consider a financial speculator, whose margin of action is limited to a few seconds. The speculator needs to decide whether to buy or pass on a particular currency based on fluctuating exchange rates. She would like to receive a best guess estimate on the trend in the exchange rate in less than 5 seconds. A demanded punctuation may cause some aggregates to unblock and produce partial results. In this example, partial results are better than no results, or seeing results after the end of the margin of action. It is also possible for an operator to produce both partial and final results, but this duality would require changes to downstream operators.

Notice a system requiring ordered processing, such as a ordering based on timestamped-arrival order, would not be amicable to supporting desired and demanded feedback. By definition, this type of feedback affects the order in which a stream would be produced. Moreover, assumed and demanded also modify the content of the stream: Assume suppresses some elements in the stream, while demanded can also alter the values of an event. The bigger claim I am making by supporting these types of feedback is that the user's intent is precisely to affect stream evaluation in these ways.

3.1.2 Sources and Applications of Feedback

I have identified three kinds of feedback sources, as well as possible applications of feedback depending on the feedback's origin. The following list is not exhaustive.

Explicit. The definition of the query plan may include explicit policies that require enforcement. Consider uniting two arbitrary streams, and a policy restriction mandated by an application that requires that the result stream exhibit no more than 1 minute of disorder relative to the event timestamps. The query and policy can be expressed in a CQL-like language [4] as:

```
SELECT *
FROM stream1 UNION stream2
WITH PACE ON MAX(stream1.time, stream2.time)
     1 MINUTE
```

In the previous expression, `PACE ON MAX(stream1.time, stream2.time) 1 MINUTE` parameterizes the `PACE` operator to bound divergence below to 1 minute. One possible implementation of such a restriction may translate the expression into a query plan with a `PACE` operator at the top of the query; `PACE` will compare the timestamp attributes of `stream1` and `stream2`. If events are lagging beyond the 1-minute tolerance with respect to the latest observed timestamp, PACE could generate feedback punctuation to inform antecedent operators that events with late timestamps have been seen and are being ignored, so production of such events should be avoided.

Adaptive. I envision adaptive versions of existing operators being able to discover processing opportunities in their streams. Consider a probe-vehicle data stream and a sensor data stream. Both streams are joined on location using tumbling windows of 1 minute. Assume a punctuation arrives on the probe-vehicle stream, indicating all vehicle data has been seen for window 4. Assume the system can determine that window 4 was empty, that is no events arrived for that window. A thrifty version of `JOIN` – `THRIFTY JOIN` – could detect that window 4 is empty, and provide feedback to the sensor stream. Antecedent operators in the sensor stream can choose to stop producing events that would be part of the useless window.

Event-driven. In addition to explicit declaration of policies and adaptive generation of feedback, I have considered event-driven feedback. Consider a speed-map client application, which is a consumer of a continuous query that produces the stream of events describing road conditions. Consider zooming into a section of a speed map. Feedback could be sent to the query as a result of the action of zooming into an area of the map. The query may momentarily avoid processing events that pertain to areas of the speed map that are not in view at the moment, saving processing power. This capability is of potential interest for scenarios in which the streaming computation occurs in the device itself, or where minimizing network traffic is desirable (such as when clients interact with servers via wireless, per-megabyte priced plans, or such communication consumes power that shortens battery life).

3.2 RANGE OF OPERATOR RESPONSES TO FEEDBACK

Feedback punctuations enable a range of responses by the receiving operators. In this section, I analyze these responses in detail for assumed punctuations, which is the main focus of this dissertation.

Consider the stateful, windowed operator `AVERAGE`, whose input stream has the schema `probes(windowID, speed)`, and an output schema `(windowID,` avg(speed)). Assume the window IDs correspond to fixed-length windows. In these schemas, WindowID refers to a unique identifier per windowing group, as used by Li [38]. Assume AVERAGE's implementation has a simple hash table (indexed on window ID), to which two quantities, sum and count, are recorded. When an entry is covered by an input punctuation, sum is divided by count and an output event is constructed and emitted.

Suppose the AVERAGE operator receives an assumed punctuation of the form [<20,*]. AVERAGE can avoid output of results for windows with IDs less than 20, perhaps simply by discarding the result once it is computed, or more aggressively, purging its internal state and avoiding recreation of windows known to be unneeded. Furthermore, this contextual observation about a window no longer being needed can be propagated to the antecedent operator.

Consider AVERAGE receiving an assumed punctuation in the form of [*,≥50], indicating that windowed averages of speeds greater than or equal to 50 will be ignored. Local enforcement by purging of active windows is not a correct response for this feedback. Suppose Window 4 is active and has a current partial average of 51. Purging this window from the hash table would be a mistake, as future events arriving on its input could cause the average for Window 4 to drop below 50. Similarly, propagating assumed punctuation is an incorrect response, as no assumptions on unseen events can be made. Correct response options that optimize processing exist. If grammatical punctuation arrives on its input and indicates that all events for Window 4 have been seen, and the current partial average of that window is 51, `AVERAGE` can avoid constructing and outputting an output event.

The range of correct responses may also be limited by the form of the punctuation. Consider probe-vehicle and traffic-sensor streams, with the following schemas:

`detector(id, freeway_id, milepost, timestamp, speed)`, and

`probe(id, freeway_id, milepost, timestamp, speed)`.

Consider joining the streams on `freeway_id`, `milepost`, and `timestamp`, leading to an output schema `(probe.id, detector.id, detector.speed, freeway_id, milepost, timestamp, probe.speed)`. If `JOIN` receives the assumed punctuation `[*,11,30,10:00:00,*,*,*]` it can safely propagate the feedback to both of its inputs, albeit not in that exact form. In contrast, if `JOIN` receives the assumed punctuation `[*,*,*,*,≥50,*,*]`, feedback can only be propagated to the antecedent operators of the probe stream, as the feedback refers only to attributes present in that stream.

Appropriate responses can also depend on operator state. Consider a tumbling window `MAX` on the speed values from the probe stream, whose output schema is `(window id, max(speed))`. The `MAX` operator maintains a partial aggregate per active window. If `MAX` receives a punctuation of the form `[*,≥50]`, `MAX` has the opportunity to perform at least two actions: It can close all open windows for which the partial aggregate matches the assumed punctuation, and should prevent those windows that match the assumed punctuation from re-forming. Prevention can be locally enforced with a guard on `MAX`'s input, since the antecedent operator may still produce events leading to undesired window values before it receives and acts on propagated feedback.

While I have identified what operators may discover and propagate as feedback, a critical component of the architecture consists of defining what it means for operators to correctly exploit processing opportunities expressed by received feedback.

3.3 CORRECT EXPLOITATION

I characterized assumed punctuation using the following notation: Consider an operator `O`, which consumes an input stream SI and produces an output stream SR. `O` receives assumed feedback punctuation f, and produces assumed feedback punctuation g. The expression $\mathit{subset}(\mathit{stream}, \mathit{punctuation})$ refers to the set of events in stream that match the predicate of the punctuation.

Definition 1. An operator `O` *correctly exploits* a processing opportunity expressed by feedback punctuation f if, after exploitation, `O` produces an output stream S such that $SR - \mathit{subset}(SR, f) \subseteq S \subseteq SR$. This correctness range is reasonable, since the subsequent operator has indicated it will ignore everything in $\mathit{subset}(SR, f)$, but does not expect strict removal.

By bounding correctness, we allow operators to exhibit a null response ($S \equiv SR$) and still deem that response as correct. Alternatively, other operators may aim for maximum exploitation, that is producing $SR - subset(SR, f)$. Allowing an operator to work within these two endpoints gives a very useful advantage, since the operator may in fact be maximally efficient from the point of receiving the feedback onwards, but it may have emitted events covered by the feedback before the feedback arrived. However, no exploitation of assumed punctuation should insert events in S that would not have normally appeared in SR. Note this condition implies $S \subseteq SR$.

For illustration, consider the `SELECT` operator, which receives assumed feedback punctuation f. If `SELECT` does not exploit f, its output stream is SR. If `SELECT` exploits f by adding f's predicate to the selection condition and no matching output has been emitted already, its output is $SR - subset(SR, f)$. If some output has been emitted before f is added to the selection condition, any events that match f in that output are in SR. `SELECT` correctly exploits f in this scenario, since its output S, whether exploiting f maximally or not, is in the $SR - subset(SR, f) \subseteq S \subseteq SR$ range.

3.4 SAFE ISSUANCE

Feedback may be propagated to antecedent operators. Such propagation depends on the ability to compute a function that maps from output to input schema, which may not exist for all operators and attributes. Even when the mapping exists, special care on the construction of the propagated feedback punctuation g must take place.

Definition 2. Operator `O` *correctly issues* g, if it will continue to behave correctly if any antecedent operator correctly exploits the opportunity expressed by g.

Consider two streams with the following schemas: `A(a, t, id)` and `B(t, id, b)`. Consider an equi-join of both streams on t and id. The output schema of `JOIN` is `C(a, t, id, b)`. For feedback `f [*,3,4,*]`, `JOIN` can correctly issue g_A `[*,3,4]` and g_B `[3,4,*]` to inputs `A` and `B`, respectively. If f is `[50,*,*,*]`, a correct issuance is `[50,*,*]` to input stream `A`. One cannot issue anything meaningful to `B`, since that input does not know about `A`, and `[*,*,*]` would mean we are not interested in anything. For the feedback `f [50,*,*,50]`, no correct issuance exists. One might be tempted to issue `[50,*,*]` to `A` and `[*,*,50]` to `B`. Assume we in fact do that. Consider the event `<50,1,1>` from `A`, and the event `<1,1,1>` from `B`. They would join and produce `<50,1,1,1>`, which is not covered by the feedback `[50,*,*,50]`. However if we did issue the aforementioned portions, we could have missed the `<50,1,1>` event from `A`, hence causing an incorrect suppression from the `JOIN`.

In this Chapter, I have presented you with an intuition as to how the feedback model can work in a streaming engine. I have shown examples of various aspects of the model by referring to common streaming operators, and have hinted as to how various operators can exploit feedback to save resources. Later in this Thesis, I will formally characterize what operators can correctly do in the presence of feedback (Chapter 4). I am also concerned about state introduced by feedback handling, such as the guards that are mounted on inputs and outputs. In Chapter 5, I introduce a framework that enables us to reason about state and make guarantees based on describing the punctuations one expects to see in a stream, as well as by describing feedback punctuations expected to be generated.

Chapter 4

OPERATOR CHARACTERIZATION

A central component of the feedback architecture consists in extending the design of operators to react to feedback. An operator's reactions can range from simply ignoring feedback to cleansing internal state, depending on the feasibility of implementing a response given a particular physical implementation of the operator. In this chapter, I characterize some possible responses to assumed punctuation for a representative set of operators in the NiagaraST algebra. I present proofs of correctness for these responses. In some cases, I also illustrate some responses that might at first seem reasonable, but are actually incorrect, and indicate the problem with each.

4.1 CHARACTERIZATION STRATEGY

First, let us review the various mechanisms an operator has at its disposal in the feedback architecture. An operator can (1) activate a guard on the output (avoiding emission of an event that matches the feedback since the subsequent operator presumably will ignore it), (2) activate a guard on the input (avoiding computation on a tuple that corresponds to the feedback), (3) purge internal state of tuples that match the feedback, (4) translate the feedback punctuation and propagate it upstream, or (5) a combination of the above. It is of course possible to have more than one physical implementation of the operator, and have each one implement one or more response strategies.

For the remainder of the thesis, I will use the symbol $\neg$ to denote assumed feedback punctuation. The expression $\neg[*, \leq 4, *]$ denotes a feedback punctuation where the second attribute has a predicate. An assumed punctuation has, at most, one comparison predicate per attribute. A predicate can be one of $\leq, <, \geq$, or $>$, followed by a value in the domain of the attribute, or just a value, which is interpreted as equality. The limitations I impose in the framework (one comparator per attribute and only supporting a few of them) are merely for expositional convenience. Conceivably, one could use regular expressions for each attribute, or some other pattern specification.

To characterize an operator's response, I will examine properties of the assumed punctuations the operator will receive as input. These feedback punctuations are always expressed in terms of the receiving operator's output schema. An important distinction among operators is what the attributes in the output can tell us about the work the operator is doing. For example, if we examine the output of a hash-join, its schema contains attributes on which a join condition was evaluated. The join operator is accumulating and organizing state by these attributes, i.e., they are the keys in each side's hash tables. Intuitively, if the join operator will respond to assumed punctuation by cleansing the corresponding state, it is then of interest to distinguish whether the feedback refers to attributes in the join condition.

In general, I will pay attention to feedback that refers to attributes that functionally determine event-associated state for a given operator. My strategy consists in examining how operators behave and identify these *interesting attributes*. I then characterize what an output schema for an instance of the operator would be, enumerate potential responses, and prove (or disprove) their correctness.

4.2 OPERATOR CHARACTERIZATION

In this section I present characterizations for `SELECT`, `PROJECT`, `UNION`, `JOIN`, `BUCKET`, `AVERAGE`, `COUNT`, `MIN`, and `MAX`. These operators are representative of what one could find in other streaming algebras, with perhaps the most distinct being `BUCKET`. I describe what each operator does and then proceed to its characterization.

To guide the discussion, we will adopt the following concepts: Let A be the set of attributes in the schema of a stream S. A feedback punctuation fp defined in terms of the schema of S contains an entry for each attribute $a \in A$. The entry can either be a predicate p (with a comparator and a value v), or the symbol $*$. A value v is in the domain of the attribute a it appears on. The symbol $*$ means "any", similar as we use it with forward punctuation. Let SR' be the output of the operator after enacting a response to fp. Recall that a particular response is correct if $SR - subset(SR, fp) \subseteq SR' \subseteq SR$.

4.2.1 Guards

A guard is the in-operator mechanism I use to store the information from a given feedback punctuation. Specifically, a guard is a collection of records. A guard record contains a set of predicates. Each predicate in a guard record comes from a feedback punctuation. If an event e matches at least one record in the guard, it is removed. A guard can be mounted on the input or the output of an operator. A record in an operator's input guard can only refer to attributes in the operator's input schema, and correspondingly for the output.

4.2.2 SELECT

The `SELECT` operator applies the predicate f to its input stream S and produces an output stream SR such that $SR = \{e \in S | f(e) = true\}$. Let us consider a possible response to feedback, in which we add the feedback's predicate to an input guard.

Response:

1. Create a new guard record r.
2. Add all $p \in fp$ to r.
3. Add r to the input guard.

Proof of correctness of exploit:

Recall that a response to a feedback punctuation (fp) is correct if the output of the operator contains elements originally in the output of the operator and at most excludes all elements described by fp, that is, $SR - \mathit{subset}(SR, fp) \subseteq SR' \subseteq SR$. I will first prove the first containment, $SR' \subseteq SR$ directly, and the second one by showing $(SR - SR') \subseteq \mathit{subset}(SR, fp)$, *i.e.,* any dropped output matches fp.

Part 1: $SR' \subseteq SR$. An event e in SR' is also in SR since all events output by the operator come directly from the input and are not modified, which means the output when enacting this response contains only events that the operator would have produced without the response.

Part 2: $(SR - SR') \subseteq \mathit{subset}(SR, fp)$. Take an event $e \in (SR - SR')$. This event is not in SR' as a consequence of the exploit. The input guard is only dropping events that would be in SR if they match one of its records. The record added to the guard at the time of the exploit contains exactly the predicates in fp, which means e must also be in $\mathit{subset}(SR, fp)$. $\square$

With a correct exploit in place, that is the input guard, let us consider the following propagation:

Propagation:

1. Propagate fp.

Propagation requires no translation in this case, since the input schema and the output schema of the operator are the same.

For propagation proofs, let S' be any input allowed with the propagated feedback fp.

Proof of correctness of propagation:

We need to show that a specific propagation does not cause the operator's output to be incorrect. In the case of `SELECT`, we prove a stronger statement: the operator's output is unchanged, given the response above.

Recall the input guard is in place before propagation occurs. Going from S to S' could only potentially reduce `SELECT`'s output, since correct responses to feedback do not generate more events in an operator's output. However, an element $e \in (S - S')$ must also match fp, which would cause the input guard to drop it, hence, e could not be part of SR'. $\square$

For the rest of the operators, I will follow the same pattern: Offer a response to feedback, prove its correctness in two parts by showing containment, offer a propagation action with the response enacted, and prove correctness of the response.

4.2.3 PROJECT

`PROJECT` is a stateless operator. Like its relational counterpart, the operator is used to indicate which attributes from the input schema appear in the output schema. It is also used to create new attributes in the output schema as a result of a stateless computation over an event's values. The operator maintains a 1:1 relation between its input and output streams, that is, every event in its input S causes an event in its output SR.

The operator can modify the output schema in four non-mutually exclusive ways:

1. An attribute appearing in the schema of S does not appear in the schema of SR.
2. An attribute appearing in the schema of S is renamed in the schema of SR.
3. An attribute appearing in the schema of S is duplicated and renamed in the schema of SR.
4. An attribute appearing in the schema of SR is the result of a stateless computation over values for attributes in the schema of S.

Recall we have input and output guards as mechanisms to exploit feedback. When discussing `SELECT`, I used an input guard. In this case, I consider using an output guard to avoid computing a translation from output schema to input schema. Consider the ways in which `PROJECT` alters the schema. The first case does not present any particular complications. For the second case, we would need the operator to be able to invert a name mapping. For the third case, we would need to decide whether a reference to a duplicate element (but not the original one) maps back to only the originating attribute. For the fourth case, we would need to compute the pre-image of a given value to correctly guard against its creation. Using an output guard avoids these complications.

1. Create a new guard record r.
2. Add all $p \in fp$ to r.
3. Add r to the output guard.

Proof of correctness of exploit:

The proof is organized as for `SELECT`, with SR' being `PROJECT`'s output after enacting the response.

Part 1: $SR' \subseteq SR$. By construction, the output guard does not create events that would not have appeared in the output had the operator not enacted a guard on fp. Thus, an event e in SR' is also in SR.

Part 2: $(SR - SR') \subseteq subset(SR, fp)$. The set $(SR - SR')$ describes exactly the elements that have been left out by the output guard, that is, if an event e is in $(SR - SR')$, but not in SR', it must also match fp, which means e must also be in $subset(SR, fp)$. □

With a correct exploitation in place, that is, the output guard, let us consider the following propagation:

Propagation:

1. If and only if all p in fp refer to attributes present in both the output and input schema (case 1):
2. Create fp' with each p in fp and with "*" for each attribute a in the schema of S but not in the schema of SR.
3. Propagate fp'.

Proof of correctness of propagation:

We can prove that given the placement of the output guard and since we propagate only if attributes mentioned in fp are both in the output and input schemas, the output of `PROJECT` is unchanged if an antecedent operator correctly exploits fp'.

Recall the output guard is in place before propagation occurs. Let S be the original input to the operator, and S' be a possible input stream after an antecedent operator exploits fp'. Going from S to S' could only potentially reduce `PROJECT`'s output, since correct responses to feedback do not generate more events in an operator's output.

If e is in $(S - S')$, it must be true that e matches fp'. It must also be true that the event e' resulting from performing the projection matches fp, because of how we construct fp'. e' would then be discarded by the output guard, which has a record for fp. Hence, the output of the operator is unchanged by an antecedent operator acting on fp'. $\square$

If we are interested in supporting propagation in the presence of renames, all that is required is support for reverse-renaming of an attribute's name when creating fp'. This capability would allow us to relax the if and only if constraint in the propagation strategy. Note we could support this propagation with the same output guard.

4.2.4 UNION

`UNION` is a stateless operator which generally does not apply any function or transformation on the content of incoming events, and does not eliminate duplicates. The only restriction it imposes is that both of its inputs are union-compatible, which means corresponding attributes are in the same domain. In NiagaraST, because of a unique-name limitation of the XML-QL query language, the operator must map explicitly distinct names to one unique output name, which requires some translation of an incoming fp to the schema names of the input. For simplicity, let us characterize the operator ignoring this implementation detail.

To exploit feedback punctuation, I will employ an output guard.

Response:

1. Create a new guard record r.

2. Add all $p \in fp$ to r.

3. Add r to the output guard.

Proof of correctness of exploit:

The proof is organized in two parts as before.

Part 1: $SR' \subseteq SR$. In this case, an event e in SR' comes from one of the `UNION` inputs, and was not modified or created in any way, which means it is also in SR.

Part 2: $(SR - SR') \subseteq \mathit{subset}(SR, fp)$. An event e in $(SR - SR')$ but not in SR' must have been dropped by the output guard, which means e matches fp, and must be in $\mathit{subset}(SR, fp)$. □

With this correct exploitation in place (output guard), consider the following propagation:

Propagation:

1. Propagate fp to both inputs.

Proof of correctness of propagation:

We can prove that the output of `UNION` is unchanged.

Recall the output guard is in place. Unlike unary operators, we have two input streams here: SL and SR for the left and right input. Let SL' and SR' be the inputs after an antecedent operator in each input exploits the propagated feedback.

On the left side, if an event e is in $(SL - SL')$ but not in SL', it matches fp. This means e will also be discarded by the output guard. And since no exploitation of assumed punctuation creates new elements but can only reduce the number of elements in a stream, the exploitation on the left side did not contribute to a change in the `UNION` output. The same argument applies to the right side. □

4.2.5 JOIN

`JOIN` is the first operator where the notion of attributes of interest comes to play. Consider the output schema of a join operator (for simplicity, I only consider a equi-join). In general, it contains three distinct sets of attributes: The set L which contains attributes exclusive to the operator's left input, the set R which contains attributes exclusive to the operator's right input, and the set J which contains the attributes named in the join condition. This last set J contains the interesting attributes.

`JOIN` can have various physical implementations. One of the most common ones in stream systems is a variant of hybrid hash-join, which maintains two pieces of state: on each input side, the operator maintains a hash table indexed on the join attributes, adding every incoming event to the table. When an event comes to one input, the operator looks for a match on the opposite side's hash table, and if there is a match, the operator constructs and emits an event. The operator can release state when it receives a punctuation on one input, by scanning and deleting entries in the opposite input's hash table. Li *et al.* [40] provide a detailed description of the NiagaraST implementation of this join strategy.

Let us consider the following cases for feedback punctuation:

1. Feedback punctuation refers to attributes exclusive to one input (L or R).
2. Feedback punctuation refers to attributes exclusive to the join condition (J).
3. Feedback punctuation refers to attributes not in the join predicates, but involving both inputs (L and R).
4. Feedback punctuation refers to attributes involving both inputs including some in the join predicates (J, L, and R).
5. Feedback punctuation refers to attributes in the join condition and one of its inputs (J and L, or J and R).

Let HL be the hash table associated with the left input, and HR be the hash table associated with the right input.

Case 1. Feedback punctuation refers to attributes exclusive to one input (L or R).

Without loss of generality, let us consider only feedback punctuation on L. Let X be the set of tuples in HL described by fp.

Response:

1. Update $HL \leftarrow HL - X$
2. Create a new guard record r.
3. Add all $p \in fp$ to r.
4. Add r to the left input guard.

The external effect of this strategy is the same as using an output guard on fp. Notice by purging state and guarding the inputs we avoid the work associated with constructing some tuples that would be thrown out by an output guard.

Proof of correctness of exploitation:

The proof is organized in two parts, as before.

Part 1: $SR' \subseteq SR$. An event e is in SR' if and only if join creates it based on arrival on one input and a match on the other input's state occurs. In our exploitation, we did not add any state to either hash table (in fact, we removed state from HL). Thus, e must also be in SR, since the exploitation only reduced the possible elements from SR.

Part 2: $(SR - SR') \subseteq subset(SR, fp)$. Take an event $e \in (SR - SR')$. This event is not in SR' as a consequence of the exploit. The input guard is only dropping events that would contribute to output in SR if they match one of its records. In addition, the updated hash table may have removed state contributing to SR. The record added to the guard at the time of the exploit contains exactly the predicates in fp, which means e must also be in $subset(SR, fp)$. $\square$

Let us now assume that the hash table purge occurred and the input guard is in place for the left input.

Propagation

1. Send a feedback punctuation fp' with all p in fp to the left input only.

Proof of correctness of propagation:

I will prove that the output of the operator is unchanged as a consequence of the propagation (and the exploitation in place):

Recall the input guard is in place before propagation occurs. Going from S to S' could only potentially reduce `JOIN`'s output, given the nature of exploitation of assumed punctuation. However, an element $e \in (S - S')$ must also match fp', which would cause the input guard to drop it, hence, e could not contribute to elements in SR'. $\square$

Case 2. Feedback punctuation refers to attributes exclusive to the join condition (J).

Attributes in J are present in both input schemas, which means all p in fp can be evaluated against both HL and HR. Let X be the set of events in HL that match all p in fp, and Y be the set of events in HR that match all p in fp.

Local exploitation

1. Update $HL \leftarrow HL - X$
2. Update $HR \leftarrow HR - Y$
3. Create a new guard record r.
4. Add all $p \in fp$ to r.
5. Add r to the left input guard.
6. Add r to the right input guard.

Proof of correctness of exploit:

We follow the same strategy as before. Let SR' being `JOIN`'s output after enacting the response.

Part 1: $SR' \subseteq SR$. The exploitation has performed two main actions: first, it purged both hash tables from entries that matched fp. Second, it added records to the input guards to specifically leave out further events that match fp. Neither of these actions is generating an element not originally in SR, but merely subtracting – hence, an event e in SR' is also in SR.

Part 2: $(SR - SR') \subseteq subset(SR, fp)$. Take an event $e \in (SR - SR')$, and note that e is obtained as $e_L \bowtie e_R$, where e_L is an event from the left input and e_R is an event from the right input. Since fp's predicates are declared only on common columns, if e matches fp each of e_L and e_R do so as well. This event e is not in SR' as a consequence of the exploit – either a constituent for it was removed from state, or an input guard prevented a constituent event from entering the operator. An input guard is only dropping an event that would contribute to an event in SR if the incoming event matches one of the guard's records. The record added to the guard at the time of the exploit contains exactly the predicates in fp. Additionally, constituents in the hash tables that were purged at the time of exploitation strictly matched fp, therefore, e must also be in $subset(SR, fp)$. $\square$

With the input guards in place and the state purged after this exploitation, let us consider the following propagation strategy:

Propagation

1. Send a feedback punctuation fpl with all p in fp to the left input and "*" in the rest of the schema.

2. Send a feedback punctuation fpr with all p in fp to the right input and "*" in the rest of the schema.

Proof of correctness of propagation:

I will prove that the output of the operator is unchanged as a consequence of the propagation (and the exploitation in place).

Consider SL and SL' as the left input stream before and after propagation, respectively. Recall the input guards are in place before propagation occurs. As stated before, going from SL to SL' could only potentially reduce `JOIN`'s output. An element $e_L \in (SL - SL')$ must also match fpl, which also means it is a constituent for an event e that matches fp, therefore, e_L could not contribute to elements in SR'. This argument also holds for the right hand side. $\square$

Case 3. Feedback punctuation refers to attributes not in the join predicates, but present in both inputs (L and R)

Local exploitation

1. Create a new guard record r.
2. Add all $p \in fp$ to r.
3. Add r to the output guard.

Proof of correctness of exploitation:

The proof structure follows the same pattern as before. Notice that when proving correctness of an exploit that only mounts an output guard the correctness doesn't really depend on the specifics of the operator. The proof used for `PROJECT`, for example, works here as well.

Part 1: $SR' \subseteq SR$. By construction, the output guard does not create events that would not have appeared in the output had the operator not enacted a guard on fp, thus, an event e in SR' is also in SR.

Part 2: $(SR - SR') \subseteq subset(SR, fp)$. The set $(SR - SR')$ describes exactly the elements that have been left out by the output guard, that is, if an event e is in $(SR - SR')$, but not in SR', it must also match fp, which means e must also be in $subset(SR, fp)$. $\square$

Notice unlike Case 1 and 2, in this exploitation I did not purge state. I will illustrate why that response to feedback could lead to incorrect behavior.

One may be tempted to purge both HL and HR and send feedback, but this will yield incorrect results. To prove this behavior is wrong, consider the streams $S1$ and $S2$ with schemas $S1(a, b), S2(b, c)$, and consider the join $S1 \bowtie S2$. Now consider the following events:

S1	**S2**
<1,1>	<1,2>
<2,1>	
<3,1>	
<4,1>	
[*,1]	

Now consider the feedback punctuation $\neg[\leq 3, *, = 2]$. If we were to decompose this punctuation in left and right constituents, we would be tempted to remove state associated with the tuple <1,2> on the right side. Now suppose the fp arrives before the tuple <4,1> is seen on the left input. There would be no state for it to match with, hence, the tuple <4,1,2>, which would have been in SR, and is not in $subset(SR, fp)$, would not be present in the output of the operator, hence this being an incorrect response. Any propagation would also be incorrect for the same reason.

Case 4. Feedback punctuation refers to attributes involving both inputs including some in the join predicates (J, L, and R)

Guarding output, just as we did in Case 3, applies to this case and can be proved the same way. However, I would like to discuss why a response involving purging would be incorrect in this case.

One may be tempted to purge both HL and HR, but this exploitation will yield incorrect results. To show this behavior is wrong, consider the streams $S1$ and $S2$ with schemas $S1(a, b), S2(b, c)$, and consider the join $S1 \bowtie S2$. Now consider the following events:

S1	**S2**
<1,1>	<1,1>
<2,1>	<1,2>
<3,1>	
<4,1>	
[*,1]	

Now consider the feedback punctuation $\neg[\leq 3, 1, = 2]$. If we were to decompose this punctuation in left and right constituents, we would be tempted to remove state associated with the tuple `<1,2>` on the right side, and tuples `<1,1>, <2,1>, and <3,1>` on the left side. Now suppose the fp arrives before the tuple `<4,1>` is seen on the left input. There would be no state for it to match with, hence, the tuple `<4,1,2>`, which would have been in SR, and is not in $subset(SR, fp)$, would not be present in the output of the operator, hence this response is incorrect. Similarly, propagating the constituents would lead to incorrect output by the operator.

Case 5. Feedback punctuation refers to attributes in the join condition and one of its inputs (J and L, or J and R)

We can adopt an exploitation and propagation strategy as the one shown in Case 1, with identical derivation. I want to discuss a seemingly correct propagation strategy that one may consider in this case, but can be shown to be incorrect.

We could be tempted to send all p in fp that refer to the join condition to the right input as well. This behavior would be incorrect, as the following counter-example shows:

Consider the streams $S1$ and $S2$ with schemas $S1(a, b), S2(b, c)$, and consider the join $S1 \bowtie S2$. Now consider the following events:

S1	**S2**
<1,1>	<1,1>
<2,1>	<1,2>
<3,1>	
<4,1>	
[*,1]	

Now consider the feedback punctuation $\neg[\leq 3, 1, *]$. If we were to decompose this punctuation in left and right constituents, we would send $\neg[1, *]$ to the right input. Suppose fp arrives before `<1,2>` on the right side. This tuple may have been suppressed by antecedent operators, and there would be no way to produce `<4,1,2>`, which would have been in SR, and is not in $subset(SR, fp)$, which would yield an incorrect result, hence, it is not correct to propagate the feedback to the right side.

4.2.6 BUCKET

In NiagaraST, the `BUCKET` operator encapsulates the windowing logic. This operator, fully parameterized by naming the progressing attribute over which windowing is performed, as well as the slide and range, appends a window-identifier called `WID` to every output event. Li *et al.* introduced and formalized the windowing semantics, and further exemplify its uses [40]. By decoupling windowing specification and computation from other commonly used window aggregate functions, such as sum or average, an aggregate operator need not implement windowing semantics – it simply performs a group-by `WID` and carries on its computation.

For example, consider counting the number of events in a one-minute sliding window over the following stream with schema `s(time,value)`:

```
<‘10:00:00’,‘a’>
<‘10:00:30’,‘b’>
<‘10:01:00’,‘b’>
<‘10:01:30’,‘a’>
[≤‘10:01:30’,*]
```

`BUCKET` is parameterized by declaring `slide`, `range`, and the windowing attribute `wattr`. Suppose we set $slide = range = 1\ minute$, and use `time` as the windowing attribute. `BUCKET` would output the following stream with schema `t(wid,value)`:

```
<1,‘a’>
<1,‘b’>
<2,‘b’>
<2,‘a’>
[≤ 2,*]
```

Notice the operator works both on events and punctuations. For example, a subsequent `COUNT` operator need not worry about windowing semantics, and can simply count the number of events grouped by `WID`.

Notice `BUCKET` is not truly stateless in its described form. Aside from the initial parameters, `BUCKET` needs to hold runtime-related state, namely, the timestamp t_o to which it assigned the first `WID`, "1". There are some consequences with this approach: First, since NiagaraST does not require ordered input to process a stream, the window with ID 1 may not necessarily be the first chronological window. Operationally, we may see a `WID` "0", or negative `WIDs`. Second, from the output-schema perspective, the meaning of the `WID` is lost – it is merely a number used to group on.

In order to interpret a `WID`'s temporal meaning, we need to know the range, slide, and initial timestamp from `BUCKET`, the last one being runtime state of the operator. Arguably, one could redesign `BUCKET` so the initial timestamp is a query-declaration-time parameter, giving control to the query writer as to what a specific `WID` represents. These considerations are of interest to us, since we will need to map a `WID` to timestamps if we want to propagate a translated feedback punctuation referring to that `WID`.

Exploitation for `BUCKET` does not present a significant challenge, and can be well-served by the use of an output guard.

Response:

1. Create a new guard record r.
2. Add all $p \in fp$ to r.
3. Add r to the output guard.

Proof of correctness of exploit:

As we have show before, since the response is strictly mounting an output guard, we can use the same proof structure shown in `PROJECT` (and repeated in `JOIN`.

Propagation of feedback, however requires carefully computing and crafting a description of the window constituents one is trying to avoid. Tatbul *et al.* had to similarly examine window contents in the context of load shedding [49].

Consider the schemata $S(wattr, A)$ and $SR(WID, A)$ respectively, where $wattr$ is the windowing attribute, WID is the emitted window ID computed over $wattr$, and A is a set of other attributes.

In practice, we have encountered use of two types of windows: tumbling and sliding. I will discuss them separately.

Tumbling windows (slide = range)

In a tumbling window, an instance of `BUCKET` maps a range of values in $wattr$ to a single WID as a function f of t_o:

$$WID = f(wattr, t_o, slide) = \lfloor (wattr - t_o)/slide \rfloor + 1$$

For example, consider the integers in the range $[1, 5]$, and $slide = range = 2$. Suppose we see the number 1 first, therefore assigning the $WID = 1$ to it, i.e., $t_o = 1$. We then notice the following WID assignments:

integer	**assigned WID**
1	1
2	1
3	2
4	2
5	3

Notice each window contains a possible range of values. In this example, $WID = 1$ includes the range $[1, 3)$. We can calculate the upper and lower bounds of a window, given its WID, $slide$, $range$, and t_o by applying the following functions:

$$upper = Upper(WID, t_o, slide) = WID * slide + t_o$$

$$lower = Lower(WID, t_o, slide, range) = WID * slide + t_o - range$$

The main consideration is how to compute the interesting set to avoid and describe it in a feedback punctuation to be propagated. Suppose, for example, that fp contains the predicate '$= n$' over the WID attribute. Ideally, we could formulate an fp' for which the range $[lower, upper]$ for n is named in as a value in $wattr$. Since we restricted feedback punctuations to name a comparator and a value only, this translation is not possible. We therefore consider only the cases using the $<$, $\leq$, $>$, and $\geq$ comparators. Assume we receive feedback with the comparator $<$ on WID. Also assume the correct exploitation using an output guard is enacted.

Propagation:

1. Compute $upper = Upper(WID - 1, t_o, slide)$.
2. Create a feedback punctuation fp' with the predicate over $wattr$: $wattr < upper$.
3. Propagate fp'.

Proof of correctness of propagation:

Recall the output guard is in place before propagation occurs. As discussed before, going from S to S' only potentially reduces the operator's output. Assume there is an element e in $(S - S')$ that matches fp'. Now assume the operator creates e' by computing WID and creating e. It is also true that this e' matches fp, which is in the output guard. Hence, e could not contribute to output in SR', which means SR' is unchanged. $\square$

The rest of the comparators require changes to how $upper$ is computed and fp'

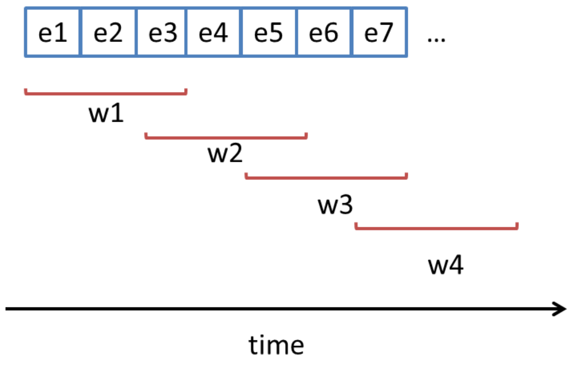


Figure 4.1: Sliding Windows illustration. Events $e_1, e_2, ...$ are assigned to windows. In this example, $range = 3$ and $slide = 2$. Notice an event can participate in more than one window.

is constructed. For example, assume $slide = range = 2$, $t_o = 0$, and feedback on WID with the predicate ≤ 4. If we compute $upper$ as is, we will be propagating feedback that does not cover the constituents of $WID4$.

When encountering $\leq$, one needs to compute $upper = Upper(WID, t_o, slide)$ and propagate. For $>$, we need to find the upper limit of the window $upper = Upper(WID, t_o, slide)$ but construct fp' with the condition over $wattr$: $wattr > upper$. For $\geq$, the propagation strategy is the same as for $>$, only computing $upper = Upper(WID - 1, t_o, slide)$.

Sliding windows (slide < range)

Sliding windows are important features of stream processing. In contrast with tumbling windows, sliding windows overlap, meaning an event e may participate in more than one window. In Figure 4.1 I illustrate this characteristic.

The main concern when translating feedback is to make sure we do not cover overlapping windows by accident. For a time-based slide and range, there is a k such that windows more than k apart do not overlap. For example, with $range = 10$ and $slide = 3$, we have $k = 4$ ($k = \lceil range \div slide \rceil$). By finding this k and avoiding the overlaps in each direction, we can support sliding windows by simply offsetting the *upper* quantity.

4.2.7 AGGREGATES

When one needs to compute a quantity over a finite set of events, one generally uses a group-by to define the groups and then applies the computation. These computations are usually called aggregates. In streaming systems, one most commonly computes *windowed* aggregates, that is, aggregates over time as well as groups. In NiagaraST, the engine computes windowed-aggregates in aggregate operators, which need only know of group-by mechanics, but not time. This simplification is achieved by computing the WID using `BUCKET`, as discussed in Section 4.2.6. In this section, we will characterize the aggregate operators available in the NiagaraST algebra: `AVERAGE`, `COUNT`, `MIN`, and `MAX`.

Consider the aggregate operator `AGG`, with input schema `s(G,X)`, and output schema `sr(G,Y)`, where G is a set of grouping attributes. Let S be the input stream, and Y the result of computing the aggregate agg over each distinct subset $S_i = \{e|e \in S \wedge e[G] = g_i\}$, where all events in S_i have the same values for attributes in G. Let fp be a feedback punctuation. Let us also consider the aggregate's internal state, H, which is a set of events arranged by grouping attributes. Let SR be the output stream.

Let us first consider the case in which fp's predicates p only refer to attributes in G.

Response:

1. Create a new guard record r.
2. Add all $p \in fp$ to r.
3. Add r to the input guard.
4. Remove all events $h \in H$ for which $match(fp, h)$ is true.

Enacting a response with only steps 1–3 would be incorrect, as we would potentially introduce elements not originally in SR, *i.e.,* intermediate results for which we suppressed further input. Enacting a response with only step 4 would also be incorrect, as partial groups could re-form.

Proof of correctness of exploit:

The proof follows the same two-part strategy as before. Let SR' be `AGGREGATE`'s output after enacting the response.

Part 1: $SR' \subseteq SR$. The exploitation is similar to the one we used in `SELECT`, from the point of view of the guard – only elements that match fp are being left out. Notice however that we also purged partial state from H – no tuple that was not originally in SR will ever be emitted. Since we purge state and prevent re-formation of purged groups, the operator only works over complete versions of S_i. Hence, if e is in SR', so is it in SR.

Part 2: $(SR - SR') \subseteq subset(SR, fp)$. Let $e[G] = g_i$, e is derived from S_i. If e is not in SR', it must be the case that S_i itself is missing as a consequence of the exploit (removing state and preventing re-formation). Now suppose $e \in (SR - SR')$. There cannot be output in SR' for g_i with a different value (as in Part 1), so it must be the case that SR' has no event for group g_i, which means e must also be in $subset(SR, fp)$. $\square$

Propagation:

1. Create a feedback punctuation fp'.
2. Add all p in fp to fp'.
3. Propagate fp'.

Proof of correctness of propagation:

We can prove that the output of the aggregate as of the time of exploitation remains unchanged by the propagation.

Recall the input guard is in place, and recall going from S to S' only reduces the number of elements. If an element e is in $(S - S')$, it must also match fp', which would cause the input guard to drop it, hence, e cannot contribute to any e' in SR'. $\square$

Let us now consider the case in which a predicate p in fp refers to Y. One could potentially go through the intermediate operator state and purge existing matches, but this would be incorrect, as fp is referring to values over complete S_is, and not partial ones.Instead, guarding the output and performing the check on a value computed with a complete S_i is a better strategy.

Response:

1. Create a new guard record r.
2. Add all $p \in fp$ to r.
3. Add r to the output guard.

Proof of correctness of exploit:

We can prove correctness as we have done in previous operators since we are only mounting an output guard, and not affecting any of the operator's internal structures.

In the case in which a predicate p in fp refers to Y, it is not possible to compute a satisfactory pre-image of fp to propagate further as feedback for any arbitrary aggregate. In the next sections we discuss whether propagation exists for this case for specific aggregates, and assume the correct response with an output guard is enacted.

AVERAGE

For `AVERAGE`, no fp' can be propagated safely, since the evaluation against the predicates in fp requires the aggregate value v to be fully computed. A change in the set of events used to compute a particular v could result in a new value, v', originally not in SR.

COUNT

The count of events in a group increases monotonically as events are added to said group. For this reason, little can be done when the predicate over the count is one of $<$, $\leq$, or $=$. However, if the comparator is one of $>$ or $\geq$, we can advantageously cleanse state and propagate.

Consider the case in which the comparator is one of $<, \leq, or =$. In this case, no fp' can be propagated safely, as any incoming event may increase the count and take the value for a particular group above the guard's condition. Avoiding these events would be incorrect, as the set of events used to compute a particular v could result in a new value, v', originally not in SR.

Now consider the case in which the comparator is $> or \geq$. We could intuitively examine the intermediate state, identify groups that already match or exceed the guarded quantity, and propagate those group IDs as feedback.

MIN

The minimum value of a specific attribute in a group of events can only decrease as events are added to said group. Assume H holds key value pairs $h = (g, v)$ where g is the grouping attributes and v is the current minimum.

Consider the case in which the comparator is one of $>, \geq, or =$. Similarly to `COUNT` and `AVERAGE`, no fp' can be propagated safely, as any incoming event may decrease the minimum value of its group below the guard's condition. Avoiding these events would be incorrect, as the set of events used to compute a particular v could result in a new value, v', originally not in SR.

Consider the case when the comparator is one of $< or \leq$. As we did with `COUNT`, perusing the intermediate state to find group ids for which the value is already below the guarded value, we could propagate those group IDs as feedback.

MAX

The `MAX` operator is handled similarly to `MIN`, but with the inequalities reversed.

4.3 REMARKS

In this chapter, I have characterized a set of response actions and propagation rules for commonly used operators in the NiagaraST algebra. We specifically described the effect responses have on events in an operator. I have shown the actions in terms of the constructs introduced in Chapter 3, namely input and output guards. Note that more actions can still be possible. For example, consider the `COUNT` operator. I have proposed a characterization in which one pays the cost of sweeping its state H and comparing it to a feedback punctuation fp only when said punctuation arrives. We then mount guards to avoid re-forming deleted groups, or emitting results which match the fps received. `COUNT` and `MIN` could also be amenable to state purges, provided one also creates an input guard to prevent removed groups from re-forming.

The proofs of correctness of propagation I showed benefited from having an exploit in place, so it was easy to show that the overall output of the operator remained unchanged as a consequence of the propagation. This need not be the case. For example, assume `SELECT` simply propagates the feedback it receives, without exploiting it. Proving the propagation is correct would then have to show that for a possible S' in which elements originally in S matching fp have been removed, the output of `SELECT` remains in the correct range, *i.e.,* $SR - subset(SR, fp) \subseteq SR' \subseteq SR$. The change here is that SR' materializes as a consequence of S', and not as a consequence of a local guard.

One additional exploitation side effect for aggregates can involve a state sweep every time there is a state change to compare the updated value to existing guards. Adding this check would enable cleansing state early and propagating new feedback referring to that state. This response can be proven correct, and opens up the possibility of scheduling opportunity-discoveries in these types of operators. I think a cost-based selection of responses could be a rather challenging undertaking, since benefit is dependent both on timing of feedback and additional measurable characteristics of the input stream. In my work, however, I have chosen not to explore the mechanics of optimizing such a system, but notice the framework does not prohibit sophisticated operator implementations. As I will show in Chapter 7, with these basic constructs it is already possible to observe significant processing savings.

Chapter 5

EXECUTION GUARANTEES

One of the main concerns in DSMS design is the accumulation of event-related state. As I mentioned in Chapter 1, punctuated data-stream processing has enabled DSMSs to cleanse event-related state as the stream progresses over time. While developing the feedback framework, I noticed that the technique introduces context-related state into the operators; operator guards may keep accumulating guard conditions as more and more feedback is sent to operators. Concerned about unbounded feedback-related state, I started development of a framework to guarantee this context-related state's correct disposal.

The initial intuition is that guard state can be cleansed exploiting the same mechanism as event-state: Punctuation. For example, if an operator is guarding on timestamps before "11:00 a.m.", incoming punctuation that states the current time is "past 11:00 a.m." can correctly trigger this guard's removal. The problem is if one wants to make a universal statement about managing state, one needs to know about future punctuation. In general, we have no guarantee about incoming punctuation describing attribute values (or derivatives of these, such as WIDs) on which operators may be blocking or accumulating state. In some systems such as CEDR [27], CTIs only refer to time-progressing attributes, hence, blocking or stateful processing associated with time *may be* safe, provided CTIs continue to arrive. I wish to make a similar safety claim about arbitrarily punctuated queries.

In order to reason about context-related state, I set out to understand the implications of punctuations. I developed a framework, *Inter-Operator Contracts*, that not only annotates streams with their expected punctuations, but also enables pre-evaluation analysis of a query to reason about its execution safety. In Sections 5.3 – 5.5, I present the framework from the perspective of regular punctuated stream processing. This derivation was previously published in Fernández-Moctezuma *et al.* [22]. In Section 5.6, I extend the framework to to describe feedback and guarantee removal of feedback-related state. In Section 5.7, I present a recursive algorithm to determine whether a query plan is safe to execute given its contracts. The derivation presented in this chapter uses a generic stream algebra, largely inspired by NiagaraST's. My intention is to show the broad applicability of the technique without tying it to a specific system.

5.1 MOTIVATING EXAMPLE

Consider two streams, `Msg1` and `Msg2`, that contain records of messages sent between operating system processes hosted in two separate processing units. Both streams have identical schemas `(timestamp, sourceID, destinationID)`, where the latter two attributes refer to process IDs, and processes are uniquely identified across the lifetime of the stream. One wishes to compute the messaging ratio of processes hosted in the processing unit from which `Msg1` comes with respect to processes in the other source, in 1-minute tumbling windows. For each source pair, this computation determines which of the two sources dominates messaging. A query plan that computes this ratio is in Figure 5.1.

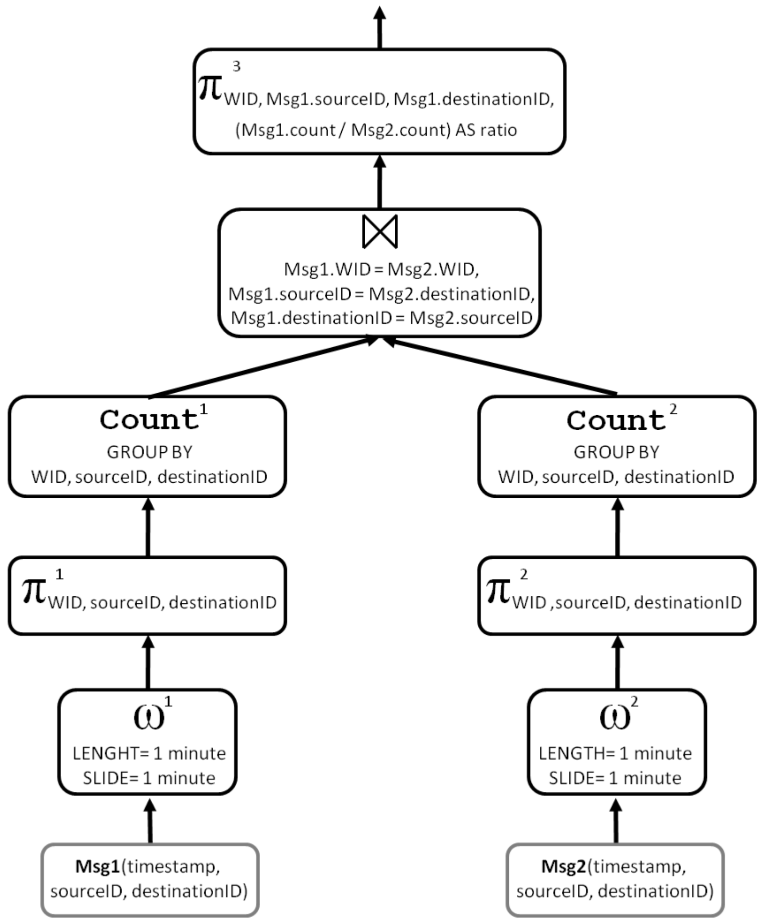

π³ WID, Msg1.sourceID, Msg1.destinationID, (Msg1.count / Msg2.count) AS ratio
⋈ Msg1.WID = Msg2.WID, Msg1.sourceID = Msg2.destinationID, Msg1.destinationID = Msg2.sourceID
Count¹ GROUP BY WID, sourceID, destinationID
Count² GROUP BY WID, sourceID, destinationID
π¹ WID, sourceID, destinationID
π² WID ,sourceID, destinationID
ω¹ LENGHT= 1 minute SLIDE= 1 minute
ω² LENGTH= 1 minute SLIDE= 1 minute
Msg1(timestamp, sourceID, destinationID)
Msg2(timestamp, sourceID, destinationID)

Figure 5.1: Sample streaming query plan to compare the ratio of sent to received messages among pairs of processes located in different CPUs over 1-minute tumbling windows. Operator instances are uniquely identified by a superscript.

For each stream, the count of messages, grouped by process pair, per window is computed separately. First, the `WINDOW` operator (ω) produces a window id (`WID`) for each event by applying its windowing function over the timestamp attribute. Next, the `PROJECT` operator (π) projects the timestamp out. The `COUNT` operator produces a count of the events seen in a given window for every pair of messaging processes. The two substreams are then window-joined, and the resulting substream is projected to compute the ratio.

Both instances of `COUNT` will block until the correct count can be emitted – i.e., when there is certainty that all elements in a particular group have been seen. `JOIN` ($\bowtie$) will accumulate state associated with both of its inputs until the state is not required to produce future results.

Since the data in the stream progresses through time, punctuation based on the `timestamp` attribute is intuitive. In NiagaraST, the `WINDOW` operator maps timestamp punctuation to `WID` punctuation [40]. `COUNT` can correctly produce results after seeing `WID` punctuation, and `JOIN` can cleanse state as time progresses. For example, consider the following stream in `Msg1`, where events are bracketed with "$< >$":

```
<‘9:59:30 a.m.’, 1, 2>
<‘9:59:45 a.m.’, 1, 3>
[≤‘9:59:59 a.m.’, *, *].
```

Assume the two events' timestamps map to the window with WID = 959. The output stream of the `WINDOW` operator has schema `(WID, timestamp, sourceID, destinationID)`, and contains the following events and punctuation:

```
<959, '9:59:30 a.m.', 1, 2>
<959, '9:59:45 a.m.', 1, 3>
[≤959, *, *, *].
```

`COUNT` processes its input, which causes it to track two groups, and holds result production until the aggregates are known to be complete – specifically, when the operator has certainty that no other events corresponding to current groups will arrive. The punctuation `[≤959, *, *]` (from `PROJECT`) closes the two existing groups, and `COUNT` outputs the stream with schema `(WID, sourceID, destinationID, count)`:

```
<959, 1, 2, count:1>
<959, 1, 3, count:1>
[≤959, *, *, *].
```

Assume the right input to `JOIN` is

```
<959, 2, 1, count:1>
[≤959, *, *, *],
```

JOIN produces the following result stream with schema (WID, sourceID, destinationID, Msg1.count, Msg2.count):

```
<959, 1, 2, 1, 1>
[≤959, *, *, *, *].
```

Stream progress in this example was tracked with punctuations covering the passage of time – in particular, punctuating events in the timestamp attribute. Progress in this example can also be measured differently. When a process terminates, a punctuation on events on the process ID attribute can be inserted into the stream. Let us assume that the process with ID = 1 terminates. The input stream on Msg1 is:

```
<‘9:59:30 a.m.’, 1, 2>
<‘9:59:45 a.m.’, 1, 3>
[*, 1, *].
```

The output stream of the `WINDOW` operator is

```
<959, ‘9:59:30 a.m.’, 1, 2>
<959, ‘9:59:45 a.m.’, 1, 3>
[*, *, 1, *].
```

After projection, `COUNT` processes its input. In this revised example, the punctuation `[*, 1, *]` (from `PROJECT`) closes the two existing groups, and `COUNT` outputs the stream

```
<959, 1, 2, count:1>
<959, 1, 3, count:1>
[*, 1, *, *].
```

Assume the right input to `JOIN` is

```
<959, 2, 1, 1>
[*, *, 1, *],
```

`JOIN` produces the following result stream with schema `(WID, sourceID, destinationID, Msg1.count, Msg2.count)`:

```
<959, 1, 2, 1, 1>
[*, 1, *, *, *].
```

This example shows that tracking progress on more than one attribute is possible. Tucker [62] provides additional examples of queries whose progress is not necessarily measured in time.

5.2 REASONING ABOUT UNBOUNDED STREAMS

One of the main challenges when processing unbounded streams consists in reasoning formally about all elements in a stream. Even if one could show that an operator produces all the right output once it sees all of its input, that guarantee has little value in practice, since one would like a query to behave well during execution, not just after seeing all input. This requirement suggests one needs a way to characterize "good behavior" incrementally.

We use a contracts framework to make statements about the relationship between an operator's input stream(s) and it output stream(s). One way to reason about processing consists in decomposing the stream into discrete, bounded subsets both at the input and the output. Let T be the set of all events (tuples) in a stream. Let h be a partition function that maps an event to a natural number, $h(t) \to \mathbb{N}$, defined over all possible events in the domain of T. Although the partition function is defined here as a mapping to the natural numbers, any other countable set (such as timestamps or processIDs) would work as well. Let us consider an unbounded stream S, and let T be the set of all events in S. We define a set H_i as the events $t \in T$ such that $h(t) = i$: $H_i(S) = \{t \in T | h(t) = i\}$. We say that stream S has a *Piece-wise Finite Decomposition* (PFD) with respect to h if every $H_i(S)$ is finite.

Consider an operator O that maps streams over domain $\mathbb{D}$ to streams over domain $\mathbb{E}$. O is *PFD-definable* if there exists h, j such that: whenever S is PFD with respect to h, $O(S)$ has a PFD relative to j, and each J_k of $O(S)$ depends on a finite number of H_i's from S. For binary operators, we consider their outputs J_k as dependent on a co-occurrence on finite subsets of its inputs, H_i^1, H_l^2, that is an output is the product of a pair of partitions.

These operator semantics allows one to consider finite subsets, each with finite events, as input to an operator, which produces finite subsets, each containing a finite number of events. With punctuation, it is possible to reason incrementally about the correctness of an implementation I for a PFD-definable operator O: If punctuation indicates J_k is complete on the output, then we can check that all corresponding H_i's are complete on the input and that J_k agrees with the definition of O. If we can further reason that punctuation guarantees that every J_k is eventually complete, then we know that I delivers all correct output per the definition of O. In punctuated data streams, punctuations allow analysis of any prefix of an unbounded stream to determine which input and output subsets have been closed.

It is not sufficient to characterize an operator's responses to events and punctuations alone in order to guarantee it will not hold on to state indefinitely, or that for a given punctuated data stream it will eventually produce all correct output. We also want to make statements about the incremental nature of stream processing, that is how operators work on incremental partitions of streams. In essence, this partitioning is what I propose as a more complete definition of correctness.

To illustrate the intuition of partitioning, consider the `SUM` operator, applied to the stream `s(w,v)`, where it sums `v`'s for every `w`. One way to hint at the causality intuition is to recognize `SUM` can only output an event for a given `w` once a punctuation that covers said group arrives. Punctuation over `w` is then decomposing the input stream in fixed pieces. Similarly on the output, punctuations delimit these natural partitions.

Consider the `SELECT` operator, operating on the input stream S, and producing an output stream O. Here, the partition function h maps each stream element (event and punctuation) to a unique natural number, based on event arrival. Any partition e in `SELECT`'s output corresponds to one partition d in its input, hence `SELECT` is PFD-definable.

For the rest of this exposition we consider only PFD-definable operators when characterizing their behavior with respect to punctuated data streams and provide workable contracts for each operator.

5.3 EXECUTION GUARANTEES

The presence of punctuation alone in a stream is no guarantee that a continuous query over that stream will execute successfully. In particular, it is no guarantee that all operators involved in the query will eventually unblock and release state. A continuous query plan will execute successfully if the following statements hold:

- No piece of state remains indefinitely in any operator.
- Every *correct* output will be delivered eventually.

For a given query, there could be no punctuation pattern that guarantees good behavior on the query, or there may be several patterns that allow it to *execute successfully*. Let us make the definition of successful execution more precise. For the first statement, I am concerned with event-dependent state. I disregard constant-size parts of operator data structures, such as a hash-table header or a variable holding a selection condition. For the second statement, we need to know that for every PFD piece of the output of an operator, all the constituent pieces have been received in the input. And in order to account for every PFD piece of the input, we need to know that they are covered by punctuation.

To develop this framework, I use operators that are PFD-definable relative to some PFD of their input streams, and assume that input streams meet that PFD condition. Common stream operators are all PFD-definable. Pointwise operators such as `SELECT` and `PROJECT` are easy to show as PFD-definable: each value in the input domain is in its own partition. A `JOIN` is PFD-definable if we can decompose it into a series of joins of finite subsets of the input. Examples of this property include a `JOIN` on WID where the number of messages per unit of time is finite, or a band join where any band has a finite number of events. Aggregates depend on the stream and the grouping. Time-based windowed aggregates are PFD-definable under the reasonable assumption that only a finite number of events can be received by a finite time.

Existing stream algebras support static query analysis to determine, among other things, if a query is syntactically correct. I aim to extend this query-level analysis to also determine query properties related to state management and output delivery, moving from single-operator processing guarantees (determined at operator design time) toward query-level successful execution guarantees determined at query construction. Figure 5.2 presents a high-level overview of the proposed technique.

To perform query-level analysis, one needs mechanisms to describe the style of punctuations being used to talk about the set of all punctuations expected in the stream, and to represent operator activity with respect to input and output punctuations. In the remainder of this section, I introduce a framework that addresses these requirements.

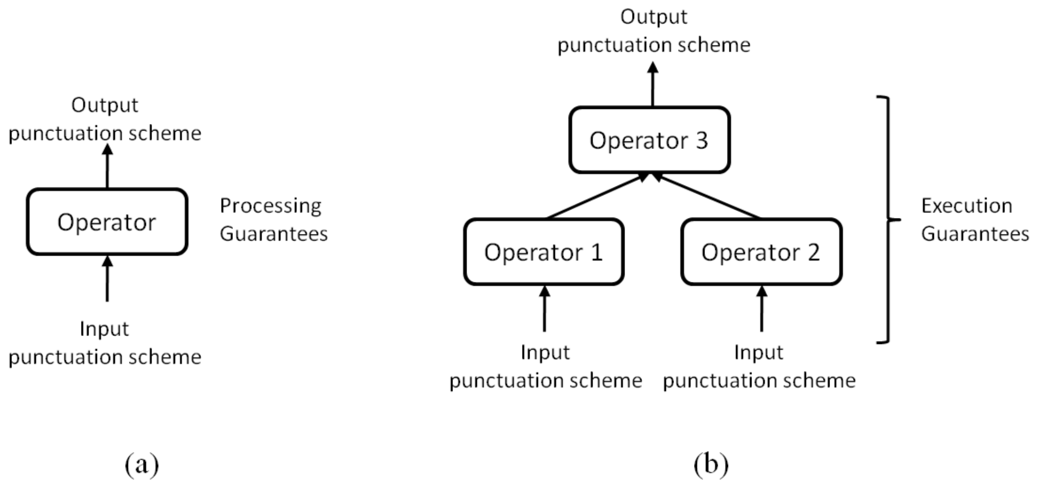


Figure 5.2: From operator processing guarantees (a) to query-level execution guarantees (b).

5.3.1 Punctuation Templates and Schemes

The example in Section 5.1 had two possible punctuation *types*: Punctuating on `timestamp` suggested stream progress based on advancement of time, while punctuating on a process ID gave a different notion of progress based on process completion. I seek to represent the form of punctuation that is expected in a data stream, specifying the types of input and output punctuation for operators. In the example in Section 5.1, I observed three *classes* of patterns in the punctuations: some patterns referred to "up to here" progress ($<$, $\leq$), others specified a particular value (such as a process ID), and a third class specified all values ("*"). For "up-to" conditions, I will use the symbol "+". For patterns that specify values, such as a process ID, I use the symbol "#", and use the symbol "-" for the "everything" pattern.

A *Punctuation Template* is a construct that restricts the syntactic form of the punctuations in a stream, with its format given in Figure 5.3. I require all attributes in the schema to be present in the punctuation template.

```
TEMPLATE    :=  [[ DESCRIPTOR
                (, DESCRIPTOR )*]]
DESCRIPTOR  :=  ATTRIBUTE:CLASS
ATTRIBUTE   :=  Attribute name in the schema
CLASS       :=  (+|#|-)
```

Figure 5.3: Punctuation template syntax.

For example, the punctuation template

```
[[a:+, b:#, c:-]]
```

allows the punctuation

```
[≤‘11:30 p.m.’, 26, *],
```

but not

`[*, 26, *]`,

`[‘11:30 p.m.’, <26, *]`, nor

`[≤‘11:30 p.m.’, 26, 3]`.

Formally, a punctuation p *conforms* to a template t if each component of p matches the corresponding description expressed by t.

To describe the set of punctuations expected in a stream, I define a *Punctuation Scheme* as a set of one or more punctuation templates. The set describes the styles of punctuations seen in a stream. `PS1` and `PS2` below are examples of punctuation schemes:

```
PS1 = {[[a:+, b:#, c:-]]},

PS2 = {[[a:+, b:-, c:-]], [[a:-, b:#, c:-]]}.
```

A stream `S` *obeys* a punctuation scheme PS if:

Condition 1. Any punctuation $p \in S$ conforms to *at least one* punctuation template $T \in PS$, and

Condition 2. For any event $e \in S$, and *each* template $T \in PS$, $\exists p \in S$ such that p conforms to T and e matches p. In other words, every stream element must be covered by a punctuation corresponding to each of the templates in the scheme.

The following sub-stream obeys $PS1$:

```
<‘9:45 p.m.’, 1, 7>
[<‘10:00 p.m.’, 1, *]
[<‘10:00 p.m.’, 2, *]
```

while the following does not, since it contains punctuations that do not match any templates in $PS1$:

```
<‘9:45 p.m.’, 1, 7>
[<‘10:00 p.m.’, *, *]
[*, 2, *]
[<‘10:05 p.m.’, <1, *]
```

A stream S obeys Condition 1 of $PS2$ if all its punctuations have either “up-to” content on attribute `a` or “value” content on attribute `b` (and “any value” content elsewhere for both cases). For example, the following sub-stream obeys Condition 1 of $PS2$:

```
<‘10:16 p.m.’,2,23>
<‘10:17 p.m.’,1,13>
<‘10:17 p.m.’,3,31>
```

```
[<‘10:20 p.m.’, *, *]
<‘10:21 p.m.’,1,11>
[*, 1, *]
<‘10:21 p.m.’,2,21>
<‘10:21 p.m.’,3,31>
[<‘10:25 p.m.’, *, *]
[*, 2, *]
<‘10:27 p.m.’,3,30>
[<‘10:30 p.m.’, *, *]
```

but it doesn't obey Condition 2 of $PS2$, since no punctuation covers the stream elements with the value '3' in the second attribute. The stream would if the punctuation `[*,3,*]` were to appear at the end, for example.

A related notion from the literature is *linear punctuation* [61] in which punctuations are strictly increasing on a timestamp attribute. This notion corresponds to a punctuation scheme with a single template that has one "up-to" component, such as `{[[time:+, value:-]]}`, plus the condition that punctuations are non-redundant. That is, we will not see punctuation `[<‘10:15 p.m.’, *]` after we have seen punctuation `[<‘10:30 p.m.’, *]`.

5.3.2 Punctuation Contracts

I have suggested that an operator can be prepared to process different punctuation schemes. A simple example is the windowing operator, ω. One processing mode consists of mapping punctuation in the windowing attribute (`timestamp` in the motivating example) to the Window ID attribute in the output. A second processing mode can produce windowed groups by grouping on additional attributes, such as `timestamp` *and* `sourceID`, and mapping punctuation that refers to those two attributes to a window id punctuation. To represent the capabilities of operators with respect to punctuation schemes, a *Punctuation Contract* is a record of punctuation schemes corresponding to each input and output of an operator. If an operator R is operating under contract CT and gets inputs that obey the input punctuation schemes in CT, then the following *guarantees* hold:

Guarantee 1: R produces an output that obeys the output punctuation scheme specified in CT,

Guarantee 2: No piece of event-state remains in R's state forever, and

Guarantee 3: R eventually produces all of its correct output, according to the notions presented in Section 5.2.

Consider the `SELECT` operator. An intuitive operating mode for select consists of outputting the punctuations seen on its input. The following contracts exemplify this behavior on an input stream with schema `s(a,b,c)`. I label each attribute name in this notation to make input-output correspondences clearer:

```
CT1 = <In={[[a:+,b:-,c:-]]},
        Out={[[a:+,b:-,c:-]]}>
CT2 = <In={[[a:+,b:-,c:-]], [[a:-,b:#,c:-]]},
        Out={[[a:+,b:-,c:-]], [[a:-,b:#,c:-]]}>
```

A contract CT for operator R is *realizable* if a physical implementation of R can be constructed such that the contract CT is guaranteed. For example, `SELECT` may have been designed to only propagate linear punctuation, discarding hash punctuation:

```
CT3 = <In={[[a:+,b:-,c:-]], [[a:-,b:#,c:-]]},
        Out={[[a:+,b:-,c:-]]}>
```

Similarly, one could realize a stateful version of `SELECT` that obeys the following contract:

```
CT4 = <In={[[a:+,b:-,c:-]], [[a:-,b:#,c:-]]},
        Out={[[a:+,b:#,c:-]]}>.
```

Such an implementation of `SELECT` could accumulate state associated with the `b` values from the events consumed. Once punctuation informs `SELECT` all events with a specific value of `b` have been seen, the operator can be ready to emit punctuation as long as it ensures that the emitted punctuation increases on `a`. Notice that unlike linear punctuation, we do not require strictly increasing punctuation to be emitted.

I distinguish multiple inputs in n-ary operators numerically. For example, a contract for `JOIN` over streams `s(a,b)` and `t(b,c)` on `b` can be expressed as:

```
CT5 = <In1={[[a:-,b:+]]}, In2={[[b:+,c:-]]},
        Out={[[a:-,b:+,c:-]]}>
```

Not all expressible contracts are realizable. Consider the following contract for `SELECT`:

```
CT6 = <In={[[a:+,b:-,c:-]]},
        Out={[[a:-,b:#,c:-]]}>.
```

`CT6` is unrealizable; there is no way to infer all events for particular values of `b` have been seen by only examining progress as expressed by values in attribute `a`.

A physical implementation of an operator might not include all realizable contracts. Moreover, during execution, there may be different physical implementations of the same logical operator in use. A *Contract Offering* for logical operator R is a set of deliverable contracts for which fulfillment is realizable by some implementation of R. For example, a contract offering for `SELECT` might be: `CO = {CT1, CT2, CT4}`, where the constituent contracts are as given above.

5.4 CONTRACTS FOR STREAM OPERATORS

In this section, I present realizable contract offerings for some stream operators. The list is not exhaustive, but illustrates operator modeling under the contracts framework. I also show how contract offering designs for operators can be synthesized for arbitrary input schemas by examining the operator's processing logic. For discussion, I will use the schemas `s(a,b)` and `t(b,c)`.

Since one is concerned with latency and state management, one needs to keep track of the attributes on which blocking and state accumulation occur. I call these *Interesting Attributes*. Examples include the joining attributes for `JOIN` and the grouping attributes in an aggregate.

5.4.1 SELECT

I used `SELECT` to guide our exposition of contracts in Section 5.3.2. Let us say that one is interested in modeling `SELECT`'s contract offerings as the identity on punctuation schemes. For schema `s`, one would need nine different contracts for each possible singleton punctuation scheme, one of which is `CT7` below. One would need to enumerate the powerset for all possible n-ary schemes.

```
CT7 = <In ={[[a:-,b:+]]}, Out = {[[a:-,b:+]]}>
```

Moreover, one needs a way to describe contracts independent of a specific schema. A *General Contract Form* (GCF) describes contracts for an arbitrary schema for an operator. In a GCF, I represent punctuation schemes or groups of attributes with variables. The general contract form `GCF1` below synthetizes the "identity" behavior of `SELECT`. In it, the variable PS is any punctuation scheme defined over `SELECT`'s input schema.

`GCF1 = <In=`PS`, Out=`PS`>.`

A GCF provides a blueprint to generate specific contract offerings. For example, if a physical instance of `SELECT` is presented with an input punctuation scheme `[[b:+, c:-]]`, the contract `CT8` can be derived from `GCF1`:

```
CT8 = <In={[[b:+,c:-]]}, Out={[[b:+,c:-]]}>}.
```

5.4.2 PROJECT

GCFs for an operator are implementation-specific. A straightforward implementation of `PROJECT` does not accumulate state or block output, and propagates punctuation when values in the punctuation appear *only* in the projected attributes. Let I be the input schema, X be the set of projected attributes and Y be the set of attributes projected out, hence $X \cup Y = I$. Let PSI be a punctuation scheme whose templates PSI_i only refer to attributes in X. Let PSO be a punctuation scheme in which each template $PSO_i \in PSO$ corresponds to exactly one template $PSI_i \in PSI$, with attributes in Y removed. The general contract form `GCF2` represents the described PROJECT behavior:

`GCF2 = <In=`PSI`, Out=`PSO`>`.

Consider an instance of `PROJECT` that takes a schema `s(a,b)` as input and projects out the `b` attribute. Suppose the input punctuation scheme is `[[a:+, b:-]]`. The following contract can be derived from `GCF2`:

```
CT9 = <In={[[a:+, b:-]]}, Out={[[a:+]]}>}.
```

5.4.3 JOIN

Consider a symmetric hash-join implementation of equi-join. From the sample streams `s` and `t`, one can derive the following realizable contracts:

```
CT10 = <In1={[[a:-,b:+]]}, In2={[[b:+,c:-]]},
        Out={[[a:-,b:+,c:-]]}>, and
CT11 = <In1={[[a:-,b:#]]}, In2={[[b:#,c:-]]},
        Out={[[a:-,b:#,c:-]]}>.
```

Both `CT10` and `CT11` are realizable since punctuation arrives on both inputs and in the join attribute only, which guarantees removal of the associated entries in each side's state. Contrast this behavior with `CT12`, another expressible contract:

```
CT12 = <In1={[[a:#,b:-]]}, In2={[[b:-,c:#]]},
        Out={[[a:#,b:-,c:#]]}>,
```

`CT12` is not realizable, as it fails to provide Guarantee 2 (but it does meet Guarantees 1 and 3) from Section 5.3.2 – there is no mechanism to clear state associated with the join attribute by looking only at punctuation values on the other attributes. The set of interesting attributes for `JOIN` contains all attributes named in the join condition. One can synthesize a generalized contract as follows: Let $I1$

and $I2$ be the input schemas to the operator. Let J be the set of joining attributes ($J \subseteq I1, J \subseteq I2$). Let L and R be the sets of attributes exclusive to inputs 1 and 2 respectively: $L = I1 - J, R = I2 - J$. One can synthesize the described behavior of JOIN with the following GCFs:

```
GCF3 = <In1={[[L:-,J:+]]}, In2={[[J:+,R:-]]},
        Out={[[L:-,J:+,R:-]]}>
GCF4 = <In1={[[L:-,J:#]]}, In2={[[J:#,R:-]]},
        Out={[[L:-,J:#,R:-]]}>.
```

Consider the streams `l(a,b)` and `r(b,c)`. Consider $l \bowtie r$. Out of `GCF3` and `GCF4` we can derive a specific contract offering for `JOIN`:

```
CT13 = <In1={[[a:-, b:+]]}, In2={[[b:+, c:-]]},
        Out={[[a:-, b:+, c:-]]}>
CT14 = <In1={[[a:-, b:#]]}, In2={[[b:#, c:-]]},
        Out={[[a:-,b:#,c:-]]}>.
```

These are not the only possible GCFs for `JOIN`, nor is it required for all of the join attributes to be punctuated in the same way for an implementation of join to offer realizable contracts. Consider the following contract for schemas `l(a,b,c)` and `r(b,c,d)`:

```
CT15 = <In1={[[a:-, b:+, c:-]]}, In2={[[b:-, c:+, d:-]]},
        Out={[[a:-, b:+, c:+, d:-]]}>.
```

`CT15` is realizable since the left input's punctuation on `b` is sufficient to clear the right input's state, similarly for the other sides on `c`. The additional operational requirement is to hold punctuation production until there is progress from both sides. Notice one attribute can progress more frequently than the other (both attributes are increasingly progressing).

5.4.4 WINDOW

In NiagaraST, the windowing operator applies a windowing function f to incoming events. The windowing function maps values from a subset of the input schema's attributes to a new attribute, called the window ID (WID). This functional application may block or accumulate state, as discussed by Li [38]. As a simplifying assumption, I consider simple functional application with no state accumulation or blocking. For example, for the stream with schema s, a windowing function $f(b) = \lceil b/10 \rceil$ would operate on this stream:

```
<a:1, b:19>
<a:2, b:21>
```

and produce the following output:

```
<WID:2, a:1, b:19>
<WID:3, a:2, b:21>
```

For punctuation, `WINDOW` typically maps linear punctuation on its windowing attribute to the WID attribute. This map applies the windowing function $f'(b) = \lceil b/10 \rceil - 1$ to the punctuation value in b. The use of f' guarantees we are punctuating only on clearly closed windows, and not windows in flight. For example, for the input punctuation `[a:*,b:22]`, `WINDOW` would output `[WID:2,a:*,b:*]`. Let us say that W is the set of windowing attributes in its input schema, and X all other attributes. A generalized contract for `WINDOW` is

`GCF5 = <In={`W`:+, `X`:-},`

`Out={WID:+, `W`:-, `X`:-}>`, from which one can derive a contract for `s(a,b)` where `b` is the windowing attribute:

```
CT16 = <In={a:-, b:+},
        Out={WID:+, a:-, b:-}>.
```

5.4.5 COUNT

As a representative of aggregate operators, consider the grouped `COUNT` operator. `COUNT` accumulates state organized by its grouping attributes, one of which can be a `WID`. For example, if one counts the b's per a's seen in stream s, one cannot correctly emit a count until one knows all events for a value of a are seen. The output of the operator is the grouping attributes plus the result of the calculation. Let I be the input schema, $G \in I$ be the set of grouping attributes, and $R = I - G$ be the rest of the attributes. The following GCFs can be used to generate realizable offerings, since in all cases punctuation can be used to clear the associated state and emit output:

```
GCF6 = <In={[[G:+, R:-]],},
        Out={[[G:#, COUNT(R):-]]}>
GCF7 = <In={[[G:#, R:-]]},
        Out={[[G:#, COUNT(R):-]]}>
GCF8 = <In={[[G:+, R:-]], [[G:#, R:-]]},
        Out={[[G:#, COUNT(R):-]]}>.
```

The following contracts are specific examples of the three aforementioned GCFs:

```
CT17 = <In={[[a:+, b:-]],},
        Out={a:#, COUNT(R):-}>
CT18 = <In={[[a:#, b:-]]},
        Out={a:#, COUNT(R):-}>
CT19 = <In={[[a:+, b:-]], [[G:#, R:-]]},
        Out={a:#, COUNT(R):-}>.
```

Recall the proposed list of GCFs is not exhaustive; as it is the case for join, contracts in which only a subset of the grouping attributes are punctuated are also realizable. Also note that `GCF6` can in fact describe contracts that are unlikely to be found in practice, for example, when there are two attributes in the grouping key. The following is more precise generalized contract form in which $g \in I$ and g is the grouping key:

```
GCF9 = <In={[[g:+, R:-]],},
        Out={[[g:+, COUNT(R):-]]}>.
```

5.5 FULL-QUERY ANALYSIS

Once a query plan is registered to the DSMS, I wish to verify that its execution will uphold the three guarantees posed in Section 5.3.2. I assume operator instances in the plan first instantiate their contract offerings and bind them to their specific input and output schema. A *contract accordance* is a selection of one contract offering per operator. A *consistent contract accordance* is an accordance where each antecedent operator's output punctuation scheme matches its subsequent operator's input punctuation scheme. A query can execute successfully if it has at least one consistent contract accordance.

Testing a single specific contract accordance for consistency is easy. Consider a query with two operators, `A` and `B`, with contract offerings `COA` and `COB` respectively. Assume `B` is the subsequent operator of `A`. Consider the following offerings:

```
COA={ <In={[[a:+, b:-]]}, Out={[[a:+, b:-]]}>,
      <In={[[a:#, b:-]]}, Out={[[a:#, b:-]]}>}
COB={ <In={[[a:+, b:-]]}, Out={[[a:+, b:-]]}>}.
```

In this case, two possible accordances exist. I represent them as pairs (α,β) where α is a contract from `A` and β is a contract from `B`:

```
(<In={[[a:+, b:-]]}, Out={[[a:+,b:-]]}>,
 <In={[[a:+, b:-]]}, Out={[[a:+,b:-]]}>),
(<In={[[a:#, b:-]]}, Out={[[a:#,b:-]]}>,
 <In={[[a:+, b:-]]}, Out={[[a:+,b:-]]}>).
```

Of these accordances, the first one is a consistent contract accordance, as the antecedent's output punctuation scheme matches the subsequent's input punctuation scheme.

Finding a consistent contract accordance for a query given a set of offerings requires more work. A simple approach consists of computing all possible accordances and testing them individually – a task that entails analyzing the cross-product space of contract offerings. Notice that finding a consistent accordance between two consecutive operators is equivalent to computing an equijoin of contract offerings on the antecedent's output punctuation scheme and the subsequent's input punctuation scheme. One can see contract offerings as tables, with each contract offered as a row. I illustrate this representation in Figure 5.4, with a new contract offering `COC` for operator `C`.

Consider a query plan with C as the antecedent of A, and A as the antecedent of B. The expression $COC \bowtie_{COC.Out=COA.In} COA \bowtie_{COA.Out=COB.In} COB$ will be empty if no consistent accordances are found, or will return a consistent accordance per row if not empty. If one assumes queries are trees, an efficient evaluation of this expression is achievable via *full reducers* [44].

Bernstein *et al.* showed that a class of queries, specifically tree queries, are efficiently solvable by full reducers [10]. By using semi-joins, one can reduce the number of tuples that are involved in the evaluation of a query. A full reduction of a database is then the smallest possible reduction of a database for a given query. A full reducer is a semi-join program that is guaranteed to produce a full

COA	In	Out
	{[[a:+, b:-]]}	{[[a:+,b:-]]}
	{[[a:#, b:-]]}	{[[a:#,b:-]]}

COB	In	Out
	{[[a:+, b:-]]}	{[[a:+,b:-]]}

COC	In	Out
	{[[a:+, b:-]],[[a:#, b:-]]}	{[[a:+,b:-]],[[a:#, b:-]]}
	{[[a:+, b:-]]}	{[[a:+,b:-]]}
	{[[a:#, b:-]]}	{[[a:#,b:-]]}

Figure 5.4: Database representation of operator offerings COA, COB, and COC for operators A, B, and C, respectively.

reduction. A full reduction is non-empty if and only if the join is non-empty. For this example, the full reductions are shown in Figure 5.5, and the result of the join, a consistent accordance for the query, is shown in Figure 5.6.

For a tree with n nodes, If we were to simply perform joins to find consistent accordances, assuming the largest cardinality of offerings is k, we would expect to do $O(k^n)$ work. By reducing the space of offerings to only those that would participate in a join, *i.e.,* by finding a full reduction, we can reduce the cost to find one accordance to linear time in the number of nodes. The cost of using the full reducers is $O(nk^2)$, since the cardinalities in each step are guaranteed to only

COA-R	**In**	**Out**
	{[[a:+, b:-]]}	{[[a:+,b:-]]}

COB-R	**In**	**Out**
	{[[a:+, b:-]]}	{[[a:+,b:-]]}

COC-R	**In**	**Out**
	{[[a:+, b:-]]}	{[[a:+,b:-]]}

Figure 5.5: Full Reduction of operator offerings COA, COB, and COC for operators A, B, and C, respectively.

reduce in size in each step, and we need only $2n - 2$ semijoins on a tree of n nodes.

5.6 EXTENDING THE CONTRACT FRAMEWORK TO SUPPORT FEEDBACK

I have presented a framework that allows us to check for expected behavior of a continuous query before it is executed, aimed towards providing the three guarantees discussed in Section 5.3.

As I mentioned before, we started investigating this problem motivated by the need to remove feedback-related state from the operators, such as a guard an operator may mount as a response to feedback. In this section, I extend the framework for use with assumed feedback punctuation as follows: (1) Inclusion of feedback-related state elimination in the execution guarantees, (2) addition of a definition

Consistent Accordance	**COC.In**	**COC.Out**	**COA.In**
	{[[a:+,b:-]]}	{[[a:+,b:-]]}	{[[a:+,b:-]]}
	COA.Out	**COB.In**	**COB.Out**
	{[[a:+,b:-]]}	{[[a:+,b:-]]}	{[[a:+,b:-]]}

Figure 5.6: Consistent Accordance for the "C, A, B" query.

of *coverage* of feedback punctuations by regular punctuation, (3) extension to the syntax and semantics of the contract framework to describe feedback, and (4) an extension to the query-analysis technique to account for feedback.

5.6.1 Revised Guarantees

In the revised contract framework, one needs to account for a new guarantee. If an operator R is operating under contract CT, the following guarantees hold:

Guarantee 1: R produces an output that obeys the output punctuation scheme specified in CT,

Guarantee 2A: No piece of event-state remains in R's state forever, and

Guarantee 2B: No piece of feedback-state remains in R's state forever, and

Guarantee 4: R eventually produces all of its correct output, according to the notions presented in Section 5.2.

While Guarantee 3 is stated as in the original framework, correct output now refers to the definition of correctness when exploiting feedback, *i.e.*, some portion of the output may be dropped.

5.6.2 Extended Syntax of Punctuation Schemes

To represent feedback punctuation schemes I will extend the punctuation template syntax used for the contract framework by a new symbol: $\circlearrowleft$. When this symbol substitutes for a template, it means that an operator is able to accept any incoming feedback. This symbol is introduced to distinguish the case where an operator cannot accept any feedback, in which case we use $\emptyset$.

```
TEMPLATE    :=  ↺ | [[ DESCRIPTOR
                (, DESCRIPTOR )*]]
DESCRIPTOR  :=  ATTRIBUTE:CLASS
ATTRIBUTE   :=  Attribute name in the schema
CLASS       :=  (+|#|-)
```

Figure 5.7: Revised punctuation template syntax.

5.6.3 Extended Contract Representation

In addition to enumerating input and output punctuation schemes, contracts will now mention the type of feedback they can receive and emit. For example, consider the following contract for the `PROJECT` operator:

```
CT20 = <In ={[[a:-, b:+, c:-]] Out = {[[a:-, b:+]]},
        Feedback.In = {[[a:-,b:+]]}, Feedback.Out = {[[a:-, b:+, c:-]]}>
```

Contract `CT20` describes the input and output punctuation schemes as well as the incoming and outgoing feedback schemes supported. Contract $CT21$ below describes the case where any feedback punctuation is accepted, but no feedback is propagated:

```
CT21 = <In ={[[a:-,b:+]] Out = {[[a:-,b:+]]},
        Feedback.In = ↺, Feedback.Out = ∅>
```

5.6.4 Consistent Accordances with Feedback

A further refinement to the contract framework consists in working out how accordances are formed. First, let us remember that by construction in the original framework we are guaranteed domain coverage for attributes that progress linearly or are punctuated with a specific value ("+" and "#", respectively). This observation enables us to introduce a notion of *feedback coverage*. A feedback punctuation f is covered by the punctuations described by a template p if each attribute tagged with "+" or "#" in f is also marked with the same tag in a punctuation in p. For example, the feedback punctuation $\neg$`[a:`$\leq$ `‘2:00 a.m.’, b:*]` is covered by the template `[[a:+, b:-]]`.

Recall that feedback-related state is directly linked to the feedback punctuation that created it. If an input punctuation covers the state created by a feedback punctuation, said state can be purged. For example, if an operator's input guard has a record for $WID < 2$, and it receives forward punctuation with $WID < 3$, the guard record for $WID < 2$ can be removed.

Contract offerings with feedback are *well-formed* if the feedback they receive as input (Feedback.In) is covered by punctuations in its data input (In).

A second refinement requires changing the notion of equality to account for the symbol $\circlearrowleft$, which now matches "any", including an empty scheme. The *match* operator $\preceq$ compares punctuation schemes, including the empty set and the "any" scheme, as defined in Table 5.1. Notice the $\preceq$ operator does not commute – $R.scheme \preceq S.scheme$ does not necessarily imply $S.scheme \preceq R.scheme$.

A consistent accordance using feedback is now defined not only by matching output schemas with input schemas among operators, but also by examining the Feedback.Out and Feedback.In schemas in the offerings. I call this requirement the *match condition.* For example, consistent accordances for unary antecedent operator S and unary subsequent operator R exist if $S.Out = R.In \wedge S.Feedback.In \preceq R.Feedback.Out$. N-ary operators need to expand this match condition for each input-output relation.

Antecedent	**Subsequent**	**Result**
$\emptyset$	$\emptyset$	`True`
$\emptyset$	$\circlearrowleft$	`False`
$\emptyset$	s	`False`
$\circlearrowleft$	$\emptyset$	`True`
$\circlearrowleft$	$\circlearrowleft$	`True`
$\circlearrowleft$	s	`True`
s	$\emptyset$	`True`
s	$\circlearrowleft$	`False`
s	s'	`True` if $s = s'$, `False` otherwise.

Table 5.1: Definition of the *match* ($\preceq$) operator, applied as antecedent $\preceq$ subsequent. The letter s denotes a punctuation scheme.

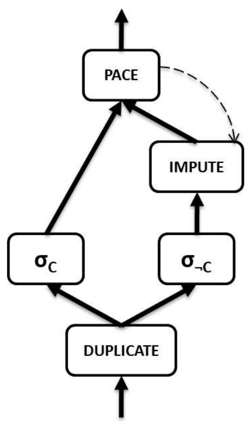


Figure 5.8: `PACE` communicates downstream context information to `IMPUTE`.

To illustrate accordance usage under the proposed feedback-augmented contract framework, consider the query plan shown in Figure 5.8, which we saw before in Chapter 1. The schema `sensor(time:timestamp, id:integer, speed:float)` and the contract offerings shown in Figure 5.9.

The aforementioned contract offerings form a consistent accordance for the query plan in Figure 5.8. Notice how all operators in the query plan have offerings that both (a) have Feedback.In covered by the punctuations present in their input, and (b) all operator pairs have contracts that satisfy the match condition.

5.7 FINDING CONSISTENT ACCORDANCES

Since I have modified the notion of equality of schemes by introducing the $\preceq$ operator, one cannot directly cast the accordance-finding problem to a join program, as done in 5.5. The structure of the search however remains quite similar, since

```
DUPLICATE  =  {<In = {[[time:+, id:-, speed:-]]},
               Out1 = {[[time:+, id:-, speed:-]]},
               Out2 = {[[time:+, id:-, speed:-]]},
               Feedback.In1 = ↺, Feedback.In2 = ↺,
               Feedback.Out = ∅>}
  SELECT1  =  {<In = {[[time:+, id:-, speed:-]]},
               Out = {[[time:+, id:-, speed:-]]},
               Feedback.In = ↺, Feedback.Out = ∅>}
  SELECT2  =  {<In = {[[time:+, id:-, speed:-]]},
               Out = {[[time:+, id:-, speed:-]]},
               Feedback.In = ↺, Feedback.Out = ∅>}
   IMPUTE  =  {<In = {[[time:+, id:-, speed:-]]},
               Out = {[[time:+, id:-, speed:-]]},
               Feedback.In = {[[time:+, id:-, speed:-]]},
               Feedback.Out = ∅>}
     PACE  =  {<In1 = {[[time:+, id:-, speed:-]]},
               In2 = {[[time:+, id:-, speed:-]]},
               Out = {[[time:+, id:-, speed:-]]},
               Feedback.In = ∅,
               Feedback.Out = [[time:+, id:-, speed:-]]>}
```

Figure 5.9: Contract offerings for the query in Figure 5.8.

one still performs it over a query tree. The match condition has more predicate elements than the original equality condition, which only tested schemes on output and input.

Algorithm 1 returns true if consistent accordances exist for a query plan where operators have multiple offerings. The general intuition is to begin the search at the top of the query plan, prune the offerings with each operator's antecedent (children) operators using Algorithm 2, recurse on each child, and reduce the set of offerings again. This procedure preserves the spirit of a Full Reducers approach, by traversing the tree downward and reducing the number of candidate tuples, shipping the results upward. The implementation of these procedures was straightforward in C♯. A reference implementation and examples are available at `http://www.cs.pdx.edu/~rfernand/dissertation.html`.

The proposed approach to find consistent accordances over query trees could be extended to include DAGs, albeit not without issues. DAGs can occur in queries if we use operators such as `DUPLICATE`. While we could turn the DAG into a tree, there could be an exponential blow up in the number of nodes in the tree, potentially cancelling the savings experienced by using Full Reducers.

So far, I have discussed the intuition behind the thesis, introduced both the feedback model and the contracts framework, and defined the associated notions of correctness. The next two chapters of the thesis look at the implementation and evaluation of the feedback framework, specifically assumed punctuation, in the

Algorithm 1 Consistent Accordance and Reduction

```
1: procedure ConsistentAccordance($root$)  ▷ Traverses the query plan and returns true if consistent accordances exist.
2:     Reduce($root$);
3:     return ($root.Contracts! = \emptyset$);
4: end procedure

5: procedure Reduce($node$)  ▷ Reduces a node's contract offerings.
6:     for all $c \in node.Children$ do
7:         $node.Contracts \leftarrow$ Match($node.Contracts, c.Contracts$);
8:         Reduce($c$);
9:         $node.Contracts \leftarrow$ Match($node.Contracts, c.Contracts$);
10:    end for
11:    if $node.Parents! = \emptyset$ then
12:        for all $p \in node.Children$ do
13:            $p.Contracts \leftarrow$Match($p.Contracts, n.Contracts$);
14:        end for
15:    end if
16: end procedure
```

Algorithm 2 Scheme matching

```
1: procedure MATCH($(a, b)$)          ▷ Returns the set of contracts in a that have matches in s
2:     $result \leftarrow \emptyset$
3:     for all $c \in a.Contracts$ do
4:         for all $d \in s.Contracts$ do
5:             if $c.Out = d.In \wedge c.Feedback.In \preceq d.Feedback.Out$ then
6:                 $result \leftarrow result \bigcup c$
7:             end if
8:         end for
9:     end for
10:    return $result$
11: end procedure
```

NiagaraST DSMS. In Chapter 6 I discuss concepts and architecture of NiagaraST, and detail the changes made to implement the feedback framework. Chapter 7 describes an experimental evaluation of these ideas.

Chapter 6

SYSTEM DESIGN AND ARCHITECTURE

NiagaraST is a stream-processing system in active development at Portland State University[1]. The system is a derivative of the Niagara project at the University of Wisconsin–Madison and the Oregon Graduate Institute [47]. NiagaraST specialized in punctuated data stream processing [62], advanced windowed-query processing techniques [39, 40, 41], stream and archive queries [64], and out-of-order stream processing [42]. The system is written in Java.

This research extends the NiagaraST system in two ways: First, it enables the creation of operators to support assumed feedback punctuation, and second, it enables operators to be instrumented to measure relevant statistics. In this chapter, I describe the general architecture of NiagaraST, focusing on the features that made it a good vehicle for the implementation of feedback (such as existing support for punctuation processing). I also detail the changes that went into the system, and describe design decisions around the feedback-punctuation mechanism.

[1]`http://datalab.cs.pdx.edu/niagaraST/`

6.1 NIAGARAST SYSTEM PRIMER

The NiagaraST system is a derivative of the Niagara Internet Query System from the University of Wisconsin–Madison [47]. The Niagara Internet Query System set out to find XML files in the Internet relevant to an issued query, recognizing that various sources may be unreliable, that different sources provide information at varying rates, and that the stream of inputs has no a priori bound. Specifically, the system works on streams of tuples of XML structures. The architecture accounted these difficulties, which are similar to the difficulties encountered in stream processing. NiagaraCQ [18] set out to provide a more complete support for continuous queries by leveraging a trigger system. NiagaraST evolved from these architectures, and was specialized for stream processing. Architectural changes addressed stream-specific issues, such as punctuated data stream processing, window operations, and various performance enhancements.

Queries are expressed in an XML-based query language called XML-QL. In XML-QL, a query is written as a tree of operators. This specification includes the source of the data stream and its internalization into the system. Let us examine the query file in Figure 6.1, also represented as a tree in Figure 6.2. First, notice how a stream is internalized from a text file data source. The `FILESCAN` operator reads input from an XML file. This input is then *unnested* by a series of instantiations of the `UNNEST` operator, which extracts an XML element and expresses it as one of the supported types in NiagaraST. In this example, in order to parse the selected stream, seven physical operators are instantiated. I bring this overhead to the reader's attention, as the current implementation of the NiagaraST system suffers from parsing costs, in particular due to the number of operators required to unnest XML data, and the cost of parsing XML. Nicola *et al.* report on the negative effect of XML parsing in data processing systems [50]. For clarity, an equivalent query in LINQ to XML is shown in Figure 6.1

The next part of the query plan in Figure 6.1 is a `SELECT` operator with a simple predicate. Its input is the last node in the unnesting hierarchy, and its output is consumed by the `CONSTRUCT` operator, which defines the output event elements.

There are logical and physical operators in NiagaraST. Logical operators describe, in general, what the operator does. A physical operator is a specific way of achieving the computation intended by some logical operator. A query plan is a tree built out of physical operators. Each operator runs in its own thread. There is no explicit scheduler in NiagaraST, rather, the system relies on the OS thread scheduler to switch control to the various active operators. An operator's main loop occurs in the `run` method of the thread. In general, all operators implement the following main execution loop shown in Algorithm 3.

The query processing architecture in NiagaraST interconnects query operators in two ways: via a data queue, where events and punctuations are placed, and via a control channel, through which operational messages are sent. The data queue is divided in pages. Each page contains a number of events and punctuations up

```
<?xml version="1.0"?>

<!DOCTYPE plan SYSTEM "queryplan.dtd">

  <plan top="cons">

    <filescan id ="data" isstream="yes"

     delay="0" filename="C:\data.xml"/>

    <unnest id="detectors" regexp="detectors"

     datatype="XML" input="data"/>

    <unnest id="time_t" regexp="time_t"

     datatype="TS" input="detectors"/>

    <unnest id="detector" regexp="detector"

     root="$detectors" datatype="XML" input="time_t"/>

    <unnest id="speed" regexp="speed"

     root="$detector" datatype="Integer" input="detector"/>

    <select id="se" input="speed">

     <pred op="gt">

       <var value="$speed" /><number value ="10" />

     </pred></select>

    <construct id="cons" input="se">

     <![CDATA[<result> $detector_id $time_t</result>

     ]]> </construct></plan>
```

Figure 6.1: Simple SELECT query plan in XML-QL.

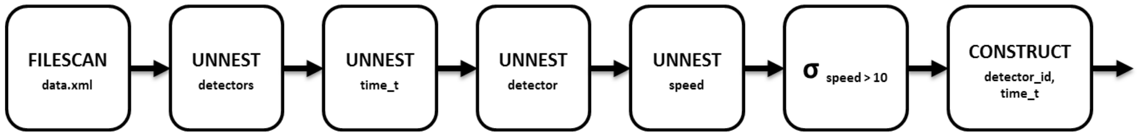


Figure 6.2: Detailed Representation of the XML query plan. For simplicity and to focus on the algebraic operators, `FILESCAN`, `UNNEST`, and `CONSTRUCT` have been shorthanded in all other query plans in this thesis.

```
var xml = XDocument.Load(@"C:\data.xml");
var query = from d in xml.Descendants("detector")
            where (int)d.Element("speed") > 10
            select new
            {
                 time_t      = (string)d.Parent.Element("time_t"),
                 detector_id = d.Element("detector_id")
            };
```

Figure 6.3: Simple `SELECT` query plan in LINQ to XML.

Algorithm 3 OperatorRun method

```
1: procedure Run                          ▷ Main execution loop of physical operators
2:     Initialize;                        ▷ Bindings, stream registration.
3:     while sources are not closed do
4:         $t \leftarrow$ getInput;
5:         process($t$);    ▷ Processing differs if input is a tuple or a punctuation.
6:         $cS \leftarrow$ getControlMessageFromSubsequentOperator;
7:         processCtrl($cS$);
8:         $cA \leftarrow$ getControlMessageFromAntecedentOperator;
9:         processCtrl($cA$);
10:    end while
11: end procedure
```

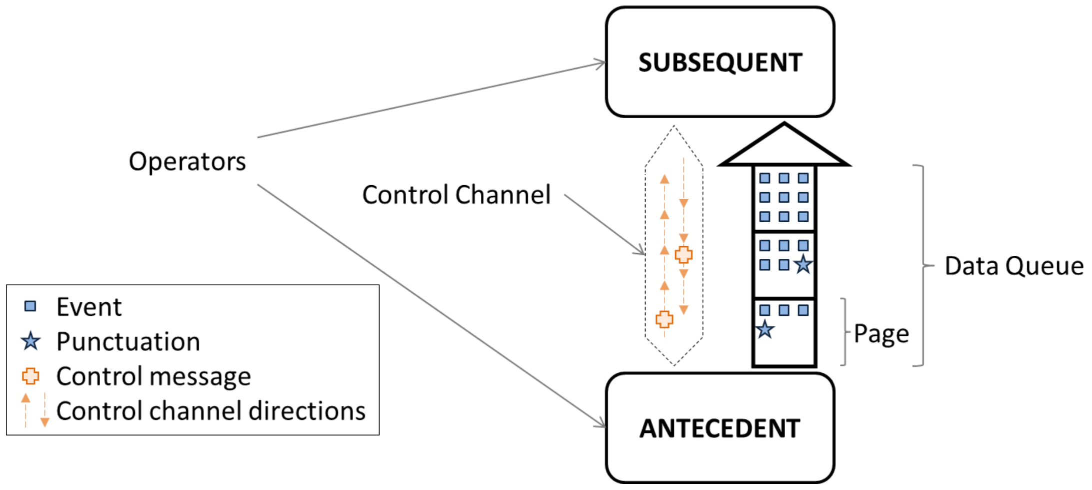


Figure 6.4: Inter-operator communication in the NiagaraST system. Operators are interconnected by a data queue and a control channel. A data queue contains a fixed number of pages. A page contains a maximum number of events and punctuations. The control channel carries control messages such as "end of stream".

to a fixed maximum. In NiagaraST, an event is the minimum data transfer unit between operators, and punctuations are expressed as specializations of events. Control messages in the NiagaraST system are inherited from the original Niagara architecture. Control messages (such as "get partial" or "end of stream") are used to trigger special system functions, such as emitting partial aggregates or shutting down a query. Events and punctuations flow unidirectionally from antecedent to subsequent operator (i.e., "stream direction"), while the control messages channel is bi-directional. For example, the "end of stream" control message always flows downstream, while the "get partial" message always flows upstream. An illustration of this architecture is shown in Figure 6.4.

As mentioned before, the control messages received are used to initiate system-related activities, such as shutting down the operator and closing its output stream. Shanmugasundaram *et al.* extended this basic functionality to also trigger production of partial results in the context of querying data sources over the Internet [36]. Tufte *et al.* exploited this mechanism to make data requests to an operator that performed adaptive queries over archived data [64].

6.2 INTER-OPERATOR COMMUNICATION

In an earlier class project, I looked at ways in which stream systems could support Intelligent Transportation System needs. One such need is imputation of missing or dubious values. I built an operator that leveraged archived data to provide online estimates to impute replacements for missing values. It was here where I first observed that such an operator was very expensive and delayed tuples considerably, sometimes to the point where the output was not useful if such a system were to support a live application. This is the scenario I described in Chapter 1. While trying to improve that particular query, I decided to simply send a new type of control message to the imputation operator, similar to the messages used for the archival query by Tufte *et al.* [64]. I later decided to make the signal more expressive, and refer to the timestamp of earliest interest to the query output. I realized at this moment we could communicte richer messages if we thought of them as types of punctuation.

I built an early prototype of the feedback machinery to demonstrate the ideas presented in Fernández-Moctezuma *et al.* [24]. In that prototype, feedback punctuations were a new type of message, with the control message string containing a *flattened* text-representation of the feedback punctuation. For example, we would send the string `#5#*#*#` to denote that events under a 3-attribute schema with a value less or equal than 5 in the first attribute were not being processed by the subsequent operator. With this simplification, we could demonstrate the benefit of sending contextual information upstream in some scenarios.

Conceptually, I extended the NiagaraST architecture to support a new type of contextual description that flows between operators. This revised architecture is shown in Fig 6.5. My approach consists in sending an object, rather than a flattened message, as feedback punctuation. This class does not specialize the Punctuation class in NiagaraST, because it must support additional comparators. The current implementation of punctuation in NiagaraST always checks for equality. My use cases required support for comparators such as greater or equal. Creating a new class and localizing these changes minimized the surface area and effect in the rest of the code base. A class diagram is shown in Figure 6.6.

A second extension to NiagaraST required by the proposed research is extending operators to summarize context and propagate it as feedback, as well as receiving feedback and exploiting it. While discovery is heavily dependent on the operator's main function (as discussed in Chapter 3), the repertoire of actions to exploit

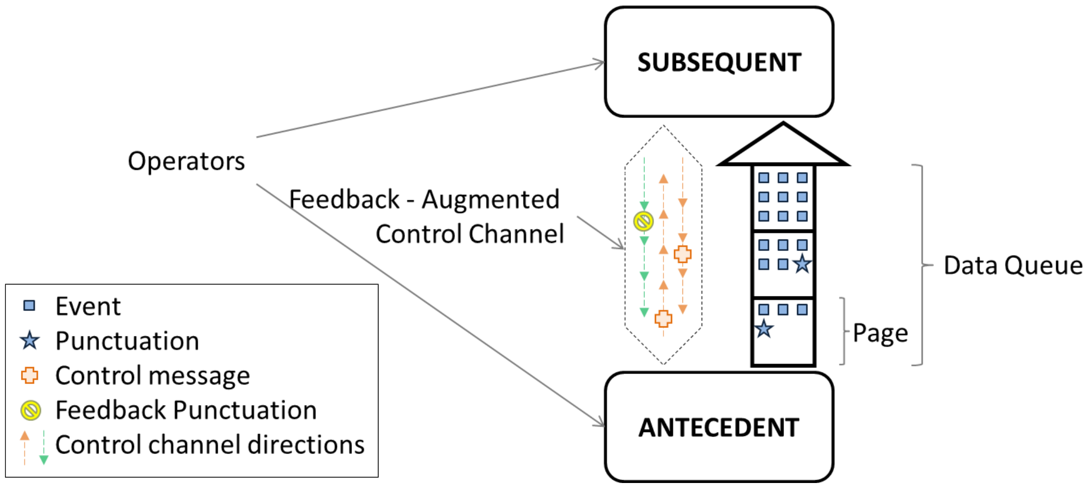
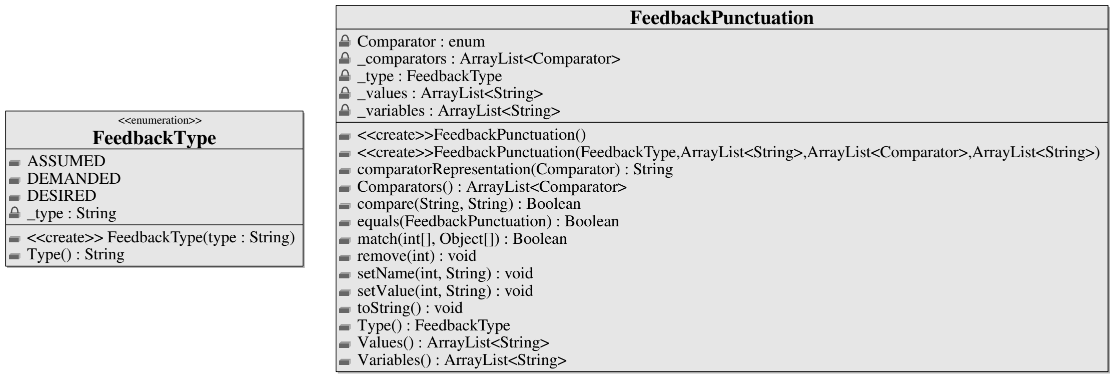


Figure 6.5: Feedback-augmented architecture. Operators are now able to send and receive Feedback Punctuations.



```
<<enumeration>>
FeedbackType
ASSUMED
DEMANDED
DESIRED
_type : String
<<create>> FeedbackType(type : String)
Type() : String
```

```
FeedbackPunctuation
Comparator : enum
_comparators : ArrayList<Comparator>
_type : FeedbackType
_values : ArrayList<String>
_variables : ArrayList<String>
<<create>>FeedbackPunctuation()
<<create>>FeedbackPunctuation(FeedbackType,ArrayList<String>,ArrayList<Comparator>,ArrayList<String>)
comparatorRepresentation(Comparator) : String
Comparators() : ArrayList<Comparator>
compare(String, String) : Boolean
equals(FeedbackPunctuation) : Boolean
match(int[], Object[]) : Boolean
remove(int) : void
setName(int, String) : void
setValue(int, String) : void
toString() : void
Type() : FeedbackType
Values() : ArrayList<String>
Variables() : ArrayList<String>
```

Figure 6.6: Feedback Punctuation UML.

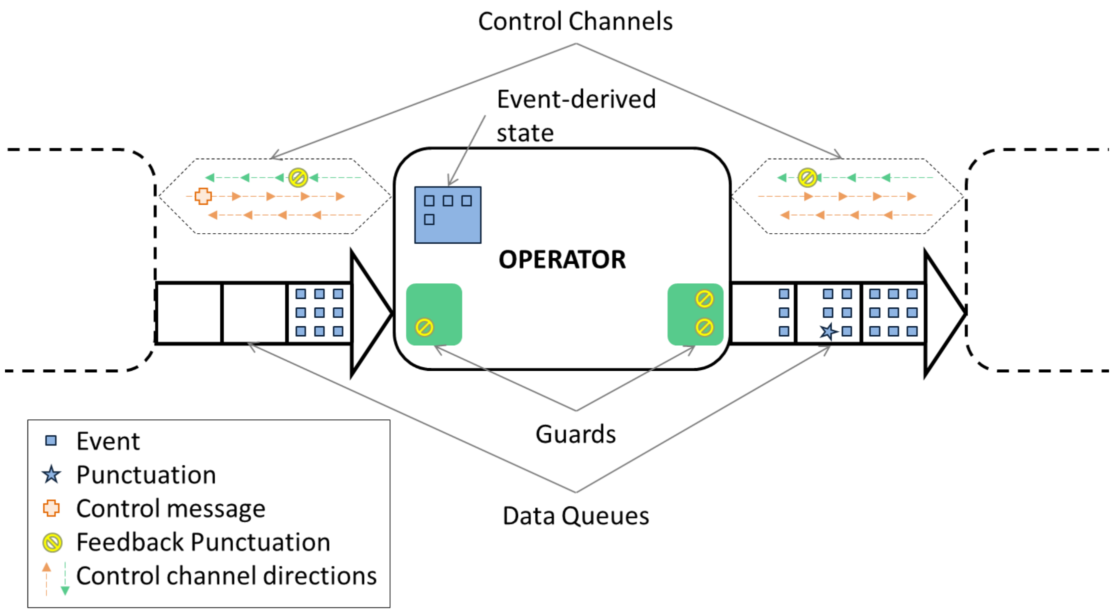


Figure 6.7: Feedback-compatible operators are capable of mounting guards (on input or output) to implement feedback processing strategies. Additionally, stateful operators can purge portions of their event-related state. Punctuation propagation is unchanged.

feedback can be generalized. For example, one common piece of exploitation is the mounting of guards. This revised *operator model* is illustrated in Figure 6.7.

Correct responses may entail mounting input or output guards to avoid outputting or processing events. In NiagaraST, events are objects of class `Tuple`. While in some operators (such as `SELECT`) the guard could be set up by modifying operator parameters, in others, such as `WINDOW-AVERAGE`, it must be mounted separately from the operational mechanics. To encapsulate guard mounting and processing, I created a “Guard” class, which can be instantiated by an operator that needs this capability. This class encapsulates guard-related activity, such as maintaining the current list of predicates and purging.

Feedback punctuations arrive as part of a control signal, which enabled maintaining the original execution sequence of an operator (see Algorithm 3 in Section 6.1), by localizing changes in the process function.

In the NiagaraST codebase we use inheritance to avoid duplication of commonly used codepaths. For example, the `WINDOWED-AVERAGE` physical operator derives from aggregate operators, which derive from grouping operators, which derive in turn from the common operator base class. For this work, you will notice how feedback responses are closely related to an operator's specialized class, which is where I implement them, but also some generic machinery (such as guard management), which is not as operator-specific as a response strategy, is implemented higher in the hierarchy. In Figure 6.8 I illustrate the architectural implications of guards and feedback processing localizations. The design choices try to minimize overall system design changes, but still provide accessibility and extensibility to the feedback machinery. The proposed design can be evaluated by observing the ease with which alternative feedback-handling mechanisms can be implemented – for example, the design allows multiple implementations of logical operators that have different physical responses, a characteristic amicable to supporting different contracts.

I have posited that there may be more than one correct response to assumed

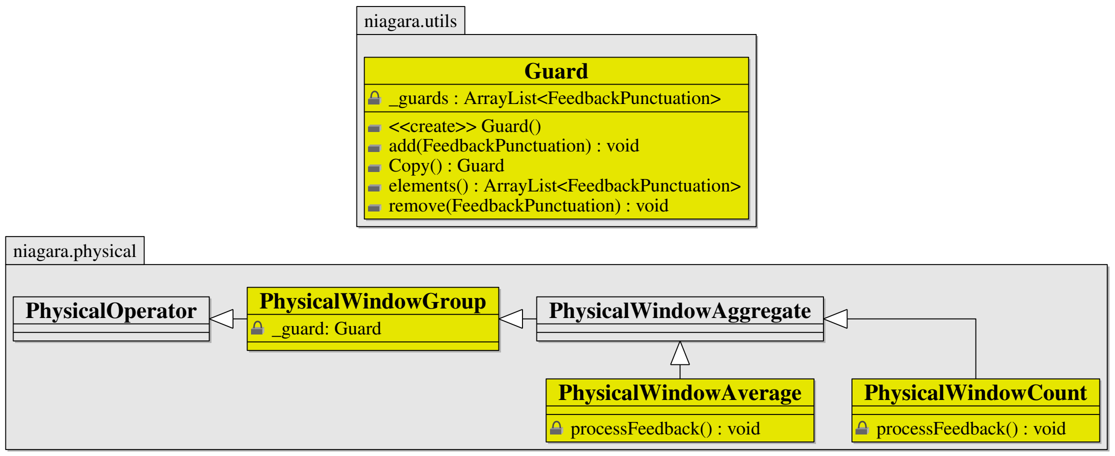


Figure 6.8: Architectural implications of adding guard support. A new “Guard” class is added to the niagara.utils package, and is used by various operators. Discovery occurs in specialized classes (such as “PhysicalWindowAverage”), while general guard management is localized higher in the hierarchy. Yellow-colored classes are where feedback-related changes have been placed.

```
<select id="se" input="speed" exploit="yes" propagate="no">
  <pred op="gt">
    <var value="$speed" /><number value ="10" />
[...]
```

Figure 6.9: Simple SELECT operator in XML-QL illustrating configurability of feedback processing.

punctuation. To facilitate experimentation, I gave operators the ability to be parameterized with respect to feedback, primarily in two ways: whether exploitation should occur or not, and whether propagation should occur or not. This parameters are adjustable in the query plan, as exemplified in Figure 6.2. Notice the two parameters, `exploit` and `propagate`. Keeping the two knobs separate is merely a code-reuse convenience. In practice, some operators (such as an aggregate) should not propagate without exploiting. In this example, `SELECT` would incorporate assumed punctuation to its selection condition, and would not propagate feedback.

In summary, the design cost of adding support for feedback to NiagaraST entailed: (1) adding feedback punctuation class and associated methods, (2) specializing how control signals are processed for cases where the signal is a feedback punctuation, (3) adding a guard class that can be instantiated as input or output guard by a feedback-enabled operator.

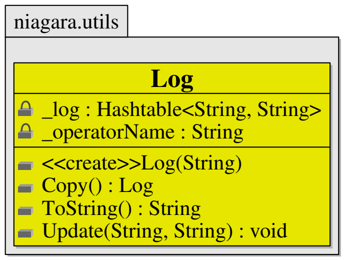


Figure 6.10: The `Log` class.

6.3 INSTRUMENTATION

To execute my evaluation strategy (see Chapter 7), I needed a reliable set of metrics that would allow me to demonstrate several aspects of the proposed techniques. Assessing memory usage accurately is tricky, since the Java VM's garbage collector is non-deterministic. I also needed a way to track when feedback was sent, and how effective the guards were at pruning off work. I designed a simple extension to the NiagaraST system to enable operator-lever logging.

First, the log structure is intended to be maintained in main memory, to minimize disk-access effects. Second, an operator's individual log is only reported to standard output during shutdown – this means operators do not constantly report their log contents, nor can the contents of the logs be pulled during execution. Third, the log is a very generic key-value-pair collection, meant to be extensible and customizable depending on the operator I was studying. Last, logging can be turned on or off on a per-operator basis. The class diagram for the log structure is shown in Figure 6.10.

To illustrate the use of logging in the system, consider the `SELECT` operator Suppose we are interested in tracking the number of tuples it reads in, the number of tuples it produces, and how many tuples it drops. The API to manipulate the log is fairly simple. The following code updates the value under the key "TupleDrop" with a new string (in this case, the representation of the integer variable `tupleDrop`):

```
log.Update("TupleDrop", String.valueOf(tupleDrop));
```

The operator “se” (an instance of select) in Figure 6.10 produces output similar to the one shown in Figure 6.11 after the query shuts down. Having this logging ability enabled me to collect relevant statistics for the performance evaluation.

In the next chapter, we will evaluate assumed feedback punctuation using NiagaraST with the modifications and extensions described so far.

```
<PhysicalSelect(se)>
  <record>
    <key>TuplesIn</key>
    <value>362</value>
  </record>
  <record>
    <key>TupleDrop</key>
    <value>65</value>
  </record>
  <record>
    <key>TupleOut</key>
    <value>297</value>
  </record>
</PhysicalSelect(se)>
```

Figure 6.11: Sample log output.

```
<select id="se" input="occupancy" log="yes">
  <pred op="gt">
    <var value="$speed"></var><number value ="10"></number>
</pred>
```

Figure 6.12: Simple SELECT query plan in XML-QL showing logging. By simply setting log="yes" the operator will print out an XML-formatted summary of its logged contents.

Chapter 7

EXPERIMENTAL EVALUATION

In this chapter I present an experimental evaluation of the feedback framework as implemented and tested in the NiagaraST DSMS. In particular, I have focused on testing both the efficacy of the approach, the effect of system design, and its cost-to-benefit ratio. I present and discuss this evaluation as a series of experiments designed to answer the following questions:

1. What is the overhead associated with checking for and communicating feedback punctuations apart from any actions taken in response to feedback? I think understanding this cost is of interest, especially since there will be times when feedback does not alter data processing, for example, when no events match feedback.

2. Does using assumed punctuations increase the utility of events in the output? (Utility as defined by the client; examples include timeliness and relevance.) One expected benefit of avoiding unnecessary work is to use those resources to address work that is relevant. In doing so, it is possible that more events in a query's output are useful. This scenario is inspired by the discussion

from Section 1.2.

3. Does increased aggressiveness in feedback propagation translate into increased performance? An aggressive propagation strategy involves propagating feedback as far upstream as possible. Since an operator can simply exploit feedback and not propagate it any further, it is interesting to examine scenarios in which seeking to propagate as far upstream as possible positively effects performance.

In addition, we wish to observe the effect of using and exploiting assumed feedback punctuation in queries involving various operators from NiagaraST.

In the following sections I discuss the data sources, experiments, and observed results. My experiments exercise NiagaraST with the following system parameters:

Java Virtual Machine Settings. The Java Virtual Machine is allotted 2048 Mb. of heap size. I chose this size since the host machine can fit that in main memory and avoid paging.

Parser Pages. The SAXDOM parser in NiagaraST can request up to 50000 memory pages. I found this setting to be sufficient to parse the various input files.

Inter-operator Buffers. Operators in NiagaraST are connected by paged buffers. The NiagaraST team's experience with the system shows that 30 tuples per page and 5 pages per buffer are reasonable parameters for the system in terms of performance. Less buffering results in better latency between operators, but is offset by process-switching overhead.

7.1 DATASETS

I created synthetic datasets inspired by the data archived in the Portland Oregon Regional Transportation Archive Listing (PORTAL), which is a data archive of traffic-sensor data from the Portland metropolitan area freeway system.[1] The operating scenario of this environment influenced my choice of scenarios and queries: Each traffic sensor in the freeway reports the number of vehicles, average speed, and percentage of time there was a vehicle present on top of the sensor at 20-second intervals. There is one sensor per lane at a sensing station, and sensing stations are spaced about 1 miles apart.

Probe vehicles are another common data source in traffic-analysis scenarios. Probes can both provide insight into areas where sensor coverage is sparse and validate sensor estimates. They also give higher-granularity profiles. I also created data that simulates probe vehicles traveling through the freeway system. Probes are common data sources in the transportation domain, where, for example, bus data is used to get a sense of an arterial road's performance.

This data is amenable for writing a series of representative streaming queries. While there is still no consensus as to what the typical streaming query looks like,

[1]The archive, as well as various visualization applications, can be accessed at `http://portal.its.pdx.edu/`

many queries over punctuated data streams have the following characteristics:

- They are expressed in windowed semantics.
- They leverage temporal operations, such as window-join.
- They involve a stored relation (persistent data).

7.1.1 Schemata and Examples

In general, I will consider two streams: the `sensor` stream and the `probe` stream, plus the stored relation `location`. The schemata for these streams and relation are given in Tables 7.1 - 7.3. The data received by the PORTAL project from the Oregon DOT's sensors is rounded to integers. The three quantities reported by these sensors – volume, speed, and occupancy – are used to analyze traffic according to the fundamental relationship of traffic flow theory: Flow is the product of density and speed. Occupancy in this scenario serves as a surrogate for density. For an overview and discussion of this fundamental relation, I encourage the reader to review Chapter 7 in May [45]. Example tuples and punctuation from this schemata are shown in Figures 7.1 – 7.3, in XML format for consumption by the NiagaraST system.

7.1.2 Properties

I conducted the experiments described in this section over three datasets, each corresponding to one of the aforementioned streams (probe, sensor, and location).

Attribute	**Type**	**Description**
sensor_id	integer	Sensor identifier.
timestamp	timestamp	Time of reading in ticks. 1 tick is one ten-millionth of a second. Semantically, a tuple summarizes the previous 20 seconds.
timestamp_text	string	Date and time of reading in English.
volume	integer	Number of vehicles detected by the sensor.
speed	integer	Average speed (in miles per hour) of vehicles detected by the sensor.
occupancy	integer	Percentage of time a vehicle was on top of the sensor.

Table 7.1: Schema for the `sensor` stream

Attribute	**Type**	**Description**
probe_id	integer	Probe vehicle identifier.
timestamp	timestamp	Time of reading in ticks. Semantically, a tuple summarizes the previous 5 minutes.
timestamp_text	string	Date and time of reading in English.
freeway_id	integer	Freeway identifier.
milepost	integer	Freeway milepost.
speed	integer	Average speed (in miles per hour) of probe vehicle.

Table 7.2: Schema for the `probe` stream

Attribute	Type	Description
sensor_id	integer	Sensor identifier.
freeway_id	integer	Freeway identifier
milepost	integer	Freeway milepost where this sensor is placed.

Table 7.3: Schema for the `locations` relation

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<niagara:stream xmlns:niagara="http://datalab.cs.pdx.edu/niagaraST"
  xmlns:punct="http://datalab.cs.pdx.edu/niagaraST/punct">
  <sensor>
    <timestamp>634018212000000000</timestamp>
    <timestamp_text>
      Monday, February 15, 2010 9:00:00 AM
    </timestamp_text>
    <sensor_id>1001</sensor_id>
    <volume>20</volume>
    <speed>50</speed>
    <occupancy>50</occupancy>
  </sensor>
</niagara:stream>
```

Figure 7.1: A tuple from the `sensors` stream.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<niagara:stream xmlns:niagara="http://datalab.cs.pdx.edu/niagaraST/"
  xmlns:punct="http://datalab.cs.pdx.edu/niagaraST/punct">
  <punct:probe>
    <timestamp>634018212000000000</timestamp>
    <timestamp_text>*</timestamp_text>
    <probe_id>1</probe_id>
    <freeway_id>*</freeway_id>
    <milepost>*</milepost>
    <speed>*</speed>
  </punct:probe>
</niagara:stream>
```

Figure 7.2: A punctuation from the `probes` stream.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<niagara:stream xmlns:niagara="http://datalab.cs.pdx.edu/niagaraST"
  xmlns:punct="http://datalab.cs.pdx.edu/niagaraST/punct">
  <location>
    <freeway_id>1</freeway_id>
    <sensor_id>2002</sensor_id>
    <milepost>102</milepost>
  </location>
</niagara:stream>
```

Figure 7.3: A tuple from the `location` relation.

The location file contains 150 tuples, representing 5 freeways. Each freeway has 10 sensor stations, and each station has 3 sensors. This dataset's size is 15 KB.

The probe dataset contains 14,400 tuples and 14,400 punctuations, with a tuple reporting a probe's data appearing every five minutes. The dataset covers a 24 hour period. This dataset's size is ˜5 MB.

The sensor dataset contains 648,000 tuples and 1,440 punctuations. It covers a 24 hour simulation period, with punctuation appearing every minute, and a tuple representing a sensor's data every 20 seconds. This dataset's size is ˜140 MB.

7.2 EXPERIMENTAL SETUP

I conducted all experiments on a desktop computer equipped with an Intel Core 2 Duo CPU (E8600 @ 3.33 GHz) and 4 GB of RAM. The operating system, Windows Server 2008 R2 Standard, is 64-bit capable.

NiagaraST is written in Java. I exercised bytecode compiled using Oracle's Java SE version 6, Update 20. The runtime provided with this version is 64-bit.

All data was consumed from files in a local disk. The `FILESCAN` operator in NiagaraST consumes XML data files and internalizes tuples into the system. Having static datafiles consumed at the maximum rate minimizes network effects and contributes to easier set ups for experiment reproducibility. Note part of the processing cost in NiagaraST comes from parsing XML, therefore, there is a fixed cost to internalizing every tuple. Nicola *et al.* reported on the negative effects of parsing XML and overall system degradation performance [50]. In the case of my experiments, feedback does not propagate through the query tree to prevent this parsing, in order to concentrate on the effect of avoiding tuple-processing work.

When evaluating result correctness, query output is output to disk via capture of standard console output. This activity dominates in-memory processing, therefore, unless otherwise noticed, performance metrics are reported on queries that emit results to main memory but not to disk. When timing queries, the clock starts once the query is issued to the server, and is stopped once the `EOS` (end of stream) signal is processed by the last operator.

I claim that CPU usage and execution time are intimately related. If we run a program at the maximum rate `FILESCAN` can deliver, we expect to be CPU-bound. Hence, if a program executes in 10 seconds, and an optimized version of the program executes in 8 seconds, were the input stream actually arrive over 40 seconds, we would expect to see 35% CPU usage in the first case and 20% usage in the second case. I propose to measure the total execution time of a query as a representative of the CPU usage it incurred in.

I claim that memory allocation in the context of query execution is directly related to the number of tuples that are processed by the query, hence, a query that avoids allocating a tuple utilizes less memory. Counting tuples processed per operator does not account for state, but we know there is a relation between tuples input to an operator, tuples output, and state sizes. For this purpose, I instrumented operators to keep internal counters for these quantities, as a surrogate for memory utilization.

Collecting both CPU-usage estimates and memory-usage estimates allows us to test our resource utilization hypothesis in the following experiments.

7.3 ASSESSMENT OF FEEDBACK OVERHEAD (QUESTION 1)

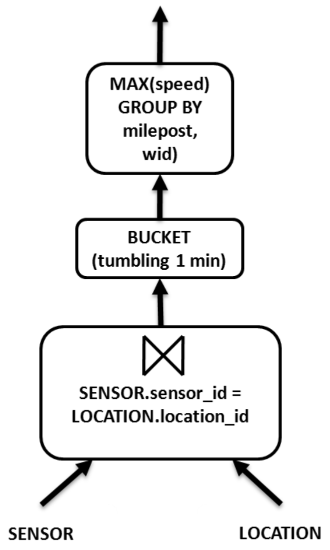
I introduced two changes into NiagaraST to incorporate feedback support (as described in Chapter 6). First, I changed the main execution loop of the base class from which operators derive. This change causes an operator to check the control queue for feedback messages on each iteration of the loop. Second, when a feedback message is sent upstream, the system performs work to encode and enqueue the message.

We expect to see some overhead in the system due to the steps added to an operator's lifetime. We also expect more work to be performed when sending feedback. The intuition, however, is that these two changes add negligible overhead, that is, perceivable differences will be at the level of run-to-run variance. Sources of run-to-run variation in NiagaraST include garbage collection and scheduling choices made by the Java VM.

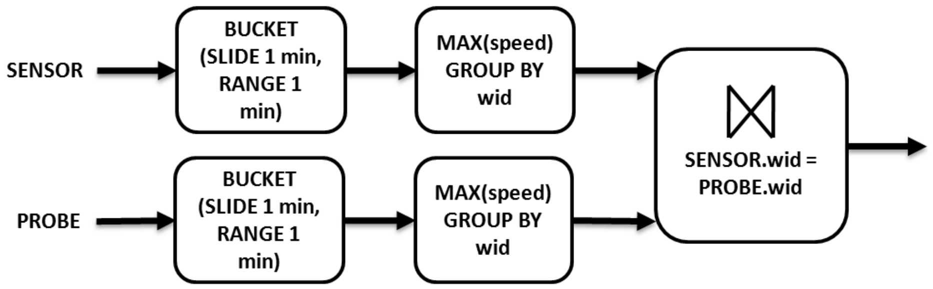
To test this hypothesis, I use two sets of experiments: the first set to test the behavior of the system before and after the implementation of feedback support, and the second set to test the overhead introduced by actually producing feedback.

7.3.1 Experimental Suite #1

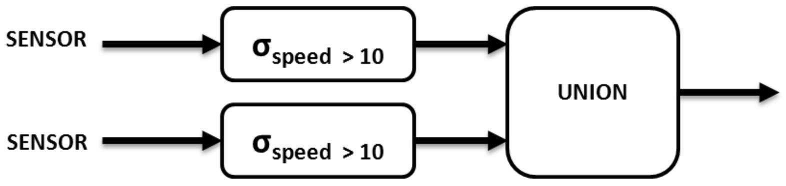
Consider the query plans shown in Figure 7.4. These query plans exercise a representative set of operators from the NiagaraST algebra: `BUCKET`, `JOIN`, `MAX`, `SELECT`, and `UNION`. Each query plan was executed in two different version of NiagaraST: NiagaraST-Prime, compiled against sources before I introduced any support for feedback, and NiagaraST, compiled after the introduced changes.


MAX(speed) GROUP BY milepost, wid)
BUCKET (tumbling 1 min)
⋈
SENSOR.sensor_id = LOCATION.location_id
SENSOR
LOCATION

(a)



(b)



(c)

Figure 7.4: Experimental Suite #1: Three query plans used to test the overhead of feedback-awareness changes to the operator base class' execution loop in NiagaraST.

	Prime μ	**Prime σ**	**NiagaraST μ**	**NiagaraST σ**	**Diff.**	**% increase**
Query (a)	9.26	1.04	9.43	0.86	0.17	1.8%
Query (b)	6.18	0.43	6.43	0.63	0.25	4%
Query (c)	11.76	0.54	11.92	0.42	0.16	1.4%

Table 7.4: Experimental Suite #1 results. *Prime* is the system before feedback was introduced. NiagaraST is the system with the feedback machinery in place. Table contains the mean execution time (μ) in seconds, and the standard deviation (σ), across 20 runs. Also included is the absolute and percentage increase in mean execution time.

Results

The execution times and variances reported confirm the expected result: overhead on the order of run-to-run variance. For example, look at the mean execution times for Query (a). The difference is 0.17 seconds (a 1.8% increase). Notice in all cases the run-time difference is within run-to-run variance.

We note that while modifying the system, we addressed some minor implementation defects. One defect caused Prime to stop processing input data when the End-Of-Stream (EOS) signal arrives at an operator, regardless of whether the operator's input queues were full. This defect caused early termination of processing, potentially neglecting small percentages of work remaining. The semantics of EOS in NiagaraST are modified to account for all work, that is, the EOS signal is not propagated until the input buffers are empty. The minutia of this defect

```
<instrument id="instrument" input="select1"
            interval="1"
            log="no"
            propagate="yes"
            fattrs="timestamp sensor_id"/>
```

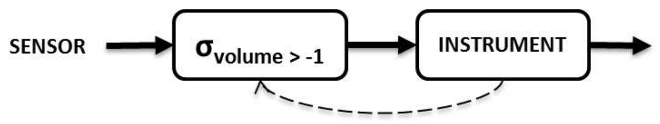
Figure 7.5: An instance of the instrument operator.

is not interesting, but the effects contribute to the small increase in work rof the feedback-enabled version, which is why I disclose this detail.

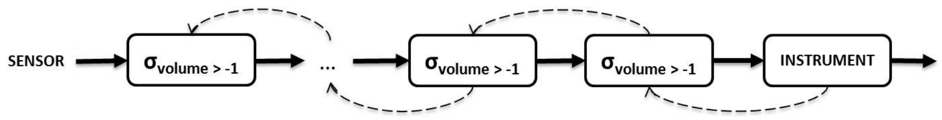
7.3.2 Experimental Suite #2

In the rest of the thesis, I will be using an operator to send feedback into a query tree. The `INSTRUMENT` operator has simple semantics: It sends a feedback punctuation after receiving n tuples, and then starts counting again. Operationally, I control which attributes it punctuates on, which values are used, and other activities such as logging memory usage. In Figure 7.5 I show how to express it in XML-QL. Notice that just like any other operator in the tree, it has a unique identity. The feedback periodicity is controled by the `interval` attribute, and the attributes to send feedback on are specified in the `fattrs` attribute.

To evaluate the effect of feedback, consider the query in 7.6 (a). In this query, the operator `INSTRUMENT`, parameterized as shown in Figure 7.5, issues a feedback



(a)



(b)

Figure 7.6: Experimental Suite #2: Testing overhead of feedback communication in NiagaraST. (a) A simple query plan with the `INSTRUMENT` operator. (b) An extended query plan with more inner operators.

punctuation after having consumed 1 event. I use the value 1 to maximize feedback volume for this experiment. The `SELECT` operator is feedback-aware, but does not exploit it. A way to compare whether messages flowing have an effect larger than run-to-run variance in execution time, we can execute two instances of this query plan: one in which `INSTRUMENT` sends feedback punctuation and one where it does not. Moreover, we can run a series of queries with more than one `SELECT` operator in the chain. For each configuration (as shown in Figure 7.6 (b)), one run will have both `INSTRUMENT` and the inner `SELECT` operators propagating the feedback punctuation but not exploiting the feedback.

I ran experiments with query plans using 1, 5, 10, and 15 distinct `SELECT` operators. For each plan, I executed two versions: one with feedback propagation

	Off μ	On μ	Off σ	On σ	Diff.
1	5.77	5.76	0.16	0.11	-0.01
5	6.29	6.30	0.10	0.12	0.01
10	7.18	7.21	0.10	0.10	0.03
15	7.94	7.97	0.11	0.16	0.03

Table 7.5: Experimental Suite #2 results. Each row represents how many `SELECT` operators are instantiated in the executed query plan. Off and On indicate whether propagation was turned off or on, respectively. The Diff. column reports the difference in mean execution times. Execution time is in seconds.

turned on and one with feedback propagation turned off. An example query plan with two `SELECT` operators and propagation turned on is shown in Figure 7.7. In Table 7.5 I report mean execution time and variance over 20 runs over the large sensors dataset for each configuration. The experimental results confirm our hypothesis: the overhead in execution time is within run-to-run variance. Notice in this set of experiments, as opposed to Experimental Suite #1, we ran on the same codebase, therefore there are no effects of defect fixes.

7.4 EFFECT OF ASSUMED FEEDBACK IN OUTPUT UTILITY (QUESTION 2)

In Section 1.2, I gave a brief overview of some of the early work that motivated this thesis. The original work was motivated by the speedmap example I have

```
<?xml version="1.0"?>

<!DOCTYPE plan SYSTEM "queryplan.dtd">

<plan top="cons">

<!-- unnests ending in node occupancy -->

<select id="select1" input="occupancy" propagate="no">

 <pred op="gt"><var value="$volume"/><string value="-1"/></pred>

</select>

<select id="select2" input="select1" propagate="yes">

 <pred op="gt"><var value="$volume"/><string value="-1"/></pred>

</select>

<instrument id="instrument" input="select2" interval="1" log="no"

            propagate="yes" fattrs="timestamp sensor_id"/>

<construct id="cons" input="instrument">

<!-- rest of construct with full sensor schema -->

</plan>
```

Figure 7.7: Sample query plan instance from Experimental Suite #2.

used to motivate the discussion in this thesis, which we documented in Fernández-Moctezuma *et al.* [24]. In this section I revisit the problem space and reevaluate an equivalent workflow with the feedback machinery as implemented in NiagaraST.

I previously discussed a notion of the utility of an event in the output of a continuous query in the context of the speedmap example from Chapter 1. One of the observations we made for that workflow was that expensive processing over some events lead to output in which some events were “too late to be of use for updating a speedmap. Let us adopt the following definition of utility for this example:

Utility. An event e in the output of a query is useful if its timestamp ts is within an application-specific period λ of the timestamp high watermark observed in the output at the time e is observed.

In the motivating example, relative disorder was induced by a very expensive disk-accessing operator in one of two branches. In order to avoid tuples that are not useful, I introduced a binary operator called `PACE`, which enforces a simple notion of utility: either an event is useful or it is not. `PACE` is a type of conditional union, in which an event `e` is output only if its timestamp is within λ of the highest-watermark timestamp seen by the operator. The operator is parameterized by stating its inputs, the name of the progressing attribute, and the maximum disorder threshold λ. As the inputs to `PACE` progress, the operator checks and updates the high watermark accordingly. Note that the semantics of `PACE` are not sensitive to which input is lagging behind.

By having PACE simply discard late work, we only avoid having events with no utility in the output. My hypothesis is that by propagating feedback that communicates the current timestamp of tuples being dropped by PACE, an antecedent operator can exploit this information to avoid useless work and catch up with relevant events. The following experiments seek to evaluate this hypothesis in practice.

7.4.1 Experimental Suite #3

Consider the three query plans shown in Figure 7.8. In these plans, we route an event to the expensive operator if its speed value is -1, and route it directly downstream otherwise. The EXPENSIVE operator is meant to represent the behavior of an imputation process for missing values that might, for example, need to consult archived data, or perform a statistical computation, as we reported previously [24].

I executed these three query plans with a modified version of the sensors dataset described in Table 7.1.1. While the schemata is the same, the file has only 324,000 tuples, simulates 12 hours of execution, has 5 freeways, 3 sensors per milepost, and 10 mileposts per freeway. In addition, every other event in it has a speed value of -1.

The EXPENSIVE operator is parameterized by an integer, which simply sets the upper limit of an internal for-loop. When an event enters the operator, it is placed

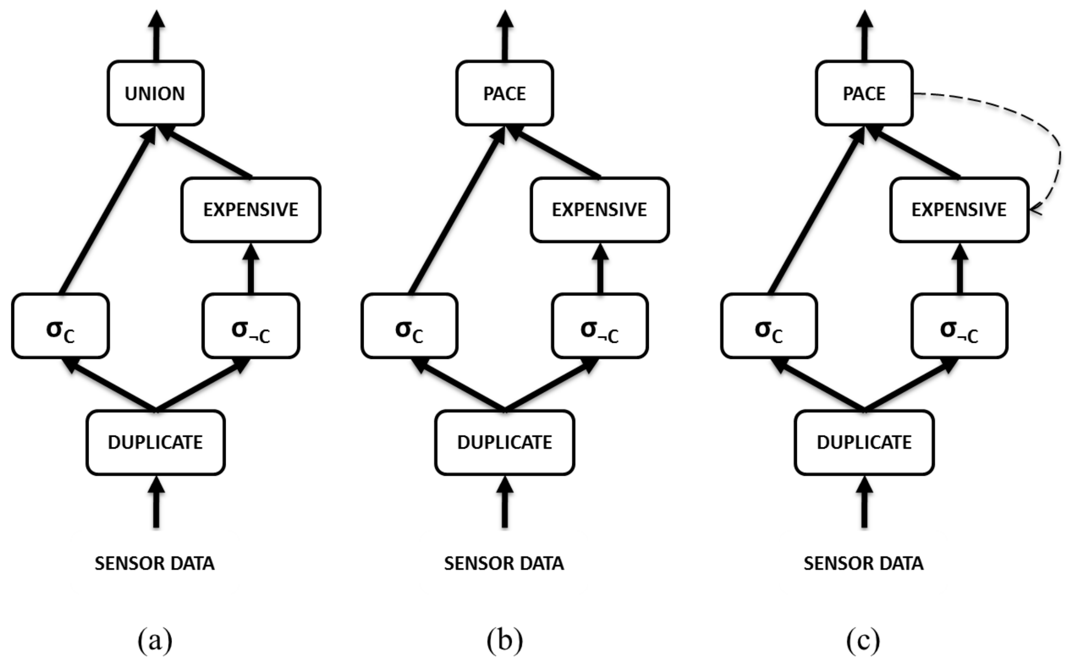


Figure 7.8: Experimental Suite #3. Plan (a) uses UNION to produce the final output. Plan (b) uses `PACE` to enforce the utility restriction. Plan (c) has `PACE` sending feedback to `EXPENSIVE`, with the latter exploiting it to resume work on potentially more useful events.

in an array of pending work in arrival order. The operator performs the expensive loop for each element in the array. Once the expensive work finishes, the event is output, and the operation repeats. When a feedback punctuation is received, the list of pending work is pruned of events covered by the assumed punctuation. `EXPENSIVE` does not mount guards, and does not propagate this feedback farther upstream.

7.4.2 Results discussion

Plan (a) establishes the baseline for comparison. The `EXPENSIVE` operator receives and outputs exactly 162,000 events, in addition to 720 punctuations. The `UNION` operator receives 324,000 events over both of its inputs, 1440 punctuations over both of its inputs, and emits 162,000 events and 720 punctuations. It should be noted that in these runs I wrote the output of the queries to disk in order to measure utility.

An execution of Plan (b) illustrates the effect of having `PACE` set up to uphold a maximum threshold of 5 minutes of tolerance for late events. Here, `PACE` dropped 160,171 events, emitting only 163,829 events, essentially losing almost all of the events that came from the `EXPENSIVE` branch.

For single runs, I calculated the utility on the output of Plan (a) by tagging events whose timestamp was beyond five minutes of a rolling-high watermark as not useful. In Plan (a), only 50.5% of the tuples meet the utility criterion. In Plan (b), 100% of them met the utility criterion.

Executing Plan (c) shows an encouraging increase in the number of useful tuples output. `EXPENSIVE` only outputs 30,931 tuples, and PACE only drops 17,712 of those, for a total number of tuples output of 175,219. In this extreme case with 50% of the events requiring an expensive intervention, the query is able to output an additional 13,219 useful events, in which all of them meet the utility criterion.

These experiments have focused solely on the utility of tuples in the output, and validate the hypothesis that by propagating feedback upstream and acting on it, a query's output utility improves. Utility itself is a query-dependent notion. In these experiments, I had a scenario-driven definition of utility and an operator to enforce the utility function. For other queries, the utility function can be much simpler. For example, an event in the output is not useful if it matches feedback sent to the query directly. I do not think such a generic utility function is interesting to explore *in extenso*. In the following sections, I will introduce queries with feedback-enabled versions of the Niagara operators, and focus not on a semantic notion of utility, but on measuring processing-time and memory reductions achieved by using feedback.

7.5 EFFECT OF PROPAGATION AGGRESSIVENESS IN QUERY OUTPUT UTILITY (QUESTION 3)

While evaluating Experimental Suite #3, one can observe that propagating feedback farther upstream proved beneficial when the goal was to make the query produce more useful output. I have stated another hypothesis, in which I expect work avoided to have a direct effect in resource utilization, and I expect the effect to increase as feedback is acted on and propagated farther upstream a query plan. To test this hypothesis, I propose the following experimental suite, in which I gradually change how far upstream feedback is propagated. I call this change aggressiveness, in which propagation is most aggressive when it is propagated as far upstream as possible.

7.5.1 Experimental Suite #4

Consider the query plan shown in Figure 7.9 (a). In it, the output of a temporal join of two streams with the same schema and data (sensor data) is window-averaged on the speed attribute. The instrument operator sends an assumed punctuation as soon as it is scheduled indicating tuples with a `WID` less than 100 are not desired in the output. If feedback is both acted on and propagated upstream, I expect to see the number of tuples worked on reduced as the level of feedback increases, from (b), in which the average operator exploits locally, to (d), in which the feedback reaches the join and is exploited there.

To test this hypothesis, I used the logging infrastructure I added to NiagaraST to simply account for the number of tuples that enter and leave an operator. I am using event count as a surrogate for memory utilization, in particular when a stateful operator adds an event to its state or when inter-operator output is

Operator	Events output (a)	Events output (b)	Events output (c)	Events output (d)
Join	648,000	648,000	648,000	603,019
Bucket	648,000	648,000	603,010	603,012
Average	1,440	1,340	1,340	1,340

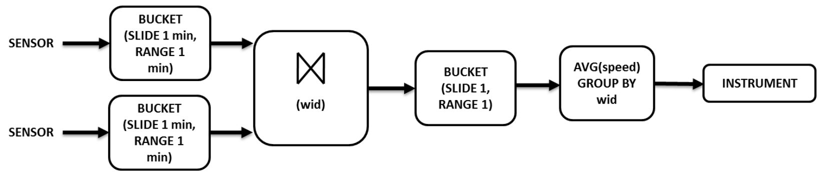
Table 7.6: Experimental Suite #4 results. Events output for plans (a) - (d) for operators considered in the backpropagation. Counts for single representative runs.

materialized.

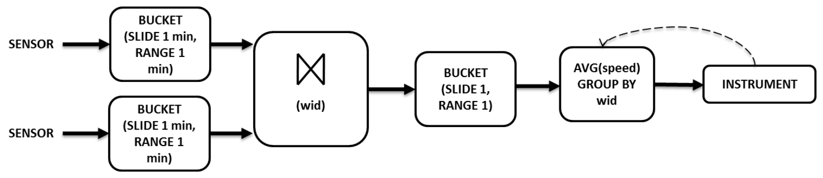
7.5.2 Results discussion

In Table 7.6, I summarize the change in event out count for the `JOIN` operator and the subsequent `BUCKET` and `AVERAGE` operators.

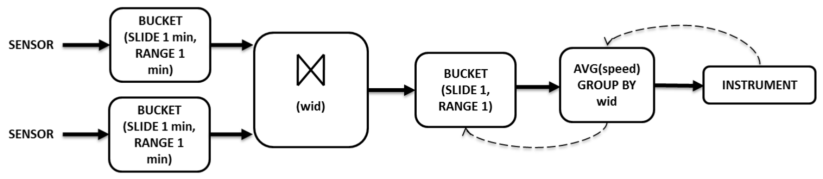
The effect of exploiting feedback in total events output by join and bucket on the query is not changed from (a) to (b), although there is a small change in the events output by average. For the run profiled here, we see a 44,900 event reduction in the output of bucket when the feedback is propagated in (c). The output of the query at this point does not change, but the cost of delivering this output in terms of resource utilization, *i.e.* the number of tuples materialized for output by bucket, is perceivable. A fourth run propagating feedback all the way down to the join shows savings of 44,981 events in the join, which means bucket processes fewer tuples on its input. Since join is also guarding its output, we see a reduction of 44,988 events in its output when compared to the baseline (a). There


SENSOR
SENSOR
BUCKET (SLIDE 1 min, RANGE 1 min)
BUCKET (SLIDE 1 min, RANGE 1 min)
⋈ (wid)
BUCKET (SLIDE 1, RANGE 1)
AVG(speed) GROUP BY wid
INSTRUMENT

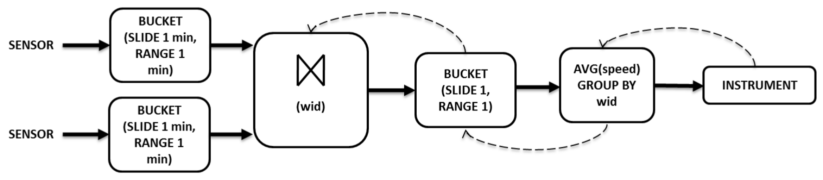
(a)


SENSOR
SENSOR
BUCKET (SLIDE 1 min, RANGE 1 min)
BUCKET (SLIDE 1 min, RANGE 1 min)
⋈ (wid)
BUCKET (SLIDE 1, RANGE 1)
AVG(speed) GROUP BY wid
INSTRUMENT

(b)


SENSOR
SENSOR
BUCKET (SLIDE 1 min, RANGE 1 min)
BUCKET (SLIDE 1 min, RANGE 1 min)
⋈ (wid)
BUCKET (SLIDE 1, RANGE 1)
AVG(speed) GROUP BY wid
INSTRUMENT

(c)


SENSOR
SENSOR
BUCKET (SLIDE 1 min, RANGE 1 min)
BUCKET (SLIDE 1 min, RANGE 1 min)
⋈ (wid)
BUCKET (SLIDE 1, RANGE 1)
AVG(speed) GROUP BY wid
INSTRUMENT

(d)

Figure 7.9: Self-join and average with varying levels of feedback propagation. The query in (a) without feedback, progressing until (d) with the farthest upstream operators receiving and exploiting feedback.

is run-to-run variance on the number of tuples saved, due to scheduling changes and when the feedback is received.

In these experiments, we have confirmed our hypothesis that propagating feedback farther upstream leads to observable decreases in resource utilization. In the next Section, I will take a look at the effect of aggressive propagation of feedback in a few queries.

7.6 EFFECT OF ASSUMED FEEDBACK IN RESOURCE UTILIZATION

At this point, I have shown that the overhead of sending feedback messages seems to be within run-to-run noise, and that feedback can have perceivable advantages in improving a query's output utility and also avoiding work the farther upstream it is propagated and acted upon. Motivated by these observations, I want to show how feedback affects resource utilization in queries when it both describes a large number of tuples in flight and is enacted as early (and as far upstream) as possible.

7.6.1 Experimental Suite #5

I used the `INSTRUMENT` operator to send feedback that covers most tuples in the stream. In each query, I propagate and exploit the feedback as far upstream as possible. The goal is to observe dramatic reductions both in the number of tuples sent by each operator and the overall execution time, as we hypothesize large

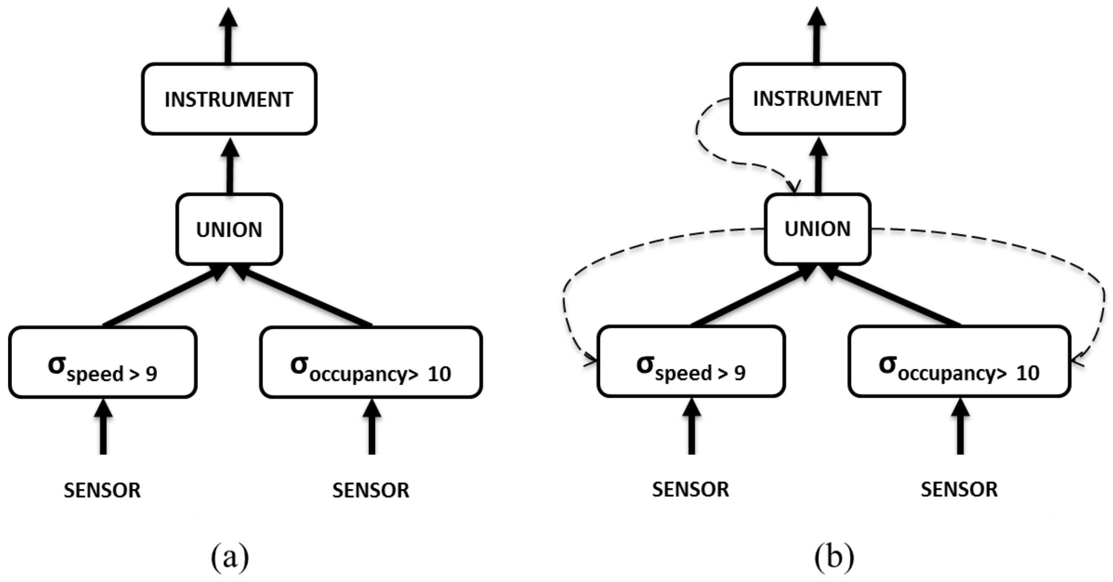
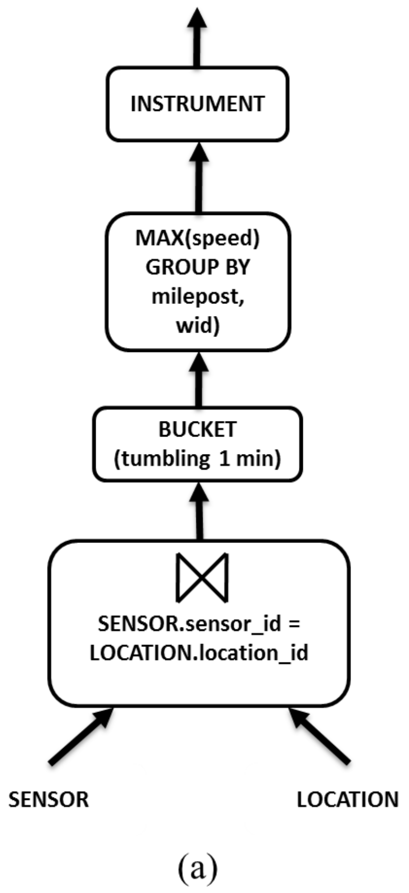


Figure 7.10: Two filters and a union. The query in (a) without feedback, the query in (b) with feedback.

numbers of tuples avoided equate large savings in resource consumption. I test for each query with and without feedback to compare.

I tried three queries that exercise operators in NiagaraST that I have modified to support feedback processing and exploitation. Queries are shown in Figures 7.10, 7.11, and 7.12. The first query has two streams with identical schema and data being united after a selection predicate is applied to each. The second query is a window max over a sensor and location join. The third query joins windowed averages from the sensor and probe streams.


INSTRUMENT
MAX(speed) GROUP BY milepost, wid)
BUCKET (tumbling 1 min)
⋈
SENSOR.sensor_id = LOCATION.location_id
SENSOR
LOCATION
(a)

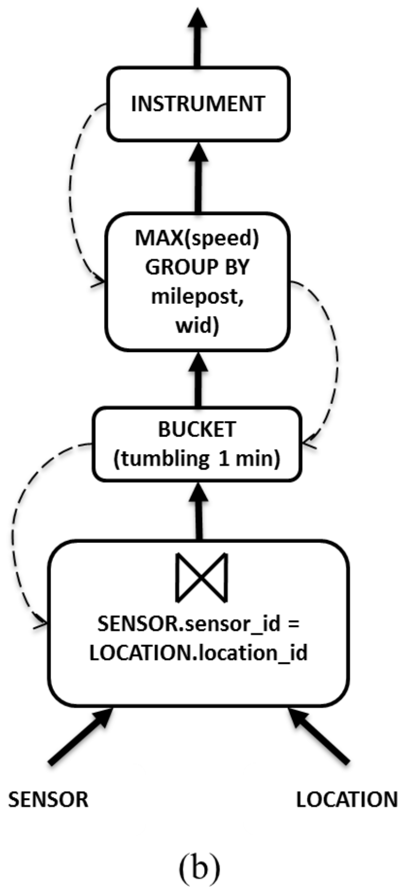
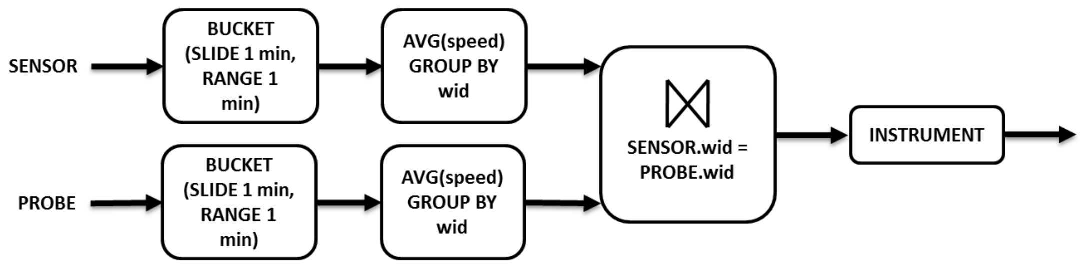
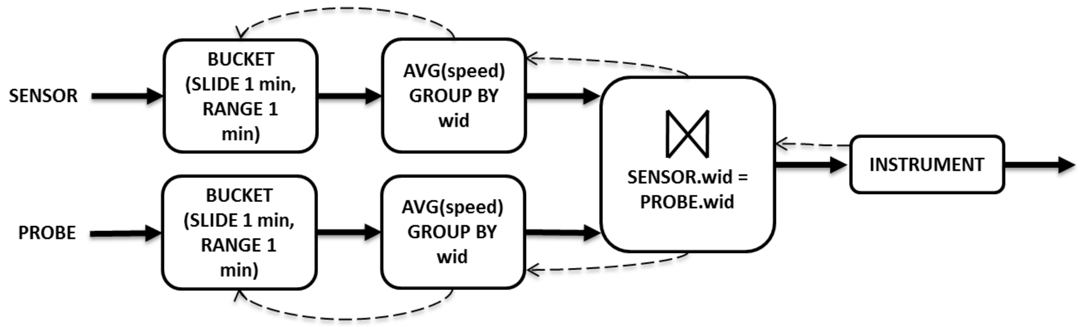

INSTRUMENT
MAX(speed) GROUP BY milepost, wid)
BUCKET (tumbling 1 min)
⋈
SENSOR.sensor_id = LOCATION.location_id
SENSOR
LOCATION
(b)

Figure 7.11: Windowed max. The query in (a) without feedback, the query in (b) with feedback.



(a)



(b)

Figure 7.12: Join of windowed-averages. The query in (a) without feedback, the query in (b) with feedback.

Operator	Events output (a)	Events output (b)
Select (occupancy)	514,421	287
Select (speed)	505,481	0
Union	1,019,902	150

Table 7.7: Experimental Suite #5, Query 1 results. Baseline counts in column (a), feedback-enabled counts in column(b).

7.6.2 Results discussion

Over 20 runs, Query 1 executed on average in 12.22 seconds, with a standard deviation of 0.18, as a baseline. When maximally exploiting the very restrictive feedback, average execution time went down to 9.66 seconds and a standard deviation of 0.16. Table 7.7 shows the extensive reduction in inter-operator event flow and final output. Recall that in NiagaraST we pay a high cost of parsing XML input, which is probably where most of the time is being spent in these runs. Still, on this simple workload, we can see a clear effect of exploiting feedback.

I observed similar results in Query 2, which had an average execution time of 7.34 seconds and a standard deviation of 0.23, going down to an average execution time of 5.54 seconds and a standard deviation of 0.25. These dramatic reductions accompany the observations of a profile run in which I count operator output, shown in Table 7.8.

Operator	Baseline count (a)	Feedback-enabled count (b)
Join	648,000	464
Bucket	648,000	457
Average	72,000	7

Table 7.8: Experimental Suite #5, Query 2 results. Baseline count of events output in column (a), feedback-enabled count of events output in column (b).

Operator	Baseline count (a)	Feedback-enabled count (b)
Bucket (sensor)	648,000	1,800
Average (sensor)	1,440	3
Bucket (probe)	14,400	20
Average (probe)	1,445	2
Join	1,440	2

Table 7.9: Experimental Suite #5, Query 3 results. Baseline count of events output in column (a), feedback-enabled count of events output in column (b).

Last, Query 3 shows results consistent with the other two, where average execution time drops from 7.19 seconds to 6.31 seconds (with standard deviations of 0.13 and 0.22, respectively) over 20 runs. Dramatic savings in state reduction as well, as operators produce very few events in the output due to receiving very restrictive feedback very early in query processing, as shown in Table 7.9.

In this Chapter I have both observed the feasibility of implementing feedback support as designed and discussed in Chapter 6. The set of NiagaraST operators

I modified are representative of the NiagaraST algebra. One of the main concerns I had in the design was not to negatively effect the system so that feedback messaging and handling would eventually dominate any substantial gains. The second intention was to validate the intuition that shedding work in the system, even when only accounting for processing and not transmission outside of the query, leads to observable resource conservation.

The use of feedback is likely to deliver the biggest gains when it is acted upon the earliest and there is a maximum range of tuples effected. I discuss these observations and offer other final remarks in Chapter 8.

Chapter 8

CONCLUSIONS AND OPPORTUNITIES FOR FUTURE WORK

In this thesis, I have introduced two tools that address several challenges in Stream Processing: The Feedback Framework, which consists in synthesizing downstream context as a data description and intent for upstream operators to exploit, and The Contracts Framework, a methodology to characterize queries and streams to perform static analysis of its probable runtime characteristics. In this thesis, I have formalized both frameworks, discussed and evaluated an implementation of the Feedback Framework in NiagaraST, and provided a reference implementation of the algorithms detailed in the consistent-accordance phase of the Contracts Framework.[1] In this chapter, I will look at what I learned from this work and speculate on possible future directions and applications.

A key insight for me was to interpret traditional punctuation (as defined by Tucker [62]) as a mechanism to communicate upstream context. In a sense, an operator receiving a punctuation as input knows that antecedent operators (if any) have finished processing the stream up to a certain point. A punctuation has an implicit action suggested that goes along the context it is communicating: It is

[1]See `http://web.cecs.pdx.edu/~rfernand/dissertation.html`

safe for the receiving operator to clear up internal state covered by the punctuation and stimulate output production. This interpretation holds for punctuation in general in NiagaraST, and for CTIs (a temporal-only, monotonically increasing punctuation) in CEDR [27] and StreamInsight [46].

The way NiagaraST has exploited this contextual information is to act immediately upon punctuation arrival. I argue that the time of action on the punctuation need not be immediate: For example, consider the following input to the `SUM` operator, which is grouping by `wid` over the schema `s(gid,value)`:

```
<1,2>
<1,3>
[≤1,*]
<2,1>
<2,2>
[≤2,*]
```

Operationally, a physical implementation of the `SUM` operator could act on each input element upon arrival. The first tuple in the above sequence causes a new partial aggregate group to be created and updates its current sum (2). The second tuple updates that partial aggregate. The third tuple causes a sweep of the internal state, identifies groups covered, outputs and cleanses. Processing continues in this manner. This description is reasonable if we are willing to perform more than one sweep of the internal state per scheduled time of the operator. Another reasonable implementation would minimize this action by remembering the last seen punctuation and acting on it when the scheduler signals it is time to surrender control. In this example, assuming the `SUM` operator is scheduled long enough to process the example input, it would sweep and clean its internal state only once.

I want to bring attention to two points here: (1) Immediate action on a punctuation may not necessarily be the most efficient processing choice, and (2) operator scheduling may influence the choice of when to act on a punctuation. The example above works only if we know that any non-final punctuation is covered by a later punctuation, *e.g.,* `SUM` is operating on an contract with an input scheme of `[[gid:+,value:-]]`. I think affecting what an operator does when scheduled is a potential application of the Contracts Framework, in which these type of guarantees can unlock processing optimizations.

Another optimization opportunity that the Contracts Framework offers is finding a physical query plan for which a consistent accordance exists. Consider the following streams:

- `r(a,b)`, with punctuation scheme `[[a:-, b:+]]`
- `s(b,c,d)`, with punctuation scheme `[b:+, c:-, d:-]`
- `t(c,d)`, with punctuation scheme `[c:+,d:-]`

Now consider the query $r \bowtie s \bowtie t$. There are two ways to group the joins in order to construct a query plan that computes the result: $(r \bowtie s) \bowtie t$ and $r \bowtie (s \bowtie t)$.[2] Unlike scenarios in DBMSs, where the choice of physical join and known statistics would assist in choosing an optimal query plan, but either one is correct, in a DSMS one of these two equivalent query plans does not execute succesfully as it is unable to free its state. In the first option, the $r \bowtie s$ join cleanses state because the attribute named in the join condition is punctuated on both sides, and enables it to offer the punctuation scheme `[[a:-,b:+,c:-]]` in the output part of its contract. This punctuation scheme enables cleansing state in the $(r \bowtie s) \bowtie t$ plan (from the t input), and the contract of t cleanses state on the $(r \bowtie s)$ input. It so happens that the cleanse works because we have full coverage of all b and c values. The alternate plan, $r \bowtie (s \bowtie t)$, does not work. The $s \bowtie t$ join can only offer `[[b:+,c:+,d:-]]` or `[[b:-,c:-,d:-]]` in its output contract, neither of which enable cleansing state from the r side of the $r \bowtie (s \bowtie t)$. I suspect there are similar cases when a query parser needs to consider alternate physical query plans, where not all of them have a consistent accordance. Identifying a feasible reordering in a query plan can be yet another application of the Contracts Framework to query optimization.

A third application of the Contracts Framework that may be an interesting opportunity for future work is in providing guidance as to how input streams may be

[2]Notice that $(r \bowtie t) \bowtie s$ would not work since r and t have no attributes in common

punctuated. The treatment of the framework I presented in Chapter 5 was driven from the point of view of streams coming in with defined punctuation schemes, and having a menu of physical operators with various available contracts proven to be correct for each operator. Turn the problem around, and say we compute the consistent accordances leaving the input schemes open – thus providing an answer to the question: "which punctuation schemes are adequate for this physical query?" While in practice one is more likely to query data streams with known, fixed properties, it is not unreasonable to believe one may be designing the punctuation scheme of several streams in order to guarantee successful execution of plans of interest.

Incorporating information about punctuation periods to the Contracts Framework strikes me as an interesting fourth opportunity for future work, specifically when it comes to capacity planning. I think it is possible to estimate bounds on resource usage for a given operator, as a function of event type. This bound can be used to statically compute how many resources a given query would hold for a given punctuation period – say we punctuate on time every 5 seconds, and we know we see at most 1,000 tuples for that range. If we can annotate a punctuation scheme not only with the information about which attributes are punctuated, but also how often, it may be possible to establish resource utilization and latency bounds. This type of analysis could prove useful and help determine whether a continuous computation can be hosted with the existing resources. Potentially, one could even simulate changes and find inflection points when horizontal scale-out may need to be considered. I am optimistic that this type of analysis can be done statically instead of experimentally.

In Chapter 3 I already hinted at various uses of the feedback framework beyond avoiding work, which was the application I studied in this thesis. The two intentions for the downstream context I have hinted at are priorization and production of partial results. I think the power of expressing a subset of interest and describing the intended action on said subset is generic enough and may enable uses beyond the ones I described. However, the two areas I described deserve some study. Bhat [11] introduced the concept of *prodding* a continuous query to enable the production of partial results. His prodding occurs from the input side, but could theoretically be triggered from the output side and communicated as inter-operator feedback. The scenarios where prodding would prove interesting will probably come from time-sensitive speculation, such as a quick reaction to a market event that requires a best-estimate performance indicator to trigger a market alert. Interestingly, streaming technology excels in that domain, but going further down the latency chain to speculate *faster* than the processing of the query may be advantageous.

Priorization may find a niche use in distributed monitoring, where dynamic shifts in subsets of interest ma be discovered during execution. One scenario that comes to mind is focusing on freeway segments that are exhibiting behavior likely to lead to congestion, as this detection may enable a quicker reaction time to alter ramp metering (or dynamic speed limit adjustment) strategies and delay congestion formation as long as possible. This priorization would assume willingness to delay free-flowing segment analysis, but without avoiding processing those segments.

The work realized by recommendation engines, such as the collaborative-filtering work described by Amazon.com [43], tends to contain a continuous-event processing workload. These type of systems are now running in near-real time on top of distributed systems, which often requires coordination amongst nodes. Twitter's STORM[3] itself is already designed for distributed scale-out. One scenario that I can imagine useful in this type of systems is minimizing the amount of intermediate state processing from one node to the next when the runtime-analysis catches a new feature set. For example, imagine the task of harvesting all URLs containing the current set of top-ten twitter words. As soon as the top-ten set changes (which could be computed by another query), one would probably like to adjust the ongoing computation without restarting it or even restating it. I can imagine sending a form of demanded computation to trigger the updated harvest and a form of assumed computation that stops the harvesting that is no longer interesting. I think this idea of sending description and intent to an in-progress continuous computation can prove very advantageous in distributed systems with stringent latency requirements.

[3]`https://github.com/nathanmarz/storm/wiki`

In the specific case of assumed punctuation, I think there is still room for more work. One thing I observed while executing workloads for the experimental evaluation in Chapter 7 is that the amount of savings in a query are clearly linked to when the feedback is received and how much of the remaining work is avoided. I think it is also possible for feedback punctuations to have no visible effect in stream processing, since they may describe events that are not present in the stream. To me, this suggests there is room for optimization on when (and if) assumed punctuation is sent.

There seems to be a *range of effectiveness* for any given assumed punctuation. For a fixed number of avoidable events N in a stream, an assumed punctuation is maximally effective if it is received and exploited by an antecedent operator before work on those N events has started. An assumed punctuation is minimally effective if it is received and exploited after those N events have been processed. To me, this range suggests a series of interesting observations one could conduct on a feedback-aware system and a well-characterized workload. While I have presented how to implement such a system, and observed some of the runtime characteristics of these concepts put in practice, I have not conducted a study for a continuous workload in order to optimize feedback messaging. I think some of my observations, such as the effect of processing feedback as far upstream as possible, in combination with this notion of a range of effectiveness, can probably motivate follow-up work that characterizes strategies for determining an optimal feedback strategy for a given query and workload.

NiagaraST has a very malleable architecture that proved to be a great testbed for me to implement feedback. Other systems may very well make different design choices, for example, operators may be fused with no inter-operator queue in order to minimize inter-operator communication costs, or to enable operator block scheduling in different cores (similar to the techniques described in the Aurora system [9], or the operator fusion techniques in System S [25]). I think it would be interesting to validate some of my observations in stream systems with very different architectures. The premise of this line of work could generalize some of the observations I make for inter-operator communication to inter-processing unit communication optimization. It is possible that the feedback framework may also lead to interesting savings in distributed systems as well.

I hope the two main contributions of my dissertation, the concept of exploiting downstream context via inter-operator feedback, and the Contracts Framework, find a home in stream system designs. I remain most curious about applications of these techniques I have not foreseen (such as the feedback signals used by Chandramouli *et al.* when merging streams [16]), and hope to read about them in the near future.

REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*, pages 277–289, 2005.

[2] Daniel J. Abadi, Don Carney, Ugur 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 VLDB Journal: The International Journal on Very Large Databases*, 12(2):120–139, 2003.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. Technical Report 2004-20, Stanford InfoLab, 2004.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB*

Journal: The International Journal on Very Large Databases, 15(2):121–142, 2006.

[5] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pages 261–272, 2000.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pages 253–264, 2003.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proc. of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*, pages 1–16, 2002.

[8] Shivnath Babu and Jennifer Widom. StreaMon: An Adaptive Engine for Stream Query Processing. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*, pages 931–932, 2004.

[9] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective

on Aurora. *The VLDB Journal: The International Journal on Very Large Databases*, 13(4):370–383, 2004.

[10] Philip A. Bernstein and Dah-Ming W. Chiu. Using Semi-Joins to Solve Relational Queries. *J. ACM*, 28(1):25–40, 1981.

[11] Amit Bhat. Low-Latency Estimates for Window-Aggregate Queries over Data Streams. Master's thesis, Portland State University, 2011.

[12] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris N. Koutsopoulos, and Carlos Moran. IBM InfoSphere Streams for Scalable, Real-Time, Intelligent Transportation Services. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, pages 1093–1104. ACM, 2010.

[13] Barbara Carminati, Elena Ferrari, and Kian Lee Tan. Enforcing Access Control over Data Streams. In *Proc. of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT '07)*, pages 21–30, New York, NY, USA, 2007. ACM.

[14] 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 *Proc. of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 215–226, 2002.

[15] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator Scheduling in a Data Stream Manager. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB '03)*, pages 838–849, 2003.

[16] Badrish Chandramouli, David Maier, and Jonathan Goldstein. Physically Independent Stream Merging. In *Proc. of the 28th IEEE International Conference on Data Engineering (ICDE '12)*. IEEE, 2012.

[17] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the 1st Biennial Conference on Innovative Data Systems Research (CIDR '03)*, 2003.

[18] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pages 379–390, New York, NY, USA, 2000. ACM.

[19] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A Language for Analyzing Transactional Data Streams. *ACM Transactions on Programming Languages and Systems*, 26(2):301–338, 2004.

[20] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pages 647–651, 2003.

[21] Umeshwar Dayal, Barbara Blaustein, Alejandro Buchmann, Upen Chakravarthy, Meichun Hsu, R Ledin, Denis McCarthy, Arnon Rosenthal, Sunil Sarin, Michael John Carey, Miron Livny, and Rajiv Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17:51–70, March 1988.

[22] Rafael J. Fernández-Moctezuma, David Maier, and Kristin A. Tufte. Towards Execution Guarantees for Stream Queries. In *Proc. of the 3rd International Workshop on Scalable Stream Processing Systems (SSPS '10)*, 2010.

[23] Rafael J. Fernández-Moctezuma, James F. Terwilliger, Lois M. L. Delcambre, and David Maier. Toward Formal Semantics for Data and Schema Evolution in Data Stream Management Systems. In *Proc. of the 1st International Workshop on Evolving Theories of Conceptual Modeling (ETheCom '09)*, pages 85–94, 2009.

[24] Rafael J. Fernández-Moctezuma, Kristin Tufte, and Jin Li. Inter-Operator Feedback in Data Stream Management Systems via Punctuation. In *Proc.*

of the 4th Biennial Conference on Innovative Data Systems Research (CIDR '09), 2009.

[25] Buğra Gedik, Henrique Andrade, and Kun-Lung Wu. A Code Generation Approach to Optimizing High-Performance Distributed Data Stream Processing. In *Proc. of the 18th International Conference on Information and Knowledge Management (CIKM '09)*, pages 847–856. ACM, 2009.

[26] Lukasz Golab and M. Tamer Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2):5–14, June 2003.

[27] Jonathan Goldstein, Mingsheng Hong, Mohamed Ali, and Roger Barga. Consistency Sensitive Operators in CEDR. Technical Report MSR-TR-2007-158, Microsoft Research, 2007.

[28] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. Scheduling for Shared Window Joins over Data Streams. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB '03)*, pages 297–308, 2003.

[29] Moustaffa Hammad, Walid G. Aref, Michael J. Franklin, Mohamed Mokbel, and Ahmed K. Elmagarmid. Efficient Execution of Sliding Window Queries over Data Streams. Technical Report CSD TR 03-035, Purdue University, 2003.

[30] Jeong-Hyon Hwang, Sanghoon Cha, Uğur Çetintemel, and Stan Zdonik. Borealis-R: A Replication-Transparent Stream Processing System for Wide-Area Monitoring Applications. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 1303–1306, New York, NY, USA, 2008. ACM.

[31] IBM. InfoSphere Streams. `http://www-01.ibm.com/software/data/infosphere/streams/`, 2011. [Online; accessed 5-June-2011].

[32] Zachary G. Ives and Nicholas E. Taylor. Sideways Information Passing for Push-Style Query Processing. In *Proc. of the 24th IEEE International Conference on Data Engineering (ICDE '08)*, pages 774–783, 2008.

[33] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A Heartbeat Mechanism and its Application in Gigascope. In *Proc. of the 31st International Conference on Very Large Data Bases (VLDB '05)*, pages 1079–1088, 2005.

[34] Jürgen Krämer. *Continuous Queries over Data Streams - Semantics and Implementation*. PhD thesis, University of Marburg, 2007.

[35] Jürgen Krämer and Bernhard Seeger. Semantics and Implementation of Continuous Sliding Window Queries over Data Streams. *ACM Transactions on Database Systems*, 34:4:1–4:49, April 2009.

[36] Jayavel Shanmugasundaram Kristin, Kristin Tufte, David Dewitt, Jeffrey Naughton, and David Maier. Architecting a Network Query Engine for Producing Partial Results. In *Proc. of the 3rd International Workshop on the Web and Databases (WebDB '00)*, pages 17–20, 2000.

[37] Hua-Gang Li, Songting Chen, Junichi Tatemura, Divyakant Agrawal, K. Selçuk Candan, and Wang-Pin Hsiung. Safety Guarantee of Continuous Join Queries over Punctuated Data Streams. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB '06)*, pages 19–30, 2006.

[38] Jin Li. *Window Queries over Data Streams.* PhD thesis, Portland State University, 2008.

[39] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *ACM SIGMOD Record*, 34(1):39–44, March 2005.

[40] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pages 311–322, 2005.

[41] Jin Li, Kristin Tufte, David Maier, and Vassilis Papadimos. AdaptWID: An Adaptive, Memory-Efficient Window Aggregation Implementation. *IEEE Internet Computing*, 12(6):22–29, 2008.

[42] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-Order Processing: a New Architecture for High-Performance Stream Systems. *Proc. VLDB Endow.*, 1(1):274–288, 2008.

[43] Gregory D. Linden, Jennifer A. Jacobi, and Eric A. Benson. Collaborative Recommendations Using Item-to-Item Similarity Mappings. USPTO #6,266,649, 2001.

[44] David Maier. *The Theory of Relational Databases.* Computer Science Press, Inc., 1983.

[45] Adolf D. May. *Traffic Flow Fundamentals.* Prentice Hall, 1990.

[46] Microsoft Corp. Microsoft StreamInsight. `http://www.microsoft.com/sqlserver/en/us/solutions-technologies/mission-critical-operations/complex-event-processing.aspx`, 2011. [Online; accessed 15-February-2011].

[47] Jeffrey Naughton, David Dewitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, and Stratis Viglas. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24:27–33, 2001.

[48] Rimma V. Nemme, Karen E. Works, and Elke A. Rundensteiner. Query Mesh:

Multi-Route Query Processing Technology. In *Proc. of the 35th International Conference on Very Large Data Bases (VLDB '09)*, pages 1530–1533, 2009.

[49] Stan Zdonik Nesime Tatbul. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB '06)*, pages 799–810. VLDB Endowment, 2006.

[50] Matthias Nicola and Jasmi John. XML Parsing: A Threat to Database Performance. In *Proc. of the 12th International Conference on Information and Knowledge Management (CIKM '03)*, pages 175–178, 2003.

[51] Oracle Corporation. Oracle CEP. `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html`, 2011. [Online; accessed 2-May-2011].

[52] V. Raghavan, E. Rundensteiner, J. Woycheese, and A. Mukherji. FireStream: Sensor Stream Processing for Monitoring Fire Spread. In *Proc. of the 23rd IEEE International Conference on Data Engineering (ICDE '07)*, pages 1507–1508, 2007.

[53] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proc. of the 17th International Conference on Very Large Data Bases (VLDB '91)*, pages 469–478. VLDB Endowment, 1991.

[54] Utkarsh Srivastava and Jennifer Widom. Flexible Time Management in Data Stream Systems. In *Proc. of the 23st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '04)*, pages 263–274, New York, NY, USA, 2004. ACM.

[55] StreamBase Systems Inc. StreamBase. `http://streambase.com/`, 2011. [Online; accessed 15-February-2011].

[56] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB '03)*, pages 309–320, 2003.

[57] James F. Terwilliger, Rafael Fernández-Moctezuma, Lois M. L. Delcambre, and David Maier. Support for Schema Evolution in Data Stream Management Systems. *Journal of Universal Computer Science*, 16(20):3073–3101, 2010.

[58] Truviso Inc. TruViso. `http://www.truviso.com/`, 2010. [Online; accessed 15-February-2011].

[59] Yi-Cheng Tu, Song Liu, Sunil Prabhakar, and Bin Yao. Load Shedding in Stream Databases: A Control-Based Approach. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB '06)*, pages 787–798. VLDB Endowment, 2006.

[60] Yi-Cheng Tu and Sunil Prabhakar. Control-Based Load Shedding in Data Stream Management Systems. In *Proc. of the 22nd IEEE International Conference on Data Engineering Workshops (ICDEW '06)*, pages 144–148. IEEE Computer Society, 2006.

[61] P.A. Tucker, D. Maier, T. Sheard, and P. Stephens. Using Punctuation Schemes to Characterize Strategies for Querying over Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1227–1240, Sept. 2007.

[62] Peter A. Tucker. *Punctuated Data Streams*. PhD thesis, OGI School of Science & Engineering at OHSU, 2005.

[63] 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.

[64] Kristin Tufte, Jin Li, David Maier, Vassilis Papadimos, Robert L. Bertini, and James Rucker. Travel Time Estimation Using NiagaraST and Latte. In *Proc. of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*, pages 1091–1093, 2007.

[65] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proc. of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*, 1990.

[66] Karen Works and Elke A. Rundensteiner. The Proactive Promotion Engine. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB '03)*, pages 838–849, 2003.